GeDual-DBSCAN: A Generalized Dual-tree Algorithm for the DBSCAN problem

Hengzhao Ma, Jianzhong Li, Yong Zhang
Shenzhen Institute of Advanced Technology, Chinese Academy of Sciences
{hz.ma,lijzh,zhangyong}@siat.ac.cn

ABSTRACT

In this paper a novel approach to the classical DBSCAN problem is proposed which is based on two-fold innovative ideas. On the one hand, several fine-grained sub-problems of the DBSCAN problem are identified for the first time, which address certain sub-cases of the DBSCAN problem with less distance computations. On the other hand, the DBSCAN problem is considered from the perspective of a *dual-set* problem, and for the first time a delicate dual-tree algorithm surpassing the trivial dual-tree range search algorithm is proposed. The GeDual-DBSCAN algorithm proposed in this paper, named after Generalized Dual-tree algorithm, links the two new ideas using the new concept of *intermediate cases*. When the tree traversal encounters an intermediate case, the algorithm will be switched to a different sub-traversal to solve the sub-problem corresponding to this case, which is exactly one of the new sub-problems defined in this paper. This paper uses three variants to introduce the GeDual-DBSCAN algorithm, where the first one establishes the traversal order, the second one integrates the intermediate cases into the basic traversal, and the third one involves a new idea of *state transitions*. In addition to providing correctness and time complexity analysis of these algorithms, substantial experimental results are presented to evaluate their performance. It is shown that the final variant, named GeDual-DBSCAN-Transitions, outperforms the state-of-the-art DBSCAN algorithm under almost all the parameter settings on all datasets tested. In summary, the new GeDual-DBSCAN algorithm brings not only conceptual advancements for both the DBSCAN problem and dual-tree algorithms, and also efficiency improvement against existing DBSCAN algorithms.

PVLDB Reference Format:
Hengzhao Ma, Jianzhong Li, Yong Zhang. GeDual-DBSCAN: A Generalized Dual-tree Algorithm for the DBSCAN problem. PVLDB, 18(1): XXX-XXX, 2024.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and other artifacts have been made available at https://github.com/Haposola/GeDual-DBSCAN.

1 INTRODUCTION

DBSCAN [14] is a well-acknowledged density-based clustering approach with long research history [7, 8, 19, 22, 28, 30, 35, 41, 47] and wide application [12, 13, 43]. It was included in various textbooks as a classical clustering method [26, 44], and inspired many other density-based methods such as OPTICS [1] and HDBSCAN [33].

full version doi:XX.XX/XXX.XX

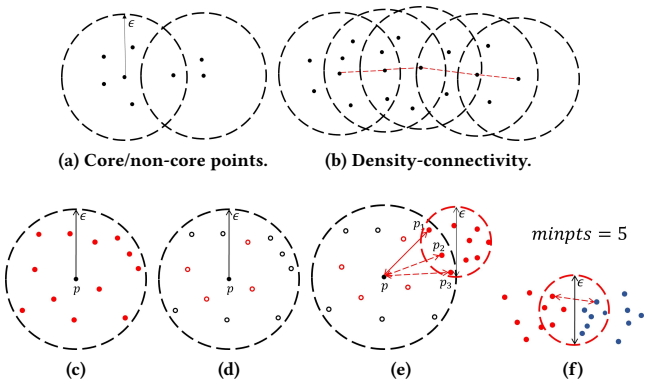


Figure 1: (a)(b) Demonstrates DBSCAN concepts ($minpts = 5$). (c) Points visited by range search. In contrast with: minimal points needed to (d) identify a core point, (e) merge a core point with a cluster, and (f) merge two clusters.

The DBSCAN problem involves two parameters, $minpts$ which is the threshold for number of neighboring points, and ϵ which is the neighborhood radius. Let $B(p, \epsilon)$ be the d-dimensional ball centered at p with radius ϵ, and let P be the point set to be clustered. A point $p \in P$ is called core point if $|B(p, \epsilon) \cap P| \geq minpts$, and non-core point otherwise. Note that p itself is included. 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 1a and 1b demonstrates the idea of core/non-core points and density-connectivity, respectively.

It is well known that the DBSCAN problem can be divided into three critical components [42], i.e., *identifying core/non-core points*, *clustering core points*, and *assigning non-core points*. The three parts of tasks can be trivially solved by executing a *range search* for each point $p \in P$, i.e., find $B(p, \epsilon) \cap P$ for $\forall p \in P$. This is also the initial method adopted by the seminal paper [14].

It has been long recognized that the original solution based on range search will incur a lot of unnecessary distance computations and lead to sub-optimal algorithms. Many efforts have been taken to develop more efficient DBSCAN algorithms in almost 30 years' research history. See a brief survey in Section 2.

Different with all the existing works, in this paper, the way to reduce unnecessary distance computations and improve algorithmic efficiency originates from a careful conceptual inspection of the DBSCAN problem. For the first time, several fine-grained sub-problems of the DBSCAN problem are identified, which solve specific sub-cases of the three critical components mentioned above

with less distance computations. Let us introduce the idea of these sub-problems using the following examples shown in Figure 1.

Example 1. In Figure 1c, a range search is executed on the black-colored point p, incuring distance computation between p and all the red points. However, as Figure 1d shows, knowing $minpts = 5$, it only needs to compute the distance from p to the four red-circled points to know p is a core point. Thus, a lot of distance computations can be spared during the procedure of identifying core points.

Example 2. In Figure 1e, assume that p is known to be a core point, and the red points are known to be a density-connected cluster of core points. Then, it only needs to compute the distance between p_1 and p to know p is density connected to the red cluster. Compared with the range search in Figure 1c, the distance computation between p_2, p_3 and p can be spared.

Example 3. In Figure 1f, assume that the red points and blue points are known to be two density-connected clusters of core points. To further know the two clusters are density-connected, it only needs to know the distance between one pair of points. The distance computation between the other pairs of points in these two clusters can all be avoided, saving a lot of time.

Summarizing the above examples, the central idea to avoid unnecessary distance computations is that, different procedures with less distance computations should be carried out for points known to be core point and point sets known to be one density-connected cluster. This insight has not been clearly stated in former research works as far as we know. We then define several new problems to represent the idea of the above examples. The problems in Figure 1d, 1e and 1f are called the ϵ-minpts, ϵ-connect-single, ϵ-connect-dual problems, respectively. This is the first time these sub-problems are defined for solving the DBSCAN problem, and the formal definitions will be given in Section 3.2.

With these new sub-problems defined, there comes new questions to answer, i.e., how to solve these new sub-problems, and how these new sub-problems contribute to solving the DBSCAN problem. Our second part of main contribution gives answer to these questions, that is the DBSCAN problem can be solved elegantly by a *generalized dual-tree* algorithm that integrates all these sub-problems in a one-pass tree traversal. Generalizing the dual-tree algorithm described by Curtin et al. [11] which includes four parts of logic split, the tree index structure, the traversal rule, the point-to-point base case, and the pruning rule, our proposed *generalized dual-tree* algorithm introduces a new logic part which is called the *intermediate cases*. When the traversal encounters an intermediate case which is specified by the information of the current visiting node, it will switch to a new traversal where different traversal rules would be applied on the sub-tree below the current node. In terms of the DBSCAN problem, the intermediate cases exactly correspond to the sub-problems newly identified in this paper. Thus, the proposed generalized dual-tree algorithm is a conceptual advancement not only against the existing DBSCAN approaches, and also against the original dual-tree algorithm.

The concrete contributions of this paper are listed below.

First, a novel algorithm for the DBSCAN problem is proposed named as GeDual-DBSCAN. This algorithm incorporates the newly defined DBSCAN sub-problems with the new idea of generalizing dual-tree algorithms by the so-called *intermediate cases*. The name comes from the Generalized Dual-tree algorithm, and it is introduced using three variants, each established based on the former one. The three variants are postfixed by Basic, InterCases and Transitions, respectively. The Basic invariant establishes the overall traversal order. The InterCases variant is a generalized dual-tree algorithm which integrates the intermediate cases with the basic traversal. The Transitions variant involves a novel idea of *state transitions* for the tree nodes, and reduces the number of distance computations as mush as possible. The correctness and complexity of these algorithms are also provided.

Second, substantial experimental results are presented to evaluate the performance of the GeDual-DBSCAN algorithms. It is shown that the final variant, GeDual-DBSCAN-Transitions, outperforms the state-of-the-art DBSCAN algorithm in almost all parameter settings on all datasets tested.

The rest of this paper is organized as follows. Section 2 overviews the related works. Section 3 introduces the preliminary knowledge, as well as the new DBSCAN sub-problems. Section 4 proposes the GeDual-DBSCAN algorithms progressively using the Basic, InterCases and Transitions variants. The experimental results are presented in Section 5. Finally Section 6 concludes this paper.

2 RELATED WORKS

2.1 Existing approaches for solving DBSCAN

There is vast literature on the DBSCAN problem due to many years of research efforts. Here we mention some important categories of methods.

2.1.1 Partition based methods. The idea is to first divide the input point set into smaller subsets, then try to utilize the intra-subset and inter-subset properties to reduce the number of distance computations. Existing works exhibit a multitude of ideas for partitioning. For example, there are ball-based partition methods such as k-nearest-neighbors balls [8], $\epsilon/2$-balls [7], and δ-relaxed balls [23]. The well-known baseline algorithm for grid-based partition was proposed by Gan and Tao [17], and recently a grid-tree data structure was introduced in [24]. A pre-clustering method proposed by Gholizadeh et al. [20] used k-means++ [2] to produce the initial partition.

2.1.2 Parallel implementations. Parallelization is widely used to accelerate DBSCAN algorithms. To mention just a few, Fries et al. [15] implemented DBSCAN on MapReduce. Wang et al. [48] proposed a theoretical and empirical fast parallel DBSCAN algorithm based on solving bichromatic closest pairs problem in parallel.

2.1.3 Non-exact methods. More efficient DBSCAN algorithms are possible if non-exact clustering result is allowed. For instance, a sub-sampled thus non-exact neighborhood graph was used for DBSCAN clustering in [25]. Weng et al. employed the HNSW index to support range query [49], which may produce error when identifying core points since HNSW index can not guarantee perfect recall [31]. The above mentioned k-means++ based method [20] also produces non-exact result since it ignores some inter-cluster neighbors.

2.2 DBSCAN Generalizations

While this paper focuses on designing more efficient DBSCAN algorithms under the original problem definition, the shortcomings

of DBSCAN have been long recognized, such as varied density [33], hardness for determining parameters [27], which are also the limitation of this paper. Numerous efforts have been made to generalize and enhance DBSCAN. For example, the OPTICS method is able to automatically extract the intrinsic density-based clustering structure [1]. Campello et al. introduced a hierarchical clustering method by which a tree of significant clusters can be constructed [6]. Another generalization allows the DBSCAN parameters to vary across different regions, thus well-suited for unbalanced data [34].

2.3 Dual-tree algorithm and dual-set problems

A dual-tree algorithm [11, 21] is given two tree structures called query tree $\mathcal{T}_Q$ and reference tree $\mathcal{T}_R$, respectively (it is allowed that $\mathcal{T}_Q = \mathcal{T}_R$), to traverse the node combinations $\mathcal{N}_q \in \mathcal{T}_Q$ and $\mathcal{N}_r \in \mathcal{T}_R$ to solve a specific problem. It is an algorithm framework parameterized by the problem specific `BaseCase` and `Score` functions. The `BaseCase` function describes the action to take on a combination of points, and the `Score` function describes the pruning rule.

Dual-tree algorithms are naturally suitable for *dual-set* problems, that is, given two points sets Q and R, where Q is called the query set and R is called the reference set, find a problem-specific subset $R_{p_q} \subset R$ for each $p_q \in Q$. Lots of problems can be characterized as a dual-set problem. For example, the All-Nearest-Neighbors problem [46], is to find the nearest neighbor of p_q in R for each $p_q \in Q$. The All-range-search problem, is to find $B(p_q, r) \cap R$ for each $p_q \in Q$ given a distance range r. The seminal approach for DBSCAN [14] actually solves the All-Range-Search problem.

Surprisingly, there are some problems that implicitly possesses dual-set problem characteristics. The most notable example is the Euclidean minimum spanning tree (EMST) problem. In a dual-set problem point of view, EMST is given one point set P (the query and reference set is the same in this case), to find the neighborhood points $R_p \in P$ for each p such that any shortest path from p to other points in P must be via R_p, and the EMST is constructed by adding edges between p and points in such R_p. Finding EMST can be elegantly solved by a dual-tree Boruvka algorithm [32].

The DBSCAN problem can also be regarded as a dual-set problem, that is, for each $p \in P$, find all $q \in P$ density-connected to p. While the seminal approach for DBSCAN [14] actually performs All-Range-Search which is redundant for solving DBSCAN, so far there are no existing dual-tree DBSCAN algorithms better than the trivial All-Range-Search algorithm, other than the newly proposed dual-tree algorithm in this paper.

3 PRELIMINARIES

3.1 Terminologies and notations

3.1.1 Metric space. This paper adopts the Euclidean distance, i.e., $D(x, y) = \sqrt{\sum_{i=1}^{d}(x_{(i)} - y_{(i)})^2}$ where $x = (x_{(1)}, \cdots, x_{(d)})$ and $y = (y_{(1)}, \cdots, y_{(d)})$ are d-dimensional points. Note that the capital $D(\cdot, \cdot)$ is used to denote the Euclidean distance function, and d is used to denote the number of dimension. Let $B(p, r)$ be the d-dimensional ball centered at p with radius r, i.e. $B(p, r) = \{x \in \mathbb{R}^d \mid D(x, p) \leq r\}$. The minimum bounding rectangle (MBR) of a point set P, is the d-dimensional axis-parallel rectangle with minimum length in each dimension that contains all points in P.

3.1.2 DBSCAN terminologies. From now on the input point set of the DBSCAN problem is denoted as P. The DBSCAN problem involves the following well-known terminologies based on two input parameters ϵ and $minpts$.

(*Core and non-core points*). A point p is called core point if $|B(p, \epsilon) \cap P| \geq minpts$, and non-core point otherwise.

(*Density-connectivity*). 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 $1 \leq i \leq k$.

(*Noise and border points*). A non-core point $p \in P$ is called a border point, if there exists at least one $q \in B(p, \epsilon)$ that is a core point. Otherwise it is called a noise point.

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

It is worth mentioning that our proposed algorithms always produce clusters exactly satisfying the above definition.

The following extra denotations are used in this paper.

(**Dense point sets**). A set of points S is called *dense*, if $\forall p \in S$ is a core point and $\forall p_1, p_2 \in S$ are density-connected.

(ϵ-**neighbors**). If $D(q, p) \leq \epsilon$ then p and q are called ϵ-neighbors of each other. It is also said that (p, q) is an ϵ-neighbor pair.

3.1.3 Union-find set. The union-find set is a very efficient data structure to represent a set of disjoint sets [9] and has been used for solving DBSCAN long ago [36]. Given a set S of elements, the union-find set stores a representative element $rep(p)$ for each element $p \in S$. There are two operations, `Union` and `Find`, where `Union`(p, q) sets the representative for p and q as the same, and `Find`(p) returns the representative $rep(p)$. By known results [45], the `Union` and `Find` operations can be executed in amortized $O(1)$ time.

Slightly abusing notations, in this paper the `Union` and `Find` operations are allowed to be applied on a set of points S, but only when it is known that all points in S have the same representative, i.e., `Find`(p)=`Find`(p') for all $p, p' \in S$. Under such constraints, `Find`(S) returns the single representative of the points in S. `Union`(p, S) is equivalent to `Union`(p, p') where $p' \in S$. `Union`(S_1, S_2) is equivalent to `Union`(p_1, p_2) where $p_1 \in S_1$ and $p_2 \in S_2$.

3.1.4 KDTree. KDTree [4] is a well-known classical space partition tree, and is adopted in this paper as the index structure for traversal due to its good theoretical properties. Given a point set P, each KDTree node is associated with a subset of P together with the minimal bounding box (MBR) of this subset. KDTree is a balanced binary tree, where the two children nodes are obtained by splitting the MBR of parent node at the middle point in one dimension, and the splitting dimension is circularly chosen as the tree is descended. In the following, a KDTree node will be denoted as $\mathcal{N}$. For convenience, $\mathcal{N}$ will sometimes be used to refer to the point set contained in this node if no ambiguity is involved.

There are several well-known properties for the KDTree.

(*KDTree for range search*). It is well known that the range search on KDTree takes $O(n^{1-1/d})$ time in worst case [29].

(*MBR for distance bounds*). Since each KDTree node is associated with an MBR, point-to-node and node-to-node distance bounds

Table 1: Summary of notations

Notation	Meaning
KDTree node information	
$\mathcal{N}$	a KDTree node
$\mathcal{N}$.left	left child of node $\mathcal{N}$
$\mathcal{N}$.right	right child of node $\mathcal{N}$
$\mathcal{N}$.diag	the longest diagonal of the MBR of $\mathcal{N}$
$\mathcal{N}$.count	the number of points contained in $\mathcal{N}$
algorithmic notations	
ϵ, $minpts$	the two DBSCAN parameters
$\mathcal{N}_q, \mathcal{N}_r$	the query and reference node
p_q, p_r	the query and reference point
$Nbrs(p)$	the ϵ-neighbors found so far for point p
$ufset$	a union-find set maintaining clustering information

are available by regarding the MBR as a boundary of the node. Let $Dmax(p, \mathcal{N})$, $Dmin(p, \mathcal{N})$ be the maximum and minimum distance from point p to a KDTree node $\mathcal{N}$, and let $Dmax(\mathcal{N}_1, \mathcal{N}_2)$, $Dmin(\mathcal{N}_1, \mathcal{N}_2)$ be the maximum and minimum distance between the two nodes, respectively. The four distance bounds can all be computed in $O(d)$ time based on the MBR.

In this paper some new notations related to the KDTree are used.

(**KDTree node as a compond data structure**). In algorithmic point of view, a KDTree node can be regarded as a compound data structure storing various information. Table 1 lists the notations.

(**Dense nodes**). A KDTree node is called dense, if it contains no less than $minpts$ points and the longest diagonal of its MBR is no more than ϵ, i.e., $\mathcal{N}$.count $\geq minpts$ and $\mathcal{N}$.diag $\leq \epsilon$. Defining dense KDTree node follows the idea of defining dense point sets.

(**Left-to-right node ordering**). Based on the notation of left and right children, a left-to-right relationship can be defined across the KDTree nodes. A node $\mathcal{N}_1$ is said to be on the left to anther node $\mathcal{N}_2$, intuitively, if $\mathcal{N}_1$ appears left to $\mathcal{N}_2$ in the tree, and formally, if $\mathcal{N}_{A1}$ is one ancestor of $\mathcal{N}$, $\mathcal{N}_{A2}$ is one ancestor of $\mathcal{N}_2$, such that $\mathcal{N}_{A1} \neq \mathcal{N}_{A2}$, $\mathcal{N}_{A1}$ and $\mathcal{N}_{A2}$ have the same parent, $\mathcal{N}_{A1}$ is the left child, and $\mathcal{N}_{A2}$ is the right child.

3.1.5 Dual-tree algorithm terminologies. The following terminologies are used to refer to different kinds of node combinations visited in a dual-tree algorithm. The phrase *node combination* may be abbreviated to *combination* if the context is clear, and expanded to *query-reference* combination for emphasizing. If both nodes in the combination are leaves, it will be called a *leaf combination*. If the two nodes in the combination are the same, it is called a *self combination*. The special combination where both nodes are the root is called the *root combination*. Note that the root combination will always be firstly visited at the beginning of the traversal. For a specific combination $(\mathcal{N}_q, \mathcal{N}_r)$, the term *sub-combination* is used to refer to any combination $(\mathcal{N}'_q, \mathcal{N}'_r)$ where $\mathcal{N}'_q$ and $\mathcal{N}'_r$ are descendant nodes of $\mathcal{N}_q$ and $\mathcal{N}_r$, respectively.

3.2 The newly defined DBSCAN sub-problems

Four new sub-problems of the DBSCAN problem are defined here. Besides the classical **Input** and **Output** statement for problem definitions, the definitions given below additionally have **Early Termination** and **Side Effect** statements. The **Early Termination** statement forces any algorithm for this problem to stop once the stated condition is satisfied, incurring less distance computations. The **Side Effect** statement describes some additional actions executed on the inputs after the algorithm is terminated, which may help other procedures and reflects the essence that they are sub-procedures of the DBSCAN problem.

Definition 3.1. The *ϵ-minpts* problem.
Input: query point p_q, reference set R.
Output: answer YES if $|B(p_q, \epsilon) \cap R| \geq minpts$, and NO otherwise.
Early Termination: terminate once $minpts$ number of ϵ-neighbors have been found.
Side Effect: (1) record the ϵ-neighbors of p_q encountered in the set $Nbrs(p_q)$; (2) mark p_q as core point if the answer is YES.

The *ϵ-minpts* problem is the initial problem to be solved for each point. For a core point p, the early termination rule should be applied after the algorithm discovers that p is core point. For a non-core point p, the side effect ensures that all its ϵ-neighbors are recorded, and the final process to decide whether it is border/noise is feasible by examining whether $Nbrs(p)$ contains a core point.

Definition 3.2. The *ϵ-minpts-in-dense* problem.
Input: query point p_q, the recorded ϵ-neighbors $Nbrs(p_q)$, reference set R which is dense.
Output: answer YES if $|Nbrs(p_q)| + |B(p_q, \epsilon) \cap R| \geq minpts$, and NO otherwise.
Early Termination: terminate once $minpts$ number of ϵ-neighbors have been found.
Side Effect: (1) put the newly found ϵ-neighbors for p_q into $Nbrs(p_q)$; (2) mark p_q as core point if the answer is YES; (3) add p_q into R if the answer is YES.

The *ϵ-minpts-in-dense* problem is the only one that is not illustrated in Section 1. Compared with the ϵ-minpts problem above, it utilizes the fact that the reference set R is dense, and involves an extra side effect that expands the cluster of R by adding p_q into it if the query points p_q is identified as a core point.

Definition 3.3. The *ϵ-connected-single* problem.
Input: core point p_q, reference set R which is dense.
Output: answer YES if $\exists p_r \in R$ such that $D(p_q, p_r) \leq \epsilon$, and NO otherwise.
Early Termination: terminate once such p_r is found.
Side Effect: add p_q to R if the answer is YES.

ϵ-connected-single deals with the relationship between a core point and a dense set. If it is found that p_q and R and density-connected via some point $r \in R$, p_q will be added into R.

Definition 3.4. The *ϵ-connected-dual* problem.
Input: query set Q, reference set R, which are both dense.
Output: answer YES if $\exists p_q \in Q, p_r \in R$ such that $D(p_q, p_r) \leq \epsilon$, and NO otherwise.
Early Termination: terminate once such pair is found.
Side Effect: union Q and R if the answer is YES.

ϵ-connected-dual deals with the relationship between two dense sets, and union the two dense sets if they are density-connected, expanding the cluster of dense sets.

Table 2: Part of traversal on KDTree in Figure 2.

Step	Contents and order	Selection	Descend
1	$(root, root)$	$(root, root)$	$(L, L), (L, R), (R, R)$
2	$(L, L) \prec (L, R) \prec (R, R)$	(L, L)	$(LL, LL), (LL, LR), (LR, LR)$
3	$(L, R) \prec (LL, LL) \prec (LL, LR) < (LR, LR) \prec (R, R)$	(L, R)	$(LL, R), (LR, R)$
4	$(LL, LL) \prec (LL, LR) \prec (LL, R) \prec (LR, LR) \prec (LR, R) \prec (R, R)$	(LL, LL)	$(LLL, LLL), (LLL, LLR), (LLR, LLR)$
5	$(LL, LR) \prec (LL, R) \prec (LLL, LLL) \prec (LLL, LLR) \prec (LLR, LLR) \prec (LR, LR) \prec (LR, R) \prec (R, R)$	(LL, LR)	$(LLL, LR), (LLR, LR)$
6	$(LL, R) \prec (LLL, LLL) \prec (LLL, LLR) \prec (LLL, LR) \prec (LLR, LR) \prec (LLR, LLR) \prec (LR, LR) \prec (LR, R) \prec (R, R)$	(LL, R)	(LLL, R, LLR, R)
7	$(LLL, LLL) < (LLL, LLR) < (LLL, LR) < (LLL, R) < (LLR, LR) < (LLR, LLR) < (LLR, R) < (LR, LR) < (LR, R) < (R, R)$	(LLL, LLL)	Leaf combination reached. Invoke `BaseCase`.
...	...	...	...

4 THE GEDUAL-DBSCAN ALGORITHM

The following auxiliary data structures are available for the algorithms introduced in this section. A KDTree $\mathcal{T}$ is built on the input set P, and root node is denoted as $root$. A priority queue $\mathcal{PQ}$ is used to ensure the traversal order. A union-find set $ufset$ is used to store the information of clusters. A set $Nbrs(p)$ is used to record the ϵ-neighbors encountered during the search for each point $p \in P$. See the second part of Table 1 for algorithmic notations.

The GeDual-DBSCAN algorithm is introduced using three variants, each enhancing the former one with some new ideas. The first variant mainly describes the order of tree traversal and how to implement the desired traversal order. The second variant introduces the *intermediate cases* corresponding to the sub-problems defined above, and describes how to solve these intermediate cases and how they are integrated in the overall traversal. The third variant, which is the most powerful, considers the fact that a node may be discovered as dense during the traversal and explains how to utilize this information on-the-fly using *state transition* functions.

4.1 GeDual-DBSCAN variant 1: basic traversal

The basic traversal algorithm is introduced in three parts, the traversal order, the `BaseCase` function, and the `Score` function.

4.1.1 Traversal order. Let us first consider the basic dual-tree traversal on KDTree without any pruning, where each leaf combination should be visited once and only once. The basic traversal is established on the following simple idea of avoiding duplicate distance computations. (1) If $D(p_q, p_r)$ is computed, then $D(p_r, p_q)$ can be avoided to compute. (2) If the combination $(\mathcal{N}_q, \mathcal{N}_r)$ is visited, then the combination $(\mathcal{N}_r, \mathcal{N}_q)$ can be avoided to visit.

A straightforward idea to avoid duplicate visit is to label the nodes in numbers $1, 2, \ldots, n$, and visit the combinations in an ordered way like $(1, 2), (1, 3), \ldots (2, 3), (2, 4) \ldots, (n-1, n), (n, n)$. For KDTree nodes, the numbering is unnecessary since an left-to-right order has already been defined. Using the KDTree in Figure 2 as an example, the traversal order should be as follows. First using LLL as the query node, visit reference nodes in the following order: $LLL, LLR, LRL, \cdots, RRR$. Then, using LLR as the query node, visit reference nodes in the following order: $LLR, LRL, \cdots, RRR$. So forth, until the last leaf combination (RRR, RRR) is visited. This order will be called the nested-loop order.

To realize the above order to visit the combinations, the algorithm uses a priority queue $\mathcal{PQ}$ to guide the traversal where the elements in it are the query-reference node combinations. Two parts of traversal rules are used to make a correct traversal order, which are the descending rules and priority rules.

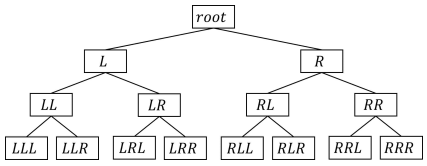


Figure 2: An example of KDTree node notations.

Descending rules. The descending rules apply to a visited combination $(\mathcal{N}_q, \mathcal{N}_r)$ which are not both leaves, to decide subsequent node combinations to be pushed into the priority queue for further traversal. The descending rules include the following three terms.

- (*Self descending*) If $\mathcal{N}_q = \mathcal{N}_r$, push the following combinations into $\mathcal{PQ}$: ($\mathcal{N}_q$.left, $\mathcal{N}_q$.left), ($\mathcal{N}_q$.left, $\mathcal{N}_q$.right), ($\mathcal{N}_q$.right, $\mathcal{N}_q$.right).
- (*Query descending*) If $\mathcal{N}_q \neq \mathcal{N}_r$ and $\mathcal{N}_q$ is not leaf, push ($\mathcal{N}_q$.left, $\mathcal{N}_r$) and ($\mathcal{N}_q$.right, $\mathcal{N}_r$) into $\mathcal{PQ}$.
- (*Reference descending*) If $\mathcal{N}_q$ is leaf, push ($\mathcal{N}_q, \mathcal{N}_r$.right) and ($\mathcal{N}_q, \mathcal{N}_r$.right) into $\mathcal{PQ}$.

The idea of query descending and reference descending rules is to first descend the query node down to the leaf, then start to descend the reference node using a leaf node as the query node. The self descending rule is to visit the node combinations recursively and ensure the traversal indeed visits all possible combinations.

Priority rules. The priority rules determine the order of visiting the node combinations stored in the priority queue. For two combinations $(\mathcal{N}_{q1}, \mathcal{N}_{r1})$ and $(\mathcal{N}_{q2}, \mathcal{N}_{r2})$, $(\mathcal{N}_{q1}, \mathcal{N}_{r1})$ should be visited before $(\mathcal{N}_{q2}, \mathcal{N}_{r2})$ if one of the following conditions happen:

- (*Ancestor query first*) $\mathcal{N}_{q1}$ is an ancestor of $\mathcal{N}_{q2}$;
- (*Left query first*) $\mathcal{N}_{q1}$ is left to $\mathcal{N}_{q2}$;
- (*Left reference first*) $\mathcal{N}_{q1} = \mathcal{N}_{q2}$ but $\mathcal{N}_{r1}$ is left to $\mathcal{N}_{r2}$.

The basic traversal works by integrating the descending rules and priority rules. An example traversal on the KDTree shown in Figure 2 is given in Table 2. The first column in the table is the number of steps. The second column shows the contents in the priority queue $\mathcal{PQ}$, and the priority order is shown using '$\prec$'. The third column is the current combination to be visited and it will be popped out of $\mathcal{PQ}$ after visited. The fourth column is the subsequent combinations to be pushed into $\mathcal{PQ}$. At the beginning of the algorithm the root combination $(root, root)$ is pushed into $\mathcal{PQ}$, and then the traversal goes as follows.

(1) $(root, root)$ is the top combination which is popped out. By *Self descending* rule, $(L, L), (L, R), (R, R)$ are pushed into $\mathcal{PQ}$.

(2) (L, L) is of highest priority, which is popped out. By *Self descending* rule, (LL, LL), (LL, LR), (LR, LR) are pushed into $\mathcal{PQ}$.
...
(6) (LL, R) should be visited by the *Ancestor query first* rule. By *Query descending* rule, (LLL, R), (LLR, R) are pushed in to $\mathcal{PQ}$.
(7) Now a leaf combination (LLL, LLL) is of highest priority where descending is not applicable. The `BaseCase` function should be invoked on leaf combinations, which will be introduced next.

It can be noticed that the first leaf combination visited is the left-most leaf LLL. Also, before the next leaf LLR is visited as query node, all combinations using LLL as query node will be traversed since the combinations with LLL as query node all have a higher priority. Thus, it is true that the desired traversal order is ensured.

Lemma 4.1. *The traversal induced by the the descending rules and priority rules ensures that the leaf combinations are visited in nested-loop order.*

Proof. Suppose to the contrary that there is a violation of the traversal order, that is, there exist two leaf combinations $(\mathcal{N}_{q1}, \mathcal{N}_{r1})$, $(\mathcal{N}_{q2}, \mathcal{N}_{r2})$, such that $(\mathcal{N}_{q1}, \mathcal{N}_{r1})$ is left to but visited after $(\mathcal{N}_{q2}, \mathcal{N}_{r2})$.

There are the following cases where $(\mathcal{N}_{q1}, \mathcal{N}_{r1})$ is left to $(\mathcal{N}_{q2}, \mathcal{N}_{r2})$.

(1) $(\mathcal{N}_{q1}$ is left to $\mathcal{N}_{q2})$. In such case, if $(\mathcal{N}_{q1}, \mathcal{N}_{r1})$ is visited after $(\mathcal{N}_{q2}, \mathcal{N}_{r2})$ it is a violation of the Left Query First rule, which must not exist.

(2) $(\mathcal{N}_{q1} = \mathcal{N}_{q2}$ and $(\mathcal{N}_{r1}$ is left to $\mathcal{N}_{r2})$. In such case if $(\mathcal{N}_{q1}, \mathcal{N}_{r1})$ is visited after $(\mathcal{N}_{q2}, \mathcal{N}_{r2})$ it is a violation of the Left Reference first rule, which must not exist.

The lemma is proved by contradiction, that is, there must be no violation of the nested-loop traversal order. □

4.1.2 `BaseCase` *function.* The `BaseCase` function is invoked on a leaf combination $(\mathcal{N}_q, \mathcal{N}_r)$. It is responsible to solve the DBSCAN problem partially on $(\mathcal{N}_q, \mathcal{N}_r)$, so that the DBSCAN problem will be solved after `BaseCase` is invoked on each leaf combination once. Specified to a single query point $p_q \in \mathcal{N}_q$, after the `BaseCase` function is invoked on all leaf combinations using $\mathcal{N}_q$ as query node, all its ϵ-neighbors should be found, and p_q should be joined with the core points in $Nbrs(p_q)$ if p_q is a core point.

This goal is divided into two stages. In the first stage, the ϵ-minpts problem is executed on p_q and the ϵ-neighbors are gradually added into $Nbrs(p_q)$ as the `BaseCase` function is invoked left-to-right across the leaf nodes. This stage is terminated once $|Nbrs(p_q)|$ exceeds $minpts$, and the `PointTurnToCore` function is invoked to join p_q with the core points in $Nbrs(p_q)$. Then it begins the second stage, where ϵ-neighbors newly found this stage will be joined with p_q into one cluster if the neighbor point is also a core point.

There are several important notes. First, if two points are already known in the same cluster, there is no need to compute the distance between them. Second, if $D(p_q, p_r) \le \epsilon$ and the reference point p_r is a non-core point, p_q should be added to $Nbrs(p_r)$ since p_q is also an ϵ-neighbor of p_r. In such way, although the `BaseCase` function is invoked in a left-to-right order on the leaf combinations, a point $p_r \in \mathcal{N}_r$ may be discovered as a core point before $\mathcal{N}_r$ is visited as query node. It needs to prove that such unordered discovery of core points still manages to find the correct DBSCAN clusters.

```
1  Function BaseCase_BothLeaves(N_q, N_r):
2      foreach p_q ∈ N_q, p_r ∈ N_r do
3          if |Nbrs(p_q)| ≥ minpts then
4              if ufset.Find(p_q)=ufset.Find(p_r) then
5                  continue;
6              Compute D(p_q, p_r);
7              if D(p_q, p_r) ≤ ε then
8                  if |Nbrs(p_r)| < minpts then AddNbr(p_r, p_q);
9                  else ufset.Union(p_q, p_r);
10         else
11             Compute D(p_q, p_r);
12             if D(p_q, p_r) ≤ ε then
13                 AddNbr(p_q, p_r);
14                 if |Nbrs(p_r)| < minpts then AddNbr(p_r, p_q);
15 Function AddNbr(p, nbr):
16     Add nbr into Nbrs(p);
17     if |Nbr(p)| ≥ minpts then PointTurnToCore(a);
18 Function PointTurnToCore(p):
19     foreach p' ∈ Nbrs(p) do
20         if |Nbrs(p')| ≥ minpts then ufset.Union(p, p');
```

Lemma 4.2. *For each pair of core points p_1, p_2 s.t. $D(p_1, p_2) \le \epsilon$, they have the same representative in the union-find set after the* `BaseCase` *function is invoked on all leaf combinations.*

Proof. This proof considers the time order of three events, which are $|Nbrs(p_1)|$ exceeding $minpts$, $|Nbrs(p_2)|$ exceeding $minpts$, and $D(p_1, p_2)$ being computed, denoted as C_1, C_2 and D, respectively. Suppose w.l.o.g. that C_1 happens earlier than C_2, and there are the following three cases, where '$\prec$' indicates the time order.

Case 1: $C_1 \prec C_2 \prec D$. The `Union` function must be called on the two points at Line 9, if p_1 and p_2 are not in the same cluster yet.

Case 2: $C_1 \prec D \prec C_2$. When $D(p_1, p_2)$ is computed, p_2 is not a core point and p_1 is added to $Nbrs(p_2)$ by Line 14. When $|Nbrs(p_2)|$ exceeds $minpts$, `Union` is called in *PointTurnToCore* procedure.

Case 3: $D \prec C_1 \prec C_2$. p_2 is added to $Nbrs(p_1)$ by Line 13, p_1 is added to $Nbrs(p_2)$ by line 14. When *PointTurnToCore* is called on p_1, p_2 is not core point yet then they can not be united. But when $|Nbrs(p_2)|$ exceeds $minpts$, `Union` will be successfully called.

It is proved that `Union` operation is executed the two points in every case, and they must have the same representative in the union-find set. □

4.1.3 `Score` *function.* By combining the traversal rules with the `BaseCase` function, a naive traversal which invokes the `BaseCase` function on each possible leaf combination can be established. However, an efficient tree traversal algorithm must have the ability to prune nodes unnecessary to be visited, which is the reason to introduce the `Score` function.

The `Score` function used here is invoked on a query-reference combination $(\mathcal{N}_q, \mathcal{N}_r)$, to test whether it is valid for further traversal. The return value of `Score` function is a non-negative real number *score*. If $score = 0$, it means that $\mathcal{N}_q$ and $\mathcal{N}_r$ are close enough that all points in $\mathcal{N}_q$ are ϵ-neighbors of all points in $\mathcal{N}_r$. If the *score* is a large enough number, which is supposed to be ∞,

it means that $\mathcal{N}_q$ and $\mathcal{N}_r$ are so far that it is impossible to find ϵ-neighbor pairs between the two nodes. For both cases there is no need to traverse the sub-combinations of $(\mathcal{N}_q, \mathcal{N}_r)$. If non of the two cases happen, the combination $(\mathcal{N}_q, \mathcal{N}_r)$ should be pushed into the priority queue $\mathcal{PQ}$ for further traversal.

Intuitively, the `Score` function must utilize upper and lower bounds on the pairwise distances between $p_q \in \mathcal{N}$ and $p_r \in \mathcal{N}_r$. Recall that $Dmax(\mathcal{N}_q, \mathcal{N}_r)$ and $Dmin(\mathcal{N}_q, \mathcal{N}_r)$, which are the maximum and minimum distances between the MBR of the two nodes, can well serve this goal. Thus, the `Score` function works by compare these two values against the DBSCAN parameter ϵ.

The `Score` function cooperates with `Decide` and `BaseCase_NearNodes` functions. The `Decide` function is invoked during the traversal when it needs to decide whether a combination $(\mathcal{N}_q, \mathcal{N}_r)$ should be pushed into the priority queue for further traversal. It invokes the `Score` function to make the decision. The `BaseCase_NearNodes` function is invoked when $score = 0$ to deal with the case that all points in $\mathcal{N}_q$ are ϵ-neighbors of all points in $\mathcal{N}_r$.

```
Function Decide(N_q, N_r, PQ):
    score ← Score(N_q, N_r);
    if score = 0 then BaseCase_NearNodes(N_q, N_r);
    else if score < ∞ then push (N_q, N_r) into PQ;
Function Score(N_q, N_r):
    Compute Dmax(N_q, N_r) and Dmin(N_q, N_r);
    if Dmax(N_q, N_r) ≤ minpts then return 0;
    else if Dmin(N_q, N_r) > ε then return ∞;
    else return Dmin(N_q, N_r);
Function BaseCase_NearNodes(N_q, N_r):
    foreach p_q ∈ N_q do
        if |Nbrs(p_q)| ≤ minpts then
            foreach p_r ∈ N_r do AddNbr(p_q, p_r);
    foreach p_r ∈ N_r do
        if |Nbrs(p_r)| ≤ minpts then
            foreach p_q ∈ N_q do AddNbr(p_r, p_q);
    foreach p_q ∈ N_q, p_r ∈ N_r that are both core do
        ufset.Union(p_q, p_r);
```

4.1.4 Overall algorithm. Now we are ready to present the overall GeDual-DBSCAN-Basic algorithm, which is given in Algorithm 1. The correctness and complexity of the algorithm are given below.

Theorem 4.3. *GeDual-DBSCAN-Basic produces correct DBSCAN clusters that are connected and maximized.*

Sketch. Claim: after the `BaseCase` function is invoked on all leaf combinations, all the ϵ-neighbors of all non-core points have been found.

Lemma 4.2 states that if the `BaseCase` function is invoked on all leaf combinations, then all pairs of density-connected core points should have the same representative in the union-find set. By the definition of `Score` function, any leaf combinations pruned by the `Score` function must not contain any ϵ-neighbor pair. This means any possible ϵ-neighbor pair of core points must have been found by the algorithm, which in turn means that the clusters of core points have been correctly found.

Algorithm 1: GeDual-DBSCAN-Basic

```
1  Push (root, root) into PQ;
2  while PQ is not empty do
3      (N_q, N_r) ← PQ.pop();          // gets the top element
4      if N_q is leaf then
5          if N_r is leaf then
6              BaseCase_BothLeaves(N_q, N_r);
7          else                  // Apply Reference descending rule
8              Decide(N_q, N_r.left, PQ); Decide(N_q, N_r.right, PQ);
9      else
10         if N_q = N_r then          // Apply Self descending rule
11             Push(N_q.left, N_q.left) and (N_q.right, N_q.right) into PQ;
12             Decide(N_q.left, N_q.right, PQ);
13         else                    // Apply Query descending rule
14             Decide(N_q.left, N_r, PQ); Decide(N_q.right, N_r, PQ));
15 AssignNonCorePoints();
16 Function AssignNonCorePoints():
17     foreach p s. t. |Nbrs(p)| ≤ minpts do
18         if ∃p′ ∈ Nbrs(p) s. t. |Nbrs(p′)| ≥ minpts then
19             ufset.Union (p, p′) and continue;
```

The connectivity comes from the connectivity of ϵ-neighboring core points, and the maximum come from all the possible ϵ-neighbor pairs have been found. Finally, the non-core points added to the cluster makes the correct clustering result. □

Theorem 4.4. *The time complexity of GeDual-DBSCAN-Basic algorithm is* $O(n^{2-1/d}) + O(n \log n) = O(n^{2-1/d})$.

Proof. We count the number of distance computations incurred by the algorithm. It can be divided into two parts, depending on whether the query node is leaf. The following proofs analyze the two parts respectively.

Part 1: We first claim that, for each leaf node $\mathcal{N}$, when it is the first time visited as a query node, the combinations $(\mathcal{N}, \mathcal{N}')$ in the priority queue involving $\mathcal{N}$ as query node, excepting the self-combination, satisfies that $\mathcal{N}'$ is a right child of some ancestor of $\mathcal{N}$. As Table 2 shows, when *LLL* is the first time visited as a query node, the combinations in $\mathcal{PQ}$ where $\mathcal{N}$ serves as query node, are (*LLL*, *LLR*), (*LLL*, *LR*), (*LLL*, *R*). The reference nodes, *LLR*, *LR*, *R*, all satisfy the claim that it is the right child of some ancestor of *LLL*. We now proceed to prove this claim.

First, such combination must appear in the priority queue. When an ancestor $\mathcal{N}_a$ of $\mathcal{N}$ is visited as a self combination, the child combination $(\mathcal{N}_{a,lc}, \mathcal{N}_{a,rc})$ is pushed into $\mathcal{PQ}$ by self descending rule. Then by query descending rule, before $\mathcal{N}_{a,lc}$ is descended to the leaf node $\mathcal{N}$, the reference node $\mathcal{N}_{a,rc}$ will not be descended, yielding the combination $(\mathcal{N}, \mathcal{N}_{a,rc})$ in the priority queue.

Second, there is no other combinations involving $\mathcal{N}$ as query node. At the time that $\mathcal{N}$ is first visited as a query node, the leaf combination $(\mathcal{N}, \mathcal{N})$ is with the highest priority. Thus, all the other combinations $(\mathcal{N}, \mathcal{N}')$ will not be visited, leaving them in the form that $\mathcal{N}'$ is a right child of some ancestor of $\mathcal{N}$.

Next, we claim that the subsequent traversal on each such combination $(\mathcal{N}, \mathcal{N}')$ is a range search on the sub-tree rooted at $\mathcal{N}'$.

This is easily verified by the *Score* function. The complexity of the range search is $|\mathcal{N}'|^{1-1/d}$, where $|\mathcal{N}'|$ denotes the number of leaf nodes in the sub-tree rooted at $\mathcal{N}'$. It can be noticed that the time complexity of these range searches depends on the tree height, and for each leaf node, range search will be executed on sub-trees with different height.

Now we label each node with a binary string which represents the left-right path from root to it. The root is labeled with the empty string. For the other nodes $\mathcal{N}$, the label is $label(\mathcal{N}_p) \oplus 0$ if $\mathcal{N}$ is the left child of the parent node $\mathcal{N}_p$, and $label(\mathcal{N}_p) \oplus 1$ if it is the right child, where $\oplus$ denotes string concatenation. For example, the label string of node *LLL* is 000, of *LR* is 01, etc. We number the bits starting from the right end, namely, the 0-th bit refers to the right-most bit in the string.

Under such denotations, we claim that, for each leaf node $\mathcal{N}$, a range search will be executed on a tree with height in $\{2^i \mid \text{the } i\text{-th bit in label is } 0\}$. For example, the leaf node *LLL* whose label is 000, will execute range search on sub-trees rooted at *LLR*, *LR*, *R*, whose height are $2^0, 2^1, 2^2$, matching the claim. Another example, the leaf node *LRL* whose label is 010, will execute range search on *LRR*, *R*, whose heights are $2^0, 2^2$, also matching the claim. This claim is easy to verify, since range search will and only will be executed on the right children of the ancestor nodes of $\mathcal{N}$, and only these 'left' nodes, indicated by the 0s in the label, have right children.

Now we count for each height $i \in 0, 1, 2, \cdots, \log n - 1$ the number of leaf nodes that will execute range search on a tree with height i. That is equivalent to counting the number of 0s on the i-th bit of all the binary strings of the leaf nodes. It is obvious that the binary strings represents numbers from 0 to n. Then it is easily verified that the total number of 0s on each bit of all the binary strings representing from 0 to n is no more than $\frac{n}{2}$.

Now we have the following summation for the total time of range search, that is the first part of time complexity.

$$\frac{n}{2} \sum_{i=0}^{\log n - 1} (2^i)^{1-1/d} = \frac{n}{2} \times \frac{1 - n^{1-1/d}}{1 - 2^{1-1/d}} = O(n^{2-1/d})$$

We conclude the first part of analysis with the following remark. Although the asymptotic time complexity is the same with executing n range searches on the whole KDTree, the constant hidden by the Big-O notation is reduced by approximately $2^{2-1/d}$.

Part 2: Now we analyze the number of distance computations incurred by non-leaf query nodes. We have a similar claim with that in Part 1. For each internal node $\mathcal{N}$, the distance computation will happen between $\mathcal{N}$ with those $\mathcal{N}'$ which is a right child of some ancestor of $\mathcal{N}$. For example, the internal node *LL*, will compute distance with *LR*, *R*, while the `Score` function is invoked on these combinations.

The proof for this claim is similar with that in Part 1 which is based on the descending rules. It is omitted for simplicity.

We continue with the binary string labeling introduced above. For an internal node $\mathcal{N}$, the number of distance computations happened on $\mathcal{N}$ exactly equals to the number of 0's in its binary string label. It is easy to prove since only an ancestor indicated by a 0 has a right child.

Now, for each layer of the KDTree, starting from the root as the 0-th layer, there are 2^i nodes in the i-th layer. For the j-th bit in the binary string label, $0 \le j \le i$, there are at most 2^{i-1} binary strings with 0 on the j-th bit. Therefore, the total number of 0s in the binary string labels of the nodes in the i-th layer, is $i \cdot 2^{i-1}$, since the length of the binary string of nodes in the i-th layer is i.

Now we can sum up, $\sum_{i=0}^{\log n} i \cdot 2^{i-1}$, yielding $O(n \log n)$.

We are finished counting the two parts of distance computations, and proved the claimed complexity in the theorem. □

Theorem 4.5. *The space complexity of the algorithm is* $O(n \log n)$.

Proof. The space complexity is divided into two parts, including the space for KDTree, and for the combinations pushed into the priority queue during traversal.

First, the KDTree takes $O(n \log n)$ space by known results.

Second, whenever a leaf node $\mathcal{N}$ is visited as query node, the range search takes $O(n \log n)$ space since there are at most $O(n \log n)$ nodes in the KDTree to visit. Also, before the range search is finished for this leaf node, the other combinations in the priority queue will remain untouched by the priority rules. We claim that all these combinations satisfy the following property: the query node and the reference node can only be the right child of some ancestor of $\mathcal{N}$. It can be easily proved by the descending rules and priority rules. Since the depth of the KDTree is $O(\log n)$, the number of the untouched combinations in the priority queue is $O(\log^2 n)$ at any time during the algorithm execution.

In conclusion, the space complexity of the algorithm is proved to be $O(n \log n)$. □

4.2 GeDual-DBSCAN variant 2: intermediate cases

The GeDual-DBSCAN-Basic algorithm establishes the framework of traversal order, but did not concern the sub-problems introduced in Section 3.2, i.e., the ϵ-minpts-in-dense, ϵ-connect-single and ϵ-connect-dual problems. The GeDual-DBSCAN-InterCases algorithm to be introduced in this section works by integrating these sub-problems into the traversal framework of the basic algorithm, which further reduces the number of distance computations.

Note that in this paper all these three sub-problems are solved by the most straightforward way. The point is that our generalized dual-tree algorithm framework is an orthogonal contribution with how these sub-problems are solved. By the experimental results in Section 5, it can be noticed that even no intricate techniques are used to solve these sub-problems, our GeDual-DBSCAN algorithm already outperforms the state-of-the-art.

Recall that the sub-problems defined in Section 3.2 require the input point set is known to be dense. To produce such information so that these sub-problems are applicable, an extra sub-procedure `FindInitialDenseNodes` is executed at the beginning of the algorithm. It is to check all the nodes in the KDTree to find the *dense* nodes, i.e, the nodes satisfying $\mathcal{N}.\text{diag} \le \epsilon$ and $\mathcal{N}.\text{count} \ge minpts$. It is easy to see points in such node forms a density-connected cluster of core points, meeting the definition of dense point sets.

The traversal begins after the dense nodes are identified. The priority rules, `Decide` and `Score` functions of the former Basic variant still completely apply for this InterCases variant, but they differ

Algorithm 2: GeDual-DBSCAN-InterCases

```
Push (root, root) into PQ;
FindInitialDenseNodes();
while PQ is not empty do
    (N_q, N_r) ← PQ.pop();
    if N_q is leaf or dense then
        if N_r is leaf or dense then
            if N_q = N_r and N_q is dense then continue;
            if N_q is dense and N_r is dense then
                InterCase_BothDense(N_q, N_r);
            else if N_q is leaf and N_r is dense then
                InterCase_LeafDense(N_q, N_r);
            else if N_q is dense and N_r is leaf then
                InterCase_LeafDense(N_r, N_q);
            else if N_q is leaf and N_r is leaf then
                BaseCase_BothLeaves(N_q, N_r);
        else
            Decide (N_q, N_r.left); Decide (N_q, N_r.right);
    else
        if N_q = N_r and N_q is not dense then
            Push(N_q.left, N_q.left) and (N_q.right, N_q.right) into PQ;
            Decide (N_q.left, N_q.right);
        else
            Decide (N_q.left, N_r); Decide (N_q.right, N_r);
AssignNonCorePoints();
Function FindInitialDenseNodes():
    foreach N in KDTree T do
        if N.diag≤ ϵ and N.count≥ ϵ then mark N as dense;
```

in descending rules. The Basic variant descends until leaf combinations, but in the InterCases variant, if a combination containing dense nodes is encountered during the traversal, it is said that an intermediate case is met, and the subsequent traversal will be substituted by what is defined by the `InterCase` functions. If both nodes in the combination are dense, the `InterCase_BothDense` function will be invoked. The other case is one node in the combination is leaf and the other is dense, which resorts to the `InterCase_LeafDense` function. A special case is that if a self combination of dense node is visited, there is no need for further descending. Before diving into the details of the intermediate cases, keep in mind that to ensure the correctness of the traversal algorithm, the intermediate cases should also solve the partial DBSCAN problem on the visited combination, so that the DBSCAN problem would be solved after all possible base cases and intermediate cases are visited.

4.2.1 `InterCase_BothDense`. If a combination $(\mathcal{N}_q, \mathcal{N}_r)$ with two dense nodes is visited, to produce the correct DBSCAN clusters it needs to test whether the two nodes are density-connected. Recall that it is exactly the functionality of the ϵ-connect-dual problem. Thus, the `InterCase_BothDense` function will change the traversal beneath $(\mathcal{N}_q, \mathcal{N}_r)$ so as to solve the ϵ-connect-dual problem.

Using a way similar with introducing the GeDual-DBSCAN-Basic algorithm, the traversal algorithm for ϵ-connect-dual problem

Intermediate Cases 1: both dense

```
Procedure InterCase_BothDense(N_q, N_r):
    if Score(N_q, N_r)= 0 then
        ufset.Union(N_q, N_r);
    else if Score(N_q, N_r)< ∞ then
        found ← false;
        EpsConnect_Dual(N_q, N_r, found);
        if found = true then ufset.Union(N_q, N_r);
```

is split into the following parts including the descending and priority rules, as well as the `BaseCase`, `Decide` and `Score` functions. Additionally, a *termination rule* should be involved to meet the Early Termination statement in the problem definition.

Descending rule. The descending rule is simple which only ensures all the possible leaf combinations are visited. Specifically, when a non-leaf combination is visited, descend to the four children combinations. If a combination with one leaf and one non-leaf is visited, descend the non-leaf node.

Priority rule. Unlike the GeDual-DBSCAN-Basic algorithm where the priority rule applies to all the combinations in a global priority queue, the priority rule of ϵ-connect-dual problem applies locally only to the sibling combinations obtained by the descending rule. It simply asks to visit the local sibling combinations in ascending order of the *score*.

Termination rule. A boolean value $found$ is maintained indicating whether a ϵ-neighbor pair is found. The algorithm will be terminated once $found$ turns true.

BaseCase. The `BaseCase` function simply iterates through all pairs of points to find an ϵ-neighbor pair, and terminate the iteration once such pair is found. It indeed solves the ϵ-connect-dual problem on the leaf combination as `BaseCase` functions should do.

Decide. There are three cases for the `Decide` function. The near case is that all points in $\mathcal{N}_q$ are within distance ϵ to all points in $\mathcal{N}_r$. In such a case the algorithm should be early terminated by setting $found$ to be true. The far case is that the points between the two nodes are at least ϵ apart, and this combination is pruned. Beside the two cases is the normal case, where the combination is put into the local traversal list for descending.

Score. The `Score` function used in GeDual-DBSCAN-Basic is reused, since the near and far case in the above `Decide` function can be successfully identified by that `Score` function.

The cooperation of priority rule, termination rule and `Score` function makes it possible to find an ϵ-neighbor pair earlier in combinations with lower score, resulting in earlier termination and less distance computations.

4.2.2 `InterCase_LeafDense`. The `InterCase_LeafDense` function deals with the combinations containing one leaf and one dense node. This intermediate case is separated into two sub-cases.

The first sub-case is that query node $\mathcal{N}_q$ is leaf and reference node $\mathcal{N}_r$ is dense. To solve the partial DBSCAN problem on $(\mathcal{N}_q, \mathcal{N}_r)$ knowing $\mathcal{N}_r$ is dense, further consider whether $p_q \in \mathcal{N}_q$ is a core point. For core point p_q, it exactly needs to solve the ϵ-connect-single problem, i.e., find but one ϵ-neighbor of p_q in $\mathcal{N}_r$ to know p_q and $\mathcal{N}_r$ are density-connected. If p_q is not known to be a core point

Sub-problem 1: ϵ-connect-dual

Procedure `EpsConnect_Dual`($\mathcal{N}_q, \mathcal{N}_r, found$):
  if $\mathcal{N}_q$ *and* $\mathcal{N}_r$ *are both leaves* **then**
    `BaseCase_EpsConnect_Dual`($\mathcal{N}_q, \mathcal{N}_r, found$);
    if found **then terminate**;
  else if *one of* $\mathcal{N}_q$ *and* $\mathcal{N}_r$ *is leaf* **then**
    Let $\mathcal{N}_l$ be the leaf one, and $\mathcal{N}_{nl}$ be the other;
    Make an empty local traversal list LTL;
    `Decide_EpsConnect_Dual`($\mathcal{N}_l, \mathcal{N}_{nl}$.*left*, $found, LTL$);
    `Decide_EpsConnect_Dual`($\mathcal{N}_l, \mathcal{N}_{nl}$.*right*, $found, LTL$);
    foreach $(\mathcal{N}_1, \mathcal{N}_2)$ *in* LTL **do**
      if $found = true$ **then terminate**;
      `EpsConnect_Dual`($\mathcal{N}_1, \mathcal{N}_2, found$);
  else
    Make an empty local traversal list LTL;
    `Decide_EpsConnect_Dual`($\mathcal{N}_q$.*left*, $\mathcal{N}_r$.*left*, $found, LTL$);
    `Decide_EpsConnect_Dual`($\mathcal{N}_q$.*left*, $\mathcal{N}_r$.*right*, $found, LTL$);
    `Decide_EpsConnect_Dual`($\mathcal{N}_q$.*right*, $\mathcal{N}_r$.*left*, $found, LTL$);
    `Decide_EpsConnect_Dual`($\mathcal{N}_q$.*right*, $\mathcal{N}_r$.*right*, $found, LTL$);
    foreach $(\mathcal{N}_1, \mathcal{N}_2)$ *in* LTL **do**
      if $found = true$ **then terminate**;
      `EpsConnect_Dual`($\mathcal{N}_1, \mathcal{N}_2, found$);
Function `BaseCase_EpsConnect_Dual`($\mathcal{N}_q, \mathcal{N}_r, found$):
  foreach $p_q \in \mathcal{N}_q, p_r \in \mathcal{N}_r$ **do**
    if $D(p_q, p_r) \leq \epsilon$ **then**
      $found \leftarrow true$ and **terminate**;
Function `Decide_EpsConnect_Dual`($\mathcal{N}_q, \mathcal{N}_r, found, LTL$):
  $score \leftarrow$`Score`($\mathcal{N}_q, \mathcal{N}_r$);
  if $score = 0$ **then**
    $found \leftarrow true$ and **terminate**;
  else if $score < \infty$ **then**
    Add $(\mathcal{N}_q, \mathcal{N}_r)$ into LTL, and sort in ascending order of $score$;

yet, the ϵ-minpts-in-dense problem should be solved. Namely, first find enough ϵ-neighbors of p_q in $\mathcal{N}_r$ to know it is a core point, and it is immediately known that p_q and $\mathcal{N}_r$ are density-connected.

The second sub-case is that $\mathcal{N}_q$ is dense and $\mathcal{N}_r$ is leaf. To solve the partial DBSCAN problem on $(\mathcal{N}_q, \mathcal{N}_r)$, the following actions should be executed for each $p_q \in \mathcal{N}_q$. First find all the ϵ-neighbors of p_q in $\mathcal{N}_r$. If the neighbor point p_r is already a core point, join p_q and p_r into one cluster. Otherwise, add p_q into $Nbrs(p_r)$. Note that these actions are initiated by query points and the reference point are passively visited, but it can be equivalently rephrased using actions initiated by the reference points. For each $p_r \in \mathcal{N}_r$, if p_r is a core point, then if there exist $p_q \in \mathcal{N}_q$ such that $D(p_r, p_q) \leq \epsilon$, p_r will be joined with $\mathcal{N}_q$. If p_r is not known to be a core point yet, find enough ϵ-neighbors of p_r in $\mathcal{N}_q$ to discover it is a core point, and then join p_r with $\mathcal{N}_q$ into one cluster. Thus, it is equivalent to solve ϵ-connect-single problem for core points in $\mathcal{N}_r$ and solve ϵ-minpts-in-dense problem for the other points.

In both sub-cases, ϵ-connect-single and ϵ-minpts-in-dense problems should be solved, using points in the leaf node as query point and using the dense node as the reference node. The traversal algorithm for these two problems are introduced below.

Intermediate Cases 2: one leaf and one dense

Procedure `InterCase_LeafDense`($\mathcal{N}_{leaf}, \mathcal{N}_{dense}$):
  foreach $p \in \mathcal{N}_{leaf}$ **do**
    if $|Nbrs(p)| \geq minpts$ **then**
      $found \leftarrow false$;
      `EpsConnect_Single`($p, \mathcal{N}_{dense}, found$);
      if $found = true$ **then** $ufset$.`Union`($p, \mathcal{N}_{dense}$);
    else
      `EpsMinptsInDense`($p, \mathcal{N}_{dense}$);
      if $|Nbr(p)| \geq minpts$ **then** $ufset$.`Union`($p, \mathcal{N}_{dense}$);

ϵ-connect-single. The algorithm is similar to but simpler than that for ϵ-connect-dual problem since there is only one tree to descend for ϵ-connect-single problem.

Descending rule. Simply descend the reference node to the two children nodes, until leaf node is reached.

Priority rule. Locally visit the child node with lower score first.

Termination rule. The algorithm should be terminated once the boolean indicator $found$ turns true.

BaseCase. Iterates all points in the reference node, and terminate once an ϵ-neighbor is found.

Decide. Similar with the `Decide` function of ϵ-connect-dual problem, the near case is that all points in $\mathcal{N}_r$ are within ϵ distance to p_q, the far case is that all points in $\mathcal{N}_r$ are ϵ far away from p_q, and excepting the near and far case is the normal case.

Score. The `Score` function is based on the point-to-node distance bound of KDTree nodes, i.e., $Dmax(p_q, \mathcal{N}_r)$ and $Dmin(p_q, \mathcal{N}_r)$. The near and far cases are identified by comparing these two values with ϵ. For normal case $Dmin(p_q, \mathcal{N}_r)$ is used as the $score$ value.

ϵ-minpts-in-dense. The algorithm has a similar base case with the GeDual-DBSCAN-Basic algorithm since they both solve the ϵ-minpts problem for query point p_q, but traversal body is simpler since there is only one tree to descend.

Descending rule. Descend the reference node to the two children nodes until leaf node.

Priority rule. Locally visit the child node with lower score first.

Termination rule. Terminate once $|Nbrs(p_q)|$ exceeds $minpts$.

BaseCase. Iterate all points in the reference node to find the ϵ-neighbors. Terminate once $|Nbrs(p_q)| \geq minpts$.

Decide. The `Decide` function is similar with that of the GeDual-DBSCAN-Basic algorithm. The near case is dealt by the `BaseCase_EpsMinptsInDense_NearNode` function where all points in $\mathcal{N}_r$ are ϵ-neighbor of p_q. The far case is pruned since it is impossible to find an ϵ-neighbor of p_q in $\mathcal{N}_r$.

Score. The `Score_Single` function in ϵ-connect-single algorithm is reused, since it suffices to separate the near and far cases for the ϵ-minpts-in-dense problem.

The correctness and complexity of the overall GeDual-DBSCAN-InterCases algorithm are given below.

Theorem 4.6. *GeDual-DBSCAN-InterCases produces correct DBSCAN clusters that are connected and maximized.*

Sub-problem 2: ϵ-connect-single

Procedure EpsConnect_Single($p_q, \mathcal{N}_r, found$):
  if $\mathcal{N}_r$ *is leaf* **then**
    BaseCase_EpsConnect_Single($p_q, \mathcal{N}_r, found$);
  else
    Make an empty local traversal list LTL;
    Decide_EpsConnect_Single($p_q, \mathcal{N}_r.left, found, LTL$);
    Decide_EpsConnect_Single($p_q, \mathcal{N}_r right, found, LTL$);
    foreach $refNode$ *in* LTL **do**
      if $found = true$ **then terminate**;
      EpsConnect_Single($p_q, refNode, found$);
Function BaseCase_EpsConnect_Single($p_q, \mathcal{N}_r, found$):
  foreach $p_r \in \mathcal{N}_r$ **do**
    if $D(p_q, p_r) \leq \epsilon$ **then**
      $found \leftarrow true$ and **terminate**;
Function Decide_EpsConnect_Single($p_q, \mathcal{N}_r, found, LTL$):
  $score \leftarrow$Score_Single($p_q, \mathcal{N}_r$);
  if $score = 0$ **then**
    $found \leftarrow true$ and **terminate**;
  else if $score < \infty$ **then**
    Add $\mathcal{N}_r$ into LTL, and sort in ascending order of $score$;
Function Score_Single($p_q, \mathcal{N}_r$):
  Compute $Dmax(p_q, \mathcal{N}_r)$ and $Dmin(p_q, \mathcal{N}_r)$;
  if $Dmax(p_q, \mathcal{N}_r) \leq \epsilon$ **then return** 0;
  else if $Dmin(p_q, \mathcal{N}_r) > \epsilon$ **return** ∞;
  else return $Dmin(p_q, \mathcal{N}_r) > \epsilon$;

Sub-problem 3: ϵ-minpts-in-dense

Procedure EpsMinptsInDense($p_q, \mathcal{N}_r$):
  if $\mathcal{N}_r$ *is leaf* **then**
    BaseCase_EpsMinptsInDense($p_q, \mathcal{N}_r$);
  else
    Make an empty local traversal list LTL;
    Decide_EpsMinptsInDense($p_q, \mathcal{N}_r.left, LTL$);
    Decide_EpsMinptsInDense($p_q, \mathcal{N}_r.right, LTL$);
    foreach $refNode$ *in* LTL **do**
      if p_q becomes core **then terminate**;
      EpsMinptsInDense($p_q, refNode$);
Function BaseCase_EpsMinptsInDense($p_q, \mathcal{N}_r$):
  foreach $p_r \in \mathcal{N}_r$ **do**
    if $D(p_q, p_r) \leq \epsilon$ **then** AddNbr(p_q, p_r);
    if $|Nbrs(p_q)| \geq minpts$ **then terminate**;
Function Decide_EpsMinptsInDense($p_q, \mathcal{N}_r, LTL$):
  $score \leftarrow$Score_Single($p_q, \mathcal{N}_r$);
  if $score = 0$ **then**
    BaseCase_EpsMinptsInDense_NearNodes($p_q, \mathcal{N}_r$);
  else if $score < \infty$ **then**
    Add $\mathcal{N}_r$ into LTL, and sort in ascending order on score;
Function BaseCase_EpsMinptsInDense_NearNodes($p_q, \mathcal{N}_r$):
  foreach $p_r \in \mathcal{N}_r$ **do**
    AddNbr(p_q, p_r);
    if $|Nbrs(p_q)| \geq minpts$ **then terminate**;

Sketch. Claim 1: Each intermediate case is able to successfully solve the partial DBSCAN problem on the visited combination. This follows by the definition and pseudo codes.

Claim 2: The traversal considers all kinds of intermediate cases. This follows by the pseudo codes. Any combination containing a dense node corresponds to one of the intermediate case. The Both-Dense, Leaf-Dense, Dense-Leaf cases are handled by the pseudo codes. Any other combination are correctly traversed by the priority rules and descending rules.

Claim 2: Every combination that possibly containing an ϵ-neighbor pair is ensured to be visited. This follows by the Score functions of each sub-problem. □

Lemma 4.7. *The time complexity of ϵ-minpts-in-dense is $O(n^{1-1/d} + minpts)$, the expected time complexity of ϵ-connect-dual is $O(\sqrt{mn})$, and the time expected complexity of ϵ-connect-single is $O(\sqrt{n})$. For ϵ-connect-dual, m and n respectively denote the number of points contained in the two sub-trees.*

Proof. We proof the complexity of the three problems one by one.

(1) The ϵ-minpts-in-dense problem is the early-termination version of range search. By applying the classical complexity of range search on KDTree, the complexity $O(n^{1-1/d} + minpts)$ is obtained since at most $minpts$ number of neighbors should be reported.

(2) Let $T(n)$ be the time complexity of executing ϵ-connect-single on a sub-tree with n leaf nodes. We then have a recursive formula $T(n) = aT(n/2) + O(1)$, where a is the number of sub-trees needed to be visited. Obviously, $a = 2$ in worst case and $a = 1$ in best case, inducing to $O(n)$ worst case and $O(\log n)$ best case complexity, respectively. On average case, we extend the recursive formula by one step, writing it as $T(n) = a'T(n/4) + O(1)$. It is then reasonable to set $a' = 2$, since it is reasonable to consider that the algorithm could be terminated by visiting the former two sub-trees on average case. This leads to an average case complexity of $O(\sqrt{n})$.

(3) Let $T(m, n)$ be the time complexity of executing ϵ-connect-dual on two sub-trees with m and n leaf nodes, respectively. We have $T(m, n) = aT(m/2, n/2) + O(1)$. Assume w.l.o.g that $m \geq n$. By similar argument, the worst and best case would have $a = 4$ and $a = 1$ and lead to $O(mn)$ and $O(\log mn)$ time complexity, respectively. In average case, setting $a = 2$ is reasonable. We work with the $a = 2$ to get the claimed complexity. Extend the recursive formula till n reaches 1, and we get $T(m, n) = nT(m/n, 1) + O(n)$. Now $T(m/n)$ reduces to ϵ-connect-single problem and solves to $T(m/n) = \sqrt{m/n}$. The final complexity is $O(\sqrt{mn}) + O(n) = O(\sqrt{mn})$ since $m \geq n$. □

Theorem 4.8. *Suppose all the dense nodes contain n_0 points. The time complexity of GeDual-DBSCAN-InterCases is $\Theta((n - n_0)n^{1-1/d})$.*

Proof. The idea of the proof is to give an upper bound and a lower bound for the complexity of the algorithm.

For the upper bound side, we consider the following naive algorithm which must visit more nodes than our GeDual-DBSCAN-InterCases. For each point other than those contained in the dense nodes, execute a range search on the whole KDTree. The procedure definitely compute the correct DBSCAN result and runs in $O((n - n_0)n^{1-1/d})$ time. This gives a time complexity upper bound on our algorithm.

For the lower bound side, we consider the following special case. Suppose all these dense nodes form one sub-tree rooted at N_0, which contains only dense nodes, and N_0 is on the right-most position. In such a case, the operations executed on all the non-dense nodes by the InterCases variant, is almost the same with those operations executed by the Basic variant, except for executing the ϵ-minpts-in-dense algorithm on the last right-most sub-tree rooted at N_0. Furthermore, notice that ϵ-minpts-in-dense is an early-termination version of range search with the same asymptotic time complexity $O(n^{1-1/d})$, and we can conclude that the time complexity incurred by those non-dense leaf nodes are the same with that in the Basic variant, which sum up to $O((n-n_0)n^{1-1/d})$. The final complexity for this special case is $O((n-n_0)n^{1-1/d})$ since no further operations will be executed on the dense nodes in the sub-tree rooted at N_0. Recall that this is a special case, and the worst case time complexity must be no less than $O((n-n_0)n^{1-1/d})$, which gives a lower bound.

Since the upper and lower bound matches, we conclude that the time complexity of GeDual-DBSCAN-InterCases is $\Theta((n-n_0)n^{1-1/d})$. □

4.3 GeDual-DBSCAN variant 3: state transitions

The GeDual-DBSCAN-InterCases algorithm uses the information of dense nodes but not to the maximum extent. The dense nodes are identified at the beginning and the whole algorithm depends only on this initial knowledge. However, a node initially not dense may become dense as the algorithm proceeds, since points in the node are continuously discovered to be core points and neighboring core points are joined into a cluster. To utilize the information of dense nodes on-the-fly, the final variant of our algorithm is proposed which is called GeDual-DBSCAN-Transitions.

The core idea of GeDual-DBSCAN-Transitions is to attach some boolean *state indicators* the nodes showing certain information. Specifically, each node is associated with three indicators named *AllCore*, *OneCluster*, and *Dense*. The *AllCore* indicator is to show whether all points in this node are core points. The *OneCluster* indicator is to show whether all points in this node are in the same cluster. The *Dense* indicator is, by the definition of dense nodes, the conjunction of the *OneCluster* and *AllCore* indicators.

All indicators are false at the beginning of the algorithm. Then the `InitialCheck` function is invoked on each node $\mathcal{N}$ to set the initial states of the nodes. If $\mathcal{N}$.diag$\leq \epsilon$ then the *OneCluster* indicator is set to be true. If further $\mathcal{N}$.count$\geq$ *minpts* then *AllCore* and *Dense* indicators are set to be true. The `InitialCheck` function should replace the `FindInitialDenseNodes` function in the InterCases variant. Next the transition functions for these three state indicators are introduced respectively.

***AllCore* indicator.** An auxiliary function `IncNumCore` operating a counter $\mathcal{N}$.numCores helps to set the *AllCore* indicator of each node $\mathcal{N}$. The `PointTurnToCore` function invoked on point p triggers the `IncNumCore` function on the node $\mathcal{N}$ containing p to increase the $\mathcal{N}$.numCores counter. When $\mathcal{N}$.numCores reaches $\mathcal{N}$.count, the transition function `NodeTurnTo_AllCore` will be invoked to set the *AllCore* indicator. It further triggers the checking process for the *OneCluster* indicator, and invokes the transition function for *Dense* indicator if *OneCluster* is set to be true.

AllCore indicator related functions.

```
1  Function InitialCheck():
2      foreach KDTree node N do
3          if N.diag≤ ε then
4              N.OneCluster← true;
5              if N.count≥ minpts then
6                  N.AllCore← true and N.Dense← true;
7  Function IncNumCore(N, N_q):
8      Increase N.numCores by 1;
9      if N.numCores= N.count then
10         NodeTurnTo_AllCore(N, N_q);
11 Function NodeTurnTo_AllCore(N, N_q):
12     N.AllCore ← true;
13     if N = N_q then flag ←CheckOneCluster(N);
14     else flag ←CheckOneCluster_BruteForce(N);
15     if flag = true then NodeTurnTo_Dense(N, N_q);
```

***OneCluster* indicator.** The checking process is triggered by the `NodeTurnTo_AllCore` function to test whether the *OneCluster* indicator should be set, and there are two different processes for query nodes and reference nodes respectively. For query nodes, a cheaper method based on union-find set is used. The other function for reference nodes computes all the pairwise distances among the points in $\mathcal{N}$. The reason to use two difference functions is as follows. For a query node $\mathcal{N}_q$, by that time `CheckOneCluster` is triggered by `NodeTurnTo_AllCore`, all the pairwise distances among points in $\mathcal{N}_q$ must have been computed since the self combination of $\mathcal{N}_q$ must have been visited. But it is not the case for a reference node $\mathcal{N}_r$. To know whether the reference node is of one cluster as early as possible, it is better to compute the pairwise distances among points in $\mathcal{N}_r$ at this time before $\mathcal{N}_r$ is visited as query node.

OneCluster indicator related functions.

```
1  Function CheckOneCluster(N):
2      p_1 ← an arbitrary points in N;
3      foreach p ∈ N do
4          if ufset.Find(p)≠ ufset.Find(p_1) then return false;
5      N.OneCluster ← true;
6      return true;
7  Function CheckOneCluster_BruteForce(N):
8      foreach p_1, p_2 ∈ N do
9          if ufset.Find(p_1)= ufset.Find(p_2) then continue;
10         if D(p_1, p_2) ≤ ε then ufset.Union(p_1, p_2);
11     return CheckOneCluster(N);
```

***Dense* indicator.** The transition function for the *Dense* indicator tries to propagate the *Dense* state to the ancestors so that subsequent traversal could switch to the less expensive intermediate cases earlier. The propagation starts from the $\mathcal{N}_p$ which is the parent node of $\mathcal{N}_r$. When the two children nodes of $\mathcal{N}_p$ are both dense, the ϵ-connect-dual algorithm will be used to check whether they are density-connected. If so, $\mathcal{N}_p$ is set to *Dense* and the procedure recursively ascends in the tree. The propagation

Dense indicator related functions.

```
Function NodeTurnTo_Dense(N, N_q):
    N.Dense← true and N_p ← the parent of N;
    while N_p is not null and is not ancestor of N_q do
        if the children of N_p are both dense then
            if ufset.Find(N_p.left)=ufset.Find(N_p.right) then
                N_p.Dense← true, and N_p ← the parent of N_p;
                continue;
            else
                found ← true;
                EpsConnect_Dual(N.left, N.right, found);
                if found then
                    ufset.Union(N_p.left,N_p.right);
                    N_p.Dense← true, and N_p ← the parent of N_p;
                    continue;
                else break;
        else break;
```

must stop when $\mathcal{N}_p$ is an ancestor of $\mathcal{N}_q$ to avoid duplicate distance computations between $\mathcal{N}_p$ and $\mathcal{N}_r$.

Finally, the overall traversal body of GeDual-DBSCAN-Transitions is given in Algorithm 3. The algorithm cooperates with the state transition functions via the `AddNbr` function invoked in the base case functions, since it triggers the `IncNumCore` function which is the single entrance point for all the state transition functions.

There are several new skipping conditions in Algorithm 3. If a self combination is visited and one of the *OneCluster* or *AllCore* indicators is true, this node will not be recursed into. The rational is that, if a query node is *AllCore* or *OneCluster*, it must be set by the `InitialCheck` function, and there is no need to compute the distances among the points. If a reference node is *AllCore*, `NodeTurnTo_AllCore` must have been invoked and the pairwise distances among points in this node have already been computed.

The correctness and time complexity of the overall GeDual-DBSCAN-Transitions algorithm are given below.

Theorem 4.9. *GeDual-DBSCAN-Transitions produces correct DBSCAN clusters that are connected and maximized.*

Sketch. The traversal body of Transitions and InterCases is the same except for the skip condition on *OneCluster* and *AllCore* nodes.

Claim 1: The skipping condition on *OneCluster* and *AllCore* nodes is correct and it never throws a possible ϵ-neighbor pair. It follows from our former explanations on the new skipping conditions.

The other difference of the two variants are the state transitions. The idea to prove the correctness divides into two parts. The first is to prove the state transitions always put a node into a correct state, which is easy. The other part is to prove any state transition on a combination $\mathcal{N}_q, \mathcal{N}_r$ will never affect the state of the combinations which should be visited later than it. If so, any combination will be visited in a fresh new correct state.

Claim 2: The state transition functions are correct so that any state transition function always sets the node into a correct state.

Algorithm 3: GeDual-DBSCAN-Transitions

```
Push (root, root) into PQ;
InitialCheck();
while PQ is not empty do
    (N_q, N_r) ← PQ.pop();
    if N_q is leaf or dense then
        if N_r is leaf or dense then
            if N_q = N_r and N_q is OneCluster or AllCore then
                continue;
            if N_q is dense and N_r is dense then
                InterCase_BothDense(N_q, N_r);
            else if N_q is leaf and N_r is dense then
                InterCase_LeafDense(N_q, N_r);
            else if N_q is dense and N_r is leaf then
                InterCase_LeafDense(N_r, N_q);
            else if N_q is leaf and N_r is leaf then
                BaseCase_BothLeaves(N_q, N_r);
        else
            Decide (N_q, N_r.left); Decide (N_q, N_r.right);
    else
        if N_q = N_r and N_q is not OneCluster nor AllCore then
            Push(N_q.left, N_q.left) and (N_q.right, N_q.right) into PQ;
            Decide (N_q.left, N_q.right);
        else
            Decide (N_q.left, N_r); Decide (N_q.right, N_r);
AssignNonCorePoints();
```

Claim 3: Any time a state transition happens on node $\mathcal{N}$ when combination $(\mathcal{N}_q, \mathcal{N}_r)$ is being visited, there does not exist a combination $(\mathcal{N}'_q, \mathcal{N}'_r)$ in the priority queue $\mathcal{PQ}$ such that $\mathcal{N}_q \neq \mathcal{N}'_q$ and $\mathcal{N}'_r$ is a descendant of $\mathcal{N}$. This claim can be proved by the descending rules and priority rules of the basic traversal.

□

Theorem 4.10. *Suppose there are n_1 points that are identified as core point before it is visited as a query point, then the time complexity of GeDual-DBSCAN-Transitions is $O((n-n_1)n^{1-1/d})$.*

Proof. This theorem can be proved by similar techniques used in Theorem 4.8. and it is omitted. □

4.4 Complexity comparison

Our Basic algorithm has the same asymptotic complexity with the original method [14], and similar with the algorithm proposed by Gan and Tao [17]. Note that Gan and Tao proposed both exact and non-exact algorithms for DBSCAN, and the complexity shown in the table corresponds to the exact one. Their algorithm is based on another theoretical algorithm for the bichromatic closest pair problem which was not given in detail. To the contrary, our Basic algorithm is not only given in detailed implementation. Furthermore, the hidden constants in the complexity of our Basic algorithm is approximately halved by the proof of Theorem 4.4.

SJ-DBSCAN [5] implements the nested-loop similarity join on the leaf nodes, which can be regarded as invoking `BaseCase` function for $O(n^2)$ times without pruning.

For the other two variants, the complexities are instance-specific, where n_0 and n_1 are the number of early-identified core points which vary with ϵ and $minpts$. The comparative analysis shows our methods have instance-specific advantage and thus are expected to exhibit better performance on a per-instance basis.

Table 3: Complexity comparison

method	complexity
Original [14]	n range searches ($O(n^{2-1/d})$ if using KDTree)
GanTao [17]	$O(n^{2-\frac{2}{\lceil d/2 \rceil+1}+\delta})$
SJ-DBSCAN [5]	$O(n^2)$
GeDual-DBSCAN-Basic	$O(n^{2-1/d})$
GeDual-DBSCAN-InterCases	$O((n-n_0)n^{1-1/d})$
GeDual-DBSCAN-Transitions	$O((n-n_1)n^{1-1/d})$

5 EXPERIMENTS

5.1 Preparations

Our methods. The three algorithm variants proposed in this paper will be referred as *Basic*, *InterCases* and *Transitions*, respectively.

Methods compared. The following methods are considered. *GanTao* [16] and *SJ-DBSCAN* are two baseline algorithms mentioned before. The binary executable of *GanTao* is available at [16], and *SJ-DBSCAN* is implemented by executing `BaseCase` function on leaf combinations in nested-loop order. *mlpack* [10] implements the dual-tree range search algorithm for solving DBSCAN, and it is used as a baseline to compare with our generalized dual-tree algorithms. *GAP* [23] and *GriT* [24] are the most recent implementations and the state-of-the-art DBSCAN algorithms.

The following four methods are considered. *GanTao* is a baseline DBSCAN algorithm provided by Gan and Tao [18], and the binary executable is available at [16]. *mlpack* [10] implements the dual-tree range search algorithm for solving DBSCAN, and it is used as a baseline to compare with our generalized dual-tree algorithms. *GAP* [23] and *GriT* [24] are the most recent implementations and the state-of-the-art DBSCAN algorithms.

Datasets. Both synthetic and real-world datasets are used. The synthetic datasets are generated by the seed spreader generator provided by Gan and Tao [18] which is widely used in related works. The generator can only produce integral data in range $[1, 10^5]$. It generates data in a random walk manner and can generate data with similar or varied density across clusters, which produces the **GTGen_Var, GTGen_Sim** datasets, respectively. For datasets generated with different dimensionality, the notation of its name further adds a postfix **_dx**, e.g., **GTGen_Var_d9** refers to the synthetic dataset with varied density and the dimensionality is 9.

6 real-world datasetsare used. **Audio** is a set of audio waveform data. **Bitcoin** contains address features on the heterogeneous Bitcoin network used for outlier detection to identify ransomware payments [40]. **Household** is a measurement of electric power consumption in one household [37]. **GasSensor** contains the acquired time series from 16 chemical sensors exposed to gas mixtures [39]. **PAMAP2** is a dataset for human physical activity monitoring collected by different monitors [38]. **Yahoomusic** is the latent factors of user preferences for musics on the yahoo music website. The information of number and dimension of the datasets are listed in Table **??**.

Since the *GanTao* method only receives integer input in range $[1, 10^5]$ and it indeed runs significantly slower on the synthetic datasets than *GAP* method, it is not compared with the other methods on the real-world datasets. Thus, unlike many other works where the real-world datasets are scaled and rounded to the integer domain $[1, 10^5]$, we do not do such manipulation to distort the original distribution of the datasets.

Environment setup. All the experiments ran on a machine with Intel Core i7-10700 CPU and 128GB RAM running Ubuntu 20.04 operating system. Except for *GanTao* which is provided in binary executable, all the other codes are written in C++ and compiled using g++ with -O3 optimization.

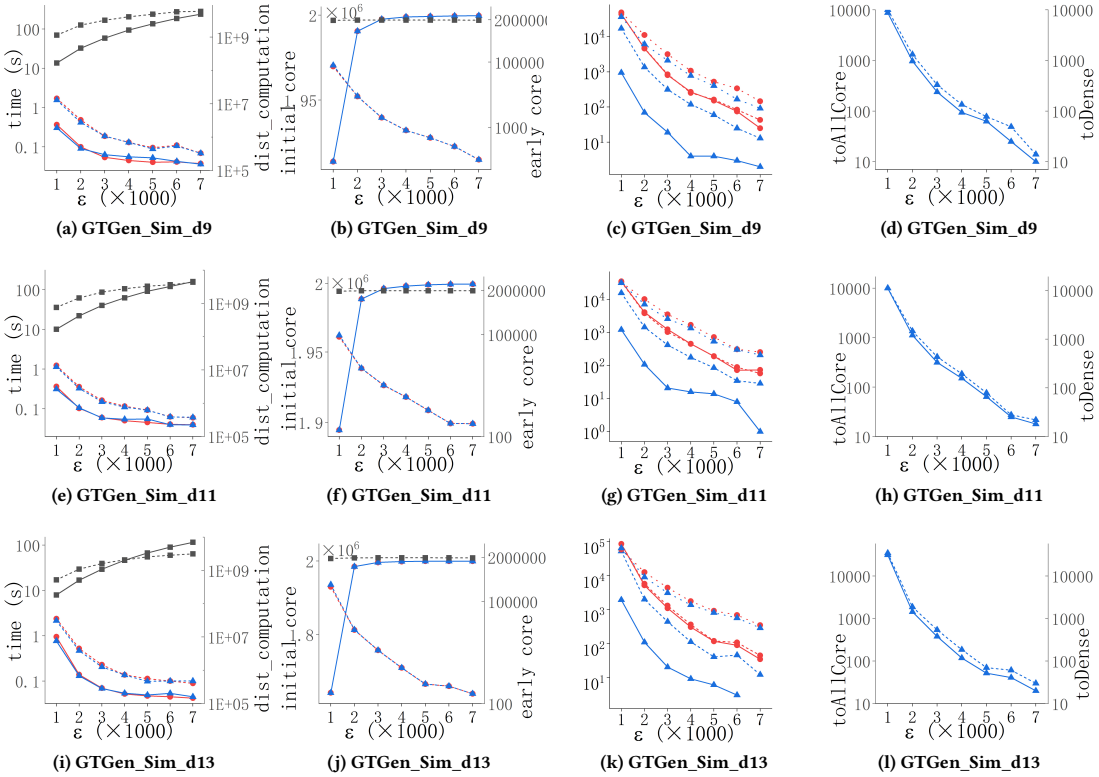
Parameter settings. The parameter settings adopted in this paper are aimed to make a clear comparison with the state-of-the-art method *GAP* [23]. For the synthetic datasets **GTGen_Var** and **GTGen_Sim**, exactly the same settings with the *GAP* paper [23] are adopted. Specifically, the $minpts$ parameter is varied from 20 to 70 with gap 10, and the default value is 50. The ϵ parameter is varied from 1000 to 7000 with gap 1000, and the default value varies for different dimensionality (see Figure 3 for the specific values). For the datasets containing different number of points, the one containing 2 million points is used by default. For the real-world datasets, since the authors of *GAP* method normalized the datasets to integer domain $[1, 10^5]$ but we use the original data, the parameters are set so that the *GAP* method exhibits similar performance with the experimental results given in their paper [23].

5.2 Self evaluation

The figures on Page 15, 16 and 17 shows the detailed running statistics of our proposed algorithms on the synthetic datasets. Black color curves belong to Basic, red color curves belong to Intercases, and blue color curves belong to Transitions.

The first column shows the running time and the number of distance computations. Left Y-axis and solid line: running time. Right Y-axis and dotted line: number of distance computations. (1) The *Basic* variant which does not utilize the information of core points and dense nodes, indeed runs orders of magnitudes slower than the other two variants. (2) It can be notice that the running time is proportional to the number of distance computations. (3) Varied density does influence the running time. (4) Although the *Transitions* variant always runs faster than the *InterCases* variant, the gap between the running time of them is usually small. Such phenomenon is supported by the theoretical complexity analysis. By comparing Theorem 4.8 and Theorem 4.10, the complexity difference of the two variants originates from the number of points that are identified as core points during the traversal but not by finding initial dense KDTree nodes. The analysis on the second column of pictures supports the theoretical analysis.

The second column shows the number of initially-identified and early-identified core points. Left Y-axis and solid line: number of core points identified by examining dense KDTree nodes. The Basic variant has no such operations and thus there is no black colored solid line. Right Y-axis and dotted line: number of core points identified before it is visited as query point. (1) It can be



noticed that for most parameter settings on the synthetic datasets, almost all points are core points and almost all core points can be initially identified. This explains why the running time is so low. (2) For Basic, most core points (more than 90%) can be early identified. (3) For the other two variants, as ϵ grows, the number of initially identified core points increases, and less points are left to be early identified and dotted lines decreases. Oppositely, the number of initially identified core points decreases as *minpts* grows, and the number of early identified core points increases. (4) Transitions always can early-identify more core points than Intercases, as blue dotted curves are always above red-dotted curve. (5) The curve for ϵ on varied-density datasets are not regular. This is because the increase of ϵ does not consistently increase the number of core points.

The third column shows the number of intermediate cases executed. Solid line: leaf-dense cases. Dashed line: dense-leaf cases. Dotted line: both-dense cases. (1) For the blue lines (Transitions algorithm), the solid lines are always the lowest, showing the number of leaf-dense cases are always the lowest. This owes to its ability to transit a leaf node to a dense node. But this is not always true for the red lines. Since the state of each leaf node for Intercases are initially determined, the number of intercases is then determined by the distribution of dense nodes across the KDTree. If leaf-dense intermediate cases are more, it means the dense nodes reside more on the right side of the KDTree.

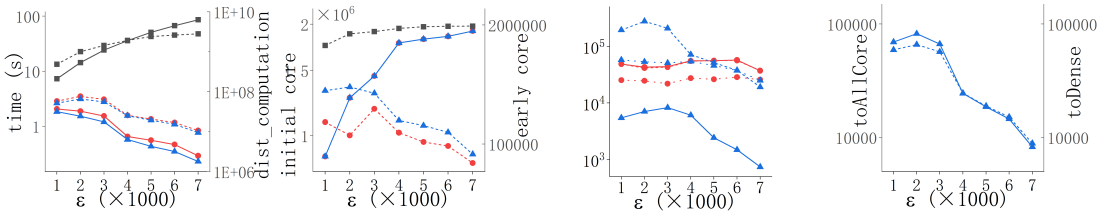
The fourth column shows the number of state transitions executed by the Transitions algorithm. They follows approximately the same trend as the running time curves.

5.3 Comparison

Explanation for missing points. First, as mentioned before, the *GanTao* method is not run on the real-world datasets since it only receives integral value in range $[1, 10^5]$. Second, the *mlpack* method runs out of memory (> 128GB) under many parameter settings, resulting in many missing points in its curves. Third, the *GriT* method reports segmentation fault on *Bitcoin* dataset when $\epsilon < 60000$ due to unknown bug in their customized codes for KDTree construction, resulting in a partial curve for *GriT* in Figure 4c. Also, the curve for *GriT* completely vanishes in Figure 4i since ϵ is set to 20000.

Overall comparison. In an overview, *SJ-DBSCAN* is the slowest due to $O(n^2)$ time complexity. The second slowest group includes *Basic*, *GanTao*, *GriT* and *mlpack*, all of which exhibit comparable running time in order of magnitude. Their subpar performance can be attribute to the failure to employee core point information. While the *GAP* method is a good competitor with our *InterCases* and *Transitions* variants, the *Transitions* variant performs the best in most cases.

There is only a few exceptions where our *Transition* algorithm shows inferior performance than *GAP* method, that are on the **GTGen_Var** datasets and on **GasSensor** dataset when ϵ is small (see the left part of Figure 3b, 4d, 4e and 4f) or *minpts* is large (see the right part of Figure 3f, 4j and 4k). The reason is that the advantage of our method lies in the utilization of core points, and the theoretical complexity decrease as number of core points increases. When ϵ is small or *minpts* is large, there will be less core points, and the theoretical complexity and also experimental running time would increase. In comparison, the *GAP* method employs geometric

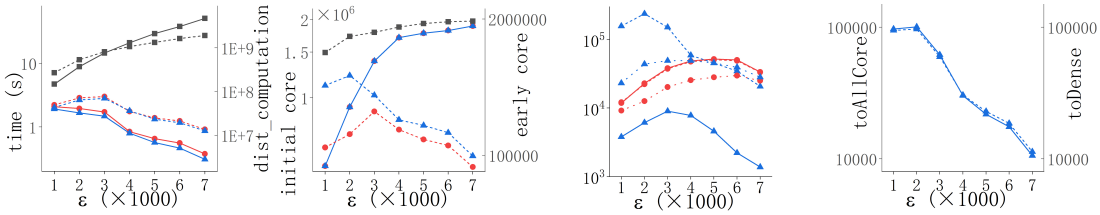


(a) GTGen_Vary_d9

(b) GTGen_Vary_d9

(c) GTGen_Vary_d9

(d) GTGen_Vary_d9

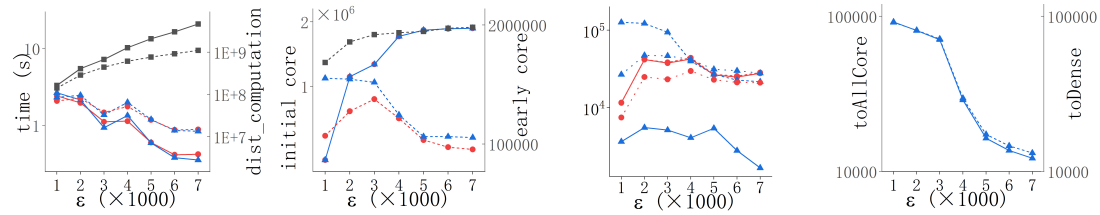


(e) GTGen_Vary_d11

(f) GTGen_Vary_d11

(g) GTGen_Vary_d11

(h) GTGen_Vary_d11

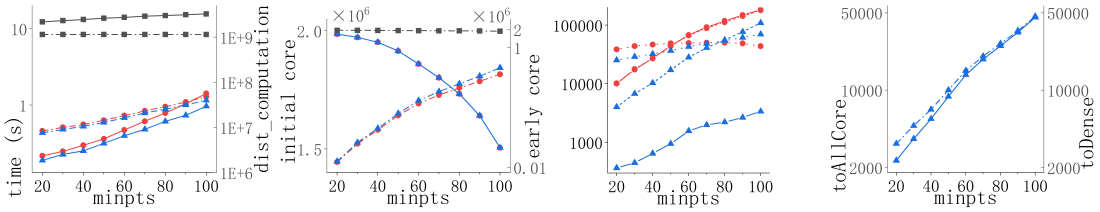


(i) GTGen_Vary_d13

(j) GTGen_Vary_d13

(k) GTGen_Vary_d13

(l) GTGen_Vary_d13

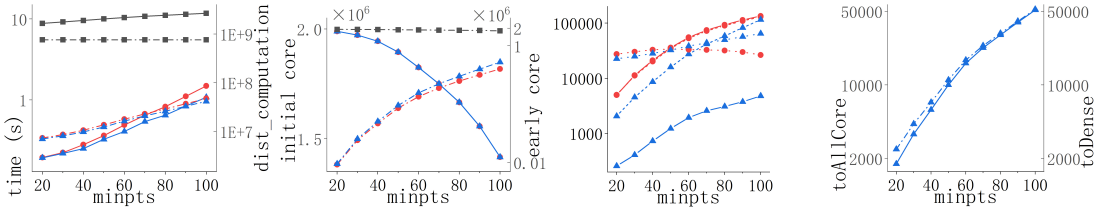


(a) GTGen_Sim_d9

(b) GTGen_Sim_d9

(c) GTGen_Sim_d9

(d) GTGen_Sim_d9

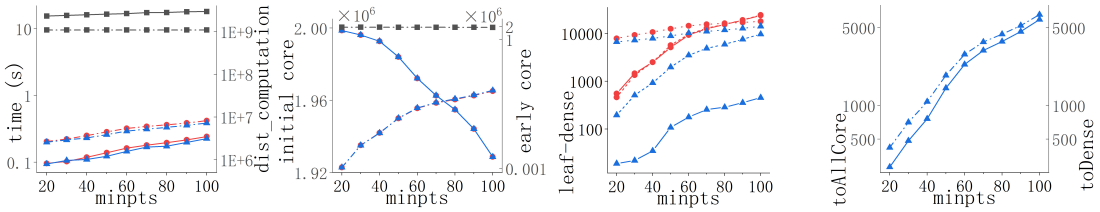


(e) GTGen_Sim_d11

(f) GTGen_Sim_d11

(g) GTGen_Sim_d11

(h) GTGen_Sim_d11

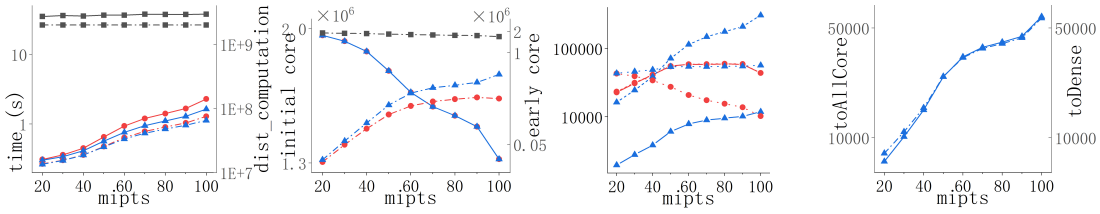


(i) GTGen_Sim_d13

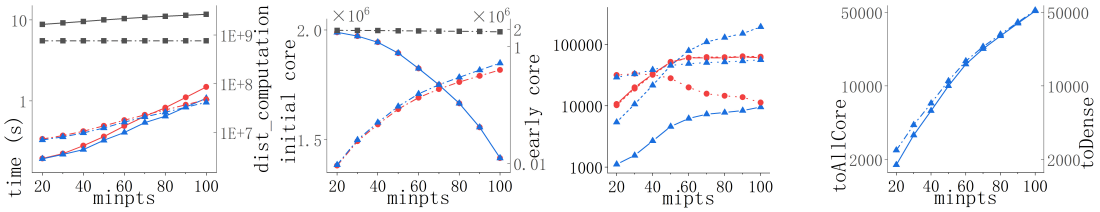
(j) GTGen_Sim_d13

(k) GTGen_Sim_d13

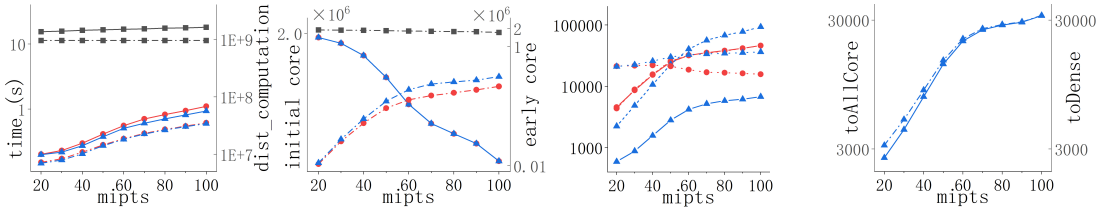
(l) GTGen_Sim_d13



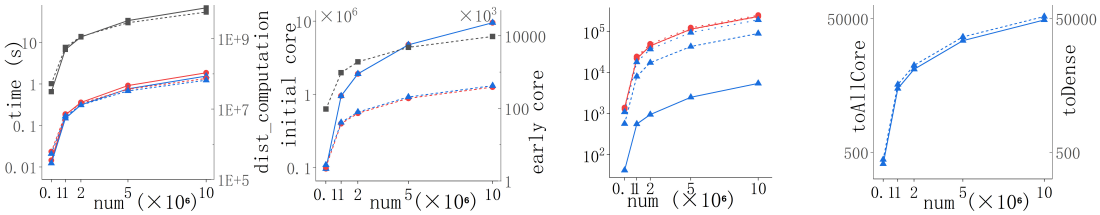
(a) GTGen_Vary_d9 (b) GTGen_Vary_d9 (c) GTGen_Vary_d9 (d) GTGen_Vary_d9



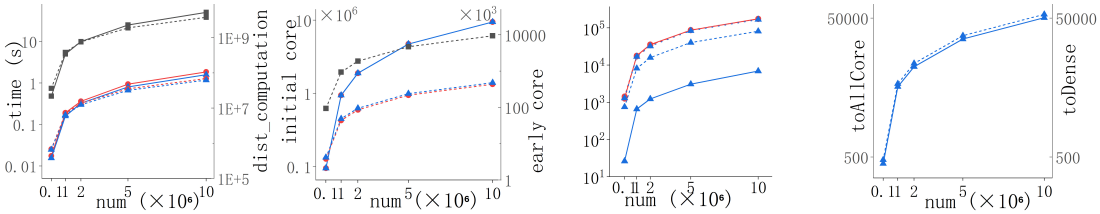
(e) GTGen_Vary_d11 (f) GTGen_Vary_d11 (g) GTGen_Vary_d11 (h) GTGen_Vary_d11



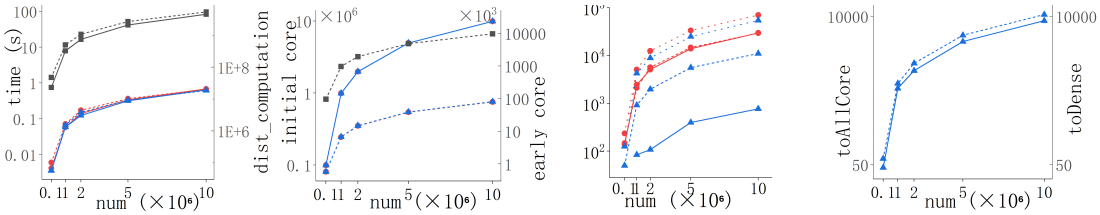
(i) GTGen_Vary_d13 (j) GTGen_Vary_d13 (k) GTGen_Vary_d13 (l) GTGen_Vary_d13



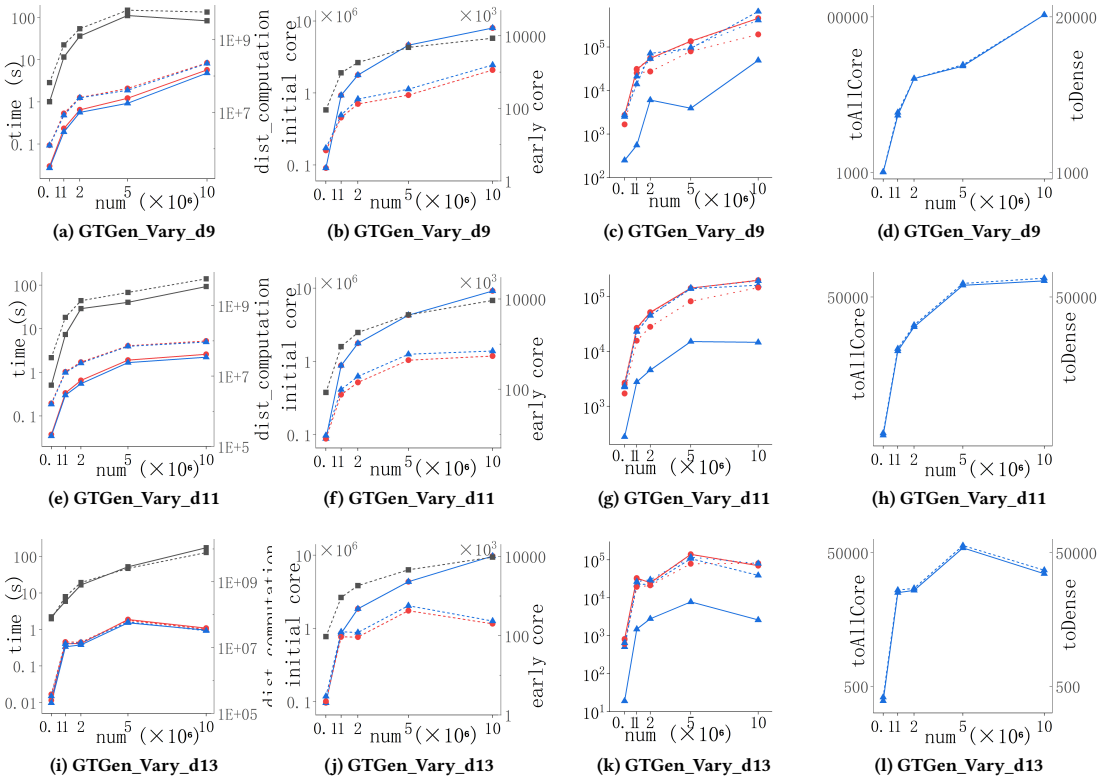
(a) GTGen_Sim_d9 (b) GTGen_Sim_d9 (c) GTGen_Sim_d9 (d) GTGen_Sim_d9



(e) GTGen_Sim_d11 (f) GTGen_Sim_d11 (g) GTGen_Sim_d11 (h) GTGen_Sim_d11



(i) GTGen_Sim_d13 (j) GTGen_Sim_d13 (k) GTGen_Sim_d13 (l) GTGen_Sim_d13



(a) GTGen_Vary_d9 (b) GTGen_Vary_d9 (c) GTGen_Vary_d9 (d) GTGen_Vary_d9

(e) GTGen_Vary_d11 (f) GTGen_Vary_d11 (g) GTGen_Vary_d11 (h) GTGen_Vary_d11

(i) GTGen_Vary_d13 (j) GTGen_Vary_d13 (k) GTGen_Vary_d13 (l) GTGen_Vary_d13

methods to prune points more efficiently, which works also well when there are less core points.

However, there are two arguments. First, when there are less core points, it is usually the case that the ϵ and *mipnts* parameter are inappropriately set to produce meaningful clusters. Second, the *generalized dual-tree* algorithm framework is an orthogonal contribution with how the intermediate cases are implemented. Actually, the traversal algorithms for the intermediate cases solving the ϵ-connect dual, ϵ-connect-single and ϵ-minpts-in-dense problems, are the most basic implementation that only uses the pruning function based on MBR of the KDTree nodes, and the pruning function of *GAP* method can also be integrated into our algorithm framework for further efficiency improvement.

Influence of number. By a vertical comparison in each figure of the bottom row in Figure 3, it can be noticed that the increasing trend of our methods is the same with the state-of-the-art *GAP* method. But the performance is also affected by the dimensionality and the varied density phenomenon. See the analysis below.

Influence of dimensionality. By a horizontal comparison between the columns of Figure 3, it can be noticed that the advantage of our methods enlarges as the dimensionality grows. It indicates that our methods is more suitable in higher dimensions. This claim consistently holds for high-dimensional data such as **Audio** and **Yahoomusic**.

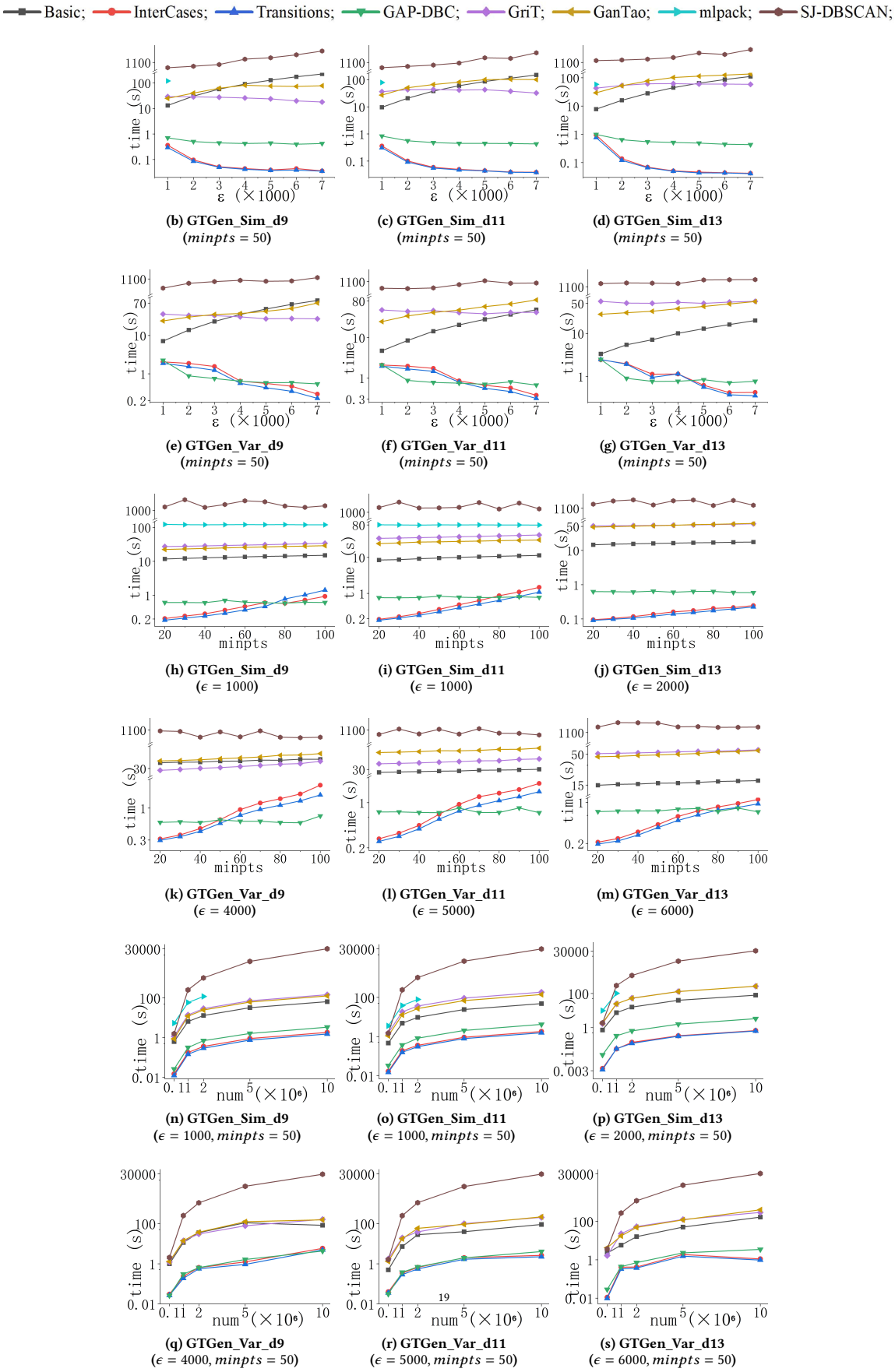
Influnce of ϵ and minpts. Observing the first row in Figure 3, the running time of the *Basic* variant increases as ϵ gets larger, since less nodes are pruned and more nodes combinations are visited when ϵ is large. For the other two variants, varying these two parameters indirectly influence the number of core points which in turn affects the performance. It can be noticed that the running time of *InterCases* and *Transitions* variant typically drop as ϵ gets larger but increases as *minpts* gets larger. Exceptions exist on high-dimensional data sets such as **Audio** and **Yahoomusic**, and also on **Household** dataset where the varied density phenonenon is dominant.

High-dimensional data. The running time of all methods on **Audio** and **Yahoomusic** first increases with ϵ, then reaches maximum at specific value of ϵ which maximizes the number of initial dense nodes and incurs maximized cost for joining clusters. The running time decreases as ϵ further increases, since the clusters expand and merge with each other, reducing the number of initial dense nodes. Such behavior owes to the nearly uniform distribution of high-dimensional data points, so that the increase of ϵ consistently expands the clusters. In contrast, the varied density phenomenon predominates in lower dimensional space, resulting in totally different behaviors.

Varied density. In Figure 4k, the strongly varied density of *Household* dataset leads to unstable running time variation as ϵ increases. However, the performance of our methods are not much affected by varied density, which is another advantage of our methods.

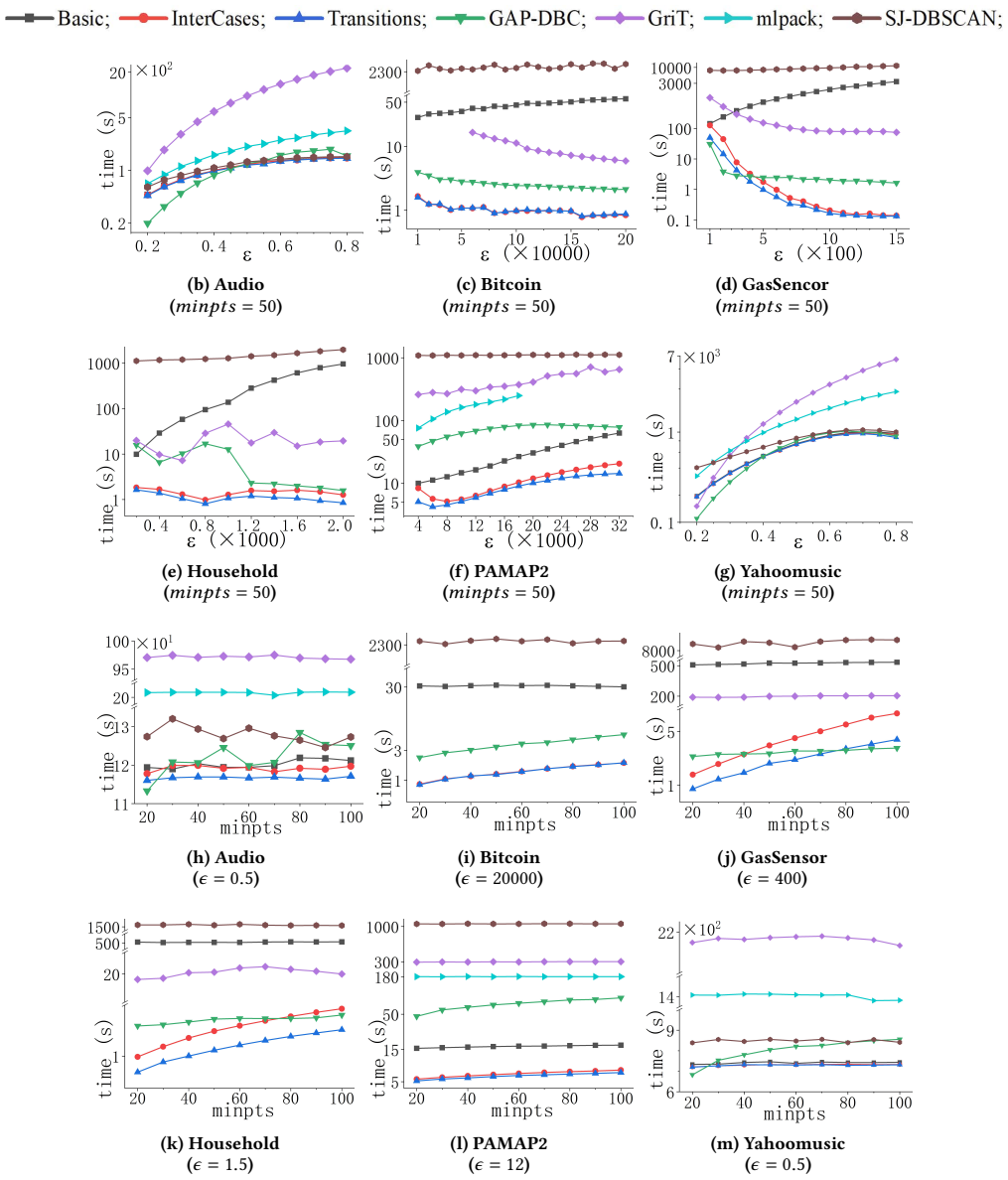
6 CONCLUSION

This paper proposes the GeDual-DBSCAN algorithms, which bring not only conceptual novelty but also efficiency improvement against existing DBSCAN algorithms. The novel idea of *intermediate cases* elegantly integrates the DBSCAN sub-problems newly defined in



(b) GTGen_Sim_d9 (*minpts* = 50)
(c) GTGen_Sim_d11 (*minpts* = 50)
(d) GTGen_Sim_d13 (*minpts* = 50)
(e) GTGen_Var_d9 (*minpts* = 50)
(f) GTGen_Var_d11 (*minpts* = 50)
(g) GTGen_Var_d13 (*minpts* = 50)
(h) GTGen_Sim_d9 (ϵ = 1000)
(i) GTGen_Sim_d11 (ϵ = 1000)
(j) GTGen_Sim_d13 (ϵ = 2000)
(k) GTGen_Var_d9 (ϵ = 4000)
(l) GTGen_Var_d11 (ϵ = 5000)
(m) GTGen_Var_d13 (ϵ = 6000)
(n) GTGen_Sim_d9 (ϵ = 1000, *minpts* = 50)
(o) GTGen_Sim_d11 (ϵ = 1000, *minpts* = 50)
(p) GTGen_Sim_d13 (ϵ = 2000, *minpts* = 50)
(q) GTGen_Var_d9 (ϵ = 4000, *minpts* = 50)
(r) GTGen_Var_d11 (ϵ = 5000, *minpts* = 50)
(s) GTGen_Var_d13 (ϵ = 6000, *minpts* = 50)

Figure 3: Running time results on synthetic datasets GTGen. Top 2 rows: varying ϵ. Middle 2 rows: varying *minpts*. Bottom 2 rows: varying *num*.



(b) Audio (*minpts* = 50) (c) Bitcoin (*minpts* = 50) (d) GasSencor (*minpts* = 50)

(e) Household (*minpts* = 50) (f) PAMAP2 (*minpts* = 50) (g) Yahoomusic (*minpts* = 50)

(h) Audio (ϵ = 0.5) (i) Bitcoin (ϵ = 20000) (j) GasSensor (ϵ = 400)

(k) Household (ϵ = 1.5) (l) PAMAP2 (ϵ = 12) (m) Yahoomusic (ϵ = 0.5)

Figure 4: Running time results on real-world datasets. Upper 2 rows: varying ϵ. Lower 2 rows: varying *minpts*.

this paper into the dual-tree algorithm framework. The power of avoiding unnecessary distance computations possessed by these intermediate cases are fully employed by the new idea of *state transitions* for tree nodes during the tree traversal. The resulted GeDual-DBSCAN-Transitions algorithm performs better than the state-of-the-art DBSCAN algorithm under almost all parameter settings on all the datasets tested. It must be noted that the generalized dual-tree algorithm framework proposed in this paper is orthogonal with the geometric approaches of other works for improving algorithmic efficiency. More efficient algorithms can be obtained by using more efficient implementations for the DBSCAN sub-problems newly defined in this paper.

ACKNOWLEDGMENTS

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

REFERENCES

[1] Mihael Ankerst, Markus M Breunig, Hans-Peter Kriegel, and Jörg Sander. 1999. OPTICS: Ordering points to identify the clustering structure. *ACM Sigmod record* 28, 2 (1999), 49–60.

[2] David Arthur and Sergei Vassilvitskii. 2006. *k-means++: The advantages of careful seeding*. Technical Report. Stanford.

[3] Arthur Asuncion and David Newman. 2007. UCI machine learning repository. http://archive.ics.uci.edu

[4] Jon Louis Bentley. 1975. Multidimensional binary search trees used for associative searching. *Commun. ACM* 18, 9 (1975), 509–517.

[5] Christian Böhm, Bernhard Braunmüller, Markus Breunig, and Hans-Peter Kriegel. 2000. High performance clustering based on the similarity join. In *Proceedings of the ninth international conference on Information and knowledge management*. 298–305.

[6] Ricardo JGB Campello, Davoud Moulavi, and Jörg Sander. 2013. Density-based clustering based on hierarchical density estimates. In *Pacific-Asia conference on knowledge discovery and data mining*. Springer, 160–172.

[7] Yewang Chen, Lida Zhou, Nizar Bouguila, Cheng Wang, Yi Chen, and Jixiang Du. 2021. BLOCK-DBSCAN: Fast clustering for large scale data. *Pattern Recognit.* 109 (2021), 107624. https://doi.org/10.1016/j.patcog.2020.107624

[8] Yewang Chen, Lida Zhou, Songwen Pei, Zhiwen Yu, Yi Chen, Xin Liu, Jixiang Du, and Naixue Xiong. 2021. KNN-BLOCK DBSCAN: Fast Clustering for Large-Scale Data. *IEEE Trans. Syst. Man Cybern. Syst.* 51, 6 (2021), 3939–3953. https://doi.org/10.1109/TSMC.2019.2956527

[9] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2022. *Introduction to algorithms*. MIT press.

[10] Ryan R. Curtin, Marcus Edel, Omar Shrit, Shubham Agrawal, Suryoday Basak, James Joseph Balamuta, Ryan Birmingham, Kartik Dutt, Dirk Eddelbuettel, Rishabh Garg, Shikhar Jaiswal, Aakash Kaushik, Sangyeon Kim, Anjishnu Mukherjee, Nanubala Gnana Sai, Nippun Sharma, Yashwant Singh Parihar, Roshan Swain, and Conrad Sanderson. 2023. mlpack 4: a fast, header-only C++ machine learning library. *J. Open Source Softw.* 8, 82 (2023), 5026. https://doi.org/10.21105/joss.05026

[11] Ryan R. Curtin, William B. March, Parikshit Ram, David V. Anderson, Alexander G. Gray, and Charles L. Isbell Jr. 2013. Tree-Independent Dual-Tree Algorithms. In *Proceedings of the 30th International Conference on Machine Learning, ICML 2013, Atlanta, GA, USA, 16-21 June 2013 (JMLR Workshop and Conference Proceedings)*, Vol. 28. JMLR.org, 1435–1443. http://proceedings.mlr.press/v28/curtin13.html

[12] Igor de Moura Ventorim, Diego Luchi, Alexandre Loureiros Rodrigues, and Flávio Miguel Varejão. 2021. BIRCHSCAN: A sampling method for applying DBSCAN to large datasets. *Expert Syst. Appl.* 184 (2021), 115518. https://doi.org/10.1016/J.ESWA.2021.115518

[13] Dingsheng Deng. 2020. Application of DBSCAN algorithm in data sampling. In *Journal of Physics: Conference Series*, Vol. 1617. IOP Publishing, 012088.

[14] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. 1996. 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*, Evangelos Simoudis, Jiawei Han, and Usama M. Fayyad (Eds.). AAAI Press, 226–231. http://www.aaai.org/Library/KDD/1996/kdd96-037.php

[15] Sergej Fries, Brigitte Boden, Grzegorz Stepien, and Thomas Seidl. 2014. Phidj: Parallel similarity self-join for high-dimensional vector data with mapreduce. In *2014 IEEE 30th International Conference on Data Engineering*. IEEE, 796–807.

[16] Junhao Gan and Yufei Tao. [n.d.]. The codes for GanTaoV2. https://sites.google.com/view/approxdbscan

[17] Junhao Gan and Yufei Tao. 2015. 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*, Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives (Eds.). ACM, 519–530. https://doi.org/10.1145/2723372.2737792

[18] Junhao Gan and Yufei Tao. 2017. On the hardness and approximation of Euclidean DBSCAN. *ACM Transactions on Database Systems (TODS)* 42, 3 (2017), 1–45.

[19] Nahid Gholizadeh, Hamid Saadatfar, and Nooshin Hanafi. 2021. K-DBSCAN: An improved DBSCAN algorithm for big data. *J. Supercomput.* 77, 6 (2021), 6214–6235. https://doi.org/10.1007/s11227-020-03524-3

[20] Nahid Gholizadeh, Hamid Saadatfar, and Nooshin Hanafi. 2021. K-DBSCAN: An improved DBSCAN algorithm for big data. *The Journal of supercomputing* 77, 6 (2021), 6214–6235.

[21] Alexander Gray and Andrew Moore. 2000. N-body'problems in statistical learning. *Advances in neural information processing systems* 13 (2000).

[22] Ade Gunawan and M de Berg. 2013. A faster algorithm for DBSCAN. *Master's thesis* (2013).

[23] Xiaogang Huang and Tiefeng Ma. 2023. Fast Density-Based Clustering: Geometric Approach. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–24.

[24] Xiaogang Huang, Tiefeng Ma, Conan Liu, and Shuangzhe Liu. 2023. GriT-DBSCAN: A spatial clustering algorithm for very large databases. *Pattern Recognition* (2023), 109658.

[25] Heinrich Jiang, Jennifer Jang, and Jakub Lacki. 2020. Faster DBSCAN via subsampled similarity queries. *Advances in Neural Information Processing Systems* 33 (2020), 22407–22419.

[26] Han Jiawei and Kamber Micheline. 2006. *Data mining: concepts and techniques*. Morgan kaufmann.

[27] Amin Karami and Ronnie Johansson. 2014. Choosing DBSCAN parameters automatically using differential evolution. *International Journal of Computer Applications* 91, 7 (2014), 1–11.

[28] Sonal Kumari, Poonam Goyal, Ankit Sood, Dhruv Kumar, Sundar Balasubramaniam, and Navneet Goyal. 2017. 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, 14. http://dl.acm.org/citation.cfm?id=3007773

[29] Der-Tsai Lee and Chak-Kuen Wong. 1977. Worst-case analysis for region and partial region searches in multidimensional binary search trees and balanced quad trees. *Acta Informatica* 9, 1 (1977), 23–29.

[30] Shan-Shan Li. 2020. An Improved DBSCAN Algorithm Based on the Neighbor Similarity and Fast Nearest Neighbor Query. *IEEE Access* 8 (2020), 47468–47476. https://doi.org/10.1109/ACCESS.2020.2972034

[31] Yu A Malkov and Dmitry A Yashunin. 2018. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE transactions on pattern analysis and machine intelligence* 42, 4 (2018), 824–836.

[32] William B March, Parikshit Ram, and Alexander G Gray. 2010. Fast euclidean minimum spanning tree: algorithm, analysis, and applications. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*. 603–612.

[33] Leland McInnes, John Healy, Steve Astels, et al. 2017. hdbscan: Hierarchical density based clustering. *J. Open Source Softw.* 2, 11 (2017), 205.

[34] Negar Ohadi, Ali Kamandi, Mahmood Shabankhah, Seyed Mohsen Fatemi, Seyed Mohsen Hosseini, and Alireza Mahmoudi. 2020. Sw-dbscan: A grid-based dbscan algorithm for large datasets. In *2020 6th International conference on web research (ICWR)*. IEEE, 139–145.

[35] Md. Mostofa Ali Patwary, Diana Palsetia, Ankit Agrawal, Wei-keng Liao, Fredrik Manne, and Alok N. Choudhary. 2012. 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*, Jeffrey K. Hollingsworth (Ed.). IEEE/ACM, 62. https://doi.org/10.1109/SC.2012.9

[36] Md Mostofa Ali Patwary, Diana Palsetia, Ankit Agrawal, Wei-keng Liao, Fredrik Manne, and Alok Choudhary. 2012. A new scalable parallel DBSCAN algorithm using the disjoint-set data structure. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–11.

[37] UCI Repository. 2012. Household. Avaible at http://archive.ics.uci.edu/ml/datasets/Individual+household+electric+power+consumption.

[38] UCI Repository. 2012. PAMAP2. Avaible at http://archive.ics.uci.edu/dataset/231/pamap2+physical+activity+monitoring.

[39] UCI Repository. 2015. GasSensor. Avaible at http://archive.ics.uci.edu/dataset/322/gas+sensor+array+under+dynamic+gas+mixtures.

[40] UCI Repository. 2020. Bitcoin. Avaible at http://archive.ics.uci.edu/ml/datasets/BitcoinHeistRansomwareAddressDataset.

[41] Aditya Sarma, Poonam Goyal, Sonal Kumari, Anand Wani, Jagat Sesh Challa, Saiyedul Islam, and Navneet Goyal. 2019. 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, 1–11. https://doi.org/10.1109/CLUSTER.2019.8891020

[42] Erich Schubert, Jörg Sander, Martin Ester, Hans Peter Kriegel, and Xiaowei Xu. 2017. DBSCAN revisited, revisited: why and how you should (still) use DBSCAN. *ACM Transactions on Database Systems (TODS)* 42, 3 (2017), 1–21.

[43] Kevin Sheridan, Tejas G Puranik, Eugene Mangortey, Olivia J Pinon-Fischer, Michelle Kirby, and Dimitri N Mavris. 2020. An application of dbscan clustering for flight anomaly detection during the approach phase. In *AIAA Scitech 2020 Forum*. 1851.

[44] Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. 2016. *Introduction to data mining*. Pearson Education India.

[45] Robert E Tarjan and Jan Van Leeuwen. 1984. Worst-case analysis of set union algorithms. *Journal of the ACM (JACM)* 31, 2 (1984), 245–281.

[46] Pravin M. Vaidya. 1989. AnO(n logn) algorithm for the all-nearest-neighbors Problem. *Discrete & Computational Geometry* 4, 2 (mar 1989), 101–115. https://doi.org/10.1007/BF02187718

[47] Yiqiu Wang, Yan Gu, and Julian Shun. 2020. 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*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 2555–2571. https://doi.org/10.1145/3318464.3380582

[48] Yiqiu Wang, Yan Gu, and Julian Shun. 2020. Theoretically-efficient and practical parallel DBSCAN. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2555–2571.

[49] Shaoyuan Weng, Jin Gou, and Zongwen Fan. 2021. h-DBSCAN: A simple fast DBSCAN algorithm for big data. In *Asian Conference on Machine Learning*. PMLR, 81–96.