Fast and memory-efficient approximate minimum spanning tree generation for large data sets

Mahmood K. M. Almansoori[1], Andras Meszaros[1,2] , Miklos Telek[1,2],
[1] *Department of Networked Systems and Services*,
Budapest University of Technology and Economics, Budapest, Hungary
[2] *ELKH-BME Information Systems Research Group, Budapest, Hungary*
{almansoori,meszarosa,telek}@hit.bme.hu

Abstract

Minimum Spanning Tree (MST) based clustering algorithms effectively identify clusters of different shapes and densities within data sets. Conventional MST algorithms typically start by creating a distance matrix of the $n(n-1)/2$ pairs of data points, leading to a time complexity of $O(n^2)$. However, this initial step poses a computational bottleneck. To overcome this limitation, we present a novel method that constructs an initial random k-neighbor graph and optimizes this graph by employing a crawling technique to approximate the k Nearest Neighbors (kNN) graph efficiently. This crawling approach allows us to approximate the closest neighbors of each node. Subsequently, we use the approximate kNN graph to build an initial approximate MST and iteratively refine it by the same crawling process. Using this approach, we can obtain an approximate MST for a data set of size n with empirical cost around $O(n^{1.07})$ and a minimal $O(n)$ memory consumption. We have shown that the proposed method achieves such a level of performance with only a marginal accuracy reduction between 0.5% and 6%. The magnitude of the efficiency reduction depends on intrinsic data set characteristics and the value of k.

1 Introduction

Minimum Spanning Tree (MST) algorithms have numerous applications in the domain of big data due to their ability to extract essential structures from large data sets. Some of the notable applications are clustering, network design and optimization, and anomaly detection. In clustering, the goal is to group similar data points together into clusters while keeping dissimilar points separate. MST is a graph-theoretic concept that can be used to identify groups of closely related data points that can be clustered together [1, 2, 3]. MST is a tree that spans all the nodes of a graph while minimizing the total weight of the edges. In the context of clustering, the nodes represent data points, and the weights of the edges represent the similarity or dissimilarity between the points. A low-weight edge between two nodes indicates a high similarity, while a high-weight edge indicates a low similarity.

MST can be used to identify clusters of data points by partitioning the tree into subtrees, where each subtree represents a cluster [4, 5]. The subtrees can be identified by cutting the edges of the tree at a certain threshold weight. The resulting subtrees consist of groups of closely related points that can be clustered together. Additionally, MST can also be used as a pre-processing step for clustering algorithms [6, 7, 8]. It can reduce the complexity of clustering algorithms by focusing on the most relevant pairwise relationships and filtering out noisy or less informative connections. The resulting MST can serve as a compact representation of the data, facilitating faster and more efficient clustering.

Although numerous algorithms construct MST accurately and efficiently, these algorithms typically require a graph that reflects the relationships among the data points [9, 10]. However, building such a graph can be computationally expensive. In recent years, many algorithms have utilized the k Nearest Neighbors (kNN) graph to construct the MST. The kNN graph approximates the underlying data relationships by connecting each data point to its k nearest neighbors. By leveraging the kNN graph, MST construction algorithms can effectively capture the data structure and generate an MST that represents the relationships among the data points. This approach offers a computationally feasible and scalable solution for constructing MSTs in large data sets.

There are many algorithms that use kNN for classification and regression tasks in the field of machine learning and data mining. These algorithms are non-parametric (do not make any assumptions about the underlying data distribution) and instance-based (relies on the entire data set during the prediction phase) learning algorithms [11, 12]. However, finding the exact k nearest neighbors in high dimensional data sets is computationally expensive, especially for large data sets [13], since the brute force approach, which involves computing the distance between all point pairs in the data set, has a time complexity of $O(n^2)$. Therefore, approximate kNN search techniques have gained significant attention, since they are able to find near neighbors in large data sets much more efficiently [14].

While the approximate kNN intends to approximate the true kNN closely, due to the reduced computational complexity, the two can be different. The accuracy of approximate kNN search techniques for the entire data set can be measured by the difference between the approximate kNN distances and the true kNN distances:

$$\text{Accuracy} = \sum_{i=1}^{n} \sum_{j=1}^{k} \tilde{D}_{ij} - D_{ij}, \tag{1}$$

where n is the number of data points in the data set, k is the considered number of nearest neighbors, D_{ij} is the true distance to the jth nearest neighbor of the ith data point, and $\tilde{D}_{ij}$ refers to the distance of the ith data point to its jth nearest neighbor in the approximate kNN.

These techniques aim to get a balance between accuracy and computational complexity, allowing for faster search times while still providing reasonably accurate results. Several methods have been developed to address the approximate kNN search problem. The three main categories of such methods are spatial-tree-based, hashing-based, and graph-based methods.

Spatial tree-based methods partition the space into smaller regions or nodes, creating a hierarchical structure. The goal is to efficiently organize the data so that spatial queries like nearest-neighbor searches can be performed quickly. Examples of spatial trees include KD-tree [15], Ball-tree [16, 17] and K-Means tree [18]. The construction of a KD-tree involves recursively partitioning a set of points in a K-dimensional space. A splitting axis is chosen, often alternating between dimensions, and data points are separated based on their values along that axis. The median point along the chosen axis becomes the current node, and the process continues for the left and right subtrees until all points are organized into the tree structure.

Spatial-tree-based methods offer fast searching by traversing the tree structure. The advantages of tree-based methods include logarithmic search complexity, good performance for low-dimensional data, and support for range searches. However, they can struggle with high-dimensional data due to the curse of dimensionality and may suffer from unbalanced trees or inefficiency when the data distribution is skewed [19].

Hashing-based methods [20] map high-dimensional data points into lower-dimensional binary codes, enabling efficient indexing and searching. They offer fast query processing and are effective for large-scale data sets. These methods allow for efficient retrieval of approximate kNN using the Hamming distance. They have many advantages, including sublinear search complexity, scalability, and the ability to handle high-dimensional data [19]. However, there is a trade-off between accuracy and speed, as hashing-based methods may not always guarantee that all nearby vectors will fall into the same or nearby buckets, which can affect the accuracy of the search [21].

In approximate kNN search, graph-based methods represent data points as nodes in a graph and establish edges based on their distances [14]. These methods utilize graph traversal techniques to find approximate kNN. The advantages of graph-based methods include handling both sparse and dense data, flexibility in defining proximity relationships, and good performance for high-dimensional data [22, 23]. They can also support incremental updates to the graph. However, constructing and maintaining the graph can be computationally expensive, and the search complexity can be higher than with tree-based methods.

This paper provides a method for the construction of the approximate kNN graph, which is customized to generate a high-quality approximate MST. Our method is able to derive an approximate MST that is close to the exact MST with a lower computational cost, thereby enhancing the overall performance and efficiency of clustering algorithms, such as Hierarchical Density-Based Spatial Clustering of Applications with Noise (HDBSCAN) [24], for which MST generation is a crucial step. Two implementations of our algorithm are presented . The first implementation stands out for its memory efficiency. The second implementation maintains a balance between memory efficiency and execution time. It stores a reduced number of the computed distances during the optimization process, thereby ensuring a shorter execution time than the memory-efficient version and a lower memory consumption than the brute force method.

The rest of the paper is structured as follows. In Section 2, the relevant solutions from the literature are presented. The proposed method is detailed in Section 3. Results and discussions are introduced in Section 4. Finally, a

conclusion of the paper is given in Section 5.

2 Existing MST generation methods

In graph theory, an MST is a subgraph of a given undirected graph that connects all the data points in a data set while minimizing the total sum of edge weights. The construction of an MST must follow the cut and cycle properties. The cut property of MSTs states that the lightest edge crossing any cut must be part of the MST, while the cycle property dictates that the heaviest edge in any cycle whose removal results in lighter edges cannot be in the MST.

MSTs are employed in clustering and data analysis algorithms to identify patterns and relationships within large data sets. Constructing an MST on data points makes it possible to identify clusters and hierarchies, enabling efficient data exploration and visualization. MSTs can indeed serve as the primary and exclusive technique for unsupervised data clustering. For instance, MST plays a crucial role as a fundamental step within the HDBSCAN clustering framework. An MST can be generated using exact or approximate methods.

2.1 Exact methods

Constructing an MST from a data set of n points in a d-dimensional space consists of two steps. First, a weighted undirected full graph is constructed from the data set, where each node represents a data point, and edges between nodes convey the similarity between the points. Second, the MST is constructed from this full graph. The process of constructing an MST can be achieved using various algorithms, such as Prim's algorithm [25], Boruvka's algorithm [26], or Kruskal's algorithm. Here is a general outline of the procedure for building an MST from a full graph using Kruskal's algorithms [25, 27]:

- Create a set for each node, initially containing only that node.
- Create a list of all edges in the graph.
- Sort the list of edges in non-decreasing order of their weights.
- Starting from the smallest edge.
- Select the next edge from the sorted list. On the condition of not forming a cycle, add this edge to the current MST and merge the sets of the two endpoints of the edge.
- Repeat the last step until the MST contains (n - 1) edges, where n is the number of nodes in the full graph (identical to the size of the data set).

The process of building the MST from the full graph has a time complexity of $O(n^2)$. This quadratic complexity poses challenges when applying exact MST algorithms to large data sets. Consequently, handling such data sets becomes difficult due to the significant time and memory requirements. Therefore, more efficient algorithms or specialized techniques are pursued to address this limitation and enable cluster analysis on very large data sets at the price of computing an approximate MST with a higher weight than the MST, which we refer to exact MST in the sequel for distinction.

2.2 Approximate methods

Many algorithms initiate the construction of the approximate MST by generating a graph with a reduced number of edges such as kNN graphs. When the $O(n^2)$ time complexity of exact kNN graph generation is infeasible due to the large size of the data set, efficiently generated approximate kNN graphs can be used. Many of the efficient approximate kNN graph generation procedures are based on spatial trees, since the cost of their construction is low. For example, the time complexity of constructing a KD-tree from a d dimensional data set is $O(d \cdot n \log n)$ and to generate an approximate kNN graph from a balanced KD-tree is $O(n \cdot k \log n)$.

The price of this reduced computational cost is the reduced accuracy. The sum of the weights of the generated approximate kNN graphs is higher than that of the exact kNN graph, but the structure of the approximate kNN graphs is usually quite similar to the exact kNN graph, thus the classification based on approximate kNN graphs is similar to the one based on an exact kNN graph.

One of the most efficient approximate MST generation methods using KD-tree is the Well-Separated Pair Decomposition (WSPD) [28, 29, 30] based method proposed in [27]. In computational geometry, a WSPD of a data set X in d-dimensional space is a sequence of pairs of sets (A_i,B_i). Each pair maintains a significant distance between subsets, and for any two distinct points p,$q \in X$, there is exactly one pair that separates them, placing p in A_i and q in B_i. WSPD helps organize points for efficient geometric algorithms by ensuring separation and clear pairwise relationships. A and B are considered ε-well-separated if

$$\varepsilon \cdot distance(A_i, B_i) \geq \max(diameter(A_i), diameter(B_i))$$

where the distance of set A_i and B_i is defined as the minimum distance between the bounding boxes of set A_i and B_i, the diameter of set A_i is the length of the diagonal of the bounding box of the A_i set, and ε is a hyperparameter.

In [27], Wang et al. propose a fast approximate MST construction approach utilizing the WSPD technique. Their proposed method, is composed of three steps:

1) building a spatial tree (they use KD-tree),
2) building a WSPD based on the spatial tree,
3) connecting pairs of WSPD using Bichromatic Closest Pair (BCCP) [28], which involves finding the pair of points (x_i, x_j) that have the smallest distance between set A and set B where $x_i \in A$ and $x_j \in B$.

In [27], they present a parallel version of this approach as a step in the parallel HDBSCAN clustering process.

The graph obtained through WSPD has only $O(n\ log\ n)$ edges and is computed in $O(dn\ log\ n\ + \varepsilon^d d^{d/2} n)$ time [31]. However, steps 1)-3) are considerably slower than the method proposed in this paper because WSPD relies on spatial trees, which scale poorly with the dimensionality of the data [31]. Additionally, adapting this algorithm for an incremental extension of the data set is non-trivial.

In this work, the WSPD-based method proposed in [27] is considered to be the most efficient among the existing approximate MST generation methods [32], and we compare our proposed method with it in the numerical section.

In [33], an enhanced and efficient implementation of HDBSCAN is introduced. The authors employ a technique referred to as Hierarchical Navigable Small World (HNSW) to achieve this improvement. While HNSW is efficient in approximate kNN search, the construction of HNSW is less efficient [14]. The approach also demands considerable memory resources [34]. Furthermore, the method may approximate the MST by a spanning forest initially, and the creation of a single approximate MST from this forest is improperly documented in [33].

In [35], Wang et al. present a fast approximate MST-based clustering algorithm that employs a divide-and-conquer strategy. This method efficiently identifies long edges in the initial stages to decrease the required distance computations by utilizing the cut and cycle properties of MSTs. However, the worst-case complexity of the algorithm in [35] remains $O(n^2)$, suggesting that it could face performance challenges when dealing with large data sets.

A K-means-based fast approximate MST computation was proposed in [36]. The solution consists of the following stages:

- $\lfloor \sqrt{n} \rfloor$ partition
  - dividing the data set into $\lfloor \sqrt{n} \rfloor$ clusters (using the K-means method)
  - computing the exact MST for each cluster
  - connecting these MSTs into an approximate MST of the entire data set
- $\lfloor \sqrt{n} \rfloor - 1$ partition
  - repeating the same step with $\lfloor \sqrt{n} \rfloor - 1$ clusters
- the two approximate MSTs are combined into a single approximate MST.

The expected time complexity of the procedure is $O(n^{1.5})$. However, it is crucial to acknowledge that the actual execution time might vary highly. This variation is due to the strong dependence on the distribution of partitions generated from the K-means algorithm. The computational complexity of the step of computing the exact MST for each cluster depends on the size of the largest cluster. A strongly unbalanced cluster structure might result in significant running time.

Jothi et al. propose a recursive approach for clustering data in [18]. The data set is recursively divided into tree of clusters by repeatedly splitting the data set into two sets until the partition size meets a cluster size limit. This process creates a tree where the leaf nodes represent the final partitions of the data set. A full subgraph is then generated for each partition. The following steps involve identifying neighbor partitions and boundary points. Finally, the subgraphs are connected, and an approximate MST is constructed from them, achieving an expected time complexity of $O(n^{1.5}\ log\ n)$ [37]. However, the accuracy of the final approximate MST heavily depends on the quality of the inter-set edges calculations used to identify neighboring partitions. This dependency may lead to less accurate approximate MSTs, especially when dealing with complex, high-dimensional data sets [38].

3 The Proposed Method

Our goal in this paper is to construct an approximate MST efficiently, which is similar to the exact MST, from a data set denoted as X, where $X = \{x_1, x_2, \ldots, x_n\}$, comprises n data points in a d-dimensional space.

Our proposed method is composed of the following steps:

- Random k-neighbor graph generation.
- Optimization of the random k-neighbor graph to obtain an approximate kNN graph
- Generation of an approximate MST from the approximate kNN graph
- Optimization of the approximate MST.

3.1 Random k-neighbor graph generation

The initial step in constructing an approximate kNN graph is creating a random directed graph with $(n \cdot k)$ edges, such that each node has k neighbors. The weight associated with the edges between node x_i and x_j is the distance between them. This way the weight of edge (x_i, x_j) and edge (x_j, x_i) are identical. Algorithm 1 generates a random directed graph by assigning k random data points to each node as neighbors. Each node is represented by a unique key in the graph dictionary, and the corresponding value is a list of k indices representing its neighbors. We construct a random graph of this nature with a time complexity of $O(n \cdot k)$.

Algorithm 1: Random k-neighbor Graph

Data: $Nodes$, k
Result: $Neighbors$: random k-neighbor list
$Neighbors \leftarrow \{\}$
for x_i *in* $Nodes$ **do**
  $Neighbors(x_i) \leftarrow$ list of k random nodes from $Nodes$
end

3.2 Optimizing the random k-neighbor graph

The optimization of the random graph generated by Algorithm 1 aims to eliminate long edges based on the procedure proposed in [39]. The elementary step of the procedure is as follows. If x_j is a neighbor of x_i and x_z is a neighbor of x_j and $distance(x_i, x_j)$ is greater than $distance(x_i, x_z)$, then x_j is replaced by x_z in the neighbors list of x_i. This elementary step is performed for all nodes and all neighbors of neighbors according to the pseudo-code in Algorithm 2. Since the replaced list of neighbors changes the structure of the graph, further iterations of the same steps might further refine the list of neighbors.

The random k-neighbor graph optimization process, according to Algorithm 2, can be continued until either convergence is achieved (where no further neighbors of neighbors are found to be closer to a given node) or a predetermined number of iterations is reached. We introduce two versions of the random k-neighbor graph optimization process with different iteration policies based on hyperparameters Δ and δ ($\Delta >> \delta$) as follows:

Algorithm 2: Optimizing the random graph

```
Data: Random k neighbor list, Data set X
Result: Optimized k neighbor list, Edges, Converged
Edges ← []
Converged ← True
for x_i ∈ Nodes do
    for x_j ∈ Neighbors(x_i) do
        D ← distance(x_i, x_j)
        D̂ ← D
        for x̂_z ∈ Neighbors(x_j) do
            if x̂_z ≠ x_i then
                D_temp ← distance(x_i, x̂_z)
                if D_temp < D̂ and x̂_z ∉ Neighbors(x_i) then
                    D̂ ← D_temp
                    x_z ← x̂_z
                end
            end
        end
        if D̂ < D then
            Neighbors(x_i) ← Neighbors(x_i) \ x_j        ▷ Remove far neighbor
            Neighbors(x_i) ← Neighbors(x_i) ∪ x_z        ▷ Add closer one
            Edges ← Edges ∪ (x_i, x_z, D̂)
            Converged ← False
        else
            Edges ∪ (x_i, x_j, D)
        end
        ▷ We denote D as the Euclidean distance between a given node and its corresponding neighbor.
        Similarly, D̂ represents the optimal Euclidean distance between the node and a neighbor of a
        neighbor. Here, x_z refers to the neighbor of a neighbor, while D_temp refers to the current Euclidean
        distance between the node and a neighbor of a neighbor.
    end
end
```

- *Long k-neighbor graph optimization* (Algorithm 3): In this procedure, the random k-neighbor graph is optimized according to Algorithm 2 until either converge or reaching Δ iterations. At this point, an undirected graph is generated from the directed k-neighbor graph such that one of the edges is dropped in case of a node pair connected with two opposite directed edges. This way, the number of undirected edges is between $n \cdot k/2$ and $n \cdot k$ and the weights of the edges are the distances between the corresponding nodes. The approximate MST is obtained as the exact MST of this undirected graph. (The obtained MST is suboptimal because only a limited number of edges of the full graph is used to compute it.)

- *Short k-neighbor graph optimization* (Algorithm 4): In this procedure, the random k-neighbor graph is optimized by Algorithm 2 only δ times (if it does not converge before), and after that an undirected graph and its MST of $n-1$ edges is generated as in the previous case. In contrast to the *long k-neighbor graph optimization* method, the obtained approximate MST is further optimized in this procedure. First, the undirected approximate MST is transformed to a directed graph of $2n-2$ edges such that the undirected edges substituted with directed edges in both directions and Algorithm 2 is applied again until convergence or reaching Δ iterations. The final step of the procedure is to transform the obtained directed graph to undirected again and compute its MST.

The memory consumption of both methods can be adjusted by the number of edge distances that are stored during the computations. If the computed edge distances are stored in the memory, the time of the consecutive iterations reduces because fewer distances have to be computed in one iteration. On the other hand, the memory consumption continuously increases during the procedure. In the *long k-neighbor graph optimization* method at most $\Delta \cdot n \cdot k^2$

distances need to be stored and in the *short k-neighbor graph optimization* method it depends on the structure of the data set. In our experiments it was always less than the memory consumption of the *long k-neighbor graph optimization* method. We note that the second phase of the *short k-neighbor graph optimization* method, where the number of edges is $2n-2$, converges rather fast in practice, thus its memory consumption and running time are much lower than the ones of the *long k-neighbor graph optimization* method.

If the edge distances are not stored during the computation, then the memory consumption is negligible ($O(n \cdot k)$) since we only store the indexes of k neighbors of each node, however, the computation time increases with the repeated computation of the edge distances.

Essentially, the approach of the *long k-neighbor graph optimization* method resembles the approach used in [39], while the approach of the *short k-neighbor graph optimization* method was not considered before.

Algorithm 3: Long k-neighbor graph optimization

Data: X, k, Δ
Result: MST
$Random\_k\_neighbor \leftarrow Algorithm\_1(Nodes, k)$
$Optimized\_k\_neighbor \leftarrow Random\_k\_neighbor$
$Epoch \leftarrow 1$
$Converged \leftarrow False$
while *Not Converged and Epoch* $< \Delta$ **do**
  $Optimized\_k\_neighbor, Edges, Converged \leftarrow Algorithm\_2(Optimized\_k\_neighbor, X)$
  $Epoch \leftarrow Epoch + 1$
end
$MST \leftarrow Exact\_mst(n, Edges)$

Algorithm 4: Short k-neighbor graph optimization

Data: X, k, Δ, δ
Result: MST
$Random\_k\_neighbor \leftarrow Algorithm\_1(Nodes, k)$
$Optimized\_k\_neighbor \leftarrow Random\_k\_neighbor$
Epoch $\leftarrow 1$
$Converged \leftarrow False$
while *Not Converged and Epoch* $< \delta$ **do**
  $Optimized\_k\_neighbor, Edges, Converged \leftarrow Algorithm\_2(Optimized\_k\_neighbor, X)$
  $Epoch \leftarrow Epoch + 1$
end
$MST \leftarrow Exact\_mst(n, Edges)$
while *Not Converged and Epoch* $< \Delta$ **do**
  $MST, Edges, Converged \leftarrow Algorithm\_2(MST, X)$
  $MST \leftarrow Exact\_mst(n, Edges)$
  $Epoch \leftarrow Epoch + 1$
end

4 Application examples

4.1 Applied data sets

To demonstrate the effectiveness of the proposed method, we conducted a series of experiments on various data sets. Specifically, we utilized 13 different data sets as shown in Table 1. Speech, MNIST, and Shuttle are from Multi-dimensional point data sets of the Outlier Detection Data Sets[1]. CelebA was introduced in Ref. [40] and can be

[1] https://odds.cs.stonybrook.edu/#table1

Table 1: Data sets of different sizes and dimensions

Data Set	**type**	**dimensions** (d)	**Size** (n)
Speech	real	400	3,686
Miss America		16	6,480
MNIST		100	7,603
Shape		544	28,775
House		3	34,112
Shuttle		9	49,097
Audio		192	54,387
Europe		2	169,308
Celeba		39	202,599
Corel		14	662,317
Unbalanced	synthetic	2	6,500
Birch1		2	10^5
Make_moons		2	40
Make_blobs		2 - 1000	10^3-10^6

accessed online[2]. Miss America, House, Europe, Unbalanced, and Birch can be found on the web[3]. The first three of this group are real-world data, and the last two are synthetic data sets. Corel, Shape, and Audio were used by [39, 41] and are available online[4]. We used the *make_moons* library from *Sklearn data sets*[5] to generate the Make_moons data set. Finally, we used the *make_blobs* library from *Sklearn data sets*[6] to generate data sets of different sizes and dimensions.

4.2 Comparison of short and long k-neighbor graph optimization methods

We compared the run time and the accuracy of the *short and long k-neighbor graph optimization* methods using the data sets in Table 1. with hyperparameters $k = 20$, $\delta = 8$, $\Delta = \infty$ in each cases, starting from independently sampled initial random k-neighbor graphs. In order to demonstrate the number of iteration cycles required until convergence we set $\Delta = \infty$, which results that Algorithms 3 and phase 2 of 4 run until the graph can not be improved any more with the iteration cycle of Algorithm 2.

The obtained results, in Table 2, demonstrate that the *short k-neighbor graph optimization* method provides more accurate approximate MST and better execution time. Its lower execution time is due to the fact that it converges in fewer iterations because Algorithm 4 perform at most δ iterations of optimizing the random k-neighbor graph and then generate a directed graph with fewer edges whose optimization converges quickly. The intuitive explanation for the improvement in the total weight of obtained approximate MST is that after we optimized the random k-neighbor graph, during the optimization of the approximate MST, we add the reverse edge of every existing edge in the approximate MST to get an undirected graph, which helps avoiding the convergence to a local optima and enhances the accuracy of the final approximate MST.

The only data set for which optimization of the random k-neighbor graph converges within less than δ iterations is the Speech data set. In that case the independent sampling of the initial random k-neighbor graph results in the differences in the number of required iterations for Algorithm 3 and 4.

4.3 Demonstration of the behavior of Algorithm 4

Figure 1 presents an overview of the approximate MST generation process facilitated by Algorithms 4 for a data set generated by the *make_moons* algorithm from *Scikit-learn*, comprising only 40 data points, to enhance visual clarity. In the initial step, Algorithms 4 generates a random k-neighbor graph with $n = 40$ vertices and $n \cdot k = 160$ edges,

[2] https://paperswithcode.com/dataset/celeba
[3] https://cs.joensuu.fi/sipu/datasets/
[4] https://code.google.com/archive/p/nndes/downloads
[5] https://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_moons.html
[6] https://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_blobs.html

Table 2: Comparison of Algorithms 3 and 4 using different data sets with $k = 20$, $\delta = 8$, $\Delta = \infty$ (optimization until convergence).

Data Set (d_n)	Algorithm	# Iterations		Time	Weight	Relative error %
		phase 1	phase 2			
Shuttle	Alg. 3	42	-	490	424288	30.328
	Alg. 4	8	15	**93**	**325554**	reference
Mnist	Alg. 3	14	-	27.8	2769425	0.034
	Alg. 4	8	3	**14.8**	**2768488**	reference
Speech	Alg. 3	7	-	11.8	70772	0.082
	Alg. 4	6	1	11.8	**70714**	reference
Celeba	Alg. 3	13	-	801	191995	3.596
	Alg. 4	8	4	**459**	**185331**	reference
Shape	Alg. 3	25	-	236	4237.55	5.949
	Alg. 4	8	7	**80**	**3999**	reference
Audio	Alg. 3	17	-	232	36797	1.519
	Alg. 4	8	6	**109**	**36246**	reference
Unbalance	Alg. 3	27	-	36.3	3108182	1.62
	Alg. 4	8	3	**10.7**	**3058632**	reference
Make blobs (2_100000)	Alg. 3	88	-	2022	2455	4.437
	Alg. 4	8	11	**180**	**2350**	reference
Make blobs (1000_50000)	Alg. 3	12	-	333	2076043	0.001
	Alg. 4	8	2	**237**	**2076041**	reference

assuming $k = 4$, which is depicted in Figure 1a. (A higher value for k would result in a denser graph and potentially better approximate MST, but with poorer graphical visibility). In this example, we set $\delta = 2$ to executed only two iteration of random k-neighbor graph optimization due to the relatively small number of edges. The result of the first iteration is depicted in Figure 1b. One can observe that the first iteration already significantly decreased the number of long edges in the graph, which results in the visually less dense impression of the graph in Figure 1b.

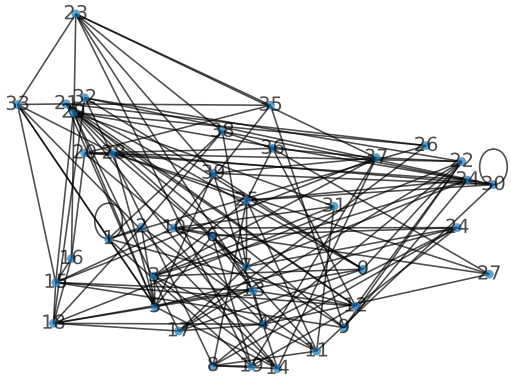
The primary approximate MST, as a result of the first phase of Algorithm 4, is obtained from the second iteration of the k-neighbor graph optimization and it is depicted in Figure 1c. Figure 1d depict the result of the first optimization cycles of the second phase of Algorithm 4. This phase converges in 2 cycles, and no further optimization is possible for this example. The MST generated from the optimized graph is provided in 1e. For comparison purposes, Figure 1f presents the exact MST obtained using the brute force method. Most of the approximate and the exact MST are identical in Figure 1e and 1f except some minor differences indicated by circles.

As another example, Figure 2 presents the results of Algorithm 4 using a 2D synthetic unbalanced data set generated by the Make blobs procedure containing 6500 data points, which are distributed among eight distinct clusters. It is worth noting that the size of this data set is relatively modest, and our primary objective is to demonstrate the efficacy of our algorithm in handling unbalanced data sets. Similar to the previous example, Figure 2 suggests that most of the approximate and the exact MST are identical with some minor exceptions, and the weights of the approximate and the exact MST are close.

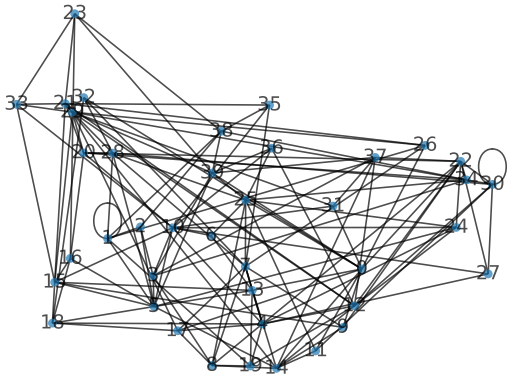
4.4 Comparison of Algorithm 4 and the WSPD based method

To the best of the authors' knowledge, there is currently no available implementation for obtaining an MST using sequential WSPD. While an implementation for sequential WSPD exists[7], its output consists of a set of ε-separated pairs, as detailed in [31]. These pairs must be connected by an edge between the closest points for each pair using BCCP to obtain a connected graph from which an MST can be generated. We will utilize the implementation of [31] to illustrate that WSPD has a strong dependence on the dimension of the data set, because WSPD suffer in higher-dimensional spaces as the number of pairwise comparisons that need to be stored increases exponentially, leading to

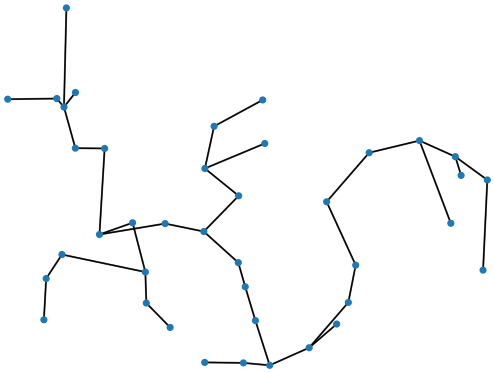
[7] https://github.com/dmatijev/wspd_pip



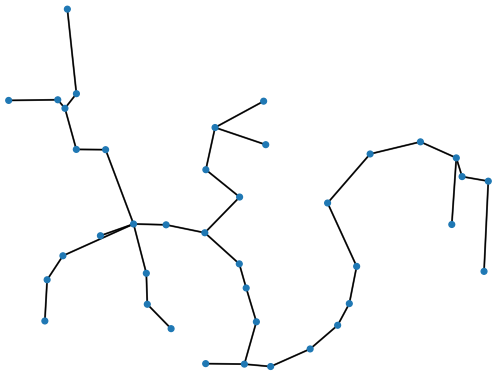
(a) Random k-neighbor graph, 40 data points and $k = 4$



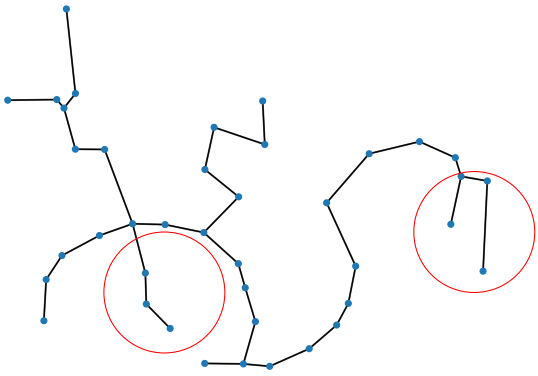
(b) k-neighbor graph after the first iteration of the optimization process



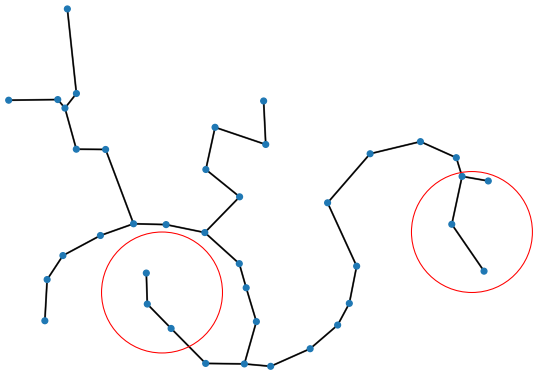
(c) Approximate MST generated from the k-neighbor graph after the second iteration of the optimization process



(d) The graph after the first iteration of the approximate MST

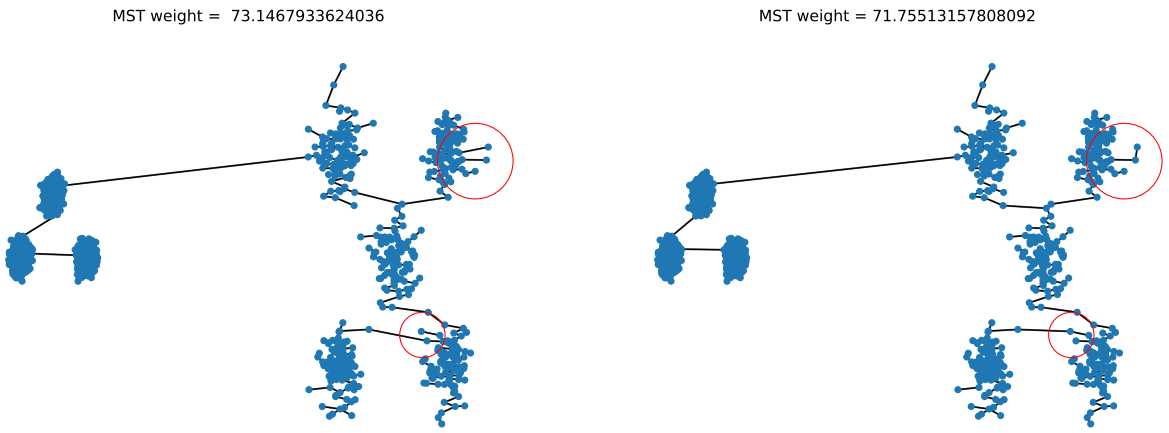


(e) The approximate MST



(f) MST obtained by brute force method

Figure 1: Process of generating approximate MST based on Algoritm 4 and the associated MST. The circles indicate the differences of the obtained approximate MST compared to the MST.



(a) Approximate MST obtained by Algorithm 4 with $k = 20$

(b) MST obtained by brute force

Figure 2: Unbalanced $2d$ synthetic data set [42] with $n = 6500$ data points and 8 Gaussian clusters. Some differences between the approximate and the exact MSTs are marked with circles.

Table 3: Execution time and memory consumption of obtaining ε-separated pairs using WSPD for different data sets from Table 1. Memory error occurs after consuming all available memory (64GB)

Data set	n	d	$\varepsilon = 2$		$\varepsilon = 8$	
			Time(s)	Memory(GB)	Time(s)	Memory(GB)
Shuttle	49097	9	13	6	96	35
Mnist	7603	100	21	7.5	28	9
Speech	3686	400	9	2	9	2
Celeba	202599	39	memory error			
Shape	28775	544	192	50	memory error	
Audio	54387	192	memory error			
Corel	662317	14	memory error			
Corel (subset)	30000	14	memory error			
Birch1	100000	2	3	1.7	15.3	5
Europe	169308	2	9	13	30	18

higher memory requirements [31]. Therefore, the dimension of the data can significantly impact memory usage. Table 3 illustrates the execution time and memory usage of obtaining ε-separated pairs using the implementation of WSPD presented in [31]. We can see that WSPD is memory hungry when the dimension of the data set and ε are high. We note that the execution time in Table 3 is the required time to obtain ε-separated pairs, and the time to connect the ε-separated pairs and to generate the approximate MST is not considered.

We implemented a sequential version of the WSPD-based MST method in Python. Our WSPD implementation is not as fast as the one presented in[8], however, it gives very accurate results in terms of relative error, where

$$relative\ error = \frac{approximate\ MST\ weight - MST\ weight}{MST\ weight}$$

and the weight of an undirected graph is the sum of the weights of its edges. Table 4 presents the results of an experiment performed on a $2d$ synthetic data set (*make_blobs*) comprising between 10,000 and 100,000 data points. The table displays the running time, the weight of the approximate MST obtained by Algorithm 4, and our sequential implementation of the WSPD-based MST method. The experiment demonstrates that Algorithm 4 achieves remarkable computational time efficiency compared to the sequential WSPD implementation. It provides a speed up between 3.4

[8] https://github.com/dmatijev/wspd_pip

and 17, with only a low relative error between 3.5% and 6%. The memory error of the brute force method occurs when the size n^2 weight matrix exceeds the available memory (64GB).

The results in Table 4 verifies the time efficiency of Algorithm 4, mainly when dealing with large data sets, as it outperforms the WSPD implementation significantly in terms of run time while maintaining reasonably accurate approximate MST construction.

Table 4: Comparison with the sequential WSPD and the brute force approach using $2d$ synthetic data sets of different sizes.

Data Set (n)	Algorithm	time(s)	MST weight	Relative error %
10,000	Brute force	79	632.16	reference
	WSPD	126	632.16	0
	Alg. 4	**33**	655.54	3.56
20,000	Brute force	371	890.05	reference
	WSPD	492	890.05	0
	Alg. 4	**75**	946.19	5.93
40,000	Brute force	memory error		
	WSPD	1993	1265.61	reference
	Alg. 4	**146**	1345	5.90
80,000	Brute force	memory error		
	WSPD	8174	1796.03	reference
	Alg. 4	**445**	1865.72	3.73
100,000	Brute force	memory error		
	WSPD	10915	2010.46	reference
	Alg. 4	**640**	2088.77	3.74

4.5 Empirical time complexity

To investigate the impact of the size of the data set on the running time of Algorithm 4, we evaluated both real-world and synthetic data sets of varying sizes. Figure 3 shows the results of our experiments. A least squares fitting of the obtained data points suggests that the empirical computational time of Algorithm 4 is around $O(n^{1.07})$.

The performance of the proposed method on real-world data sets of different sizes and dimensions (all the real-world data sets in Table 1) follows the same trend as for the synthetic data sets. Additionally, we used subsets of the *Corel* data set to ensure the same structure but different sizes.

4.6 The impact of data dimension

In our approach the dimension of the data set play role only in the computation of the edge weights of the graph, i.e., the distance between data points. As a result, our approach is only slightly affected by the dimension of the data set compared to the algorithms that rely on spatial trees or partitioning the space into subsets.

To validate this assumption, we performed multiple experiments using data sets of varying dimensions to investigate the effect of dimension on the performance of our approach. The experiments were carried out in two distinct settings: one where we stored the calculated distances in the memory throughout the optimization process and another where we did not.

Figure 4 plots the result of the experiments with and without storing of the calculated distances using the synthetic and real-world data sets from Table 1. As expected the execution time is higher when the calculated distances are not stored in the memory, but are recalculated each time they are needed. The trend of the execution time as a function of the data set dimension suggests that the function which computes the distance has a significant overhead in our python implementation, which dominates the running time for dimensions less than 10^3. That is why the running time is seemingly independent of the dimension for dimensions less than 10^3 and starts increasing only after that limit.

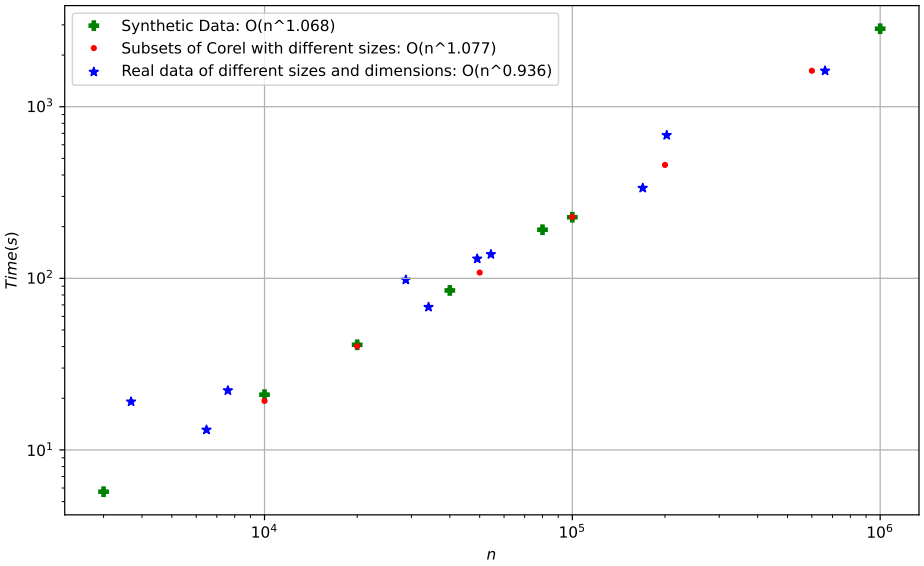
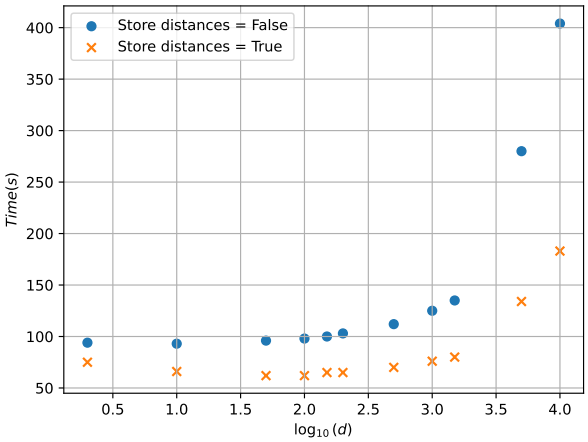
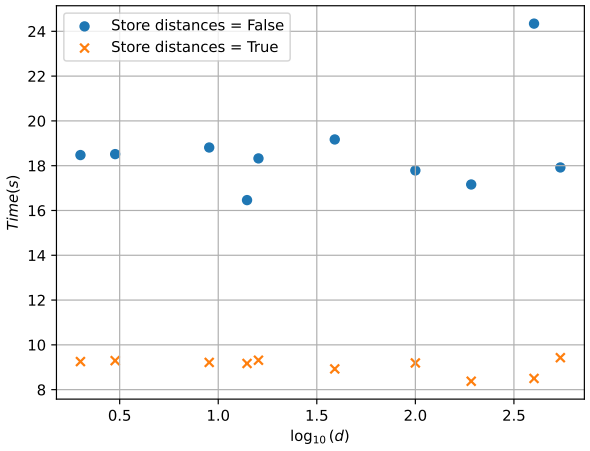


Figure 3: Execution time (in seconds) using synthetic and real-world data sets where the dimension of data is shown in Table 1



(a) Synthetic Make blobs data sets ($n = 20000$)



(b) Sub sets (first $n = 3500$ data points) of all real-world data sets from Table 1

Figure 4: Impact of data dimension on the computational time with synthetic and real-world data sets ($k = 20$).

Table 5: The effect of dimensions on the performance with $k = 25$.

Data Set (d;n)	Algorithm	time(s)	MST weight	Relative error %
2;20,000	Brute force	351	777.16	reference
	Alg. 4	**96**	798.8	2.70
3;20,000	Brute force	348	3072.22	reference
	Alg. 4	**98**	3172.54	3.16
4;20,000	Brute force	365	2573.16	reference
	Alg. 4	**92**	2645.35	2.72
10;20,000	Brute force	342	30853.21	reference
	Alg. 4	**95**	31835.75	3.08
20;20,000	Brute force	338	65014.7	reference
	Alg. 4	**90**	66473.51	2.19
50;20,000	Brute force	360	136885	reference
	Alg. 4	**88**	138682.77	1.29
100;20,000	Brute force	379	219377.16	reference
	Alg. 4	**88**	221160.83	0.80
1000;20,000	Brute force	407	830245.81	reference
	Alg. 4	**97**	831499.29	0.5

5 Conclusion

Large data sets are often represented as a graph with as many nodes as the number of data points, and efficient graph procedures are needed to process such large graphs. One of the essential graph processing steps is the computation of the MST, which plays a role, e.g., in clustering, anomaly detection, etc.

Since exact methods are infeasible for large data sets, the paper presents an algorithm for approximate MST constructing accurately and computationally efficiently. The algorithm starts by creating a random k-neighbor graph and then optimizes it by crawling toward the optimal neighbors of each node to obtain an approximate kNN graph. Using this graph, the algorithm calculates an initial MST and optimizes it using the same crawling technique. Numerical experiments indicate the properties and the effectiveness of the proposed method.

Although the current paper primarily focuses on the static, sequential implementation of the proposed algorithm, it can be enhanced with dynamic parallel implementation, which makes our algorithm compelling for a wide range of MST-based clustering tasks.

References

[1] C. T. Zahn, "Graph-theoretical methods for detecting and describing gestalt clusters," *IEEE Transactions on computers*, vol. 100, no. 1, pp. 68–86, 1971.

[2] D. Xu and Y. Tian, "A comprehensive survey of clustering algorithms," *Annals of Data Science*, vol. 2, pp. 165–193, 2015.

[3] R. Jothi, S. K. Mohanty, and A. Ojha, "Functional grouping of similar genes using eigenanalysis on minimum spanning tree based neighborhood graph," *Computers in biology and medicine*, vol. 71, pp. 135–148, 2016.

[4] P. Juszczak, D. M. Tax, E. Pe, R. P. Duin, *et al.*, "Minimum spanning tree based one-class classifier," *Neurocomputing*, vol. 72, no. 7-9, pp. 1859–1869, 2009.

[5] C. Zhong, D. Miao, and R. Wang, "A graph-theoretical clustering method based on two rounds of minimum spanning trees," *Pattern Recognition*, vol. 43, no. 3, pp. 752–766, 2010.

[6] C. Zhong, D. Miao, and P. Fränti, "Minimum spanning tree based split-and-merge: A hierarchical clustering method," *Information Sciences*, vol. 181, no. 16, pp. 3397–3410, 2011.

[7] X. Wang, X. L. Wang, C. Chen, and D. M. Wilkes, "Enhancing minimum spanning tree-based clustering by removing density-based outliers," *Digital Signal Processing*, vol. 23, no. 5, pp. 1523–1538, 2013.

[8] O. Grygorash, Y. Zhou, and Z. Jorgensen, "Minimum spanning tree based clustering algorithms," in *2006 18th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'06)*, pp. 73–81, IEEE, 2006.

[9] D. Cheriton and R. E. Tarjan, "Finding minimum spanning trees," *SIAM journal on computing*, vol. 5, no. 4, pp. 724–742, 1976.

[10] C. Stam, P. Tewarie, E. Van Dellen, E. Van Straaten, A. Hillebrand, and P. Van Mieghem, "The trees and the forest: characterization of complex brain networks with minimum spanning trees," *International Journal of Psychophysiology*, vol. 92, no. 3, pp. 129–138, 2014.

[11] M. Sha'Abani, N. Fuad, N. Jamal, and M. Ismail, "knn and svm classification for eeg: a review," in *InECCE2019: Proceedings of the 5th International Conference on Electrical, Control & Computer Engineering, Kuantan, Pahang, Malaysia, 29th July 2019*, pp. 555–565, Springer, 2020.

[12] K. Taunk, S. De, S. Verma, and A. Swetapadma, "A brief review of nearest neighbor algorithm for learning and classification," in *2019 international conference on intelligent computing and control systems (ICCS)*, pp. 1255–1260, IEEE, 2019.

[13] S. Dhanabal and S. Chandramathi, "A review of various k-nearest neighbor query processing techniques," *International Journal of Computer Applications*, vol. 31, no. 7, pp. 14–22, 2011.

[14] M. Wang, X. Xu, Q. Yue, and Y. Wang, "A comprehensive survey and experimental comparison of graph-based approximate nearest neighbor search," *arXiv preprint arXiv:2101.12631*, 2021.

[15] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Communications of the ACM*, vol. 18, no. 9, pp. 509–517, 1975.

[16] M. Dolatshah, A. Hadian, and B. Minaei-Bidgoli, "Ball*-tree: Efficient spatial indexing for constrained nearest-neighbor search in metric spaces," *arXiv preprint arXiv:1511.00628*, 2015.

[17] S. M. Omohundro, *Five balltree construction algorithms*. International Computer Science Institute Berkeley, 1989.

[18] R. Jothi, S. K. Mohanty, and A. Ojha, "Fast approximate minimum spanning tree based clustering algorithm," *Neurocomputing*, vol. 272, pp. 542–557, 2018.

[19] J. Wang, W. Liu, S. Kumar, and S.-F. Chang, "Learning to hash for indexing big data—a survey," *Proceedings of the IEEE*, vol. 104, no. 1, pp. 34–57, 2015.

[20] O. Jafari, P. Maurya, P. Nagarkar, K. M. Islam, and C. Crushev, "A survey on locality sensitive hashing algorithms and their applications," *arXiv preprint arXiv:2102.08942*, 2021.

[21] Z. Jin, D. Zhang, Y. Hu, S. Lin, D. Cai, and X. He, "Fast and accurate hashing via iterative nearest neighbors expansion," *IEEE transactions on cybernetics*, vol. 44, no. 11, pp. 2167–2177, 2014.

[22] L. C. Shimomura, R. S. Oyamada, M. R. Vieira, and D. S. Kaster, "A survey on graph-based methods for similarity searches in metric spaces," *Information Systems*, vol. 95, p. 101507, 2021.

[23] R. Paredes and E. Chávez, "Using the k-nearest neighbor graph for proximity searching in metric spaces," in *String Processing and Information Retrieval: 12th International Conference, SPIRE 2005, Buenos Aires, Argentina, November 2-4, 2005. Proceedings 12*, pp. 127–138, Springer, 2005.

[24] L. McInnes, J. Healy, and S. Astels, "hdbscan: Hierarchical density based clustering.," *J. Open Source Softw.*, vol. 2, no. 11, p. 205, 2017.

[25] A. Kershenbaum and R. Van Slyke, "Computing minimum spanning trees efficiently," in *Proceedings of the ACM annual conference-Volume 1*, pp. 518–527, 1972.

[26] S. Pettie and V. Ramachandran, "An optimal minimum spanning tree algorithm," *Journal of the ACM (JACM)*, vol. 49, no. 1, pp. 16–34, 2002.

[27] Y. Wang, S. Yu, Y. Gu, and J. Shun, "Fast parallel algorithms for euclidean minimum spanning tree and hierarchical spatial clustering," in *Proceedings of the 2021 international conference on management of data*, pp. 1982–1995, 2021.

[28] G. Narasimhan and M. Zachariasen, "Geometric minimum spanning trees via well-separated pair decompositions," *Journal of Experimental Algorithmics (JEA)*, vol. 6, pp. 6–es, 2001.

[29] P. B. Callahan and S. R. Kosaraju, "A decomposition of multidimensional point sets with applications to k-nearest-neighbors and n-body potential fields," *Journal of the ACM (JACM)*, vol. 42, no. 1, pp. 67–90, 1995.

[30] T. M. Chan, "Well-separated pair decomposition in linear time?," *Information Processing Letters*, vol. 107, no. 5, pp. 138–141, 2008.

[31] D. Matijević, "Well-separated pair decompositions for high-dimensional datasets," *Algorithms*, vol. 16, no. 5, p. 254, 2023.

[32] A. Prokopenko, P. Sao, and D. Lebrun-Grandie, "A single-tree algorithm to compute the euclidean minimum spanning tree on gpus," in *Proceedings of the 51st International Conference on Parallel Processing*, pp. 1–10, 2022.

[33] M. Dell'Amico, "Fishdbc: Flexible, incremental, scalable, hierarchical density-based clustering for arbitrary data and distance," *arXiv preprint arXiv:1910.07283*, 2019.

[34] Y. A. Malkov and D. A. Yashunin, "Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs," *IEEE transactions on pattern analysis and machine intelligence*, vol. 42, no. 4, pp. 824–836, 2018.

[35] X. Wang, X. Wang, and D. M. Wilkes, "A divide-and-conquer approach for minimum spanning tree-based clustering," *IEEE Transactions on Knowledge and Data Engineering*, vol. 21, no. 7, pp. 945–958, 2009.

[36] C. Zhong, M. Malinen, D. Miao, and P. Fränti, "A fast minimum spanning tree algorithm based on k-means," *Information Sciences*, vol. 295, pp. 1–17, 2015.

[37] Y. Ma, H. Lin, Y. Wang, H. Huang, and X. He, "A multi-stage hierarchical clustering algorithm based on centroid of tree and cut edge constraint," *Information Sciences*, vol. 557, pp. 194–219, 2021.

[38] G. Mishra and S. K. Mohanty, "Efficient construction of an approximate similarity graph for minimum spanning tree based clustering," *Applied Soft Computing*, vol. 97, p. 106676, 2020.

[39] W. Dong, C. Moses, and K. Li, "Efficient k-nearest neighbor graph construction for generic similarity measures," in *Proceedings of the 20th international conference on World wide web*, pp. 577–586, 2011.

[40] Z. Liu, P. Luo, X. Wang, and X. Tang, "Deep learning face attributes in the wild," in *Proceedings of the IEEE international conference on computer vision*, pp. 3730–3738, 2015.

[41] S. Sieranoja and P. Fränti, "Constructing a high-dimensional k nn-graph using a z-order curve," *Journal of Experimental Algorithmics (JEA)*, vol. 23, pp. 1–21, 2018.

[42] M. Rezaei and P. Fränti, "Set matching measures for external cluster validity," *IEEE transactions on knowledge and data engineering*, vol. 28, no. 8, pp. 2173–2186, 2016.