Pattern Recognition 142 (2023) 109658

Contents lists available at ScienceDirect

Pattern Recognition

journal homepage: www.elsevier.com/locate/patcog

GriT-DBSCAN: A spatial clustering algorithm for very large databases

Xiaogang Huang[a], Tiefeng Ma[a,*], Conan Liu[b], Shuangzhe Liu[c]

[a] School of Statistics, Southwestern University of Finance and Economics, Chengdu, Sichuan 611130, China
[b] UNSW Business School, University of New South Wales, Sydney, NSW 2052, Australia
[c] Faculty of Science and Technology, University of Canberra, Canberra, ACT 2617, Australia

ARTICLE INFO

Article history:
Received 14 November 2022
Revised 3 April 2023
Accepted 29 April 2023
Available online 6 May 2023

Keywords:
DBSCAN
Clustering
Indexing methods
Spatial databases

ABSTRACT

DBSCAN is a fundamental spatial clustering algorithm with numerous practical applications. However, a bottleneck of DBSCAN is its $O(n^2)$ worst-case time complexity. To address this limitation, we propose a new grid-based algorithm for exact DBSCAN in Euclidean space called GriT-DBSCAN, which is based on the following two techniques. First, we introduce grid tree to organize the non-empty grids for the purpose of efficient non-empty neighboring grids queries. Second, by utilizing the spatial relationships among points, we propose a technique that iteratively prunes unnecessary distance calculations when determining whether the minimum distance between two sets is less than or equal to a certain threshold. We theoretically demonstrate that GriT-DBSCAN has excellent reliability in terms of time complexity. In addition, we obtain two variants of GriT-DBSCAN by incorporating heuristics, or by combining the second technique with an existing algorithm. Experiments are conducted on both synthetic and real-world data sets to evaluate the efficiency of GriT-DBSCAN and its variants. The results show that our algorithms outperform existing algorithms.

1. Introduction

Spatial clustering is a fundamental technique in data analysis and has extensive applications in the field of pattern recognition. For example, a local tangent plane distance-based approach is taken in 3D point cloud segmentation via clustering [1], a spectral clustering algorithm based on particle swarm optimization is proposed for text clustering [2], and unsupervised person re-identification via simultaneous clustering and mask prediction is studied [3]. In general, the objective of spatial clustering is to partition a given data set into several clusters, such that objects in the same cluster are homogeneous, and objects from different clusters are heterogeneous. A large number of algorithms have been proposed for spatial clustering.

Among all the spatial clustering algorithms, DBSCAN [4] is perhaps one of the most widely used since it can discover clusters of arbitrary shapes and noises. However, the worst-case running time complexity of DBSCAN is $O(n^2)$ [5], regardless of the value of *MinPts* (the density threshold, with the radius ε, which are the parameters of DBSCAN). This is because DBSCAN uses pointwise ε-neighborhood queries. When all points are within an $\varepsilon/2$-ball, the running time for all of these queries is $O(n^2)$. Many improved algorithms have been proposed to reduce the complexity of DBSCAN.

In general, there are three main strategies to reduce the complexity of DBSCAN: grid-based, ball-based, and sampling-based strategies. The grid-based algorithms, such as G13 introduced by Gunawan [5], use the grid structure to reduce the range query time by considering the neighboring grids. They proved that the running time of G13 is $O(n \log n)$. However, G13 is only capable of handling 2-dimensional data. Recently, G13 was extended by Gan and Tao [6] to handle higher dimensions with sub-quadratic complexity, and a linear time approximate algorithm was proposed. On the other hand, ball-based algorithms, such as NQ-DBSCAN [7], use 2ε-ball (the d-dimensional ball with radius 2ε) to perform local neighborhood searching to reduce the time of range query. However, the ball-based algorithms require $O(n^2)$ time in the worst case. Unlike the two strategies just mentioned, the sampling-based algorithms improve DBSCAN by reducing the number of range queries. For example, IDBSCAN [8] expands the cluster by performing range queries on some representatives sampled from a core point's neighborhood. Although sampling-based algorithms are faster, their results may be inconsistent with those of DBSCAN. To the best of our knowledge, there is no algorithm that can produce the same results as DBSCAN, while exhibiting complexity that is linear to n.

* Corresponding author.
E-mail address: matiefeng@swufe.edu.cn (T. Ma).

https://doi.org/10.1016/j.patcog.2023.109658
0031-3203/

In this paper, we propose a simple and efficient grid-based DBSCAN algorithm for low-dimensional data. The main contributions of this paper are as follows:

- We introduce a novel tree-like data structure, namely, grid tree, to organize the non-empty grids. Using grid tree, we propose an efficient non-empty neighboring grids query technique. For each non-empty grid, the running time to find its non-empty neighboring grids is linear to the number of non-empty neighboring grids in the best case.
- We propose an efficient technique, namely, FastMerging, to determine whether the minimum distance between two sets is less than or equal to a certain threshold, which iteratively prunes unnecessary distance calculations by considering the spatial relationships among points.
- Based on the above two techniques, we propose a new grid-based exact DBSCAN algorithm called GriT-DBSCAN[1] whose complexity is almost linear in n.
- We conduct extensive experiments to evaluate the performance of GriT-DBSCAN and its two variants as well. The results show the superiority of our algorithm and its variants.

The rest of this paper is organized as follows. Section 2 presents the related work. Section 3 revisits the existing algorithms that are related to our algorithm. Section 4 presents our algorithm, including a detailed description of our algorithm and an analysis of its theoretical properties. Section 5 compares the performance of our algorithm and its variants to existing algorithms. Finally, conclusions are provided in Section 6.

2. Related work

In order to reduce the complexity of DBSCAN, many algorithms have been proposed. This section briefly reviews the existing algorithms, that are related to our algorithm.

2.1. Grid-based algorithms

Grid-based algorithms aim to reduce the complexity of DBSCAN by partitioning the feature space into grids so as to reduce the range query time by only considering neighboring grids. GriDBSCAN [9] partitions the feature space into equally sized grids. The points in each grid and the points in the ε-enclosure around the grid are considered a group. For each point in a certain grid, its ε-neighbors are inside the corresponding group, which reduces the time for range queries. However, GriDBSCAN requires one more parameter specified by the user. Recently, Gunawan [5] proposed a grid-based algorithm called G13, which in the worst case has time complexity $O(n \log n)$. Nonetheless, G13 is only suitable for 2-dimensional data. Gan and Tao [6,10] extended G13 to higher dimensions. To be specific, Gan and Tao proposed an exact DBSCAN algorithm that runs in sub-quadratic time and an approximate DBSCAN algorithm that runs in $O(n)$ expected time. When merging the core grids, the approximate DBSCAN algorithm proposed by Gan and Tao exploits a quadtree-like hierarchical grid structure for approximate range count, resulting in an expected complexity of $O(n)$ for the entire algorithm. As pointed out in Boonchoo et al. [11], different ordering in forming clusters in the exact DBSCAN algorithm proposed by Gan and Tao [6] leads to different running times. Hence, they proposed two strategies, namely uniform random order and low density first order, to reduce the running time of forming clusters. However, the results of Boonchoo et al. [11] do not improve the theoretical running time of previous work.

[1] The source code is available at: https://github.com/XiaogangHuang/GriT-DBSCAN.

To summarise, there is no known grid-based DBSCAN that can obtain the same results as DBSCAN while exhibiting complexity linear to n.

2.2. Ball-based algorithms

Ball-based algorithms divide the data set into several subsets using d-dimensional balls. Both AnyDBC [12] and IncAnyDBC [13] use ε-ball to divide the data set into primitive clusters. Then, the cluster structure of the data set is iteratively and actively learned. In each iteration, a small set of the most promising points are selected for refining clusters. G-DBSCAN [14] employs the group method based on ε-ball to obtain a set of groups and runs DBSCAN using groups to accelerate range queries. NQ-DBSCAN [7] uses a local neighborhood searching technique based on 2ε-ball to reduce the time of range query. By using kNN-ball (k-nearest neighbors ball), KNN-BLOCK DBSCAN [15] can quickly identify the core points and partition the data set into core-blocks, noncore-blocks, and noise-blocks. Then the core blocks are merged into clusters. However, KNN-BLOCK DBSCAN is an approximate algorithm. Recently, BLOCK-DBSCAN [16] employs $\varepsilon/2$-ball to quickly identify inner core blocks, within which all points are core points, and applies a fast approximate algorithm to classify whether two inner core blocks are density reachable from each other.

Although the ball-based algorithms improve the performance of DBSCAN to some extent, their worst-case time complexity is still $O(n^2)$.

2.3. Sampling-based algorithms

DBSCAN expands clusters by performing range queries for each point in the data set, which is particularly time-consuming. The sampling-based algorithms are designed to perform range queries on a subset of the data set or reduce the range query time by finding ε-neighbors on a subset, thereby reducing the complexity of DBSCAN. SDBSCAN [17] selects a small number of representative points from the data set to perform DBSCAN. l-DBSCAN [18] and Rough-DBSCAN [19] use the leaders algorithm [20] to obtain prototypes from the data set and run DBSCAN on the prototypes to form clusters. Unlike the sampling-based algorithms mentioned above, IDBSCAN [8] reduces the number of range queries by selecting representatives inside the ε-neighborhood of each core point when expanding clusters. FDBSCAN [21] omits the unnecessary range queries by selecting representatives outside the ε-neighborhood of a core point. Recently, Jang and Jiang [22] suggested DBSCAN++, which reduces the complexity of DBSCAN by performing range queries on a subset of the data set obtained by uniform sampling or k-center algorithm [23]. For each point, SNG-DBSCAN [24] finds its ε-neighborhood on a subset of the data set to reduce the range query time.

Nevertheless, the clustering results of the sampling-based algorithms may still be inconsistent with the results of DBSCAN.

3. Preliminaries

This section covers the definitions and basic clustering processes of the existing algorithms that are related to our algorithm, which facilitates the comprehension of our algorithm. Section 3.1 reviews the original DBSCAN algorithm in Ester et al. [4]. Section 3.2 reviews the G13 proposed by Gunawan [5] that improves the performance of DBSCAN in the 2-dimensional space. Section 3.3 reviews the state-of-the-art approximate algorithm—ρ-approximate DBSCAN in Gan and Tao [10]—that solves the problem with slight inaccuracy in $O(n)$ expected time.

3.1. DBSCAN

Consider a set of n points P in d-dimensional space with a distance function $dist$: $R^d \times R^d \to R$ giving the Euclidean distance $dist(p,q)$ between $p,q \in P$. $N_\varepsilon(p)$ denotes the ε-neighborhood of $p \in P$ with radius ε, i.e. $N_\varepsilon(p) = \{q | dist(p,q) \le \varepsilon, q \in P\}$.

Some important concepts of DBSCAN are defined as follows.

Definition 1. A point $p \in P$ is a **core point** if it's ε-neighborhood satisfies $|N_\varepsilon(p)| \ge MinPts$.

Definition 2. A point $p \in P$ is **directly density-reachable** from a point $q \in P$ wrt. $\varepsilon, MinPts$ if q is a core point and $p \in N_\varepsilon(q)$.

Definition 3. A point $p \in P$ is **density-reachable** from a point $q \in P$ wrt. $\varepsilon, MinPts$ if there is a sequence of points $p_1, p_2, \ldots, p_z \in P$ such that $p_1 = q$, $p_z = p$, and p_i is directly density-reachable from p_{i-1}, where $i = 2, \ldots, z$.

Definition 4. A point $p \in P$ is **density-connected** to a point $q \in P$ wrt. $\varepsilon, MinPts$ if there is a point $b \in P$ such that p and q are density-reachable from b wrt. $\varepsilon, MinPts$.

Definition 5. A non-empty set C is a **cluster** wrt. $\varepsilon, MinPts$ if C satisfies the following conditions:

- If $p \in C$ and p is a core point, then all points density-reachable from p wrt. $\varepsilon, MinPts$ also belong to C. (Maximality)
- $\forall p, q \in C$, p is density-connected to q wrt. $\varepsilon, MinPts$. (Connectivity)

A cluster found by DBSCAN contains both core and non-core points. A non-core point is called a border point if it belongs to at least one cluster. And points that do not belong to any cluster are called noise points.

3.2. G13

For 2-dimensional data, G13 solves the exact DBSCAN with $O(n \log n)$ time complexity using the grid structure. The algorithm consists of four major steps. First, partition the feature space into equal-sized grids. Second, identify core points in the data set. Third, merge core points to form clusters. Lastly, assign non-core points to clusters according to density reachability.

In the first step, the feature space is divided into multiple grids of the same size with side length $\varepsilon/\sqrt{2}$, and then each point will be assigned to the grid it lies in. A grid is called an empty grid if there are no points inside it. Otherwise, it is a non-empty grid.

The second step is to identify all core points in the data set. A grid containing at least one core point is called a core grid. Recall that the side length of each grid is $\varepsilon/\sqrt{2}$, which ensures the distance between any two points in the same grid will not be greater than ε. Hence, if there are more than $MinPts$ points in the same grid, then all those points are core points. For a grid g_i with points less than $MinPts$, the algorithm checks each point $p \in g_i$ and determines whether it is a core point. Let $\mathcal{N}_\varepsilon(g_i)$ denote the neighboring grids of g_i, where $\mathcal{N}_\varepsilon(g_i) = \{g' | ditance(g', g_i) \le \varepsilon\}$ and $ditance(g', g_i)$ is the minimum distance between g' and g_i. To determine whether p is a core point or not, the algorithm detects its neighbors in the non-empty grids of $\mathcal{N}_\varepsilon(g_i)$.

The third step, namely the merging step, is to find the clusters formed by core points. In order to seek out clusters, we need to iterate through all core grids. Initially, each core grid can be treated as an individual cluster based on the fact that all points in the same core grid are density-reachable from each other. Then those core grids that are density-reachable from each other will be merged into one cluster. Let $G = (V, E)$ be a graph, where each vertex $v \in V$ corresponds to a core grid, and each edge $(g_i, g_j) \in E$ represents that g_i and g_j can be merged. Given two core grids g_i and g_j, the algorithm determines whether they can be merged by the following definition.

Definition 6. For two core grids g_i and g_j, they can be **merged** if and only if there are core points $p \in g_i$ and $q \in g_j$ such that $dist(p,q) \le \varepsilon$.

Given a core grid g_i, for each core grid $g_j \in \mathcal{N}_\varepsilon(g_i)$, we explain in detail how to verify whether G has an edge between g_i and g_j. For each core point $p \in g_i$, G13 uses the Voronoi diagram to find the core point $q \in g_j$ closest to p. If the distance between p and q is not greater than ε, then edge (g_i, g_j) can be added to G. After the graph G has been created, each connected component of G represents a cluster.

The last step is to assign non-core points to clusters according to density reachability. For each non-core point p, if there is at least one core point in p's ε-neighborhood, p is called a border point. Otherwise, it is a noise point.

As shown in Gunawan [5], the time complexity of the merging step is $O(n \log n)$ if the Voronoi diagram is used, and the rest of the algorithm's time complexity is $O(MinPts \cdot n)$. Hence, the overall complexity of G13 is $O(n \log n)$ which is dominated by the merging step.

3.3. ρ-approximate DBSCAN

ρ-approximate DBSCAN is an approximate algorithm that extends G13 to higher dimensions. To ensure the distance between any two points in the same grid is not greater than ε, the side length of the grid is set to $\varepsilon/\sqrt{d}$. In addition to the different settings of the side length of the grid, the rule of merging core grids is different as well. ρ-approximate DBSCAN uses a quadtree-like hierarchical grid structure for approximate range count query when merging core grids. Formally, we fix a core grid g_i and check each non-empty neighboring grid g_j of g_i. An edge (g_i, g_j) is added to G if there is a core point $p \in g_i$ and the approximate range count query on the core points of g_j returns yes. Gan and Tao [10] showed that the overall complexity of ρ-approximate DBSCAN is $O(n)$ in expectation, regardless of the constant dimension d, the constant approximation ratio ρ, and the parameter ε.

4. The proposed algorithm

In this section, we propose the algorithm which aims to solve the exact DBSCAN with time complexity almost linear to the data set size. Firstly, we present a new approach to construct grids. Secondly, we introduce grid tree which allows us to efficiently find the non-empty neighboring grids for each grid. Thirdly, we introduce a critical new technique to swiftly determine whether any two core grids should be merged. Finally, we combine the above techniques to obtain our algorithm called GriT-DBSCAN.

4.1. Constructing the grids

When constructing the grids, each dimension of the feature space is divided into equal-sized intervals of length $\varepsilon/\sqrt{d}$. Then, the feature space is divided into multiple equal-sized grids and each grid g_i is uniquely determined by a d-dimensional vector $(g_{i1}, g_{i2}, \ldots, g_{ij}, \ldots, g_{id}) \in N^d$, denoted as identifier. A point p lies in g_i if and only if the following equality holds:

$$g_{ij} = \left\lfloor \frac{p(j) - mn_j}{\varepsilon/\sqrt{d}} \right\rfloor \tag{1}$$

where $\lfloor \cdot \rfloor$ is the floor function, $p(j)$ is the coordinate of p in dimension j, and $mn_j = \min_{p \in P} p(j)$ for $j = 1, 2, \ldots, d$.

We can now cover how to construct grids. Firstly, the identifier of the grid each point $p \in P$ lies in is calculated by Eq. (1).

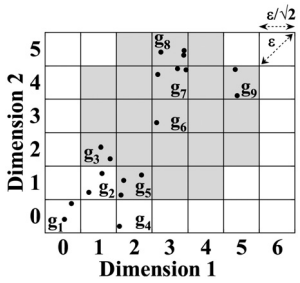

Fig. 1. Grids in 2-dimensional space ($\mathcal{N}_\varepsilon(g_6)$ is shown in gray).

Secondly, the n points are sorted according to their identifiers. Initially, the n points are sorted using counting sort [25] according to the value of their identifiers on dimension d. Then, the n points are sorted according to the value of their identifiers on dimension $d-1$. Repeat this process for each dimension. After sorting, the points in the same grid are placed adjacently.

Finally, we traverse the sorted points and find all non-empty grids, denoted as G_s. For any two grids $g_i, g_j \in G_s$ with $i < j$, there exists an integer $z \in [1, d)$ such that $g_{iz} < g_{jz}$ and $g_{iw} = g_{jw}$, for each $w \leq z-1$.

To illustrate, Fig. 1 shows the grid structure on a set with 19 points, where $\eta = 5$. There are 9 non-empty grids in the feature space, and $G_s = \{g_1, g_2, \ldots, g_9\}$. In addition, the neighboring grids of g_6 are shown in gray.

It's easy to verify that the second step runs in $O(d(n+\eta))$, where $\eta = \max\{g_{ij} | g_i \in G_s, 1 \leq j \leq d\}$ is a constant associated with ε, d, the maximum and minimum coordinates in each dimension. The rest of the algorithm's time complexity is $O(dn)$. Therefore, the overall complexity of constructing the grids is $O(d(n+\eta))$.

4.2. Indexing the grids

G13 and ρ-approximate DBSCAN significantly improve the performance of DBSCAN using the grid structure. However, for a given grid g_i, the number of neighboring grids of g_i increases exponentially with the dimension [11]. One important innovation of this paper is to introduce grid tree as a data structure to organize the non-empty grids G_s. Given a non-empty grid g_i, the grid tree enables us to find all non-empty grids in $\mathcal{N}_\varepsilon(g_i)$ more efficiently. We first describe the definition of the grid tree. Then, we introduce an algorithm for finding non-empty neighboring grids using the grid tree. Finally, we present theoretical evidence showing the grid tree is an effective structure for non-empty neighboring grids queries.

4.2.1. Grid tree

Grid tree is a tree-like structure built to organize non-empty grids. Denoted as T, a grid tree has $d+1$ levels. At the 1st level, there is a root node ($root$) which contains child pointers to child nodes. The jth child pointer of $root$ will be denoted as $CHILD_j(root)$. All nodes in the ith level, where $1 < i \leq d$, are called internal nodes. In addition to the child pointers, each internal node t contains one key denoted as $KEY(t)$, and a next pointer $NEXT(t)$ which is either null or points to another node in the same level. If the next pointer is not null, then we have $KEY(t) < KEY(NEXT(t))$. Each leaf node t is in the $(d+1)$th level, containing a key, a next pointer, and a pointer to a non-empty grid denoted as $GRID(t)$. For each path $t_{(1)}, t_{(2)}, \ldots, t_{(d+1)}$ from the root node to leaf node, where $t_{(1)}$ is the root node and $t_{(i+1)}$ is a child of $t_{(i)}$, $i = 1, 2, \ldots, d$, the identifier of the non-empty grid which $t_{(d+1)}$ points to is equal to $(KEY(t_{(2)}), KEY(t_{(3)}), \ldots, KEY(t_{(d+1)}))$.

Furthermore, for each node t in the grid tree, if it has more than $MinPts$ child nodes, then there are many items associated with t in the hash table h of T that can be used to identify which child node to visit at the next level when searching the non-empty neighboring grids for a particular grid. Let $\lceil \cdot \rceil$ denote the ceiling function. Each item of h is a $\{t, key\}-pt$ pair, where pt is the node with the smallest key among the child nodes of t whose keys are between $key - \lceil\sqrt{d}\rceil$ and $key + \lceil\sqrt{d}\rceil$.

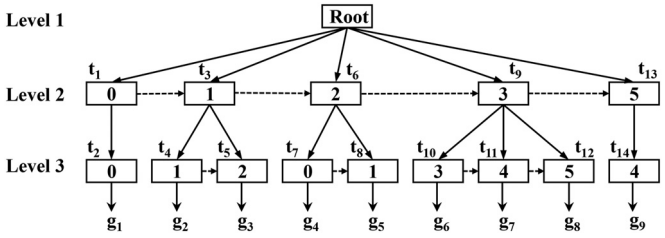

Fig. 2. Grid tree for the non-empty grids in Fig. 1.

To illustrate, Fig. 2 shows the grid tree for the non-empty grids in Fig. 1. Suppose that $MinPts = 3$. Then there are six items associated with the root node in h: $\{root, 0\}-t_1$, $\{root, 1\}-t_1$, $\{root, 2\}-t_1$, $\{root, 3\}-t_3$, $\{root, 4\}-t_6$, and $\{root, 5\}-t_9$.

4.2.2. Non-empty neighboring grids query

With a grid tree in place, we are now ready to describe the algorithm using the grid tree to identify all non-empty neighboring grids for each grid.

Given a grid g_i, to find its non-empty neighboring grids, the algorithm recursively traverses down the tree starting from the root node. The algorithm first finds all child nodes of the root node with keys between $g_{i1} - \lceil\sqrt{d}\rceil$ and $g_{i1} + \lceil\sqrt{d}\rceil$. If the root node has more than $MinPts$ child nodes, the item whose $\{t, key\}$ is equal to $\{root, g_{i1}\}$ in h is used to identify the child node with the smallest key that meets the condition, and then iteratively call $NEXT$ to find all nodes with keys between $g_{i1} - \lceil\sqrt{d}\rceil$ and $g_{i1} + \lceil\sqrt{d}\rceil$. Otherwise, these nodes are found by examining each child node of the root node. Denote these nodes as Φ. For each node $t \in \Phi$, the current $offset$ of t is $t.offset = \max\{|KEY(t) - g_{i1}| - 1, 0\}^2$. The offset of t indicates that the minimum distance between g_i and any grid in this subtree will not be less than $\sqrt{t.offset} \cdot \varepsilon/\sqrt{d}$. Therefore, all non-empty neighboring grids of g_i are in the subtrees of nodes in Φ. This is because, for each grid g_j not in the subtrees of the nodes in Φ, we have $g_{j1} - g_{i1} > \lceil\sqrt{d}\rceil$. It follows that the minimum distance between g_i and g_j is greater than ε ($(\lceil\sqrt{d}\rceil + 1) \cdot \varepsilon/\sqrt{d} > \varepsilon$). Therefore, there is no need to consider all nodes not in the subtrees of the nodes in Φ. As a result, a lot of redundant computations can be avoided. This significantly improves the performance of our algorithm.

After that, for each $t \in \Phi$, its child nodes with keys between $g_{i2} - \lceil\sqrt{d}\rceil$ and $g_{i2} + \lceil\sqrt{d}\rceil$ are found. For each such node t', its offset is calculated by

$$t'.offset = t.offset + \max\{|KEY(t) - g_{i2}| - 1, 0\}^2. \quad (2)$$

All these nodes will be stored in the set $temp$. Nodes in $temp$ with $offset$ greater than or equal to d will be further excluded. Then, Φ is updated to $temp$. This procedure is repeated until all nodes in Φ are leaf nodes.

For example, consider the query of $g_6 = (3, 3)$ in Fig. 2. Start from the root node; the algorithm first finds its child nodes with keys between 1 and 5, and the result is $\Phi = \{t_3, t_6, t_9, t_{13}\}$. Then for t_3, the algorithm first finds the child nodes of t_3 with keys between 1 and 5. Clearly, both child nodes of t_3 meet the condition. But $t_4.offset = 2$, so it is excluded. Other nodes in Φ are processed in a similar way. Finally, Φ is updated to $\{t_5, t_8, t_{11}, t_{12}, t_{14}\}$. Therefore, the non-empty neighboring grids of g_6 are g_3, g_5, g_7, g_8, g_9.

To sum up, we find all non-empty neighboring grids of g_i by calling $NeighboringGridsQuery(T, g_i)$, which is summarized in

Algorithm 1. We denote the non-empty neighboring grids of g_i as $Nei(g_i)$.

Algorithm 1 NeighboringGridsQuery.

Require: T: a grid tree; g_i: a grid.
Ensure: N: the non-empty neighboring grids of g_i.
1: $T.root.offset = 0$; $\Phi = \{T.root\}$; $temp = \emptyset$
2: **for** $j = 1 : d$ **do**
3: **for all** $nd \in \Phi$ **do**
4: $childNodes$ = all child nodes of nd with keys between $g_{ij} - \lceil\sqrt{d}\rceil$ and $g_{ij} + \lceil\sqrt{d}\rceil$
5: **for all** $child \in childNodes$ **do**
6: Calculate $child$'s offset by Eq. (2).
7: **if** $child.offset \geq d$ **then**
8: Remove $child$ from $childNodes$.
9: $temp = temp \cup childNodes$
10: $\Phi = temp$; $temp = \emptyset$
11: $N = \emptyset$
12: **for all** $nd \in \Phi$ **do**
13: $N = N \cup \{GRID(nd)\}$
14: Sort N by counting sort according to their offsets.
15: **return** N

It is worth noting that the non-empty neighboring grids are sorted in ascending order using counting sort according to their offset. This will significantly accelerate the performance of identifying core points. For one thing, a point's neighbors are more likely to be in the grid close to it. For another, given a point, once we confirm that it has more than $MinPts$ neighbors, it will be identified as a core point without finding all its neighbors. This saves a lot of unnecessary calculations.

4.2.3. Complexity analysis

A grid tree has $d+1$ levels, and each level contains at most $|G_s|$ nodes. In addition, the space complexity of the hash table is $O(d \cdot \sqrt{d} \cdot |G_s|) = O(d^{3/2} \cdot |G_s|)$. So the overall space complexity of grid tree is $O(d^{3/2} \cdot |G_s|)$. To build the grid tree, we only need to scan G_s once and update the hash table. Obviously, the expected complexity of updating the hash table is $O(d^{3/2} \cdot |G_s|)$. Hence, the expected time to build the grid tree is $O(d^{3/2} \cdot |G_s|)$.

To find the non-empty neighboring grids of a given grid, at the jth level, there are at most $(2\lceil\sqrt{d}\rceil + 1)^{j-1}$ nodes in Φ, which also implies that $|Nei(g_i)| \leq (2\lceil\sqrt{d}\rceil + 1)^d$. Using the hash table, finding the child nodes for any node in Φ (line 6 in Algorithm 1) runs in $O(2\lceil\sqrt{d}\rceil + 1)$ expected time. Moreover, sorting $Nei(g_i)$ runs in $O(|Nei(g_i)| + d)$ time because the offset of each node in $Nei(g_i)$ is less than d. It follows that Algorithm 1 runs in $O(\sum_{j=1}^{d+1}(2\lceil\sqrt{d}\rceil + 1)^{j-1} + |Nei(g_i)| + d) = O((2\lceil\sqrt{d}\rceil)^d)$ expected time. On the other hand, if each node in $nodes$ contains at least one neighboring grid of g_i in its subtree, then the running time will be $O(d \cdot |Nei(g_i)|)$.

4.3. Merging two grids

Apart from finding non-empty neighboring grids, the merging step is also time-consuming as mentioned in Section 3. In order to reduce the execution time of the merging step, we propose a new merging algorithm that can prune unnecessary distance calculations by making use of the spatial relationships among points. We also show that the proposed merging algorithm is theoretically effective.

4.3.1. Motivation

Given two core grids g_i and g_j, our goal is to determine whether they can be merged. Let s_i and s_j be the sets of core points in g_i and g_j, respectively. Let m_i and m_j be the number of points in s_i and s_j, respectively. Clearly, s_i and s_j are linearly separable, that is, there exists a hyperplane such that all points in s_i are below the hyperplane, and all points in s_j are above the hyperplane. Based on Definition 6, g_i and g_j can be merged if and only if the minimum distance between s_i and s_j is not greater than ε ($\min_{p\in s_i, q\in s_j} dist(p,q) \leq \varepsilon$).

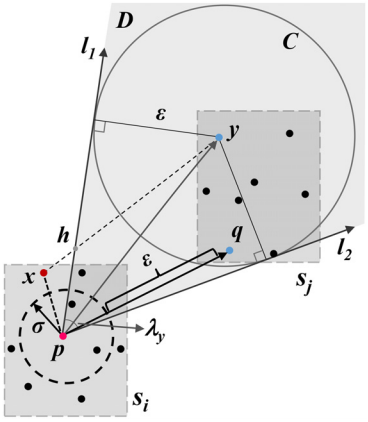

Fig. 3. The general ideas of our two pruning strategies.

A straightforward approach is to calculate the minimum distance between s_i and s_j using the brute force algorithm. However, it takes $O(m_i m_j)$ time to calculate the minimum distance between s_i and s_j with the brute force algorithm as it needs to calculate the distances between every pair of points $p \in s_i$ and $q \in s_j$. This is unacceptable when both m_i and m_j are large. In this paper, we propose a fast merging algorithm based on the spatial relationships between points to reduce unnecessary distance calculations.

4.3.2. Fast merging algorithm

Recall that our goal is to check whether two core grids can be merged. Here, we introduce a fast merging algorithm to efficiently solve the problem with fewer required distance calculations. We first introduce two pruning strategies founded on spatial relationships among points. Then, we introduce the merging algorithm in detail.

Without loss of generality, in Fig. 3 we take two linearly separable sets as an example to illustrate the general ideas of these two pruning strategies since the new merging algorithm can be generalized to determine whether the minimum distance between any two linearly separable sets is not greater than a certain threshold. Assume that we have already calculated the distance from point $p \in s_i$ to all points in s_j. The closest point to p is point $q \in s_j$, and $dist(p,q) > \varepsilon$. For convenience, let $\sigma = dist(p,q) - \varepsilon$. Based on the spatial relationships among points, we propose the following two pruning strategies used in our merging algorithm.

In fact, if p is far away from every point in s_j, the point close enough to p will be far away from every point in s_j as well. This motivates us to propose the first pruning strategy, namely, the triangle inequality pruning strategy, to prune trivial points. Here, a point x is a trivial point if the minimum distance between set $\{x\}$ and s_j is greater than ε. For each point $x \in s_i$, if $dist(x,p) < \sigma$, then x is a trivial point. The reasoning is that for every point $y \in s_j$, the distance between x and y satisfies

$$\begin{aligned} dist(x,y) &\geq dist(p,y) - dist(x,p) \\ &> dist(p,q) - \sigma \\ &= \varepsilon. \end{aligned} \tag{3}$$

The first inequality is based on the triangle inequality. As a result, all points inside the σ-ball centered at p can be pruned.

The second pruning strategy is the angle-based pruning strategy. As shown in Fig. 3, C is a circle with radius ε and y as its center. l_1 and l_2 are the two tangent lines of circle C passing through p. Evidently, the distance from y to any point on the tangent line is greater than or equal to ε. Let D denote the region surrounded by the two tangent lines l_1 and l_2. It is simple to verify that, for every point in $x \in s_i$, if it is outside D, the distance between x and y is greater than ε. The reasoning is that the line xy must intersect one of the tangent lines, and the distance from the intersection point to y is not less than ε. For example, in Fig. 3, the line segment xy intersects l_1 at h, it can be concluded that $dist(x, y)$ is greater than ε since $dist(x, y) = dist(x, h) + dist(y, h)$, $dist(y, h) \geq \varepsilon$, and $dist(x, h) > 0$.

For each point $y \in s_j$, the maximum angle of y used to determine which points in s_i are not in $N_\varepsilon(y)$ is defined as follows.

Definition 7. Given s_i, s_j, and $p \in s_i$, the **maximum angle** of $y \in s_j$ wrt. p is defined as

$$\lambda_y = \arcsin \frac{\varepsilon}{dist(p, y)} + \arccos \frac{\overrightarrow{pq} \cdot \overrightarrow{py}}{dist(p, q) \times dist(p, y)} \tag{4}$$

where $q = \arg\min_{y \in s_j} dist(p, y)$.

The following lemma shows that for $\forall x \in s_i$, if the angle between $\overrightarrow{pq}$ and $\overrightarrow{px}$ is greater than λ_y, then x is not in the ε-neighborhood of y.

Lemma 1. *Given s_i, s_j, and a point $p \in s_i$ such that $dist(p, q) > \varepsilon$, where $q = \arg\min_{z \in s_j} dist(p, z)$. For each $x \in s_i$, if the angle between $\overrightarrow{pq}$ and $\overrightarrow{px}$ is greater than λ_y, then*

$$dist(x, y) > \varepsilon. \tag{5}$$

Proof. For convenience, we use $\langle \overrightarrow{px}, \overrightarrow{pq} \rangle$ to represent the angle between $\overrightarrow{px}$ and $\overrightarrow{pq}$. Let $\gamma = \langle \overrightarrow{px}, \overrightarrow{pq} \rangle$, $\theta_1 = \langle \overrightarrow{pq}, \overrightarrow{py} \rangle$, and $\theta_2 = \langle \overrightarrow{px}, \overrightarrow{py} \rangle$. We only need to prove: if $\gamma > \lambda_y$, $dist(x, y) > \varepsilon$ holds for every $y \in s_2$. Given the corresponding triangle on the surface of the unit sphere centered at p, the lengths of its three sides are γ, θ_1, and θ_2, respectively. According to the spherical law of cosines [26], we have

$$\cos\gamma = \cos\theta_1 \cos\theta_2 + \sin\theta_1 \sin\theta_2 \cos\omega$$

where ω is the angle of the corner opposite to the side of length γ. Further, we have

$$\begin{aligned} \cos\gamma &\geq \cos\theta_1 \cos\theta_2 - \sin\theta_1 \sin\theta_2 \\ &= \cos(\theta_1 + \theta_2) \end{aligned} \tag{6}$$

since $0 \leq \omega \leq \pi$. Based on Eq. (6) and $0 \leq \gamma, \theta_1, \theta_2 \leq \pi$, we have

$$\begin{aligned} \theta_2 &\geq \gamma - \theta_1 \\ &> \lambda_y - \theta_1 \\ &= \arcsin \tfrac{\varepsilon}{dist(p,y)}. \end{aligned}$$

Then, the distance from y to x satisfies

$$\begin{aligned} dist(x, y) &\geq dist(p, y) \sin\theta_2 \\ &> \varepsilon. \end{aligned}$$

This completes the proof. □

Based on the above findings, we propose the angle-based pruning strategy to further prune trivial points. Let $\lambda = \max_{y \in s_j} \lambda_y$. Then, if the angle between $\overrightarrow{pq}$ and $\overrightarrow{px}$ is greater than λ, we can conclude that x is a trivial point without calculating the minimum distance between set $\{x\}$ and s_j.

According to the above two pruning strategies, for each point x in s_i, x is a trivial point if it lies in the σ-neighborhood of p or the angle between $\overrightarrow{pq}$ and $\overrightarrow{px}$ is greater than λ. Consider the sets

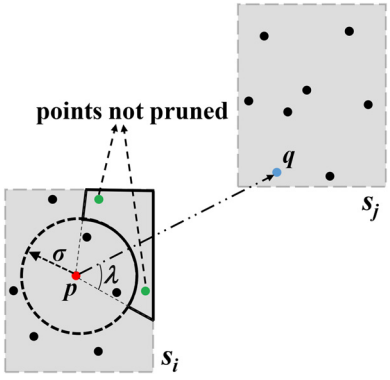

Fig. 4. An example of pruning trivial points in s_i.

in Fig. 3. The result of pruning trivial points in s_i is illustrated in Fig. 4. Points inside the region surrounded by solid lines won't be pruned. This is because the region is surrounded by vectors whose angle to $\overrightarrow{pq}$ equals λ and an arc whose radius is σ and center is p. It follows that a point x lying in this region is not a trivial point since $dist(p, x) \geq \sigma$ and the angle between $\overrightarrow{pq}$ and $\overrightarrow{px}$ is less than λ. Ultimately, seven of nine points in s_i have been pruned, with three pruned by the angle-based pruning strategy and four pruned by the triangle inequality pruning strategy. It is easier to determine whether the minimum distance between s_i and s_j is not greater than ε when there are only two points in s_i.

We can now state the fast merging algorithm. The fast merging algorithm uses the above two pruning strategies to iteratively remove trivial points when checking whether two grids can be merged, thus reducing unnecessary distance calculations. The pseudocode of the fast merging algorithm is summarized in Algorithm 2.

Algorithm 2 FastMerging.

Require: point sets s_i, s_j; ε.
Ensure: If $\min_{p \in s_i, q \in s_j} dist(p, q) \leq \varepsilon$, return yes. Otherwise, return no.

```
Randomly selected a point from s_i, denoted by p.
repeat
  q = arg min_{y∈s_j} dist(p, y); σ = dist(p, q) − ε
  if σ ≤ 0 then
    return yes
  else
    /*Remove trivial points in s_i.*/
    Calculate λ_y for each y ∈ s_j and let λ = max_{y∈s_j} λ_y.
    for all x ∈ s_i do
      if dist(x, p) < σ or λ < ⟨px, pq⟩ then
        remove x from s_i
  p = arg min_{x∈s_i} dist(x, q); σ = dist(p, q) − ε
  if σ ≤ 0 then
    return yes
  else
    /*Remove trivial points in s_j.*/
    Calculate λ_x for each x ∈ s_i and let λ = max_{x∈s_i} λ_x
    for all y ∈ s_j do
      if dist(y, q) < σ or λ < ⟨qy, qp⟩ then
        remove y from s_j
until |s_i| = 0 or |s_j| = 0
return no
```

Remark 1. The convergence of the fast merging algorithm is based on the following two facts. First, in each iteration, if $dist(p, q) \leq \varepsilon$,

the fast merging algorithm returns yes immediately. Otherwise, based on the triangle inequality pruning strategy, the points in s_i whose distance to p is less than $dist(p,q)-\varepsilon$ will be removed (i.e., points in a ball with radius greater than 0 will be excluded from consideration in the subsequent iterations). Combining this and the fact that all points in s_i are in a grid with edge length $\varepsilon/\sqrt{d}$, the fast merging algorithm will converge after a finite number of iterations (the upper bound on the number of iterations is analyzed in Section 4.3.3).

Lemma 2. *Algorithm 2 returns yes if and only if there are points $p \in s_i$ and $q \in s_j$ such that $dist(p,q) \le \varepsilon$.*

Proof. Obviously, Algorithm 2 returns yes only if there are core points p and q satisfying $dist(p,q) \le \varepsilon$. Therefore, if Algorithm 2 returns yes, grids g_i and g_j are density reachable from each other.

On the contrary, if Algorithm 2 returns no, there are no p and q such that $dist(p,q) \le \varepsilon$. Because Algorithm 2 returns no only when one of the sets is empty, and recall that only the trivial points will be pruned. □

4.3.3. Complexity analysis

Obviously, the complexity of the fast merging algorithm depends on two factors: (I) the number of distance calculations in each iteration, and (II) how many iterations it takes to terminate.

Let $r_{i,u}$ and $r_{j,u}$ be the number of points in s_i and s_j at the beginning of uth iteration, respectively. In each iteration, the algorithm calculates the distance $r_{i,u}+3r_{j,u}+2r_{i,u+1}$ times. Hence, in each iteration, the algorithm calculates the distance at most $3(m_i+m_j)$ times since at least one point is removed from each set in each iteration.

Then, the complexity of the fast merging algorithm satisfies

$$O(\sum_{u=1}^{\kappa_{g_ig_j}} d(r_{i,u}+3r_{j,u}+2r_{i,u+1})) \le O(\sum_{u=1}^{\kappa_{g_ig_j}} d(m_i+m_j)) = O(\kappa_{g_ig_j} d(m_i+m_j))$$

where $\kappa_{g_ig_j}$ is the number of iterations. The following lemma gives the upper bound on the number of iterations.

Lemma 3. *Let s_i' denote the points in s_i whose distances from each point in s_j are all greater than ϵ. If s_i' is empty, then $\kappa_{g_ig_j}=1$. Otherwise, let*

$$\tau = \min_{x\in s_i', y\in s_j} dist(x,y) - \epsilon.$$

Then $\kappa_{g_ig_j}$ is at most $V_g/V_{\tau/2}$, where $V_g=(\varepsilon/\sqrt{d}+\tau)^d$ and $V_{\tau/2}$ is the volume of a ball with radius $\tau/2$.

Proof. If s_i' is empty, then in the 1st iteration $p \notin s_i'$. Based on the definition of s_i', there is at least one point $q \in s_j$ such that $dist(p,q) \le \epsilon$. Therefore, the algorithm returns yes in the 1st iteration, and $\kappa_{g_ig_j}=1$ holds.

On the contrary, say s_i' is non-empty. If $p \in s_i'$, all points inside the σ-ball centered at p or whose angle exceeds λ will be removed. By the definition of τ, we have $\sigma \ge \tau$. Consequently, at the minimum, the points inside the τ-ball centered at p are removed. Then, the number of iterations $\kappa_{g_ig_j}$ is not greater than the number of leaders in s_i' using τ as parameters. According to the analysis of [19], the number of leaders is not greater than $V_g/V_{\tau/2}$. It follows that $\kappa_{g_ig_j} \le V_g/V_{\tau/2}$. This completes the proof. □

Note that the upper bound on $\kappa_{g_ig_j}$ is exponential to the dimension d. Nevertheless, there are two reasons that $\kappa_{g_ig_j}$ will not meet the upper bound. First, it should be noted that if $s_i \backslash s_i'$ is non-empty, the algorithm will return yes quickly. This is because q (p) is updated to the nearest point to p (q) in s_j (s_i), so the algorithm will obtain the local optimal value quickly and return yes immediately if the local optimal value is not greater than ε. Second, in each iteration the volume of the non-empty area from which the points will be removed is greater than the volume of a ball with radius τ, so the algorithm will terminate faster than the version described in the proof of Lemma 3 which only removes the points inside the τ-ball centered at p.

4.4. Overall algorithm

Now, we can introduce the GriT-DBSCAN algorithm with complexity almost linear to the data set size by combining the above techniques. Our algorithm consists of four steps as below.

First, the data set P is partitioned into several grids. All non-empty grids are organized in a grid tree. For each grid, its non-empty neighboring grids are found by Algorithm 1 and stored in a vector. Second, all core points in the data set are identified like G13. Third, core grids are merged to form clusters. In this step, we use Algorithm 2 to check whether two core grids can be merged. Finally, non-core points are identified as border points or noise points. In particular, for each non-core point $p \in P$ and the grid g within which p lies, we calculate the distances from p to all the core points in the non-empty neighboring grids of g. If q is a core point and $dist(p,q) \le \varepsilon$, then p is a border point and is assigned to the cluster of q. If there does not exist a core point q such that $dist(p,q) \le \varepsilon$, then p is a noise point.

The pseudocode of GriT-DBSCAN is summarized in Algorithm 3.

Algorithm 3 GriT-DBSCAN.

Require: point sets P; parameters: $MinPts, \varepsilon$.
Ensure: C: the clustering result.
/*step 1: partitioning*/
Constructing the grids: G_s.
Based on G_s, build the grid tree T.
For each non-empty grid $g_i \in G_s$, use Algorithm 1 to find $Nei(g_i)$.
/*step 2: identify core points*/
Identify all core points in the data set like G13.
/*step 3: merging*/
Each core grid is marked as *unclassified*.
for all core grid $g \in G_s$ **do**
  if g is *unclassified* **then**
    Mark g as *classified*
    $seeds = \{g\}$; $pos = 1$
    while $pos \le seeds.size()$ **do**
      $cur = seeds[pos]$
      for all *unclassified* core grid $g' \in Nei(cur)$ **do**
        s and s' are the sets of core points in cur and g', respectively.
        if $FastMerging(s,s') = yes$ **then**
          Mark g' as *classified*; $seeds = seeds \cup \{g'\}$
    All grids in *seeds* form a cluster.
/*step 4: assign non-core points*/
For each non-core point, check whether it is a noise or a border point.
return The clustering result.

For the correctness and the time complexity of GriT-DBSCAN, we present the following theorem.

Theorem 1. *Let $\kappa = \max\{\kappa_{g_i,g_j} | g_i, g_j \in G_s, g_j \in Nei(g_i)\}$ be the maximum number of iterations in the merging step. Then, GriT-DBSCAN runs in $O((2\lceil\sqrt{d}\rceil)^{d+2}\kappa n + d\eta)$ expected time, regardless of the value of MinPts. In addition, the clustering result of GriT-DBSCAN is consistent with that of DBSCAN.*

Proof. First, we prove the correctness of our algorithm, that is, the clustering result of GriT-DBSCAN is consistent with the result of DBSCAN. To this end, we only need to show that the clusters found by our algorithm conform to the requirements of Definition 5. Let c be an arbitrary cluster found by GriT-DBSCAN.

- For every core point $p \in c$ and any point $q \in P$ such that q is density-reachable from p wrt. $\varepsilon, MinPts$. By the definition of density-reachable, there is a sequence of points $p_1, p_2, \ldots, p_o \in P$ such that $p_1 = p$, $p_o = q$, and p_{i+1} is directly density-reachable from p_i for each $1 \le i \le o-1$. Furthermore, $p_1, p_2, \ldots, p_{o-1}$ are core points and $dist(p_i, p_{i+1}) \le \varepsilon$ for $1 \le i \le o-1$. Let $g_{(i)}$ denote the grid in which p_i lies, where $i = 1, 2, \ldots, o$. It follows from Lemma 2 and the definition of G that $g_{(i)}$ and $g_{(i+1)}$ must be in the same connected component for $1 \le i \le o-2$. Hence, $p_1, p_2, \ldots, p_{o-1}$ are in c. If p_o is a non-core point, then in the last step of GriT-DBSCAN, p_o will also be assigned to c. Otherwise, $g_{(o)}$ and $g_{(o-1)}$ must be in the same connected component of G. It thus follows that $p_o \in c$. Therefore, p and q are in the same cluster of GriT-DBSCAN.
- Let p and q be two arbitrary points in c. We will show that p is density-connected to q. If p is a non-core point, based on the last step of GriT-DBSCAN, there is a core point $p' \in c$ such that p is directly density-reachable from p'. Otherwise, we set $p' = p$. Similarly, let q' be a core point in c such that q is directly density-reachable from q'. Denote by $g_{p'}$ and $g_{q'}$ the grids covering p' and q', respectively. Since $p', q' \in c$, $g_{p'}$ and $g_{q'}$ must be in the same connected component of G. Then, there is a sequence $g_{(1)}, g_{(2)}, \ldots, g_{(o)}$ such that $g_{(1)} = g_{p'}, g_{(o)} = g_{q'}$. For each $i \in [1, o-1]$, $g_{(i)}$ and $g_{(i+1)}$ can be merged, which means that there are two core points $p_i^2 \in g_{(i)}$ and $p_{i+1}^1 \in g_{(i+1)}$ satisfying $dist(p_i^2, p_{i+1}^1) \le \varepsilon$ (Lemma 2). Moreover, we have $dist(p_i^1, p_i^2) \le \varepsilon$ since p_i^1, p_i^2 in the same grid. Therefore, p' and q' are density-reachable from each other wrt. $\varepsilon, MinPts$. It follows that p is density-connected to q wrt. $\varepsilon, MinPts$.

This completes the proof of correctness.

Next, we show that GriT-DBSCAN runs in $O((2\lceil\sqrt{d}\rceil)^{d+2}\kappa n + d\eta)$ expected time. Based on the analysis in Gunawan [5] and the fact that each grid has at most $O((2\lceil\sqrt{d}\rceil+1)^d) = O((2\lceil\sqrt{d}\rceil)^d)$ non-empty neighboring grids, the second and fourth steps of Algorithm 3 run in $O((2\lceil\sqrt{d}\rceil)^d n \cdot MinPts \cdot d)$.

It takes $O(d(n+\eta))$ time to partition the feature space. The expected time complexity of building the grid tree is $O(d^{3/2}|G_s|)$. The expected time complexity of finding the non-empty neighboring grids of a specific grid is $O((2\lceil\sqrt{d}\rceil)^d)$. It follows that finding the non-empty neighboring grids for all grids runs in $O(\sum_{i=1}^{|G_s|}(2\lceil\sqrt{d}\rceil)^d) = O((2\lceil\sqrt{d}\rceil)^d|G_s|)$ expected time. Overall, the first step runs in $O(dn + d\eta + (2\lceil\sqrt{d}\rceil)^d|G_s|) = O((2\lceil\sqrt{d}\rceil)^d n + d\eta)$ expected time since $|G_s| \le n$.

In the third step, for any two core grids g_i, g_j, where $g_j \in Nei(g_i)$, we need to check whether they can be merged. It follows from the analysis of Section 4.3.3 that the time complexity of this step is less than

$$\begin{aligned}\sum_{\substack{g_i\in G_s\\ g_j\in Nei(g_i)}} O\big(\kappa d(m_i+m_j)\big) &= \sum_{\substack{g_i\in G_s\\ g_j\in Nei(g_i)}} O(2\kappa d m_i)\\ &= \sum_{g_i\in G_s} O(\kappa d m_i)|Nei(g_i)|\\ &\le \sum_{g_i\in G_s} O(\kappa d m_i)\Big(2\lceil\sqrt{d}\rceil+1\Big)^d\\ &\le O\bigg(\Big(2\lceil\sqrt{d}\rceil\Big)^{d+2}\kappa n\bigg)\end{aligned}$$

where m_i and m_j are the number of core points in g_i and g_j, respectively.

In summary, GriT-DBSCAN runs in $O((2\lceil\sqrt{d}\rceil)^d n + d\eta + (2\lceil\sqrt{d}\rceil)^{d+2}\kappa n + (2\lceil\sqrt{d}\rceil)^{d+2} n \cdot MinPts) = O((2\lceil\sqrt{d}\rceil)^{d+2}\kappa n + d\eta)$ expected time, regardless of the value of $MinPts$. This completes the proof. □

Remark 2. Note that the expected time complexity of GriT-DBSCAN is exponential to d. Therefore, our algorithm is only suitable for low-dimensional data. Besides, in the experiments, $\kappa \le 11$ is much smaller than the number of data points, which verifies that the complexity of GriT-DBSCAN is almost linear in n.

Remark 3. Interestingly, a border point p may belong to more than one cluster. In practice, it is legitimate for p to be assigned to only one of these clusters [4]. Therefore, in reality, we do not find every core point q of p satisfying $dist(p, q) \le \varepsilon$ in the fourth step of GriT-DBSCAN. Instead, we compute the distances from p to the core points in the non-empty neighboring girds of g (the grid within which p lies) until we find a core point such that p can be assigned to the cluster of that core point.

5. Experiments

Extensive experiments are conducted to evaluate GriT-DBSCAN and its two variants defined in Section 5.2 by comparing them with existing algorithms. All the experiments are implemented on a machine equipped with a 2.5 GHz CPU and 16 GB memory using C++. The operating system is Windows 10 64-bit.

5.1. Data sets and parameter settings

In order to investigate the performance of the proposed algorithms, we conducted experiments on synthetic and real-world data sets. In all data sets, we normalize each column to the integer domain of $[0, 10^5]$.

The synthetic data sets are generated using the seed spreader (SS) generator proposed in Gan and Tao [10]. The seed spreader maintains a location when generating a synthetic data set. It generates points uniformly in the neighborhood of the current location and jumps to a random location with a certain probability. In addition, the seed spreader can generate data sets with either similar or variable density clusters. We denote data sets with similar density clusters and variable density clusters as SS-simden and SS-varden, respectively. For $d \in \{2, 3, 5, 7, 9\}$, we generate data sets with n varying from 0.1 million to 10 million. The default value of n is 2 million; see Table 1 for details.

Table 1
The synthetic data sets used in our experiments.

Parameter	Values
n	0.1 m, 0.5 m, 1 m, 2 m, 5 m, 10 m
d	2, 3, 5, 7, 9

We also use three real-world data sets to evaluate the performance of our algorithm. PAM4D is a 4-dimensional data set with 3,850,505 points, obtained by taking the first 4 principle components after performing PCA on the PAMAP2 data set available at the UCI machine learning archive [27]. Farm is a 5-dimensional data set with 3,627,086 points containing the VZ-features [28] of a satellite image of a farm in Saudi Arabia[2] House is a 7-dimensional data set with 2,049,280 points obtained from the UCI archive [27], excluding date and time information.

[2] https://www.satimagingcorp.com/gallery/ikonos/ikonos-tadco-farms-saudi-arabia.

For the synthetic data sets, we select the default values of ε and *MinPts* to be those that produce the correct clustering results. The default parameters for real-world data sets are similar to those found by Gan and Tao [10]. Unless specified otherwise, we use $\rho = 0.01$ in the appr-DBSCAN algorithm.

5.2. Experiments for $d \geq 3$

For $d \geq 3$, we compare the performance of the following algorithms:

- DBSCAN [4] is the original DBSCAN algorithm.
- gan-DBSCAN [10] is a grid-based exact DBSCAN for $d \geq 3$.
- appr-DBSCAN [10] is the state-of-the-art grid-based approximation DBSCAN reviewed in Section 3.3.
- IncAnyDBC [13] is the state-of-the-art ball-based exact DBSCAN algorithm. In particular, IncAnyDBC is a parallel incremental algorithm. In our experiments, we only use the sequential clustering part of IncAnyDBC.
- BLOCK-DBSCAN [16] is an approximation DBSCAN algorithm. It first uses $\epsilon/2$-norm ball to identify inner core blocks, outer core points, and border points. Second, it merges density-reachable inner core blocks into one cluster by an approximation algorithm. Then, each outer core point is merged into a cluster that it is density-reachable. Finally, border points will be assigned to corresponding clusters.
- BLOCK-DBSCAN-FM is a variant of our algorithm obtained by combining FastMerging with BLOCK-DBSCAN. BLOCK-DBSCAN-FM differs from BLOCK-DBSCAN only in the second step. BLOCK-DBSCAN-FM utilizes cover tree [29] to index touch inner core points for fast 2ϵ-neighborhood queries. Furthermore, the FastMerging algorithm is used to merge density-reachable inner core blocks. Based on Lemma 2, BLOCK-DBSCAN-FM is an exact DBSCAN algorithm.
- GriT-DBSCAN is the exact DBSCAN algorithm we proposed in Section 4.4.
- GriT-DBSCAN-LDF is another variant of our algorithm by incorporating heuristics. It is an exact DBSCAN algorithm that differs from GriT-DBSCAN only in the merging step. In GriT-DBSCAN-LDF, core grids are organized using a union-find data structure [30]. In addition, the core grids are sorted in ascending order according to the number of core points using counting sort. Then, these core girds are traversed in ascending order. Fix a core grid g_i. For each core grid $g_j \in Nei(g_i)$, we first check whether they are in the same set in the union-find data structure. If so, we do nothing. Otherwise, we perform a "union" operation if they can be merged. Finally, the core grids that belong to the same set in the union-find data structure belong to the same cluster. (The reason why these core girds are traversed in ascending order is that we first perform merging checks on low-density core grids such that the cluster is established soon. Consequently, high-density core grids can skip the merging checks since the cluster is already established [11]. This reduces redundant merging checks.)

We use the binary code which is written in C++ for gan-DBSCAN and appr-DBSCAN [31]. The C++ code of IncAnyDBC [13] is kindly provided by the original authors. Besides, we use the source code written in C++ for BLOCK-DBSCAN, which is publicly available on [32].

Influence of ε. The first set of experiments aims to determine how ε affects the running time of each algorithm. We fix *MinPts* to the default value and vary the parameter ε from 500 to 5000. If DBSCAN had no result in an experiment, then it did not terminate within 10,000 seconds in that experiment. Fig. 5 shows the running time as a function of ε, and we have the following observations. First, from Fig. 5, we can see that DBSCAN becomes slower as ε increases. This is because DBSCAN requires range queries for all points and the cost of each range query becomes more expensive as ε grows.

Second, for IncAnyDBC, BLOCK-DBSCAN, and BLOCK-DBSCAN-FM, there is a trade-off between the number of points filtered and the cost of each range query [13,16]: a large ε can filter a large number of points (pruning power), but the cost of each range query becomes more expensive as ε grows. From Fig. 5, the first effect dominates in most cases. Therefore, the running time tends to decrease as ε increases. Notably, BLOCK-DBSCAN-FM with our FastMerging algorithm for fast merging checks and the cover tree to index touch inner core points obtains accurate clustering results with almost the same running time as BLOCK-DBSCAN or even less. BLOCK-DBSCAN-FM's superiority to BLOCK-DBSCAN primarily depends on two factors. The first factor is whether ε is small or large. BLOCK-DBSCAN becomes slower as ε decreases because there are more touch inner core points, which increases the running time of using linear search to find the 2ε-neighborhood of each touch inner core point. On the contrary, BLOCK-DBSCAN-FM benefits from applying a cover tree index, thus a small ε has limited impact on its runtime. The other factor is whether the densities of clusters are similar or variable. For a data set with variable densities, the number of inner core blocks will be relatively large for the dense clusters; this makes BLOCK-DBSCAN slower. For the SS-varden-7D data set as an example, with $\varepsilon = 2000$, BLOCK-DBSCAN took 14.48 seconds and discovered 5096 clusters. BLOCK-DBSCAN-FM took 5.05 seconds to produce 1563 clusters like other exact DBSCAN algorithms. However, the superiority diminished when $\varepsilon = 4500$; BLOCK-DBSCAN took 6.65 seconds and discovered 1117 clusters, while BLOCK-DBSCAN-FM took 6.53 seconds to produce 15 clusters. The superiority also disappeared in data sets with similar density clusters. For the SS-simden-7D data set, with $\varepsilon = 2000$, BLOCK-DBSCAN took 5.71 seconds and discovered 166 clusters, while BLOCK-DBSCAN-FM took 7.26 seconds to produce 15 clusters.

Third, for gan-DBSCAN, appr-DBSCAN, GriT-DBSCAN, and GriT-DBSCAN-LDF, the performance improves as ε increases since there are fewer non-empty grids, which speeds up the merging step. It can be observed that GriT-DBSCAN and GriT-DBSCAN-LDF outperform gan-DBSCAN, appr-DBSCAN, and BLOCK-DBSCAN on all data sets. In addition, benefiting from the union-find data structure and the low density first traverse strategy for reducing redundant merging checks, GriT-DBSCAN-LDF significantly outperforms GriT-DBSCAN in most cases. However, the improvement of GriT-DBSCAN-LDF over GriT-DBSCAN is negligible when ε is small. The reason is that the number of non-empty grids is large when ε is small and most non-empty grids have one or two points, making the low density first traverse strategy ineffective. Alternatively, GriT-DBSCAN-LDF is more competitive with big ε.

Besides, as discussed in Section 4.4, GriT-DBSCAN is only suitable for low-dimensional data since its expected time complexity is exponential to d (as a variant of GriT-DBSCAN that incorporates heuristics, GriT-DBSCAN-LDF also has this feature). This can be observed in Fig. 5. When $d \leq 7$, GriT-DBSCAN and GriT-DBSCAN-LDF significantly outperform the six other algorithms for almost all values of ε. In contrast, IncAnyDBC, BLOCK-DBSCAN, and BLOCK-DBSCAN-FM become more competitive when $d > 7$. Particularly, BLOCK-DBSCAN-FM outperforms GriT-DBSCAN and GriT-DBSCAN-LDF for some ε values due to its pruning power.

Influence of *MinPts*. The next set of experiments aims to inspect how *MinPts* influences the running time of each algorithm. Therefore, we fix ε of each data set to the default value and vary *MinPts* from 10 to 100. Fig. 6 shows the effects of *MinPts* on the performance of different algorithms. GriT-DBSCAN, GriT-DBSCAN-LDF, gan-DBSCAN, and appr-DBSCAN become slower as *MinPts* increases. This is because the running time to identify all core points is $O((2\lceil\sqrt{d}\rceil)^{d+2}n \cdot MinPts)$. Generally, IncAnyDBC, BLOCK-DBSCAN,

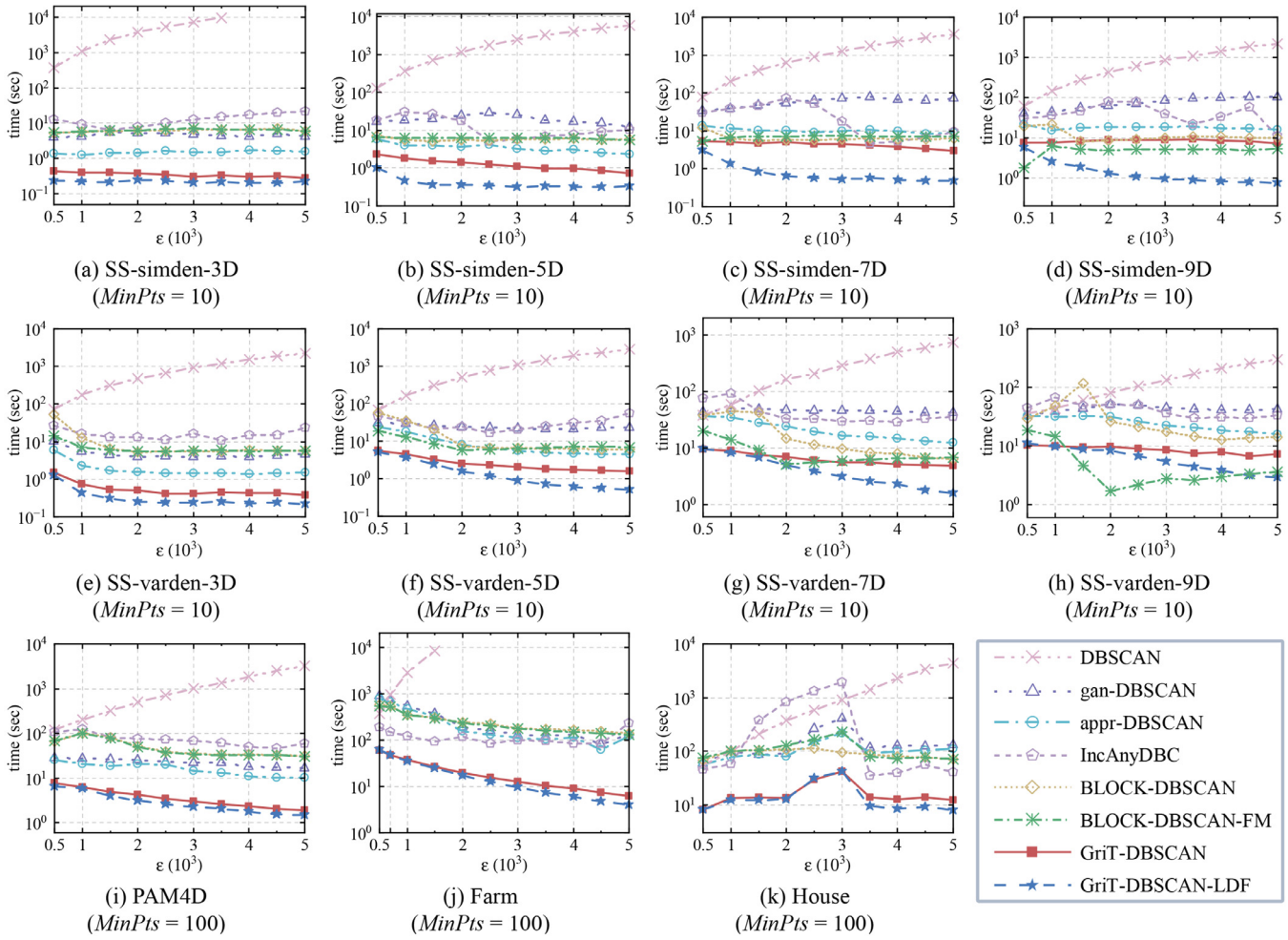

Fig. 5. Running time vs. ε.

and BLOCK-DBSCAN-FM become slower as *MinPts* increases. However, as illustrated in Fig. 6, the influence of *MinPts* is limited. In addition, GriT-DBSCAN and GriT-DBSCAN-LDF outperform DBSCAN, IncAnyDBC, gan-DBSCAN, appr-DBSCAN, and BLOCK-DBSCAN on all data sets. For low-dimensional data (i.e., $d \leq 7$), BLOCK-DBSCAN-FM is slower than GriT-DBSCAN and GriT-DBSCAN-LDF. For high-dimensional data (i.e., $d > 7$), BLOCK-DBSCAN-FM is comparable to or better than GriT-DBSCAN and GriT-DBSCAN-LDF. The reason is that the time complexity of both GriT-DBSCAN and GriT-DBSCAN-LDF are exponential to d, and their performance decreases as d grows.

Scalability with n. In the last set of experiments, we investigate the scalability of each algorithm with n using the synthetic data sets. To this end, we vary the number of points from 100,000 to 10 million. Other parameters are given their default values. The resulting running times are presented against n in Fig. 7. It can be seen that GriT-DBSCAN-LDF consistently outperforms all other algorithms. Again, BLOCK-DBSCAN and BLOCK-DBSCAN-FM become competitive as d increases, and BLOCK-DBSCAN is considerably slower than BLOCK-DBSCAN-FM on data sets with variable density clusters, which confirms our analysis in the first set of experiments.

5.3. Experiments for $d = 2$

Next, we perform a set of experiments to compare the efficiency of our algorithms with existing algorithms in $d = 2$. In particular, we compare the performance of our algorithms with DBSCAN [4], IncAnyDBC [13], BLOCK-DBSCAN [16], Wavefront [10], and G13 [5]. For Wavefront and G13, we use the binary code written in C++ that is publicly available on [31].

Influence of ε. We compare the performance of the six algorithms by varying ε. As shown in Fig. 8, the performance of all algorithms except DBSCAN improves with the growth of ε. DBSCAN becomes slower as ε increases because the cost of each range query gets more expensive. In addition, GriT-DBSCAN and GriT-DBSCAN-LDF outperform the other six algorithms in all cases, and Wavefront is comparable to BLOCK-DBSCAN and BLOCK-DBSCAN-FM.

Influence of *MinPts*. We investigate the influence of *MinPts* on the running time of each algorithm. Fig. 9 shows the results. The relative superiorities of all algorithms remain unchanged.

Scalability with n. We vary the number of points in the data sets to examine how each algorithm scales with n. The results are shown in Fig. 10. It can be seen that the relative superiority of all algorithms remain the same. However, in this set of experiments, BLOCK-DBSCAN is considerably slower than BLOCK-DBSCAN-FM on SS-simden-2D. The reason is that a small ε leads to a large number of inner core blocks, which makes the cost of forming clusters expensive. BLOCK-DBSCAN-FM uses cover tree to index touch inner core points, thus a small ε has little impact on its performance.

5.4. Efficiency of grid tree

To demonstrate the efficiency of grid tree, experiments are conducted on the three real-world data sets. In this set of experiments, we compare the running time of grid tree and R-tree as neighbor-

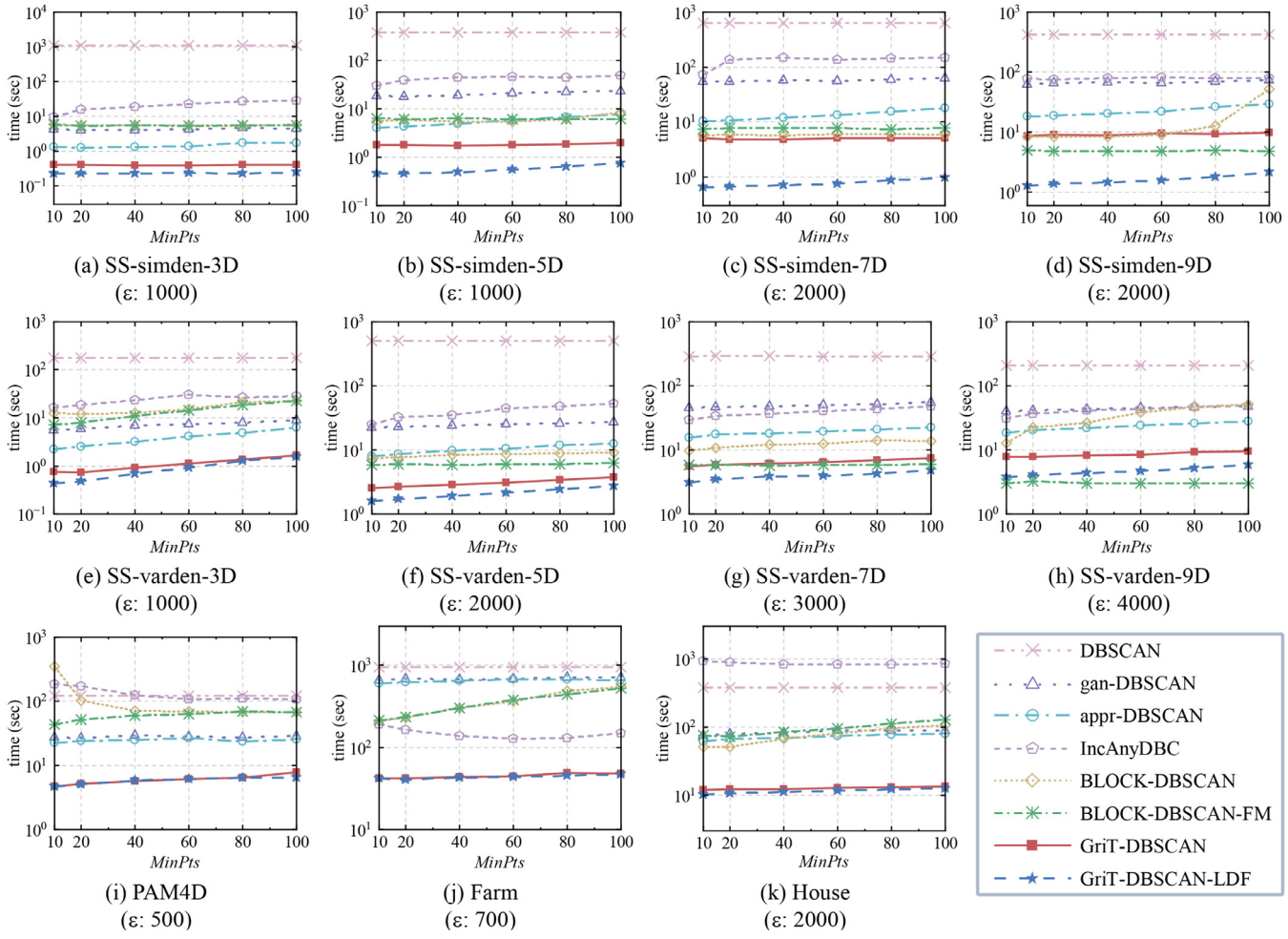

Fig. 6. Running time vs. *MinPts*.

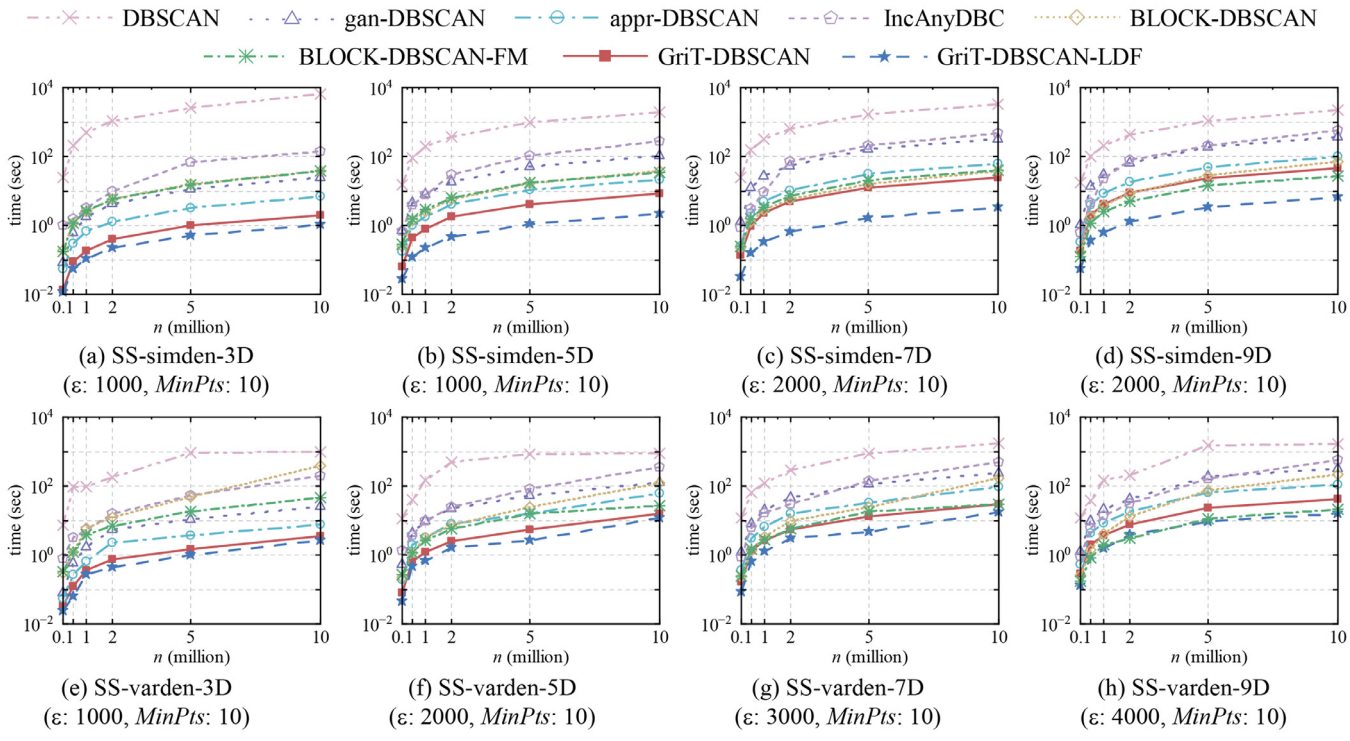

Fig. 7. Running time vs. *n*.

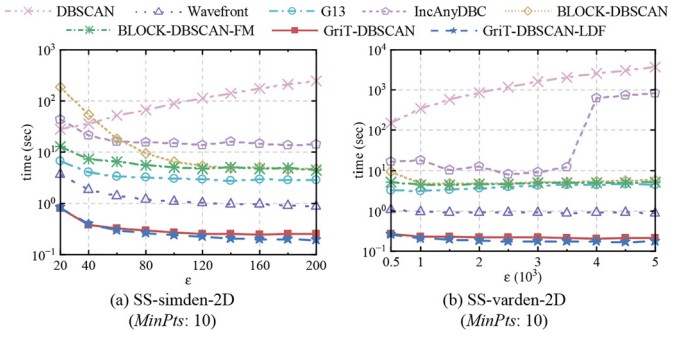

Fig. 8. Running time vs. ε.

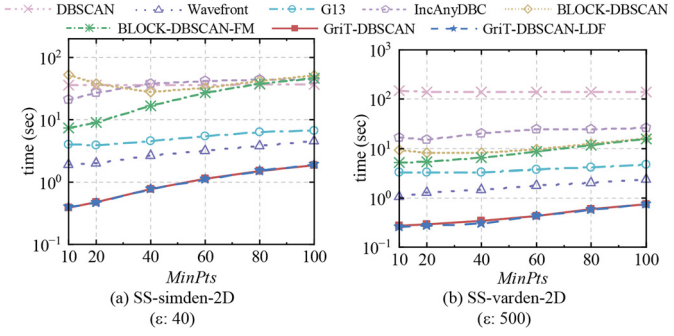

Fig. 9. Running time vs. *MinPts*.

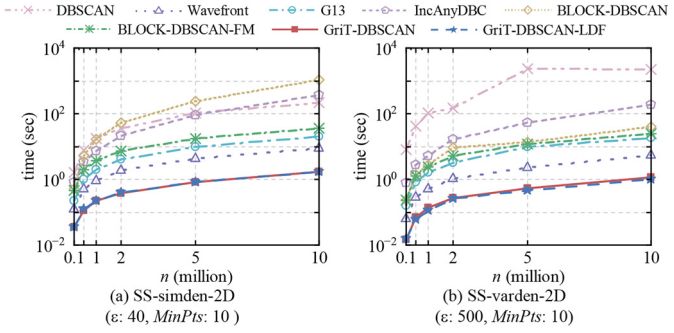

Fig. 10. Running time vs. *n*.

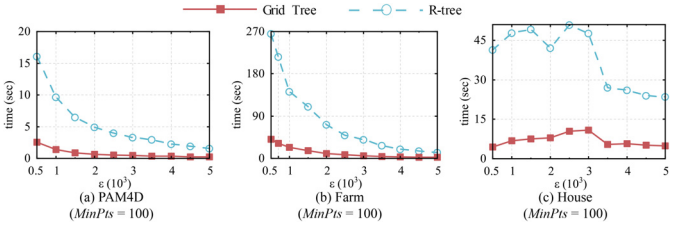

Fig. 11. The efficiency of grid tree.

ing grid query techniques by varying ε from 500 to 5000 and fixing *MinPts* to the default value. The results are shown in Fig. 11. In general, following an increase in ε, both grid tree and R-tree become faster. The reason is that the number of non-empty grids decreases with the increase of ε, resulting in fewer neighboring grid queries. On the other hand, the average number of non-empty neighboring grids increases with larger ε, making the neighboring grid query more expensive, thereby increasing the running time. Specifically, when ε is less than 3000, both grid tree and R-tree become slower with the increase of ε on the House data set. In addition, it is obvious from Fig. 11 that grid tree significantly outperforms R-tree on all data sets. Therefore, we can conclude that grid tree clearly speeds up the neighboring grid query.

Table 2
The accuracy of GriT-DBSCAN, GriT-DBSCAN-LDF, and BLOCK-DBSCAN-FM on the House data set.

[ε, *MinPts*]	BLOCK-DBSCAN-FM	GriT-DBSCAN	GriT-DBSCAN-LDF
[500, 10]	1.00	1.00	1.00
[1000, 10]	1.00	1.00	1.00
[2000, 10]	1.00	1.00	1.00
[2000, 20]	1.00	1.00	1.00
[2000, 40]	1.00	1.00	1.00
[2000, 80]	1.00	1.00	1.00
[2000, 100]	1.00	1.00	1.00
[3000, 10]	1.00	1.00	1.00
[4000, 10]	1.00	1.00	1.00
[5000, 10]	1.00	1.00	1.00

5.5. Correctness of the proposed algorithms

This subsection provides empirical evidence that GriT-DBSCAN and its two variants are exact DBSCAN algorithms. For this purpose, we calculate the accuracy of GriT-DBSCAN and its two variants using the clustering results obtained by the original DBSCAN algorithm [4] as the ground truth. Similar to the original DBSCAN algorithm, in GriT-DBSCAN and its two variants, each border point is assigned to the cluster of one of the core points in its ε-neighborhood. Some border points may be labeled differently depending on the order in which the points are traversed. Therefore, the accuracy is calculated by comparing the labels of core and noise points only. Table 2 shows the correctness of GriT-DBSCAN and its two variants under different ε and *MinPts* on the House data set.

6. Conclusions

In this paper, we introduce a new exact DBSCAN algorithm with complexity almost linear to the number of data points, called GriT-DBSCAN. The key idea of GriT-DBSCAN is to utilize the spatial relationships among points to efficiently determine whether two core grids can be merged in the merging step. More specifically, when judging whether two core grids can be merged, the trivial points in each grid are iteratively removed through the triangle inequality and the angle information, so as to prune unnecessary distance calculations. In addition, we introduce grid tree to organize non-empty grids and an algorithm using it for efficient non-empty neighboring grids queries. We prove theoretically that GriT-DBSCAN presents excellent improvement in terms of computational efficiency.

We also obtain two variants of GriT-DBSCAN, namely, GriT-DBSCAN-LDF and BLOCK-DBSCAN-FM. To further improve the performance of GriT-DBSCAN, GriT-DBSCAN-LDF incorporates the union-find data structure and the low density first traverse strategy to reduce redundant merging checks: once two core grids are in the same set, it is unnecessary to check whether they can be merged. By combining Algorithm 2 with BLOCK-DBSCAN, we obtain the second variant BLOCK-DBSCAN-FM. Due to the accuracy and efficiency of Algorithm 2, BLOCK-DBSCAN-FM obtains accurate clustering results with almost the same running time as BLOCK-DBSCAN or even less.

We conduct extensive experiments to evaluate the performance of GriT-DBSCAN and its two variants. The results demonstrate that our algorithms are more efficient than existing algorithms.

However, there are some limitations of the proposed algorithms that require further investigation. For instance, GriT-DBSCAN and GriT-DBSCAN-LDF are only suitable for low-dimensional data be-

cause their time complexities are both exponential to d. Although BLOCK-DBSCAN-FM is more scalable in terms of dimensionality, its performance is sensitive to the two parameters (i.e., ε and *MinPts*). Therefore, it will be useful in future to research and develop an algorithm that can handle large databases, is scalable in terms of dimensionality, and whose performance is insensitive to these two parameters, which will greatly increase the utility of DBSCAN in the field of pattern recognition.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

The datasets used in this paper are easily accessible, as described in the paper.

Acknowledgments

We would like to thank the Editors and Reviewers very much for their insightful and constructive comments which have significantly helped us to improve the presentation of the manuscript.

References

[1] H. Chen, T. Xie, M. Liang, W. Liu, A local tangent plane distance-based approach to 3Dpoint cloud segmentation via clustering, Pattern Recognit. 137 (2023) 109307.
[2] R. Janani, S. Vijayarani, Text document clustering using spectral clustering algorithm with particle swarm optimization, Expert Syst. Appl. 134 (2019) 192–200.
[3] J. Yin, S. Zhang, J. Xie, Z. Ma, J. Guo, Unsupervised person re-identification via simultaneous clustering and mask prediction, Pattern Recognit. 126 (2022) 108568.
[4] M. Ester, H.P. Kriegel, J. Sander, X. Xu, A density-based algorithm for discovering clusters in large spatial databases with noise, in: Proceedings of the 2nd ACM International Conference on Knowledge Discovery and Data Mining, 1996, pp. 226–231.
[5] A. Gunawan, A Faster Algorithm for DBSCAN, Technische Universiteit Eindhoven, 2013 Master's thesis.
[6] J. Gan, Y. Tao, DBSCAN revisited: mis-claim, un-fixability, and approximation, in: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, 2015, pp. 519–530.
[7] Y. Chen, S. Tang, N. Bouguila, C. Wang, J. Du, H. Li, A fast clustering algorithm based on pruning unnecessary distance computations in DBSCAN for high-dimensional data, Pattern Recognit. 83 (2018) 375–387.
[8] B. Borah, D.K. Bhattacharyya, An improved sampling-based DBSCAN for large spatial databases, in: Proceedings of the 2004 International Conferfence on Intelligent Sensing and Information Processing, 2004, pp. 92–96.
[9] S. Mahran, K. Mahar, Using grid for accelerating density-based clustering, in: Proceedings of the 2008 IEEE International Conference on Computer and Information Technology, 2008, pp. 35–40.
[10] J. Gan, Y. Tao, On the hardness and approximation of euclidean DBSCAN, ACM Trans. Database Syst. 42 (3) (2017) 1–45.
[11] T. Boonchoo, X. Ao, Y. Liu, W. Zhao, F. Zhuang, Q. He, Grid-based DBSCAN: indexing and inference, Pattern Recognit. 90 (2019) 271–284.
[12] S.T. Mai, I. Assent, M. Storgaard, AnyDBC: an efficient anytime density-based clustering algorithm for very large complex datasets, in: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2016, pp. 1025–1034.
[13] S.T. Mai, J. Jacobsen, S. Amer-Yahia, I.T.A. Spence, N.-P. Tran, I. Assent, Q.V.H. Nguyen, Incremental density-based clustering on multicore processors, IEEE Trans. Pattern Anal. Mach. Intell. 44 (3) (2020) 1338–1356.
[14] K.M. Kumar, A.R.M. Reddy, A fast DBSCAN clustering algorithm by accelerating neighbor searching using groups method, Pattern Recognit. 58 (2016) 39–48.
[15] Y. Chen, L. Zhou, S. Pei, Z. Yu, Y. Chen, X. Liu, J. Du, N. Xiong, KNN-BLOCK DBSCAN: fast clustering for large-scale data, IEEE Trans. Syst., Man, Cybern. 51 (6) (2019) 3939–3953.
[16] Y. Chen, L. Zhou, N. Bouguila, C. Wang, Y. Chen, J. Du, BLOCK-DBSCAN: fast clustering for large scale data, Pattern Recognit. 109 (2021) 107624.
[17] S. Zhou, A. Zhou, J. Cao, J. Wen, Y. Fan, Y. Hu, Combining sampling technique with DBSCAN algorithm for clustering large spatial databases, in: Proceedings of the 2000 Pacific-Asia Conference on Knowledge Discovery and Data Mining, 2000, pp. 169–172.
[18] P. Viswanath, R. Pinkesh, *l*-DBSCAN: a fast hybrid density based clustering method, in: Proceedings of the18th International Conference on Pattern Recognition, 2006, pp. 912–915.
[19] P. Viswanath, V.S. Babu, Rough-DBSCAN: a fast hybrid density based clustering method for large data sets, Pattern Recognit. Lett. 30 (16) (2009) 1477–1488.
[20] J.A. Hartigan, Clustering Algorithms, John Wiley & Sons, 1975.
[21] B. Liu, A fast density-based clustering algorithm for large databases, in: Proceedings of the 2006 International Conference on Machine Learning and Cybernetics, 2006, pp. 996–1000.
[22] J. Jang, H. Jiang, DBSCAN++: towards fast and scalable density clustering, in: International Conference on Machine Learning, 2019, pp. 3019–3029.
[23] T.F. Gonzalez, Clustering to minimize the maximum intercluster distance, Theor. Comput. Sci. 38 (1985) 293–306.
[24] H. Jiang, J. Jang, J. Lacki, Faster DBSCAN via subsampled similarity queries, Adv. Neural Inf. Process. Syst. 33 (2020) 22 407–22 419.
[25] D.E. Knuth, The Art of Computer Programming, Volume 3: Sorting and Searching, third ed., Addison-Wesley, 1997.
[26] I. Todhunter, Spherical Trigonometry, Macmillan, 1863.
[27] D. Dua, C. Graff, UCI machine learning repository, 2017. [Online]. Available: http://archive.ics.uci.edu/ml.
[28] M. Varma, A. Zisserman, Texture classification: are filter banks necessary? in: Proceedings of the 2003 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, vol. 2, 2003, pp. II–691.
[29] A. Beygelzimer, S. Kakade, J. Langford, Cover trees for nearest neighbor, in: Proceedings of the 23rd International Conference on Machine Learning, 2006, pp. 97–104.
[30] R.E. Tarjan, A class of algorithms which require nonlinear time to maintain disjoint sets, J. Comput. Syst. Sci. 18 (2) (1979) 110–127.
[31] J. Gan, APPROXIMATE DBSCAN, [Online]. Available: http://sites.google.com/view/approxdbscan
[32] Y. Chen, BLOCK-DBSCAN, [Online]. Available: https://github.com/XFastDataLab/BLOCK-DBSCAN

Xiaogang Huang is currently a Ph.D. Candidate in the School of Statistics, Southwestern University of Finance and Economics, Chengdu, China. His research interests include data mining, especially data mining algorithms for large datasets, and data management.

Tiefeng Ma received his Ph.D. degree in probability theory and mathematical statistics from Beijing University of Technology, Beijing, China, in 2008. He is currently a professor in the School of Statistics, Southwestern University of Finance and Economics, Chengdu, China. His research interests include data mining, clustering, change point, power data analysis. He has published more than 50 journal and conference papers.

Conan Liu is an Actuarial Studies Co-op Scholar with the University of New South Wales Business School, Sydney, Australia. His research interests lie within ethical artificial intelligence for insurance, business analytics, and data science.

Shuangzhe Liu is currently a professor with the University of Canberra, Canberra, Australia. His research interests include matrix differential calculus, multivariate analysis, and machine learning.