Approximate Single-Linkage Clustering Using Graph-based Indexes: MST-based Approaches and Incremental Searchers

Camilla Birch Okkels[1], Erik Thordsen[3], Martin Aumüller[1], Arthur Zimek[2], and Erich Schubert[3]

[1] IT University of Copenhagen, Denmark
{cabi, maau}@itu.dk
[2] University of Southern Denmark, Denmark
zimek@imada.sdu.dk
[3] TU Dortmund, Germany
{erik.thordsen, erich.schubert}@tu-dortmund.de

Abstract. Current exact single-linkage clustering algorithms have asymptotically quadratic complexity. We present algorithms for *approximate* single-linkage clustering with empirically near-linear scalability. We explore both graph index-based incremental nearest neighbor search and an iterative exploration scheme on the graph index approximating the MST of the reachability graph similar to Kruskal. As graph index, we use both the bottom layer and a combination of all layers of an HNSW as a stand-in for connected search graphs. We provide experiments comparing the clusterings to baselines such as exact single linkage implementation and an algorithm using metric tree-based searchers. We explore the impact of the HNSW hyperparameters on the performance in terms of running time and clustering quality and evaluate the empirical asymptotic complexity.

Keywords: Clustering · Hierarchical Clustering · Approximate Clustering.

1 Introduction

Clustering is one of the fundamental data mining tasks aiming at identifying groups in data. What makes good groups depends on the adopted clustering paradigm. Some methods aim at finding compact clusters (typically assuming a given number of clusters), others aim at finding groups of connected points, where the connectivity is a transitive property, allowing for non-compact, arbitrarily shaped clusters. The first group is often named "partitioning", the second "density-based". Particular examples are k-means for a partitioning-based method, and DBSCAN [4] for a density-based method. While these two categories both result in a set of "flat" clusters, a third category finds hierarchies of clusters, where the clusters could fall in the "partitioning" or in the "density-based" category, and larger clusters can contain smaller clusters (and those in turn even smaller clusters), allowing to mine complex concept hierarchies or studying a dataset at different levels of granularity.

Traditionally, clustering algorithms need to determine distances between points in a metric space, often in the form of neighborhood searches from a given point. In big datasets containing high-dimensional vectors, these operations are often the limiting factor in terms of efficiency and, hence, applicability. Indexing methods, which organize the data points in a way that support neighboring searches, can be employed to facilitate efficient search where needed in the clustering routine [8,16]. However, for challenging datasets, in particular high-dimensional ones, traditional indexing methods would often degenerate and the runtime of the algorithm would become more expensive than a linear scan [23]. While there is little hope [2] for exact similarity search with "better-than-linear" worst case performance, researchers have made significant progress on using approximate, "inexact" techniques to speed up neighborhood searches. In particular, the graph-based approach HNSW [10] led to a paradigm shift from tree- or hashing-based approaches towards graph-based approaches, see for example the recent survey by Azizi et al. [3].

In the context of clustering, some work has been done to adapt or enhance "flat" clustering methods for handling large datasets efficiently using approximate neighbors [5,11,14,24]. On the other hand, hierarchical clustering has been more often sped up with other techniques (e.g., parallelization) [22,13,6]. Notable exceptions for a concrete linkage criterion are the hashing-based methods that were described by Koga et al. [7] (for a single-linkage heuristic) and Abboud et al. [1] for Ward's linkage. The main conceptual challenge of efficient, hierarchical clustering is that all points have to be merged into a single cluster eventually. However, this means that in the beginning close neighbors have to be merged together, while at the end almost unrelated points are the ones that lead to a cluster being merged. Approximate neighbor search methods—if used as a surrogate for the exact indexing methods—can only help with the first step.

In this paper we explore the use of HNSW [10] in a whitebox manner (cf. the discussion of blackbox vs. whitebox use of approximate nearest neighbor methods albeit for outlier detection by Okkels et al. [12]). This means that we operate directly on the approximate neighborhood graph derived by HNSW to derive an approximate hierarchical clustering structure, which is a new direction in approximate hierarchical clustering. The main challenge remains how to use the information present in the graph. We consider three different approaches: one uses the HNSW graph to generate candidate pairs for an incremental searcher as described by Schubert [15], while the other two build a hierarchical clustering directly on top of the graph.

Our contributions can be summarized as follows:

- We initiate the study of graph-based hierarchical clustering and explore the design space of such algorithms. In particular, we propose three hierarchical clustering algorithms based on single-linkage clustering. As a byproduct we propose a novel method for incremental searching a graph-based index.
- We carry out an empirical evaluation of the proposed methods, identifying suitable hyperparameter settings and showing their competitiveness against state-of-the-art baselines.
- We identify advantages and disadvantages of using approximate, graph-based hierarchical clustering methods.

In the following, we discuss technical background and closely related methods (Section 2), propose our methods of hierarchical clustering with HNSW (Section 3), evaluate the method (Section 4), and conclude our findings (Section 5).

2 Background and Related Work

Let $(\mathcal{X},d)$ be a point space equipped with a distance measure $d\colon \mathcal{X}\times\mathcal{X}\to\mathbb{R}^+$, for example d-dimensional Euclidean space with $\mathcal{X}:=\mathbb{R}^d$ with $d(a,b):=\|a-b\|_2$. Given a dataset $S\subseteq\mathcal{X}$ of N points, the goal of hierarchical clustering is to publish a rooted binary tree that has n leaves which represent the n data points. The children in the subtree rooted at an inner node belong to the same clustering, thus introducing a hierarchy of clusterings. The selection of which nodes to merge is known as the *linkage criterion*. Traditionally, the tree representation is called a *dendrogram*.

2.1 Hierarchical Single Link

Hierarchical Agglomerative Clustering is probably the oldest clustering method, whose origins can be traced back to numerical taxonomy in the 1950s [18]. Several variants and algorithms of this method exist, and the basic textbook version can be explained as treating each point as a separate cluster initially, then always merging the two closest clusters. Depending on how we measure the distance of clusters, we can obtain very different behavior. In this work, we will focus on *single-linkage clustering*, where the distance of two clusters C_A and C_B is defined as $\min_{a\in C_A,b\in C_B} d(a,b)$: intuitively, the distance is defined by the single shortest edge between the two clusters.

Single-linkage clustering is effectively equivalent to finding the minimum spanning tree (MST) in the complete graph (S,E,w) induced by the dataset S, where w represents the distance between a pair of points. The early method SLINK [17] runs in quadratic time and linear memory, and closely resembles Prim's algorithm for the minimum spanning tree. Another classic algorithm for MST is Kruskal's algorithm [9]. In classic Kruskal, all edges are inserted into a heap, ordered by increasing edge length; then we repeatedly poll the next shortest edge and merge these clusters with a union-find data structure. The complexity of the algorithm is $O(E\log E)$ where E is the number of edges. For a fully connected graph, this hence is $O(N^2\log N)$.

2.2 Hierarchical Clustering using Incremental Neighbor Search

For hierarchical clustering with incremental similarity search, Schubert [15] introduced multiple algorithm variants. For this work, we focus solely on single-linkage clustering, and specifically on the Heap-of-Searchers Single-Link (HSSL) algorithm. While Schubert [15] used a vantage point tree (VP-tree) for exact search, we will investigate approximate clustering with the HNSW graph. For the algorithms, we build on the HNSW graph framework created and explored by Thordsen et al. [20].

Similar to Kruskal's algorithm discussed above, the idea of this algorithm is to try to enumerate all edges in increasing length. Instead of using one large heap,

HSSL uses a heap of *incremental searchers*. For each point we create a searcher to enumerate its neighbors. After finding the first nearest neighbor, the searcher is inserted into the heap with this distance as key. We then pull from the heap the current nearest neighbor, and insert the edge into a union-find data structure, advance the searcher to the next nearest neighbor and insert it back into the heap. The efficiency of this approach depends very much on how much work we can avoid in incremental nearest neighbor search. A searcher could be implemented by computing all distances to the query point and putting them into a heap, but this clearly would yield an $O(N^2 \log N)$ run-time. Specifically, it is desirable to skip computing distances where (A) the points are already in the same cluster due to earlier cluster merges, or (B) the distances are larger than the longest edge of the minimum spanning tree. A tree-based similarity search structure such as the VP-tree used by Schubert [15] can help here, as lower bounds to subsets of the data help in postponing distance computations to these sets until we find no shorter edges. Furthermore, we may be able to skip computing distances of points that are already in the same cluster due to earlier merges.

2.3 Hierarchical Navigable Small World Graphs

Hierarchical Navigable Small World (HNSW) [10] is a graph-based indexing structure for approximate nearest neighbor search. Given a dataset $S \subseteq \mathcal{X}$ and parameters M,efC, a hierarchical graph $G=(V,E)$ is built. On each layer of the graph, a point is represented by a vertex with edges to a diverse set of nearest neighbors. The parameter M determines the maximum number of neighbors per node, and efC limits the width of the beam search during construction, see ef below. The graph structure is constructed hierarchically by adding points probabilistically to the layers with higher layers having lower probability of points appearing.

A search in this graph is carried out in the following way: starting at a node in the top layer, the graph is traversed in a greedy fashion, moving to neighbors that are closer to the query point. When no neighbors are found to be closer to the query, the search moves to the next layer down. This is repeated until the search terminates on the bottom layer and the nearest points found can be returned. Instead of making progress only for a single point, the queue of potential points is restricted to ef points, known as the *width of the beam*. HNSW builds the graph incrementally by inserting each point at a time. Inserting a point into the graph works by searching for the point to be inserted using the standard search algorithm, and choosing a diverse set of neighbors based on the nodes explored by the search, which is known as the *neighborhood selection rule*.

3 Single-Link Clustering with HNSW

In the following we discuss three approaches to perform single-linkage clustering using the HNSW graph. All approaches will start by constructing an HNSW-based index for a given set of hyperparameters. The running time of finding the hierarchical clustering is thus the sum of index building time and running the

Algorithm 1: HNSW-KRUSKAL(S, M, efC, $minPts$)

```
1  G ← HNSW(S,M,efC) // instantiate an HNSW-graph
2  H ← Min heap containing all edges from G with distance as weight
3  G_0 ← bottom layer of G
   /* Iterate over MST edges of reachability graph in ascending order */
4  while not all points are in the same cluster do
5      d_ij, (i,j) ← H.pop()
6      if i and j are in the same cluster then continue
7      Add edge i, j, d_ij to the Dendrogram.
8      foreach neighbor k of j in G_0 do
9          if i and k are in different clusters then H.push(d(s_i,s_k),(i,k))
10     foreach neighbor k of i in G_0 do
11         if j and k are in different clusters then H.push(d(s_j,s_k),(j,k))
12 return Dendrogram of merge information
```

clustering approach on the given index. In the following, we will assume that the graph is built. Note that the HNSW graph is not necessarily connected, although the "long edges" in the upper layers help. These exist because the set of near points on higher levels represent a small sample of the dataset; close neighbors are very likely to be actually far away. If the graph is not connected, we can enforce connectivity by sampling one point from each component and computing their distances. From a clustering perspective, we do want these components to be separated, and hence finding a suboptimal long edge does not negatively affect the clustering result.

3.1 The baseline: Minimum Spanning Tree on HNSW

As baseline approach, we simply compute the minimum spanning tree (MST) of all the edges in the HNSW graph. Given that we limit the number of neighbors of each point in this graph to be at most a small constant $M = O(1)$, it is easy to see that the we have $O(N)$ edges. Given the HNSW index, finding the MST takes time $O(N \log N)$ and represents a single-linkage-based, approximate clustering. We stress that the MST of the HNSW graph is not the MST of the complete graph induced by the dataset as discussed in Section 2.2.

3.2 Kruskal-style Hierarchical Clustering

To improve over the baseline, Algorithm 1 describes how to heuristically simulate running Kruskal's algorithm on the complete graph induced by the dataset. To run Kruskal's algorithm, we would need access to all $O(N^2)$ edges sorted by their distance but we want to avoid materializing all edges. We approach it like this: Initially, we begin with only the edges of the bottom layer of the HNSW graph, which only stores $O(N)$ edges, as in the baseline approach. To improve the clustering quality, when we poll an edge (a,b) from the graph, we insert new edges with one additional step in the HNSW graph: let N_a and N_b be the neighbors of a and b,

Algorithm 2: HNSW-HSSL(S,M,efC,efS,*UnionFind*)

```
1  S ← {}, H ← {}                                   // searchers and primary heap
2  G ← HNSW(S,M,efC)                                // instantiate an HNSW-graph
3  foreach p ∈ S do
4      searcher ← HNSWSearcher(p, G, UnionFind, efS)   // See Algorithm 3
5      S[p] ← searcher
6      H.push(searcher.peek(), p)
7  while not everything merged do
8      d_pq, p ← H.pop()
9      d_pq, q ← S[p].advance()
10     if not UnionFind.connected(p,q) then
11         UnionFind.merge(p,q)
12         Dendrogram.append( (p,q,d_pq) )
13     H.push(S[p].peek(), p)
14 return Dendrogram
```

then we add edges $\forall_{n_b \in N_b}(a,n_b)$ and $\forall_{n_a \in N_a}(n_a,b)$ to the graph if (1) these points are not yet connected, (2) the edge has not previously been added to the heap.

The running time of the algorithm is clearly not as straight-forward to discuss as the baseline algorithm. While initially there are only $O(N)$ edges, we might carry out the while loop without making any progress for pairs of points which are close to each other but have already been merged. Thus it will be interesting to evaluate the difference in running time and quality between the baseline and this Kruskal-style variant. The implementation of the algorithm comes with subtle challenges. Because of cliques in the graph, adding edges to all neighbors of the other point can easily cause a lot of duplicate edges to be added. Hence we need to keep track of which edges we have already explored. But in the worst case, we may have $O(N^2)$ edges and a simple hash set of all edges may already need a lot of memory. We can reduce this memory overhead by almost a factor of two if we use an array of hash sets where the smaller node id is the array index, and the larger node id is stored in the set. When running out of memory, it may be possible to add a cleanup step where all edges are removed from these maps that are connected in the union-find data structure.

3.3 Incremental Heap of Searchers using HNSW

Our final proposal is to implement the incremental searcher architecture HSSL that was considered by Schubert [15] with HNSW. Algorithm 2 presents the overview of the incremental search approach, which can be viewed as a vertex-centered version of Kruskal's algorithm. At the start, we initialize an incremental searcher for each point in the dataset. In rounds, we take the smallest edge that is present in any of the N searchers and add it to the clustering (if the edge does not connect two points which are already clustered together). Our novel contribution is the design of an approximate incremental searcher data structure for HNSW, which might be of independent interest. Our pseudocode of the incremental searcher uses the Python

Algorithm 3: HNSWSearcher(q,$\mathcal{G}$,uf,ef)

Input: q – data point; $\mathcal{G}$ – HNSW graph; uf – UnionFind-data structure; ef – number of candidates to keep
Output: Generator of candidate neighbors

```
1  E ← MinPrioQueue()                                   // expand queue
2  C ← MinPrioQueue()                                   // candidate queue
3  V ← {}                                               // visited set
4  E.push(0,q)                                          // starting vertex
   /* generator function to find neighbors in approx. ascending order */
5  while E is not empty do
6      (d,x) ← E.pop()
7      N ← G.get_neighbors(x)
8      foreach p ∈ N do
9          if not uf.connected(q,p) and p ∉ V then
10             d_xp ← d(x,p)                            // compute distance to candidate
11             C.push(d_xp,p)                           // add to candidate queue
12             E.push(d_xp,p)                           // add to expansion queue
13             V.add(p)                                 // prevent duplicates in queue
14     while |C| > ef do
15         yield C.pop()                                // return candidate and suspend
16 while C not empty do
17     yield C.pop()                                    // return remaining candidates
```

generator pattern and is presented as Algorithm 3. When `next()` is called on this generator, it runs until the next call to `yield`, then execution is suspended. The main idea is to keep a buffer C of at least ef candidate neighbors, similar to a beam search for kNN in graph indexes, and a priority queue E of vertices to expand next. In contrast to a kNN search, we cannot truncate these sets. While we have less than ef candidates, we pop the next vertex from the expansion queue E, and for any neighbor we check that (1) we have not visited it already, (2) it is not yet connected in our union-find structure. We then compute the exact distance and enqueue it both in the candidate queue C and the expansion queue E. If we have at least ef candidates, the current best candidate is returned and removed from the heap.

From the three proposed methods, the HSSL variant uses HNSW's search method almost in a black-box method and should thus—given "good hyperparameter" values—accurately represent the neighborhood of a data point. However, it should also be the slowest method since it will by design enumerate close neighbors before far away neighbors, which might become a bottleneck in the final process of merging (far away) clusters.

4 Evaluation

Implementation Details and Experimental Setup. Experiments were run on a machine with 2x14 core Intel Xeon E5-2690v4 (2.60 GHz) with 512GB RAM using

Table 1: Overview over Datasets.

Dataset	Min. sample size	Max. sample size	dim. (d)
ALOI	50 000	110 000	63
MNIST	10 000	70 000	784

Ubuntu 20.04.6 LTS. The implementation was split into a backend HNSW implementation using the Rust programming language that exposes Python bindings. These Python bindings are used in the experimental pipeline. The incremental searcher (Algorithm 3) are implemented in Rust, but the merging logic in Algorithm 2 uses a pure Python solution. The part of the implementation in rust is from the framework by Thordsen et al. [20]. The core implementation and the benchmarking framework can be found at `https://github.com/CamillaOkkels/HSSL` and `https://github.com/CamillaOkkels/singleLinkage-benchmark` respectively.

Datasets. Table 1 summarizes the datasets that we have used to carry out the evaluation. We focus our study on real-world data using the ALOI and MNIST datasets, which have been used in cluster analysis before [11]. To measure scalability of the proposed implementations, we subsample the datasets, always ensuring that a smaller dataset is fully contained in a larger sample.

Evaluation criteria. As a performance measurement, we measure the overall time it took to compute the hierarchical clustering. To measure the quality of the clustering, we compare the produced approximate clustering to an exact baseline clustering. Given two dendrograms, we first measure the cophenetic distances [19] for each dendogram, which measure the minimum height of the dendrogram needed to connect two points. Our quality measure is the Pearson correlation between these distance matrices, which we call *Cophenetic correlation*. Since the distance matrix contains $O(N^2)$ entries, we compute the correlation on a subsample of 10^7 pairs. Experiments using traditional metrics such as the adjusted Rand index (ARI) require an additional step to cut the dendrogram into partitions that introduces noticeable instability, while also offering only a limited capability to differentiate results as distances are not used by this measure.

Baselines. We compare our implementations against the industry-standard single-linkage implementation in the `SciPy` library [21]. As a direct competitor based on incremental searching, we use the VPTree-based implementation by Schubert [15].

Hyperparameter settings. All three implementation used *max_ build_ heap_ size* (*efC*) and *lowest_ max_ degree* (M) for the HNSW graph construction. The HNSW-HSSL method further took an additional parameter *ef* (*efS*). We let *efS* take values {5,11,22,47,100}, we let *efC* take values {25,42,71,119,200} and we let M take values: {14,26,51,100}. All the values were chosen to span a large interval, and to be evenly spaced in log-space. Testing of M for smaller values suggested that a minimum value of around 14 ensured the HNSW graph would be connected - with very few exceptions due to their random nature.

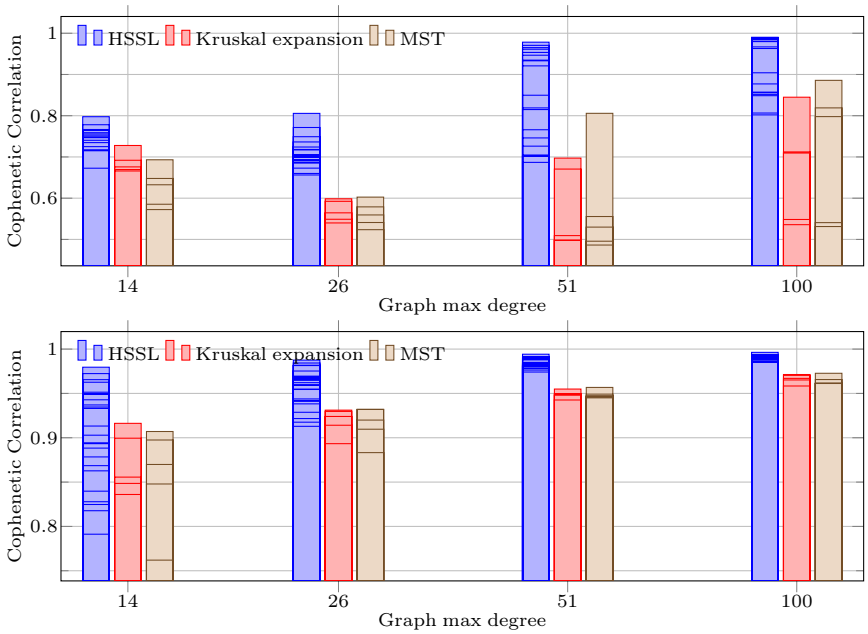

Fig. 1: Cophenetic correlation on ALOI (top) and MNIST (bottom).

Key findings. We summarize our key findings as follows and will detail them in the following subsections.

1. There is a clear difference between the three proposed methods in terms of running time and achieved quality. The MST baseline is fastest but sensitive to hyperparameter choices, whereas the HSSL variant achieves the best quality with robust parameter selection. The difference in speed between MST and HSSL can be up to a factor 100x.
2. The additional work done in the Kruskal variant (Algorithm 1) does not yield big improvements in clustering quality over the MST baseline.
3. Both Kruskal and HSSL do not exhibit worst-case quadratic scaling on the tested datasets.
4. All three variants outperform the industry-standard `SciPy` baseline.

4.1 Internal Evaluation

Accuracy. Figure 1 shows a bar plot comparing the accuracy of the different methods for varying values of M. Each thick horizontal line in a bar represents the quality achieved for a certain set of parameters (for Kruskal and MST, a single *efC*, for HSSL a pair (*efC*, *efS*)). For each variant, a cophenetic correlation of at least .8 can be achieved by a careful selection of hyperparameters. In general, it is more difficult for our variants to achieve good quality on ALOI, compared to the MNIST dataset. We

observe that the accuracy of HSSL increases with increasing M on both ALOI and MNIST. For both datasets and M choices, it outperforms the other two approaches regarding the best achievable clustering quality. The Kruskal and MST methods had increasing quality with increasing M for MNIST, but for ALOI we observed a slight decrease, until a slight increase again between $M = 51$ and $M = 100$. Interestingly, the baseline MST approach shows comparable performance, even outperfoming Kruskal for some parameter choices. However, in particular on the ALOI dataset, the MST-based approach is more sensitive to hyperparameter choices.

Running time. Figure 2 plots the performance of the individual methods in relation to the achieved clustering quality. Due to space constraints, we focus the discussion on the MNIST dataset and plot only the Pareto frontier of the measurements. The results that we measured on ALOI were similar.

We observe that for HNSW-HSSL, a larger M value allows for more accurate clustering quality. However, even the smallest value resulted in a clustering quality above .9 while leading to an efficient clustering algorithm. Getting from a clustering quality of .9 to close to 1 increases the running time by almost a factor of 10. The search parameter *efS* has little influence on the clustering quality and good results are achievable with the minimum setting of 5. This is helpful in terms of the robustness of the algorithm, since HNSW-HSSL comes with an additional search parameter that is missing in the other two variants.

For the other two variants, we again observe the trend that increasing M improved quality at the cost of slower run times on MNIST. The *max_ build_ heap_ size* (*efC*) parameter lead to smaller variations in the clustering quality at the expense of running time.

Comparing all three plots, we notice major differences in running time. A clustering quality of at least .9 can be achieved in around 2 seconds (MST), 12 seconds (Kruskal), and 100 seconds (HSSL).

While Figure 2 visualizes the Pareto frontier and gives interesting observations over the hyperparameter choices, Figure 3 provides a scatter plot over all possible hyperparameter choices and relates measured running time to clustering quality. The plot reinforces the observations that each variant provides a very distinct performance tradeoff, i.e., the slowest parameter settings for MST are faster than the fastest for Kruskal, and similarily between Kruskal and HSSL. Quality-wise we observe that HSSL provides better quality for most of the hyperparameter choices. In particular the MST algorithm requires careful hyperparameter selection. For example, for ALOI there exist hyperparameters that require more time and provide poorer quality.

4.2 Comparison to other baselines

We now turn our focus to evaluate the three proposed methods against SciPy's single linkage clustering algorithm, and the VPTree incremental searcher by Schubert [15]. For these experiments, we subsample the datasets to carry out a scalability analysis. For each sample size, we use the same parameter choices for the individual

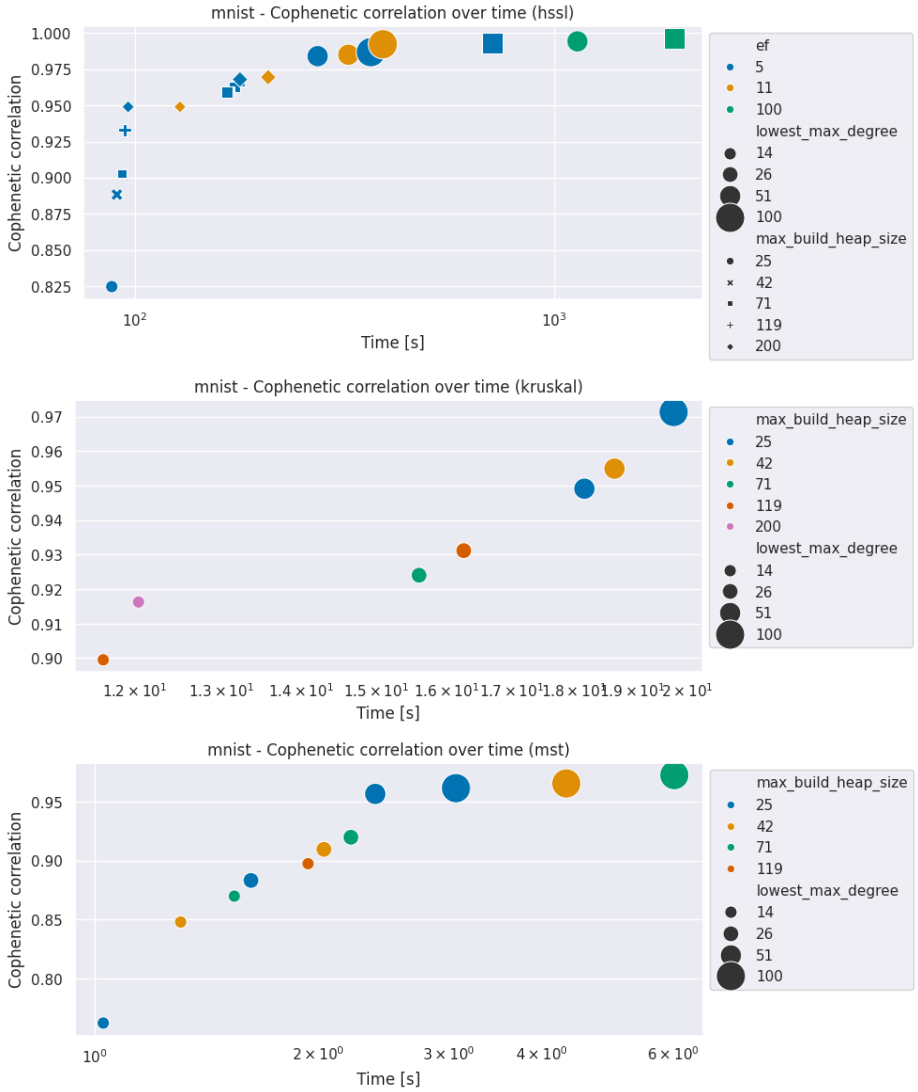

Fig. 2: Pareto frontier of optimal cophenetic correlation scores on MNIST for HNSW-HSSL (top), HNSW-Kruskal (middle), HNSW-MST (bottom).

method. For all three methods the parameters are chosen as the fastest settings above a 0.8 cophenetic correlation.

Figure 4 relates our three proposed methods to the competitors. To put the running times into context, we empirically fit regression lines to the observed running times. These running times are visible in the label of each method in the legend. Comparing to the industry-standard from `SciPy`, all of the proposed methods are

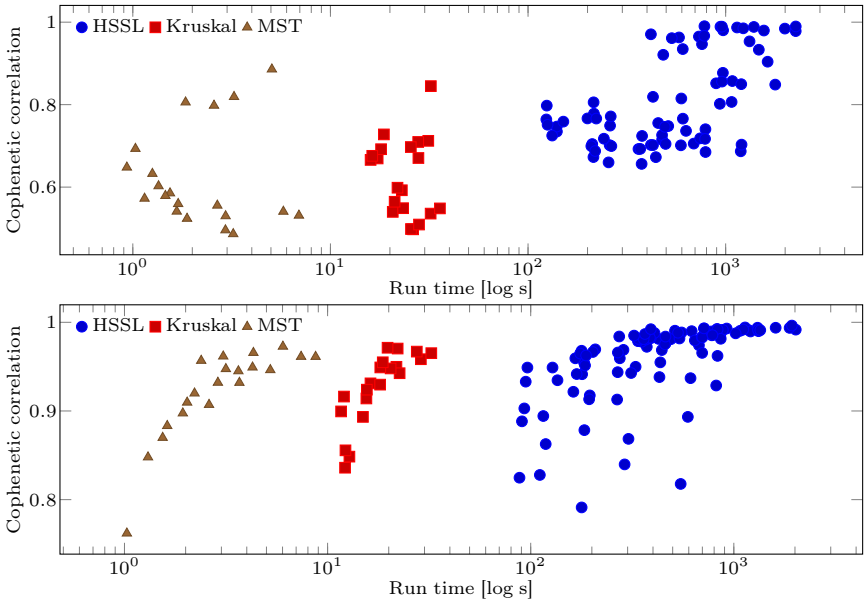

Fig. 3: Run time vs. approximation quality on ALOI (top) and MNIST (bottom)

faster. The difference is more pronounced for the MNIST dataset than for the ALOI dataset. Comparing the two implementations based on incremental searchers (`HNSWHSSL` and `VPTree`), we notice an interesting tradeoff in their performance: while the (exact) VPTree is faster on ALOI, it is slower on MNIST.

5 Conclusion

With this paper we initiated the study of efficient and accurate hierarchical clustering using HNSW as a baseline graph index. We proposed three methods: Two based on computing an MST in the HNSW graph, one based on the idea of incremental searchers. The evaluation highlighted advantages and limitations of the proposed methods. Interestingly, the most basic baseline of computing an MST of the HNSW graph can lead to a good hierarchical clustering on the tested datasets. If good quality needs to be ensured, the incremental searcher-based approach showed its advantages.

While we believe that we cover much of the design space of hierarchical clustering using graph-based approaches, there is much room for improvements. Our most intriguing question is whether we can enhance the graph building progress to build an augmented graph that contains information necessary to speed up hierarchical clustering. From a theoretical point, it would be nice to understand the theoretical limitations of the proposed methods. While we did not observe quadratic scaling on real-world data, constructing such input instances would be an important achievement for a theoretical understanding of the proposed methods.

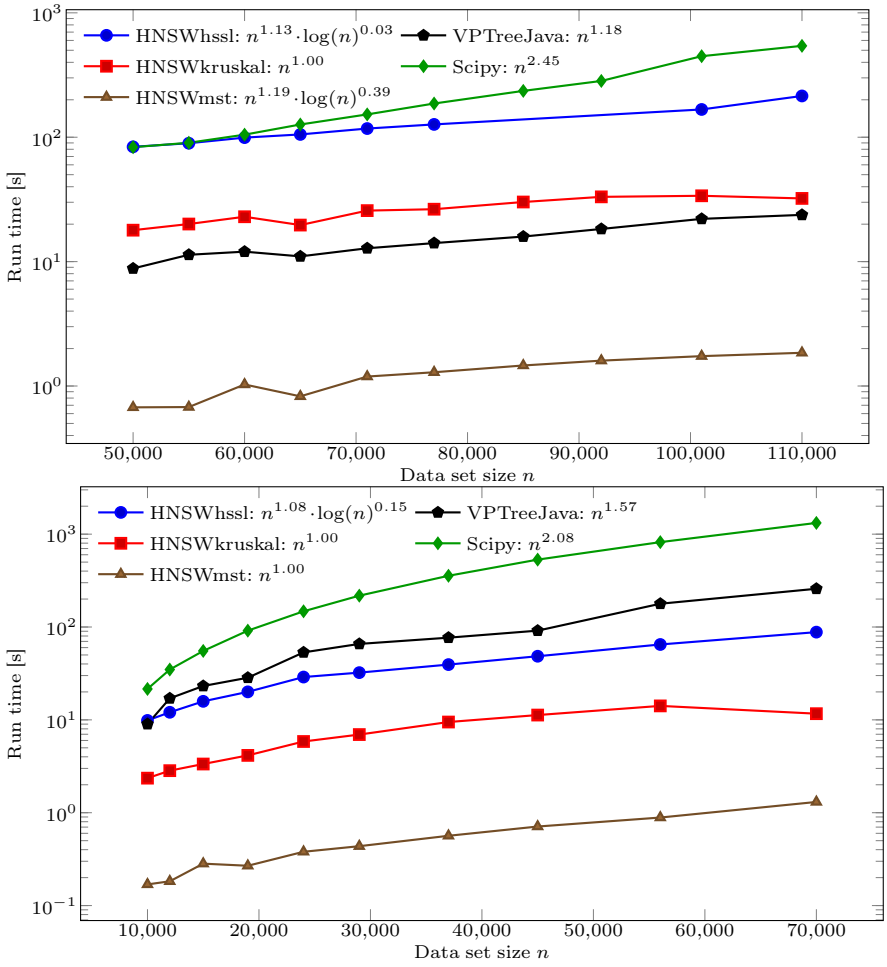

Fig. 4: Running time vs. sample size on ALOI (top) and MNIST (bottom)

Acknowledgments. This project received funding from the Innovation Fund Denmark for the project DIREC (9142-00001B).

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Abboud, A., Cohen-Addad, V., Houdrouge, H.: Subquadratic high-dimensional hierarchical clustering. In: NeurIPS. pp. 11576–11586 (2019)
2. Alman, J., Williams, R.: Probabilistic polynomials and hamming nearest neighbors. In: FOCS. pp. 136–150. IEEE Computer Society (2015)

3. Azizi, I., Echihabi, K., Palpanas, T.: Graph-based vector search: An experimental evaluation of the state-of-the-art. Proc. ACM Manag. Data **3**(1), 43:1–43:31 (2025)
4. Ester, M., Kriegel, H., Sander, J., Xu, X.: A density-based algorithm for discovering clusters in large spatial databases with noise. In: KDD. pp. 226–231 (1996)
5. Gan, J., Tao, Y.: On the hardness and approximation of euclidean DBSCAN. ACM Trans. Database Syst. **42**(3), 14:1–14:45 (2017)
6. Huang, Y., Yu, S., Shun, J.: Faster parallel exact density peaks clustering. In: ACDA. pp. 49–62. SIAM (2023)
7. Koga, H., Ishibashi, T., Watanabe, T.: Fast agglomerative hierarchical clustering algorithm using locality-sensitive hashing. Knowl. Inf. Syst. **12**(1), 25–53 (2007)
8. Kriegel, H., Schubert, E., Zimek, A.: The (black) art of runtime evaluation: Are we comparing algorithms or implementations? Knowl. Inf. Syst. **52**(2), 341–378 (2017)
9. Kruskal, J.B.: On the shortest spanning subtree of a graph and the traveling salesman problem. Proc. American Mathematical Society **7**(1), 48–50 (Feb 1956)
10. Malkov, Y.A., Yashunin, D.A.: Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs. IEEE TPAMI **42**(4) (2020)
11. Okkels, C.B., Aumüller, M., Thomsen, V.B., Zimek, A.: High-dimensional density-based clustering using locality-sensitive hashing. In: EDBT. pp. 694–706 (2025)
12. Okkels, C.B., Aumüller, M., Zimek, A.: On the design of scalable outlier detection methods using approximate nearest neighbor graphs. In: SISAP. pp. 170–184 (2024)
13. dos Santos, J.A., Iqbal, S.T., Naldi, M.C., Campello, R.J.G.B., Sander, J.: Hierarchical density-based clustering using mapreduce. IEEE Trans. Big Data **7**(1), 102–114 (2021)
14. Schneider, J., Vlachos, M.: Scalable density-based clustering with quality guarantees using random projections. Data Min. Knowl. Discov. **31**(4), 972–1005 (2017)
15. Schubert, E.: Hierarchical clustering without pairwise distances by incremental similarity search. In: Proc. SISAP. pp. 238–252 (2024)
16. Schubert, E., Sander, J., Ester, M., Kriegel, H., Xu, X.: DBSCAN revisited, revisited: Why and how you should (still) use DBSCAN. ACM Trans. Database Syst. **42**(3), 19:1–19:21 (2017)
17. Sibson, R.: SLINK: an optimally efficient algorithm for the single-link cluster method. Comput. J. **16**(1), 30–34 (1973)
18. Sokal, R.R., Michener, C.D., et al.: A statistical method for evaluating systematic relationships (1958)
19. Sokal, R.R., Rohlf, F.J.: The comparison of dendrograms by objective methods. Taxon **11**(2), 33–40 (1962)
20. Thordsen, E., Schubert, E.: Theoretical and practical insights into graph-based indexing. In: Proc. Int. Conf. on Similarity Search and Applications, SISAP (2025)
21. Virtanen, P., Gommers, R., Oliphant, T.E., Haberland, M., Reddy, T., Cournapeau, D., Burovski, E., Peterson, P., Weckesser, W., Bright, J., van der Walt, S., Brett, M., Wilson, J., Millman, K.J., Mayorov, N., Nelson, A.R.J., Jones, E., Kern, R., Larson, E., Carey, C., Polat, I., Feng, Y., Moore, E.W., VanderPlas, J., Laxalde, D., Perktold, J., Cimrman, R., Henriksen, I., Quintero, E.A., Harris, C.R., Archibald, A.M., Ribeiro, A.H., Pedregosa, F., van Mulbregt, P., SciPy: Scipy 1.0-fundamental algorithms for scientific computing in python. CoRR **abs/1907.10121** (2019)
22. Wang, Y., Yu, S., Gu, Y., Shun, J.: Fast parallel algorithms for euclidean minimum spanning tree and hierarchical spatial clustering. In: SIGMOD Conference. pp. 1982–1995. ACM (2021)
23. Weber, R., Schek, H., Blott, S.: A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In: VLDB. pp. 194–205 (1998)
24. Xu, H., Pham, N.: Scalable DBSCAN with random projections. In: NeurIPS (2024)