Pattern Recognition 90 (2019) 271–284

Contents lists available at ScienceDirect

Pattern Recognition

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

Grid-based DBSCAN: Indexing and inference

Thapana Boonchoo[a,b], Xiang Ao[a,b,*], Yang Liu[a,b], Weizhong Zhao[c], Fuzhen Zhuang[a,b], Qing He[a,b]

[a] Key Lab of Intelligent Information Processing of Chinese Academy of Sciences (CAS), Institute of Computing Technology, CAS, Beijing 100190, China
[b] University of Chinese Academy of Sciences, Beijing 100049, China
[c] School of Computer, Central China Normal University, Wuhan, China and Hubei Key Laboratory of Artificial Intelligence and Smart Learning, Central China Normal University, Wuhan, China

ARTICLE INFO

Article history:
Received 26 April 2018
Revised 3 December 2018
Accepted 24 January 2019
Available online 28 January 2019

Keywords:
Density-based clustering
Grid-based DBSCAN
Union-find algorithm

ABSTRACT

DBSCAN is one of clustering algorithms which can report arbitrarily-shaped clusters and noises without requiring the number of clusters as a parameter (unlike the other clustering algorithms, *k*-means, for example). Because the running time of DBSCAN has quadratic order of growth, i.e. $O(n^2)$, research studies on improving its performance have been received a considerable amount of attention for decades. Grid-based DBSCAN is a well-developed algorithm whose complexity is improved to $O(n\log n)$ in 2D space, while requiring $\Omega(n^{4/3})$ to solve when dimension ≥ 3. However, we find that Grid-based DBSCAN suffers from two problems: neighbour explosion and redundancies in merging, which make the algorithms infeasible in high dimensional space. In this paper we first propose a novel algorithm called GDCF which utilizes bitmap indexing to support efficient neighbour grid queries. Second, based on the concept of union-find algorithm we devise a forest-like structure, called cluster forest, to alleviate the redundancies in the merging. Moreover, we find that running the cluster forest in different orders can lead to a different number of merging operations needed to perform in the merging step. We propose to perform the merging step in a uniform random order to optimize the number of merging operations. However, for high-density database, a bottleneck could be occurred, we further propose a low-density-first order to alleviate this bottleneck. The experiments resulted on both real-world and synthetic datasets demonstrate that the proposed algorithm outperforms the state-of-the-art exact/approximate DBSCAN and suggests a good scalability.

1. Introduction

Clustering which is widely used for data mining and knowledge discovery has been intensively studied for decades. In contrast to supervised approaches, e.g., classification, clustering is an unsupervised technique which does not rely on any prior knowledge or ground truth of the data. Specifically, it attempts to group objects such that the similarities among objects in the same clusters and the differences between individual clusters are maximized. The clustering plays a crucial role in the domain of pattern recognition, e.g. image analysis [1,2], human behavior analysis [3], document analysis[4], etc [5].

A variety of clustering methods [6–14] have been proposed to perform clustering tasks. DBSCAN [9] (Density Based Spatial Clustering of Applications with Noise), as a representative density-based approach [6,9,15–17], is one of the most widely used algorithms since it performs well in discovering arbitrarily-shaped clusters and is robust to outliers (over methods such as *k*-means [14] which typically returns ball-like clusters). Many variants of DBSCAN can be found in the literature [15,16,18–27]. Considering Fig. 1, first two database examples taken from the original paper of DBSCAN [9] contain four snake-shaped clusters and three arbitrary-shaped clusters with a number of noises, while the rightmost database contains six clusters amid more complex noises. Two parameters, namely ε (a positive real number) and *MinPTS* (a natural number), are required in DBSCAN. ε denotes the maximum distance of one object to its neighbours, while *MinPTS* defines a density threshold of a given object. In other words, considering a *d*-dimensional object *p*, a *d*-ball area centered at object *p* with radius ε is considered as dense if it covers at least *MinPTS* objects. The objects within the *d*-ball area are included in the same cluster as the object *p*. Furthermore, clusters can be merged together if the centered objects can be added other clusters. The

* Corresponding author.
E-mail addresses: thapana@ict.ac.cn (T. Boonchoo), aoxiang@ict.ac.cn (X. Ao), liuyang17z@ict.ac.cn (Y. Liu), zhaowz81@163.com (W. Zhao), zhuangfuzhen@ict.ac.cn (F. Zhuang), heqing@ict.ac.cn (Q. He).

https://doi.org/10.1016/j.patcog.2019.01.034
0031-3203/

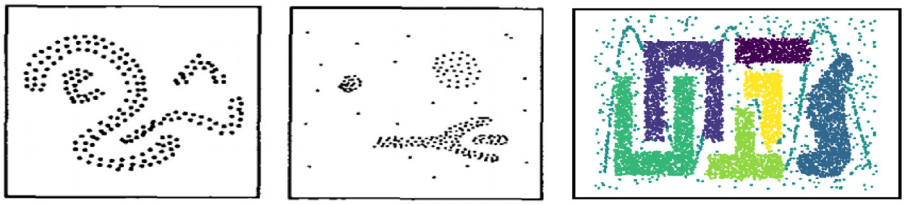


Fig. 1. 2D database examples.

merging will be carried out to the fullest extent until no more clusters can be merged. DBSCAN suffers from quadratic order of growth since it needs to perform $n\varepsilon$-queries and cluster labeling propagation for all the objects. However, even data indexing, such as kd-tree [28] or r*-tree [29] is utilized to perform ε-queries, the complexity of DBSCAN is still $O(n^2)$ in the worst case. Research efforts have been devoted to improving the efficiency of DBSCAN since it requires a high complexity.

Hence, there are several kinds of improved algorithms in the literature [19,30–38]. For example, [33] proposed a Grid-based DBSCAN to improve the complexity of DBSCAN from $O(n^2)$ to $O(n\log n)$ for 2D data by exploiting the grid topology, and [35] extended the Grid-based DBSCAN to higher dimension, and further proposed $\rho-$approximate which is an approximate algorithm for DBSCAN. [31] proposed a novel graph-based indexing structure to alleviate the bottleneck of neighbour search operations. Recently, [30] assumed that any nearby objects should have similar neighbouring objects, and proposed NQ-DBSCAN to speed up the algorithm by pruning the distance calculation to expand the cluster.

Grid-based DBSCAN is an exact algorithm that can produce the same clustering result as the original DBSCAN. The idea behind Grid-based DBSCAN is to divide the whole dataset into equal-sized square-shaped grids with the side width of $\varepsilon/\sqrt{d}$, where d denotes the data dimension. Therefore, any two objects in the same grid are within distance ε from each other. Then, Grid-based DBSCAN performs clustering based on neighbour grid query and merging them instead of ε-range queries and cluster labeling propagation.

However, we find that Grid-based DBSCAN algorithms still suffer from the following two problems. First, the number of neighbour grids increases exponentially with the number of dimensions. Specifically, the number of neighbour grids of a given grid will be $O((2\lceil\sqrt{d}\rceil+1)^d)$ in the worst case. We provide a formal proof for it in Lemma 1 in Section 4. Although $O((2\lceil\sqrt{d}\rceil+1)^d)$ is usually considered as a constant with regard to n, it may have a significant impact on the overall performance as the data dimension increases. For example, such number will be more than 10^{20} when $d=20$ such that we cannot simply neglect it. We name this problem as *neighbour explosion*. Second, we observe *symmetry* and *transitivity* in the merging of neighbour grids which enable finer management strategies for such process. However, the existing Grid-based DBSCAN algorithms focus on utilizing efficient solutions to perform the merging rather than optimizing merging management strategies. Recall that the number of neighbour grids increases exponentially with regard to data dimension. Hence we argue that an effective merging management of neighbour grids is in urgent need especially when the data dimension increases because generally we need to check every neighbour grid of a given grid to decide whether they can be merged or not. Hereafter, we will refer to the Grid-based DBSCAN algorithms as *grid-based algorithms*, unless explicitly stated otherwise.

In this paper, we aim at alleviating the above-mentioned drawbacks of grid-based algorithms and propose an efficient approach, named GDCF (**G**rid-based **D**BSCAN with **C**luster **F**orest). The approach utilizes bitmap-like structure called HyperGrid Bitmap (HGB for short) to index non-empty grids in multi-dimensional space so that neighbour grid query can be performed efficiently. For the neighbour grid merging, we follow the concept of union-find algorithm and devise a forest-like structure called *cluster forest* to maintain the cluster information for pruning unnecessary merging computations. As a result, the algorithm derives a significant performance improvement. We also prove the correctness of the GDCF. Moreover, we find that running the cluster forest in different orders can lead to a different number of merging operations needed to perform in the merging step. We suggest two different process orders, namely *Uniform Random Order (UR)* and *Low-Density-First Order (LDF)*, to optimize the number of merging operations and to alleviate a bottleneck in the merging, respectively. The proposed algorithm can produce the exact clustering results as the original DBSCAN with significant improvement on the efficiency.

The rest of this paper is organized as follows. We survey the related work in Section 2. We introduce preliminaries used in this paper in Section 3. Section 4 reveals the overall proposed framework. Sections 5 and 6 present our proposed HGB and GDCF, respectively. The experimental results and discussions are provided in Section 7. We conclude the paper in Section 8.

2. Related work

Due to the expensiveness in running time of the original DBSCAN, researchers have been developing many methods to improve the performance of DBSCAN for decades. Here we broadly categorize related papers in the literature as follows.

Sampling-based DBSCAN algorithms are designed to improve the time-consuming object neighbour query operation in DBSCAN. For example, Zhou et al. [39] proposed a sampling-based method which selects a few representatives, rather than all the neighbour objects, as seeds to expand the clusters. However, this method possibly misses some objects. Moreover, it still raises a representative selection problem which directly impacts on both performance and accuracy of DBSCAN. Another sampling-based method can be found in [40]. However, it cannot produce the exact results as well.

Grid-based DBSCAN algorithms. Zhao et al. [41] partition data layout in each dimension into intervals and the space is then partitioned into hyper-rectangular grids. The algorithm will only compute the density of a given object with the objects in its neighbouring grids. Thus, some objects can be omitted. Zhao and Song [42] attempted to improve the efficiency of [41]; however, they both still cannot produce the exact result. CIT [43] is the another one that develops grid-based algorithm for DBSCAN. The algorithm partitions the data layout into grids with width greater or equal to 2ε and sets a boundary of neighbour search to ε around the grids. Our work is related to [33] and [35]. Gunawan [33] proposed a 2D grid-based algorithm which can terminate in genuine $O(n\log n)$. Gan and Tao [35] extended the grid-based algorithm to solve DBSCAN in higher dimensions. Even though the observation of [35] makes a good contribution about the complexity of DBSCAN, recently, there is an interesting disputation [44] on some inaccurate statements of the paper [35]. In addition, the authors suggested when and why we should still use original DBSCAN. Sakai et al. [34] proposed an improved grid-based algorithm which refines the merging step by using minimum bounding rectangle criteria. While in this paper we devise a novel grid-based DBSCAN solution with an efficient and compact index as well as an im-

prove merging management strategy, which extends the algorithm to higher dimensional data.

Other efficient DBSCAN algorithms. There are other efforts for improving the efficiency of DBSCAN. Some of them focus on producing distributed or parallel version of DBSCAN [36,37,45–47]. Approximate algorithm is another kind of method to accelerate DBSCAN [35]. Mai et al. [38] proposed an anytime DBSCAN which employs active learning for learning the current cluster structure, it thus can select only necessary objects to expand clusters. Kumar and Reddy [31] proposed a novel graph-based indexing structure to alleviate the bottleneck of neighbour search operations in the traditional DBSCAN. Their method is divided into two phases. First, they scan the entire database to obtain a set of object groups. Then, the traditional DBSCAN is run based on the groups obtained from the first phase to speed up the neighbour search operations. This method can produce an exact result as the traditional DBSCAN. Recently, NQ-DBSCAN [30] was proposed. NQ-DBSCAN improves the traditional DBSCAN by proposing an efficient neighbour query to reduce the search space. It assumes that any nearby objects should have similar neighbouring objects, it uses the information of neighbouring object to speed up the algorithm by performing the distance calculation with only necessary objects to expand the cluster.

3. Preliminaries

In this section, we first review the original DBSCAN and then introduce the existing works that most relate to our method.

3.1. DBSCAN

DBSCAN groups objects of a dataset in d-dimensional space based on density with regard to two parameters: ε and *MinPTS*, where ε specifies the longest possible distance from an object to its neighbours, and *MinPTS* specifies the minimum size of subset to form a cluster. Formally, a dataset containing n objects is denoted by $\mathbb{D}$ and an object in a d-dimensional space is denoted by $p \in \mathbb{R}^d$.

Definition 1. Neighbourhood: An object q is a *neighbour* of object p if the distance between them is less or equal to ε, i.e., dist(p, q) $\leq \varepsilon$. The neighbour set of object p is denoted by N(p).

Definition 2. Core object: An object p is called a *core object* if $|\mathrm{N}(p)| \geq MinPTS$.

Definition 3. Direct density-reachability: an object p is *directly density-reachable* from an object q if q is a core object and $p \in \mathrm{N}(q)$.

Definition 4. Density-reachability: an object p is *density-reachable* from an object q if there exists an object sequence $p_1, p_2, \ldots, p_k$ where $p_1 = q$ and $p_k = p$ such that p_{i+1} is directly density-reachable from p_i, where $1 \leq i \leq k-1$.

Definition 5. Density-connectivity: an object p is *density-connected* to an object q if there exists an object o such that both p and q are density-reachable from o.

Definition 6. Cluster: a cluster C is a non-empty subset of $\mathbb{D}$ such that

- $\forall p, q$, if q is density-reachable from p w.r.t. ε and *MinPTS* and $p \in C$, then $q \in C$ *(Maximality)*,
- $\forall p, q \in C$, p and q are density-connected w.r.t. ε and *MinPTS (Connectivity)*.

Definition 7. Noise: an object p is a noise if p does not belong to any cluster in $\mathbb{D}$.

Basically, a primary cluster can be formed by a core object (refer to Definitions 1 and 2). Then, these primary clusters may be merged to the same cluster if they are density-connected (refer to Definitions 5 and 6) from each other. Finally, all non-core objects may be identified to be border objects of particular clusters or noises according to Definitions 6 and 7.

Example 1. Fig. 2 shows an example data layout. In such figure, all circle objects are core objects because the sizes of their neighbour sets are greater than or equal to *MinPTS* and thus they form primary clusters (Definitions 1 and 2). p_3 is density-reachable from p_1 because there exists a sequence p_1, p_2, p_3 (Definitions 3 and 4). And, p_1 and p_5 are density-connected since it has p_3 as a connector (Definition 5). Finally, this dataset contains 2 clusters: $C1$ and $C2$ (Definition 6) with tree triangle noise objects (Definition 7).

3.2. 2D grid-based DBSCAN

The grid-based algorithm in [33] implements the DBSCAN in real $O(n \log n)$ for 2D data. It consists of four steps, namely *partitioning step, labeling step, merging step* and *noise/border object identification step*. To the best of our knowledge, other existing grid-based DBSCAN algorithms also perform the same framework, and the only difference is the implementation details in some individual steps.

Partitioning step divides the whole dataset into equal-sized square-shaped grids, each grid has side width of $\frac{\varepsilon}{\sqrt{2}}$. This partitioning makes the possible longest distance within every grid be ε. It thus guarantees that the distance of every pair of objects resided in the same grid is at most ε. Then, the labeling step identifies whether each grid is a core grid by the following definition.

Definition 8. Core grid: A grid with at least one object is called a non-empty grid. A non-empty grid is called a *core grid* if and only if there are at least *MinPTS* objects inside or there exists at least one core object inside the grid.

The grids which are labeled as core grid can form their own individual clusters. Two or more core grids can possibly be merged to a same cluster. Hence, the merging step then needs to be carried out. The merging step will produce a graph in which vertices represent core grids, and there is possibly an edge added between the two vertices if they can be merged together. Finally, the final clusters will be obtained from its fully connected sub-graphs. In such step, the following *mergability* is utilized to determine whether the two grids can be merged or not.

Definition 9. Mergability: Given two core grids g_1 and g_2, they are considered to be in the same cluster if and only if there exists at least one core object pair p and q, where $p \in g_1$ and $q \in g_2$, such that dist(p, q) $\leq \varepsilon$.

After the merging step has been completed, all non-core objects need to be identified whether they are noises or the border objects of certain clusters. Such final step is called border/noise object identification step.

3.3. Grid-based DBSCAN when $\mathrm{d} \geq 3$

Gan and Tao [35] recently extended the above-mentioned 2D grid-based DBSCAN to higher dimensionality. In particular, for any dimension $d \geq 3$, they partitioned the dataset into grids with the side width of $\frac{\varepsilon}{\sqrt{d}}$ instead of $\frac{\varepsilon}{\sqrt{2}}$. In such case, the longest possible distance within a grid is still ε. Then an efficient nearest neighbour search method, namely Bichromatic Closest Pair (BCP) [48], is adopted to perform the *merging step* on the graph of grids in their algorithm. As a result, they claimed the algorithm has an average time complexity as $O(n^{2-\frac{2}{\lceil d/2 \rceil}+\delta})$ which is dominated by the BCP method in the *merging step*, where n denotes the number of

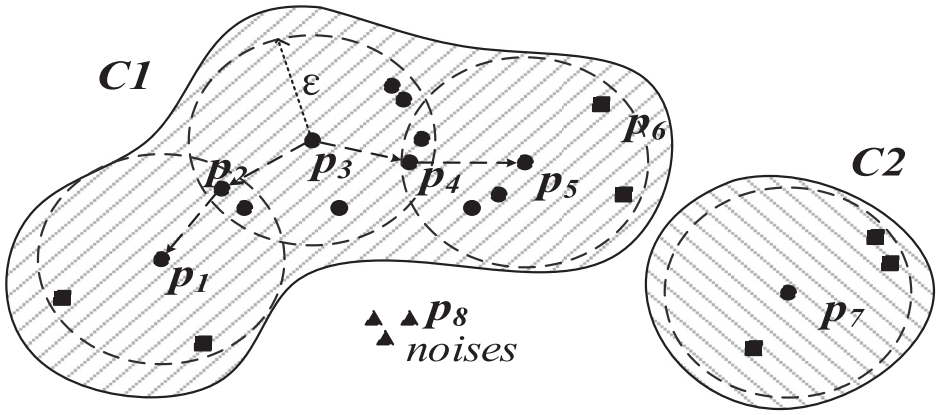


Fig. 2. An example of dataset layout. The dashed circles indicate the ε-radii of objects, and let $MinPTS = 4$.

objects, d denotes the data dimension, and δ can be an arbitrary small positive constant.

4. GDCF Framework

In this section, we detail the proposed GDCF framework. We first discuss the shortcomings of grid-based algorithms and then introduce our approach.

4.1. Motivation and framework of GDCF

In grid-based algorithms, we find that the number of neighbour grids of a given grid increases exponentially as the dimension increases, and we have the following lemma.

Lemma 1. *Considering a d-dimensional space, the number of the neighbour grids which needs to be checked for a given grid is* $(2\lceil\sqrt{d}\rceil + 1)^d$ *in the worst case.*

Proof. Without loss of generality, we denote a specific grid in the d-dimensional space as g. For each direction of individual dimension, the number of neighbour grids of g needs to be checked, denoted by $\mu \in \mathbb{N}$, is calculated as follows:

$$\mu \frac{\varepsilon}{\sqrt{d}} \leq \varepsilon, \mu \leq \sqrt{d} = \lceil\sqrt{d}\rceil.$$

Since there are two directions for each individual dimension, we need to consider $2\lceil\sqrt{d}\rceil + 1$ grids for each dimension. Here the number 1 denotes the grid g itself. Finally, combining all the dimensions together, the number of neighbour grids that the algorithm needs to consider is $(2\lceil\sqrt{d}\rceil + 1)^d - (\tau + 1)$ for every given grid g, where $\tau + 1$ is a constant indicating the number of corner grids and g itself which are not included in the set of neighbour grids. □

According to Lemma 1, given a specific grid, the running time of its neighbour grid query in the grid-based algorithms will be $O((2\lceil\sqrt{d}\rceil + 1)^d)$ in the worst case. For example, the neighbour grids of g_8 (the centered grid) in Fig. 3(a) are those 20 line-shaded grids.

We name the problem that the number of neighbour grids increases exponentially as *neighbour explosion*. Thus, an efficient indexing technique is demanded to support neighbour grid queries, especially when d increases. As a consequence, we adopt a bitmap-like structure HGB in GDCF to index non-empty grids such that it supports fast range queries.

Moreover, we find that the merging step has redundancies that are described as follows. Denoted by g_1, g_2 and g_3 as core grids, we may observe the following redundancies during the merging step.

- **Redundancy 1 (symmetry)**. g_1 needs to perform a merging operation with grid g_2 and vice versa. Both of them are equivalent, and either one can be skipped.
- **Redundancy 2 (transitivity)**. Assume the g_1 and g_2 are in the same cluster and so do g_2 and g_3. We should omit the merging operations between g_1 and g_3.

However, these redundancies are neglected by conventional grid-based algorithms in low data dimension. It is worth noting that these redundancies can become more severe due to the neighbour explosion problem in higher dimensional space. The reason is that the more neighbour grids, the more overlap neighbouring area becomes. Therefore, the grids having overlap neighbours can possibly encounter the above redundancies. Hence we devise a cluster forest to manage the merging of GDCF to alleviate the observed redundant computations.

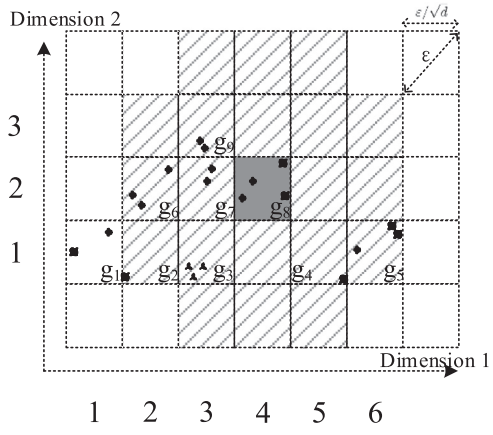
The framework of GDCF. The framework of grid-based algorithms carries over to GDCF almost verbatim. GDCF also consists of the four steps, and the only differences are the way that we index the non-empty grids (detailed in Section 5) and the way that we perform the merging step (detailed in Section 6).

5. Indexing with HGB

HGB (HyperGrid Bitmap) is a bitmap-like structure used to index non-empty grids in the GDCF framework. It is composed by d two-dimensional bit arrays, where d denotes the data dimension. We denote each of the bit array as $\mathcal{B}_i$ where $1 \leq i \leq d$. Each $\mathcal{B}_i$ can be regarded as a table with k_i rows and N_g columns, where k_i is the number of distinct positions of grids that contain objects in ith dimension, and N_g denotes the number of non-empty grids. In other words, each $\mathcal{B}_i$ is used to record the position of every non-empty grid in ith dimension. To set up each $\mathcal{B}_i$, we initially set all bits of it to 0, and encode each grid from id 1 to N_g. Considering a non-empty grid whose id $= x$, where $1 \leq x \leq N_g$, we first find the position of that grid in ith dimension, denoted as j for example, and set the corresponding bit of $\mathcal{B}_i$, i.e., $\mathcal{B}_i[j, x]$ to 1. $\mathcal{B}_i[j, x] = 1$ indicates the position of the grid x in ith dimension is j. The procedure of creating indices for non-empty grids is shown in Algorithm 1.

First, we initially set all the bits in HGB to 0 (Line 2). For each non-empty grid g, we set the bit array of dimension i and of value pos at array index k to 1 (Lines 3–7), where i is data dimension, pos is the position of grid in i-dimension and k is the grid index.

Example 2. Fig. 3(b) shows the HGB for the toy example in Fig. 3 (a). In the example, we have 9 non-empty grids in 2D space, thus $N_g = 9$ and $d = 2$. As a result, we need 2 bit arrays, namely $\mathcal{B}_1$ and $\mathcal{B}_2$, such that each of which contains 9 columns. Meanwhile,



(a) Partitioned data layout

Position	Dimension 1 g1	g2	g3	g4	g5	g6	g7	g8	g9
1	**1**	0	0	0	0	0	0	0	0
2	0	**1**	0	0	0	**1**	0	0	0
3	0	0	**1**	0	0	0	**1**	0	**1**
4	0	0	0	0	0	0	0	**1**	0
5	0	0	0	**1**	0	0	0	0	0
6	0	0	0	0	**1**	0	0	0	0

Position	Dimension 2 g1	g2	g3	g4	g5	g6	g7	g8	g9
1	**1**	**1**	**1**	**1**	**1**	0	0	0	0
2	0	0	0	0	0	**1**	**1**	**1**	0
3	0	0	0	0	0	0	0	0	**1**

(b) HGB for the non-empty grids in (a)

Fig. 3. Example of 2D data layout and its corresponding HGB.

Algorithm 1: Building HGB.

Input: $\mathcal{S}$: set of non-empty grids, d: data dimension
Output: $\mathcal{B}$: HGB on $\mathbb{D}$

```
1 k ← 0
2 Set all the bits in B to 0
3 foreach non-empty grid g ∈ S do
4     for i ← 1 to d do
5         pos ← g.pos[i]
6         B_i[pos, k] = 1
7         increase k by 1
```

$\mathcal{B}_1$ and $\mathcal{B}_2$ contain 6 and 3 rows since they have 6 and 3 distinct valid positions, respectively. For a specific grid, e.g. g_8 (id=8), we have $\mathcal{B}_1[4, 8] = 1$ and $\mathcal{B}_2[2, 8] = 1$ as shown in Fig. 3(b).

Once HGB is completely constructed, we can simply perform a conventional range query on HGB of a given grid g by setting the range for each dimension as follows:

$$\text{Range}(g, i) = [g.pos[i] - \lceil\sqrt{d}\rceil, g.pos[i] + \lceil\sqrt{d}\rceil],$$

where i denotes ith dimension. The procedure of neighbour grid query on HGB is shown in Algorithm 2.

Algorithm 2: NeighbourGridQuery(g, d, $\mathcal{B}$).

Input: g: the current considered grid
d: data dimension
$\mathcal{B}$: the HGB of dataset $\mathbb{D}$
Output: $\mathcal{G}$: the neighbour grid of the considered grid g

```
1  G ← initialize the set of neighbour grid of g as ∅
2  tmp1 ← initialize bit array of size 1 × n by 1
3  for i ← 1 to d do
4      tmp2 ← initialize bit array of size 1 × n by 0
5      for j ← g.pos[i] − ⌈√d⌉ to g.pos[i] + ⌈√d⌉ do
6          tmp2 = tmp2 OR B_i[j, :]
7      tmp1 = tmp1 AND tmp2
8  for i ← 1 to n do
9      if IsSet(tmp1[i]) then
10         G ← G ∪ GetGridByID(i)
11 return G
```

The NeighbourGridQuery performs a range query search with the support of HGB and returns all the non-empty neighbour grids of g which contain possible neighbour objects for the grid g. The maximum number of the returned grids is $(2\lceil\sqrt{d}\rceil + 1)^d$ according to Lemma 1. In particular, we first look up every table $\mathcal{B}_i$, to get rows with regard to specific ranges in each corresponding ith dimension (Lines 3–5 of the function). Then we perform bit-wise operations OR (Line 6 of the function) on among rows in such ranges which are in the same tables. The outputs of the bit-wise operations OR are d bit arrays representing the appearances of grids' intervals in each dimension. Finally, we will perform bit-wise operations AND (Line 7 of the function) on among those d bit arrays to get the final encoded bit array. We can decode this final bit array by looking up only the bit positions that are set to 1, and return the set of neighbour grids (Lines 8–10 of the function).

Example 3. In Fig. 3, considering a neighbour grid query of g_8, the ranges of dimension 1 and dimension 2 are [2, 6] and [1, 3], respectively. Then slices collected from tables corresponding to dimension 1 and dimension 2 are $\mathcal{B}_1$[2:6,:] and $\mathcal{B}_2$[1:3,:], respectively. First we perform bitwise operations OR on $\mathcal{B}_1$[2:6,:] and $\mathcal{B}_2$[1:3,:], the outputs are [0, 1, 1, 1, 1, 1, 1, 1, 1] and [1, 1, 1, 1, 1, 1, 1, 1, 1], respectively. Then the final bitwise operation AND is performed on the two bit arrays, outputting [0, 1, 1, 1, 1, 1, 1, 1, 1]. Thus, the set of neighbour grids of g_8 is all the grids except g_1.

5.1. Complexity analysis

First, we analyze the space complexity of HGB. Recall that we have d bit arrays in HGB, where d is the data dimension. For simplicity, we assume every bit array has κ distinct positions. Since each bit array has κ rows and N_g columns (the number of non-empty grids), the space complexity is thus $O(d \cdot \kappa \cdot N_g)$. Second, the time complexity of constructing HGB is $O(d \cdot N_g)$, since we need to scan all the non-empty grids and set the corresponding bits of bit arrays for every dimension. For neighbour query operation, the time complexity is $O(d \cdot (2\lceil\sqrt{d}\rceil + 1)) = O(d^{3/2})$, because it needs to collect slices (of the range size $(2\lceil\sqrt{d}\rceil + 1)$) of the d bit arrays.

6. The GDCF Algorithm

6.1. Concepts

The idea of GDCF is to reduce the redundancies of the merging step. We develop a tree-like data structure to maintain the

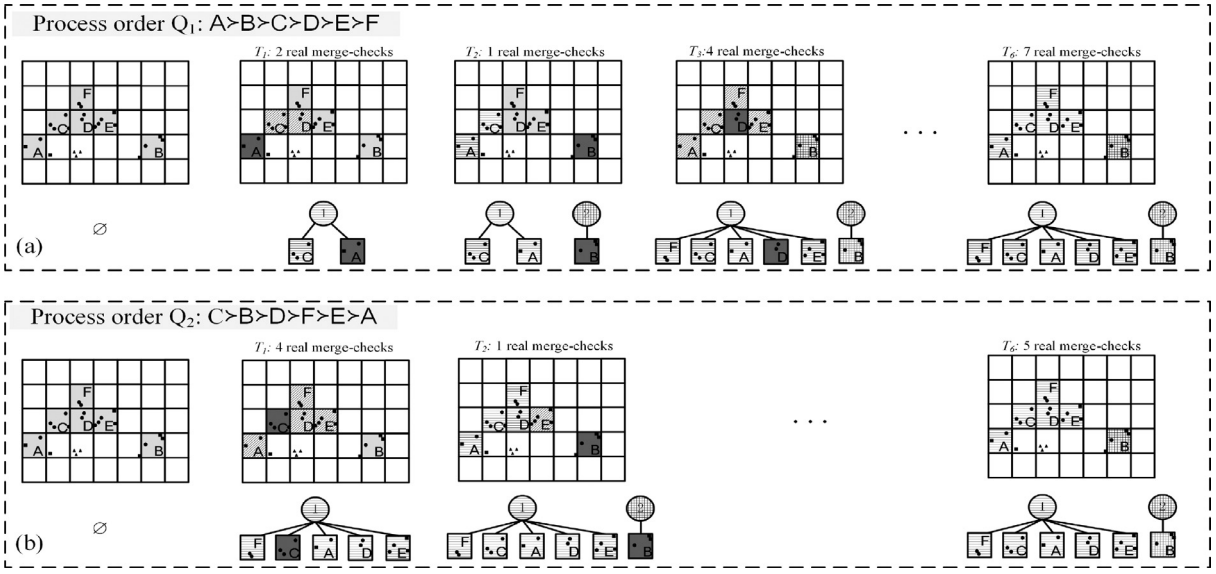


Fig. 4. shows a running example of the merging step for core grids A, B, . . . , E, F. (a) and (b) illustrate the merging step with its corresponding cluster forests at each time T_i in process orders Q_1 and Q_2, respectively. Here, we only show the case of time T_i in which there are changes on the cluster forests. Note that any pairs of core grids here can be merged with their neighbouring grids except that B cannot be merged with other grids.

clustering information by following the idea of union-find algorithm to mange the merging. We named this tree-like data structure as cluster forest. A cluster forest consists of a number of cluster trees which are used to maintain the clustering information of any states in the merging step. We give the definition, maintenance, and application of the cluster forest and discuss how the cluster forest alleviates the redundancy problem in Section 6.1.1. We observe that when the cluster forest is adopted, the order of processing has an influence on the total number of merge-checkings needed to perform. We find that performing the merging in uniform random order can optimize the number of merge-checkings in most cases. However, for high-density database, if we pick high-density grids to process first, a bottleneck could be occurred because we need to perform merge-checkings of that high-density grids and its neighbours. As a result, we propose to perform the merging in low-density-first order to alleviate the bottleneck. We discuss about orders of the merging step in Section 6.1.2.

6.1.1. Cluster forest

We implement the cluster forest such that the algorithm can make an inference whether two grids are in the same cluster or not.

Definition. A cluster forest, denoted by Π, consists of trees, and each tree is denoted by π. Each tree consists of *cluster nodes* (as intermediate nodes) and *core grid nodes* (as leaf nodes). Both cluster nodes and core grid nodes contain a *parent* field, which indicates the parent of the node. Additionally, the cluster node has another field *cluster* that indicates the cluster id that such node identifies. And, we denote the root of node p by rc(p). Note that for a core grid node g, $g.parent$ returns the cluster node that grid g belongs to. For example, Fig. 4(a) at time T_1 illustrates a cluster forest (bottom) for its corresponding data layout (top). In the cluster forest shown in Fig. 4(a), the circle and squares represent the cluster node and core grid nodes, respectively. The grids residing in the same tree are inferred that they are in the same cluster, e.g., grids A and C, in Fig. 4(a) at T_1, belong to the same cluster which is cluster 1. In the merging step, we will maintain such forest by creating trees and merging those trees if needed. The following subsections provide the details of maintenance and application of such cluster forest.

Maintenance. At the moment of processing a grid g, a new cluster node will be created and assigned to the grid g if it is a no-cluster grid. For example, in Fig. 4(b) at T_2, B is being processed, and a new cluster node ② is created and assigned as the parent of B. Then every neighbour grid g' of g, i.e., $g' \in \mathcal{G}(g)$, that can be merged with g and already belongs to some cluster, will be inserted into a set, denoted by $\mathcal{A}$. Finally, the algorithm will search the root node from among grids in the set $\mathcal{A}$ which has the smallest cluster number, denoted by $\text{lrc}(\mathcal{A})$, and assigns $\text{lrc}(\mathcal{A})$ as the parent to all the roots of every grid in the set $\mathcal{A}$. That is, in Fig. 4(a) at T_3, grids A, C, D, E, F can be merged with D. Here the set $\mathcal{A}$ will contain those grids including D. The algorithm assigns $\text{lrc}(\mathcal{A}) = ①$ as the parent of all the roots of every grid in $\mathcal{A}$ as shown in Fig. 4(a) at T_3 (the bottom forest). In the case that a grid g' in $\mathcal{G}(g)$ can be merged with g, and g' is a no-cluster grid, the algorithm will directly add g' on the cluster forest and assign the parent of grid g as the parent of g'. According to Fig. 4(a) at T_1, for example, grid C is directly added to the same tree as grid A. Notice that grids A and C share the same parent, namely ①. The forest after time T_1 can be shown in the bottom forest in Fig. 4(a).

Application. Once a cluster forest is constructed at time T, we can make an inference, by the best information at the time T, whether two grids, in the forest, p and q are already in the same cluster or not. The inference between grids p and q by a cluster forest Π is denoted by Π.Infer(p, q). Π.Infer(p, q) returns `true` if $\text{rc}(p) = \text{rc}(q)$ which means p and q are already known they are in the same cluster. Otherwise, it will return `false`. This inference turns out to be useful since the algorithm can skip real merging operation between the grids p and q if they can be inferred that they are already in the same cluster. This can lead to a tremendous reduction in the running time of algorithm.

6.1.2. Merging orders

From Section 6.1.1, we know that if we have a set of N core grids, for any time $\{T_1, T_2, \ldots, T_{N-1}\}$, if the process order is different, the *cluster forest* might be different as shown in the following example.

Example 4. Let process order $Q = \{g_1 \succ \ldots \succ g_l\}$ produce a cluster forest $\Pi_{T_l}^{Q}$ and let process order $Q' = \{g'_1 \succ \ldots \succ g'_l\}$ produce a cluster forest $\Pi_{T_l}^{Q'}$, where $l \in [1, N-1]$. g_i and g'_i are the core grids which are processed at time T_i. If there exists $g_i \neq g'_i$, then $\Pi_{T_l}^{Q}$ might not be equivalent to $\Pi_{T_l}^{Q'}$. Fig. 4 shows the merging step of two different process orders and their corresponding cluster forests at time T_i. As we can see, these two cluster forests are different at time T_2, T_3, T_4. Thus, the inference made by the two cluster forests might be different. For example, at time T_2, $\Pi_{T_2}^{Q_1}$. Infer(A, D) = `false`, while $\Pi_{T_2}^{Q_2}$. Infer(A, D) = `true`.

As a result, making inferences by different cluster forests can result in different inference results. This leads to a different number of merging operations needed to perform in the merging step. Therefore, the process order in the merging step has an influence on the overall performance of algorithm. Fig. 4 shows that the numbers of merging operations needed to be performed are different if the process orders are different, i.e., 7 merging operations (Fig. 4 (a)) versus 5 merging operations (Fig. 4(b)). In this paper, we propose two different process orders in the merging step. We argue that GDCF still performs correctly in any other orders, this thus makes room for future work.

Uniform Random order (UR). This order first shuffles the core grids from the labeling step in a uniform random manner and puts them into a queue. Then we will pick those core grids from the queue one by one and stops when the queue is empty. Since any core grids are uniformly picked to be processed, the cluster forest are likely to almost cover the core grids at the beginning. Consequently, unprocessed core grids will have much possibility to make inferences such that they can skip those costly merge-checkings with their neighbours.

Low-Density-First order (LDF). However, performing the merging in orders that the densities of grids are not taken into account, such as UR, may lead to a bottleneck that the algorithm performs merge-checkings of high-density grids. To alleviate the bottleneck, we propose to perform the merging in low-density-first order. Specifically, we pick low-density grids to perform merge-checkings with their neighbour first such that the cluster forest can be soon established. As a result, high-density grids can take the advantages of the cluster forest by making inferences to skip the actual merge-checkings. Therefore, the bottleneck can be mitigated. Comparisons and discussions between these two orders are provided in the experiment section.

6.2. The overall algorithm

In this section, we describe the overall algorithm under the proposed GDCF based on the two process orders. The pseudo code of the overall algorithm is given in Algorithm 3. First, we collect all the core grids and put them into a queue such that the order is in regard to the process order mode (Lines 1–4). Next, for every core grid $g \in \mathcal{Q}$, we invoke the NeighbourGridQuery function to query the set of neighbour grids and include g into a set, denoted by $\mathcal{A}$. After that we begin to maintain the grid g on the cluster forest (Lines 8–22, refer to Section 6.1.1).

However, we can skip such costly checking by using the cluster forest to make inferences. That is, if grid g and g' can be inferred that they are in the same cluster, where $g' \in \mathcal{G}$, we simply skip the merging operation between these two grids (Lines 14 and 15). Otherwise, we merge these two grids according to Definition 9. If they can be merged and g' does not appear in the cluster forest, we create a corresponding leaf node and connect it with

Algorithm 3: GDCF Algorithm.

```
   Input: ℂ : set of core grids
          d: the data dimension
          ℬ: the HGB
   Output: Π: the finalized cluster forest
1  if mode = LDF then
2      𝒬 ← sort ℂ by #objects in each grid in ascending order
3  else
4      𝒬 ← randomly shuffle ℂ
5  𝒳 ← 0, Π ← ∅
6  foreach g ∈ 𝒬 do
7      𝒢 ← NeighbourGridQuery(g, d, ℬ)
8      𝒜 ← {g}
9      if g ∉ Π then
10         create a new tree π with a leaf node of g and its
           cluster node g.parent, g.parent.cluster ← 𝒳
11         increase 𝒳 by 1
12         add π to Π
13     for each grid g' ∈ 𝒢 do
14         if Π.Infer(g, g') = true then
15             g and g' are in the same cluster and skip
16         else if g and g' can be merged then
17             if g' ∉ Π then
18                 create a leaf node of g'
19                 g'.parent ← g.parent
20             else
21                 𝒜 ← 𝒜 ∪ {g'}
22     set parents of all roots of cluster numbers in set 𝒜 to
       lrc(𝒜)
```

g.parent (Lines 17–19). Once all the core grids are processed, the algorithm then finishes. Although the worst-case complexities of GDCF in any orders are identical to that of the grid-based algorithm analyzed in [33], GDCF can prune the redundant operations in the merging step. Therefore, the performance can be significantly improved, this will be shown in the experiment section.

6.3. Correctness of GDCF

We prove the correctness of GDCF by the following theorems in this subsection.

Theorem 1. *(Correctness). Every tree in the final cluster forest* Π *returned by GDCF algorithm correctly denotes an individual cluster of core grids.*

To prove such theorem, we need to prove the following two propositions. Without loss of generality, we assume there are two grids g and g' and their corresponding nodes are contained by the cluster forest Π.

Proposition 1. *Given two grids* g *and* g' *which can be merged on* Π, *if* $g \in \pi \subseteq \Pi$, *then* $g' \in \pi$.

Proof. We prove it by deriving to a contradiction. Assume $g' \in \pi'$ where $\pi' \neq \pi$. Denoted by r and r' as the root nodes of the trees π and π', respectively, we have $r \neq r'$ since $\pi \neq \pi'$. Since g and g' can be merged, they will belong to the same set, namely $\mathcal{A}$ (cf. Lines 16–21). Then r and r' will be the same as they are equal to $\text{lrc}(\mathcal{A})$, where $\text{lrc}(\mathcal{A})$ is the lowest root cluster node in the set $\mathcal{A}$. We have derived a contradiction. Hence, g and g' are in the same tree. □

Proposition 2. *Given two grids g and g' which cannot be merged on Π, if $g \in \pi \subseteq \Pi$, then $g' \notin \pi$.*

The proof of such proposition is similar to that of Proposition 1, and we thus omit it due to the space limitation. With Propositions 1 and 2, we can derive the correctness of GDCF. Next, we prove the completeness of the algorithm.

Prior to the next proof, we give the following lemma.

Lemma 2. *Given a cluster forest Π at any time T, denoted by Π_T, if there exists a core grid sequence $p_1, p_2, \ldots, p_j \in \mathcal{Q}$, where $p_1 = p$ and $p_j = q$ such that p_{i+1} can be merged with p_i (according to Definition 9), where $1 \le i \le j-1$, then all $p_1, p_2, \ldots, p_j \in \pi$, where $\pi \in \Pi_{T_i}$.*

Proof. This can be proved by deriving a contradiction. Assume all the core grids in $\mathcal{Q}$ except a grid p_l are in the same tree π. Suppose $p_l \in \pi'$, where $\pi' \neq \pi$ and $l \in [1, j]$. Since p_l can be merged with at least one grid $p_o \in \mathcal{Q}$, where $l \neq o$, we derive $\pi = \pi'$ (Proposition 1). We thus induce to a contradiction. □

Theorem 2. *(Completeness) For any process order $\mathcal{Q} = \{g_1 \succ g_2 \succ \cdots \succ g_m\}$, where m is the number of core grids, after time T_m, $\Pi_{\mathcal{Q}}$ will contain equivalent cluster trees in terms of reachability.*

Let $p, q \in \mathcal{Q}$, we prove the completeness by proving that Infer(p, q) always returns a correct answer.

Proof. We divide this proof into two cases as follows.

Case 1: Infer(p, q) returns `false`. It is trivial since the algorithm will perform an actual merging operation without any inference between p and q. In case p and q can be merged, they will be in a same tree (refer to Section 6.1.1). As a result, they will reach to each other by the root node.

Case 2: Infer(p, q) returns `true`. This case is proved by Lemma 2, where $p_1 = p$ and $p_j = q$. □

As the algorithm stops after time T_m, that is, all core grids have been processed either **Case 1** or **Case 2** with all their neighbour grids. Therefore, the completeness holds. According to the Theorems 1 and 2, GDCF in any merging orders are equivalent to the *grid-based DBSCAN* which can produce the exact results as the original DBSCAN.

7. Experiments

In this section, we present the results of our experimental studies on real-world and synthetic datasets.

7.1. Experimental settings

All the experiments were conducted in a workstation equipped with four Intel (R) CPU E5-2609 v3 processors and 128GB RAM running a Linux Cent OS 6.5. We implemented our proposed GDCF with C++.

7.1.1. Datasets

We evaluated our algorithm on four synthetic and two real-world datasets. Table 1 depicts their statistics.

Synthetic Datasets We generated the synthetic datasets by a generator URG in C++. The generator takes 4 parameters: the number of objects (n), the number of clusters (c), the number of dimensions (d), and the percent of noise ($pnoise$) [default=0.0005%]. We used URG to generate five different kinds of datasets, i.e., 3-, 10-, 15-, 20-, 30-, and 40-dimensional datasets in range 1000–10,000 in each dimension. To avoid too-dense cluster, when $0.00025n$ objects have been generated, the data will have possibility to move a bit (33% for -5, 33% for $+5$) in each dimension. For simplicity and convenience, we denote each of them by its dimensionality when discussing in the following parts. For example, if we set $n = 3, c = 10, d = 3$, the URG will generate a dataset with 3 million objects grouped into 10 clusters in 3-dimensional space, and we denote it as 3D.

Table 1
Dataset statistics.

Dataset	Dimension	Type	#Objects	#Clusters
3D	3	Synthetic	3,000,000	10
10D	10	Synthetic	3,000,000	10
30D	30	Synthetic	3,000,000	10
40D	40	Synthetic	3,000,000	10
Household	7	Real	2,075,259	N/A
PAMAP2	54	Real	3,850,505	N/A

Real-world Datasets All the real datasets are obtained from UCI Machine Learning Repository [49]. We evaluated on 7- and 54-dimensional datasets. For the 7- and 54-dimensional datasets, we follow [35] to use Individual household electric power consumption (Household) and PAMAP2 [50] as the 7-, 54-dimensional datasets, respectively.

7.1.2. Compared methods and parameter settings

The compared methods in the experiments include

1. DBSCAN: original DBSCAN [9] with r*tree,
2. GRID [35]: A state-of-the-art grid-based exact DBSCAN algorithm,
3. GRID-A [35]: A state-of-the-art grid-based approximate DBSCAN algorithm,
4. HGB: our proposed method with only HGB indexing,
5. GDCF-UR: Our full proposed method in UR order.
6. GDCF-LDF: Our full proposed method in LDF order.

For the implementation of DBSCAN, GRID, and GRID-A, we used the binary code which is implemented by C++ and publicly available.[1] We investigated the datasets and followed the suggestions produced by the parameter selection tool [51] for setting ε and *MinPTS* with regard to the range and dimension of datasets, as well as the number of grids.

7.2. Experimental results

In this subsection, we demonstrate the experimental results. All the reported running time of the compared methods is a 3-time-run average value, and we did not include some experimental results of DBSCAN because it failed to report the results within 15 h. Note that all the running time reported includes the running time of the four steps in the algorithm, i.e. the partitioning step (building HGB), the labelling step, the merging step (GDCF), and the noise/border object identification step.

7.2.1. Clustering result quality

First, we examine whether the clustering quality of our proposed methods GDCF in UR and LDF orders are equivalent to the original DBSCAN. For this purpose, we use the clustering results produced by the original DBSCAN as the ground truth. We employ four validity measures, i.e. *adjusted rand index (ARI), purity, precision*, and *F1-score*. The results are presented in Table 2. We can see the consensus that all the measures suggest that our method in both UR and LDF orders can produce exactly the same results as the original DBSCAN on all the measured datasets. We also provide the 2D visualization of the compared methods in Table 3. We can see that both GDCF-UR and GDCF-LDF can group all the objects into the clusters exactly the same as the original DBSCAN does.

[1] https://sites.google.com/site/junhogan/.

Table 2
Clustering quality measures over original DBSCAN.

Aggregation [52]				
	ARI	Purity	Precision	F1-Score
UR	1.0	1.0	1.0	1.0
LDF	1.0	1.0	1.0	1.0
D31 [52]				
	ARI	Purity	Precision	F1-Score
UR	1.0	1.0	1.0	1.0
LDF	1.0	1.0	1.0	1.0
t4.8k [52]				
	ARI	Purity	Precision	F1-Score
UR	1.0	1.0	1.0	1.0
LDF	1.0	1.0	1.0	1.0

Moreover, Table 4 also shows that the numbers of clusters produced by GRID, which is an exact grid-based DBSCAN, and our proposed GDCF in UR and LDF orders are the same. In addition to the proof in Section 6.3, the experiments in this section also show that GDCF in UR and LDF orders is equivalent to the original DBSCAN.

7.2.2. Overall performance

Fig. 5 illustrates the execution time of each compared method on both synthetic and real-world datasets in **log scale**. From the results on the synthetic data shown in Fig. 5, first we observed that HGB always runs faster than DBSCAN and GRID. For example, HGB runs 197.59× faster than DBSCAN (3D dataset, $\varepsilon = 60$, $MinPTS = 20$), and almost 6× faster than GRID (40D dataset, $\varepsilon = 800$, $MinPTS = 80$). Since HGB produces efficient neighbour grid query, the proposed method obtains a clear time-saving.

Second, GDCF-UR, with the merging management strategy for reducing redundant computations, significantly outperforms the other compared methods. It achieves approximately 3000× speedup compared with DBSCAN. Moreover, it yet has almost up to three orders of magnitude of speedup compared with GRID and our HGB. In addition, we surprisingly observed that the GDCF-UR method is even faster than the recent GRID-A, which is claimed as a $O(n)$ approximate DBSCAN algorithm, when the data dimension is larger than 3. For example, GDCF-UR achieves 736× speedup compared with GRID-A (30D dataset, $\varepsilon = 600$, $MinPTS = 70$). We argue that the reason for such observation is that the number of the neighbour grids to be checked grows exponentially with the data dimension. As a consequence, we cannot neglect it as the data dimension increases, and our proposed merging management strategy suggests its effectiveness. Furthermore, Fig. 5(b), (c), and (d), respectively, also show that when the data dimension increases, GDCF-UR still achieves advantages compared with GRID-A.

Similar observations can also be found from the results of real-world data shown in Fig. 5(d) and (e). That is, HGB is faster than GRID, and GDCF-UR, moreover, clearly leads the compared two exact algorithms. For instance, HGB runs 15.38× faster than GRID, on the other hand, GDCF-UR runs 128.52× and 1191.97× faster than HGB and GRID (PAMAP2 dataset (54D), $\varepsilon = 400$, $MinPTS = 150$), respectively.

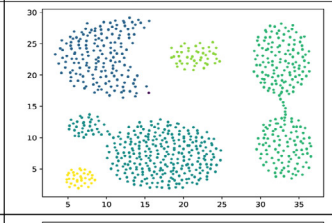
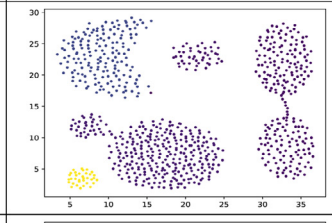
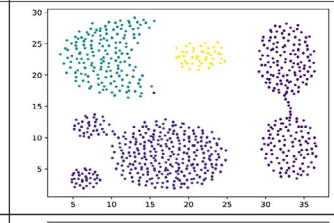
Additionally, considering the running times of 3D and 10D in Fig. 5(a) and (b), GDCF-LDF significantly runs faster than GDCF-UR. For example, GDCF-LDF runs 30.11× faster than GDCF-UR (10D dataset, $\varepsilon = 320$, $MinPTS = 50$). While, there are no significant differences of their running times on the other datasets (30D, 40D, Household, PAMAP2). The reason is that we fix the ranges

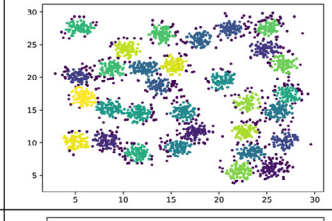
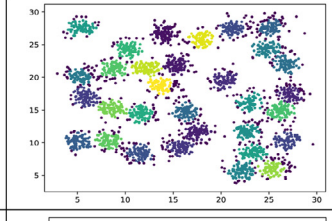
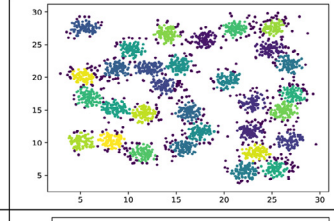
Table 4
#Clusters produced by the compared methods.

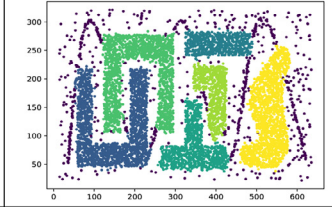
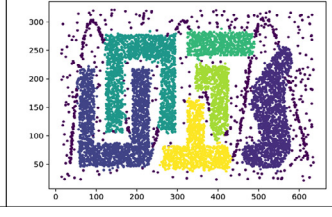
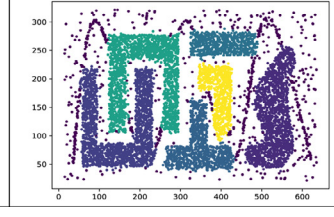
Dataset	Parameters		#Clusters			
	ε	*MinPTS*	DBSCAN	GRID	HGB	GDCF-UR
3D	60	20	10	10	10	10
10D	200	50	N/A	10	10	10
30D	600	70	N/A	10	10	10
40D	800	80	N/A	10	10	10
Household (7D)	10	20	N/A	3	3	3
PAMAP2 (54D)	350	150	N/A	1	1	1

Table 3
Clustering results of original DBSCAN and the proposed methods on 2D datasets.

Dataset Detail	Original DBSCAN	GDCF-UR	GDCF-LDF

Dataset: Aggregation
#objects = 788
Dimension = 2
$\varepsilon = 1.5$
$MinPTS = 5$
			

Dataset: D31
#objects = 3100
Dimension = 2
$\varepsilon = 0.8$
$MinPTS = 30$
			

Dataset: t4.8k
#objects = 8000
Dimension = 2
$\varepsilon = 8.5$
$MinPTS = 15$
			

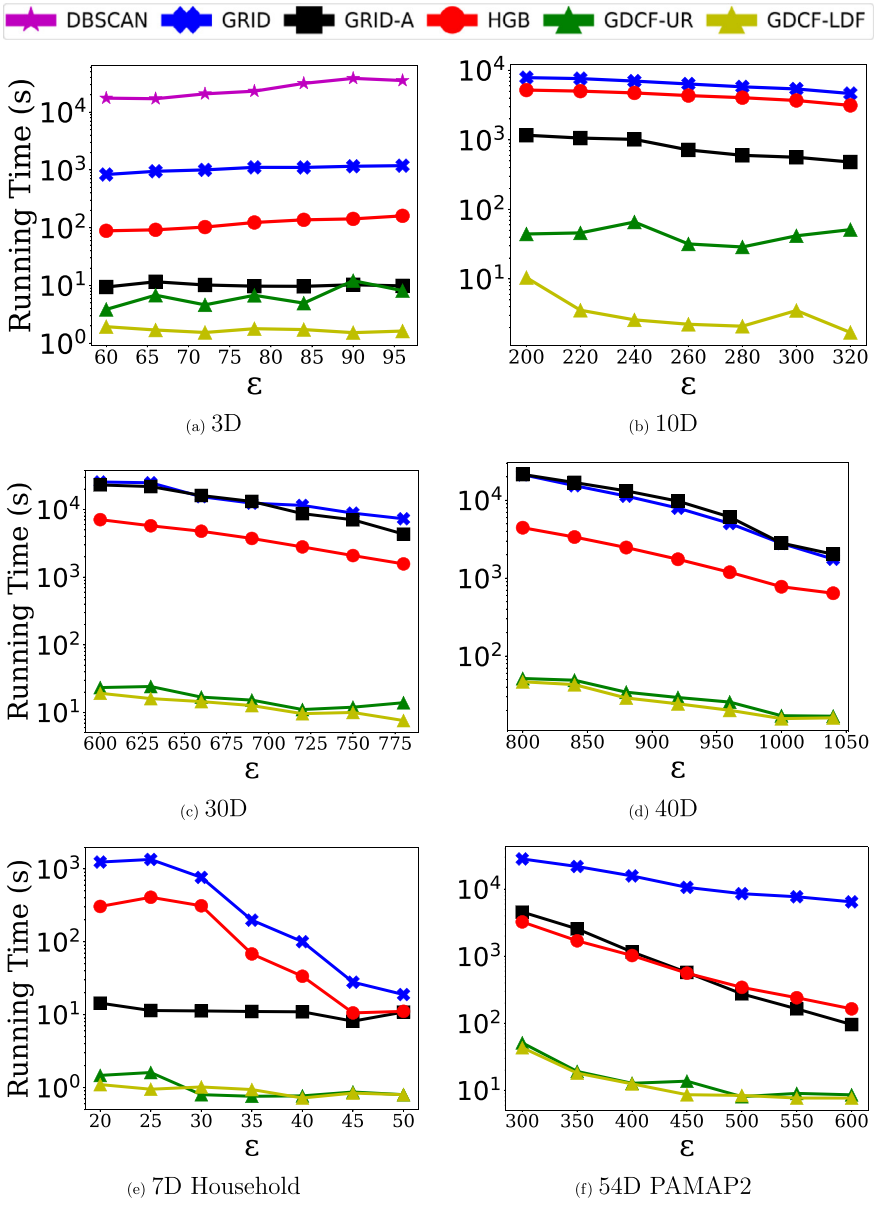


Fig. 5. Running time on synthetic and real-world datasets.

(1–10000) of synthetic datasets while varying the dimension. Consequently, lower-dimensional datasets (3D, 10D) can be denser than higher-dimensional datasets (30D, 40D). Since the distributions of real-world datasets are unknown, we then conjecture the similar reason for the real-world datasets that the density distribution in each grid of real-world datasets may be sparse. Therefore, GDCF-LDF cannot have more benefits on the cluster forest compared with GDCF-UR (refer to Section 6.1.2).

7.2.3. Effectiveness of HGB

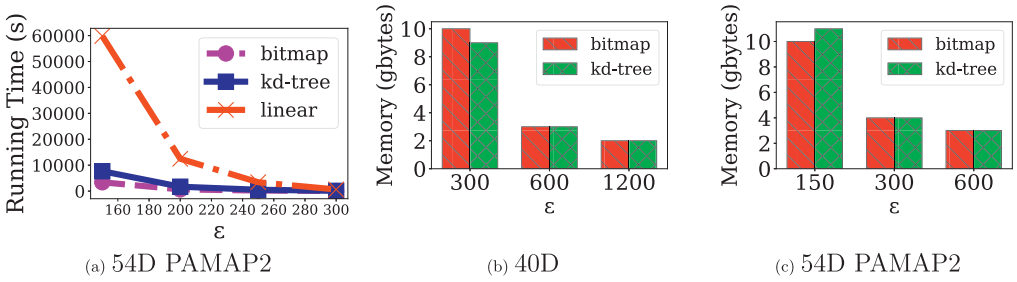
To demonstrate the effectiveness of HGB, we show the running times of our framework which employs linear-search, kd-tree, and HGB as neighbour search techniques by fixing *MinPTS* and varying ε in Fig. 6(a). First, we observe that the linear search as neighbour search performs the worst in most cases of ε since it needs to scan every single grid to check whether they are the neighbour grids of the processing grid or not. On the other hand, HGB runs 2 $\times$ faster than kd-tree and approximately 17 $\times$ faster than the linear search when ε is small ($\varepsilon = 150$, the number of the neighbour grids increases). Note that small ε challenges the clustering process as the number of partitioned grids will significantly increase. According to this experiment, indexing techniques clearly facilitate the neighbour search, and HGB can achieve the fastest running time on 54-dimensional data. In addition, Fig. 6 (b) and (c) show the memory consumptions of kd-tree and HGB indexing on 40D synthetic and 54D real-world datasets, respectively. We can observe that kd-tree and bitmap have almost a similar amount of consumed memory in most cases. We conclude that the bitmap indexing is more efficient than kd-tree for indexing non-empty grids in terms of execution time, while their memory consumptions are not significantly different.

7.2.4. Effectiveness of cluster forest

Next we visualized the number of merging operations as shown in Fig. 7 to exhibit the effectiveness of the cluster forest. It is clear that the numbers of operations used by GRID and HGB are almost the same. The reason is that we only use HGB to index non-empty grids without any specialized techniques to avoid merging redundancy. In addition, both GDCF-LDF and GDCF-UR achieve a significant operation-saving compared with HGB and GRID. For example,

Table 5
Running time in details of GRID, GRID-A and GDCF-UR on low-dimensional data (3D) and high-dimensional data (54D)

Algorithm	Dataset	Parameters			Running time (s)				
		d	ε	*MinPTS*	Partitioning	Labeling	Merging	Border/Noise	Total
GRID	Synthetic	3	60	20	0.97	0.28	833.43	**0.00**	834.70
GRID-A					0.73	0.16	8.52	0.01	9.42
GDCF-UR					**0.33**	**0.02**	**5.65**	0.01	**6.01**
GRID	PAMAP2	54	200	100	8.18	7896.77	83019.06	0.46	90924.49
GRID-A					**5.88**	6372.39	37664.18	0.51	44042.97
GDCF-UR					6.19	**183.78**	**784.19**	**0.31**	**974.70**



(a) 54D PAMAP2 (b) 40D (c) 54D PAMAP2

Fig. 6. Effectiveness of HGB.

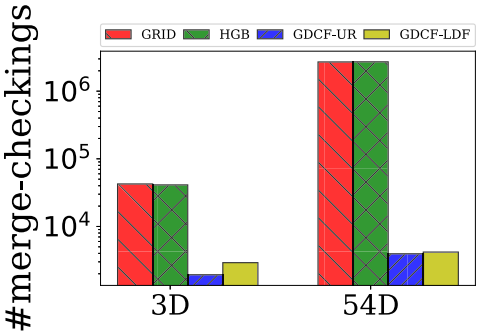


Fig. 7. Effectiveness of GDCF.

GDCF-UR performs only 0.15%, 4.62% of merging operations compared with GRID on 54D real-world and 3D synthetic datasets, respectively. As expected, we can see the saving ratio on 54D dataset is much greater than that of 3D. The reason is that we encounter more redundancies in higher dimension. The results demonstrate the effectiveness of cluster forest in redundancy reduction in the merging step.

7.2.5. A closer look

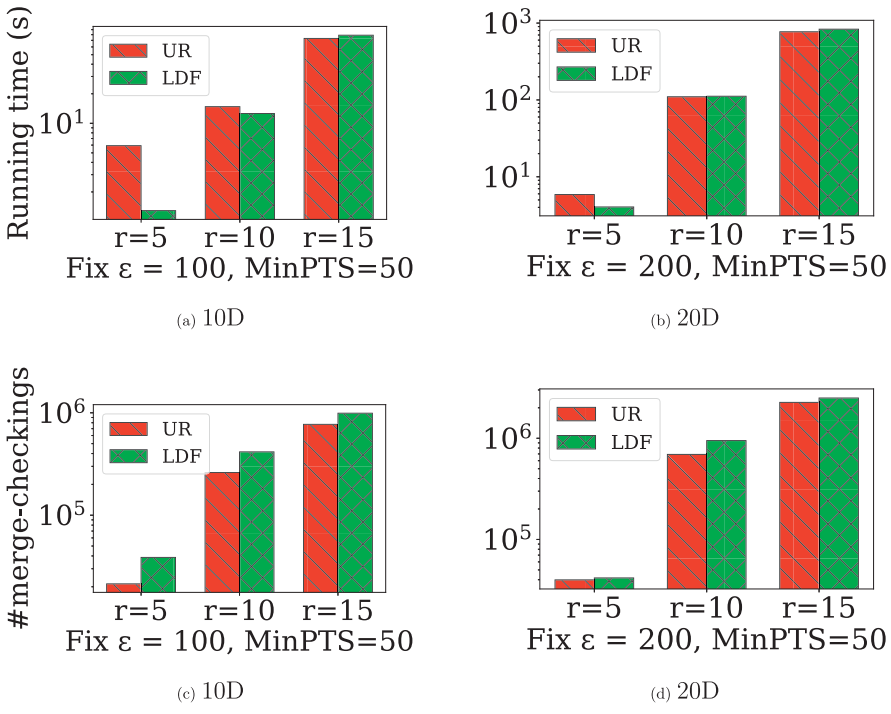
Next, we take a closer look on each step. Table 5 presents the running time of the four steps of some compared algorithms under the grid-based framework, namely GRID, GRID-A and GDCF-UR. The shortest running time on each step is marked in the boldface. First, we observe that the running time of the partitioning step of all compared methods is not significantly different, and the time used to build HGB is not excessive when considering overall running time of GDCF-UR. Next, the running time of the labeling step becomes nonnegligible on high-dimensional dataset. For example, the percentage of the running time of the labeling step of GRID is 0.03% on the 3D dataset, while such percentage becomes 8.68% on the 54D dataset. For GRID-A, the percentage is 1.7% and 14.47%, respectively. Such observation indicates that the neighbour grid query has a great impact on the overall running time on high-dimensional data. For our proposed approach, though the running time of the labeling step increases from 0.02 to 183.78, it still outperforms the baselines significantly, which derives a clear facilitation. In addition, we observed that the labeling step under the proposed HGB structure runs almost 43×, 35× faster than of GRID and GRID-A on PAMAP2 dataset, respectively, as shown in Table 5. It proves that querying neighbour grids under our proposed HGB structure is effective on high-dimensional data.

We further observed the running time of merging step dominates the overall performance on both low and high-dimensional datasets. As a result, techniques focusing on improving such step may guarantee performance gain. Our GDCF-UR, with the help of cluster forest, cuts down redundant merging operations and thus acquires the clear time-saving. For example, GDCF-UR performs 147× faster than GRID in merging step when $d = 3$. This efficiency also applies to high-dimensional data as shown in the table that GDCF-UR runs 105× faster than GRID, and even 48× faster than GRID-A in the merging step on the 54D dataset.

7.2.6. UR versus LDF

In this section, we examine the GDCF framework running in the UR and LDF orders as we know that running GDCF in different orders may lead to the different performance (refer to Section 6.1.2). For this purpose, we additionally generate six datasets using URG (see Section 7.1.1) in 10D and 20D with different spreading radii (r) that denote how far the objects are generated from the cluster origin. Note that the smaller r the denser datasets become, i.e. we can consider the dataset which is generated from $r = 5$ *density-higher* than the datasets which are generated from $r = 10$ and $r = 15$, respectively. Fig. 8(a) and (b) show the running time, and Fig. 8(c) and (d) show the number of merging operations of GDCF-UR and GDCF-LDF on the datasets. First, we can see that GDCF-LDF runs clearly faster than GDCF-UR when $r = 5$ (high-density dataset) on both datasets, as shown in Fig. 8(a) and (b). This empirically indicates that running GDCF in the LDF order can better utilize the cluster forest on high-density datasets. As we expect, when the dataset is sparser GDCF-UR can archive a better performance. We can see this phenomenon in the same figures on both datasets from $r = 5, 10, 15$, respectively. Considering 10D dataset (Fig. 8(a)), GDCF-UR is 4.5× slower than GDCF-LDF when r is set to 5; however, the gaps of their performances become closer as $r = 10$, and when $r = 15$ GDCF-UR eventually runs faster than GDCF-LDF. We can also find a similar observation on the 20D dataset. We conclude that GDCF in the LDF order can better handle the high-density databases.

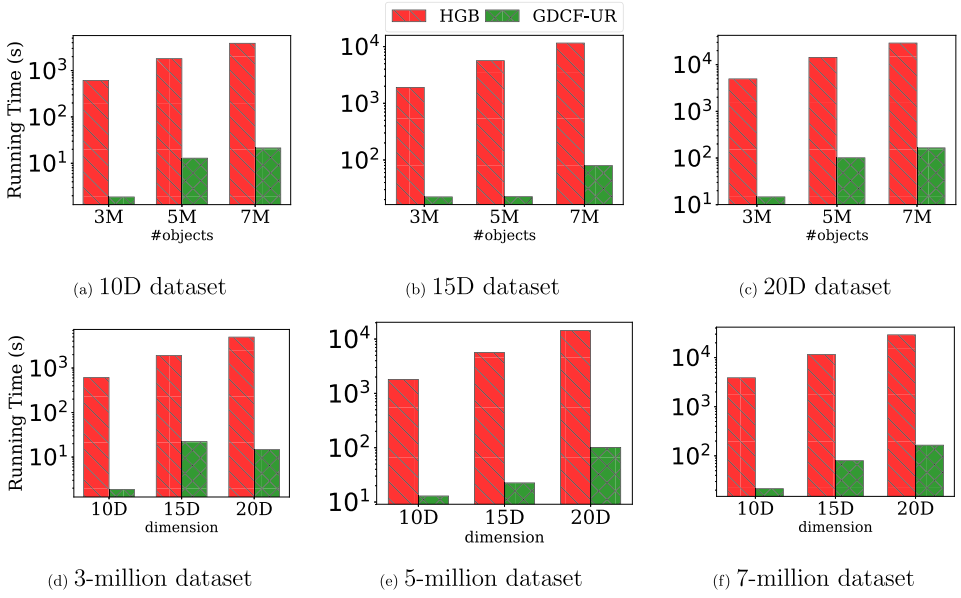
However, it is interesting to notice that GDCF-UR outperforms GDCF-LDF in most cases in terms of the number of merging operations as depicted in Fig. 8(c) and (d), even in case of the



(a) 10D (b) 20D

(c) 10D (d) 20D

Fig. 8. UR vs LDF.



(a) 10D dataset (b) 15D dataset (c) 20D dataset

(d) 3-million dataset (e) 5-million dataset (f) 7-million dataset

Fig. 9. Scalability.

high-density dataset that GDCF-LDF is found to be better in terms of running time. In general, we do not have such prior knowledge about the density of the database. Considering the real-world datasets in Fig. 5(e) and (f), the performances of GDCF-UR and GDCF-LDF are comparable; however, we know that GDCF-UR can optimize the merging operations better than GDCF-LDF does (Fig. 7, Fig. 8(c) and (d)). We then suggest running GDCF in the UR order in general cases that we do not know the distribution of the database, while GDCF-LDF is preferable if the database is known to be highly dense.

7.2.7. Scalability

Finally, we examined the scalability of the proposed algorithms as we increased the input size/data dimension. In particular, we generated datasets by using URG and set n to 3, 5, 7 on each d = 10, 15, 20. Then we obtained nine datasets, and we run both HGB and GDCF-UR on each of them. First, we visualized the execution time by fixing the data dimension and varying the data size as shown in Fig. 9(a)–(c). Interestingly, from the figure, we view the performance of both HGB and GDCF-UR that they rise lower than linear increase. Furthermore, GDCF-UR rises much slower than HGB. We analyze that though GDCF-UR theoretically has the same time complexity of merging step as HGB in the worst case, it can still dramatically reduce some of the symmetric and transitive redundant merging operations. Thus, it scales well to large datasets. We second examined the scalability to the data dimension and showed in Fig. 9(d)–(f). From the figure, we can find similar observations as the previous investigation that both HGB and

GDCF-UR scale well as the data dimension increases. Additionally, GDCF-UR is more stable to the variation of data dimension.

8. Conclusions and future work

Grid-based DBSCAN is a well-developed algorithm which requires $O(n\log n)$ time to solve for 2-dimensional data. However, we pointed out that it suffered from two problems, i.e. neighbour explosion and merging redundancies, on higher dimensional data. In this paper, we proposed a novel GDCF algorithm to address such problems. GDCF is an improved algorithm of Grid-based DBSCAN which can produce the exact same results as the original DBSCAN with significant improvement on the performance efficiency. Specifically, we devised HGB structure to index non-empty grids for efficient neighbour grid queries. Further, GDCF intergraded a merging management strategy such that we can safely prune an excessive amount of redundant merging computation. We also suggested two orders in the merging, namely UR and LDF, to optimize the merging computation. Although the complexity of proposed algorithm is the same as the traditional Grid-based DBSCAN, the proposed algorithm can run up to three orders of magnitude faster than the traditional Grid-based DBSCAN on the six real-world and synthetic datasets.

For future work, although GDCF can mitigate the redundancies in the merging step at the grid level, the merge-checking between two grids still needs to perform nearest neighbour search. As a result, it will be an interesting direction to develop an approximate algorithm that can exploit the grid shape and determine the merging without fully performing the nearest neighbour search. Second, despite the remarkable success achieved by the current density-based clustering algorithms, it is still not trivial for them to scale to very large data. Therefore, designing parallel and distributed algorithms for grid-based DBSCAN will be an interesting problem such that they can be run on very large databases. Finally, how to select the right parameters of density-based clustering algorithms, e.g. ε and *MinPTS* in DBSCAN, on various datasets remains an open research problem from the data mining perspective.

Acknowledgements

The research work is partially supported by the National Key Research and Development Program of China under Grant No. 2017YFB1002104, the National Natural Science Foundation of China under Grant No. U1811461, 61602438, 91846113, 61573335, the CCF-Tencent Rhino-Bird Young Faculty Open Research Fund No. RAGR20180111. This work is also funded in part by Ant Financial through the Ant Financial Science Funds for Security Research.

References

[1] L. Deutsch, D. Horn, The weight-shape decomposition of density estimates: a framework for clustering and image analysis algorithms, Pattern Recognit. 81 (2018) 190–199.
[2] J. Yu, R. Hong, M. Wang, J. You, Image clustering based on sparse patch alignment framework, Pattern Recognit. 47 (11) (2014) 3512–3519.
[3] M. Devanne, S. Berretti, P. Pala, H. Wannous, M. Daoudi, A.D. Bimbo, Motion segment decomposition of rgb-d sequences for human behavior understanding, Pattern Recognit. 61 (2017) 222–233.
[4] M. Carullo, E. Binaghi, I. Gallo, An online document clustering technique for short web contents, Pattern Recognit. Lett. 30 (10) (2009) 870–876, doi:10.1016/j.patrec.2009.04.001.
[5] H. Xie, G. Tian, H. Chen, J. Wang, Y. Huang, A distribution density-based methodology for driving data cluster analysis: a case study for an extended-range electric city bus, Pattern Recognit. 73 (2018) 131–143.
[6] I.A. Maraziotis, S. Perantonis, A. Dragomir, D. Thanos, K-nets: clustering through nearest neighbors networks, Pattern Recognit. (2018), doi:10.1016/j.patcog.2018.11.010.
[7] J. Wang, Z. Deng, K.-S. Choi, Y. Jiang, X. Luo, F.-L. Chung, S. Wang, Distance metric learning for soft subspace clustering in composite Kernel space, Pattern Recognit. 52 (2016) 113–134.
[8] C. Zhong, D. Miao, R. Wang, A graph-theoretical clustering method based on two rounds of minimum spanning trees, Pattern Recognit. 43 (3) (2010) 752–766.
[9] M. Ester, H.-P. Kriegel, J. Sander, X. Xu, et al., A density-based algorithm for discovering clusters in large spatial databases with noise., in: SIGKDD, 1996.
[10] R.T. Ng, J. Han, Clarans: a method for clustering objects for spatial data mining, IEEE TKDE (2002).
[11] N.A. Yousri, M.S. Kamel, M.A. Ismail, A distance-relatedness dynamic model for clustering high dimensional data of arbitrary shapes and densities, Pattern Recognit. 42 (7) (2009) 1193–1209.
[12] D. Xu, Y. Tian, A comprehensive survey of clustering algorithms, Ann. Data Sci. (2015).
[13] W. Wang, J. Yang, R. Muntz, et al., Sting: a statistical information grid approach to spatial data mining, in: VLDB, 1997.
[14] J.A. Hartigan, M.A. Wong, Algorithm as 136: a k-means clustering algorithm, J. R. Stat. Soc. (1979).
[15] M. Ankerst, M.M. Breunig, H.-P. Kriegel, J. Sander, Optics: ordering points to identify the clustering structure, in: SIGMOD, 1999.
[16] Y. Zhu, K.M. Ting, M.J. Carman, Density-ratio based clustering for discovering clusters with varying densities, Pattern Recognit. 60 (2016) 983–997.
[17] M. Chen, L. Li, B. Wang, J. Cheng, L. Pan, X. Chen, Effectively clustering by finding density backbone based-on knn, Pattern Recognit. 60 (2016) 486–498.
[18] 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.
[19] Fast density clustering strategies based on the k-means algorithm, Pattern Recognit. 71 (2017) 375–386.
[20] C. Böhm, R. Noll, C. Plant, B. Wackersreuther, Density-based clustering using graphics processors, in: CIKM, 2009.
[21] S. Brecheisen, H.-P. Kriegel, M. Pfeifle, Parallel density-based clustering of complex objects, in: W.-K. Ng, M. Kitsuregawa, J. Li, K. Chang (Eds.), Advances in Knowledge Discovery and Data Mining, 2006.
[22] D. Birant, A. Kut, St-dbscan: An algorithm for clustering spatial-temporal data, Data Knowl. Eng., 2007.
[23] P. Kröger, H.-P. Kriegel, K. Kailing, Density-connected subspace clustering for high-dimensional data, in: SIAM, 2004.
[24] X. Xu, M. Ester, H.-P. Kriegel, J. Sander, A distribution-based clustering algorithm for mining in large spatial databases, in: Proceedings of the Fourteenth International Conference on Data Engineering, in: ICDE '98, 1998.
[25] H.-P. Kriegel, M. Pfeifle, Density-based clustering of uncertain data, in: SIGKDD, 2005.
[26] X. Wang, H.J. Hamilton, Dbrs: A density-based spatial clustering method with random sampling, in: K.-Y. Whang, J. Jeon, K. Shim, J. Srivastava (Eds.), Advances in Knowledge Discovery and Data Mining, 2003.
[27] H.-P. Kriegel, M. Pfeifle, '1 + 1 > 2': merging distance and density based clustering, in: DASFAA, 2001.
[28] J.L. Bentley, Multidimensional binary search trees used for associative searching, Commun. ACM (1975).
[29] N. Beckmann, H.-P. Kriegel, R. Schneider, B. Seeger, The r*-tree: an efficient and robust access method for points and rectangles, in: SIGMOD, 1990.
[30] 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.
[31] 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.
[32] B. Borah, D. Bhattacharyya, An improved sampling-based DBSCAN for large spatial databases, in: ICISIP, 2004.
[33] A. Gunawan, M. de Berg, A faster algorithm for DBSCAN, Master's thesis, Technical University of Eindhoven, 2013.
[34] T. Sakai, K. Tamura, H. Kitakami, Cell-based dbscan algorithm using minimum bounding rectangle criteria, in: DASFAA, 2017.
[35] J. Gan, Y. Tao, Dbscan revisited: mis-claim, un-fixability, and approximation, in: SIGMOD, 2015.
[36] B. Welton, E. Samanas, B.P. Miller, Mr. scan: extreme scale density-based clustering using a tree-based network of gpgpu nodes, in: SC, 2013.
[37] M.M.A. Patwary, N. Satish, N. Sundaram, F. Manne, S. Habib, P. Dubey, Pardicle: parallel approximate density-based clustering, in: SC, 2014.
[38] S.T. Mai, I. Assent, M. Storgaard, Anydbc: an efficient anytime density-based clustering algorithm for very large complex datasets, in: SIGKDD, 2016.
[39] A. Zhou, S. Zhou, J. Cao, Y. Fan, Y. Hu, Approaches for scaling dbscan algorithm to large spatial databases, J. Comput. Sci. Technol. (2000).
[40] C.-F. Tsai, C.-T. Wu, S. Chen, Gf-dbscan; a new efficient and effective data clustering technique for large databases, in: MUSP, 2009.
[41] Y. Zhao, C. Zhang, Y.-D. Shen, Clustering high-dimensional data with low-order neighbors, in: WI, 2004.
[42] Z. Yanchang, S. Junde, Agrid: an efficient algorithm for clustering large high-dimensional datasets, in: K.-Y. Whang, J. Jeon, K. Shim, J. Srivastava (Eds.), Advances in Knowledge Discovery and Data Mining, 2003.
[43] S. Mahran, K. Mahar, Using grid for accelerating density-based clustering, in: CIT, 2008.
[44] E. Schubert, J. Sander, M. Ester, H.P. Kriegel, X. Xu, Dbscan revisited, revisited: why and how you should (still) use DBSCAN, in: ACM Trans. Database Syst., 2017.
[45] Y. He, H. Tan, W. Luo, H. Mao, D. Ma, S. Feng, J. Fan, Mr-dbscan: an efficient parallel density-based clustering algorithm using mapreduce, in: ICPADS, 2011.
[46] S.T. Mai, M.S. Dieu, I. Assent, J. Jacobsen, J. Kristensen, M. Birk, Scalable and interactive graph clustering algorithm on multicore cpus, in: ICDE, 2017.

[47] Y. Kim, K. Shim, M.-S. Kim, J.S. Lee, Dbcure-mr: an efficient density-based clustering algorithm for large data using mapreduce, Inf. Syst. 42 (2014) 15–35.
[48] P.K. Agarwal, H. Edelsbrunner, O. Schwarzkopf, E. Welzl, Euclidean minimum spanning trees and bichromatic closest pairs, in: SoCG, 1990.
[49] M. Lichman, UCI machine learning repository, 2013.
[50] A. Reiss, D. Stricker, Introducing a new benchmarked dataset for activity monitoring, in: ISWC, 2012.
[51] J. Sander, M. Ester, H.-P. Kriegel, X. Xu, Density-based clustering in spatial databases: the algorithm gdbscan and its applications, DMKD (1998).
[52] P. Fränti, S. Sieranoja, K-means properties on six clustering benchmark datasets, Appl. Intell. 48 (12) (2018) 4743–4759.

Thapana Boonchoo completed his B.S. degree in Computer Science from Thammasat University, Pathumthani, Thailand, in 2012, and M.S. degree in Computer Science from Tsinghua University, Beijing, China, in 2016. Currently, he is a Ph.D. candidate of the Key Laboratory of Intelligent Information Processing, Institute of Computing Technology, CAS, University of Chinese Academy of Sciences, Beijing, China. His research interests include data clustering algorithms, machine learning, and data mining.

Xiang Ao is an Associate Professor of Institute of Computing Technology, Chinese Academy of Sciences. He received the Ph.D. degree in Computer Science from Institute of Computing Technology, Chinese Academy of Sciences in 2015, and a B.S. degree in Computer Science from Zhejiang University in 2010. His research interests include mining patterns from large and complex data, and financial data mining. He has published several papers in some top tier journals and conference proceedings, such as IEEE TKDE, IEEE ICDE, WWW, IJCAI, SIGIR etc.

Yang Liu received the B.S. degree in Mathematics from Nanjing University, Nanjing, China, in 2017. Currently, he is a Ph.D. candidate of the Key Laboratory of Intelligent Information Processing, Institute of Computing Technology, CAS, University of Chinese Academy of Sciences, Beijing, China. His research interests include machine learning, and data mining.

Weizhong Zhao is currently an Associate Professor in the School of Computer, Central China Normal University, Wuhan, China. He received the B.E. degree and the M.S. degree from Shandong University, P.R. China, in 2004 and 2007 respectively. He received the Ph.D. degree from Institute of Computing Technology, Chinese Academy of Sciences, in 2010. His general area of research is data mining and machine learning.

Fuzhen Zhuang is an Associate Professor in the Institute of Computing Technology, Chinese Academy of Sciences. His research interests include transfer learning, machine learning, data mining, multi-task learning and recommendation systems. He has published more 70 papers in some prestigious refereed journals and conference proceedings, such as IEEE Transactions on Knowledge and Data Engineering, IEEE Transactions on Cybernetics, ACM Transactions on Intelligent Systems and Technology, Information Sciences, Neural Network, IJCAI, AAAI, WWW, ICDE, ACM CIKM, ACM WSDM, SIAM SDM and IEEE ICDM.

Qing He is a Professor as well as a doctoral tutor in the Institute of Computing Technology, Chinese Academy of Science (CAS), and he is a Professor at the University of Chinese Academy of Sciences (UCAS). He received the B.S degree from Hebei Normal University in 1985, and the M.S. degree from Zhengzhou University in 1987, both in mathematics. He received the Ph.D. degree in 2000 from Beijing Normal University in fuzzy mathematics and artificial intelligence. His interests include data mining, machine learning, classification, fuzzy clustering.