DynHAC: Fully Dynamic Approximate Hierarchical Agglomerative Clustering

Anonymous

Abstract—**We consider the problem of maintaining a hierarchical agglomerative clustering (HAC) in the dynamic setting, when the input is subject to point insertions and deletions. We introduce DynHac – the first dynamic HAC algorithm for the popular *average-linkage* version of the problem which can maintain a $1+\epsilon$ approximate solution. Our approach leverages recent structural results on $1+\epsilon$-approximate HAC [1] to carefully identify the part of the clustering dendrogram that needs to be updated in order to produce a solution that is consistent with what a full recomputation from scratch would have output.**

We evaluate DynHAC on a number of real-world graphs. We show that DynHAC can handle each update up to 423x faster than what it would take to recompute the clustering from scratch. At the same time it achieves up to 0.21 higher NMI score than the state-of-the-art dynamic hierarchical clustering algorithms, which do not provably approximate HAC.

Index Terms—**clustering, dynamic algorithms**

I. Introduction

Clustering is an unsupervised machine learning method that has been widely used in many fields including computational biology, computer vision, and finance to discover structures in a data set [2]–[5] . To group similar objects at all resolutions, a *hierarchical clustering* can be used to produce a tree that represents clustering results at different scales. *Hierarchical agglomerative clustering* (HAC) is the most prominent hierarchical clustering algorithm [6]–[11] which is particularly well-suited to finding a large number of highly precise clusters [1], [12]–[17] . The algorithm takes as input a collection of n points and a function that gives the similarity of each pair of points. It starts by putting each point in its own singleton clusters and then proceeds in up to $n-1$ steps. In each step it finds the two most similar clusters, and *merges* them together, that is it replaces them by their union. The similarity between two clusters is defined by a *linkage function* which maps the all-pair point-to-point similarities between the points in both clusters to a single similarity value. One popular similarity function is *average–linkage*, where the similarity between the clusters is the average of all the individual similarities. That is, for two clusters C and D, their similarity is the sum of all-pairs similarities between points in C and D divided by $|C| \cdot |D|$.

The output of the algorithm is a *dendrogram*: a rooted binary tree which describes the merges of the clusters performed by the algorithm. Specifically, each node of the dendrogram represents a cluster. The leaves correspond to clusters of size 1 representing the input points. For each merge performed by the algorithm, the dendrogram contains an internal node representing a cluster obtained by merging its children.

Since specifying point-to-point similarities between all $n \times n$ pairs of input points is infeasible for large datasets, recent work on scaling up HAC considered the graph-based version of the problem [1], [17]–[20] . In graph-based HAC, the input is a (typically sparse) similarity graph, whose vertices represent points, and each edge specifies the similarity between its endpoints. If the input is a metric space, a natural approach is to build the k-nearest neighbor graph, that is a graph, where each point is connected to its k most similar other points. By using the graph-based input representation and allowing approximation, average linkage HAC has been successfully scaled to datasets of billions of points [1], [17].

Contemporary data exists in a constant state of flux, e.g, users interact with platforms in frequent intervals, or financial transactions occur very frequently. This has lead researchers to focus on maintaining solutions to problems like clustering in the *dynamic* setting where nodes and edges are being inserted and deleted and the objective is to adjust the solution efficiently to reflect the new state of the data. This is not an easy task, and studies often settle for algorithms that work in the (restricted) incremental only setting, i.e., where nodes or edges are inserted but are never deleted.

In this paper our goal is to maintain a HAC dendrogram in the *fully dynamic* setting, that is update it under a sequence of both insertions and deletions. Our main focus is to obtain highly precise clustering. Because of this, we use the average-linkage similarity function, which has been shown to deliver excellent empirical quality [1], [12]–[17] and strive to obtain rigorous theoretical quality guarantees on the output dendrogram.

Our main contribution is DynHAC – a dynamic HAC algorithm which maintains a $1+\epsilon$ approximate average-linkage dendrogram under point insertions and deletions. We use the notion of approximation introduced by Moseley et al. [1], [17], [18], [21] . Namely, a $1+\epsilon$ approximate HAC algorithm is allowed to merge two clusters whose similarity is at most a $1+\epsilon$ factor away from the similarity of the two most similar clusters.

The main challenge in developing an efficient dynamic HAC algorithm is the sensitivity of the output to even small changes in the input. In particular, one can easily design instances where inserting even a single node or edge causes the resulting dendrogram to change completely. Moreover, it was recently shown that even if one allows super-constant approximation, maintaining average-linkage HAC in a dynamic setting requires $n^{\Omega(1)}$ time per update in the worst case [22]. However, the very hard instances for dynamic HAC require a very particular structure, and so they do not preclude dynamic HAC from working efficiently on real-world instances.

Our algorithm builds on the ideas behind TeraHAC [1], a recently introduced distributed HAC algorithm. TeraHAC partitions the nodes into disjoint partitions and then independently runs an algorithm called SubgraphHAC within each partition. The goal of SubgraphHAC is to perform a certain number of $1+\epsilon$ approximate HAC steps within the partition. Crucially, in order to achieve $1+\epsilon$ approximation, SubgraphHAC uses a carefully chosen stopping condition to ensure that the merges performed within each partition are consistent with what a $1+\epsilon$ HAC algorithm would have performed if it was run on the entire graph. We note that this condition is based on the nodes in the partition and the set of its incident edges, that is edges that have at least one endpoint in the partition. Once all such intra-partition merges have been performed, the graph is partitioned again and the above step is repeated. We call each step of the above algorithm a *round*. On real-world datasets the graph typically shrinks by a constant factor in each round, and so the total number of rounds is small.

DynHAC uses a similar partitioned approach. We leverage the fact that if an edge is inserted within a partition P, no other partitions within the same level are affected. For the affected partition P, we simply run SubgraphHAC from scratch. However, with multiple levels, the problem becomes more intricate. A single change within the partition P may cause SubgraphHAC to perform a very different set of merges. This in turn may cause a nontrivial change to the graph handled in the next level. Moreover, in addition to having to propagate changes to the graph, we also need to dynamically maintain the partitions to ensure that their size is balanced and ensure that SubgraphHAC runs efficiently. Hence, on each level we need to carefully decide which partitions to recompute based on the changes coming from the previous level, and the updates to the partitioning that are taking place.

We experimentally evaluate DynHAC and compare it to the existing state of the art dynamic hierarchical clustering algorithms. Compared to existing static algorithms, we observe that DynHAC delivers up to a 423x speedup over running the algorithm from scratch after each update. On the other hand, compared to the existing dynamic algorithms, none of which provably approximate HAC, we observe that DynHAC achieves up to 0.21 higher NMI score than the best existing dynamic hierarchical clustering algorithms, showcasing the value of provable approximation guarantees in practice.

Our code, data, and a full version of our paper can be found at https://anonymous.4open.science/r/dynamic-hac-2F13.

A. Related Work

Several studies have considered the problem of maintaining a hierarchical clustering with average-linkage by re-arranging the hierarchy of clusters in the presence of node insertions [23]–[25]. These attempts can only process incremental updates. BIRCH [24], [26] works for Euclidean spaces; it maintains a tree structure with each node storing statistics on the nodes in its subtree. When a new point arrives, it tracks a root-to-leaf path based on some closeness criteria, and eventually inserts the new point as a leaf in the tree. PERCH [14] and GRINCH [27] work similarly to each other, where they first identify the leaf representing the nearest neighbor of the newly inserted node, and then track the path to the root of the hierarchy and apply appropriate rotations, as well as "graft" operations which are designed to discover chain-like clustering structures. OHAC [23] processes the insertion of a node by first deleting all nodes in the path from the the nearest neighbor of a newly arrived node to the root decomposing the dendrogram to a forest and then re-runs HAC on these roots of the forest.

The work that is most similar to ours is [25]. The authors consider multiple incremental clustering approaches both top-down and bottom-up. Their approaches include 1) modifications of the top-down Stable Greedy (SG) Trees approach by [28] that also allow re-evaluating the greedy choices in a second phase, and 2) dynamic versions of bottom-up approaches like the RecipNN method [29] which is an efficient implementation of HAC, and Affinity clustering [30]. The authors show that the bottom-up approaches perform the best in terms of tradeoffs between quality and running time. Concretely, the authors present methods that maintain a hierarchical clustering updated after each node insertion, such that at any point in time the hierarchy is the same as that produced by re-running the batch clustering counterpart (that is, HAC or Affinity) on the maintained graph at the time. We note that while their Online RecipNN algorithm from [25] is supposed to be a dynamic version of exact HAC, some simplifications in the implementation make it diverge from the exact algorithm. As a result, their experimental analysis concluded that the method often produces inferior quality results compared to other methods, and it is not very competitive in terms of running time. We show that, by maintaining a $(1+\epsilon)$-approximate HAC dendrogram, we are able to improve the performance of the algorithm by allowing some approximation and that our method consistently maintains a clustering of high quality. The speedup that we obtain due to leveraging approximation is similar to the observed efficiency gains obtained by using approximation in the case of static datasets [1], [17].

While the linkage function that has received the bulk of attention is the average-linkage function, the single-linkage – the similarity between the clusters is the maximum of the-point-to-point similarities – has also been studied in the dynamic setting. In fact, the single-linkage function is closely related to the Minimum Spanning tree problem (see e.g., [22]), whose dynamic version has been extensively studied in the literature [31]–[33].

Another line of work considered optimization objectives for hierarchical clustering [34], [35] . Some of the studied objectives include the CKMM Revenue [36] , Dasgupta cost [37] , and the MW Revenue [38] . That line of work is more theory-focused. In particular the best algorithms w.r.t. Dasgupta objective give $O(\sqrt{\log n})$ approximation [36], [39], while conditional lower-bounds exclude the existence of constant factor approximate solutions [39]. In the case of Moseley-Wang objective, a random clustering (clearly very weak from a practical standpoint) has been shown to give a $1/3$ approximation, while the best approximation algorithm obtains a 0.585 approximation [40]. For the CKMM objective the best known algorithm achieves a 0.74 approximation guarantee [41]. Interestingly, HAC with average-linkage has been shown to give a $1/3$ approximation for the MW Revenue objective, and $2/3$ for the CKMM objective, and these bounds are tight [34].

Finally, [42] uses a hyperplane partitioning method to construct a hierarchical clustering over a stream of updates in a top-down manner, and has the nice property that is agnostic to the order of the updates. The method is only applicable to Euclidean spaces.

II. Preliminaries

Let $G=(V,E,w)$ be a weighted and undirected graph, where $|V|=n$ and w is the edge-weight function. We assume that all edge weights of G are positive. We use $V(G)$ and $E(G)$ to denote the vertex and edge sets of G, respectively.

In order to define $(1+\epsilon)$-approximate HAC we describe a sequential static $(1+\epsilon)$ approximate HAC algorithm, which we refer to as SeqHAC. The SeqHAC algorithm maintains a graph, in which each vertex is a cluster. Initially, we start with clusters of size 1, and so the state of the algorithm is represented by the input graph G. In addition, it also maintains the *size* of each vertex of G. The size of each vertex is 1 in the beginning. We define the *normalized* weight of an edge xy as $\bar{w}(xy)=w(xy)/(S(x)\cdot S(y))$, where $S(x)$ and $S(y)$ are the sizes of x and y respectively.

The SeqHAC algorithm proceeds as follows. While G contains at least one edge, we pick an edge xy whose normalized weight satisfies $\bar{w}(xy) \geq \bar{w}(uv)/(1+\epsilon)$, where uv is the edge that thas the highest normalized weight in G. Then, the algorithm *contracts* the edge xy. In the following we also say that it performs a *merge* of x and y of linkage similarity $\bar{w}(xy)$.

Contraction of xy merges x and y into one vertex z, whose size is set to $S(z)=S(x)+S(y)$. The parallel edges that are created are merged into one, and the corresponding edge weights are summed. Finally, any self-loops are removed. It is easy to see that when $\epsilon=0$ this algorithm is equivalent to the one that is sketched in Section I.

Observe that SeqHAC can produce multiple valid outputs, given that it can contract any edge of sufficiently high weight in each step. We say that any valid output of SeqHAC is a $(1+\epsilon)$-approximate dendrogram. Moreover, any algorithm which always produces a $(1+\epsilon)$-approximate dendrogram is called $(1+\epsilon)$-approximate HAC.

TeraHAC Algorithm TeraHAC [1] is a *parallel* $(1+\epsilon)$ approximate HAC algorithm. The DynHAC algorithm that we introduce in this paper is essentially a dynamic version of TeraHAC, and so we now briefly outline how TeraHAC works.

The TeraHAC algorithm proceeds in multiple rounds. In each round it computes a partition $\mathcal{P}$ of the vertices of G, i.e., $\bigcup_{p \in \mathcal{P}} p = V(G)$, and contracts some edges within each partition.

Definition 1 (Partition subgraph). *Let $\mathcal{P}$ be a partition of V. For each $p \in \mathcal{P}$ we define the* partition subgraph *of p, denoted by H_p, as follows. The vertex set of H_p is the set of vertices p and all their neighbors in G. The edge set of H_p is the set of all edges incident to a vertex in p. For each H_p, we say that the vertices of p are* active*, and the remaining ones are* inactive.

Observe that any edge xy of G is in at most 2 subgraphs H_p. Moreover, the number of vertices in $V(H_p) \setminus p$ is at most the number of edges in H_p. As a result, all subgraphs H_p (for $p \in \mathcal{P}$) have $O(|E(G)|)$ vertices and edges in total.

TeraHAC then runs a restricted HAC algorithm on each partition subgraph. We call this algorithm SubgraphHAC. The SubgraphHAC algorithm, given a partition subgraph H_p, merges some pairs of vertices in H_p following two important constraints. First, it only merges together active vertices of H_p, which ensures that all the SubgraphHAC calls within a round end up merging disjoint sets of vertices. We note that the vertex obtained by merging two active vertices is considered active as well. Second, SubgraphHAC ensures that the merges it makes are provably consistent with what a $(1+\epsilon)$ approximate HAC algorithm would have done if it was working on the entire graph. To this end, SubgraphHAC leverages the following notion in order to decide whether a certain pair of vertices can be merged.

Definition 2 (Good merge [1]). *Let $\epsilon \geq 0$ and G_i be a graph obtained from G by performing some sequence of merges. For each vertex v of G_i, we define $\mathrm{M}(v)$ to be the smallest linkage similarity among all merges that were performed to create vertex (cluster) v. Specifically, for each vertex v of size 1 we have $\mathrm{M}(v) = \infty$, and whenever two vertices u and v are merged and create a vertex z, we have $\mathrm{M}(z) = \min(\mathrm{M}(u), \mathrm{M}(v), \bar{w}(uv))$. Moreover, for each vertex v we use $w_{\max}(v)$ to denote the highest normalized weight of any edge incident to v. With this notation, we say that a merge of an edge uv in G_i is $(1+\epsilon)-$*good *if and only if*

$$\frac{\max(w_{\max}(u), w_{\max}(v))}{\min(\mathrm{M}(u), \mathrm{M}(v), \bar{w}(uv))} \leq 1+\epsilon.$$

SubgraphHAC ensures that each merge that it produces is $(1+\epsilon)$ good and leverages the following key property.

Lemma 1 ([1]). *Any dendrogram produced by a sequence of $(1+\epsilon)$-good merges is $(1+\epsilon)$ approximate.*

The DynHAC algorithm uses SubgraphHAC in a black-box way. We note that in addition to the current graph SubgraphHAC must also be provided the minimum linkage similarities $M(\cdot)$ of any vertex.

TABLE I: Symbols and their meanings

Symbol	Meaning
V_i	vertices to insert to G_i in round i.
V_i^d	vertices to delete from G_i in round i.
$N_G(V)$	The union of the set of neighbors of $v \in V$.
P_i	Partition of vertices in G_i.
$\texttt{VMap}_i$	Maps $V(G_i)$ to the vertices they contracted to in G_{i+1}.
M	Stores the smallest linkage similarities among all merges that were performed to create clusters.

III. DynHAC Algorithm

In this section we describe the DynHAC algorithm, which is a dynamic version of TeraHAC. Following TeraHAC, DynHAC proceeds in rounds. In each round it partitions the graph, runs SubgraphHAC on each partition subgraph and then obtains the input to the next round by applying all the merges from all SubgraphHAC calls. We denote by G_i the graph which is the input to round i. Here, G_1 is the input graph on which we run HAC.

The main principle behind DynHAC is as follows. Whenever a partition subgraph changes, we rerun SubgraphHAC on this subgraph. Since the only input to SubgraphHAC is a partition subgraph (together with the corresponding minimum merge similarities), we do not have to rerun SubgraphHAC on the partition subgraphs that have not changed. We note, however, that the partition subgraph of a partition p can contain an edge xy, such that $y \notin p$. Since the normalized weight of the edge xy depends on the size of y, the partition subgraph H_p can change as a result of a change to a vertex outside of p.

Partitioning. Because of the dynamic nature of DynHAC, the partitioning used in each round needs to be updated dynamically. Moreover, as we rerun SubgraphHAC from scratch upon a change to a partition subgraph, we would like the partitions to be relatively small.

We use the following simple partitioning scheme. We first color the vertices by randomly assigning the colors red and blue to each vertex with equal probability. Then, each partition consists of the vertices that have the same partition id, according to the following definition.

Definition 3 (Partition id)**.** *We define the* partition id *of each vertex v as follows. If v is either red or does not have a blue neighbor, the partition id is v. Otherwise, i.e., when v is blue and has a red neighbor, the partition id of v is the id of its highest normalized weight red neighbor.*

We note that the choice of the partitioning algorithm only affects the running time of the entire clustering algorithm (and not its correctness).

Data to dynamically maintain. The DynHAC algorithm dynamically maintains the following:

- For each round i:
  - G_i: the input graph to this round,
  - P_i: partition of the vertices in G_i,
  - $\texttt{VMap}_i$: a map from vertices in G_i to the corresponding vertices in G_{i+1}. If a vertex is not contracted in round i, it maps to itself.
- M: minimum merge similarity of each vertex in each G_i (Definition 2),
- $\mathcal{D}$: the $(1+\epsilon)$-approximate dendrogram.

Handling an update. The algorithm for handling an update is given as Algorithm 1. It takes as input the vertices and edges to insert V, E, the vertices to delete V^d, existing dendrogram $\mathcal{D}$, and a weight threshold $t > 0$. The weight threshold t is used to terminate the algorithm once it performs all merges of sufficiently high similarity. For simplicity, we assume that the value of t is the same across all updates, i.e., we maintain the dendrogram up to some linkage similarity. The effect of this parameter will be discussed later.

The process of handling an update starts by adding leaf nodes to the dendrogram and assigning ∞ to $\mathrm{M}(v)$ for the newly inserted nodes (Lines 3–4). Then, the algorithm updates each round one by one (Line 6). In each round it updates the partitions of the graph, and runs SubgraphHAC on each affected partition subgraph. Thus, if the update in a round is not incident to a partition p, the merges in p are all still $(1+\epsilon)$-good and p does not need to be re-clustered in this round.

Updating a single round. The process of updating a single round is shown as Algorithm 2. For each round, we 1) update G_i, $\texttt{VMap}_i$, P_i, M and $\mathcal{D}$ according to the inserted/deleted vertices and edges, and 2) compute the new vertices and edges (V_{i+1}, E_{i+1}) to add to and vertices (V^d_{i+1}) to delete from the next round to satisfy the invariant. The computation only depends on the state of the previous round.

Algorithm 1 DynHAC-ϵ

```
1:  Input: $V, E, V^d$, $t \geq 0$
2:  Update: $\mathcal{D}$, M, $\{G_i, P_i, \texttt{VMap}_i\}$
3:  $\mathcal{D}$.AddLeaves($V$) ▷ Add leaf nodes to dendrogram
4:  $\mathrm{M}(v) = \infty$ for $v \in V$    ▷ Initialize min merge similarities
5:  $V_1, E_1, V_1^d = V, E, V^d$
6:  for $i \in \{1 \ldots \infty\}$ do
7:      $V_{i+1}, E_{i+1}, V_{i+1}^d$ = DynamicHacRound($V_i$, $E_i$, $V_i^d$)
8:      if $G_i$ has no edge with weight $> t/(1+\epsilon)$ then
9:          Empty rounds $\{i+1, \ldots, \infty\}$
10:         Remove the ancestors of $V(G_i) \cup V^d$ in $\mathcal{D}$
11:         $\mathcal{D}$.RemoveLeaves($V^d$)
12:         break;
```

Algorithm 2 DynHAC Round

```
1:  Input: $V_i$, $E_i$, $V_i^d$
2:  Output: $V_{i+1}, E_{i+1}$, $V_{i+1}^d$
3:  Update: $G_i$, $P_i$, M, $\texttt{VMap}_i$, $\mathcal{D}$
4:  $\Delta_P \leftarrow$ UpdatePartition($G_i$, $V_i$, $E_i$, $V_i^d$, $P_i$)
5:  $G_i \leftarrow G_i \cup (V_i, E_i) \setminus V_i^d$        ▷ Update graph
6:  if $V_{i+1}$ is empty then        ▷ Reached last round
7:      $P_{\text{dirty}} \leftarrow$ all partitions
8:  else
9:      $P_{\text{dirty}} \leftarrow$ DirtyPartitions($\Delta_P$, $G_i$)
10: $V_{i+1}, E_{i+1}, V_{i+1}^d = \{\}$
11: for $p \in P_{\text{dirty}}$ do
12:     $H_p \leftarrow$ Subgraph($G_i$, $p$)
13:     $\texttt{merges}, \mathcal{D}_p, H_p^c \leftarrow$ SubgraphHAC($H_p$, M, $\epsilon$)
14:     UpdateDendrogram($\texttt{merges}$, $\mathcal{D}$)
15:     UpdateMinMergeSim($\texttt{merges}$, M)
16:     $V_{i+1}^d \leftarrow V_{i+1}^d \cup$ UpdateVMap($\mathcal{D}_p$, $p$, $V_i^d$, $\texttt{VMap}_i$)
17:     $H_p^c \leftarrow$ contract inactive vertices in $H_p^c$ based on $\texttt{VMap}_i$.
18:     $V_{i+1} \leftarrow V_{i+1} \cup V_{\text{active}}(H_p^c) \setminus V(G_{i+1})$
19:     $E_{i+1} \leftarrow E_{i+1} \cup E(H_p^c)$
20: Return $(V_{i+1}, E_{i+1})$, $V_{i+1}^d$
```

When a graph G_i has no edge of weight $> t/(1+\epsilon)$ (Line 8), we do not need to make more merges, so we cleanup and stop. To cleanup, we empty all future rounds and remove the dendrogram ancestors of deleted nodes V^d and vertices in the last graph $V(G_i)$. We also remove the deleted leaf nodes in the dendrogram.

We now describe the order of updating the state, and then explain the details of how each object is updated.

1) We first update the partitioning P_i, and obtain the vertices that changed partition ids due to insertions and deletions, together with their partition ids before and after the update, Δ_P (Line 4).
2) In Line 5 we update the graph in this round. When a vertex is removed, all its neighboring edges are removed as well.
3) In Lines 6–9, we compute the partitions that we rerun SubgraphHAC on, which we call the *dirty partitions* (P_{dirty}). If the graph in the next round contains no vertex, then we've reached the last round, but clustering has not finished, so all partitions are dirty and we need to cluster all of them. Otherwise, we find the dirty partitions based on Δ_P.
4) For each dirty partition p, we consider the partition subgraph H_p (see Definition 1). We run the $(1+\epsilon)$ SubgraphHAC algorithm [1], which merges some pairs of active vertices. (Lines 12–13). As a result of running SubgraphHAC, we obtain a merge sequence `merges`, the resulting dendrogram $\mathcal{D}_p$, and H_p^{c}, which is H_p after applying the merges performed by SubgraphHAC.
5) In Lines 14–15, we update the overall dendrogram $\mathcal{D}$ and the minimum similarities M based on the obtained new merges.
6) In Line 16, we update $\mathtt{VMap}_i$ and compute the vertices to delete from the next round.
7) Finally, in Lines 17–19, we obtain new vertices to insert to the next round from the contracted graph. Note that some vertices may already exist in the next round (we made the same merges as before), these vertices are excluded. The new edges to insert are also obtained from the contracted graph. Intuitively, these are the vertices and edges to insert because G_{i+1} is just the graph G_i contracted according to merges in round i.

A. Dynamic update of dendrogram and auxiliary data.

Update partitioning P_i. We observe that in order to update the partition ids to restore the property of Definition 3 *we just need to re-compute the partition ids of the new vertices, the neighbors of new vertices, and the neighbors of deleted vertices* (see Lemma 2). After we compute their new partition ids, we return these vertices $V_{\text{new and neighbors}}$ and their partition ids before and after the update. Note that this is a superset of vertices that actually changed partition ids, since some of the partition ids might not change. We still return them because they are useful when computing the dirty partitions.

To compute the new partition id of a blue vertex, we just need to scan the neighbors of the vertex. For the neighbors i of new vertices j, we can optimize this step to simply check if the new edge (i, j) has a higher normalized weight than the that of the edge between i and its previous heaviest red neighbor.

Lemma 2. *Consider a graph G and a graph $G' = G \cup (V, E) \setminus V^d$, which is obtained from G by adding a set of vertices V and edges E, and deleting vertices of V^d. Assume that each edge of E is incident to a vertex in V and not incident to any vertex in V^d. Let $U \subseteq V(G')$ be the set of vertices of G' which have different partition ids in G and G'. Then, $U \subseteq V \cup N_{G'}(V) \cup (N_G(V^d) \setminus V^d)$.*

Identify dirty partitions. In each round of the update our algorithm identifies a set of *dirty partitions*, that is a set of partitions for which it reruns SubgraphHAC from scratch. At a high level, we would like to identify all partitions in which, after the update, some merge performed in the current dendrogram is no longer $(1+\epsilon)$-good. We call these partitions *truly dirty*. To demonstrate correctness of our algorithm, we show that the set of dirty partitions identified by our algorithm is a superset of the set of truly dirty partitions.

Efficiently identifying the dirty partitions is challenging because there are multiple cases to consider. In particular, a partition is considered dirty when:

- a vertex is added to/deleted from the partition, and this can be caused by a vertex addition or deletion, or by a partition id change,
- the set of edges leaving a partition changes,
- a new partition is introduced.

Based on the above, we define Δ_P as follows.

Definition 4. *Let Δ_P contain the partition id (see Definition 3) before and after partition update of the following vertices: the new vertices, the deleted vertices, the neighbors of new vertices, and the neighbors of deleted vertices.*

Our simple algorithm for identifying dirty partitions based on Δ_P is shown as Algorithm 3.

Given $V_{\text{new and neighbors}}$ and their partition ids before and after the update, all new partitions are marked as dirty. For old partitions, we mark them as dirty if it is still in the new graph (not deleted) and it is not the same as a blue vertex. Note that each partition id is also a vertex id.

Update dendrogram. In Algorithm 4, we show how to update the dendrogram given the merges performed by SubgraphHAC in one partition subgraph. The input

Algorithm 3 DirtyPartitions

```
Input: $\Delta_P : \{v : p \to p' | v \in V_{\text{new and neighbors}}\}$, $G$
Output: $DP$ set of dirty partitions
$DP = \{\}$
for $\{v : p \to p'\} \in \Delta_P$ do    ▷ $p$ might be $\emptyset$ indicating the vertex did not exist, and $\emptyset \notin G$.
    $DP$.Insert($\{p'\}$)
    $DP$.Insert($\{p\}$) if ($p \in G$ and $p$ is not blue)
```

`merges` should be a sequence of merges that are consistent with $\mathcal{D}_p$ – the dendrogram of merges produced for that subgraph. For each merge (u, v) with parent node id a, we check if the merge (u, v) also exists in $\mathcal{D}$. If it is in the dendrogram, we do not need to update anything. If it is not in the dendrogram, we remove the ancestors of u and v, and merge them to have parent a.

It's easy to see that after each round, the dendrogram $\mathcal{D}$ contains all merges in the previous rounds.

Algorithm 4 UpdateDendrogram

```
Input: merges, $\mathcal{D}$
Output: update $\mathcal{D}$
for $((u, v) \to a) \in$ merges do    ▷ $u, v$ merge to form $a$
    if $(u, v) \notin \mathcal{D}$ then
        $\mathcal{D}$.DeleteAncestors($\{u, v\}$)
        $\mathcal{D}$.Merge($\{u, v\}$, $a$) ▷ merge $u, v$ to parent $a$
```

Updating the minimum merge similarities. We compute the new minimum merge similarities of vertices created by `merges` and store them in M. For space efficiency, we can also remove the deleted vertices V_i^d and the deleted ancestors in $\mathcal{D}$ in the previous step from M, but this is not required for correctness.

Update vertex mapping `VMap`$_i$ and compute vertices to delete from next round. To update `VMap`$_i$, we only need to update the mappings for p and V_i^d, where p is the set of the active vertices in H_p (not contracted). This is because only the mappings of these vertices can possibly change.

For each deleted vertex v, its mapped vertex `VMap`(v) should be deleted in the next round. We can also remove v from `VMap`. One exception is if `VMap`(v) is a contracted vertex in G_p, then it should not be deleted. This can happen in the following example case. u, v merges to w in round i. Originally vertex u maps to w, but now vertex w is in round i due to updates, but it maps to itself w in the next round because it does not merge in this round.

For each vertex v in the partition, let its root node in $\mathcal{D}_p$ be r. r is also the node that v is contracted to after all merges in this round. So v should be mapped to r in the next round. If its current mapping is not r, we update it to map to r. In addition, we should remove its current mapping from the next round.

We present the pseudocode in the Appendix (Algorithm 5).

IV. Analysis of DynHAC

Due to space constraints, we provide full proofs in the appendix and briefly sketch the arguments here. The next lemma shows that all merges within a partition that is *not* marked as dirty are still good.

Lemma 3. *If a partition P exists and does not become dirty upon node update, all $(1+\epsilon)$-good merges within the partition are still $(1+\epsilon)$-good.*

The proof works by case analysis of Definition 2 , and can be found in Appendix A . Using Lemma 3, we can show that all merges made by the DynHAC algorithm are $(1+\epsilon)$-good, which in turn implies that the algorithm computes a $(1+\epsilon)$-approximate dendrogram, yielding the next theorem.

Theorem 1. DynHAC *maintains a $(1+\epsilon)$-approximate dendrogram upon node insertions and deletions.*

Finally, the following two theorems bound (1) the amount of work that our algorithms perform during a single round, and (2) to initialize the data structure given an initial input.

Theorem 2. *The total size of dirty partitions in a round can be bounded by the size of the 4-hop neighborhood of all inserted and deleted nodes.*

Theorem 3. *Inserting n nodes with m edges into an empty graph using Algorithm 1 takes $O(R(m+n)\log^2 n)$, where R is the number of rounds. The space complexity is $O(Rm)$.*

V. Empirical Evaluation

We run all experiments on a `c2-standard-60` Google Cloud instance, which consists of 30 cores (with two-way hyper-threading), with 3.1GHz Intel Xeon Scalable processors and 240GB of main memory. We run all experiments sequentially, i.e., using a single core.

Datasets. We list information about the data used in our experiments in Table II. ***MNIST*** is a collection of 28x28 grayscale images of handwritten digits (0-9). We embed it to 2 dimensions using UMAP [43]

TABLE II: Datasets, including the number of data points (n), the dimension (d), and the number of ground truth clusters.

Graph Dataset	Num. Points	Dim.	Num. Clusters
MNIST	70,000	2	10
ALOI	108,000	128	1000
ILSVRC	50,000	2048	1000

before clustering. ***ALOI*** [44] is a collection of 1,000 objects recorded under various imaging circumstances. ***ILSVRC_SMALL*** (ILSVRC) [45] is a 50K subset of the Imagenet ILSVRC12 dataset. The data is embedded using Inception [46]. The same data set is used in [27].

Algorithms. We compare DynHAC with a static approximate graph HAC and a dynamic hierarchical clustering baseline.

- **DynHAC**: Our dynamic approximate graph HAC algorithm implemented in C++ that supports both insertion and deletion. We use $\mathsf{DynHAC}_{\epsilon=x}$ to denote DynHAC run with parameters $\epsilon = x$. When not specified, DynHAC denotes running the algorithm with $\epsilon = 0.1$. We use threshold $t = 0.0001$ for ***MNIST*** and $t = 0.01$ for ***ALOI*** and ***ILSVRC_SMALL***.
- ***Static HAC*** [17]: A static approximate graph HAC algorithm implemented in C++. We run it with $\epsilon = 0.1$ and the same thresholds as DynHAC. Since this is a static algorithm, it re-computes the clustering from scratch upon each update.
- ***GRINCH*** [27], [47], [48]. A dynamic algorithm working directly on points. It is implemented in Python. GRINCH inserts and deletes one point at a time. A point can be deleted from a hierarchy by simply removing the corresponding leaf node.
- ***GraphGrove*** (Grove) [25], [49]. Monath et al. [25]'s OnlineSCC algorithm on vector data. We add a uniformly random noise between 10^{-6} and -10^{-6} to each coordinate because it does not support duplicate points. We use a maximum of 50 levels with a geometric progression of thresholds from 1 to 10^{-8}. It supports insertion but not deletion.

Experiment Setup. We randomly permute the points to get an ordering $[x_1 \dots x_i \dots x_n]$. For all algorithms, we insert them in increasing order of indices i, and delete in decreasing order of i. We insert/delete one point at time.

For our insertion experiments with DynHAC, we insert a new node into the graph with edges to 50 approximately nearest neighbors of the inserted point using the Vamana [50] algorithm from ParlayANN [51]. When finding the approximate nearest neighbors, we only consider the points that are already inserted. We first batch insert 99% of the points, and then insert one point at a time. We also run an experiment on MNIST where we first insert 1,000 points (1.42% of all points), and then insert the rest of the points one at a time.

For our deletion experiments with DynHAC, we first batch insert all points, and then remove points one at a time. To batch insert all points, we construct the graph by splitting the points into 100 batches $B_1, \dots, B_{100}$ and for all points in batch B_i we add edges to 50 approximate nearest neighbors in batches $B_1, \dots, B_i$. We use this approach instead of finding the nearest neighbors considering all points to prevent each point from loosing too many neighbors during the deletion sequence, i.e. we ensure that each point has many neighbors among points that are deleted after it.

For Static HAC, we construct the graph using the same method with 100 batches, and run static HAC on the graph.

For GRINCH insertion and deletion, we insert one point at a time, and then delete one point at a time after all points are inserted. GRINCH implementation does not support batch insertion or deletion. For Grove insertion, we batch insert 99% of the points and then insert one point at a time. We also run an experiment on MNIST where we first insert 1,000 points (1.42% of all points), and then insert the rest of the points one at a time. We note that Grove does not support deletions.

Evaluation. We evaluate the clustering quality using the Normalized Mutual Index (NMI). NMI is MI normalized by the arithmetic mean of the entropy of the two clusterings. The NMI score is 1 for a perfect correlation, and 0 for no mutual information.

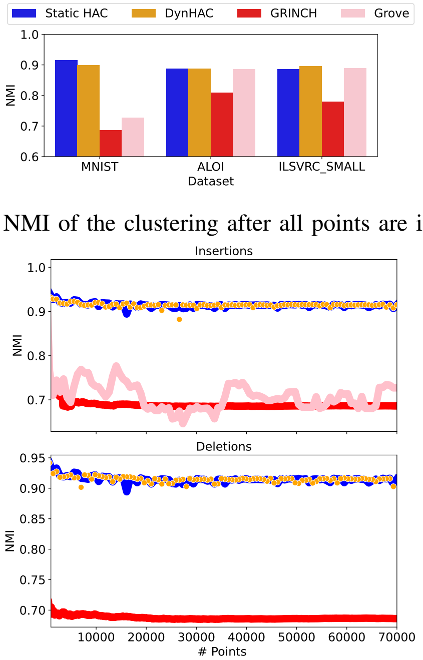
For all algorithms except Grove, a flat clustering is extracted from the hierarchical clustering by cutting the dendrogram at a particular cutting threshold. We try cutting at 40 log-spaced fixed thresholds between 10^{-4} and 0 to find the best NMI with different cutting thresholds. For Grove, we look at the clustering of all levels, and choose the one with highest NMI.

A. Comparing with Baselines

In Figure 1 and Figure 2, we show the NMI and clustering time of the algorithms on the three data sets.

Quality. We show that DynHAC maintains a high quality dendrogram. Figure 1a shows the NMI of the clustering after all points are inserted. Figure 1b shows the NMI of the algorithms after each update on MNIST.

We see that DynHAC can get NMI very close to that of static HAC, which aligns with our theoretical approximation guarantee on the dendrogram quality, and matches prior experimental studies of static approximate



(a) The NMI of the clustering after all points are inserted.

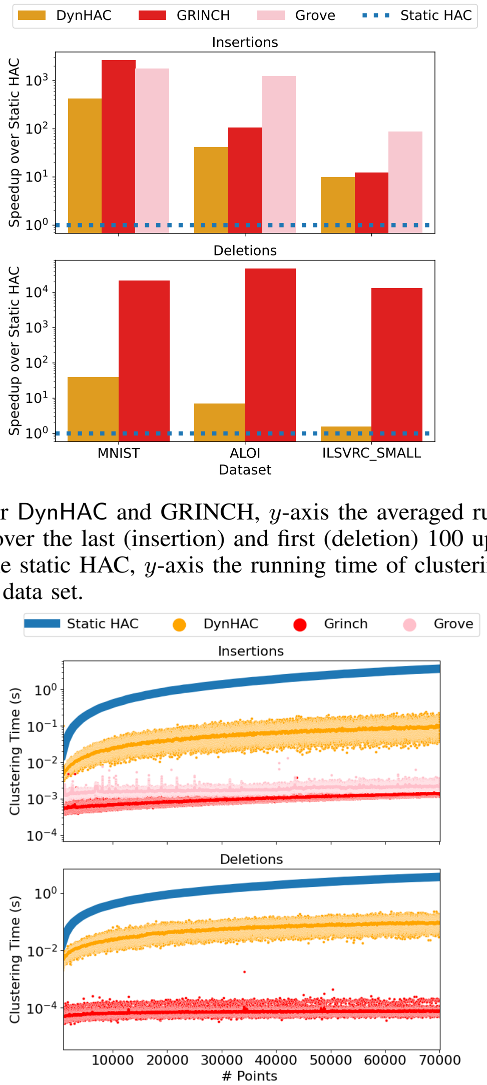
(b) The NMI after each update on MNIST. x-axis is the number of points in the data set after the update.

Fig. 1: Quality of clustering algorithm.

HAC [1], [17]. On ALOI, in the final dendrogram with all points inserted, the NMI obtained by DynHAC is only 0.0014 lower than the one of static HAC. For ILSVRC, DynHAC achieves even slightly higher NMI than the static HAC. Over all insertions and deletions on MNIST, DynHAC's NMI is at most 0.03 and 0.015 lower than the static HAC, respectively.

Comparing to DynHAC, GRINCH achieves a significantly lower NMI score. Compared to GRINCH, DynHAC's NMI is 0.21 higher on MNIST, 0.08 higher on ALOI, and 0.12 higher on ILSVRC. Grove achieves good NMI on ALOI and ILSVRC, but much lower NMI on MNIST. Compared to Grove, DynHAC's NMI 0.18 higher on MNIST, 0.001 higher on ALOI, and 0.007 higher on ILSVRC.

We conclude that DynHAC is the only method that we study that can consistently achieve quality close to that of the static HAC baseline.



(a) For DynHAC and GRINCH, y-axis the averaged running time over the last (insertion) and first (deletion) 100 updates. For the static HAC, y-axis the running time of clustering the entire data set.

(b) x-axis is the number of points in the data set after the update. The time for the static HAC is the running time of clustering the graph from scratch, and the time for dynamic algorithms is the time of making a single update. The solid line is a running window average of the running time with window size of 100.

Fig. 2: Running times.

Running time. In Figure 2(a) we show the running time of all algorithms on the three data sets. Our algorithm is slower than GRINCH, which is expected if we compare the running time bounds. Specifically, GRINCH's running time is $O(Tn + H^2)$ per data point, where T is the time spent on nearest neighbor search, and H is the height of resulting dendrogram [27]. On the other hand, in the worst case our algorithm has $\Theta((m+n)R\log^2 n)$ running time, where R is the total number of rounds, i.e., in the case of an update resulting in all clustering merges to change. However, in practice we often do not need to update all merges, so our running time is still faster than the static HAC. Though our algorithm is slower than GRINCH and Grove, our clustering quality is higher as

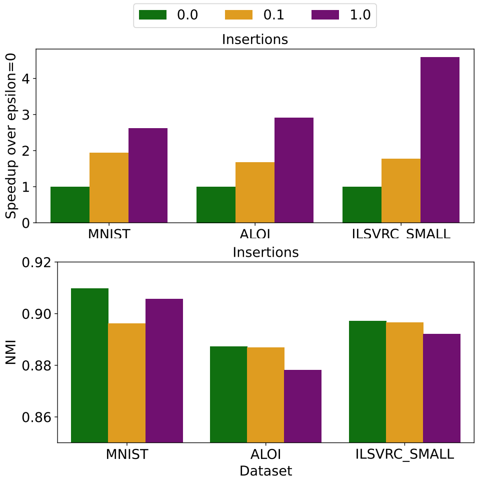


Fig. 3: Update speedup over $\epsilon = 0$ and NMI of the last 1% insertions on data sets with different ϵ values. Deletions are similar.

shown earlier in the section.

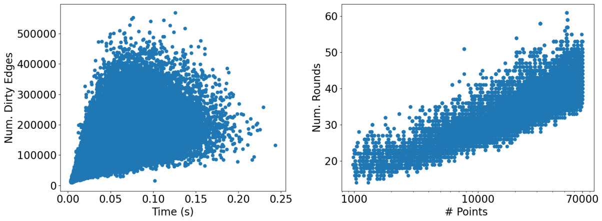
On MNIST, ALOI, and ILSVRC, we are 423x, 41x, and 10.0x faster than the Static HAC for insertion, respectively. Deletion is 39.7x, 6.9x, and 1.56x faster on MNIST, ALOI, and ILSVRC. On ILSVRC, the total size of the dirty partitions is large, and so we have a smaller speedup. In the Appendix, we also show the running time for all insertions and deletion on the last 1% of all three data sets.

B. Performance Analysis

Varying ϵ In Figure 3, we show the average running time and NMI when using different ϵ values ($[0, 0.1, 1]$). Note that $\epsilon = 0$ is the same as exact HAC. Here we show the result for insertions, but the result for deletion is similar (see Appendix). We see that with a higher ϵ, we get faster running time, but only a little degradation in clustering quality. Specifically, in insertion $\mathsf{DynHAC}_{\epsilon=0.1}$ is up to 1.93x and $\mathsf{DynHAC}_{\epsilon=1}$ is up to 2.6x times faster than $\mathsf{DynHAC}_{\epsilon=0}$. In deletion $\mathsf{DynHAC}_{\epsilon=0.1}$ is up to 2.6x and $\mathsf{DynHAC}_{\epsilon=1}$ is up to 4.22x times faster than $\mathsf{DynHAC}_{\epsilon=0}$.

Number of edges in dirty partition. In Figure 4a, we plot the number of edges in dirty partitions across all rounds against the update time. We see that the update time increases with the number of dirty edges, which aligns with our analysis that the bottle neck of the algorithm is $\mathsf{SubgraphHAC}$, whose running time is $O((m + n)\log^2 n)$ [1].

Number of rounds. In Fig. 4b, we plot the number of points inserted already and the number of rounds taken for an insertion update. We see that the number of rounds increases logarithmically with the of the number of points.



(a) Total edges in all dirty partitions vs. update time.

(b) The number of rounds vs. number of points inserted.

Fig. 4: Analysis of DynHAC on MNIST.

VI. Conclusion

We introduce the first fully dynamic HAC algorithm for the popular average-linkage version of the problem, which can maintain a $1+\epsilon$ approximate solution. DynHAC can handle each update up to 423x faster than what it would take to recompute the clustering from scratch. At the same time it achieves up to 0.21 higher NMI score than the state-of-the-art dynamic hierarchical clustering algorithms, which do not provably approximate HAC.

References

[1] L. Dhulipala, J. Łącki, J. Lee, and V. Mirrokni, "Terahac: Hierarchical agglomerative clustering of trillion-edge graphs," *SIGMOD*, 2023.

[2] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[3] P. Berkhin, "A survey of clustering data mining techniques," in *Grouping Multidimensional Data*. Springer, 2006.

[4] C. C. Aggarwal and C. K. Reddy, Eds., *Data Clustering: Algorithms and Applications*. CRC Press, 2014.

[5] B. Leibe, K. Mikolajczyk, and B. Schiele, "Efficient clustering and matching for object class recognition." in *BMVC*, 2006.

[6] F. Murtagh and P. Contreras, "Algorithms for hierarchical clustering: an overview," *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, vol. 2, no. 1, 2012.

[7] ——, "Algorithms for hierarchical clustering: an overview, II," *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, vol. 7, no. 6, 2017.

[8] D. Müllner, "Modern hierarchical, agglomerative clustering algorithms," *arXiv preprint arXiv:1109.2378*, 2011.

[9] ——, "fastcluster: Fast hierarchical, agglomerative clustering routines for r and python," *Journal of Statistical Software*, vol. 53, 2013.

[10] I. Gronau and S. Moran, "Optimal implementations of upgma and other common clustering algorithms," *Information Processing Letters*, vol. 104, no. 6, 2007.

[11] T. Stefan Van Dongen and B. Winnepenninckx, "Multiple upgma and neighbor-joining trees and the performance of some computer packages," *Mol. Biol. Evol*, vol. 13, no. 2, 1996.

[12] Y. Zhao and G. Karypis, "Evaluation of hierarchical clustering algorithms for document datasets," in *Proceedings of the eleventh international conference on Information and knowledge management*, 2002.

[13] G.-J. Hua, C.-L. Hung, C.-Y. Lin, F.-C. Wu, Y.-W. Chan, and C. Y. Tang, "MGUPGMA: a fast UPGMA algorithm with multiple graphics processing units using NCCL," *Evolutionary Bioinformatics*, vol. 13, 2017.

[14] A. Kobren, N. Monath, A. Krishnamurthy, and A. McCallum, "A hierarchical algorithm for extreme clustering," in *ACM SIGKDD*, 2017.

[15] C. Blundell and Y. W. Teh, "Bayesian hierarchical community discovery," in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 26, 2013.

[16] A. Culotta, P. Kanani, R. Hall, M. Wick, and A. McCallum, "Author disambiguation using error-driven machine learning with a ranking loss function," in *Sixth International Workshop on Information Integration on the Web (IIWeb-07), Vancouver, Canada*, 2007.

[17] L. Dhulipala, D. Eisenstat, J. Lacki, V. Mirrokni, and J. Shi, "Hierarchical agglomerative graph clustering in poly-logarithmic depth," in *NeurIPS*, 2022.

[18] L. Dhulipala, D. Eisenstat, J. Łącki, V. Mirrokni, and J. Shi, "Hierarchical agglomerative graph clustering in nearly-linear time," in *International Conference on Machine Learning (ICML)*, 2021.

[19] N. Monath, K. A. Dubey, G. Guruganesh, M. Zaheer, A. Ahmed, A. McCallum, G. Mergen, M. Najork, M. Terzihan, B. Tjanaka *et al.*, "Scalable hierarchical agglomerative clustering," in *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, 2021.

[20] M. Bateni, S. Behnezhad, M. Derakhshan, M. Hajiaghayi, R. Kiveris, S. Lattanzi, and V. Mirrokni, "Affinity clustering: Hierarchical clustering at scale," in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 30, 2017. [Online]. Available: https://proceedings.neurips.cc/paper/2017/file/2e1b24a664f5e9c18f407b2f9c73e821-Paper.pdf

[21] B. Moseley, K. Lu, S. Lattanzi, and T. Lavastida, "A framework for parallelizing hierarchical clustering methods," in *ECML PKDD*, 2019.

[22] T. Tseng, L. Dhulipala, and J. Shun, "Parallel batch-dynamic minimum spanning forest and the efficiency of dynamic agglomerative graph clustering," in *SPAA*, 2022.

[23] A. K. Menon, A. Rajagopalan, B. Sumengen, G. Citovsky, Q. Cao, and S. Kumar, "Online hierarchical clustering approximations," *arXiv preprint arXiv:1909.09667*, 2019.

[24] T. Zhang, R. Ramakrishnan, and M. Livny, "Birch: an efficient data clustering method for very large databases," *SIGMOD*, 1996.

[25] N. Monath, M. Zaheer, and A. McCallum, "Online level-wise hierarchical clustering," in *ACM SIGKDD*, 2023.

[26] A. Garg, A. Mangla, N. Gupta, and V. Bhatnagar, "Pbirch: a scalable parallel clustering algorithm for incremental data," in *IDEAS*, 2006.

[27] N. Monath, A. Kobren, A. Krishnamurthy, M. R. Glass, and A. McCallum, "Scalable hierarchical clustering with tree grafting," in *ACM SIGKDD*, 2019.

[28] M. Zaheer, G. Guruganesh, G. Levin, and A. Smola, "Terrapattern: A nearest neighbor search service," *ArXiv e-prints*, 2019.

[29] B. Sumengen, A. Rajagopalan, G. Citovsky, D. Simcha, O. Bachem, P. Mitra, S. Blasiak, M. Liang, and S. Kumar, "Scaling hierarchical agglomerative clustering to billion-sized datasets," *arXiv preprint arXiv:2105.11653*, 2021.

[30] M. Bateni, S. Behnezhad, M. Derakhshan, M. Hajiaghayi, R. Kiveris, S. Lattanzi, and V. Mirrokni, "Affinity clustering: Hierarchical clustering at scale," *NeurIPS*, vol. 30, 2017.

[31] J. Holm, K. De Lichtenberg, and M. Thorup, "Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity," *Journal of the ACM (JACM)*, vol. 48, no. 4, 2001.

[32] B. M. Kapron, V. King, and B. Mountjoy, "Dynamic graph connectivity in polylogarithmic worst case time," in *Proceedings of the twenty-fourth annual ACM-SIAM symposium on Discrete algorithms*. SIAM, 2013.

[33] D. Nanongkai, T. Saranurak, and C. Wulff-Nilsen, "Dynamic minimum spanning forest with subpolynomial worst-case update time," in *2017 IEEE 58th Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE, 2017.

[34] M. Charikar, V. Chatziafratis, and R. Niazadeh, "Hierarchical clustering better than average-linkage," in *SODA*. SIAM, 2019.

[35] D. Vainstein, V. Chatziafratis, G. Citovsky, A. Rajagopalan, M. Mahdian, and Y. Azar, "Hierarchical clustering via sketches and hierarchical correlation clustering," in *International Conference on Artificial Intelligence and Statistics*. PMLR, 2021.

[36] V. Cohen-Addad, V. Kanade, F. Mallmann-Trenn, and C. Mathieu, "Hierarchical clustering: Objective functions and algorithms," *Journal of the ACM (JACM)*, vol. 66, no. 4, 2019.

[37] S. Dasgupta, "A cost function for similarity-based hierarchical clustering," in *Proceedings of the forty-eighth annual ACM symposium on Theory of Computing*, 2016.

[38] B. Moseley and J. R. Wang, "Approximation bounds for hierarchical clustering: Average linkage, bisecting k-means, and local search," *Journal of Machine Learning Research*, vol. 24, no. 1, 2023.

[39] M. Charikar and V. Chatziafratis, "Approximate hierarchical clustering via sparsest cut and spreading metrics," in *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM, 2017.

[40] N. Alon, Y. Azar, and D. Vainstein, "Hierarchical clustering: A 0.585 revenue approximation," in *Conference on Learning Theory*. PMLR, 2020.

[41] S. Naumov, G. Yaroslavtsev, and D. Avdiukhin, "Objective-based hierarchical clustering of deep embedding vectors," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 35, no. 10, 2021.

[42] A. Rajagopalan, F. Vitale, D. Vainstein, G. Citovsky, C. M. Procopiuc, and C. Gentile, "Hierarchical clustering of data streams: Scalable algorithms and approximation guarantees," in *ICML*, 2021.

[43] T. Sainburg, L. McInnes, and T. Q. Gentner, "Parametric umap: learning embeddings with deep neural networks for representation and semi-supervised learning," *ArXiv e-prints*, 2020.

[44] J. M. Geusebroek, G. J. Burghouts, and A. W. M. Smeulders, "The amsterdam library of object images," *International Journal of Computer Vision*, vol. 61, no. 1, 2005.

[45] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *CVPR*, 2009.

[46] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *CVPR*, 2016.

[47] R. Das, A. Godbole, N. Monath, M. Zaheer, and A. McCallum, "Probabilistic case-based reasoning for open-world knowledge graph completion," in *Findings of EMNLP*, 2020.

[48] ——, "Grinch," 2020, https://github.com/ameyagodbole/Prob-CBR/blob/main/prob_cbr/clustering/grinch_with_deletes.py.

[49] N. Monath, "GraphGrove," 2023, https://github.com/nmonath/graphgrove.

[50] S. Jayaram Subramanya, F. Devvrit, H. V. Simhadri, R. Krishnawamy, and R. Kadekodi, "DiskANN: Fast accurate billion-point nearest neighbor search on a single node," in *Advances in Neural Information Processing Systems*, 2019.

[51] M. Dobson, Z. Shen, G. E. Blelloch, L. Dhulipala, Y. Gu, H. V. Simhadri, and Y. Sun, “Scaling graph-based anns algorithms to billion-size datasets: A comparative analysis,” 2023.

Appendix

In Algorithm 5, we show the pseudocode for update vertex mapping VMap and compute vertices to delete from next round.

Algorithm 5 UpdateVMap

```
Input: $\mathcal{D}_p$, $p$, $V_i^d$, $\mathcal{D}_{\text{dirty}}$, VMap
Output: $V_{i+1}^d$
Update: VMap
$V_{\text{active}} \leftarrow$ active nodes in $H_p$
$V_{i+1}^d \leftarrow \emptyset$
for $v \in V_i^d$ do
    if $\texttt{VMap}(v) \notin V_{\text{active}}^{\text{contracted}}$ then
        $V_{i+1}^d \leftarrow V_{i+1}^d \cup \texttt{VMap}(v)$
    Remove $v$ from VMap
for $v \in V_{\text{active}}$ do
    $r \leftarrow \mathcal{D}_p$.root($v$)
    if $\texttt{VMap}(v) \neq r$ then
        $V_{i+1}^d \leftarrow V_{i+1}^d \cup \texttt{VMap}(v)$
        $\texttt{VMap}(v) \leftarrow r$
Return $V_{i+1}^d$
```

Lemma 2. *Consider a graph G and a graph $G' = G \cup (V, E) \setminus V^d$, which is obtained from G by adding a set of vertices V and edges E, and deleting vertices of V^d. Assume that each edge of E is incident to a vertex in V and not incident to any vertex in V^d. Let $U \subseteq V(G')$ be the set of vertices of G' which have different partition ids in G and G'. Then, $U \subseteq V \cup N_{G'}(V) \cup (N_G(V^d) \setminus V^d)$.*

Proof. A vertex can have different partition id in two cases: 1) it does not exist in G, or 2) it has a different partition id. The set of vertices satisfying case 1 is V.

Now we look at case 2. A vertex can only change the partition id if its neighborhood changes, so only the neighbors of V and V^d can change their ids. Each such vertex is contained either in $(N_G(V^d) \setminus V^d)$ (neighbors of deleted vertices that are themselves not deleted), or $N_{G'}(V)$. □

Lemma 3. *If a partition P exists and does not become dirty upon node update, all $(1+\epsilon)$-good merges within the partition are still $(1+\epsilon)$-good.*

Proof. By Definition 2, a $(1+\epsilon)$-good merge can stop being $(1+\epsilon)$-good if (1) $w_{\max}(u)$ increases, (2) $\mathrm{M}(u)$ decreases, or (3) u is deleted (same for v). We show that if a partition does not become dirty, none of the three cases can happen for u. The same arguments can be made for v.

First, observe that u must be still in P, otherwise P will be identified as dirty partition by Algorithm 3. We now analyze the three cases above. (1) If $w_{\max}(u)$ increases, then u must have a new neighbor with higher edge weight than its current maximum neighbor edge weight. However, by Definition 4, if u is the neighbor of a new node, it will be included in Δ_P, and P will be identified as dirty partition by Algorithm 3. (2) Only nodes with the same merge sequence can have the same node id. So $\mathrm{M}(u)$ cannot change. (3) If $u \in P$ is deleted, P would be identified as a dirty partition by Algorithm 3. Since we would have u changing partition from P to $\emptyset$. So u must still in P. □

Theorem 1. DynHAC *maintains a $(1+\epsilon)$-approximate dendrogram upon node insertions and deletions.*

Proof. All merges in DynHAC are made by SubgraphHAC. So all merges are $(1+\epsilon)$-good when the merge is made. A merge is only untouched after an update if the partition of both nodes in the merge still exists and is not dirty. By Lemma 3, good merges stay good if its partition is not dirty. If a partition does not exist anymore, its red node must have been deleted. So all its neighbors (which is all nodes in the removed partition) must belong to a dirty partition, and re-merged by SubgraphHAC. So all merges in DynHAC are $(1+\epsilon)$-good after updates. By Lemma 1, any dendrogram produced by a sequence of $(1+\epsilon)$-good merges is $(1+\epsilon)$ approximate. So DynHAC maintains a $(1+\epsilon)$-approximate dendrogram upon node insertions and deletions. □

Theorem 2. *The total size of dirty partitions in a round can be bounded by the size of the 4-hop neighborhood of all inserted and deleted nodes.*

Proof. Consider an inserted/deleted node x. This update may cause the neighbor of x to change its partition. Assume that it partition id becomes $r \neq x$ (or was $r \neq x$ prior to an insertion). The partition subgraph containing r can contain at most the 2-hop neighbors of r (including the inactive nodes). r is x's 2-hop neighbor. So in total, x can make at most its 4-hop neighbor dirty. □

Theorem 3. *Inserting n nodes with m edges into an empty graph using Algorithm 1 takes $O(R(m+n)\log^2 n)$, where R is the number of rounds. The space complexity is $O(Rm)$.*

Proof. The bottleneck of Algorithm 1 is SubgraphHAC. The running time of SubgraphHAC on a graph containing n vertices and m edges is $O((m+n)\log^2 n)$ [1]. □

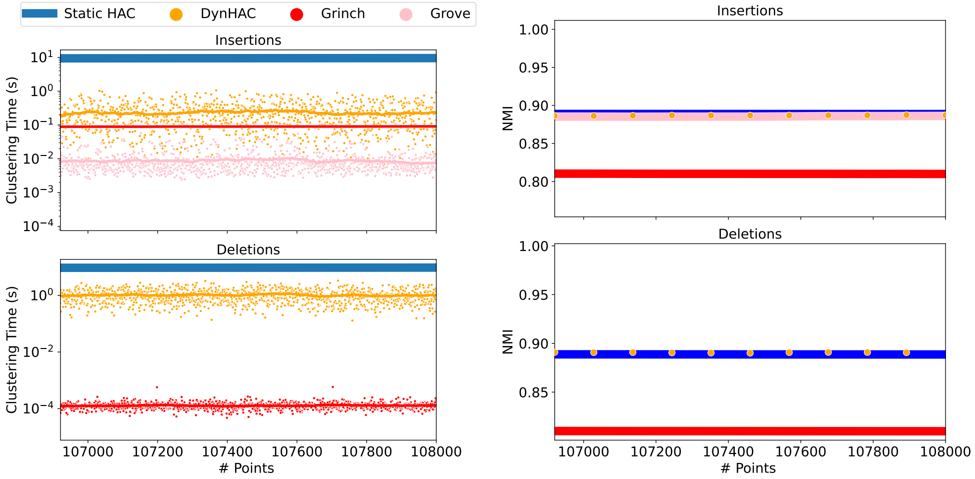


Fig. 5: Running time and quality on ALOI for static HAC and our DynHAC insertion and deletion, and GINRCH insertion and deletion.

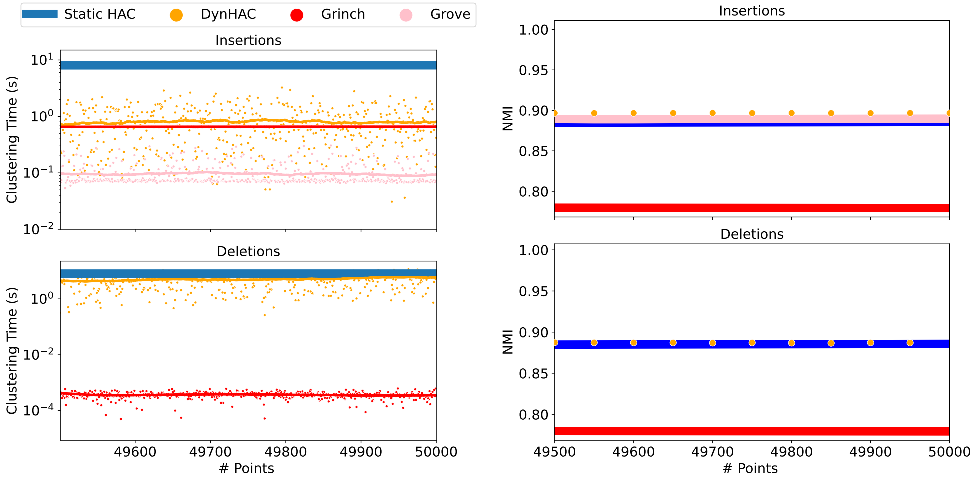


Fig. 6: Running time and quality on ILSVRC for static HAC and our DynHAC insertion and deletion, and GINRCH insertion and deletion.

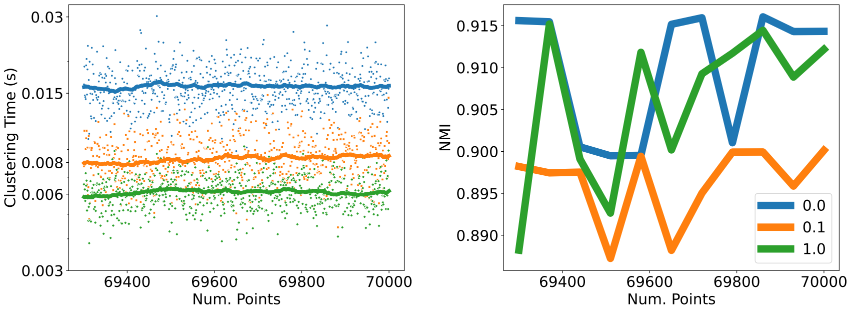


Fig. 7: DynHAC Insertion with different ϵ values on MNIST.

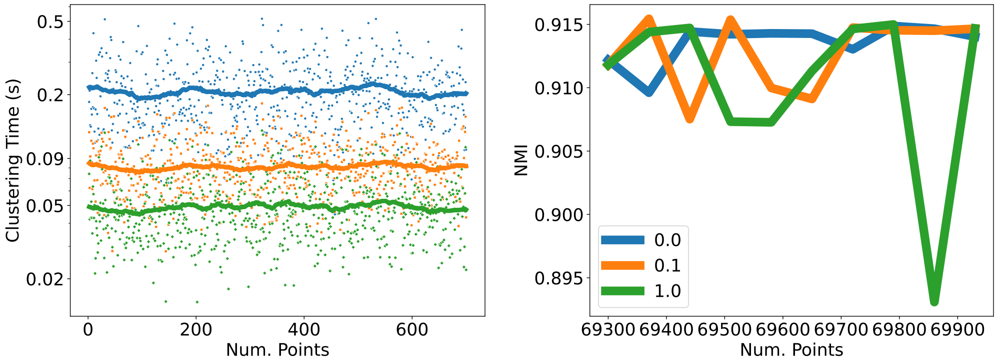


Fig. 8: DynHAC Deletion with different ϵ values on MNIST.

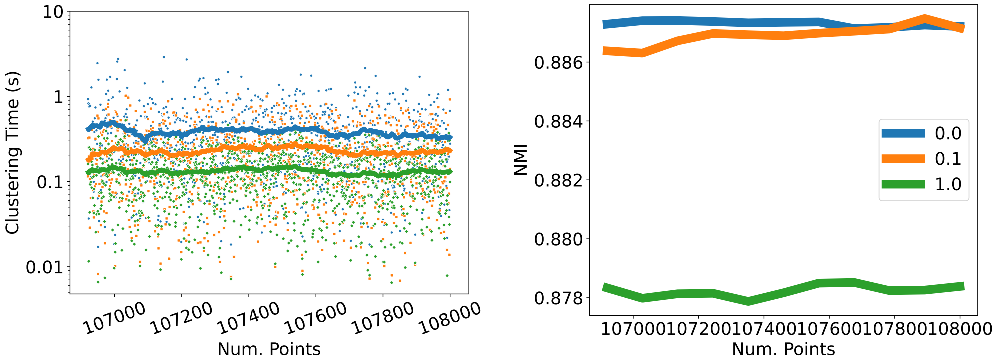


Fig. 9: DynHAC Insertion with different ϵ values on ALOI.

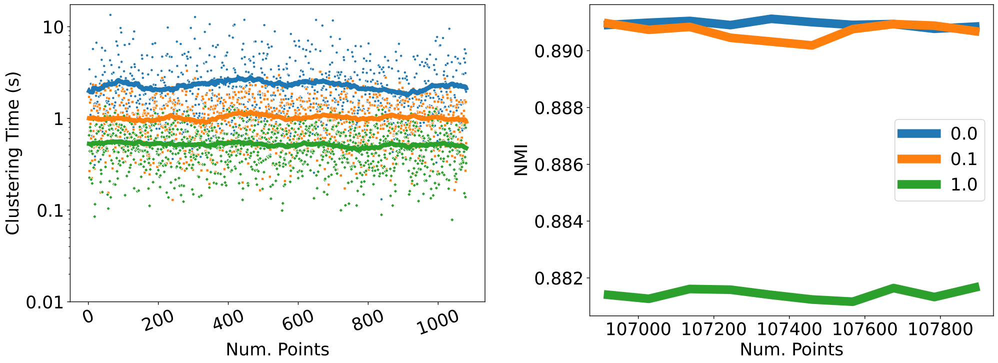


Fig. 10: DynHAC Deletion with different ϵ values on ALOI.

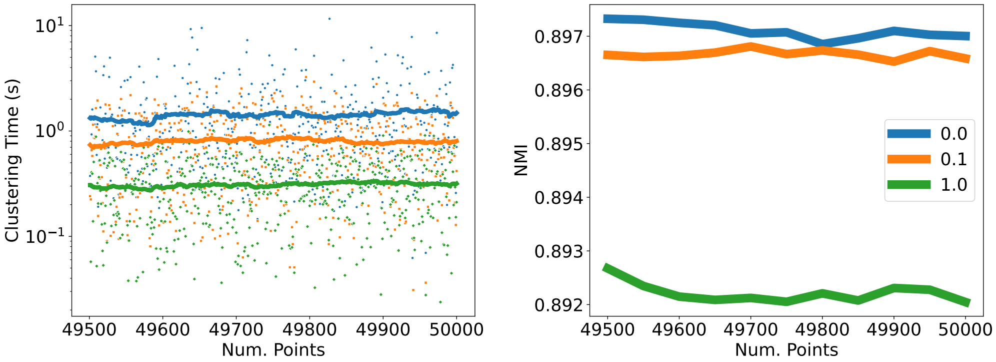


Fig. 11: DynHAC Insertion with different ϵ values on ILSVRC.

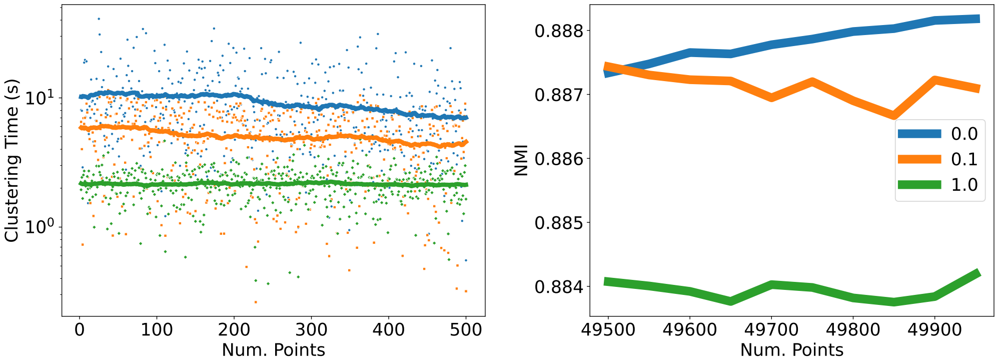


Fig. 12: DynHAC Deletion with different ϵ values on ILSVRC.

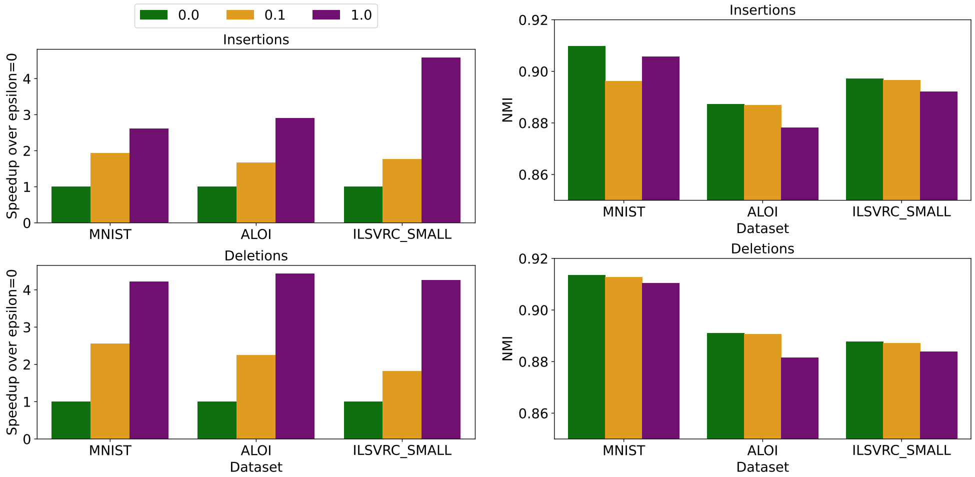


Fig. 13: Speedup of DynHAC with different epsilon values over $\epsilon = 0$, and the NMI values when using different epsilon values.

	Algorithm	Clustering	NMI	Speedup
Dataset				
MNIST	Static HAC	3.687000	0.915277	1.000000
MNIST	DynHAC	0.008703	0.900133	423.631747
MNIST	GRINCH	0.001392	0.686119	2648.991842
MNIST	Grove	0.002099	0.727114	1756.245833
ALOI	Static HAC	9.546000	0.888592	1.000000
ALOI	DynHAC	0.230439	0.887153	41.425311
ALOI	GRINCH	0.090949	0.809878	104.959618
ALOI	Grove	0.007629	0.886217	1251.328324
ILSVRC_SMALL	Static HAC	8.057000	0.885706	1.000000
ILSVRC_SMALL	DynHAC	0.804534	0.896586	10.014496
ILSVRC_SMALL	GRINCH	0.657990	0.779309	12.244861
ILSVRC_SMALL	Grove	0.093344	0.889749	86.315415

	Algorithm	Clustering	NMI	Speedup
Dataset				
MNIST	Static HAC	3.687000	0.915277	1.000000
MNIST	DynHAC	0.092865	0.900133	39.702794
MNIST	GRINCH	0.000171	0.686119	21508.204239
ALOI	Static HAC	9.546000	0.888592	1.000000
ALOI	DynHAC	1.382100	0.887153	6.906881
ALOI	GRINCH	0.000200	0.809878	47722.081030
ILSVRC_SMALL	Static HAC	8.057000	0.885706	1.000000
ILSVRC_SMALL	DynHAC	5.157170	0.896586	1.562291
ILSVRC_SMALL	GRINCH	0.000612	0.779309	13164.591869

TABLE III: Clustering time averaged over the last (insertion) and first (deletion) 100 updates. NMI after the last insertion.

	epsilon	Clustering	NMI	Speedup
Dataset				
MNIST	0.0	0.015968	0.909773	1.000000
MNIST	0.1	0.008260	0.896266	1.933223
MNIST	1.0	0.006112	0.905788	2.612785
ALOI	0.0	0.383982	0.887300	1.000000
ALOI	0.1	0.229246	0.886907	1.674979
ALOI	1.0	0.132040	0.878230	2.908074
ILSVRC_SMALL	0.0	1.413303	0.897105	1.000000
ILSVRC_SMALL	0.1	0.796748	0.896663	1.773840
ILSVRC_SMALL	1.0	0.308274	0.892214	4.584564

	epsilon	Clustering	NMI	Speedup
Dataset				
MNIST	0.0	0.206985	0.913558	1.000000
MNIST	0.1	0.080998	0.912764	2.555432
MNIST	1.0	0.049022	0.910421	4.222281
ALOI	0.0	2.292530	0.890944	1.000000
ALOI	0.1	1.017535	0.890677	2.253022
ALOI	1.0	0.517135	0.881430	4.433139
ILSVRC_SMALL	0.0	9.178508	0.887809	1.000000
ILSVRC_SMALL	0.1	5.057779	0.887121	1.814731
ILSVRC_SMALL	1.0	2.154927	0.883938	4.259313

TABLE IV: Average running time and NMI of DynHAC with different values for ϵ. Speedup is the speedup over $\epsilon = 0$.