GTCLU: A FAST TREE-BASED CLUSTERING FRAMEWORK FOR BIG DATA

Chao Fan
fanvincent@csu.edu.cn

Qilong Feng
csufeng@mail.csu.edu.cn

ABSTRACT

Density-based clustering algorithms are widely used for their ability to cluster data with arbitrary shapes and identify potential noise samples. DBSCAN, in particular, is the most widely used algorithm among them. However, the algorithm's computational time of $O(n^2)$, or $O(n \log n)$ when using an R^*-tree for acceleration, limits its applicability to large datasets. Despite the development of density-based and grid-based variants, they are not commonly used in practice due to a lack of either clustering accuracy or efficiency. In this paper, we propose a fast tree-based clustering framework named GTCLU, which is based on local density and designed to handle big data clustering. Experimental results demonstrate that GTCLU is able to cluster arbitrary data quickly while achieving better clustering quality than DBSCAN and its density-based and grid-based variants. Additionally, we also propose an $O(n)$ time algorithm, named GMCLU, that is suitable for lower dimensional datasets.

***K*eywords** Grid · DBSCAN · Clustering · Bigdata

1 Introduction

Clustering is a fundamental unsupervised learning task in data mining and machine learning that aims to group data based on similarity. This process is done automatically, with the goal of having data within the same group be similar and data within different groups be dissimilar. Over the past few decades, two main directions of clustering have been developed: distance-based and density-based clustering. Distance-based clustering algorithms measure similarity using the euclidean distance, while density-based clustering algorithms measure similarity through local density [1, 2].

Distance-based methods are commonly employed due to their ease of implementation and efficient computation. There are several notable distance-based methods, such as k-means [3], k-medians [4], k-medoids [5], and hierarchical clustering [6]. However, these methods have limitations in dealing with data of non-uniform shapes and are sensitive to outliers. In other words, these methods may produce poor results when the data is poorly distributed or contains noises [7, 8].

Density-based methods have gained increasing attention in recent years as they are robust to noise and perform well on datasets of arbitrary shapes [9, 10]. Among these methods, DBSCAN [11] is the most widely used density-based clustering algorithm in practice. DBSCAN is based on two parameters ϵ and $MinPts$, which define a point's density as the number of points within a radius ϵ, and a point is considered a core point if its density is not less than $MinPts$, otherwise it is considered a border point. The algorithm then clusters the data into groups by connecting core points within the ϵ radius, and border points are added to a group if they are within the ϵ radius of a core point, otherwise, they are marked as noises. The original DBSCAN has a computational time complexity of $O(n^2)$, however, several implementations have been developed to improve it to $O(n \log n)$ by using R^*-tree index structures such as ELKI [12], Scikit-learn [13], Pyclustering [14], etc.

There have been numerous developments and variants of DBSCAN algorithm, aimed at addressing its limitations and improving its performance. Xu proposed DBCLASD [15], a density-based algorithm that does not require any parameters and is based on the assumption of uniform distribution of points within a cluster. Ankerst proposed OPTICS [16] which is designed to detect clusters with varying densities, which is a significant drawback of DBSCAN. Other researchers such as Borah [17], Liu [18] and Liu [19] have also proposed methods to address the problem of

varying density data clustering. DENCLUE [20] proposed by Hinneburg is another density-based clustering algorithm suitable for datasets with a high degree of noise. BRIDGE [21] and CUBN [22] are hybrid clustering algorithms that combine density-based and distance-based methods. Yu proposed KNNDBSCAN [23], an algorithm that combines the KNN algorithm and only requires one parameter k to determine ϵ and $MinPts$ in an unsupervised manner.

The grid-based approach is a variation of DBSCAN that can accelerate the clustering process. This approach divides the data space into rectangles, called grids, and calculates the number of points in each grid to determine the density. The data is then clustered by connecting grids with high density. Schikuta introduced the GRIDCLUS [24] algorithm for clustering large datasets, and later developed the BANG [25] algorithm to address the inefficiencies of GRIDCLUS. Wang proposed the STING [26] and STRING+ [27] algorithms to efficiently cluster low-dimensional datasets, while Sheikholeslami presented the WAVECLUSTER [28] algorithm for the same purpose. Hinneburg proposed the OPTIGRID [29] algorithm for high-dimensional datasets, and Zhao introduced the GDICL [30] algorithm based on the density-isoline technique. Chen developed the GMDBSCAN [31] algorithm, which uses the grid technique to cluster datasets with various densities. Boonchoo [32] recently proposed a grid clustering acceleration method based on indexing and inference. Of these grid-based algorithms, the most widely used is CLIQUE [33], proposed by Agrawal.

Despite the numerous density-based and grid-based clustering algorithms that have been proposed, DBSCAN remains the most commonly used, primarily due to its superior clustering quality compared to its variants, particularly the grid-based variants. However, DBSCAN is not well-suited for big data and high-dimensional data due to its time complexity, although this can be improved to $O(n \log n)$ using the R^*-tree index. The greatest challenge in density-based clustering is to achieve a balance between quality and efficiency [34, 35].

To address the challenge of balancing clustering quality and efficiency, we propose a fast clustering framework named GTCLU, which is based on grid partition tree. GTCLU is efficient for big data clustering and can even achieve a higher quality of clustering than DBSCAN. Our main contributions are as follows:

1. We propose a new grid density measurement method that takes into account the densities and distances of neighboring grids, leading to significant improvements in grid-based clustering quality.
2. Our fast clustering framework is built on a grid partition tree, reducing the search space for clustering and improving its efficiency.
3. Our framework is flexible and can be easily adapted to parallel computing.
4. For low-dimensional datasets, we propose an algorithm named GMCLU, which achieves an exact time complexity of $O(n)$ while requiring only $O(m)$ space, where $m \ll n$ is the number of non-empty grids.

The organization of the remaining part of this paper is outlined as follows. In Section 2, we provide a comprehensive overview of DBSCAN to introduce density-based clustering. Section 3 introduces the fundamental concepts and the primary objective of our study. The details of the proposed GTCLU framework are discussed in Section 4. The results of our experimental evaluations are presented in Section 5. Finally, in Section 6, we draw our conclusions and summarize the contributions of our work.

2 DBSCAN Algorithm

The DBSCAN algorithm remains the most highly regarded and widely adopted density-based clustering algorithm, and serves as the basis for many newer algorithms. To gain a deeper understanding of density-based clustering, this section will provide an in-depth examination of the DBSCAN algorithm.

In DBSCAN, a point is considered to be an ϵ-neighbor of another point if it is located within a radius of ϵ. The local density of a point, denoted as $w(p)$, is defined as the number of its ϵ-neighbors, including the point itself. DBSCAN further classifies points into two categories: core points and border points. A point p is considered a core point if its local density, $w(p)$, is equal to or greater than $MinPts$, while a border point is defined as a point with a local density of less than $MinPts$.

Given a dataset D, DBSCAN first divide it into two parts based on the densities of each point and the parameters ϵ and $MinPts$: the core set D_c and the border set D_b.

For the core set D_c, starting with any core point $c \in D_c$, its ϵ-neighbors in D_c are identified and added to the cluster C_i. The process is then repeated iteratively for all core points in C_i, adding their ϵ-neighbors to the cluster until no new core points can be added. This process is repeated for all unvisited core points, resulting in the division of the core set into several clusters.

For the border set D_b, the ϵ-neighbors in D_c are searched for each border point, and the border point is added to one of the clusters of its core ϵ-neighbors. Border points without a core ϵ-neighbor are considered noise points and are not clustered.

As shown in Figure 1, when $MinPts = 4$ and $\epsilon = 0.5$, all red points are classified as core points as they have a density of 4 or higher. The remaining points are classified as border points. The red points can be divided into two clusters: the left cluster and the right cluster. The blue border points are added to one of these clusters as they are ϵ-neighbors of a red core point. The gray point is classified as a noise point as it is not located within the ϵ radius of any red core point.

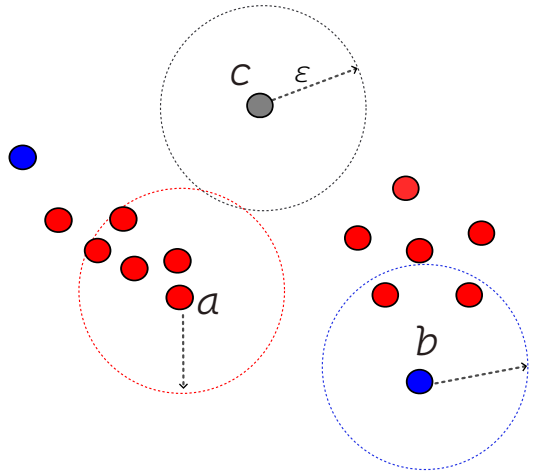


Figure 1: DBSCAN, $MinPts = 4$, $\epsilon = 0.5$.

3 Main Concepts of GTCLU Framework

In the field of density-based clustering, the challenge lies in finding the right balance between clustering quality and efficiency. While direct modifications to DBSCAN often result in good clustering quality, they tend to lack efficiency. On the other hand, grid-based variants of DBSCAN are efficient, but often result in poor clustering quality. To address this challenge, we will introduce our core ideas, which are based on grids and trees, and explain how our method is able to maintain the quality of DBSCAN clustering while also improving its efficiency.

3.1 Our Grid-based Method

The grid-based clustering methods are similar to DBSCAN in that they divide the dataset into grids and identify the core and border grids. The core grids are then clustered, and the border grids are added to the clusters of their core neighbors, just like in DBSCAN. Any un-clustered border grids are treated as noise. However, the general grid-based methods have two problems when measuring the density of each grid. Firstly, they only count the inside points of a grid as its density, without considering the influence of surrounding points outside the grid. This can lead to the misidentification of a core grid as a border grid. Secondly, they simply use the grid position to represent the position of data, which can introduce significant errors when the grid is large. To address these issues, we propose a new measurement method, which we describe in detail in the rest of this section. Furthermore, we provide a comparative example to demonstrate the advantages of our approach.

Normalization. While normalization is not prerequisite, it offers several advantages. Firstly, normalization enables the straightforward mapping of the dataset to a cubic grid space by dividing each dimension into equal parts. Secondly, finding the neighbors of a grid is simplified through basic arithmetic operations. Lastly, normalization enhances the quality of clustering as the dataset can be more accurately gridded. For the remainder of this paper, all computations presume that the dataset has undergone zero-one normalization.

Given two parameters ϵ and $MinPts$, we set the grid width $width$ as Eq.(1), where d is the dimension of the dataset. Then we build the grid space by evenly dividing each dimension into l parts as Eq.(2) and get l^d grids. This design allows our parameters to match the DBSCAN and ensure that the farthest points in a cube grid are within the radius ϵ. And we only need to store the non-empty grids that contain points, so it is very memory efficient because the number of non-empty grids is much less than the number of points, especially when the dataset is low-dimensional.

Given two parameters ϵ and $MinPts$, we determine the grid width as shown in Eq.(1), where d represents the dimension of the dataset. Subsequently, we generate a grid space by uniformly dividing each dimension into l parts, following Eq.(2), which results in l^d grids. This approach ensures that the parameters match those of DBSCAN and guarantees that the farthest points within a cubical grid fall within the radius ϵ. Notably, we only store the non-empty grids that contain points, which significantly improves memory efficiency since the number of non-empty grids is much smaller than the number of points, particularly for low-dimensional datasets.

$$width = \frac{\epsilon}{\sqrt{d}} \tag{1}$$

$$l = \lceil \frac{1}{width} \rceil \tag{2}$$

To represent a grid g, we utilize a 4D tuple depicted in Eq.(3), instead of a simple weighted position. The tuple consists of four components: pos, which indicates the grid's position, w, which denotes the number of points within the grid, ls, the linear sum of the points within the grid, and iw, the influent weight of a grid from its neighboring grids. We define the distance between two grids using Eq.(4), which represents the Euclidean distance of the mean position of the points within the two grids. If two grids' distance is less than or equal to ϵ, they will increase the weight of each other. We refer to the increased weight of a grid from others as influent weight and define it using Eq.(5). In this equation, N_{g_i} denotes the set of neighbor grids of g_i. A grid g_i's influent weight from another grid g_j is defined as Eq.(6). The density of a grid is the sum of its own weight and influent weight as defined in Eq.(7). We utilize this density to distinguish between core and border grids. Specifically, we define the core and border grids as follows:

Core Grid. A grid g is considered a core grid if its density den_g is equal to or greater than $MinPts$.

Border Grid. A grid g is considered a border grid if its density den_g is less than $MinPts$.

Once we have identified the core grids and border grids, we proceed to cluster them using a similar approach to DBSCAN. Specifically, if two grids g_i and g_j are core grids and their distance $dist(g_i, g_j)$ is less than or equal to ϵ, they are connected and assigned to the same cluster. By iteratively joining core grids, we can cluster the core grid set into distinct groups. Then, we include the border grids in these clusters if they are within the ϵ distance of any core grid. Finally, any border grids that are not included in a cluster are considered noise.

$$g = (pos, ls, w, iw) \tag{3}$$

$$dist(g_i, g_j) = \sqrt{\sum_{k=1}^{d} \left(\frac{ls_{g_i}^k}{w_{g_i}} - \frac{ls_{g_j}^k}{w_{g_j}} \right)^2} \tag{4}$$

$$iw_{g_i} = \sum_{g_j \in N(g_i)} iw_{g_i \leftarrow g_j} \tag{5}$$

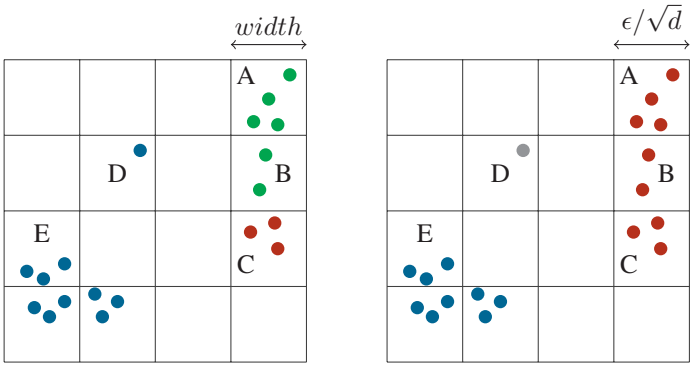
$$iw_{g_i \leftarrow g_j} = \begin{cases} \min\left(\frac{0.5 \cdot \epsilon}{dist(g_i, g_j)}, 1\right) \cdot w_{g_j} & \text{if } dist(g_i, g_j) \leq \epsilon \\ 0 & \text{otherwise} \end{cases} \tag{6}$$

$$den_g = w_g + iw_g \tag{7}$$

The key distinctions between our grid-based method and the conventional approach are as follows:

1. We propose a novel grid density calculation method that takes into account the influence of neighboring grids, which results in more accurate identification of core and border grids. As depicted in Fig.(2) for a $MinPts$ value of 3, the conventional method incorrectly labels grid B as a border grid due to the low number of points within it. This leads to a significant error that splits grids A, B, C into two clusters, even though they should be merged into a single cluster. However, our method can precisely classify grid B as a core grid by taking into account the influence of grids A and C. Consequently, we can cluster these grids correctly into a single cluster.
2. We represent the distance between grids as the mean position of the points in the grids and only connect two grids when their distance is less than or equal to ϵ. However, the common method connects grids when they are neighbors, which may be unreasonable when the grid is large or the number of points in the grid is small. For instance, as shown

in Fig.(2), the common method connects grid D to E and includes D in the blue cluster since they are neighbors. Nevertheless, the point in D is isolated and far from the blue cluster, indicating that it should be identified as a noise point. Our method avoids this by not connecting D to E, correctly identifying D as noise since their distance is greater than ϵ.



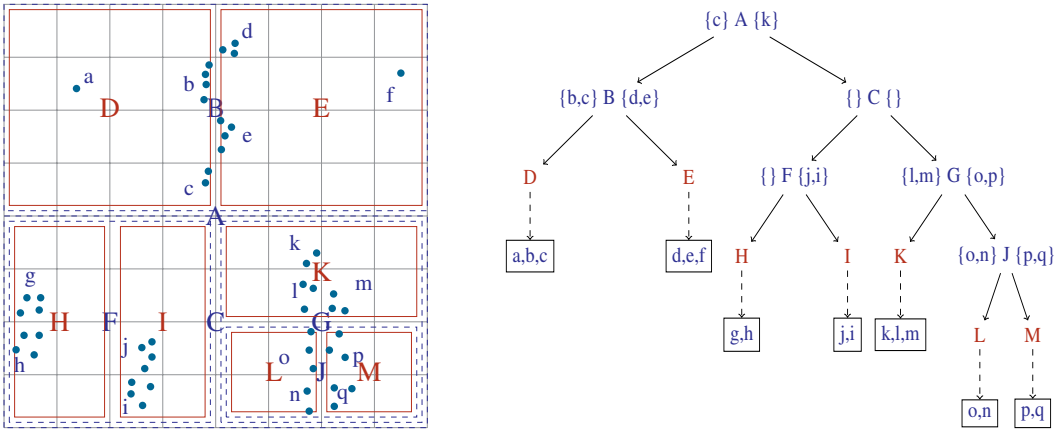
(a) Common method (b) Our method

Figure 2: The grid-based clustering ($MinPts = 3$).

3.2 Tree-based Clustering Framework

When considering grid clustering, we need to identify the non-empty neighbors for each of the m non-empty grids. This is necessary to calculate the influent weight and merge the grids into groups. Two methods can be used to find the neighbors: (1) compute all possible neighbors of a grid using simple math, but this can be time-consuming when the dataset is high-dimensional, as there are $3^d - 1$ possible neighbors surrounding the grid. (2) Search all the non-empty grids and find the ϵ-neighbors of each grid, which has a time complexity of $O(m^2)$. Although the current best implementation can reduce the time complexity to $O(m \log m)$ by using an R^*-tree index model, this is still time-consuming when the dataset is large, particularly when the dataset is high-dimensional that the dataset cannot be well gridded, and the number of grids (m) is not much smaller than the number of points (n).

Assuming that the grid space can be subdivided into smaller subspaces for clustering, we can greatly reduce the search space and improve clustering efficiency by combining the results of the subspaces using a fast method. We propose a tree-based framework consisting of three parts to expedite this process: (1) The partitioning of the grid space into smaller subspaces and construction of a grid partition tree. (2) Clustering of the subspaces stored in leaf nodes. (3) Combination of clustering results by merging clusters from leaf nodes to the root.



(a) Partition (b) Tree

Figure 3: Grid partition tree

Partitioning and Tree Construction. To construct the grid partition tree, we begin by designating the whole grid space as the root node, and subsequently subdivide it into smaller subspaces through recursive partitioning. The recursive

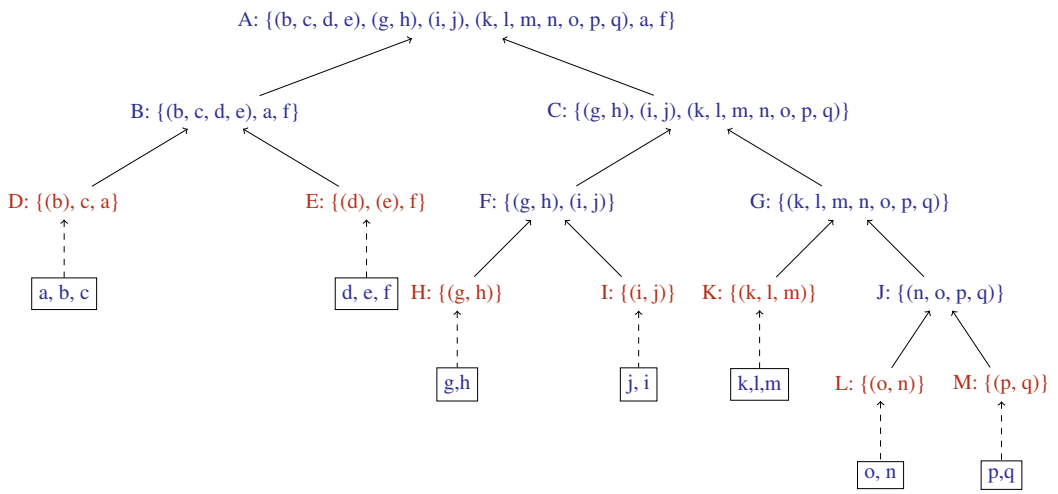


Figure 4: Tree clustering

method is as follows: (1) For a non-leaf node, we randomly select several dimensions of the subspace, determine the longest dimension from these dimensions and divide it into two subspaces by the median value, and designate the resulting subspaces as children of the node. (2) We repeat the first step to build the tree until the number of non-empty grids in the subspace is small enough. During each split, we record the left-side and right-side neighboring grids of the splitting face, and compute the influent weight of neighbors on either side from the other side's neighbors. In the leaf node, we store all non-empty grids in the subspace. In non-leaf nodes, we only store the left-side and right-side neighbor grids of the splitting face. Figure (3) illustrates this process, where the uppercase letters represent the subspaces and the lowercase letters represent the grids, with a stopping condition of 3 grids in a subspace. Node A represents the entire grid space and is divided into two subspaces, B and C. The left-side neighbor c and the right-side neighbor k are stored in node A. After repeated recursive division, we obtain the subspaces shown in Figure (3a) and the grid partition tree shown in Figure (3b).

Tree Clustering. After constructing the grid partition tree, we cluster the grids in the leaf nodes to obtain sub-clusters, and merge these sub-clusters from the leaf nodes to the root in order to obtain the final clustering results. In the leaf node, we apply the grid-based clustering method described in Section 3.1 to cluster the grids and obtain the sub-clusters. In the non-leaf node, we merge the sub-clusters of its children nodes based on the left-side and right-side neighbor grids that we stored to obtain the sub-clusters of this node. The merging steps are as follows: (1) Merge noise grids in left-side neighbors into right clusters if they are ϵ-neighbors of right-side core neighbors. (2) Merge noise grids in right-side neighbors into left clusters if they are ϵ-neighbors of left-side core neighbors. (3) Merge right clusters into left clusters if the right-side core neighbors are ϵ-neighbors of left-side core neighbors. If a right cluster can be merged into several left clusters, we merge them into a single large cluster. Figure (4) depicts the clustering process of the partition tree shown in Figure (3). The leaf nodes, marked in red, are clustered first to obtain the sub-clusters. For instance, for leaf node E, two sub-clusters (d) and (e) are obtained, along with a noise grid f. For leaf node D, a sub-cluster (b) and two noise grids c and a are obtained. Next, on non-leaf node B, we merge its children D and E to obtain a sub-cluster (b, c, d, e) and two noise grids a and f. We then cluster all leaf nodes and merge the sub-results from the bottom up, ultimately yielding the clusters (b, c, d, e), (g, h), (i, j), (k, l, m, n, o, p, q), and noise grids a and f at the root node A.

Our tree-based framework is both versatile and efficient. Firstly, it can be applied not only to grid clustering, but also to any datasets that can be subdivided into small subspaces and clustering methods that can be performed in a merging manner. Secondly, it is easy to implement and can be executed in parallel. The clustering of each leaf node is independent and can be performed concurrently. Likewise, the merging process of each non-leaf node at the same level is independent and can be carried out in parallel. Thirdly, our approach significantly enhances clustering efficiency. The tree can be conveniently constructed by mid-splitting the dimensions. As the number of grids in each leaf node is much lower than the total grid number, a fast grid-based clustering algorithm can be employed to cluster it quickly. Moreover, the merging process is also rapid since the number of side neighbors of each splitting face is very limited, and the merging algorithm is straightforward and efficient.

4 Implementations of Our Algorithms

In the previous section, we presented the main ideas and concepts behind our approach and explained why it can efficiently cluster high-dimensional data. In this section, we will provide a detailed description of the GTCLU algorithm for high-dimensional data clustering. Additionally, we propose a simple $O(n)$ algorithm called GMCLU for clustering low-dimensional data.

4.1 Implementations of GTCLU and GMCLU

To begin, we normalize the dataset, although it is not necessary, to facilitate the creation of the grid space table from the original data space. As depicted in Algorithm 1, the GTCLU framework is composed of three main steps: (1) Create the grid space table. (2) Split the grid space and construct the grid partition tree. (3) Cluster the grids based on the grid partition tree. Finally, we label the origin point dataset according to the grid clustering results and return the labels.

Algorithm 1: GTCLU Framework

Input: dataset D', radius ϵ, density threshold $MinPts$
Output: category labels array $Labels$

Function `GTCLU(`D', ϵ, $MinPts$`)`**:**
  $D \leftarrow$ Normalize dataset D' ;
  $G \leftarrow$ `CreatGridTable(`D, ϵ`)` ;
  $Tree \leftarrow$ `ConstructTree(`G, ϵ ,$MinPts$`)` ;
  `TreeClustering(`$Tree$, ϵ`)` ;
  $Labels \leftarrow$ new array $[-1, ..., -1]$;
  for $i \leftarrow 0$ **to** $|D| - 1$ **do**
    $pos \leftarrow CalPos(D[i])$;
    $Labels[i] \leftarrow G[pos].label$;
  end
return $Labels$

(1) Create Grid Table. We represent the grid space using a hash table, where the key represents the grid position and the value represents the grid. The grid width is set as $width = \frac{\epsilon}{\sqrt{d}}$, and we can divide each dimension into $l = \lfloor \frac{1}{width} \rfloor$ parts as the dataset has been normalized and each dimension falls in the range of $[0, 1]$. The grid position of a point p is a d-dimensional vector $pos = \lfloor p \cdot l \rfloor$. A grid is a structure that contains five attributes: pos, the grid position; ls, the linear sum of the points in the grid; w, the number of points in the grid; iw, the influent weight of the grid from its ϵ-neighbors; $label$, the cluster label of the grid. The hash grid table only needs to store non-empty grids. The grid table can be created by iterating through the points, as shown in Algorithm 2.

Algorithm 2: Create Grid Table

Input: normalized dataset D, radius ϵ
Output: a hash grid table G

Function `CreatGridTable(`D, ϵ`)`**:**
  Evenly split each dimension into $l \leftarrow \lceil \frac{\sqrt{d}}{\epsilon} \rceil$ parts to obtain the grid space ;
  Initialize an empty hash table G to represent the grid space ;
  foreach *point* $p \in D$ **do**
    Compute the grid position $pos \leftarrow \lfloor p \cdot l \rfloor$ of point p ;
    if G *contains* pos **then**
      $grid \leftarrow G[pos]$;
      $grid.w \leftarrow grid.w + 1$;
      $grid.ls \leftarrow grid.ls + p$;
    else
      $G[pos] \leftarrow$ new Grid(pos=pos,ls=p,w=1,iw=0,label=-1) ;
    end
  end
return G

(2) Construct Grid Partition Tree. In order to accelerate the clustering speed, we employ a binary tree to subdivide the grid space into smaller subspaces. The root node of the tree represents the entire grid space, and each non-leaf node is

divided into two subspaces by splitting one dimension until the number of grids in the node is less than or equal to a given threshold $MaxGridNum$, or the tree depth reaches a given threshold $MaxDepth$. The tree comprises two arrays, namely $Levels$ and $Leaves$. The $Levels$ array is a nested array, where each element is a list of nodes at the same level.

Algorithm 3: Construct the Grid Partition tree

Input: Grid space table G, radius ϵ, density threshold $MinPts$, recursive stopping thresholds $MaxGridNum$ and $MaxDepth$
Output: A $Tree$ of grid space G

Function `ConstructTree`(G, ϵ, $MinPts$, $MaxGridNum$, $MaxDepth$)**:**
  $DimRange \leftarrow$ [[0,...,0],[l,...,l]] which represents the range of each dimension ;
  $Tree \leftarrow$ new empty tree with empty lists $Levels$ and $Leves$;
  $Grids \leftarrow G$.values() ;
  `BuildTree`($Grids$, 0 ,$DimRange$) ;
return $Tree$

Function `BuildTree`($Grids$, $Depth$, $DimRange$, $MaxGridNum$, $MaxDepth$)**:**
  $Node \leftarrow$ new empty node ;
  if $Depth \geq MaxDepth$ *or* $Grids.size \leq MaxGridNum$ **then**
    $Node.Grids \leftarrow Grids$;
    $Tree.Leaves.Add(Node)$;
  else
    $SplitDim \leftarrow$ randomly select several dimensions and choose the one with the widest range as the splitting dimension ;
    $SplitLine \leftarrow \lfloor (DimRange[0][SplitDim] + DimRange[1][SplitDim]) / 2 \rfloor$;
    `// To split the grids within a non-leaf node, we iterate over all the grids within the node space and divide them by the median value of the splitting dimension.`
    $LeftGrids \leftarrow \{g \in Grids \mid g.pos[SplitDim] \leq SplitLine\}$;
    $RightGrids \leftarrow \{g \in Grids \mid g.pos[SplitDim] > SplitLine\}$;
    $LeftEdges \leftarrow \{g \in LeftGrids \mid g.pos[SplitDim] = SplitLine\}$;
    $RightEdges \leftarrow \{g \in RightGrids \mid g.pos[SplitDim] = SplitLine + 1\}$;
    `// While dividing a non-leaf node, it is necessary to compute the influent weights of the grids in one edge list with respect to the other.`
    foreach *grid* $g_i \in LeftEdges$ **do**
      `//` $N_{RightEdges}$ `is the` ϵ`-neighbors of` g_i `in` $RightEdges$
      $g_i.iw \leftarrow g_i.iw + \sum_{g_j \in N_{RightEdges}(g_i)} \min\left(\frac{0.5 \cdot \epsilon}{dist(g_i, g_j)}, 1\right) \cdot w_{g_j}$;
    end
    The same procedure is performed for the $RightEdges$ list. ;
    `// Obtain the node's children recursively.`
    $LDimRange \leftarrow$ set $DimRange$[$SplitDim$][1] as $SplitLine$;
    $RDimRange \leftarrow$ set $DimRange$[$SplitDim$][0] as $SplitLine + 1$;
    $LChild \leftarrow$ `BuildTree`($LeftGrids$, $Depth$ + *1*, $LDimRange$, $MaxGridNum$, $MaxDepth$) ;
    $RChild \leftarrow$ `BuildTree`($ReftGrids$, $Depth$ + *1*, $RDimRange$, $MaxGridNum$, $MaxDepth$) ;
    `// Assign the node's attributes and add it to` $Leavels$ `list.`
    $Node.LChild$,$Node.RChild \leftarrow LChild$, $RChild$;
    $Node.SplitDim \leftarrow SplitDim$;
    $Node.LeftEdges$, $Node.RightEdges \leftarrow LeftEdges$, $RightEdges$;
    while $Tree.Levels.size() \leq Depth$ **do**
      $Tree.Levels.Add([\,])$;
    end
    $Tree.Levels[Depth].Add(Node)$;
  end
return $Node$

The $Leaves$ array is a list of leaf nodes. To split a non-leaf node, we first randomly select several dimensions and choose the one with the largest range as the splitting dimension $SplitDim$. Next, we assign $SplitDim$ to the node and

split it into two sub-nodes by computing the median of the splitting dimension. During the splitting process, we store the neighboring grids of the splitting surface in two lists, $LeftEdges$ and $RightEdges$, and calculate the influence weights iw of the grids on one edge list from the grids on the other edge list. Finally, we obtain a tree consisting of $Leaves$ and $Levels$ lists. A leaf node in the $Leaves$ list holds the grids in its space, and some of its grids already contain a part of the influence weights obtained during the splitting process. A non-leaf node in a level list of $Levels$ holds the splitting dimension $SplitDim$ and two lists, $LeftEdges$ and $RightEdges$, which store the left and right neighboring grids of the splitting surface. The algorithm is illustrated in Algorithm 3.

Algorithm 4: Clustering By The Tree

Input: Grid partition $Tree$
Output: Labeled grid space G

Function `TreeClustering`($Tree$)**:**
  foreach $Node \in Tree.Leaves$ **do**
    $Node.Clusters \leftarrow$ GridClustering($Node.Grids$) ;
  end
  for $i \leftarrow Tree.Levels.size() - 1$ **to** 0 **do**
    $LevelNodes \leftarrow Tree.Levels[i]$;
    foreach $Node \in LevelNodes$ **do**
      $Node.Clusters \leftarrow$ `MergeChildren` ($Node$) ;
    end
  end
  $Clusters \leftarrow Tree.Levels[0][0].Clusters$;
  for $i \leftarrow 0$ **to** $Clusters.size() - 1$ **do**
    foreach *grid* $g \in Clusters[i]$ **do**
      $g.label \leftarrow i$;
    end
  end
return G
Function `MergeChildren`($Node$)**:**
  $LClusters, RClusters \leftarrow Node.LChild.Clusters, Node.RChild.Clusters$;
  $LeftEdges, RightEdges \leftarrow Node.LeftEdges, Node.RightEdges$;
  `// step 1. Merge noise grids to Clusters if able`
  foreach $g_l \in LeftEdges$ *and* g_l *has not been clustered* **do**
    if $g_r \in RightEdges$ *and* g_r *is core and* $dist(g_l, g_r) \leq \epsilon$ **then**
      Merge g_l into $g_r.cluster$;
    end
  end
  Do the same thing for noise grids in $RightEdges$;
  `// step 2. Merge the Clusters if able`
  foreach $g_r \in RightEdges$ *and* g_r *is core* **do**
    $LCoreNeighors \leftarrow \{g_l \mid g_l \in LeftEdges$ and g_l is core and $dist(g_l, g_r) \leq \epsilon\}$;
    $LNeighborClusters \leftarrow$ get the clusters of $LCoreNeighors$;
    $NewCluster \leftarrow$ Merge $g_r.cluster$ and $LNeighborClusters$ as one big cluster ;
    $LClusters \leftarrow LClusters \setminus LNeighborClusters \cup NewCluster$;
  end
  $Node.Clusters \leftarrow LClusters$;
return

(3) Grid Partition Tree Clustering. After constructing the grid partition tree, we use a down-to-up merging strategy to expedite the clustering process. First, we cluster the grids in the leaf nodes using any fast grid-based clustering algorithm while following the constraints outlined in Section 3.1 to ensure high clustering quality. We then obtain the sub-clusters within each leaf node. Second, we merge the clusters from the lowest non-leaf nodes up to the root node, level by level. To obtain the sub-clusters of each non-leaf node, we merge its children using the following approaches: (1) we iterate through the noise grids in its two edge lists and merge any noise grid with a core grid's cluster if the noise grid is an ϵ-neighbor of the core grid in the other edge list; (2) we iterate through the core grids in the $RightEdges$ list. For each core grid in $RightEdges$, if there exists any core ϵ-neighbor grid in the $LeftEdges$ list, we extract the clusters of these core grids and merge them to form a large cluster, which is then added to the left clusters; (3) we obtain

the final clusters of this node in the left clusters by implementing step 2, and assign them to the node. After merging the root node, we obtain the clusters for the entire grid space and relabel all grids according to the clusters. The algorithm is presented in Algorithm 4.

GMCLU for Low-dimensional Data Set. We have introduced the GTCLU framework, which is an efficient algorithm suitable for both low-dimensional and high-dimensional datasets. In addition, for low-dimensional datasets, we propose a simpler and more efficient algorithm named GMCLU. In Section 3.2, we discussed that the time-consuming process of grid clustering is to search for the ϵ-neighbors of a grid. To address this issue, we propose a grid partition tree in GTCLU that reduces the search space. In low-dimensional datasets, we can search all $3^d - 1$ possible neighbors of a grid using simple calculation and identify its actual neighbors. Although this method is impractical for high-dimensional space, the $3^d - 1$ possibilities are manageable in low-dimensional space, making the search process very fast. Therefore, we can use a straightforward breadth-first search and merge approach to cluster the grids. Initially, we iterate through each grid, search its ϵ-neighbors to mark the core grids and border grids. Then, we start from any core grid, generate a new group, and use a breadth-first method to search all unvisited core grids. When visiting a core grid, we search all of its possible neighbors and merge its ϵ-neighbors into the cluster. If a ϵ-neighbor is a core grid, we add it to the breadth queue to visit later. When all core grids are visited, the dataset is clustered into groups, and unvisited border grids are considered as noises. The algorithm is presented in Algorithm 5.

Algorithm 5: GMCLU for low-dimensional datasets

Input: Dataset D', radius ϵ, density threshold $MinPts$
Output: category labels array $Labels$

Function `GMCLU(`D', ϵ, $MinPts$`):`
  $D \leftarrow$ Normalize dataset D' ;
  $G \leftarrow$ `CreatGridTable(`D, ϵ`)` ;
  $Cores$, $Boders \leftarrow$ identify core grids and border grids in G ;
  `// BFS search to cluster and label grids.`
  $Queue \leftarrow$ new empty queue ;
  $LabelId \leftarrow 0$;
  foreach $Core \in Cores$ *and* $Core$ *has not been visited* **do**
    $Queue$.Add($Core$) ;
    $Core.label \leftarrow LabelId$;
    Mark $Core$ as visited status ;
    while Q *is not empty* **do**
      $c \leftarrow Queue$.Remove() ;
      foreach $neighbor \in$ *possible neighbors of grid* c **do**
        if $neighbor \in G$ *and* $neighbor$ *has not been visited* **then**
          $neighbor.label \leftarrow LabelId$;
          Mark $neighbor$ as visited status ;
          if $neighbor$ *is a core grid* **then**
            $Queue$.Add($neighbor$) ;
          end
        end
      end
      $LabelId \leftarrow LabelId + 1$;
    end
  end
  $Labels \leftarrow$ label all points base on the labeled grids ;
return $Labels$

4.2 Complexity Analysis

GTCLU. The GTCLU algorithm comprises three main steps, and we will analyze the complexity of each step. Firstly, the grid table is created by iterating through all data points, and for each point, its grid position is calculated and inserted into the hash table in $O(1)$ time. Thus, the time complexity of this step is $O(n)$, where n is the number of data points. The non-empty grids are stored in the hash table, with each grid costing $O(1)$ space, resulting in a space complexity of $O(m)$, where $m \ll n$ is the number of non-empty grids. Secondly, the grid partition tree is constructed by recursively splitting the grid space. To divide a non-leaf node into two sub-nodes, one dimension is split, and the influent weights of the grids in two edge lists are calculated. Dividing a node takes $O(m')$ time, where $m' \ll m$ is the number of grids

in this node. Calculating the influent weights of the grids in two edge lists takes $O(e \log e')$ time, where $e \ll m'$ and $e' \ll m'$ are the number of grids in the two edge lists. A non-leaf node requires $O(m' + e \log e')$ time, and the time of this step is the sum of the time of all non-leaf nodes. All grids in the tree are stored, resulting in a space complexity of $O(m)$. Finally, each leaf node are clustered using a $O(m' \log m')$ time and $O(m')$ extra space grid-based clustering algorithm, where $m' \ll m$ is the number of grids in the leaf node. For each non-leaf node, the clusters of its children are merged using the grids in its edge lists, which takes $O(e \cdot e' + m')$ time. The time complexity of this step is the sum of the time of all nodes. Therefore, the time complexity of the GTCLU algorithm is $O(n + x'(m' + e^2) + x''m' \log m')$, and the space complexity is $O(m)$, where x' is the number of non-leaf nodes and x'' is the number of leaf nodes.

GMCLU. GMCLU employs a simple breadth-first search and merge approach to cluster the grids. Firstly, we create the grid table at a time complexity of $O(n)$ and space complexity of $O(m)$. Next, we iterate through each grid to find its ϵ-neighbors by searching all $3^d - 1$ possible neighbors and marking the core and border grids. This step has a time complexity of $O(m \cdot 3^d)$. Finally, we cluster the grids using the breadth-first search and merge approach, which takes $O(m \cdot 3^d)$ time. Hence, the overall time complexity of GMCLU is $O(n + m \cdot 3^d)$, while the space complexity is $O(m)$. On low-dimensional datasets, where 3^d is a small constant and m is much smaller than n, the time complexity reduces to $O(n)$.

5 Experimental Evaluation

In this study, we evaluate and compare the clustering quality and efficiency of the GTCLU and GMCLU algorithms with two density-based clustering algorithms, DBSCAN [11] and OPTICS [16], as well as two grid-based clustering algorithms, CLIQUE [33] and BANG [25]. Our experiments comprise two parts: the clustering quality evaluation and the clustering efficiency evaluation. We implemented the GTCLU and GMCLU algorithms in Python 3.10 [1], while DBSCAN, OPTICS, CLIQUE, and BANG were implemented using the PyClustering 0.10.1 [14] Python clustering package. We conducted all experiments on an Ubuntu server with 64 Intel Xeon Gold 6230@3.0GHz cores and 160GB RAM, using a single thread.

Table 1: Data Sets

No.	Data Set	Number of Dimensions	Number of Clusters	Number of Data Points
1	Compound	2	6	399
2	D31	2	31	3100
3	Flame	2	2	240
4	Iris	4	3	150
5	Pathbased	2	3	312
6	R15	2	15	600
7	G2-2-30	2	2	2048
8	G2-4-30	4	2	2048
9	G2-8-30	8	2	2048
10	S1	2	15	5000
11	S2	2	15	5000
12	S3	2	15	5000
13	S4	2	15	5000
14	G10-3d-100k	3	10	100000
15	G20-3d-50k	3	20	50000
16	G10-4d-10m	4	10	10000000
17	G10-10d-10k	10	10	10000
18	G20-20d-20k	30	10	20000
19	G10-30d-1m	30	10	1000000
20	G10-30d-10m	30	10	10000000

Data Sets. In order to provide a comprehensive evaluation of the algorithms, we utilized multiple data sets, as summarized in Table 1. The data sets numbered 1-13 are commonly used benchmarks for evaluating clustering quality. The first six data sets feature various shapes and were obtained from previous studies [36–40]. The next seven data sets,

[1] https://github.com/VincentFF/GTCLU-Clustering

numbered 7-13, consist of Gaussian clusters and were cited from literature [41]. The last seven data sets, numbered 14-20, are synthetic Gaussian data sets generated using Scikit-learn [13] with a standard deviation of 0.6. All data sets were min-max normalized prior to clustering.

Table 2: Comparison of the clustering quality

	ARI | AMI					
Data Set	DBSCAN	OPTICS	BANG	CLIQUE	**GMCLU**	**GTCLU**
Compound	0.93 | 0.89	0.93 | 0.89	0.91 | 0.74	0.92 | 0.79	0.96 | 0.93	0.98 | 0.96
D31	0.84 | 0.91	0.80 | 0.90	0.66 | 0.80	0.64 | 0.86	0.83 | 0.90	0.84 | 0.91
Flame	0.97 | 0.93	0.96 | 0.92	0.92 | 0.84	0.87 | 0.73	0.94 | 0.88	0.97 | 0.93
Iris	0.94 | 0.89	0.94 | 0.89	0.99 | 0.97	1.00 | 1.00	0.92 | 0.87	1.00 | 1.00
Pathbased	0.92 | 0.89	0.95 | 0.92	0.50 | 0.59	0.67 | 0.67	0.96 | 0.93	0.95 | 0.92
R15	0.99 | 0.99	0.98 | 0.98	0.90 | 0.95	0.82 | 0.89	0.97 | 0.98	0.98 | 0.99
G2-2-30	0.91 | 0.84	0.91 | 0.84	0.68 | 0.51	0.86 | 0.70	0.91 | 0.84	0.94 | 0.88
G2-4-30	0.99 | 0.98	0.99 | 0.98	0.85 | 0.69	0.97 | 0.94	0.99 | 0.98	1.00 | 1.00
G2-8-30	0.85 | 0.79	0.85 | 0.79	1.00 | 1.00	0.98 | 0.95	1.00 | 1.00	1.00 | 1.00
S1	0.97 | 0.97	0.97 | 0.97	0.88 | 0.88	0.89 | 0.92	0.97 | 0.97	0.98 | 0.97
S2	0.75 | 0.88	0.75 | 0.88	0.67 | 0.77	0.63 | 0.80	0.74 | 0.85	0.75 | 0.85
S3	0.42 | 0.61	0.42 | 0.61	0.37 | 0.50	0.29 | 0.57	0.46 | 0.65	0.47 | 0.58
S4	0.43 | 0.54	0.43 | 0.54	0.41 | 0.50	0.19 | 0.47	0.42 | 0.54	0.45 | 0.53
Average	**0.84 | 0.85**	**0.84 | 0.85**	**0.75 | 0.75**	**0.75 | 0.79**	**0.85 | 0.87**	**0.87 | 0.89**

5.1 Clustering Quality Evaluation

To assess clustering quality, we employ the Adjusted Rand Index (ARI) and Adjusted Mutual Information (AMI) as evaluation metrics and compare GMCLU and GTCLU with four other algorithms: DBSCAN, OPTICS, CLIQUE, and BANG. ARI quantifies the similarity between actual labels and cluster labels, while AMI measures the agreement between the two label sets. These metrics range from -1 to 1, with higher values indicating better clustering quality. We test each algorithm on each dataset multiple times using various parameters and select the parameters that yield the highest ARI. The experimental results are presented in Table 2.

The results indicate that density-based methods, specifically DBSCAN and OPTICS, generally outperform grid-based methods, such as BANG and CLIQUE. Moreover, our proposed algorithms, GMCLU and GTCLU, surpass the other four compared algorithms. In the average metric analysis of these datasets, GMCLU achieves 0.85 on ARI and 0.87 on AMI, while GTCLU attains 0.87 on ARI and 0.89 on AMI. By contrast, BANG scores 0.75 on both ARI and AMI, CLIQUE reaches 0.75 on ARI and 0.79 on AMI, and DBSCAN and OPTICS attain 0.84 on ARI and 0.86 on AMI.

When compared to grid-based algorithms, GMCLU outperforms BANG by 13.3% on ARI and 16.0% on AMI, and CLIQUE by 13.3% on ARI and 10.1% on AMI. Similarly, GTCLU surpasses BANG by 16.0% on ARI and 18.7% on AMI, and CLIQUE by 16.0% on ARI and 12.7% on AMI. When compared to density-based algorithms, GMCLU exceeds both DBSCAN and OPTICS by 1.2% on ARI and 2.4% on AMI, while GTCLU outperforms the same algorithms by 3.6% on ARI and 4.7% on AMI. These results demonstrate that both GMCLU and GTCLU exhibit significantly better clustering quality than the compared grid-based methods and even slightly surpass the compared density-based methods.

We observed that GTCLU slightly outperforms GMCLU in terms of clustering quality on these datasets, even though they employ the same grid-based model. This is primarily because we did not fully implement the model proposed in this paper for GMCLU in order to achieve a time complexity of $O(n)$ in a low-dimensional space. Instead, we made trade-offs to maintain higher clustering efficiency for GMCLU, at the cost of slightly reduced clustering accuracy.

5.2 Clustering Efficiency Evaluation

To assess the effectiveness of clustering, we generated seven synthetic Gaussian cluster datasets of varying sizes, ranging from 10,000 points in three dimensions to 10,000,000 points in 30 dimensions. Details for datasets No. 14

to No. 20 can be found in Table 2. To ensure a fair comparison, we employed two strategies. First, we eliminated the influence of parameter selection by using clustering quality as the sole evaluation criterion, testing the minimal clustering time for each algorithm when the ARI on the dataset reached a specified threshold. Second, to further analyze clustering performance, we evaluated the clustering time of each algorithm, excluding BANG, under the same parameter conditions by setting distinct parameters for each algorithm. We refrained from comparing BANG in this strategy because it employs a different method for constructing grid space, making it infeasible to set the same parameters for BANG and other algorithms. We compared four algorithms—DBSCAN, OPTICS, CLIQUE, and BANG—with GMCLU on low-dimensional datasets and with GTCLU on high-dimensional datasets.

Evaluation on Low-dimensional Datasets

In the first evaluation strategy, we use the ARI as a measure of clustering quality and select the dataset G10-3d-100k for experimentation. Through extensive iterative testing of each algorithm's parameters, we ultimately chose the shortest clustering time required for each algorithm to achieve ARI values of 0.6, 0.7, 0.8, and 0.9 as the evaluation criterion. The experimental results are displayed in Figure 5.

The results reveal that, regardless of the chosen ARI threshold, the GMCLU algorithm consistently yields stable and rapid clustering outcomes on the G10-3d-100k dataset, with a maximum required time not exceeding 1.4 seconds. When the ARI threshold is 0.6, GMCLU is 24 times faster than DBSCAN, 17 times faster than OPTICS, 54 times faster than BANG, and 37 times faster than CLIQUE. When the ARI threshold is 0.7, GMCLU is 166 times faster than DBSCAN, 106 times faster than OPTICS, 80 times faster than BANG, and 37 times faster than CLIQUE. When the ARI threshold is 0.8, GMCLU is 174 times faster than DBSCAN, 106 times faster than OPTICS, 80 times faster than BANG, and 47 times faster than CLIQUE. When the ARI threshold is 0.9, GMCLU is 162 times faster than DBSCAN, 119 times faster than OPTICS, 185 times faster than BANG, and 72 times faster than CLIQUE.

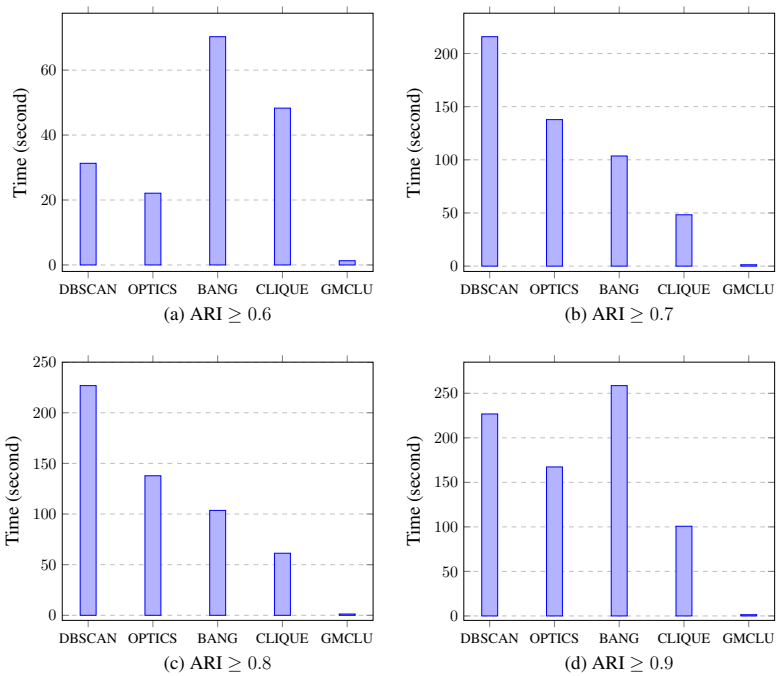
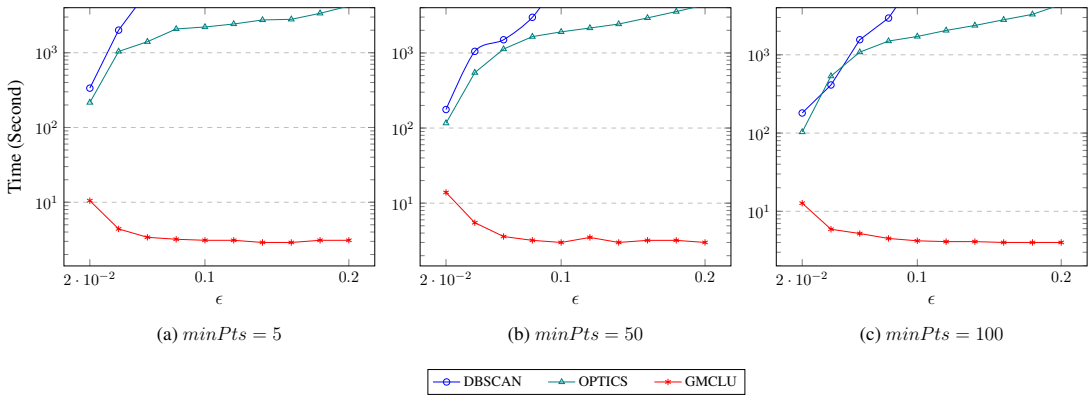


Figure 5: The fastest running time of each algorithm on the G10-3d-100k dataset when the ARI reaches 0.6, 0.7, 0.8, and 0.9, respectively.

In the second evaluation strategy, we assessed the clustering time of each algorithm under the same parameter conditions by setting varied parameters for each algorithm.

We selected the G20-3d-50k dataset with a sample size of 50,000 and tested the clustering time of the GMCLU, DBSCAN, and OPTICS algorithms under multiple sets of parameters. The ϵ parameter value range is [0.02, 0.2] with a step size of 0.02, and the $minPts$ parameter is selected as 5, 50, and 100, respectively. The experimental results are displayed in Figure 6.

The results reveal that the clustering time fluctuations of the proposed GMCLU algorithm are relatively stable with parameter changes, while the clustering times of DBSCAN and OPTICS increase sharply with increasing ϵ. When the value of ϵ is 0.02, GMCLU has the maximum clustering time of 10.5 seconds, 13.9 seconds, and 12.7 seconds with $minPts$ values of 5, 50, and 100, respectively. This is 32.0 times faster than DBSCAN and 20.5 times faster than OPTICS, 12.7 times faster than DBSCAN and 8.4 times faster than OPTICS, and 14.2 times faster than DBSCAN and 8.1 times faster than OPTICS. The advantage of GMCLU's clustering efficiency becomes more significant with the increase of ϵ. When the value of ϵ is 0.18, the clustering times of GMCLU with $minPts$ values of 5, 50, and 100 are 3.1 seconds, 3.2 seconds, and 4.0 seconds, respectively, which are 1085.2 times, 1110.6 times, and 822.4 times faster than OPTICS. When the value of ϵ is 0.2, both DBSCAN and OPTICS exceed the time limit, while the clustering times of GMCLU are only 3.1 seconds, 3.0 seconds, and 4.0 seconds.

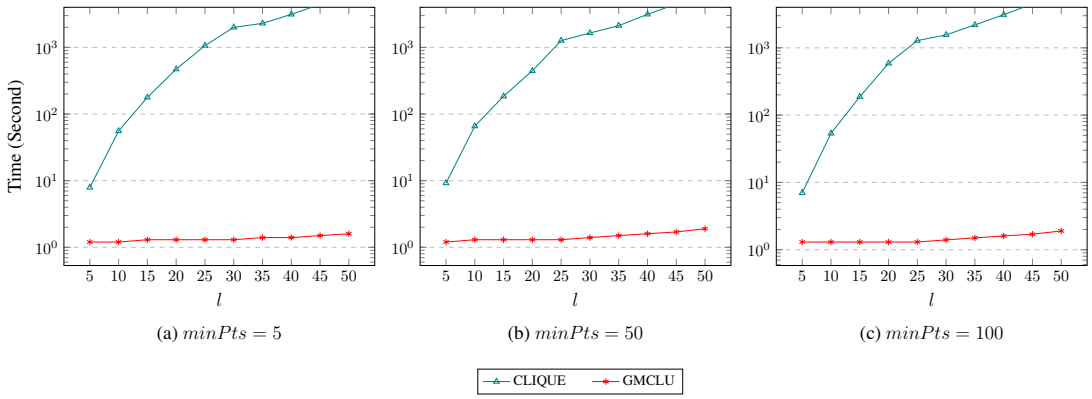


(a) $minPts = 5$ (b) $minPts = 50$ (c) $minPts = 100$

Figure 6: Comparison of clustering time of GMCLU, DBSCAN, and OPTICS on G20-3d-50k dataset.

The CLIQUE algorithm divides the original data space into a grid space by equally partitioning each dimension based on the input parameter l. To match the parameter input of CLIQUE, we also used parameter l to generate a grid space in GMCLU and calculated the radius threshold through $\epsilon = \sqrt{d}/l$. We conducted efficiency tests of CLIQUE and GMCLU on a dataset with a sample size of 100,000. The parameter l is set in the range of [5, 50] with a step size of 5, and $minPts$ is selected as 5, 50, and 100, respectively. The experimental results are displayed in Figure 7.

The results indicate that the clustering time of both CLIQUE and GMCLU is primarily influenced by the dimension partition parameter l. The clustering time of CLIQUE increases sharply with the increase of l, while the clustering time of GMCLU grows more gradually. When l is 5, the clustering times of CLIQUE with $minPts$ values of 5, 50, and 100 are 7.9 seconds, 9.2 seconds, and 7.0 seconds, respectively. Meanwhile, the corresponding clustering times of GMCLU are 1.2 seconds, 1.2 seconds, and 1.3 seconds, which are 6.5 times faster, 7.6 times faster, and 5.4 times faster than CLIQUE, respectively. When l is 40, the clustering times of CLIQUE with the corresponding $minPts$ values are 3133.5 seconds, 3136.5 seconds, and 3102.0 seconds, respectively, while the corresponding clustering times of GMCLU are 1.4 seconds, 1.6 seconds, and 1.6 seconds, which are 2238.2 times faster, 1960.3 times faster, and 1938.8 times faster than CLIQUE, respectively. When the value of l exceeds 40, the clustering time of CLIQUE surpasses the time limit of 4000 seconds. However, GMCLU's clustering time remains less than 2 seconds when l is 50.

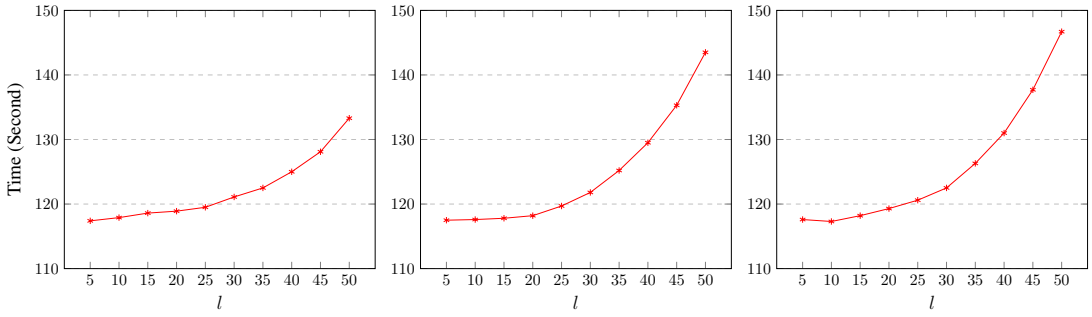


(a) $minPts = 5$ (b) $minPts = 50$ (c) $minPts = 100$

Figure 7: Comparison of clustering time between GMCLU and CLIQUE on G10-3d-100k dataset.

To further test the clustering efficiency of GMCLU on large low-dimensional datasets, we conducted experiments on the G10-4d-10m dataset, which has a sample size of 10 million and a dimension of 4. To more intuitively demonstrate the impact of grid quantity on GMCLU's algorithm efficiency, we selected the equal division of each dimension l and density threshold $minPts$ as parameter settings for the experiment. The range of l is from 5 to 50, and $minPts$ is set to 5, 50, and 100, respectively. Since the other four algorithms could not complete the clustering of this large dataset within the specified time, we only present the experimental results of GMCLU, which are displayed in Figure 8.

As the results show, GMCLU can quickly cluster the large dataset of 10 million samples under various parameters. When $minPts$ is set to 5, GMCLU only needs 117.4 seconds to complete the clustering at l of 5, and only 133.3 seconds when l is 50. When $minPts$ is set to 50, the clustering time of GMCLU with l set to 5 is 117.5 seconds, and 143.5 seconds when l is 50. When $minPts$ is set to 100, the clustering time of GMCLU with l set to 5 is 117.6 seconds, and 146.7 seconds when l is 50. It can be seen that the change of the $minPts$ parameter has little effect on the operating efficiency of the GMCLU algorithm, which is consistent with the design of the GMCLU algorithm. Although the curve shows that the clustering time of GMCLU will increase with the increase of l (the increase of l means the increase of grid quantity), the actual absolute time increase is not substantial. For example, when $minPts$ is set to 100, the clustering time of GMCLU with l set to 50 only increases by 29.1 seconds compared to l set to 5. However, the grid space size between them differs by $50^4 - 5^4 = 6,249,375$. This demonstrates that the clustering efficiency of the GMCLU algorithm is very stable and not sensitive to parameter settings.



(a) $minPts = 5$ (b) $minPts = 50$ (c) $minPts = 100$

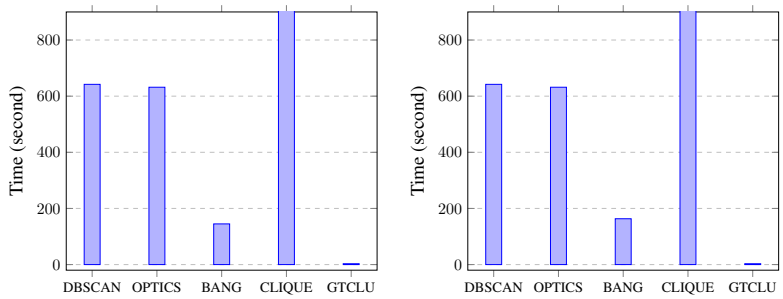
Figure 8: Clustering time of GMCLU on the G10-4d-10m dataset.

Evaluation on High-dimensional Datasets

As with the low-dimensional datasets, we also conducted experiments on high-dimensional datasets to test the clustering efficiency of GTCLU. We first tested the algorithms on the G10-20d-30k dataset with a sample size of 30,000 and a dimension of 20, using ARI values of 0.6, 0.7, 0.8, and 0.9 as clustering quality benchmarks. We iterated through a large number of parameters to select those that corresponded to the fastest time to achieve the specified ARI index for each algorithm and recorded the average clustering time of each algorithm with the selected parameter settings. The experimental results are shown in Figure 9.

From the results, it can be seen that our GTCLU algorithm can quickly complete clustering for any value of ARI ranging from 0.6 to 0.9 as a quality benchmark. When ARI $\leq$ 0.9, GTCLU only needs 3.3 seconds to complete clustering, demonstrating a substantial efficiency advantage compared to other algorithms. The two density-based algorithms, DBSCAN and OPTICS, showed very similar performance in this comparative experiment. When ARI $\leq$ 0.6, DBSCAN completed clustering the fastest at 642.1 seconds, and OPTICS at 631.8 seconds. When ARI $\leq$ 0.9, DBSCAN completed clustering the fastest at 651.0 seconds, and OPTICS at 665.1 seconds. In this test, the GTCLU algorithm was more than 197 times faster than the two density-based clustering algorithms, DBSCAN and OPTICS. The CLIQUE algorithm was unable to complete clustering within the specified time limit in all four ARI tests, indicating that the CLIQUE grid algorithm is not suitable for clustering analysis of high-dimensional data. The BANG algorithm showed good performance among the four comparison algorithms, with the fastest clustering time of 144.9 seconds when ARI $\leq$ 0.6. When ARI $\leq$ 0.9, the fastest time for BANG was 226.9 seconds. Our GTCLU algorithm was more than 68.7 times faster than BANG in this test.

We compared GTCLU with DBSCAN and OPTICS using the same parameters on the G10-10d-10k dataset, which has a sample size of 10,000 and a dimension of 10. The range of the ϵ parameter was set from 0.1 to 1, with a step size of 0.1, and $minPts$ was set to 5, 20, and 50, respectively. The experimental results are shown in Figure 10.



(a) ARI ≥ 0.6 (a) ARI ≥ 0.7

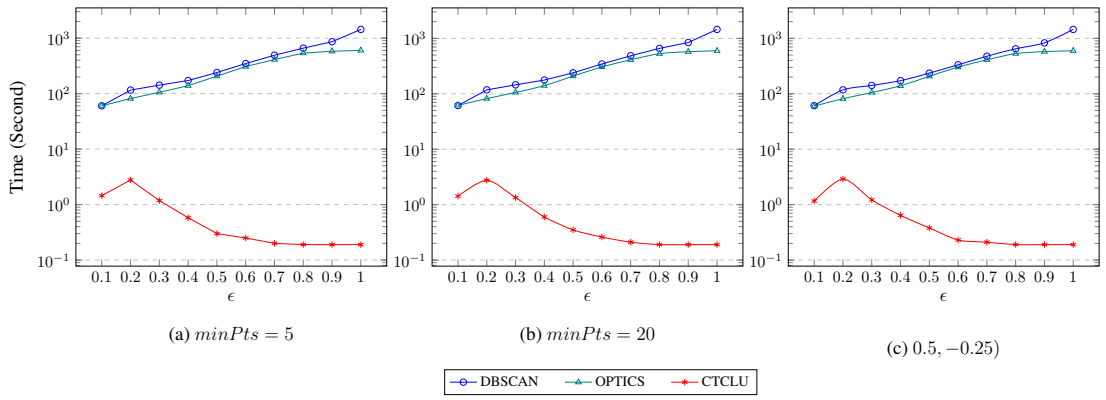
(a) ARI ≥ 0.8 (a) ARI ≥ 0.9

Figure 9: The fastest running time of each algorithm on the G10-20d-30k dataset when the ARI reaches 0.6, 0.7, 0.8, and 0.9, respectively.

From the experimental results, it can be observed that on the G10-10d-10k dataset, the clustering time of GTCLU, DBSCAN, and OPTICS algorithms is not significantly affected by the $minPts$ parameter but is mainly influenced by the ϵ parameter. The clustering time of the GTCLU algorithm decreases with the increase of ϵ and fluctuates at $\epsilon = 0.2$. Conversely, the clustering time of the DBSCAN and OPTICS algorithms increases as ϵ increases. When $minPts$ is 5, the clustering time of DBSCAN and OPTICS is 60.3 and 60.2 seconds when ϵ is 0.1, and 1435.8 and 601.0 seconds when ϵ is 1, respectively. When $minPts$ is 20, the clustering time of DBSCAN and OPTICS is 61.4 and 61.9 seconds when ϵ is 0.1, and 1451.1 and 597.6 seconds when ϵ is 1, respectively. When $minPts$ is 50, the clustering time of DBSCAN and OPTICS is 61.0 and 60.0 seconds when ϵ is 0.1, and 1447.8 and 598.0 seconds when ϵ is 1, respectively. Referring to the curves in the figure, it can be seen that the clustering efficiency of DBSCAN and OPTICS is almost the same when ϵ is less than 0.8. However, as ϵ increases, OPTICS begins to show a time advantage, with a clustering efficiency that can reach about 2.4 times that of DBSCAN when ϵ is 1. Our GTCLU algorithm consumes the most clustering time when ϵ is 0.2, but it does not exceed 3 seconds, and the clustering time does not exceed 0.2 seconds when ϵ is 1. At $\epsilon = 0.2$, the clustering efficiency advantages of GTCLU over DBSCAN and OPTICS are the smallest, about 43 times and 29 times that of DBSCAN and OPTICS, respectively. When ϵ is 1, the efficiency of GTCLU can reach more than 7600 times that of DBSCAN and more than 3100 times that of OPTICS. The clustering efficiency advantage of GTCLU will be further amplified with the growth of the dataset and data dimensions. The parameter settings of this experiment cover the extreme range, such as when ϵ is 0.1, the size of the entire grid space reaches 10 billion, far exceeding the size of the data samples.

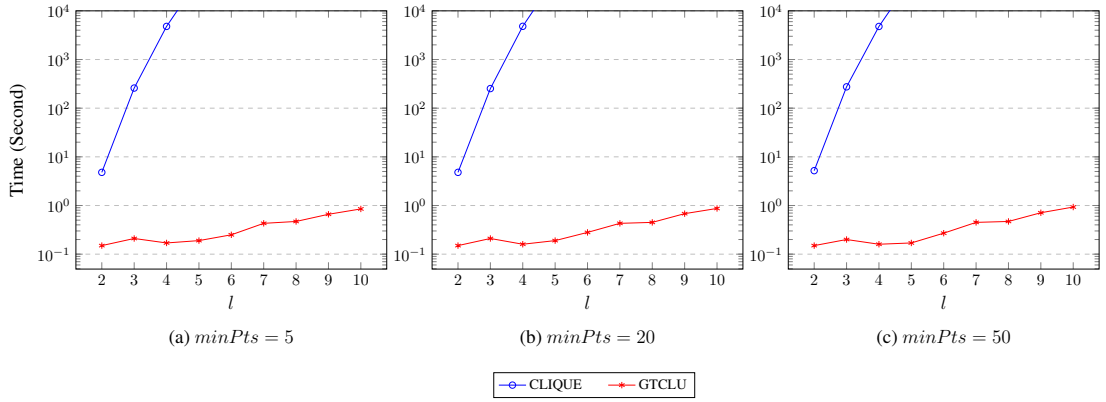
We also compared GTCLU with CLIQUE using the same parameters on the G10-10d-10k dataset. To maintain consistency in the parameters of GTCLU and CLIQUE, we chose l as the parameter and calculated the radius threshold through $\epsilon = \sqrt{d}/l$. The experiment set l ranging from 2 to 10 and selected $minPts$ to be 5, 20, and 50, respectively. The experimental results are shown in Figure 11. From the results, it can be observed that the clustering time of the CLIQUE algorithm sharply increases with the increase of l, exceeding 4500 seconds at $l = 4$ and even surpassing 10,000 seconds at $l = 5$ before the program terminates. In contrast, the clustering time of the GTCLU algorithm gradually increases with the rise of l, taking only 0.15 seconds to complete the clustering at $l = 2$. Moreover, the maximum clustering time at $l = 10$ requires a mere 0.93 seconds.

Since the other four comparative algorithms result in timeouts when the sample size is excessively high, it is not feasible to obtain comparative results of various clustering algorithms for large sample datasets. We selected two datasets for GTCLU's big data clustering test: G10-30d-1m with a sample size of one million and G10-30d-10m with a sample size of ten million. Both datasets have a dimension of 30. The l parameter was set from 2 to 5, and the $minPts$ parameter



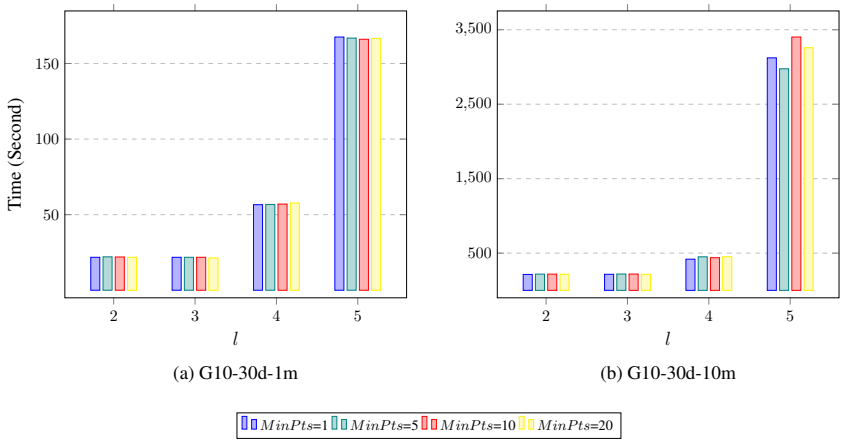
(a) $minPts = 5$ (b) $minPts = 20$ (c) $0.5, -0.25)$

Figure 10: Comparison of clustering time for DBSCAN, OPTICS, and GTCLU on the G10-10d-10k dataset.



(a) $minPts = 5$ (b) $minPts = 20$ (c) $minPts = 50$

Figure 11: Comparison of clustering time between GMCLU and CLIQUE on G10-10d-10k dataset.



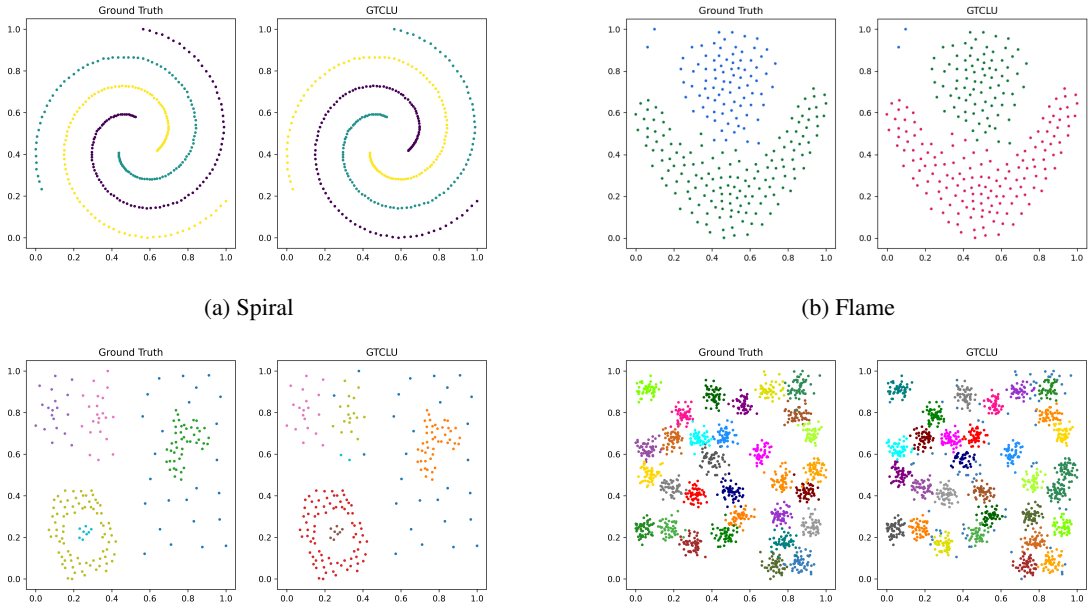
(a) G10-30d-1m (b) G10-30d-10m

Figure 12: Clustering time of GTCLU on the G10-30d-1m and G10-30d-10m datasets.

was set to 1, 5, 10, and 20, respectively. Given that the number of grids in the grid space is l^d, the entire grid space capacity exceeds 1 billion when l is set to 2 in the case of 30 dimensions, far surpassing the number of points. This is why, in this test, the maximum value of l is only set to 5.

The experimental results of this test are shown in Figure 12. It can be observed that GTCLU demonstrates low sensitivity to the minPts parameter in these two high-dimensional large datasets, and its clustering time is primarily affected by the grid partition parameter l. Moreover, the clustering time does not differ significantly when l ranges from 2 to 4, and a substantial increase in time occurs only when it reaches 5. For the 30-dimensional dataset with a sample size of one million, the clustering time of GTCLU does not exceed 22 seconds when l is set to 2 or 3, and does not surpass 58 seconds when l is set to 4. Although the clustering time increases abruptly when l is set to 5, it remains below 168 seconds. For the ten million 30-dimensional dataset, the clustering time of GTCLU does not exceed 219 seconds when l is set to 2 or 3, and does not surpass 452 seconds when l is set to 4. Although the maximum clustering time when l is set to 5 reaches 3,400 seconds, as mentioned earlier, the grid space under this parameter setting is excessively large.

At the end of the experiment, we showcase a set of comparison figures depicting the clustering results and true label sets obtained by GTCLU on the Spiral, Flame, Compound, and D31 datasets, as illustrated in Figure 13. The figure demonstrates that GTCLU attains strong performance across a diverse range of dataset types.



(a) Spiral (b) Flame

(c) Compound (d) D31

Figure 13: Clustering results of GTCLU on Spiral, Flame, Compound, and D31 datasets.

6 Conclusions

Density-based clustering algorithms are widely employed in various applications. However, their slow clustering speed renders them unsuitable for large and high-dimensional datasets. Grid-based clustering algorithms have been proposed to enhance clustering speed, but they often compromise clustering quality and are not ideal for high-dimensional datasets. In this paper, we propose a new grid-based clustering method that significantly improves the clustering quality of grids, even surpassing density-based clustering methods such as DBSCAN and OPTICS. Moreover, we introduce a tree-based clustering framework called GTCLU that substantially accelerates clustering speed, making it suitable for large and high-dimensional datasets. Furthermore, we present the GMCLU algorithm, with a time complexity of $O(n)$, for clustering in low-dimensional datasets. Experimental results demonstrate that our algorithms can efficiently cluster large datasets while maintaining high clustering quality. Future work could investigate the integration of dimension reduction techniques with GTCLU to enhance its performance on extremely high-dimensional datasets.

References

[1] Mahamed G. H. Omran, Andries P. Engelbrecht, and Ayed A. Salman. An overview of clustering methods. *Intell. Data Anal.*, 11(6):583–605, 2007.

[2] Anil K. Jain, M. Narasimha Murty, and Patrick J. Flynn. Data clustering: A review. *ACM Comput. Surv.*, 31(3):264–323, 1999.

[3] Stuart P. Lloyd. Least squares quantization in PCM. *IEEE Trans. Inf. Theory*, 28(2):129–136, 1982.

[4] Paul S. Bradley, Olvi L. Mangasarian, and W. Nick Street. Clustering via concave minimization. In *Advances in Neural Information Processing Systems 9, NIPS, Denver, CO, USA, December 2-5, 1996*, pages 368–374. MIT Press, 1996.

[5] Leonard Kaufman and Peter J. Rousseeuw. *Finding Groups in Data: An Introduction to Cluster Analysis*. John Wiley, 1990.

[6] Robin Sibson. SLINK: an optimally efficient algorithm for the single-link cluster method. *Comput. J.*, 16(1):30–34, 1973.

[7] Anil K. Jain. Data clustering: 50 years beyond k-means. *Pattern Recognit. Lett.*, 31(8):651–666, 2010.

[8] Guojun Gan, Chaoqun Ma, and Jianhong Wu. *Data clustering - theory, algorithms, and applications*. SIAM, 2007.

[9] Charu C. Aggarwal and Chandan K. Reddy, editors. *Data Clustering: Algorithms and Applications*. CRC Press, 2014.

[10] Panthadeep Bhattacharjee and Pinaki Mitra. A survey of density based clustering algorithms. *Frontiers Comput. Sci.*, 15(1):151308, 2021.

[11] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD-96), Portland, Oregon, USA*, pages 226–231. AAAI Press, 1996.

[12] Erich Schubert and Arthur Zimek. ELKI: A large open-source library for data analysis - ELKI release 0.7.5 "heidelberg". *CoRR*, abs/1902.03616, 2019.

[13] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake VanderPlas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Edouard Duchesnay. Scikit-learn: Machine learning in python. *J. Mach. Learn. Res.*, 12:2825–2830, 2011.

[14] Andrei Novikov. Pyclustering: Data mining library. *J. Open Source Softw.*, 4(36):1230, 2019.

[15] Xiaowei Xu, Martin Ester, Hans-Peter Kriegel, and Jörg Sander. A distribution-based clustering algorithm for mining in large spatial databases. In Susan Darling Urban and Elisa Bertino, editors, *Proceedings of the Fourteenth International Conference on Data Engineering, Orlando, Florida, USA, February 23-27, 1998*, pages 324–331. IEEE Computer Society, 1998.

[16] Mihael Ankerst, Markus M. Breunig, Hans-Peter Kriegel, and Jörg Sander. OPTICS: ordering points to identify the clustering structure. In Alex Delis, Christos Faloutsos, and Shahram Ghandeharizadeh, editors, *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Philadelphia, Pennsylvania, USA*, pages 49–60. ACM Press, 1999.

[17] B Borah and DK Bhattacharyya. A clustering technique using density difference. In *2007 International Conference on Signal Processing, Communications and Networking*, pages 585–588. IEEE, 2007.

[18] Peng Liu, Dong Zhou, and Naijun Wu. Vdbscan: varied density based spatial clustering of applications with noise. In *2007 International conference on service systems and service management*, pages 1–4. IEEE, 2007.

[19] Rui Liu, Hong Wang, and Xiaomei Yu. Shared-nearest-neighbor-based clustering by fast search and find of density peaks. *Information Sciences*, 450:200–226, 2018.

[20] Alexander Hinneburg, Daniel A Keim, et al. *An efficient approach to clustering in large multimedia databases with noise*, volume 98. Bibliothek der Universität Konstanz, 1998.

[21] Manoranjan Dash, Huan Liu, and Xiaowei Xu. '1+ 1> 2': Merging distance and density based clustering. In *Proceedings Seventh International Conference on Database Systems for Advanced Applications. DASFAA 2001*, pages 32–39. IEEE, 2001.

[22] Li Wang and Zheng-Ou Wang. Cubn: A clustering algorithm based on density and distance. In *Proceedings of the 2003 International Conference on Machine Learning and Cybernetics (IEEE Cat. No. 03EX693)*, volume 1, pages 108–112. IEEE, 2003.

[23] Xiaopeng Yu, Deyi Zhou, and Yan Zhou. A new clustering algorithm based on distance and density. In *Proceedings of ICSSSM'05. 2005 International Conference on Services Systems and Services Management, 2005.*, volume 2, pages 1016–1021. IEEE, 2005.

[24] Erich Schikuta. Grid-clustering: An efficient hierarchical clustering method for very large data sets. In *Proceedings of 13th international conference on pattern recognition*, volume 2, pages 101–105. IEEE, 1996.

[25] Erich Schikuta and Martin Erhart. The bang-clustering system: Grid-based data analysis. In *International Symposium on Intelligent Data Analysis*, pages 513–524. Springer, 1997.

[26] Wei Wang, Jiong Yang, Richard Muntz, et al. Sting: A statistical information grid approach to spatial data mining. In *Vldb*, volume 97, pages 186–195, 1997.

[27] Wei Wang, Jiong Yang, and Richard Muntz. Sting+: An approach to active spatial data mining. In *Proceedings 15th International Conference on Data Engineering (Cat. No. 99CB36337)*, pages 116–125. IEEE, 1999.

[28] Gholamhosein Sheikholeslami, Surojit Chatterjee, and Aidong Zhang. Wavecluster: A wavelet based clustering approach for spatial data in very large databases. *VLDB J.*, 8(3-4):289–304, 2000.

[29] Alexander Hinneburg and Daniel A. Keim. Optimal grid-clustering: Towards breaking the curse of dimensionality in high-dimensional clustering. In *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, pages 506–517. Morgan Kaufmann, 1999.

[30] Zhao Yanchang and Song Junde. Gdilc: a grid-based density-isoline clustering algorithm. In *2001 International Conferences on Info-Tech and Info-Net. Proceedings (Cat. No. 01EX479)*, volume 3, pages 140–145. IEEE, 2001.

[31] Chen Xiaoyun, Min Yufang, Zhao Yan, and Wang Ping. Gmdbscan: multi-density dbscan cluster based on grid. In *2008 IEEE International Conference on e-Business Engineering*, pages 780–783. IEEE, 2008.

[32] Thapana Boonchoo, Xiang Ao, Yang Liu, Weizhong Zhao, Fuzhen Zhuang, and Qing He. Grid-based DBSCAN: indexing and inference. *Pattern Recognit.*, 90:271–284, 2019.

[33] Rakesh Agrawal, Johannes Gehrke, Dimitrios Gunopulos, and Prabhakar Raghavan. Automatic subspace clustering of high dimensional data. *Data Min. Knowl. Discov.*, 11(1):5–33, 2005.

[34] Erich Schubert, Jörg Sander, Martin Ester, Hans-Peter Kriegel, and Xiaowei Xu. DBSCAN revisited, revisited: Why and how you should (still) use DBSCAN. *ACM Trans. Database Syst.*, 42(3):19:1–19:21, 2017.

[35] Mustafa Tareq, Elankovan A. Sundararajan, Aaron Harwood, and Azuraliza Abu Bakar. A systematic review of density grid-based clustering for data streams. *IEEE Access*, 10:579–596, 2022.

[36] Charles T. Zahn. Graph-theoretical methods for detecting and describing gestalt clusters. *IEEE Trans. Computers*, 20(1):68–86, 1971.

[37] Cor J. Veenman, Marcel J. T. Reinders, and Eric Backer. A maximum variance cluster algorithm. *IEEE Trans. Pattern Anal. Mach. Intell.*, 24(9):1273–1280, 2002.

[38] Limin Fu and Enzo Medico. Flame, a novel fuzzy clustering method for the analysis of DNA microarray data. *BMC Bioinform.*, 8, 2007.

[39] Sugato Basu. Semi-supervised clustering with limited background knowledge. In Deborah L. McGuinness and George Ferguson, editors, *Proceedings of the Nineteenth National Conference on Artificial Intelligence, Sixteenth Conference on Innovative Applications of Artificial Intelligence, July 25-29, 2004, San Jose, California, USA*, pages 979–980. AAAI Press / The MIT Press, 2004.

[40] Hong Chang and Dit-Yan Yeung. Robust path-based spectral clustering. *Pattern Recognit.*, 41(1):191–203, 2008.

[41] Pasi Fränti and Sami Sieranoja. K-means properties on six clustering benchmark datasets. *Appl. Intell.*, 48(12):4743–4759, 2018.