Solving DBSCAN using an Algorithm Framework Theoretically, Conceptually and Empirically Fast

1st Hengzhao Ma, 2nd Jianzhong Li
Shenzhen Institute of Advanced Technology, Chinese Academy of Sciences
Shenzhen, China
{hz.ma,lijzh}@siat.ac.cn

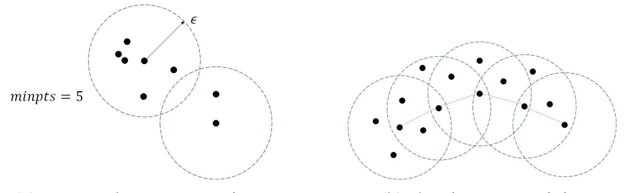
***Abstract*—DBSCAN is a well studied problem with extensive research literature. However, none of the existing works highlight the importance of the abstract algorithm framework proposed by Schubert et al., which decouples the process of identifying core-points and clustering core-points. In this paper, we further refine the algorithm framework proposed by Schubert et al. into four steps, including core-points identification, pre-clustering, final-clustering, and assigning border/noise points, which is referred as the *pre-final clustering* framework in this paper. By instantiating the algorithm framework using different techniques for the four steps, we make great improvement on the DBSCAN problem in theoretical, conceptual, and empirical aspects. Specifically, for the theoretical improvement, we propose a new algorithm, and prove that if the number of pre-clusters obtained by the pre-clustering process is n^δ, the new theoretical algorithm runs in $O(n^{1+\delta}\log n)$ time. When $\delta < \frac{1}{3}$, which can be achieved by setting the DBSCAN parameters in a reasonable range, the new algorithm bypasses Gan and Tao's $\Omega(n^{4/3})$ worst-case lower bound by the instance-specified upper bound. For the conceptual improvement, we propose two new abstract problems, named the ϵ-minpts query and ϵ-connectivity query. It is proved that if the DBSCAN problem is solved by combing these two abstract problems, the number of distance computations is always no more than any algorithm for DBSCAN using range search, k-nearest neighbors or bichromatic closest pair problem as sub-procedures. For the empirical improvement, we show that if the pre-final clustering framework is implemented appropriately, the resulted algorithm achieves better efficiency than state-of-the-art for almost all the tested DBSCAN parameters settings on all the tested datasets.**

***Index Terms*—density-based clustering, DBSCAN, pre-clustering, ϵ-minpts query, ϵ-connectivity query**

I. Introduction

Density-based clustering is an important problem with long research history. In the seminal paper Ester et al. proposed a well-acknowledged method to solve the density-based clustering problem, which is known as DBSCAN [1]. There are two parameters in DBSCAN, ϵ and $minpts$, where ϵ is a positive real number, and $minpts$ is a positive integer. Given an input point set P, the points in P are categorized into *core-points* and *non-core points* based on the two parameters. Specifically, denoting $Ball(p, \epsilon)$ as the set of points within distance ϵ from p, a point p is called core-point if $|Ball(p, \epsilon)| \geq minpts$, and non-core point otherwise. The points are clustered based on the concept of density-connectivity. It is said that two points p and q are density-connected, if there exists a chain of core-points, each within a distance of ϵ to the next one, that can connect p and q. Figure 1 demonstrates the idea of core/non-core points and density-connectivity.

This work was supported by the National Natural Science Foundation of China (Grant number 61832003, 62273322, 61972110), and National Key Research and Development Program of China (Grant number 2021YFF1200100, 2021YFF1200104).



(a) core and non-core points (b) density-connectivity

Fig. 1: Demonstration of DBSCAN clustering concepts.

Schubert et al. proposed the *abstract DBSCAN* framework in their highly cited paper [2], given in Algorithm 1. This framework divides the process of solving DBSCAN into three major steps. The first is to *identify core-points*, i.e., for each point in the input, identify whether it is a core-point. The second is to *cluster core-points*, i.e., based on the information of neighbors obtained in the first step, decide whether the core-points can be joined with each other into clusters. The third is to assign border points and noise points. The essence of this algorithm framework is that it decouples the process of identifying core-points and clustering core-points. We will call it the decoupled DBSCAN framework from now on. By contrast, the original DBSCAN algorithm uses the idea of cluster expansion, i.e., once a core-point is identified, the algorithm will continue to examine the neighbor points of the newly discovered core-point and expand the current cluster. Thus, the process of identifying and clustering core-points are coupled with each other in the original DBSCAN algorithm.

The algorithm framework proposed by Schubert et al. [2] can be further refined. In this paper we divide the step of clustering core-points into two steps, named the *pre-clustering* process and *final-clustering* process. In the pre-clustering process, the information of neighbor points obtained in the core-points identification process is utilized to cluster the core-points roughly. The clusters formed in the pre-clustering process are called pre-clusters. In the final-clustering process, the algorithm examines each pair of pre-clusters, decides whether they are density-connected, and merges them if so. This algorithm framework is called pre-final clustering framework from now on, which is given in Algorithm 2. The idea of pre-final clustering is reflected by several existing works [3]–[7]. However, the power of the pre-final clustering framework is not fully discovered as far as we know.

```
Algorithm 1: Abstract DBSCAN algorithm [2]
1 Compute neighbors of each point and identify
    core-points;        // Identify core-points
2 Join neighboring core-points into clusters;
    // Assign core-points
3 foreach non-core point do
4     Add to a neighboring core-point if possible;
        // Assign border points
5     Otherwise, add to noise ;   // Assign noise
        points
```

```
Algorithm 2: The pre-final clustering framework
1 Compute neighbors of each point and identify
    core-points;        // Identify core-points
2 Use the (possibly incomplete) information of neighbors
    to join core-points into pre-clusters;
    // Pre-clustering
3 For each pair of pre-clusters, decide whether they are
    density-connected, and merge them if so;
    // Final-clustering
4 Assign border and noise points;
```

There are three aspects of advantages for the pre-final clustering framework. The first is theoretical. The pre-clusters have good theoretical properties which can be utilized to design fast DBSCAN algorithm. These properties, which have not been discovered by existing works, will be proposed in Section IV together with the new theoretical algorithm for DBSCAN. The second is conceptual. The pre-final clustering framework is more conceptually flexible which does not necessarily use range search as the sub-procedure for identifying core-points. For the original DBSCAN algorithm, if range search is not used and the points in $Ball(p, \epsilon)$ are not fully discovered, the cluster expansion operation may fail to find the correct maximized cluster. For the decoupled DBSCAN framework, if the points in $Ball(p, \epsilon)$ are not fully discovered in the core-points identification process, the process of assigning core-points may produce wrong clustering results. However, for the pre-final clustering framework, there are other choices as the sub-procedure for identifying core-points such as k-nearest neighbors [6], [8]. Although the points in $Ball(p, \epsilon)$ is not fully discovered by k-nearest neighbors and thus the pre-clustering process may produce clusters that are not maximized, the final-clustering process will discover the possible density-connectivity between the pre-clusters and guarantee the correctness of the algorithm. The third is empirical. There is chance to avoid unnecessary distance computations in empirical algorithms based on the pre-final clustering framework.

TABLE I: Overview of the techniques

	theoretical approach (Section IV)	conceptual approach (Section V)	empirical approach (Section VI)
identifying core-points	All-kNN	ϵ-minpts query	stack-simulated tree traversal
pre-clustering	union-find set	arbitrary technique	union-find set
final-clustering	nearest neighbor query	ϵ-connectivity query	merge-on-find strategy based on lazy-construction fast-merge cover tree
assigning border and noise points	union-find set	arbitrary technique	union-find set

In the core-points identification process, there is chance to reduce the number of distance computations since the points in $Ball(p, \epsilon)$ need not to be fully visited. In the final-clustering process, unnecessary distance computations can be avoided between two points in the same pre-cluster.

Considering the above advantages of the pre-final clustering framework which are not fully exploited by former works, we propose three new approaches to solve the DBSCAN problem, each aiming to utilize one of the advantages of the pre-final clustering framework to the full extent. The three approaches, which will be refereed as the theoretical, conceptual and empirical approaches, are obtained by instantiating the four steps of the pre-final clustering framework using different techniques. Table I overviews the techniques used for the three approaches proposed in this paper, which will be introduced in detail in Section IV, V and VI, respectively. The introduction section is ended with a summary of the contributions achieved by the theoretical, conceptual and empirical approaches.

A. Contributions

First, theoretically, a new theoretical algorithm for DBSCAN is proposed, which runs in $O(n^{1+\delta} \log n)$ time where n^δ is the number of pre-clusters obtained by the pre-clustering process. This algorithm provides an instance-specified upper bound which bypasses Gan and Tao's [9] $\Omega(n^{4/3})$ worst-case lower bound of the DBSCAN problem. For instances with $\delta \geq \frac{1}{3}$, the upper bound recovers the $\Omega(n^{4/3})$ lower bound. For instances with $\delta < \frac{1}{3}$, the upper bound is strictly lower than the $\Omega(n^{4/3})$ worst-case lower bound, which indicates that there exist easy-solvable cases for the DBSCAN problem.

Second, conceptually, two new abstract problems named **ϵ-minpts** query and **ϵ-connectivity** query are proposed. It is proved that if the DBSCAN problem is solved by combining **ϵ-minpts** query and **ϵ-connectivity** query, the number of distance computations is never more than the method using range search, k-nearest neighbors, or bichromatic closest pair problem to solve DBSCAN.

Third, empirically, we propose new efficient methods to implement the pre-final clustering framework, including stack-simulated tree traversal on KDTree (for identifying core-points), and the merge-on-find strategy supported by a new variant of the cover tree [10] with lazy-construction and fast merge operation (for the final-clustering process). Using extensive experimental results, it is shown that the new empirical algorithm can achieve better performance than state-of-the-art in almost all tested parameter settings on all tested datasets.

B. Paper organization

Section II overviews the related works. Section III introduces the preliminary knowledge. In Section IV, V and VI the new theoretical, conceptual and empirical approaches for the DBSCAN problem are introduced, respectively. The experimental results are presented in Section VII. Finally Section VIII concludes this paper.

II. Related Works

A. The decoupled DBSCAN framework and pre-final clustering framework

The decoupled DBSCAN framework proposed by Schubert et al. [2] is highlighted in that paper, but we do not think the idea of decoupling the process of identifying and clustering core-points is highlighted in existing literature. As we have mentioned, an idea of cluster expansion is used in the original DBSCAN algorithm [1], and the process of identifying and clustering core-points are coupled with each other. There exist a lot of algorithms that use the same coupled process [8], [11], [12], and also a lot of algorithms that use the idea of decoupled DBSCAN framework [3], [6], [7], [13]–[15], which reflects that the importance of the decoupled DBSCAN framework is not highly recognized in the society. There are even a lot of existing works that use the idea of pre-final clustering framework, that is to use some fast method to cluster the points roughly, then refine the pre-clusters to obtain the final DBSCAN result. These works include K-DBSCAN [3], μ-DBSCAN [4], Groups-DBSCAN [5], BLOCK-DBSCAN [6], GriT-DBSCAN [7], and so on. However, the idea of pre-final clustering is only implicitly used in these works, and the advantages of the pre-final clustering framework described formerly have not been fully exploited by existing works.

B. Range search and k-nearest neighbors to solve DBSCAN

The original DBSCAN algorithm is based on range search [1]. The highly cited paper of Schubert et al. states that ”the essential question to discussing the complexity of the original DBSCAN algorithm is the runtime complexity of the neighborhood query RangeQuery” [2]. However, until now none of the existing works are aware that range search is actually redundant for solving the DBSCAN problem, and it only needs to solve the **ϵ-minpts** query. We point out that similar technique has been used in other research area, such as the minimal probing principle used for outlier detection [16]. However, such idea as the **ϵ-minpts** query is the first time applied to DBSCAN as far as we know.

There are several works such as [6], [8] that turn to use k-nearest neighbors for solving DBSCAN, but they did not realize that the k-nearest neighbors is still redundant for the DBSCAN problem. Some existing works try to utilize the information of nearest neighbors as much as possible, and identify multiple core/non-core points after a single k-nearest neighbors query [8], [12], [17]. However, they can only reduce a small portion of distance computations but can not fix the root problem, that is, it is not necessary to find the *nearest* neighbors to solve the DBSCAN problem.

III. Preliminaries

Throughout this paper we use P to denote the input point set, ϵ and $minpts$ to denote the two parameters of DBSCAN. For ease of reference ϵ and $minpts$ are called the DBSCAN parameters from now on. The metric space considered in this paper is R^d equipped by L_p metric with arbitrary p. For clearness of representation the Euclidean distance is considered by default, which is L_p metric with $p = 2$. Concretely, each point $p \in P$ is represented by a vector in R^d space, and the distance between two d-dimensional points $x = (x_1, \cdots, x_d)$ and $y = (y_1, \cdots, y_d)$ is measured by the Euclidean distance, i.e., $D(x, y) = \sqrt{\sum_{i=1}^{d}(x_i - y_i)^2}$. Note that we use capital $D(\cdot, \cdot)$ to denote the Euclidean distance function, and d to denote the number of dimension. Let $Ball(p, r)$ be the d-dimensional ball centered at p with radius r. Formally, $Ball(p, r) = \{x \in R^d \mid D(x, p) \leq r\}$. Let $|Ball(p, r) \cap P|$ be the number of points in P covered by $Ball(p, r)$, with p itself counted in.

A. DBSCAN terminologies

Given an input point set P and the DBSCAN parameters ϵ and $minpts$, a point $p \in P$ is called a *core-point* if $|Ball(p, \epsilon) \cap P| \geq minpts$. A point is called a *non-core point* if it is not a core-point.

A point $q \in P$ is said to be *density-reachable* from $p \in P$, or equivalently, p, q are *density-connected*, if there is a sequence of core-points $p_1, p_2, \cdots, p_k \in P$ such that (1) $p = p_1, p_{k+1} = q$, and (2) $D(p_i, p_{i+1}) \leq \epsilon$ for all $1 \leq i \leq k$. If $D(q, p) \leq \epsilon$ it is said that p, q are directly density-connected.

A non-core point $p \in P$ is called a noise point, if no $q \in P$ is density-reachable to p, otherwise it is called a border point.

A cluster $\mathcal{C}$ generated by DBSCAN satisfies the following two properties. (1) Connectivity. $\forall$ two points $p, q \in \mathcal{C}$ are density-connected. (2) Maximality. There is no $p^* \in P \setminus \mathcal{C}$ such that p^* is density-reachable from any point in $\mathcal{C}$.

B. Pre-final clustering framework

Definition 1 (core-points identification problem)**.** ***Input****: point set P, and DBSCAN parameters ϵ and $minpts$.*
Output*: for each $p \in P$, decide whether p is a core-point.*

Definition 2 (pre-cluster)**.** *Given a point set P and the DBSCAN parameters ϵ and $minpts$, a subset $\mathcal{C} \subset P$ is called a pre-cluster if (1) $\forall p \in \mathcal{C}$ is a core-point, and (2) $\forall$ two points $p, q \in \mathcal{C}$ are density-connected.*

A pre-cluster may not be maximized, i.e., there may exist $q \in P \backslash \mathcal{C}$ such that q is density-reachable from some point in $\mathcal{C}$. That is why it is called pre-cluster, since its maximality must be finally checked to produce the correct DBSCAN clustering result. In this sense, we define the process of pre-clustering and final-clustering. The process to assign border/noise points is easy to understand and thus its definition is omitted.

Definition 3 (pre-clustering process)**.** ***Input****: point set P, and DBSCAN parameters ϵ and $minpts$.*

Output: *a set of pre-clusters $\mathbb{C} = \{\mathcal{C}_1, \cdots, \mathcal{C}_l\}$, such that for each core-point $p \in P$, $\exists \mathcal{C}_i \in \mathbb{C}$ satisfying $p \in \mathcal{C}_i$.*

Definition 4 (final-clustering process)**.** ***Input***: *a set of pre-clusters $\mathbb{C} = \{\mathcal{C}_1, \mathcal{C}_2, \cdots, \mathcal{C}_l\}$.*
Output: *for each pair of pre-clusters $\mathcal{C}_i, \mathcal{C}_j$, decide whether they are density-connected, and merge them if so. The output is a set of clusters $\mathbb{C}' = \{\mathcal{C}'_1, \cdots, \mathcal{C}'_m\}$, such that for each core-point p, if $p \in \mathcal{C}'_i$ then all core-points density-reachable from p are in $\mathcal{C}'_i$.*

Note that although a pre-cluster may not be maximized, the set of pre-clusters generated by the pre-clustering process must include all the core-points. This ensures that after the final-clustering process, the resulted set of clusters $\mathbb{C}' = \{\mathcal{C}'_1, \cdots, \mathcal{C}'_m\}$ are all maximized except for the border points. Therefore, the clusters generated by the final-clustering process, plus joining the border points together, are the same with the clusters generated by the original DBSCAN algorithm. We make the following claim about the correctness of the pre-final clustering framework, whose proof is already explained.

Theorem 1. *The clustering result generated by Algorithm 2 is exactly the same with the original DBSCAN algorithm.*

C. Problems related to DBSCAN

Definition 5 (range search)**.** ***Input***: *point set P, query point q, and search range $r > 0$.*
Output: *the set of points $Ball(q, r) \cap P$.*

We will say p is an ϵ-neighbor of q if $D(p, q) \leq \epsilon$. In this sense, the range search on q asks to find all ϵ-neighbors of q.

Definition 6 (k-nearest neighbors, kNN)**.** ***Input***: *a point set P and a query point q.*
Output: *the set of the k nearest neighbors of q in P, denoted as $NN(k, q, P)$, which satisfies $|NN(k, q, P)| = k$ and $D(q, p) \leq D(q, p')$ for $\forall p \in NN(k, q, P)$ and $p' \in P \setminus NN(k, q, P)$.*

It is assumed that there are no two points $p, p' \in P$ with the same distance to q, and thus the set $NN(k, q, P)$ is unique.

Definition 7 (bichromatic closest pair)**.** ***Input***: *point sets P_1, P_2.*
Output: *a pair of points $p_1 \in P_1, p_2 \in P_2$ such that $D(p_1, p_2) \leq D(p'_1, p'_2)$ for any $p'_1 \in P_1, p'_2 \in P_2$.*

IV. The new theoretical approach

The techniques used in the theoretical approach in terms of the pre-final clustering framework are given in Algorithm 3, which will be introduced in the rest of this section. Due to space limitation, all the proofs are omitted in this paper and please see our full paper online [18].

A. Identifying core-points using All-kNN

Denote $NNDist(k, p, P) = \max_{q \in NN(k,p,P)} D(q, p)$ and call it the k-th NN distance of p in P. The following lemma shows the relation between core-point and k-nearest neighbors problem, which also appears in several existing works [6], [8].

Algorithm 3: The new theoretical algorithm

```
1 Identifying core-points: use the All-$k$NN algorithm;
2 Pre-clustering: use union-find set;
3 Final-clustering: use nearest neighbor;
4 Assign border and noise points;
```

Lemma 1. *Given a point set P and the DBSCAN parameters ϵ and $minpts$, for $\forall p \in P$, p is a core-point iff $NNDist(minpts, p, P) \leq \epsilon$,*

Proof. By definition, $NNDist(minpts, p, P)$ is the value r such that $|Ball(p, r) \cap P| = minpts$. Thus, if $NNDist(minpts, p, P) \leq \epsilon$ then $|Ball(p, \epsilon) \cap P| \geq |Ball(p, NNDist(minpts, p, P)) \cap P| = minpts$, which meets the definition of core-points and makes p a core-point. On the other hand, if p is a core-point, by definition it holds that $|Ball(p, \epsilon) \cap P| \geq minpts$. Since $|Ball(p, NNDist(minpts, p, P)) \cap P| = minpts$, it must hold that $NNDist(minpts, p, P) \leq \epsilon$. □

By the above lemma, the problem of core-point identification can be solved by computing the kNN results for each point in P setting $k = minpts$. There exists a well-studied problem called All-k-Nearest-Neighbors, All-kNN for short, to solve the above problem. It is defined as follows.

Definition 8 (All-kNN)**.** ***Input***: *point set P, and integer k.*
Output: *for each $p \in P$ compute $NN(k, p, P)$.*

The following result for All-kNN is well known.

Theorem 2 ([19]–[21])**.** *The All-kNN problem can be solved in $O(d^d n \log n)$ time.*

With the aid of the All-kNN algorithm promised by Theorem 2, the following algorithm is used to identify core-points in the input point set P. Invoke the All-kNN algorithm on P, setting $k = minpts$, and obtain $NN(minpts, p, P)$ and $NNDist(minpts, p, P)$ for each point $p \in P$. Then for $\forall p \in P$, mark it as core-point if $NNDist(minpts, p, P) \leq \epsilon$, and non-core point otherwise. This algorithm obviously runs in $O(d^d n \log n)$ time.

B. Pre-clustering using union-find set

The idea of using union-find set to solve DBSCAN is not new [11], [13], [22]. The union-find set is a very efficient data structure to represent a set of disjoint sets. Given a set P of elements, the union-find set stores a representative element $rep(p)$ for each element $p \in P$. There are two operations $union$ and $find$ for the union-find sets. The $find(p)$ operation will return the representative $rep(p)$. The $union(p, q)$ will set the representative element for p and q as the same. The following theorem for the time complexity of union-find sets is well known. Note that the inverse Ackerman function grows very slowly with n, and can be usually regarded as a constant.

Theorem 3 ([23]). *For a union-find set storing n elements, a sequence of m union and $find$ operations on the union-find set takes $O(m\alpha(n))$ time, where $\alpha(n)$ is the inverse Ackermann function.*

Algorithm 4 shows the pseudo codes for pre-clustering using union-find set. The algorithm takes the output of the All-kNN algorithm as input, and form a set of pre-clusters $\{\mathcal{C}_1, \mathcal{C}_2, \cdots \mathcal{C}_l\}$. The time complexity is given below.

Theorem 4. *Algorithm 4 runs in $O(n \cdot \alpha(n))$ time, where $\alpha(n)$ is the reverse-Ackemann function.*

Algorithm 4: Pre-clustering using union-find set

```
Input: The input point set $P$ and the results obtained
       by All-$k$NN on $P$.
Output: A set of pre-clusters $\{\mathcal{C}_1, \mathcal{C}_2, \cdots \mathcal{C}_l\}$.
1 Initialize the union-find set UFS on $P$, where each
    point $p \in P$ forms an individual subset;
2 foreach core-point $p$ do
3     foreach $q \in NN(minpts, p, P)$ do
4         if $q$ is core-point then
5             UFS.$union(q, p)$;
6 Initialize empty set of clusters;
7 foreach core-point $p$ do
8     Add $p$ into the cluster whose representitive is
        UFS.$find(p)$;
```

C. Final-clustering using nearest neighbor

We first define the direct density-connectivity for two pre-clusters as follows.

Definition 9. *Given two pre-clusters $\mathcal{C}_i$ and $\mathcal{C}_j$, they are called directly density-connected if there exists $p \in \mathcal{C}_i$ such that $NNDist(1, p, \mathcal{C}_j) \leq \epsilon$.*

It is enough to check the directly density-connectivity for each pair of pre-clusters to solve the final-clustering process, which is equivalent to performing the following nearest neighbor query. For each point $p \in \mathcal{C}_i$, find the nearest neighbor of p in $\mathcal{C}_j$. The idea is quite simple but the central problem is to bound the complexity of the nearest neighbor query. What we need is an algorithm satisfying the following Theorem 5.

Theorem 5. *Given a pre-cluster $\mathcal{C}$ and a query point q, there exists an algorithm that takes $O(|\mathcal{C}| \log |\mathcal{C}|)$ time to build an index structure based on $\mathcal{C}$, and finds the nearest neighbor of q in $\mathcal{C}$ in $O(\log |\mathcal{C}|)$ time. The Big-O notation suppresses a factor exponentially depending on the number of dimension d.*

The nearest neighbor (NN) problem is intensively studied in the past years, but there is no existing algorithm that can achieve the complexity claimed in Theorem 5. As a brief overview, Arya et. al. proposed an algorithm with $O(n \log n)$ preprocessing time complexity and $O(\log n)$ query time complexity, but it can only provide approximate result [24]. Another algorithm by Arya et. al can answer the nearest neighbor query in $O(\log n)$ time but needs $O(n^2)$ preprocessing time [25]. The navigating-nets technique achieves $O(\log \Delta + |Ball(q, O(D(q, p^*)))|)$ query time [26], where Δ is the ratio between the maximum and minimum pairwise distance in the point set P, and $D(q, p^*)$ is the distance of the nearest neighbor of q. While the $O(\log \Delta)$ term can be bounded using the property of the pre-clusters (See Lemma 4 below), the $|Ball(q, O(D(q, p^*)))|$ term can be unbounded. The cover tree data structure is claimed that can find the nearest neighbor in $O(\log n)$ time, but this claim turns out to be incorrect [27], and this running time only applies to datasets that have bounded expansion constant [28]. As a result, we need a new algorithm to prove Theorem 5. Note that the NN algorithm in this section takes a pre-cluster as input. The central idea of designing an efficient NN algorithm is to utilize the properties of the pre-clusters, which will be proposed next. We need to define the following constant value denoted as

$$MinptsNNDist(\mathcal{C}) = \min_{p \in \mathcal{C}} \{NNDist(minpts, p, \mathcal{C})\}$$

This is the minimum of the $minpts$-th NN distance among points in $\mathcal{C}$. Note that $MinptsNNDist(\mathcal{C})$ is a known constant since it can be obtained by examining the All-kNN results computed in core-points identification process.

Lemma 2. *Given a pre-cluster $\mathcal{C}$ and the DBSCAN parameters ϵ and $minpts$, let $Diam(\mathcal{C}) = \max_{p,q \in \mathcal{C}} D(p, q)$, then it holds that $Diam(\mathcal{C}) \leq |\mathcal{C}|\epsilon$.*

Proof. Since $\mathcal{C}$ is a pre-cluster, for each pair of point p, q, there exists a sequence of points $p_0, p_1, \cdots, p_m, p_{m+1} \in \mathcal{C}$ such that $p_0 = p, p_{m+1} = q$, and $D(p_i, p_{i+1}) \leq \epsilon$ for all $0 \leq i \leq m$. The maximum length of the sequence is obviously $|\mathcal{C}|$, and then we have $D(p, q) \leq |\mathcal{C}|\epsilon$ for all $p, q \in \mathcal{C}$, which proves the lemma. ☐

Lemma 3. *Suppose the DBSCAN problem is defined on R^d equipped by L_p metric with arbitrary p, then for any pre-cluster $\mathcal{C}$ it holds that $\frac{\epsilon}{MinptsNNDist(\mathcal{C})} = O(|\mathcal{C}|^{1/O(d)})$.*

Proof. The lemma is specialized for R^d equipped with L_p metric with arbitrary p, where the following claim holds [26]. For any ball $Ball(x, r)$ where $x \in R^d$ and $r > 0$, it can be covered by at most $2^{O(d)}$ number of balls of radius $r/2$. By applying this result repeatedly, it can be deduced that a ball with radius ϵ can be covered by at most $(\frac{\epsilon}{MinptsNNDist(\mathcal{C})})^{O(d)}$ number of balls of radius $MinptsNNDist(\mathcal{C})/2$.

By the definition of $MinptsNNDist(\mathcal{C})$, for any point $p \in \mathcal{C}$ it holds that $|Ball(p, MinptsNNDist(\mathcal{C})) \cap \mathcal{C}| \leq minpts$. We then claim that $|Ball(x, MinptsNNDist(\mathcal{C})/2) \cap \mathcal{C}| \leq minpts$ for $\forall x \in R^d$, since otherwise there exists $p \in Ball(x, MinptsNNDist(\mathcal{C})/2)$ such that $Ball(p, MinptsNNDist(\mathcal{C})) \supset Ball(x, MinptsNNDist(\mathcal{C})/2)$, and then $|Ball(p, MinptsNNDist(\mathcal{C})) \cap \mathcal{C}| \geq |Ball(x, MinptsNNDist(\mathcal{C})/2) \cap \mathcal{C}| > minpts$, which contradicts with the definition of $MinptsNNDist(\mathcal{C})$.

Combining the above two parts, for any point $x \in R^d$ we have $|Ball(x,\epsilon) \cap \mathcal{C}| \le (\frac{\epsilon}{MinptsNNDist(\mathcal{C})})^{O(d)} \cdot minpts \le |\mathcal{C}|$. This is equivalent with $\frac{\epsilon}{MinptsNNDist(\mathcal{C})} \le (\frac{|\mathcal{C}|}{minpts})^{1/O(d)} = O(|\mathcal{C}|^{1/O(d)})$. The $minpts$ factor is suppressed by the Big-O notation since it is a constant. By now the lemma is proved. $\square$

Lemma 4. *Suppose the DBSCAN problem is defined on R^d equipped with L_p metric with arbitrary p, then for any pre-cluster $\mathcal{C}$ it holds that $\frac{Diam(\mathcal{C})}{MinptsNNDist(\mathcal{C})} = O(|\mathcal{C}|^{1+1/O(d)})$.*

Having these results about pre-clusters, we are ready to propose the nearest neighbor algorithm taking pre-clusters as input. The algorithm is divided into three steps, which are building a fair split tree, adding the query point q into the fair split tree, and finally performing the NN algorithm for q on the new tree.

1) Build Fair Split Tree: The fair split tree is proposed by Bespamyatnikh et al. [29], which is used here with slight modification. We need to recite the following denotations from [29]. A d-dimensional box is defined as the Cartesian product of d semi-closed intervals, i.e., $[l_1, r_1) \times \cdots \times [l_d, r_d)$, and it is referred simply as a box if the number of dimension d is clear in the context. The interval $[l_i, r_i)$ is referred as the i-th interval, and the length of the interval $[l_i, r_i)$ is referred as the i-th side length of the box. A box with side length $s_1, \cdots s_d$ is called almost-cubical, if $s_i/s_j \in [1/3, 3]$ for all $1 \le i, j \le d$. A box B is split into two sub-boxes by an almost-middle cut, if the split hyperplane is $x_i = c$ where $c \in [\frac{l_i+2r_i}{3}, \frac{2l_i+r_i}{3}]$. For two boxes A and B, A is said to be a s-sub-box of B if there exists a sequence of boxes $B_0, B_1, B'_1 \cdots B_l, B'_l$ such that $B = B_0, A = B_l$, and the boxes B_i, B'_i are obtained by almost-middle cut on B_{i-1}, for $1 \le i \le l$. A tree structure T can be constructed based on the almost-middle-cut and s-sub-boxes relationship, which is called the fair split tree. For each node v of the tree T, two boxes $B(v)$ and $SB(v)$ are stored in v, where $SB(v)$ is called the shrunken box. The fair split tree T satisfies the following conditions.

Definition 10. *A fair split tree built on a pre-cluster $\mathcal{C}$ is a tree structure satisfying the following requirements.*

1) *$B(root)$ is the smallest almost-cubical box that contains all the points in $\mathcal{C}$, where $root$ is the root node.*
2) *$B(v)$ and $SB(v)$ are almost-cubical for any node v.*
3) *$SB(v)$ is a s-sub-box of $B(v)$ for any ndoe v.*
4) *$B(v)$ and $SB(v)$ contain the same point set for any node v.*
5) *If w has two children u and v, then the boxes $B(u)$ and $B(v)$ are the results obtained by performing an almost-middle-cut on $SB(w)$.*
6) *If the length of the longest diagonal of a box $B(v)$ is less than $MinptsNNDist(\mathcal{C})$, then $B(v)$ will not be further split, and it is a leaf.*

Note that the sixth term is the only difference between the fair split tree here and the tree structure described in [29], where in that paper the tree is split such that the leaf node contains only one point. The time complexity to build the fair split tree was not given in [29], but can be proved to be $O(dn \log n)$. We first need the following two lemmas, which are inspired by the techniques used in [20].

Lemma 5 (Split time lemma). *For a box $B = [l_1, r_1) \times \cdots \times [l_d, r_d)$, and a split hyperplane $x_i = c$ for arbitrary i and arbitrary $c \in [l_i, r_i)$, the hyperplane will split B into two sub-boxes $[l_1, r_1) \times \cdots \times [l_i, c) \times \cdots \times [l_d, r_d)$ and $[l_1, r_1) \times \cdots \times [c, r_i) \times \cdots \times [l_d, r_d)$. Let B_{large} the one in the two sub-boxes containing more points, called the larger son, and B_{small} be the other called the smaller son. The time to obtain B_{large} and B_{small}, as well as the point sets contained in them, is $O(|B_{small}|(1 + \log_2 |lsa(B)| - \log_2 |B_{small}|))$. Let $lsa(B)$ denote the lowest smaller ancestor of the box B, which is the one box in the path from B to the root which is closest to B and is a smaller son of its parent.*

Proof. We first need to cite the following ancestor lemma from [20].

Lemma 6 (Ancestors Lemma, [20])**.** *Let T be a complete binary tree of height $h(T)$ containing $2^{h(T)} - 1$ vertexes and let L be a set of leaves in T. A proper ancestor of a leaf in a rooted tree is non-leaf vertex on the path from the root to the leaf. Let $Anc(L, T)$ be the set of all the proper ancestors in T of the leaves in L. Then $|Anc(L, T)| \le |L|(h(T) - \log |L|) - 1$.*

Let $B \cap P$ be the subset of input points P contained in box B, and let $|B|$ be an abbreviation of $|B \cap P|$. For each vertex in the fair split tree and the corresponding box B, d ordered lists $List_i(B)$ are associated to the vertex. The i-th ordered list contains the coordinates of the points in $B \cap P$ on the i-th coordinate, $1 \le i \le d$. From an entry for a point p in each of these lists there are pointers to the entries for p in all the remaining $d-1$ lists. For each $1 \le i \le d$, the list $List_i(B)$ is embedded in the leaves of a corresponding complete binary search tree $T_i(B)$ of height $\lceil \log_2 |lsa(B)| \rceil$. The leaves of $T_i(B)$ are the points in $lsa(B) \cap P$ ordered on the i-th coordinate. However, only the leaves that are points in $B \cap P$ are linked together to form the ordered list $List_i(B)$. In the process of splitting B to obtain B_{large} and B_{small}, it need to obtained the d ordered lists for the two sub-boxes. The ordered list $List_i(B_{small})$ will be explicitly created from $List_i(B)$ and will be embedded in newly created binary search tree $T_i(B_{small})$ of height $\lceil log_2 |B_{small}| \rceil$. The ordered list $List_i(B_{large})$ will be implicitly obtained by deleting the points in $B_{small} \cap P$ from $List_i(B)$.

We first obtain the points in $B_{small} \cap P$ in time proportional to $|B_{small}|$. Recall that the split hyperplane is $x_i = c$, where $1 \le i \le d$ and $c \in [l_i, r_i)$. The set $B_{small} \cap P$ can be obtained by searching the ordered list $List_i(B)$ from both ends simultaneously, and stopping the first time the split hyperplane $x_i = c$ is crossed. The time is obviously $O(|B_{small}|)$.

Once the set $B_{small} \cap P$ is obtained, we get the corresponding k ordered lists as follows. For the i-th coordinate, we fist label all the points in $List_i(B)$ that are located in B_{small}. We then label all the vertexes in $T_i(B)$ that are proper ancestors of the labeled points in $List_i(B)$. The labeling of the proper ancestors can be performed in time proportional to the number of proper ancestors plus the number of points in $B_{small} \cap P$, and that is $O(|B_{small}|(1 + \log_2 |lsa(B)| - \log_2 |B_{small}|))$ according to the Ancestors Lemma. Next we traverse the labeled vertexes of $T_i(B)$ in-order and thereby obtain the labeled points in $List_i(B)$, which is the points in $B_{small} \cap P$ in sorted order on the i-th coordinate. The in-order traversal may be performed in time proportional to the number of labeled vertices. Finally, remove the labels of all the labeled vertexes.

After the d ordered lists for B_{small} have been obtained, all the points in B_{small} are deleted from each of the lists $List_i(B)$, $1 \le i \le d$. The complete binary search trees $T_i(B_{small})$ are the created based on the lists in $O(|B_{small}|)$ time.

Finally, note that deleting the points in $B_{small} \cap P$ from $List_i(B)$ implicitly leaves behind the points in B_{large}. So the d ordered lists for the points in $B_{large} \cap P$ are available now. Since $lsa(B_{large}) = lsa(B)$, a suitable embedding of these lists in a complete binary search tree of height $\lceil log_2 |lsa(B_{large})| \rceil$ is also available now.

In conclusion, the entire process of splitting B and obtaining B_{large} and B_{small} can be done in $O(d|B_{small}|(1 + \log_2 |lsa(B)| - \log_2 |B_{small}|))$ time.

□

Lemma 7 (Tree weighting lemma [20]). *Let T be a rooted tree with totally t vertices such that each non-leaf vertex has at least two and at most r children. Define the weight $w(v)$ of a non-leaf vertex by $w(v) = |B(v)|(1 + \log_2 |Ball(lsa(v))| - \log_2 |B(v)|)$, and define the weight of a leaf to be 0. Then $\sum_{v \in T} w(v) \le 4rtlog_2 t$.*

The time complexity of building the fair split tree can be directly deduced from the above two lemmas.

Theorem 6. *The time to build the fair split tree is $O(dn \log n)$.*

We also need the following two lemmas concerning the properties of the fair split tree.

Lemma 8. *Let B be a box in the leaf node of the fair split tree. The maximum number of points contained in B is $minpts$.*

Proof. By the sixth term in Definition 10, the longest diagonal of B is at most $MinptsNNDist(\mathcal{C})$. Denote $Dmax(B) = \max_{p,q \in B} D(p, q)$ which is the maximum pairwise distance among the points in B. Since the longest diagonal serves as a upper bound on $Dmax(B)$, it holds that $Dmax(B) \le MinptsNNDist(\mathcal{C})$.

If B contains more than $minpts$ number of points, then for any point $p \in B$, since $Ball(p, Dmax(B)) \cap P \supset B \cap P$ we have $|Ball(p, Dmax(B)) \cap P| \ge |B \cap P| > minpts$. Further, $|Ball(p, MinptsNNDist(\mathcal{C}))| \ge |Ball(p, Dmax(B)) \cap P| > minpts$. This contradicts with the definition of $MinptsNNDist(\mathcal{C})$, which proves that the number of points that B contains is no more than $minpts$. □

Lemma 9. *The height of the fair split tree is $O(d \log (|\mathcal{C}|d))$.*

Proof. By the first term in Definition 10, the side length of the box $B(root)$ stored in the root of the tree must be no more than $Dmax(\mathcal{C})$, since otherwise it must not be the smallest almost-cubical box that contains all the points in $\mathcal{C}$. Now we conduct the almost-middle-cut operation repeatedly on B_{max} and construct the fair split tree. By the definition of the almost-middle-cut, each such almost-middle-cut operation decreases the side length of a box in one dimension by a constant factor. Thus, using at most

$$O(d \log \frac{Dmax(\mathcal{C})}{d^{1/p} MinptsNNDist(\mathcal{C})}) = O(dp \log \frac{d \cdot Dmax}{MinptsNNDist(\mathcal{C})})$$

splits, the side length of each dimension will be no more than $d^{1/p} \cdot MinptsNNDist(\mathcal{C})$, and in this case the longest diagonal of this box is no more than $MinptsNNDist(\mathcal{C})$. By the sixth term in Definition 10, the process of splitting the box should be stopped and the leaf nodes are reached. The number of almost-middle-cut is exactly the height of the fair split tree. By Lemma 4, $Dmax/MinptsNNDist(\mathcal{C}) = |C|^{1+1/O(d)}$. Thus, the height of the tree is at most

$$O(dp \log \frac{d \cdot Dmax}{MinptsNNDist(\mathcal{C})}) = O(d \cdot p \cdot (1+1/O(d)) \log (|C|d)) = O(d$$

□

2) Add the query point into the tree: The process of adding a point into a fair split tree is also described in [29]. We use exactly the same process. The only thing needed to be cited here is the time complexity of maintaining the tree by adding or deleting a query point, which is given in Theorem 7. Note that after adding the query point, the height of the fair split tree can grow by at most 1, and the number of points contained in a leaf node would not increase.

Theorem 7 ([29]). *The time complexity to add a point q into a fair split tree is $O(\log n)$. The time complexity to delete one point from a fair split tree is $O(1)$.*

3) The NN algorithm: The following definitions are needed to introduce the algorithm. Given two boxes B_1, B_2 in a fair split tree, which are disjoint according to the definition, let $Dmin(B_1, B_2)$, $Dmax(B_1, B_2)$ be the minimum and maximum distance between a vertex point of the box B_1 and a vertex point of the box B_2, respectively. It is easy to see that $Dmin(B_1, B_2) \le \min_{p_1 \in B_1, p_2 \in B_2} D(p_1, p_2)$ and $Dmax(B_1, B_2) \ge \max_{p_1 \in B_1, p_2 \in B_2} D(p_1, p_2)$. Let $Dmax(B)$ be the maximum distance between two vertex points of B, then $Dmax(B) \ge \max_{p_1, p_2 \in B} D(p_1, p_2)$. Note that these three values can all be computed in $O(d^2)$ time. Define the set of neighbor boxes for a box B in the fair split tree T as

$$Nbr(B) = \{B' \in T \mid Dmin(B, B') \le LoBound(B)\}$$

where $LoBound(B)$ is defined as

$$LoBound(B) = \begin{cases} Dmax(B), if\ |B| \geq 2 \\ \min\limits_{B' \in Nbr(B)} Dmax(B, B'), otherwise \end{cases}$$

The definition of the Nbr set is to ensure the following lemma, which was proved in [20].

Lemma 10. *Given a box B, the nearest neighbor of any point $p \in B$ is included by some box $B' \in Nbr(B)$.*

Proof. Since $LoBound(B)$ serves as an upper bound on the nearest neighbor distance of any point in B, then $Nbr(B)$ will never throw out a box that possibly contains a nearest neighbor point. The detailed prove is omitted. ☐

The NN algorithm given in Algorithm 5 maintains a heap H. Only the boxes containing the query point q and the boxes in $Nbr(B)$ where B is some box containing q will be stored in the heap H. The following invariants are ensured.

Algorithm 5: NN based on the fair split tree

```
Input: A fair split tree $T$, obtained by adding query
       point $q$ into the tree build on $P$.
Output: The nearest neighbor of $q$.
1  Initialize a heap $H$ to store the boxes in $T$ to be
     visited , where the boxes are ordered by the
     maximum side length;
2  Push the root of $T$ into $H$;
3  Let $B_c = root(T)$;
   // Invariant: $B_c$ contains $q$ and is the
      samllest one among all the visited
      boxes containing $q$;
4  while $H$ is not empty do
5      $B \leftarrow$ the top element in $H$;
       // Invariant: $B$ is included by
          $Nbr(B_f) \cup Nbr(B_c)$ where $B_f$ is the
          father of $B_c$;
6      if $B$ contains $q$ then
7          Let $B^*$ be the one child of $B$ that contains $q$,
             and $B'$ be the other;
8          $Nbr(B^*) \leftarrow Nbr(B) \cup \{B'\}$;
9          Update $Nbr(B^*)$, deleting those not satisfying
             the definition of $Nbr(B^*)$;
10         Push $B^*$ to $H$;
11         $B_c \leftarrow B^*$;
12     else
13         Delete $B$ from $Nbr(B_c)$;
14         foreach $b \in Child(B)$ do
15             if $b$ satisfies the definition of $Nbr(B_c)$ then
16                 Add $b$ into $Nbr(B_c)$;
17                 Push $b$ into $H$;
18 Find the nearest neighbor of $q$ in all the points
     contained in $B_c \cup Nbr(B_c)$;
```

Lemma 11. *The two invariants stated in Algorithm 5 holds. Invariant 1: B_c contains q and is the smallest one among all the visited boxes containing q.*
Invariant 2: The box B visited in the $while$ loop is included by $Nbr(B_f) \cup Nbr(B_c)$ where B_f is the parent node of B_c.

Proof. Invariant 1: B_c contains q and is the smallest one among all the visited boxes containing q. The invariant holds at the beginning of the algorithm, since B_c is initially set to the box at the root. The invariant is maintained in Line 11 when a smaller boxes containing q is visited.

Invariant 2: The box B visited in the $while$ loop is included by $Nbr(B_f) \cup Nbr(B_c)$ where B_f is the parent node of B_c. This invariant holds at the beginning of the algorithm, since the first box visited is the root. The invariant is ensured by Line 10 and 17, since the box will not be added into the heap if B does not contain q, or is not added into $Nbr(B_c)$. ☐

The correctness of Algorithm 5 is easy to prove by combining Lemma 10 and the first invariant in Lemma 11. Actually, the last line in Algorithm 5 must return the nearest neighbor of q since q is contained in B_c and all possible nearest neighbors are contained in $Nbr(B_c)$.

To analyze the complexity of the algorithm, we first have the following corollary by combining the two variants proved in Lemma 11.

Corollary 1. *The box visited in Algorithm 5 is (1) either a box containing q, (2) or a box included by $Nbr(B_q)$ where B_q is a box contains q.*

According to Corollary 1, the number of boxes visited in Algorithm 5 is, the number of boxes in T that contains q, multiplying the size of Nbr set of any box. We then have to bound the two quantities.

Lemma 12. *The number of boxes in T that contains q is $O(d \log(|C|d))$.*

Proof. This is obvious because the height of the tree is $O(d \log(|C|d))$ (Lemma 9), and the boxes in the fair split tree can not overlap. ☐

Lemma 13. *For each box B in T, $|Nbr(B)| = O(d^d)$.*

Proof. To prove this lemma we need to borrow techniques from [20] and [21].

For a box B whose i-th interval is $[l_i(B), r_i(B))$, let $Len_i(B) = r_i(B) - l_i(B)$ be the i-th side length of B, $1 \leq i \leq d$. Let $Lmin(B) = \min\limits_{1 \leq i \leq d} Len_i(B)$ and $Lmax(B) = \max\limits_{1 \leq i \leq d} Len_i(B)$ be the maximum and minimum side length of B. A box B is called a d-cube if $Lmin(B) = Lmax(B)$.

We first prove the following inequalities.

1. For a vertex v and its parent vertex v_p in the fair split tree T, it holds that $Lmin(B(v)) \geq \frac{1}{27} Lmax(SBall(v_p))$. By the definition of the fair split tree, the box $B(v)$ is obtained by an almost-middle-cut on $SBall(v_p)$. Thus, for each side of $B(v)$, either $Len_i(B(v)) = Len_i(SBall(v_p))$ if the cut is not on the i-th coordinate, or $Len_i(B_v) \geq \frac{1}{3} Len_i(SBall(v_p))$ according to the definition of the almost-middle-cut. In either case, it holds that $L_i(B_v) \geq \frac{1}{3} L_i(SBall(v_p))$, $1 \leq i \leq d$. By the

almost-cubical property of the boxes, we have $Lmin(B_v) \geq \frac{1}{3}L_i(B_v)$ and $L_i(SBall(v_p)) \geq \frac{1}{3}Lmax(SBall(v_p))$. Then it can be proved that $Lmin(B(v)) \geq \frac{1}{27}Lmax(SBall(v_p))$.

2. For any box in R^d it holds that $Dmax(B) \leq d \cdot Lmax(B)$. By definition $Dmax(B) = \max_{p,p' \in B} D(p,p')$. Note that the distance function here is the L_q metric, and the L_q distance is bounded by d times of the L_∞ distance between two points. Then the inequality $Dmax(B) \leq d \cdot Lmax(B)$ follows.

Now we consider the execution of Algorithm 5. At the beginning time of each $while$ loop, let B_c be the box described in the algorithm which is the one containing the query point q. Let $B(v)$ be any box that contained in $Nbr(B_c)$ and v_p be the parent vertex of v. Let B_{top} be the current top element in the heap H. We continue to propose the following inequalities.

3. It holds that $Lmax(SBall(v_p)) \geq Lmax(B_{top})$. The reason is that $SBall(v_p)$ has already been visited by the $while$ loop, and the heap H is ordered by the maximum side length of the boxes.

4. For any box $B(v) \in Nbr(B_c)$ it holds that $Lmax(B(v)) \leq LmaxB_{top}$. This is because B_{top} is visited before any box $B(v) \in Nbr(B_c)$.

5. Define $MaxGap = \max_{B(v)\in Nbr(B_c), 1\leq i \leq d} \{l_i(B_c) - r_i(B(v)), l_i(B(v)) - r_i(B_c)\}$. Intuitively, $MaxGap$ is the maximum of the gap between a left (right) face of B_c and a right (left) face of $B(v) \in Nbr(B_c)$. Let B_{mg} be the box that realizes $MaxGap$. It must hold that $MaxGap \leq Dmin(B_c, B_{mg})$. Since B_{mg} and B_c is totally separated at a distance of $MaxGap$, then any point $p_1 \in B_{mg}$ and $p_2 \in B_c$ are separated at least $MaxGap$ distance. It then means that $Dmin(B_c, B_{mg}) \geq MaxGap$. Furthermore, we have $MaxGap \leq Dmin(B_c, B_{mg}) \leq Dmax(B_c) \leq d \cdot Lmax(B_c) \leq d \cdot LmaxB_{top}$.

6. Let $MaxLen = \max_{B(v)\in Nbr(B_c)} Lmax(B(v))$. It must hold that $MaxLen \leq LmaxB_{top}$ according to the forth inequality above.

By combining inequality 1 and 3, we have $Lmin(B(v)) \geq \frac{1}{27}Lmax(B_{top})$ for any box $B(v) \in Nbr(B_c)$. Now we place a d-cube inside the box $B(v)$, where the side length of the d-cube is $\frac{1}{27}Lmax(B_{top})$. It is feasible since $Lmin(B(v)) \geq \frac{1}{27}Lmax(B_{top})$. Do so for each box in $Nbr(B_c)$, and it can be regarded as establishing a mapping from the boxes in $Nbr(B_c)$ to a set of d-cubes. The mapping has the following properties. (1) The mapping is one-to-one, (2) The d-cubes are disjoint, and (3) the side length of the d-cubes are $\frac{1}{27}Lmax(B_{top})$.

Now consider the set of boxes $Nbr(B_c)$ and we try to cover them by a large d-cube. Let $cen(B_c)$ be the center of the box B_c. We place a d-cube $LargeCube$ whose center is $cen(B_c)$ and with enough side length to cover all the boxes in $Nbr(B_c)$. With the aid of the definition of $MaxGap$ and $MaxLen$, the side length of the $LargeCube$ is enough to be set to $2 \cdot MaxGap + Lmax(B_c) + 2 \cdot MaxLen\ le (2d+3) LmaxB_{top}$. See Figure 2.

Now, fill the $LargeCube$ with small cubes of side length $1/27Lmax(B_{v_{top}})$. The number of small cubes is at most

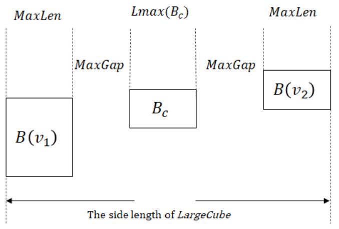


Fig. 2: Demonstration of the side length of $LargeCube$.

$O((\frac{2d+3}{1/27})^d) = O(d^d)$. Considering the one-to-one mapping between the small cubes to the boxes in $Nbr(B_c)$, the number of boxes in $Nbr(B_c)$ is then bounded by $O(d^d)$. By now the size of $Nbr(B_c)$ is proved to be $O(d^d)$.

□

We are now ready to prove the complexity of Algorithm 5.

Theorem 8. *Algorithm 5 can find the nearest neighbor of* q *in a pre-cluster* C *in* $O(d^{d+1} \log(|C|d))$ *time.*

Proof. The time complexity of Algorithm 5 can be divided into two parts. The first is the time of executing the $while$ loop, which is the number of boxes containing q (Lemma 12) multiplying the size of the Nbr set (Lemma 13). Then it can be proved to be $O(d^{d+1} \log(|C|d))$. The second part is the time of final brute-force search, which is the size of the Nbr set (Lemma 13) multiplying the maximum number of points contained in the leaf node (Lemma 8). Then it can be proved to be $O(minpts \cdot d^d)$. By adding the two terms up, and realizing $minpts$ is a constant, the final expression of the complexity is proved to be $O(d^{d+1} \log(|C|d))$. □

By combing the algorithms described in Section IV-C1, IV-C2 and IV-C3, an algorithm that can efficiently find the nearest neighbor in a pre-cluster is obtained, which proves Theorem 5. To support multiple queries on a single fair split tree, it need only to delete the added query point after the query is done, which takes $O(1)$ time by Theorem 7.

4) The overall final-clustering process and complexity: Based on the above nearest neighbor algorithm, the algorithm for final-clustering is established as follows. For each pair of pre-clusters $\mathcal{C}_i$ and $\mathcal{C}_j$, construct the fair split tree T on $\mathcal{C}_i$, and traverse all the points in $\mathcal{C}_j$ as follows. First add p into T, then conduct the nearest neighbor query. If the nearest neighbor distance is no more than ϵ, join $\mathcal{C}_i$ and $\mathcal{C}_j$ together, and continue to examine the next pair of pre-clusters. Otherwise, delete p from T and examine the next point in $\mathcal{C}_j$. The complexity of the final-clustering algorithm is as follows.

Theorem 9. *Let* $|P| = n$. *Let* $\mathbb{C} = \{\mathcal{C}_1, \mathcal{C}_2, \cdots, \mathcal{C}_l\}$ *be the set of pre-clusters obtained by performing core-points identification and pre-clustering process on* P. *Let* $l = n^\delta$. *Then, the final-clustering algorithm runs in* $O(d^{d+1} n^{1+\delta} \log(nd))$ *time.*

Proof. Let $n_i = |\mathcal{C}_i|$ for $1 \leq i \leq l$. The time complexity of final-clustering is

$$\sum_{1\leq i<j\leq l} \big(O(n_i \log n_i) + O(d^{d+1} n_j \log(n_i d))\big)$$
$$=O\left(d^{d+1} \sum_{1\leq i<j\leq l} ((n_i + n_j)\log(n_i d))\right) = O(d^{d+1} nl \log(nd))$$
$$=O(d^{d+1} n^{1+\delta} \log(nd))$$

□

D. Bypassing the $\Omega(n^{4/3})$ worst-case lower bound by instance-specified upper bound

Summarizing the results above, we have a new algorithm for DBSCAN as promised in Algorithm 3. The complexity of the whole process is as follows, which can be proved by combining Theorem 2, 4, and 9.

Theorem 10. *The time complexity of Algorithm 3 is $O(d^d n \log n + d^{d+1} n^{1+\delta} \log nd)$, where n^δ is the number of pre-clusters obtained after the pre-clustering process. The complexity is $O(n^{1+\delta} \log n)$ if d is regarded as a constant.*

Corollary 2. *Algorithm 3 runs in $o(n^{4/3})$ time if $\delta < 1/3$.*

This corollary is intuitive but counter-intuitive simultaneously. It is intuitive since it is easily deduced from Theorem 10, and counter-intuitive since it seemingly violates Gan and Tao's $\Omega(n^{4/3})$ lower bound for DBSCAN problem [9]. In fact, this corollary is correct and it does not violate Gan and Tao's lower bound. The reason is that the $\Omega(n^{4/3})$ lower bound considers only the **worst** case, but Corollary 2 describes a per-instance bound. Our bound meets Gan and Tao's lower bound in bad cases when $\delta \geq \frac{1}{3}$, but it also indicates there do exist *easier* instances such that faster running time can be realized.

V. The new conceptual approach

The new conceptual approach for the DBSCAN problem is given in Algorithm 6 in terms of the pre-final clustering framework. It is called a *conceptual* algorithm since the ϵ-**minpts** query and ϵ-**connectivity** query, which are defined next, are abstract problems that can be implemented by any kinds of techniques.

Algorithm 6: The new conceptual approach

```
1 Core-points identification: use ε-minpts query;
2 Pre-clustering: use arbitrary technique;
3 Final-clustering: use ε-connectivity query;
4 Assign border/noise points: use arbitrary technique;
```

Definition 11 (ϵ-**minpts** query)**.** ***Input****: point set P, query point $p \in P$, and DBSCAN parameters ϵ and $minpts$.*
Output*: answer Yes if $|Ball(p, \epsilon) \cap P| \geq minpts$, and answer No otherwise. Additionally, return the points in $|Ball(p, \epsilon) \cap P|$ visited by the algorithm together with their distances to p.*

Definition 12 (ϵ-**connectivity** query)**.** ***Input****: two point sets P_1 and P_2, and DBSCAN distance parameter ϵ.*
Output*: answer Yes if $\exists p_1 \in P_1, p_2 \in P_2$ s.t. $D(p_1, p_2) \leq \epsilon$, and answer No otherwise.*

We will show the non-redundancy of the two new abstract problems, by comparing them with other problems formerly used for DBSCAN. The non-redundancy here means that, the ϵ-**minpts** query and ϵ-**connectivity** query never perform more distance computations than the problems they compare against.

Theorem 11. *The number of distance computations using ϵ-**minpts** query to identify core-points, is always no more than using range search or k-nearest neighbors.*

Proof. Let $\mathcal{A}$ be any algorithm that solves the range search problem. Let the algorithm $\mathcal{A}'$ to solve ϵ-**minpts** query problem be as follows. $\mathcal{A}'$ conducts the same operation as $\mathcal{A}$, except for stopping once $minpts$ number of points in $Ball(p, \epsilon)$ are found. In such way, $\mathcal{A}'$ always performs no more distance computations than $\mathcal{A}$.

Let $\mathcal{B}$ be any algorithm that solves the k-nearest neighbors problem. Let the algorithm $\mathcal{B}'$ to solve ϵ-**minpts** query problem be as follows. $\mathcal{B}'$ conducts the same operation as $\mathcal{B}$, except for recording every point in $Ball(p, \epsilon)$ encountered and stopping once $minpts$ number of points in $Ball(p, \epsilon)$ are found. Note that the extra work of recording points involves no distance computations. In such way, while algorithm $\mathcal{B}$ is still searching the nearest neighbors, algorithm $\mathcal{B}'$ will stop as early as possible. Thus, algorithm $\mathcal{B}'$ always performs no more distance computations than algorithm $\mathcal{B}$. □

Theorem 12. *The number of distance computations using ϵ-**connectivity** query to perform final-clustering process, is always no more than using bichromatic closest pair.*

Proof. Let $\mathcal{A}$ be any algorithm that solves bichromatic closest pair problem. Let the algorithm $\mathcal{A}'$ for ϵ-**connectivity** query be as follows. $\mathcal{A}'$ conducts the same operation as $\mathcal{A}$, except for stopping once a directly density-connected pair is found. In such way, $\mathcal{A}'$ always performs no more distance computations than $\mathcal{A}$. □

The idea of early stopping in ϵ-**connectivity** query was also reflected by the the RangeCount algorithm proposed in [13]. Howver, the authors of [13] regard their RangeCount algorithm as an optimization heuristic for the bichromatic closest pair problem, while in this paper it is regarded as a new abstract problem. In such a sense, the ϵ-**connectivity** query can be studied and optimized independent with the bichromatic closest pair problem.

There remains another problem to consider. Since the original DBSCAN algorithm merely uses range search as sub-procedure, we must compare the method of combining ϵ-**minpts** query and ϵ-**connectivity** query with the original method which merely uses range search.

Theorem 13. *If the DBSCAN problem is solved by combining ϵ-**minpts** query and ϵ-**connectivity** query, the number of dis-*

tance computations is always no more than solving it using range search.

Proof. Let OLD be the original DBSCAN algorithm that is solved by range search. Let NEW be the new algorithm using the idea of combining ϵ-**minpts** query and ϵ-**connectivity** query. The algorithm NEW works as follows in three stages.

In the first stage, NEW simulates OLD to solve the ϵ-**minpts** query problem. The goal of the first stage is that for all points p in the input set P, either find at least $minpts$ number of points in $Ball(p, \epsilon)$ (for core-points), or the points in $Ball(p, \epsilon)$ are all discovered (for non-core points). We say a point is $decided$ if so. In this stage the algorithm NEW works as follows. For each range search procedure started by OLD, NEW simulates the range search procedure the same way as provided in the proof of Theorem 11. Once the simulation of the range search is early stopped, the stop point and the state of the simulation is recorded so that the simulation can be restarted at the stop point. The algorithm NEW will proceed the simulation until the goal of the first stage is fulfilled. Note that there may be multiple interrupted range search procedures, and the stop point for each of them should be recorded.

Once the goal of the first stage is fulfilled, the algorithm NEW will enter the second stage. In this stage NEW will conduct the pre-clustering process based on the neighbor points found in the first stage. A union-find set is used to perform the pre-clustering process. A set of pre-clusters $\{\mathcal{C}_1, \cdots, \mathcal{C}_l\}$ will be obtained.

Now the algorithm NEW enters the third stage to solve the ϵ-**connectivity** query. It will restart the interrupted range search procedures from the recorded stop point and continue simulating the OLD algorithm. For each distance computation executed by OLD, supposing the distance computation is upon p and q, the algorithm NEW will check if p and q are in the same pre-cluster using the union-find set. If they are in the same pre-cluster, the distance computation will not be executed by NEW. Otherwise, the algorithm NEW will compute the distance between p and q which behaves the same with OLD. If $D(p, q) \leq \epsilon$, the algorithm NEW will merge the two pre-clusters where p and q reside in, using the union-find set. The simulation will restore and finish all the interrupted range search procedures, and proceed until the algorithm OLD is finished.

It is obvious that the above algorithm NEW indeed returns the same clustering result with the original DBSCAN algorithm OLD. It is also obvious that the algorithm NEW never performs more distance computations than OLD. Besides, all the extra works of NEW do not involve distance computations, such as recording and restoring stop point, and union-find set operations. □

VI. The new empirical approach

The empirical algorithm proposed in this section is given in Algorithm 7. This implementation uses KDTree [30] as the index structure to support the ϵ-**minpts** query. Realizing that each node of the KDTree represents a d-dimensional box, an extra heuristic is used for identifying core-points besides the ϵ-**minpts** query, which is reflected by Line 3-7 in Algorithm 7. Let $Dmax(N)$ denote the longest diagonal of a KDTree node N. It is easy to see that, if $Dmax(N) \leq \epsilon$ and N contains more than $minpts$ points, then all points in N are core-points and they belong to the same cluster.

Algorithm 7 uses stacks to simulate the tree traversal procedure on KDTree to solve the ϵ-**minpts** query, which is described in detail in the $EpsMinptsSimulation$ procedure. For each KDTree node N, let $Dmin(p, N)$ and $Dmax(p, N)$ denote the minimum and maximum distance between p and a vertex point of the d-dimensional box represented by the node N, respectively. These two values will guide the stack-simulated traversal on the KDTree. The stack for p stores the un-visited nodes in the KDTree, and the node N with lower $Dmin(p, N)$ value would be visited with higher priority. When a leaf node or a node N satisfying $Dmax(p, N) \leq \epsilon$ is visited, the points $q \in N$ which satisfy $D(p, q) \leq \epsilon$ would be added into $Neighbors(p)$. Finally, when the size of $Neighbors(p)$ becomes larger than $minpts$ after a node is visited, the boolean value $isCore[p]$ will be set to be true, and the tree traversal procedure will be interrupted. Note that the stack for p will not be destroyed and the tree traversal procedure could be restarted based on the stack whenever needed. Also, if p is a non-core point, the tree traversal procedure will proceed until the whole KDTree is visited and all points in $Neighbors(p)$ will be found, where $|Neighbors(p)| < minpts$ since it is a non-core point. The pre-clustering process is the same with Algorithm 4 which uses union-find set. The final-clustering process, given in $FinalClusteringSimulation$ procedure, is to restore all the interrupted tree traversal procedures using the recorded stacks.

A. Merge-on-find strategy based on lazy-construction fast-merge cover tree

Since some core-points are identified by examining the diagonal of KDTree nodes without performing tree traversal, the pre-clusters containing such core-points may not be maximized. By referring Line 3-7 of Algorithm 7 as the KDTreeGrid procedure, we have the following lemma.

Lemma 14. *A pre-cluster is not-maximized only if it contains a core-point which is identified in the KDTreeGrid procedure.*

By Lemma 14 we can know that if a pre-cluster contains only the core-points identified by tree traversal, then it must be maximized. For the other pre-clusters that containing core-points identified by the KDTreeGrid procedure, the following merge-on-find strategy is used to determine the possible density-connectivity among them.

1) The merge-on-find strategy: The merge-on-find strategy is given in Algorithm 9, which takes the set of pre-clusters that may not be maximized as input. As we have mentioned, these clusters are those containing core-points identified by the KDTreeGrid procedure. The algorithm first constructs a mapping $MapRepCluster$, where the value is the point set of the cluster, and the corresponding key is the representative

Algorithm 7: The empirical algorithm

```
Input: point set $P$, and DBSCAN parameters $\epsilon$ and $minpts$.
Output: the DBSCAN clustering result.
Build KDTree $T$ based on $P$;
Initialize the union-find set $UFS$ on $P$;
// The KDTreeGrid procedure
foreach $node\ N\ of\ T$ do
    if $Dmax(N) \leq \epsilon$ and $|N \cap P| \geq minpts$ then
        Mark each point $p \in N$ as core-point;
        foreach $p, q \in N$ do
            $UFS.join(p, q)$;
// Identifying core-points
foreach $p \in P$ do
    if $isCore[p] = false$ then
        Initialize the stack $stack_p$ for $p$;
        $EpsMinptsSimulation(p, stack_p, T)$
// Pre-clustering
foreach core-point $p$ do
    foreach core-point $q \in Neighbors(p)$ do
        $UFS.union(p, q)$;
// Final-clustering
foreach core-point $p$ do
    $FinalClusteringSimulation(p, stack_p, T)$;
$FinalMergeOnFind()$;
// assigning border points
foreach non-core point $p$ do
    if $\exists q \in Neighbors(p)$ is core-point then
        $UFS.union(p, q)$;
```

Algorithm 8: Stack simulation procedures

```
Procedure $EpsMinptsSimulation(p, S, T)$:
    $S.push(T)$;
    while $S$ is not empty do
        if $isCore[p]$ then break;
        $N \leftarrow stack.pop()$;
        if $N$ is leaf then
            foreach point $q$ in $N$ do
                if $D(p, q) \leq \epsilon$ then
                    Add $q$ into $Neighbors(p)$;
            if $|Neighbors(p)| \geq minpts$ then
                $isCore[p] \leftarrow true$;
        foreach child node $N_c$ of $N$ do
            if $Dmax(p, N_c) \leq \epsilon$ then
                foreach point $q$ in $N$ do
                    Add $q$ into $Neighbors(p)$;
                if $|Neighbors(p)| \geq minpts$ then
                    $isCore[p] \leftarrow true$;
                continue;
            if $Dmin(p, N_c) \leq \epsilon$ then $S.push(N_c)$;
            if both child should be pushed, push the
             one with lager $Dmin(p, N_c)$ first;
Procedure $FinalClusteringSimulation(p, S, T)$:
    while $S$ is not empty do
        $N \leftarrow S.pop()$;
        if $N$ is leaf then
            foreach core-point $q$ in $N$ do
                if $UFS.find(p) \neq UFS.find(q)$ then
                    if $D(p, q) \leq \epsilon$ then
                        $UFS.union(p, q)$;
        foreach child node $N_c$ of $N$ do
            if $Dmax(p, N_c) \leq \epsilon$ then
                foreach core-point $q$ in $N$ do
                    $UFS.union(p, q)$;
                continue;
            if $Dmin(p, N_c) \leq \epsilon$ then $S.push(N_c)$;
```

of this cluster in the union-find set. The algorithm then iterates the pre-clusters to discover the density-connectivity and merge them if they are density-connected.

An appropriate index structure is needed to support the merge-on-find strategy, and it should satisfy the following three requirements. First, the index structure must support **ϵ-connectivity** query efficiently, since in Line 8 of Algorithm 9 the **ϵ-connectivity** query must be executed with the aid of the index structure. Second, it must support the merge operation efficiently. In Line 10 of the merge-on-find algorithm, when the two clusters are merged the index structures of them must be merged too. Some index structures such as KDTree do not support the merge operation, and thus it can not be used to support the merge-on-find algorithm. Third, the time of constructing the index must be small. Comparing with the original algorithm for DBSCAN which only needs one index structure for the input point set to support the range search, the merge-on-find algorithm must build multiple indexes, each for one pre-cluster to support the **ϵ-connectivity** query. Thus, the index structures must be constructed very efficiently.

2) The lazy-construction fast-merge cover tree: Fortunately, there exists one index structure that satisfies all the three requirements described above, which is the cover tree with lazy-construction and fast-merge strategy. This index structure is a new variant of the cover tree proposed in [10]. Let us first introduce the standard cover tree.

A cover tree is a leveled tree where each node represents only one data point. An integer scale i is associated for each level of the tree which decreases as the tree is descended. Let S_i be the set of nodes in level i, and the cover tree ensures the following invariants for all i.
(1) (Nesting) $S_i \subset S_{i-1}$.
(2) (Covering) For every $p \in S_{i-1}$, there exists a $q \in S_i$ s.t. $D(p, q) \leq 2^i$, and exactly one such q is the parent of p.
(3) (Separation) $D(p, q) > 2^i$ for all $p, q \in S_i$.

Due to the idea proposed in [10], the cover tree can be lazily constructed, namely, the tree can be initialized as only the root and the rest of the nodes can be constructed only when needed. However, the authors of [10] describe a single-point lazy insertion strategy, where each lazy insertion operation adds one node into the tree. Our implementation here is different, which

Algorithm 9: Merge-on-find final-clustering

```
Input: the pre-clusters that may not be maximized,
       and the union-find set $UFS$.
1  Procedure $FinalMergeOnFind$:
2      Construct $MapRepCluster$ based on the input;
3      $(p_0, \mathcal{C}_0) \leftarrow$ the first key-value pair in
         $MapRepCluster$;
4      remove $(p_0, \mathcal{C}_0)$ from $MapRepCluster$;
5      while $|MapRepCluster| > 0$ do
6          $\mathcal{C}_{next} \leftarrow \emptyset$;
7          foreach $(p_i, \mathcal{C}_i) \in MapRepCluster$ do
8              if $\epsilon$-connected($\mathcal{C}_i, \mathcal{C}_0$) then
9                  $UFS.union(p_0, p_i)$;
10                 $\mathcal{C}_{next} \leftarrow \mathcal{C}_{next} \cup \mathcal{C}_i$, and merge indexes;
11                 remove $(p_i, \mathcal{C}_i)$ from $MapRepCluster$;
12         if $\mathcal{C}_{next} \neq \emptyset$ then
13             $\mathcal{C}_0 \leftarrow \mathcal{C}_{next}$;
14         else
15             $(p_0, \mathcal{C}_0) \leftarrow$ the first key-value pair in
                 $MapRepCluster$;
16             remove $(p_0, \mathcal{C}_0)$ from $MapRepCluster$;
```

is a node-expansion lazy construction strategy. Concretely, when a node in the cover tree is asked to be expanded, all its children nodes will be constructed and added to the cover tree. While there is no difference between the single-point-insertion strategy and node-expansion strategy in tree building time, the node-expansion strategy is more suitable for supporting the implementation of the ϵ-**connectivity** query. Furthermore, the cover tree should support the merge operation. There exists an implementation of the merge operation of cover tree [31], but we propose a simplified and much efficient version here.

Algorithm 10 shows the pseudo codes for $Expand$ and $Merge$ operation on the cover tree. To support the two operations the following information is stored in each node N of the cover tree. $N.scale$ is the scale of the node described in the definition of standard cover tree. $N.point$ is the corresponding data point of this node. $N.descendants$ is the set of data points in the sub-tree rooted at N, including $N.point$ itself. For the root node, $N_{root}.descendants$ is the whole input point set. Finally, $N.isExpanded$ is a boolean value indicating whether the node N has been expanded.

*3) ϵ-**connectivity** query:* The algorithm for ϵ-**connectivity** query using lazy-construction cover tree as the index structure is given in Algorithm 11. The algorithm will traverse the two input cover trees T_1 and T_2 to find at least one directly density-connected pair. A boolean value $found$ is maintained globally to the algorithm, and the algorithm will terminate once the $found$ value is set to be true. The algorithm will invoke the $Expand$ operation of the lazy-construction cover tree if an un-expanded node is visited. In such way, only when a node of the cover tree is visited in the ϵ-**connectivity** query will it be expanded. If the ϵ-**connectivity** query can be early stopped, some of the nodes in the cover tree will remain un-expanded,

Algorithm 10: Expand and Merge operation

```
1  Function Expand($CoverTree\ N$):
2      $N.isExpanded \leftarrow true$;
3      while $N.descendants$ is not empty do
4          Pick out the first point $p$ in $N.descendants$;
5          Create new node $N_c$ and let it be a child of $N$;
6          $N_c.point \leftarrow p$, $N_c.scale \leftarrow N.scale - 1$;
7          $N_c.isExpanded \leftarrow false$;
8          foreach point $p'$ in $N.descendants$ do
9              if $D(p, p') \leq 2^{N.scale-1}$ then
10                 Remove $p'$ from $N.descendants$ and
                     add it to $N_c.descendants$;
11 Function Merge($CoverTree\ T_1$, $CoverTreeT_2$):
12     Let $T_1$ be the one with larger scale;
13     if $D(T_1.point, T_2.point) > 2^{T_1.scale}$ then
14         $T_1.scale \leftarrow \lceil \log_2 D(T_1.point, T_2.point) \rceil$;
15     Let $T_2$ be a child node of $T_1$;
```

thus saving a lot of time for constructing the cover tree.

Algorithm 11: ϵ-**connectivity** query

```
1  Function $\epsilon$-connected(CoverTree $T_1$,CoverTree $T_2$):
2      if $T_1$ and $T_2$ are both leaves then return;
3      Invoke $Expand(T_1)$ or $Expand(T_2)$ if it is not
         expanded;
4      foreach child node $T_{1c}$ of $T_1$, $T_{2c}$ of $T_2$ do
5          $dist \leftarrow D(T_{1c}.point, T_{2c}.point)$;
6          if $dist \leq \epsilon$ then
7              $found \leftarrow true$ and return;
8          else
9              $score \leftarrow dist - 2^{T_{1c}.scale+1} - 2^{T_{2c}.scale+1}$;
10     Sort child node pairs in ascending order of the
         score;
11     foreach child node pair $(T_1, T_2)$ do
12         Invoke $\epsilon$-connected$(T_1, T_2)$;
13         if $found$ is true then return;
```

By now we are finished describing the merge-on-find final-clustering process. Back to Algorithm 9 for the merge-on-find process, the ϵ-**connectivity** query at Line 8 is implemented by Algorithm 11, and the merge operation at Line 10 is implemented by the $Merge$ operation given in Algorithm 10.

VII. Experiments

A. Preparations

1) Our method: Our proposed method is named as *EMEC* from the $\underline{\epsilon}$-**$\underline{\text{m}}$inpts** and $\underline{\epsilon}$-**$\underline{\text{c}}$onnectivity** query.

2) Methods compared: The following three methods are considered. *GanTao* is a baseline implementation of DBSCAN provided by Gan and Tao [22], and the binary executable is available at [32]. *sigmod20* [13] is described as a parallel implementation of DBSCAN, but can be run in single-thread mode with good efficiency. *GriT* [33] is the most recent implementation of DBSCAN.

TABLE II: datasets information summarization

dataset	**gangen_vary_no**	**gangen_vary_yes**	**bitcoin**	**household**	**corel**
num	1000000	1000000	2916697	2075259	662317
dim	3, 7	3, 7	5	7	14
skewness	no	yes	yes	yes	yes

3) Datasets: Since the *GanTao* method only receives integer input in range $[1, 10^5]$, all the following datasets are scaled to this range and rounded to integer values.

Two kinds of synthetic datasets are used, which are generated by the generator provided by Gan and Tao [22] which generates data in a random walk manner, and the two kinds of generators are with/without varied density, respectively. The two kinds of datasets are called **gangen_vary_yes, gangen_vary_no**, respectively. The number of points is fixed to 1 million (10^6), and two datasets with dimension 3 and 7 are generated for each of these two generators, which represent the lower and higher dimensional cases, respectively. Three kinds of real-world datasets are used, which are **household**[1], **bitcoin**[2] and **corel**[3]. The detailed information of the datasets are listed in Table II.

We will use num as an alias for number, dim as an alias for dimension, and $dbscan\_eps$ as an alias for parameter ϵ.

4) Environment setup: All the experiments are run on a machine with Intel Core i7-10700 CPU and 128GB RAM running Ubuntu 20.04 operating system.

B. Results and analysis

The running time of *EMEC* and the compared methods are tested for a wide range of DBSCAN parameters settings, which are shown in Figure 2-6. The number of generated clusters varied with ϵ is also shown in the top part of each figure. The ϵ parameter are set in the range started from small enough such that no points are identified as core-points, and ended to large enough such that all the points are grouped into a very smaller number of clusters. Note that we do not care about whether the DBSCAN parameters are appropriate such that the clustering result is meaningful. Our goal is to design an efficient algorithm that runs fast no matter how the DBSCAN parameters are set. The experimental results show that *EMEC* runs indeed faster than the three compared methods in almost all the tested parameter settings.

However, there are several cases that *EMEC* is less efficient than the compared methods. See Figure 2(c), 2(d), 3(c), 3(d), and 4(d). The reason is that the $minpts$ parameter is set to be larger in these cases, and the running time of the ϵ**-minpts** query is more affected by the $minpts$ parameter. Nevertheless, for datasets with higher dimension, the deficiency brought by large $minpts$ parameter is compromised, which is because the ϵ**-connectivity** query implemented by the lazy-construction fast-merge cover tree is more efficient in higher dimensions. The supremacy of *EMEC* is most reflected on the **corel** dataset with dimension 14, which is shown in Figure 6.

[1]archive.ics.uci.edu/ml/datasets/Individual+household+electric+power+consumption

[2]archive.ics.uci.edu/ml/datasets/BitcoinHeistRansomwareAddressDataset

[3]code.google.com/p/nndes/

VIII. Conclusion

We have brought some new insights and techniques to the well-studied DBSCAN problem. First, the pre-final clustering framework, while has been implicitly used in existing works, is the first time explicitly highlighted in this paper. The advantages of the pre-final clustering framework, including the theoretical, conceptual and empirical aspects, are fully exploited in this paper. Second, a new theoretical algorithm for DBSCAN is proposed which bypasses the $\Omega(n^{4/3})$ worst-case lower bound by instance specified upper bound $O(n^{1+\delta} \log n)$. Third, the ϵ**-minpts** and ϵ**-connectivity** queries are proposed, which will not incur unnecessary distance computations compared with other abstract problems formerly used for DBSCAN. Finally, a new empirical algorithm for DBSCAN named *EMEC* is proposed, which have better efficiency than state-of-the-art in almost all the tested parameter settings.

References

[1] M. Ester, H. Kriegel, J. Sander, and X. Xu, "A density-based algorithm for discovering clusters in large spatial databases with noise," in *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD-96), Portland, Oregon, USA*, E. Simoudis, J. Han, and U. M. Fayyad, Eds. AAAI Press, 1996, pp. 226–231. [Online]. Available: http://www.aaai.org/Library/KDD/1996/kdd96-037.php

[2] E. Schubert, J. Sander, M. Ester, H. P. Kriegel, and X. Xu, "Dbscan revisited, revisited: why and how you should (still) use dbscan," *ACM Transactions on Database Systems (TODS)*, vol. 42, no. 3, pp. 1–21, 2017.

[3] N. Gholizadeh, H. Saadatfar, and N. Hanafi, "K-DBSCAN: an improved DBSCAN algorithm for big data," *J. Supercomput.*, vol. 77, no. 6, pp. 6214–6235, 2021. [Online]. Available: https://doi.org/10.1007/s11227-020-03524-3

[4] A. Sarma, P. Goyal, S. Kumari, A. Wani, J. S. Challa, S. Islam, and N. Goyal, "μdbscan: An exact scalable DBSCAN algorithm for big data exploiting spatial locality," in *2019 IEEE International Conference on Cluster Computing, CLUSTER 2019, Albuquerque, NM, USA, September 23-26, 2019*. IEEE, 2019, pp. 1–11. [Online]. Available: https://doi.org/10.1109/CLUSTER.2019.8891020

[5] K. M. Kumar and A. R. M. Reddy, "A fast DBSCAN clustering algorithm by accelerating neighbor searching using groups method," *Pattern Recognit.*, vol. 58, pp. 39–48, 2016. [Online]. Available: https://doi.org/10.1016/j.patcog.2016.03.008

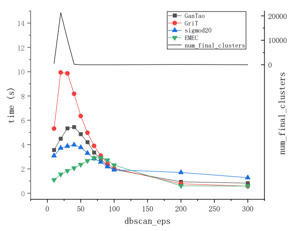
[6] Y. Chen, L. Zhou, N. Bouguila, C. Wang, Y. Chen, and J. Du, "BLOCK-DBSCAN: fast clustering for large scale data," *Pattern Recognit.*, vol. 109, p. 107624, 2021. [Online]. Available: https://doi.org/10.1016/j.patcog.2020.107624

[7] X. Huang, T. Ma, C. Liu, and S. Liu, "Grit-dbscan: A spatial clustering algorithm for very large databases," *CoRR*, vol. abs/2210.07580, 2022. [Online]. Available: https://doi.org/10.48550/arXiv.2210.07580

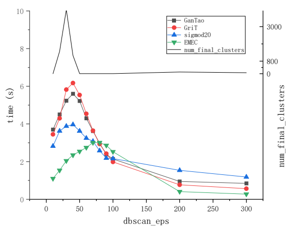
[8] Y. Chen, L. Zhou, S. Pei, Z. Yu, Y. Chen, X. Liu, J. Du, and N. Xiong, "KNN-BLOCK DBSCAN: fast clustering for large-scale data," *IEEE Trans. Syst. Man Cybern. Syst.*, vol. 51, no. 6, pp. 3939–3953, 2021. [Online]. Available: https://doi.org/10.1109/TSMC.2019.2956527

[9] J. Gan and Y. Tao, "DBSCAN revisited: Mis-claim, un-fixability, and approximation," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, T. K. Sellis, S. B. Davidson, and Z. G. Ives, Eds. ACM, 2015, pp. 519–530. [Online]. Available: https://doi.org/10.1145/2723372.2737792

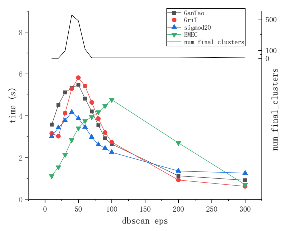
[10] A. Beygelzimer, S. M. Kakade, and J. Langford, "Cover trees for nearest neighbor," in *Machine Learning, Proceedings of the Twenty-Third International Conference (ICML 2006), Pittsburgh, Pennsylvania, USA, June 25-29, 2006*, ser. ACM International Conference Proceeding Series, W. W. Cohen and A. W. Moore, Eds., vol. 148. ACM, 2006, pp. 97–104. [Online]. Available: https://doi.org/10.1145/1143844.1143857



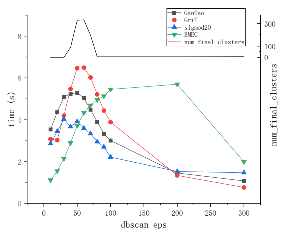
(a) dim=3, minpts=5



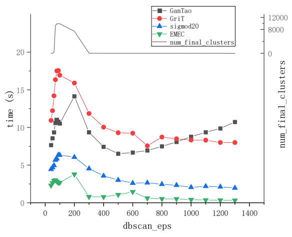
(b) dim=3, minpts=10



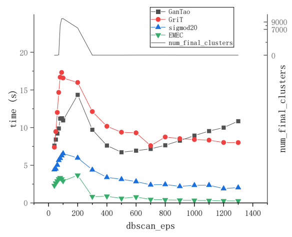
(c) dim=3, minpts=30



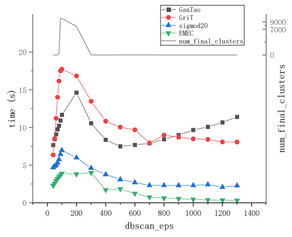
(d) dim=3, minpts=50



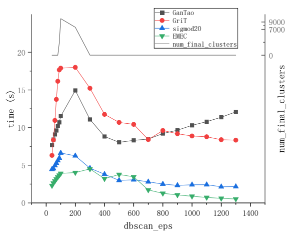
(e) dim=7, minpts=5



(f) dim=7, minpts=10

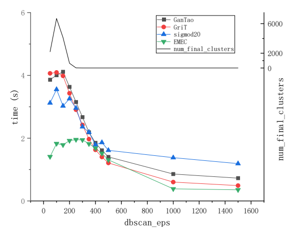


(g) dim=7, minpts=30

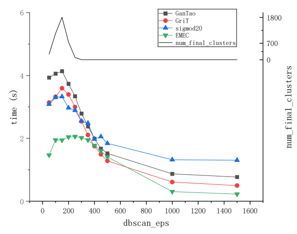


(h) dim=7, minpts=50

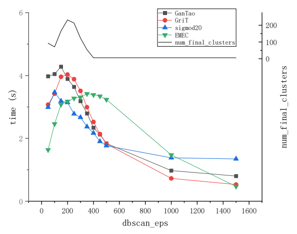
Fig. 3: Running time on gangen_vary_no.



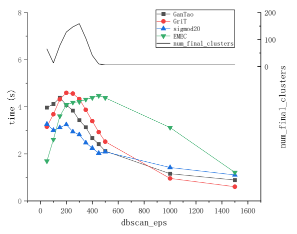
(a) dim=3, minpts=5



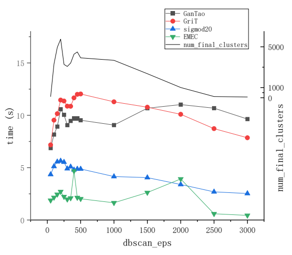
(b) dim=3, minpts=10



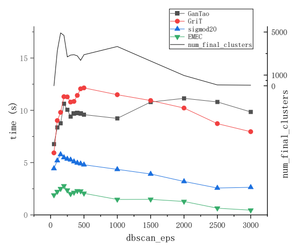
(c) dim=3, minpts=30



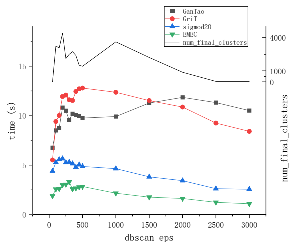
(d) dim=3, minpts=50



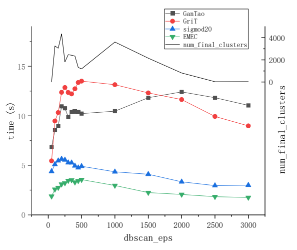
(e) dim=7, minpts=5



(f) dim=7, minpts=10

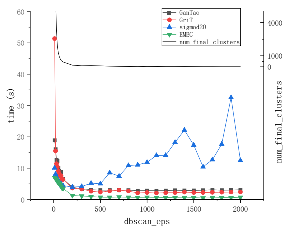


(g) dim=7, minpts=30

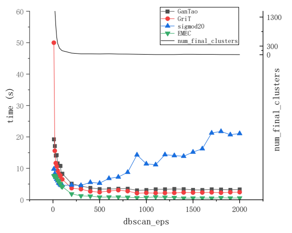


(h) dim=7, minpts=50

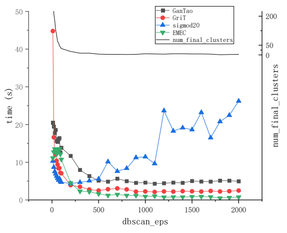
Fig. 4: Running time on gangen_vary_yes.



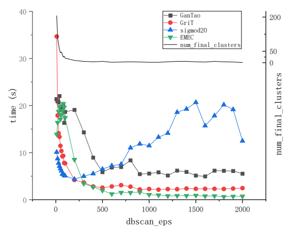
(a) minpts=5



(b) minpts=10

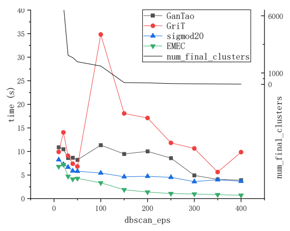


(c) minpts=30

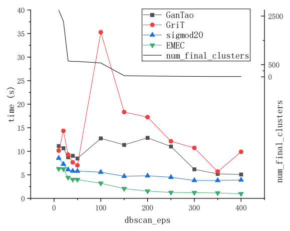


(d) minpts=50

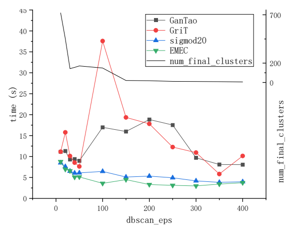
Fig. 5: Running time on **bitcoin**.



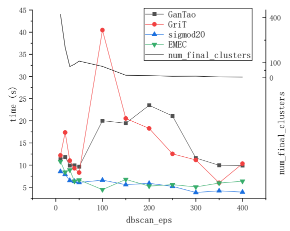
(a) minpts=5



(b) minpts=10

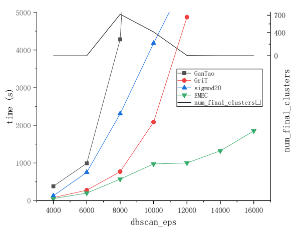


(c) minpts=30

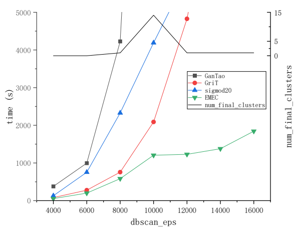


(d) minpts=50

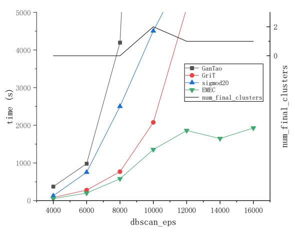
Fig. 6: Running time on **household**.



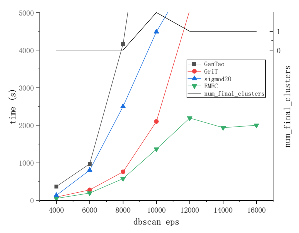
(a) minpts=5



(b) minpts=10



(c) minpts=30



(d) minpts=50

Fig. 7: Running time on **corel**.

[11] M. M. A. Patwary, D. Palsetia, A. Agrawal, W. Liao, F. Manne, and A. N. Choudhary, "A new scalable parallel DBSCAN algorithm using the disjoint-set data structure," in *SC Conference on High Performance Computing Networking, Storage and Analysis, SC '12, Salt Lake City, UT, USA - November 11 - 15, 2012*, J. K. Hollingsworth, Ed. IEEE/ACM, 2012, p. 62. [Online]. Available: https://doi.org/10.1109/SC.2012.9

[12] S. Li, "An improved DBSCAN algorithm based on the neighbor similarity and fast nearest neighbor query," *IEEE Access*, vol. 8, pp. 47 468–47 476, 2020. [Online]. Available: https://doi.org/10.1109/ACCESS.2020.2972034

[13] Y. Wang, Y. Gu, and J. Shun, "Theoretically-efficient and practical parallel DBSCAN," in *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, D. Maier, R. Pottinger, A. Doan, W. Tan, A. Alawini, and H. Q. Ngo, Eds. ACM, 2020, pp. 2555–2571. [Online]. Available: https://doi.org/10.1145/3318464.3380582

[14] S. Kumari, P. Goyal, A. Sood, D. Kumar, S. Balasubramaniam, and N. Goyal, "Exact, fast and scalable parallel DBSCAN for commodity platforms," in *Proceedings of the 18th International Conference on Distributed Computing and Networking, Hyderabad, India, January 5-7, 2017*. ACM, 2017, p. 14. [Online]. Available: http://dl.acm.org/citation.cfm?id=3007773

[15] A. Gunawan and M. de Berg, "A faster algorithm for dbscan," *Master's thesis*, 2013.

[16] L. Cao, D. Yang, Q. Wang, Y. Yu, J. Wang, and E. A. Rundensteiner, "Scalable distance-based outlier detection over high-volume data streams," in *IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014*, I. F. Cruz, E. Ferrari, Y. Tao, E. Bertino, and G. Trajcevski, Eds. IEEE Computer Society, 2014, pp. 76–87. [Online]. Available: https://doi.org/10.1109/ICDE.2014.6816641

[17] Y. Chen, S. Tang, N. Bouguila, C. Wang, J. Du, and H. Li, "A fast clustering algorithm based on pruning unnecessary distance computations in DBSCAN for high-dimensional data," *Pattern Recognit.*, vol. 83, pp. 375–387, 2018. [Online]. Available: https://doi.org/10.1016/j.patcog.2018.05.030

[18] H. Ma and J. Li, "Solving dbscan using an algorithm framework theoretically, conceptually and empirically fast." [Online]. Available: https://github.com/Haposola/EMEC-DBSCAN

[19] P. M. Vaidya, "An optimal algorithm for the all-nearest-neighbors problem," in *27th Annual Symposium on Foundations of Computer Science (sfcs 1986)*, Intergovernmental Panel on Climate Change, Ed. Cambridge: IEEE, oct 1986, pp. 117–122.

[20] ——, "AnO(n logn) algorithm for the all-nearest-neighbors Problem," *Discrete & Computational Geometry*, vol. 4, no. 2, pp. 101–115, mar 1989.

[21] H. Ma and J. Li, "A true o(nlog n) algorithm for the all-k-nearest-neighbors problem," in *Combinatorial Optimization and Applications - 13th International Conference, COCOA 2019, Xiamen, China, December 13-15, 2019, Proceedings*, ser. Lecture Notes in Computer Science, Y. Li, M. Cardei, and Y. Huang, Eds., vol. 11949. Springer, 2019, pp. 362–374. [Online]. Available: https://doi.org/10.1007/978-3-030-36412-0_29

[22] J. Gan and Y. Tao, "On the hardness and approximation of euclidean dbscan," *ACM Transactions on Database Systems (TODS)*, vol. 42, no. 3, pp. 1–45, 2017.

[23] R. E. Tarjan and J. Van Leeuwen, "Worst-case analysis of set union algorithms," *Journal of the ACM (JACM)*, vol. 31, no. 2, pp. 245–281, 1984.

[24] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Y. Wu, "An optimal algorithm for approximate nearest neighbor searching fixed dimensions," *J. ACM*, vol. 45, no. 6, pp. 891–923, 1998. [Online]. Available: https://doi.org/10.1145/293347.293348

[25] S. Arya and D. M. Mount, "Approximate nearest neighbor queries in fixed dimensions," in *Proceedings of the Fourth Annual ACM/SIGACT-SIAM Symposium on Discrete Algorithms, 25-27 January 1993, Austin, Texas, USA*, V. Ramachandran, Ed. ACM/SIAM, 1993, pp. 271–280. [Online]. Available: http://dl.acm.org/citation.cfm?id=313559.313768

[26] R. Krauthgamer and J. R. Lee, "Navigating nets: simple algorithms for proximity search," in *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, 2004, pp. 798–807.

[27] R. R. Curtin, "Improving dual-tree algorithms," Ph.D. dissertation, Georgia Institute of Technology, Atlanta, GA, USA, 2016. [Online]. Available: https://hdl.handle.net/1853/54354

[28] D. R. Karger and M. Ruhl, "Finding nearest neighbors in growth-restricted metrics," in *Proceedings on 34th Annual ACM Symposium on Theory of Computing, May 19-21, 2002, Montréal, Québec, Canada*, J. H. Reif, Ed. ACM, 2002, pp. 741–750. [Online]. Available: https://doi.org/10.1145/509907.510013

[29] S. N. Bespamyatnikh, "An optimal algorithm for closest pair maintenance," in *Proceedings of the eleventh annual symposium on Computational geometry*, 1995, pp. 152–161.

[30] J. Bentley, "Multidimensional binary search trees used for associative

searching," *Communications of the ACM*, vol. 18, no. 9, pp. 509–517, 1975.

[31] M. Izbicki and C. R. Shelton, "Faster cover trees," in *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*, ser. JMLR Workshop and Conference Proceedings, F. R. Bach and D. M. Blei, Eds., vol. 37. JMLR.org, 2015, pp. 1162–1170. [Online]. Available: http://proceedings.mlr.press/v37/izbicki15.html

[32] J. Gan and Y. Tao, "The codes for gantaov2." [Online]. Available: https://sites.google.com/view/approxdbscan

[33] X. Huang, T. Ma, C. Liu, and S. Liu, "Grit-dbscan: A spatial clustering algorithm for very large databases," *Pattern Recognition*, p. 109658, 2023.