On the Hardness and Approximation of Euclidean DBSCAN

Junhao Gan, University of Queensland
Yufei Tao, Chinese University of Hong Kong

DBSCAN is a method proposed in 1996 for clustering multi-dimensional points, and has received extensive applications. Its computational hardness is still unsolved to this date. The original KDD'96 paper claimed an algorithm of $O(n \log n)$ "average run time complexity" (where n is the number of data points) without a rigorous proof. In 2013, a genuine $O(n \log n)$-time algorithm was found in 2D space under Euclidean distance. The hardness of dimensionality $d \geq 3$ has remained open ever since.

This article considers the problem of computing DBSCAN clusters from scratch (assuming no existing indexes) under Euclidean distance. We prove that, for $d \geq 3$, the problem requires $\Omega(n^{4/3})$ time to solve, unless very significant breakthroughs—ones widely believed to be impossible—could be made in theoretical computer science. Motivated by this, we propose a relaxed version of the problem called *ρ-approximate DBSCAN*, which returns the same clusters as DBSCAN, unless the clusters are "unstable" (i.e., they change once the input parameters are slightly perturbed). The ρ-approximate problem can be settled in $O(n)$ expected time regardless of the constant dimensionality d.

The article also enhances the previous result on the exact DBSCAN problem in 2D space. We show that, if the n data points have been pre-sorted on each dimension (i.e., one sorted list per dimension), the problem can be settled in $O(n)$ worst-case time. As a corollary, when all the coordinates are integers, the 2D DBSCAN problem can be solved in $O(n \log \log n)$ time deterministically, improving the existing $O(n \log n)$ bound.

Categories and Subject Descriptors: H3.3 [**Information search and retrieval**]: Clustering

General Terms: Algorithms, Theory, Performance

Additional Key Words and Phrases: DBSCAN, Density-Based Clustering, Hopcroft Hard, Algorithms, Computational Geometry

1. INTRODUCTION

Density-based clustering is one of the most fundamental topics in data mining. Given a set P of n points in d-dimensional space $\mathbb{R}^d$, the objective is to group the points of P into subsets—called *clusters*—such that any two clusters are separated by "sparse regions". Figure 1 shows two classic examples taken from [Ester et al. 1996]: the left one contains 4 snake-shaped clusters, while the right one contains 3 clusters together with some noise. The main advantage of density-based clustering (over methods such as *k-means*) is its capability of discovering clusters with arbitrary shapes (while k-means typically returns ball-like clusters).

Density-based clustering can be achieved using a variety of approaches, which differ mainly in their (i) definitions of "dense/sparse regions", and (ii) criteria of how dense regions should be connected to form clusters. In this article, we concentrate on DBSCAN, which is an approach invented by [Ester et al. 1996], and received the test-of-time award in KDD'14. DBSCAN characterizes "density/sparsity" by resorting to two parameters:

— ϵ: a positive real value;
— $MinPts$: a small positive constant integer.

Let $B(p, \epsilon)$ be the d-dimensional ball centered at point p with radius ϵ, where the distance metric is Euclidean distance. $B(p, \epsilon)$ is "dense" if it covers at least $MinPts$ points of P.

DBSCAN forms clusters based on the following rationale. *If $B(p, \epsilon)$ is dense, all the points in $B(p, \epsilon)$ should be added to the same cluster as p.* This creates a "chained effect": whenever a new point p' with a dense $B(p', \epsilon)$ is added to the cluster of p, all the

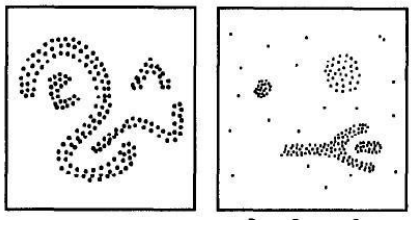


Fig. 1. Examples of density-based clustering from [Ester et al. 1996]

points in $B(p', \epsilon)$ should also join the same cluster. The cluster of p continues to grow in this manner to the effect's fullest extent.

1.1. Previous Description of DBSCAN's Running Time

The original DBSCAN algorithm of [Ester et al. 1996] performs a *region query* for each point $p \in P$, which retrieves $B(p, \epsilon)$. Regarding the running time, [Ester et al. 1996] wrote:

> *"The height an R*-tree is $O(\log n)$ for a database of n points in the worst case and a query with a "small" query region has to traverse only a limited number of paths in the R*-tree. Since the Eps-Neighborhoods are expected to be small compared to the size of the whole data space, the average run time complexity of a single region query is $O(\log n)$. For each of the points of the database, we have at most one region query. Thus, the average run time complexity of DBSCAN is $O(n \log n)$."*

The underlined statement lacks scientific rigor:

— Consider a dataset where $\Omega(n)$ points coincide at the same location. No matter how small is ϵ, for every such point p, $B(p, \epsilon)$ always covers $\Omega(n)$ points. Even just reporting the points inside $B(p, \epsilon)$ for all such p already takes $\Theta(n^2)$ time—this is true regardless of how good is the underlying R*-tree or any other index deployed.

— The notion of "average run time complexity" in the statement does not seem to follow any of the standard definitions in computer science (see, for example, Wikipedia[1]). There was no clarification on the mathematical meaning of this notion in [Ester et al. 1996], and neither was there a proof on the claimed complexity. In fact, it would have been a great result if an $O(n \log n)$ bound could indeed be proved under any of those definitions.

The "$O(n \log n)$ average run time complexity" has often been re-stated with fuzzy or even no description of the accompanying conditions. A popular textbook [Han et al. 2012], for example, comments in Chapter 10.4.1:

> *If a spatial index is used, the computational complexity of DBSCAN is $O(n \log n)$, where n is the number of database objects. Otherwise, the complexity is $O(n^2)$.*

Similar statements have appeared in many papers: [Böhm et al. 2004] (Sec 3.1), [Chaoji et al. 2008] (Sec 2), [Ester 2013] (Chapter 5, Sec 2), [Klusch et al. 2003] (Sec 2), [Lu et al. 2011] (Sec 5.4), [Milenova and Campos 2002] (Sec 1), [Patwary et al. 2012] (Sec 2), [Sheikholeslami et al. 2000] (Sec 3.3), [Wang et al. 1997] (Sec 2.2.3), [Wen et al. 2002] (Sec 5.2), mentioning just 10 papers. Several works have even utilized the

[1] *Https://en.wikipedia.org/wiki/Average-case_complexity*

$O(n \log n)$ bound as a building-brick lemma to derive new "results" incorrectly: see Sec D.1 of [Li et al. 2010], Sec 3.2 of [Pei et al. 2006], and Sec 5.2 of [Roy and Bhattacharyya 2005]).

[Gunawan 2013] also showed that all of the subsequently improved versions of the original DBSCAN algorithm either do not compute the precise DBSCAN result (e.g., see [Borah and Bhattacharyya 2004; Liu 2006; Tsai and Wu 2009]), or still suffer from $O(n^2)$ running time [Mahran and Mahar 2008]. As a partial remedy, he developed a new 2D algorithm which truly runs in $O(n \log n)$ time, without assuming any indexes.

1.2. Our Contributions

This article was motivated by two questions:

— For $d \geq 3$, is it possible to design an algorithm that genuinely has $O(n \log n)$ time complexity? To make things easier, is it possible to achieve time complexity $O(n \log^c n)$ even for some *very large* constant c?
— If the answer to the previous question is no, is it possible to achieve linear or near-linear running time by sacrificing the quality of clusters slightly, while still being able to give a strong guarantee on the quality?

We answer the above questions with the following contributions:

(1) We prove that the DBSCAN problem (computing the clusters from scratch, without assuming an existing index) requires $\Omega(n^{4/3})$ time to solve in $d \geq 3$, unless very significant breakthroughs—ones widely believed to be impossible—can be made in theoretical computer science. Note that $n^{4/3}$ is arbitrarily larger than $n \log^c n$, regardless of constant c.

(2) We introduce a new concept called *ρ-approximate DBSCAN* which comes with strong assurances in both *quality* and *efficiency*. For quality, its clustering result is guaranteed to be "sandwiched" between the results of DBSCAN obtained with parameters $(\epsilon, MinPts)$ and $(\epsilon(1+\rho), MinPts)$, respectively. For efficiency, we prove that ρ-approximate DBSCAN can be solved in linear $O(n)$ expected time, for any ϵ, arbitrarily small constant ρ, and in any fixed dimensionality d.

(3) We give a new algorithm that solves the exact DBSCAN problem in 2D space using $O(n \log n)$ time, but in a way substantially simpler than the solution of [Gunawan 2013]. The algorithm reveals an inherent geometric connection between (exact) DBSCAN and *Delaunay graphs*. The connection is of independent interests.

(4) We prove that the 2D exact DBSCAN problem can actually be settled in $O(n)$ time, provided that the n data points have been sorted along each dimension. In other words, the "hardest" component of the problem turns out to be sorting the coordinates, whereas the clustering part is easy. Immediately, this implies that 2D DBSCAN can be settled in $o(n \log n)$ time when the coordinates are integers, by utilizing fast integer-sorting algorithms [Andersson et al. 1998; Han and Thorup 2002]: (i) deterministically, we achieve $O(n \log \log n)$ time—improving the $O(n \log n)$ bound of [Gunawan 2013]; (ii) randomly, we achieve $O(n\sqrt{\log \log n})$ time in expectation.

(5) We perform an extensive experimental evaluation to explore the situations in which the original DBSCAN is adequate, and the situations in which ρ-approximate DBSCAN serves as a nice alternative. In a nutshell, when the input data is "realistic" and it suffices to play with small ϵ, existing algorithms may be used to find precise DBSCAN clusters efficiently. However, their cost escalates rapidly with ϵ. The proposed ρ-approximate version is fast for a much wider parameter range. The performance advantage of ρ-approximate DBSCAN is

most significant when the clusters have varying densities. In that case, a suitable ϵ is decided by the sparsest cluster, and has to be large with respect to the densest cluster, thus causing region queries in that cluster to be expensive (this will be further explained in Section 6).

A short version of this article appeared in SIGMOD'15 [Gan and Tao 2015]. In terms of technical contents, the current article extends that preliminary work with Contributions 3 and 4. Furthermore, the article also features revamped experiments that carry out a more complete study of the behavior of various algorithms.

1.3. Organization of the Article

Section 2 reviews the previous work related to ours. Section 3 provides theoretical evidence on the computational hardness of DBSCAN, and presents a sub-quadratic algorithm for solving the problem exactly. Section 4 proposes ρ-approximate DBSCAN, elaborates on our algorithm, and establishes its quality and efficiency guarantees. Section 5 presents new algorithms for solving the exact DBSCAN problem in 2D space. Section 6 discusses several issues related to the practical performance of different algorithms and implementations. Section 7 evaluates all the exact and approximation algorithms with extensive experimentation. Finally, Section 8 concludes the article with a summary of findings.

2. RELATED WORK

Section 2.1 reviews the DBSCAN definitions as set out by [Ester et al. 1996]. Section 2.2 describes the 2D algorithm in [Gunawan 2013] that solves the problem genuinely in $O(n \log n)$ time. Section 2.3 points out several results from computational geometry which will be needed to prove the intractability of DBSCAN later.

2.1. Definitions

As before, let P be a set of n points in d-dimensional space $\mathbb{R}^d$. Given two points $p, q \in \mathbb{R}^d$, we denote by $dist(p, q)$ the Euclidean distance between p and q. Denote by $B(p, r)$ the ball centered at a point $p \in \mathbb{R}^d$ with radius r. Remember that DBSCAN takes two parameters: ϵ and $MinPts$.

Definition 2.1. A point $p \in P$ is a **core point** if $B(p, \epsilon)$ covers at least $MinPts$ points of P (including p itself).

If p is not a core point, it is said to be a *non-core point*. To illustrate, suppose that P is the set of points in Figure 2, where $MinPts = 4$ and the two circles have radius ϵ. Core points are shown in black, and non-core points in white.

Definition 2.2. A point $q \in P$ is **density-reachable** from $p \in P$ if there is a sequence of points $p_1, p_2, ..., p_t \in P$ (for some integer $t \geq 2$) such that:

— $p_1 = p$ and $p_t = q$
— $p_1, p_2, ..., p_{t-1}$ are core points
— $p_{i+1} \in B(p_i, \epsilon)$ for each $i \in [1, t-1]$.

Note that points p and q do *not* need to be different. In Figure 2, for example, o_1 is density-reachable from itself; o_{10} is density-reachable from o_1 and from o_3 (through the sequence o_3, o_2, o_1, o_{10}). On the other hand, o_{11} is *not* density-reachable from o_{10} (recall that o_{10} is not a core point).

Definition 2.3. A **cluster** C is a non-empty subset of P such that:

— (Maximality) If a core point $p \in C$, then all the points density-reachable from p also belong to C.

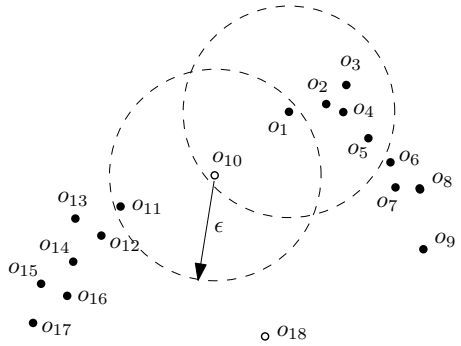


Fig. 2. An example dataset (the two circles have radius ϵ; $MinPts = 4$)

— (Connectivity) For any points $p_1, p_2 \in C$, there is a point $p \in C$ such that both p_1 and p_2 are density-reachable from p.

Definition 2.3 implies that each cluster contains at least a core point (i.e., p). In Figure 2, $\{o_1, o_{10}\}$ is *not* a cluster because it does not involve all the points density-reachable from o_1. On the other hand, $\{o_1, o_2, o_3, ..., o_{10}\}$ is a cluster.

[Ester et al. 1996] gave a nice proof that P has a unique set of clusters, which gives rise to:

PROBLEM 1. *The* **DBSCAN problem** *is to find the unique set $\mathscr{C}$ of clusters of P.*

Given the input P in Figure 2, the problem should output two clusters: $C_1 = \{o_1, o_2, ..., o_{10}\}$ and $C_2 = \{o_{10}, o_{11}, ..., o_{17}\}$.

Remark. A cluster can contain both core and non-core points. Any non-core point p in a cluster is called a *border point*. Some points may not belong to any clusters at all; they are called *noise points*. In Figure 2, o_{10} is a border point, while o_{18} is noise.

The clusters in $\mathscr{C}$ are not necessarily disjoint (e.g., o_{10} belongs to both C_1 and C_2 in Figure 2). In general, if a point p appears in more than one cluster in $\mathscr{C}$, then p must be a border point (see Lemma 2 of [Ester et al. 1996]). In other words, a core point always belongs to a unique cluster.

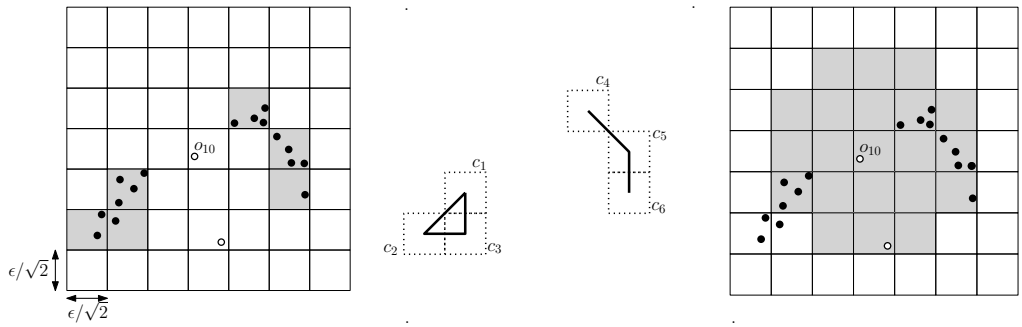
2.2. The 2D Algorithm with Genuine $O(n \log n)$ Time

Next, we explain in detail the algorithm of [Gunawan 2013], which solves the DBSCAN problem in 2D space in $O(n \log n)$ time. The algorithm imposes an arbitrary grid T in the data space $\mathbb{R}^2$, where each cell of T is a $(\epsilon/\sqrt{2}) \times (\epsilon/\sqrt{2})$ square. Without loss of generality, we assume that no point of P falls on any boundary line of T (otherwise, move T infinitesimally to make this assumption hold). Figure 3a shows a grid on the data of Figure 2. Note that any two points in the same cell are at most distance ϵ apart. A cell c of T is *non-empty* if it contains at least one point of P; otherwise, c is *empty*. Clearly, there can be at most n non-empty cells.

The algorithm then launches a *labeling process* to decide for each point $p \in P$ whether p is core or non-core. Denote by $P(c)$ the set of points of P covered by c. A cell c is a *core cell* if $P(c)$ contains at least one core point. Denote by S_{core} the set of core cells in T. In Figure 3a where $MinPts = 4$, there are 6 core cells as shown in gray (core points are in black, and non-core points in white).

Let $G = (V, E)$ be a graph defined as follows:

— Each vertex in V corresponds to a distinct core cell in S_{core}.



(a) Core cells are shown in gray

(b) Graph G

(c) ϵ-neighbor cells (in gray) of the cell of o_{10}

Fig. 3. DBSCAN with a grid ($\boldsymbol{MinPts = 4}$)

— Given two different cells $c_1, c_2 \in S_{core}$, E contains an edge between c_1 and c_2 if and only if there exist *core* points $p_1 \in P(c_1)$ and $p_2 \in P(c_2)$ such that $dist(p_1, p_2) \leq \epsilon$.

Figure 3b shows the G for Figure 3a (note that there is no edge between cells c_4 and c_6).

The algorithm then proceeds by finding all the connected components of G. Let k be the number of connected components, V_i ($1 \leq i \leq k$) be the set of vertices in the i-th connected component, and $P(V_i)$ be the set of core points covered by the cells of V_i. Then:

LEMMA 2.4 ([GUNAWAN 2013]). *The number k is also the number of clusters in P. Furthermore, $P(V_i)$ ($1 \leq i \leq k$) is exactly the set of core points in the i-th cluster.*

Figure 3b, $k = 2$, and $V_1 = \{c_1, c_2, c_3\}, V_2 = \{c_4, c_5, c_6\}$. It is easy to verify the correctness of Lemma 2.4 on this example.

Labeling Process. Let c_1 and c_2 be two different cells in T. They are *ϵ-neighbors* of each other if the minimum distance between them is less than ϵ. Figure 3c shows in gray all the ϵ-neighbor cells of the cell covering o_{10}. It is easy to see that each cell has at most 21 ϵ-neighbors. If a non-empty cell c contains at least $MinPts$ points, then *all* those points must be core points.

Now consider a cell c with $|P(c)| < MinPts$. Each point $p \in P(c)$ may or may not be a core point. To find out, the algorithm simply calculates the distances between p and *all* the points covered by *each* of the ϵ-neighbor cells of c. This allows us to know exactly the size of $|B(p, \epsilon)|$, and hence, whether p is core or non-core. For example, in Figure 3c, for $p = o_{10}$, we calculate the distance between o_{10} and all the points in the gray cells to find out that o_{10} is a non-core point.

Computation of G. Fix a core cell c_1. We will explain how to obtain the edges incident on c_1 in E. Let c_2 be a core cell that is an ϵ-neighbor of c_1. For each core point $p \in P(c_1)$, we find the core point $p' \in c_2$ that is the nearest to p. If $dist(p, p') \leq \epsilon$, an edge (c_1, c_2) is added to G. On the other hand, if all such $p \in P(c_1)$ have been tried but still no edge has been created, we conclude that E has no edge between c_1, c_2.

As a corollary of the above, each core cell c_1 has $O(1)$ incident edges in E (because it has $O(1)$ ϵ-neighbors). In other words, E has only a linear number $O(n)$ of edges.

Assigning Border Points. Recall that each $P(V_i)$ ($1 \leq i \leq k$) includes only the core points in the i-th cluster of P. It is still necessary to assign each non-core point q (i.e.,

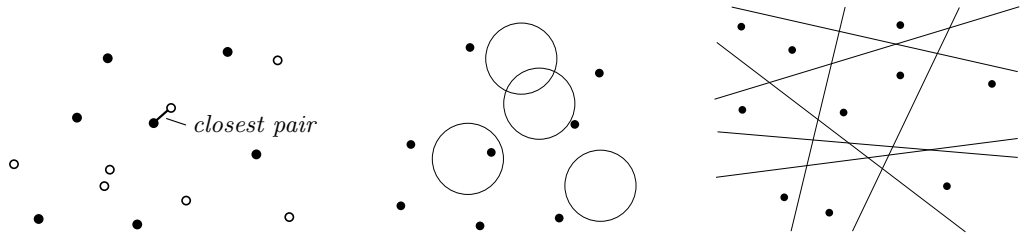


Fig. 4. Three relevant geometric problems

border point) to the appropriate clusters. The principle of doing so is simple: *if p is a core point and $dist(p, q) \leq \epsilon$, then q should be added to the (unique) cluster of p.* To find all such core points p, [Gunawan 2013] adopted the following simple algorithm. Let c be the cell where q lies. For each ϵ-neighbor cell c' of c, simply calculate the distances from q to *all* the core points in c'.

Running Time. [Gunawan 2013] showed that, other than the computation of G, the rest of the algorithm runs in $O(MinPts \cdot n) = O(n)$ expected time or $O(n \log n)$ worst-case time. The computation of G requires $O(n)$ nearest neighbor queries, each of which can be answered in $O(\log n)$ time after building a Voronoi diagram for each core cell. Therefore, the overall execution time is bounded by $O(n \log n)$.

2.3. Some Geometric Results

Bichromatic Closest Pair (BCP). Let P_1, P_2 be two sets of points in $\mathbb{R}^d$ for some constant d. Set $m_1 = |P_1|$ and $m_2 = |P_2|$. The goal of the BCP problem is to find a pair of points $(p_1, p_2) \in P_1 \times P_2$ with the smallest distance, namely, $dist(p_1, p_2) \leq dist(p'_1, p'_2)$ for any $(p'_1, p'_2) \in P_1 \times P_2$. Figure 4 shows the closest pair for a set of black points and a set of white points.

In 2D space, it is well-known that BCP can be solved in $O(m_1 \log m_1 + m_2 \log m_2)$ time. The problem is much more challenging for $d \geq 3$, for which currently the best result is due to [Agarwal et al. 1991]:

LEMMA 2.5 ([AGARWAL ET AL. 1991]). *For any fixed dimensionality $d \geq 4$, there is an algorithm solving the BCP problem in*

$$O\left((m_1 m_2)^{1-\frac{1}{\lceil d/2 \rceil + 1}+\delta'} + m_1 \log m_2 + m_2 \log m_1\right)$$

expected time, where $\delta' > 0$ can be an arbitrarily small constant. For $d = 3$, the expected running time can be improved to

$$O((m_1 m_2 \cdot \log m_1 \cdot \log m_2)^{2/3} + m_1 \log^2 m_2 + m_2 \log^2 m_1)).$$

Spherical Emptiness and Hopcroft. Let us now introduce the *unit-spherical emptiness checking* (USEC) *problem*:

> Let S_{pt} be a set of points, and S_{ball} be a set of balls *with the same radius*, all in data space $\mathbb{R}^d$, where the dimensionality d is a constant. The objective of USEC is to determine whether there is a point of S_{pt} that is covered by some ball in S_{ball}.

For example, in Figure 4b, the answer is *yes*.

Set $n = |S_{pt}|+|S_{ball}|$. In 3D space, the USEC problem can be solved in $O(n^{4/3} \cdot \log^{4/3} n)$ expected time [Agarwal et al. 1991]. Finding a 3D USEC algorithm with running time $o(n^{4/3})$ is a big open problem in computational geometry, and is widely believed to be impossible; see [Erickson 1995].

Strong hardness results are known about USEC when the dimensionality d is higher, owing to an established connection between the problem to the *Hopcroft's problem*:

> Let S_{pt} be a set of points, and S_{line} be a set of lines, all in data space $\mathbb{R}^2$ (note that the dimensionality is always 2). The goal of the Hopcroft's problem is to determine whether there is a point in S_{pt} that lies on some line of S_{line}.

For example, in Figure 4c, the answer is *no*.

The Hopcroft's problem can be settled in time slightly higher than $O(n^{4/3})$ time (see [Matousek 1993] for the precise bound), where $n = |S_{pt}| + |S_{line}|$. It is widely believed [Erickson 1995] that $\Omega(n^{4/3})$ is a lower bound on how fast the problem can be solved. In fact, this lower bound has already been proved on a broad class of algorithms [Erickson 1996].

It turns out that the Hopcroft's problem is a key reason of difficulty for a large number of other problems [Erickson 1995]. We say that a problem X is *Hopcroft hard* if an algorithm solving X in $o(n^{4/3})$ time implies an algorithm solving the Hopcroft's problem in $o(n^{4/3})$ time. In other words, a lower bound $\Omega(n^{4/3})$ on the time of solving the Hopcroft's problem implies the same lower bound on X.

[Erickson 1996] proved the following relationship between USEC and the Hopcroft's problem:

LEMMA 2.6 ([ERICKSON 1996]). *The USEC problem in any dimensionality $d \geq 5$ is Hopcroft hard.*

3. DBSCAN IN DIMENSIONALITY 3 AND ABOVE

This section paves the way towards approximate DBSCAN, which is the topic of the next section. In Section 3.1, we establish the computational hardness of DBSCAN in practice via a novel reduction from the USEC problem (see Section 2.3). For practitioners that *insist* on applying this clustering method with the utmost accuracy, in Section 3.2, we present a new exact DBSCAN algorithm that terminates in a sub-quadratic time complexity.

3.1. Hardness of DBSCAN

We will prove:

THEOREM 3.1. *The following statements are true about the DBSCAN problem:*

- *It is Hopcroft hard in any dimensionality $d \geq 5$. Namely, the problem requires $\Omega(n^{4/3})$ time to solve, unless the Hopcroft problem can be settled in $o(n^{4/3})$ time.*
- *When $d = 3$ (and hence, $d = 4$), the problem requires $\Omega(n^{4/3})$ time to solve, unless the USEC problem can be settled in $o(n^{4/3})$ time.*

As mentioned in Section 2.3, it is widely believed that neither the Hopcroft problem nor the USEC problem can be solved in $o(n^{4/3})$ time—any such algorithm would be a celebrated breakthrough in theoretical computer science.

Proof of Theorem 3.1. We observe a subtle connection between USEC and DBSCAN:

LEMMA 3.2. *For any dimensionality d, if we can solve the DBSCAN problem in $T(n)$ time, then we can solve the USEC problem in $T(n) + O(n)$ time.*

PROOF. Recall that the USEC problem is defined by a set S_{pt} of points and a set S_{ball} of balls with equal radii, both in $\mathbb{R}^d$. Denote by $\mathcal{A}$ a DBSCAN algorithm in $\mathbb{R}^d$ that runs in $T(m)$ time on m points. Next, we describe an algorithm that deploys $\mathcal{A}$ as a *black box* to solve the USEC problem in $T(n) + O(n)$ time, where $n = |S_{pt}| + |S_{ball}|$.

Our algorithm is simple:

(1) Obtain P, which is the union of S_{pt} and the set of centers of the balls in S_{ball}.
(2) Set ϵ to the identical radius of the balls in S_{ball}.
(3) Run $\mathcal{A}$ to solve the DBSCAN problem on P with this ϵ and $MinPts = 1$.
(4) If any point in S_{pt} and any center of S_{ball} belong to the same cluster, then return *yes* for the USEC problem (namely, a point in S_{pt} is covered by some ball in S_{ball}). Otherwise, return *no*.

It is fundamental to implement the above algorithm in $T(n) + O(n)$ time. Next, we prove its correctness.

Case 1: We return yes. We will show that in this case there is indeed a point of S_{pt} that is covered by some ball in S_{ball}.

Recall that a *yes* return means a point $p \in S_{pt}$ and the center q of some ball in S_{ball} have been placed in the same cluster, which we denote by C. By connectivity of Definition 2.3, there exists a point $z \in C$ such that both p and q are density-reachable from z.

By setting $MinPts = 1$, we ensure that *all* the points in P are core points. In general, if a *core point* p_1 is density-reachable from p_2 (which by definition must be a core point), then p_2 is also density-reachable from p_1 (as can be verified by Definition 2.2). This means that z is density-reachable from p, which—together with the fact that q is density-reachable from z—shows that q is density-reachable from p.

It thus follows by Definition 2.2 that there is a sequence of points $p_1, p_2, ..., p_t \in P$ such that (i) $p_1 = p, p_t = q$, and (ii) $dist(p_i, p_{i+1}) \leq \epsilon$ for each $i \in [1, t-1]$. Let k be the smallest $i \in [2, t]$ such that p_i is the center of a ball in S_{ball}. Note that k definitely exists because p_t is such a center. It thus follows that p_{k-1} is a point from S_{pt}, and that p_{k-1} is covered by the ball in S_{ball} centered at p_k.

Case 2: We return no. We will show that in this case no point of S_{pt} is covered by any ball in S_{ball}.

This is in fact very easy. Suppose on the contrary that a point $p \in S_{pt}$ is covered by a ball of S_{ball} centered at q. Thus, $dist(p, q) \leq \epsilon$, namely, q is density-reachable from p. Then, by maximality of Definition 2.3, q must be in the cluster of p (recall that all the points of P are core points). This contradicts the fact that we returned *no*. □

Theorem 3.1 immediately follows from Lemmas 2.6 and 3.2.

3.2. A New Exact Algorithm for $d \geq 3$

It is well-known that DBSCAN can be solved in $O(n^2)$ time (e.g., see [Tan et al. 2006]) in any constant dimensionality d. Next, we show that it is possible to *always* terminate in $o(n^2)$ time regardless of the constant d. Our algorithm extends that of [Gunawan 2013] with two ideas:

— Use a d-dimensional grid T with an appropriate side length for its cells.
— Compute the edges of the graph G with a BCP algorithm (as opposed to nearest neighbor search).

Next, we explain the details. T is now a grid on $\mathbb{R}^d$ where each cell of T is a d-dimensional hyper-square with side length $\epsilon/\sqrt{d}$. As before, this ensures that any two points in the same cell are within distance ϵ from each other.

The algorithm description in Section 2.2 carries over to any $d \geq 3$ almost *verbatim*. The only difference is the way we compute the edges of G. Given core cells c_1 and c_2 that are ϵ-neighbors of each other, we solve the BCP problem on the sets of core points in c_1 and c_2, respectively. Let (p_1, p_2) be the pair returned. We add an edge (c_1, c_2) to G *if and only if* $dist(p_1, p_2) \leq \epsilon$.

The adapted algorithm achieves the following efficiency guarantee:

THEOREM 3.3. *For any fixed dimensionality $d \geq 4$, there is an algorithm solving the DBSCAN problem in $O(n^{2-\frac{2}{\lceil d/2 \rceil + 1} + \delta})$ expected time, where $\delta > 0$ can be an arbitrarily small constant. For $d = 3$, the running time can be improved to $O((n \log n)^{4/3})$ expected.*

PROOF. It suffices to analyze the time used by our algorithm to generate the edges of G. The other parts of the algorithm use $O(n)$ expected time, following the analysis of [Gunawan 2013].

Let us consider first $d \geq 4$. First, fix the value of δ in Theorem 3.3. Define: $\lambda = \frac{1}{\lceil d/2 \rceil + 1} - \delta/2$. Given a core cell c, we denote by m_c the number of core points in c. Then, by Lemma 2.5, the time we spend generating the edges of G is

$$\sum_{\substack{\epsilon\text{-neighbor} \\ \text{core cells } c,\, c'}} O\left((m_c m_{c'})^{1-\lambda} + m_c \log m_{c'} + m_{c'} \log m_c\right). \tag{1}$$

To bound the first term, we derive

$$\begin{aligned}
&\sum_{\epsilon\text{-neighbor core cells } c,\, c'} O\left((m_c m_{c'})^{1-\lambda}\right) \\
&= \sum_{\substack{\epsilon\text{-neighbor} \\ \text{core cells } c,\, c' \\ \text{s.t. } m_c \leq m_{c'}}} O\left((m_c m_{c'})^{1-\lambda}\right) + \sum_{\substack{\epsilon\text{-neighbor} \\ \text{core cells } c,\, c' \\ \text{s.t. } m_c > m_{c'}}} O\left((m_c m_{c'})^{1-\lambda}\right) \\
&= \sum_{\substack{\epsilon\text{-neighbor} \\ \text{core cells } c,\, c' \\ \text{s.t. } m_c \leq m_{c'}}} O\left(m_{c'} \cdot m_c^{1-2\lambda}\right) + \sum_{\substack{\epsilon\text{-neighbor} \\ \text{core cells } c,\, c' \\ \text{s.t. } m_c > m_{c'}}} O\left(m_c \cdot m_{c'}^{1-2\lambda}\right) \\
&= \sum_{\substack{\epsilon\text{-neighbor} \\ \text{core cells } c,\, c' \\ \text{s.t. } m_c \leq m_{c'}}} O\left(m_{c'} \cdot n^{1-2\lambda}\right) + \sum_{\substack{\epsilon\text{-neighbor} \\ \text{core cells } c,\, c' \\ \text{s.t. } m_c > m_{c'}}} O\left(m_c \cdot n^{1-2\lambda}\right) \\
&= O\Big(n^{1-2\lambda} \sum_{\epsilon\text{-neighbor core cells } c,\, c'} m_c\Big) = O(n^{2-2\lambda})
\end{aligned}$$

where the last equality used the fact that c has only $O(1)$ ϵ-neighbor cells as long as d is a constant (and hence, m_c can be added only $O(1)$ times). The other terms in (1) are

easy to bound:

$$\sum_{\epsilon\text{-neighbor core cells } c,\, c'} O\left(m_c \log m_{c'} + m_{c'} \log m_c\right)$$
$$= \sum_{\epsilon\text{-neighbor core cells } c,\, c'} O\left(m_c \log n + m_{c'} \log n\right) = O(n \log n).$$

In summary, we spend $O(n^{2-2\lambda} + n \log n) = O(n^{2-\frac{2}{\lceil d/2 \rceil + 1} + \delta})$ time generating the edges of E. This proves the part of Theorem 3.3 for $d \geq 4$. An analogous analysis based on the $d = 3$ branch of Lemma 2.5 establishes the other part of Theorem 3.3. □

It is worth pointing out that the running time of our 3D algorithm nearly matches the lower bound in Theorem 3.1.

4. ρ-APPROXIMATE DBSCAN

The hardness result in Theorem 3.1 indicates the need of resorting to approximation if one wants to achieve near-linear running time for $d \geq 3$. In Section 4.1, we introduce the concept of ρ-approximate DBSCAN designed to replace DBSCAN on large datasets. In Section 4.2, we establish a strong quality guarantee of this new form of clustering. In Sections 4.3 and 4.4, we propose an algorithm for solving the ρ-approximate DBSCAN problem in time *linear* to the dataset size.

4.1. Definitions

As before, let P be the input set of n points in $\mathbb{R}^d$ to be clustered. We still take parameters ϵ and $MinPts$, but in addition, also a third parameter ρ, which can be any arbitrarily small positive constant, and controls the degree of approximation.

Next, we re-visit the basic definitions of DBSCAN in Section 2, and modify some of them to their "ρ-approximate versions". First, the notion of *core/non-core point* remains the same as Definition 2.1. The concept of *density-reachability* in Definition 2.2 is also inherited directly, but we will also need:

Definition 4.1. A point $q \in P$ is **ρ-approximate density-reachable** from $p \in P$ if there is a sequence of points $p_1, p_2, ..., p_t \in P$ (for some integer $t \geq 2$) such that:

— $p_1 = p$ and $p_t = q$
— $p_1, p_2, ..., p_{t-1}$ are core points
— $p_{i+1} \in B(p_i, \epsilon(1 + \rho))$ for each $i \in [1, t - 1]$.

Note the difference between the above and Definition 2.2: in the third bullet, the radius of the ball is increased to $\epsilon(1 + \rho)$. To illustrate, consider a small input set P as shown in Figure 5. Set $MinPts = 4$. The inner and outer circles have radii ϵ and $\epsilon(1 + \rho)$, respectively. Core and non-core points are in black and white, respectively. Point o_5 is ρ-approximate density-reachable from o_3 (via sequence: o_3, o_2, o_1, o_5). However, o_5 is *not* density-reachable from o_3.

Definition 4.2. A **ρ-approximate cluster** C is a non-empty subset of P such that:

— (Maximality) If a core point $p \in C$, then all the points density-reachable from p also belong to C.
— (ρ-Approximate Connectivity) For any points $p_1, p_2 \in C$, there exists a point $p \in C$ such that both p_1 and p_2 are ρ-approximate density-reachable from p.

Note the difference between the above and the original cluster formulation (Definition 1): the connectivity requirement has been weakened into ρ-approximate

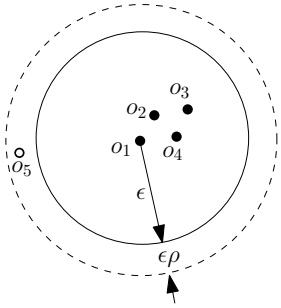


Fig. 5. Density-reachability and ρ-approximate density-reachability ($\boldsymbol{MinPts = 4}$)

connectivity. In Figure 5, both $\{o_1, o_2, o_3, o_4\}$ and $\{o_1, o_2, o_3, o_4, o_5\}$ are ρ-approximate clusters.

PROBLEM 2. *The* **ρ-approximate DBSCAN problem** *is to find a set $\mathscr{C}$ of ρ-approximate clusters of P such that every core point of P appears in exactly one ρ-approximate cluster.*

Unlike the original DBSCAN problem, the ρ-approximate version may not have a unique result. In Figure 5, for example, it is legal to return either $\{o_1, o_2, o_3, o_4\}$ or $\{o_1, o_2, o_3, o_4, o_5\}$. Nevertheless, *any* result of the ρ-approximate problem comes with the quality guarantee to be proved next.

4.2. A Sandwich Theorem

Both DBSCAN and ρ-approximate DBSCAN are parameterized by ϵ and $MinPts$. It would be perfect if they can always return exactly the same clustering results. Of course, this is too good to be true. Nevertheless, in this subsection, we will show that this is *almost* true: the result of ρ-approximate DBSCAN is guaranteed to be somewhere between the (exact) DBSCAN results obtained by $(\epsilon, MinPts)$ and by $(\epsilon(1+\rho), MinPts)$! It is well-known that the clusters of DBSCAN rarely differ considerably when ϵ changes by just a small factor—in fact, if this really happens, it suggests that the choice of ϵ is very bad, such that the exact clusters are not stable anyway (we will come back to this issue later).

Let us define:

—$\mathscr{C}_1$ as the set of clusters of DBSCAN with parameters $(\epsilon, MinPts)$
—$\mathscr{C}_2$ as the set of clusters of DBSCAN with parameters $(\epsilon(1+\rho), MinPts)$.
—$\mathscr{C}$ as an arbitrary set of clusters that is a legal result of $(\epsilon, MinPts, \rho)$-approx-DBSCAN.

The next theorem formalizes the quality assurance mentioned earlier:

THEOREM 4.3 (SANDWICH QUALITY GUARANTEE). *The following statements are true:*

(*1*) *For any cluster $C_1 \in \mathscr{C}_1$, there is a cluster $C \in \mathscr{C}$ such that $C_1 \subseteq C$.*
(*2*) *For any cluster $C \in \mathscr{C}$, there is a cluster $C_2 \in \mathscr{C}_2$ such that $C \subseteq C_2$.*

PROOF. To prove Statement 1, let p be an arbitrary core point in C_1. Then, C_1 is precisely the set of points in P density-reachable from p.[2] In general, if a point q

[2]This should be folklore but here is a proof. By maximality of Definition 2.3, all the points density-reachable from p are in C_1. On the other hand, let q be any point in C_1. By connectivity, p and q are both

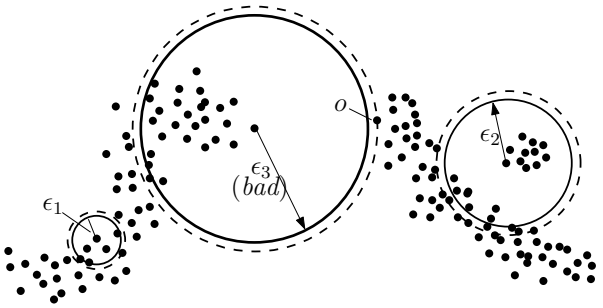


Fig. 6. Good and bad choices of ϵ

is density-reachable from p in $(\epsilon, MinPts)$-exact-DBSCAN, q is also density-reachable from p in $(\epsilon, MinPts, \rho)$-approx-DBSCAN. By maximality of Definition 4.2, if C is the cluster in $\mathscr{C}$ containing p, then all the points of C_1 must be in C.

To prove Statement 2, consider an arbitrary core point $p \in C$ (there must be one by Definition 4.2). In $(\epsilon(1+\rho), MinPts)$-exact-DBSCAN, p must also be a core point. We choose C_2 to be the cluster of $\mathscr{C}_2$ where p belongs. Now, fix an arbitrary point $q \in C$. In $(\epsilon, MinPts, \rho)$-approx-DBSCAN, by ρ-approximate connectivity of Definition 4.2, we know that p and q are both ρ-approximate reachable from a point z. This implies that z is also ρ-approximate reachable from p. Hence, q is ρ-approximate reachable from p. This means that q is density-reachable from p in $(\epsilon(1+\rho), MinPts)$-exact-DBSCAN, indicating that $q \in C_2$. □

Here is an alternative, more intuitive, interpretation of Theorem 4.3:

— Statement 1 says that if two points belong to the same cluster of DBSCAN with parameters $(\epsilon, MinPts)$, they are *definitely* in the same cluster of ρ-approximate DBSCAN with the same parameters.
— On the other hand, a cluster of ρ-approximate DBSCAN parameterized by $(\epsilon, MinPts)$ may also contain two points p_1, p_2 that are in different clusters of DBSCAN with the same parameters. However, this is not bad because Statement 2 says that as soon as the parameter ϵ increases to $\epsilon(1+\rho)$, p_1 and p_2 will fall into the same cluster of DBSCAN!

Figure 6 illustrates the effects of approximation. How many clusters are there? Interestingly, the answer is *it depends*. As pointed out in the classic OPTICS paper [Ankerst et al. 1999], different ϵ values allow us to view the dataset from various granularities, leading to different clustering results. In Figure 6, given ϵ_1 (and some $MinPts$ say 2), DBSCAN outputs 3 clusters. Given ϵ_2, on the other hand, DBSCAN outputs 2 clusters, which makes sense because at this distance, the two clusters on the right merge into one.

Now let us consider approximation. The dashed circles illustrate the radii obtained with ρ-approximation. For both ϵ_1 and ϵ_2, ρ-approximate DBSCAN will return exactly the same clusters, because these distances are *robustly chosen* by being insensitive to small perturbation. For ϵ_3, however, ρ-approximate DBSCAN may return only one cluster (i.e., all points in the same cluster), whereas exact DBSCAN will return only two (i.e., the same two clusters as ϵ_2). By looking at the figure closely, one can realize that this happens because the dashed circle of radius $(1+\rho)\epsilon_3$) "happens" to pass a

density-reachable from a point z. As p is a core point, we know that z is also density-reachable from p. Hence, q is density-reachable from p.

point—namely point o—which falls outside the solid circle of radius ϵ_3. Intuitively, ϵ_3 is a poor parameter choice because it is *too* close to the distance between two clusters such that a small change to it will cause the clustering results to be altered.

Next we present a useful corollary of the sandwich theorem:

COROLLARY 4.4. *Let $\mathscr{C}_1, \mathscr{C}_2$, and $\mathscr{C}$ be as defined in Theorem 4.3. If a cluster C appears in both $\mathscr{C}_1$ and $\mathscr{C}_2$, then C must also be a cluster in $\mathscr{C}$.*

PROOF. Suppose, on the contrary, that $\mathscr{C}$ does not contain C. By Theorem 4.3, (i) $\mathscr{C}$ must contain a cluster C' such that $C \subseteq C'$, and (ii) $\mathscr{C}_2$ must contain a cluster C'' such that $C' \subseteq C''$. This means $C \subseteq C''$. On the other hand, as $C \in \mathscr{C}_2$, it follows that, in $\mathscr{C}_2$, every core point in C belongs also to C''. This is impossible because a core point can belong to only one cluster. □

The corollary states that, even if some exact DBSCAN clusters have changed when ϵ increases by a factor of $1+\rho$ (i.e., ϵ is not robust), our ρ-approximation still captures all those clusters that do not change. For example, imagine that the points in Figure 6 are part of a larger dataset such that the clusters on the rest of the points are unaffected as ϵ_3 increases to $\epsilon_3(1+\rho)$. By Corollary 4.4, all those clusters are safely captured by ρ-approximate DBSCAN under ϵ_3.

4.3. Approximate Range Counting

Let us now take a break from DBSCAN, and turn our attention to a different problem, whose solution is vital to our ρ-approximate DBSCAN algorithm.

Let P still be a set of n points in $\mathbb{R}^d$ where d is a constant. Given any point $q \in \mathbb{R}^d$, a distance threshold $\epsilon > 0$ and an arbitrarily small constant $\rho > 0$, an *approximate range count query* returns an integer that is guaranteed to be between $|B(q,\epsilon) \cap P|$ and $|B(q,\epsilon(1+\rho)) \cap P|$. For example, in Figure 5, given $q = o_1$, a query may return either 4 or 5.

[Arya and Mount 2000] developed a structure of $O(n)$ space that can be built in $O(n \log n)$ time, and answers any such query in $O(\log n)$ time. Next, we design an alternative structure with better performance in our context:

LEMMA 4.5. *For any fixed ϵ and ρ, there is a structure of $O(n)$ space that can be built in $O(n)$ expected time, and answers any approximate range count query in $O(1)$ expected time.*

Structure. Our structure is a simple quadtree-like hierarchical grid partitioning of $\mathbb{R}^d$. First, impose a regular grid on $\mathbb{R}^d$ where each cell is a d-dimensional hyper-square with side length $\epsilon/\sqrt{d}$. For each *non-empty* cell c of the grid (i.e., c covers at least 1 point of P), divide it into 2^d cells of the same size. For each resulting *non-empty* cell c', divide it recursively in the same manner, until the side length of c' is at most $\epsilon\rho/\sqrt{d}$.

We use H to refer to the hierarchy thus obtained. We keep *only* the non-empty cells of H, and for each such cell c, record $cnt(c)$ which is the number of points in P covered by c. We will refer to a cell of H with side length $\epsilon/(2^i\sqrt{d})$ as a *level-i cell*. Clearly, H has only $h = \max\{1, 1 + \lceil \log_2(1/\rho) \rceil\} = O(1)$ levels. If a level-$(i+1)$ cell c' is inside a level-i cell c, we say that c' is a *child* of c, and c a *parent* of c'. A cell with no children is called a *leaf cell*.

Figure 7 illustrates the part of the first three levels of H for the dataset on the left. Note that empty cells are *not* stored.

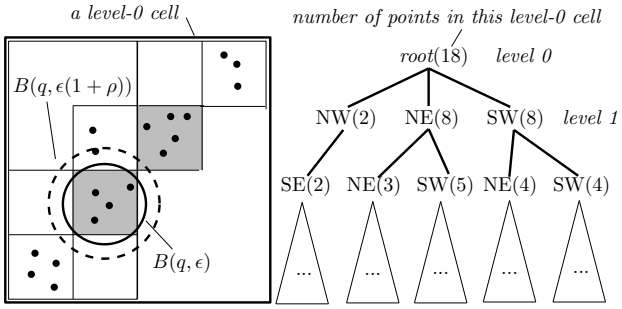


Fig. 7. Approximate range counting

Query. Given an approximate range count query with parameters q, ϵ, ρ, we compute its answer ans as follows. Initially, $ans = 0$. In general, given a non-empty level-i cell c, we distinguish three cases:

— If c is disjoint with $B(q, \epsilon)$, ignore it.
— If c is fully covered by $B(q, \epsilon(1+\rho))$, add $cnt(c)$ to ans.
— When neither of the above holds, check if c is a leaf cell in H. If not, process the child cells of c in the same manner. Otherwise (i.e., c is a leaf), add $cnt(c)$ to ans only if c intersects $B(q, \epsilon)$.

The algorithm starts from the level-0 non-empty cells that intersect with $B(q, \epsilon)$.

To illustrate, consider the query shown in Figure 7. The two gray cells correspond to nodes SW(5) and NE(4) at level 2. The subtree of *neither* of them is visited, but the reasons are different. For SW(5), its cell is disjoint with $B(q, \epsilon)$, so we ignore it (even though it intersects $B(q, \epsilon(1+\rho))$). For NE(4), its cell completely falls in $B(q, \epsilon(1+\rho))$, so we add its count 4 to the result (even though it is not covered by $B(q, \epsilon)$).

Correctness. The above algorithm has two guarantees. First, if a point $p \in P$ is inside $B(q, \epsilon)$, it is definitely counted in ans. Second, if p is outside $B(q, \epsilon(1+\rho))$, then it is definitely *not* counted in ans. These guarantees are easy to verify, utilizing the fact that if a leaf cell c intersects $B(p, \epsilon)$, then c must fall *completely* in $B(p, \epsilon(1+\rho))$ because any two points in a leaf cell are within distance $\epsilon\rho$. It thus follows that the ans returned is a legal answer.

Time Analysis. Remember that the hierarchy H has $O(1)$ levels. Since there are $O(n)$ non-empty cells at each level, the total space is $O(n)$. With hashing, it is easy to build the structure level by level in $O(n)$ expected time.

To analyze the running time of our query algorithm, observe that each cell c visited by our algorithm must satisfy one of the following conditions: (i) c is a level-0 cell, or (ii) the parent of c intersects the boundary of $B(q, \epsilon)$. For type-(i), the $O(1)$ level-0 cells intersecting $B(q, \epsilon)$ can be found in $O(1)$ expected time using the coordinates of q. For type-(ii), it suffices to bound the number of cells intersecting the boundary of $B(q, \epsilon)$ because each such cell has $2^d = O(1)$ child nodes.

In general, a d-dimensional grid of cells with side length l has $O(1 + (\frac{\theta}{l})^{d-1})$ cells intersecting the boundary of a sphere with radius θ [Arya and Mount 2000]. Combining this and the fact that a level-i cell has side length $\epsilon/(2^i\sqrt{d})$, we know that the total

number of cells (of all levels) intersecting the boundary of $B(q, \epsilon)$ is bounded by:

$$\sum_{i=0}^{h-1} O\left(1 + \left(\frac{\epsilon}{\epsilon/(2^i\sqrt{d})}\right)^{d-1}\right) = O\left((2^h)^{d-1}\right)$$
$$= O\left(1 + (1/\rho)^{d-1}\right)$$

which is a constant for any fixed ρ. This concludes the proof of Lemma 4.5.

4.4. Solving ρ-Approximate DBSCAN

We are now ready to solve the ρ-approximate DBSCAN problem by proving:

THEOREM 4.6. *There is a ρ-approximate DBSCAN algorithm that terminates in $O(n)$ expected time, regardless of the value of ϵ, the constant approximation ratio ρ, and the fixed dimensionality d.*

Algorithm. Our ρ-approximate algorithm differs from the exact algorithm we proposed in Section 3.2 *only* in the definition and computation of the graph G. We re-define $G = (V, E)$ as follows:

— As before, each vertex in V is a core cell of the grid T (remember that the algorithm of Section 3.2 imposes a grid T on $\mathbb{R}^d$, where a cell is a core cell if it covers at least one core point).
— Given two different core cells c_1, c_2, whether E has an edge between c_1 and c_2 obeys the rules below:
  — yes, if there exist core points p_1, p_2 in c_1, c_2, respectively, such that $dist(p_1, p_2) \leq \epsilon$.
  — no, if no core point in c_1 is within distance $\epsilon(1 + \rho)$ from any core point in c_2.
  — *don't care*, in all the other cases.

To compute G, our algorithm starts by building, for each core cell c in T, a structure of Lemma 4.5 on the set of core points in c. To generate the edges of a core cell c_1, we examine each ϵ-neighbor cell c_2 of c_1 in turn. For every core point p in c_1, do an approximate range count query on the set of core points in c_2. If the query returns a non-zero answer, add an edge (c_1, c_2) to G. If all such p have been tried but still no edge has been added, we decide that there should be no edge between c_1 and c_2.

Correctness. Let C be an arbitrary cluster returned by our algorithm. We will show that C satisfies Definition 4.2.

Maximality. Let p be an arbitrary core point in C, and q be any point of P density-reachable from p. We will show that $q \in C$. Let us start by considering that q is a core point. By Definition 2.2, there is a sequence of core points $p_1, p_2, ..., p_t$ (for some integer $t \geq 2$) such that $p_1 = p$, $p_t = q$, and $dist(p_{i+1}, p_i) \leq \epsilon$ for each $i \in [1, t-1]$. Denote by c_i the cell of T covering p_i. By the way G is defined, there must be an edge between c_i and c_{i+1}, for each $i \in [1, t-1]$. It thus follows that c_1 and c_t must be in the same connected component of G; therefore, p and q must be in the same cluster. The correctness of the other scenario where q is a non-core point is trivially guaranteed by the way that non-core points are assigned to clusters.

ρ-Approximate Connectivity. Let p be an arbitrary core point in C. For *any* point $q \in C$, we will show that q is ρ-approximate density-reachable from p. Again, we consider first that q is a core point. Let c_p and c_q be the cells of T covering p and q, respectively. Since c_p and c_q are in the same connected component of G, there is a path $c_1, c_2, ..., c_t$ in G (for some integer $t \geq 2$) such that $c_1 = c_p$ and $c_t = c_q$. Recall that any two points in the same cell are within distance ϵ. Combining this fact with how the edges of G are

defined, we know that there is a sequence of core points $p_1, p_2, ..., p_{t'}$ (for some integer $t' \geq 2$) such that $p_1 = p$, $p_{t'} = q$, and $dist(p_{i+1}, p_i) \leq \epsilon(1 + \rho)$ for each $i \in [1, t' - 1]$. Therefore, q is ρ-approximate density-reachable from p. The correctness of the other scenario where q is a non-core point is again trivial.

Time Analysis. It takes $O(n)$ expected time to construct the structure of Lemma 4.5 for all cells. The time of computing G is proportional to the number of approximate range count queries issued. For each core point of a cell c_1, we issue $O(1)$ queries in total (one for each ϵ-neighbor cell of c_2). Hence, the total number of queries is $O(n)$. The rest of the ρ-approximate algorithm runs in $O(n)$ expected time, following the same analysis in [Gunawan 2013]. This completes the proof of Theorem 4.6. It is worth mentioning that, intuitively, the efficiency improvement of our approximate algorithm (over the exact algorithm in Section 3.2) owes to the fact that we settle for an imprecise solution to the BCP problem by using Lemma 4.5.

Remark. It should be noted that the hidden constant in $O(n)$ is at the order of $(1/\rho)^{d-1}$; see the proof of Lemma 4.5. As this is exponential to the dimensionality d, our techniques are suitable only when d is low. Our experiments considered dimensionalities up to 7.

5. NEW 2D EXACT ALGORITHMS

This section gives two new algorithms for solving the (exact) DBSCAN problem in $\mathbb{R}^2$. These algorithms are based on different ideas, and are interesting in their own ways. The first one (Section 5.1) is conceptually simple, and establishes a close connection between DBSCAN and Delaunay graphs. The second one (Section 5.2) manages to identify coordinate sorting as the most expensive component in DBSCAN computation.

5.1. DBSCAN from a Delaunay Graph

Recall from Section 2.2 that Gunawan's algorithm runs in three steps:

(1) Label each point of the input set P as either core or non-core.
(2) Partition the set P_{core} of core points into clusters.
(3) Assign each non-core point to the appropriate cluster(s).

Step 2 is the performance bottleneck. Next, we describe a new method to accomplish this step.

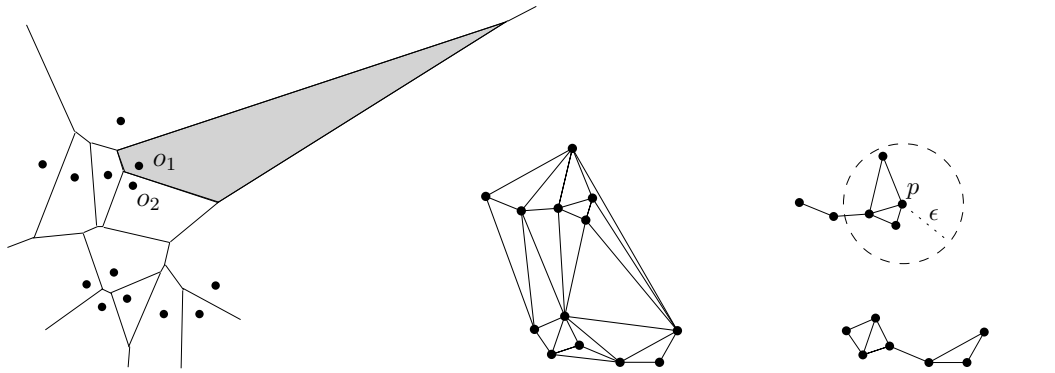
Algorithm for Step 2. The Delaunay graph of P_{core} can be regarded as the dual of the Voronoi diagram of P_{core}. The latter is a subdivision of the data space $\mathbb{R}^2$ into $|P_{core}|$ convex polygons, each of which corresponds to a distinct $p \in P_{core}$, and is called the *Voronoi cell* of p, containing every location in $\mathbb{R}^2$ that finds p as its Euclidean nearest neighbor in P_{core}. The *Delaunay graph* of P_{core} is a graph $G_{dln} = (V_{dln}, E_{dln})$ defined as follows:

— $V_{dln} = P_{core}$, that is, every core point is a vertex of G_{dln}.
— E_{dln} contains an edge between two core points p_1, p_2 if and only if their Voronoi cells are adjacent (i.e., sharing a common boundary segment).

G_{dln}, in general, *always* has only a linear number of edges, i.e., $|E_{dln}| = O(|P_{core}|)$.

Figure 8a demonstrates the Voronoi diagram defined by the set of black points shown. The shaded polygon is the Voronoi cell of o_1; the Voronoi cells of o_1 and o_2 are adjacent. The corresponding Delaunay graph is given in Figure 8b.

Provided that G_{dln} is already available, we perform Step 2 using a simple strategy:



(a) Voronoi diagram (b) Delaunay graph (c) Remainder graph after edge removal

Fig. 8. Illustration of our Step-2 Algorithm in Section 5.1

(2. 1) Remove all the edges (p_1, p_2) in E_{dln} such that $dist(p_1, p_2) > \epsilon$. Let us refer to the resulting graph as the *remainder graph*.
(2. 2) Compute the connected components of the remainder graph.
(2. 3) Put the core points in each connected component into a separate cluster.

Continuing the example in Figure 8b, Figure 8c illustrates the remainder graph after the edge removal in Step 2.1 (the radius of the circle centered at point p indicates the value of ϵ). There are two connected components in the remainder graph; the core points in each connected component constitute a cluster.

In general, the Delaunay graph of x 2D points can be computed in $O(x \log x)$ time [de Berg et al. 2008]. Clearly, Steps 2.1-2.3 require only $O(|P_{core}|) = O(n)$ time. Therefore, our Step 2 algorithm finishes in $O(n \log n)$ time overall.

Correctness of the Algorithm. It remains to explain why the above simple strategy correctly clusters the core points. Remember that a core point p ought to be placed in the same cluster as another core point q if and only if there is a sequence of core points $p_1, p_2, ..., p_t$ (for some $t \geq 2$) such that:

— $p_1 = p$ and $p_t = q$
— $dist(p_i, p_{i+1}) \leq \epsilon$ for each $i \in [1, t-1]$.

We now prove:

LEMMA 5.1. *Two core points p, q belong to the same cluster if and only if our Step-2 algorithm declares so.*

PROOF. *The If Direction.* This direction is straightforward. Our algorithm declares p, q to be in the same cluster only if they appear in the same connected component of the remainder graph obtained at Step 2.1. This, in turn, suggests that the connected component has a path starting from p and ending at q satisfying the aforementioned requirement.

The Only-If Direction. Let p, q be a pair of core points that should be placed in the same cluster. Next, we will prove that our Step-2 algorithm definitely puts them in the same connected component of the remainder graph.

We will first establish this fact by assuming $dist(p, q) \leq \epsilon$. Consider the line segment pq. Since Voronoi cells are convex polygons, in moving on segment pq from p to q, we

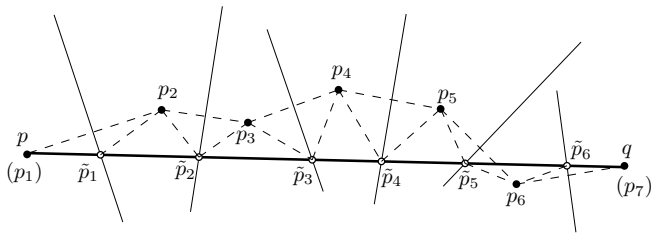


Fig. 9. Correctness proof of our Step-2 algorithm

must be traveling through the Voronoi cells of a sequence of distinct core points—let them be $p_1, p_2, ..., p_t$ for some $t \geq 2$, where $p_1 = p$ and $p_t = q$. Our goal is to show that $dist(p_i, p_{i+1}) \leq \epsilon$ for *all* $i \in [1, t-1]$. This will indicate that the remainder graph must contain an edge between each pair of (p_i, p_{i+1}) for all $i \in [1, t-1]$, implying that all of $p_1 = p, p_2, ..., p_t = q$ must be in the same connected component at Step 2.3.

We now prove $dist(p_i, p_{i+1}) \leq \epsilon$ for an arbitrary $i \in [1, t-1]$. Let $\tilde{p}_i$ (for $i \in [1, t-1]$) be the intersection between pq and the common boundary of the Voronoi cells of p_i and p_{i+1}. Figure 9 illustrates the definition with an example where $t = 7$. We will apply triangle inequality a number of times to arrive at our target conclusion. Let us start with:

$$dist(p_i, p_{i+1}) \ \leq \ dist(p_i, \tilde{p}_i) + dist(p_{i+1}, \tilde{p}_i). \tag{2}$$

Regarding $dist(p_i, \tilde{p}_i)$, we have:

$$\begin{aligned}
dist(p_i, \tilde{p}_i) \ &\leq \ dist(p_i, \tilde{p}_{i-1}) + dist(\tilde{p}_{i-1}, \tilde{p}_i) \\
&= \ dist(p_{i-1}, \tilde{p}_{i-1}) + dist(\tilde{p}_{i-1}, \tilde{p}_i) \\
&\quad \text{(note: } dist(p_{i-1}, \tilde{p}_{i-1}) = dist(p_i, \tilde{p}_{i-1}) \text{ as } \tilde{p}_{i-1} \text{ is on the} \\
&\quad \text{perpendicular bisector of segment } p_i p_{i-1}\text{)} \\
&\leq \ dist(p_{i-1}, \tilde{p}_{i-2}) + dist(\tilde{p}_{i-2}, \tilde{p}_{i-1}) + dist(\tilde{p}_{i-1}, \tilde{p}_i) \\
&\quad \text{(triangle inequality)} \\
&= \ dist(p_{i-1}, \tilde{p}_{i-2}) + dist(\tilde{p}_{i-2}, \tilde{p}_i) \\
&... \\
&\leq \ dist(p_2, \tilde{p}_1) + dist(\tilde{p}_1, \tilde{p}_i) \\
&= \ dist(p_1, \tilde{p}_1) + dist(\tilde{p}_1, \tilde{p}_i) \\
&= \ dist(p_1, \tilde{p}_i).
\end{aligned} \tag{3}$$

Following a symmetric derivation, we have:

$$dist(p_{i+1}, \tilde{p}_i) \ \leq \ dist(\tilde{p}_i, p_t). \tag{4}$$

The combination of (2)-(4) gives:

$$\begin{aligned}
dist(p_i, p_{i+1}) \ &\leq \ dist(p_1, \tilde{p}_i) + dist(\tilde{p}_i, p_t) \\
&= \ dist(p_1, p_t) \leq \epsilon
\end{aligned}$$

as claimed.

We now get rid of the assumption that $dist(p, q) \leq \epsilon$. This is fairly easy. By the given fact that p and q should be placed in the same cluster, we know that there is a path $p_1 = p, p_2, p_3, ..., p_t = q$ (where $t \geq 2$) such that $dist(p_i, p_{i+1}) \leq \epsilon$ for each $i \in [1, t-1]$. By our earlier argument, each pair of (p_i, p_{i+1}) must be in the same connected component

of our remainder graph. Consequently, all of $p_1, p_2, ..., p_t$ are in the same connected component. This completes the proof. □

Remark. The concepts of Voronoi Diagram and Delaunay graph can both be extended to arbitrary dimensionality $d \geq 3$. Our Step-2 algorithm also works for any $d \geq 3$. While this may be interesting from a geometric point of view, it is not from an algorithmic perspective. Even at $d = 3$, a Delaunay graph on n points can have $\Omega(n^2)$ edges, necessitating $\Omega(n^2)$ time for its computation. In contrast, in Section 3.2, we already showed that the exact DBSCAN problem can be solved in $o(n^2)$ time for any constant dimensionality d.

5.2. Separation of Sorting from DBSCAN

We say that the 2D input set P is *bi-dimensionally sorted* if the points therein are given in two sorted lists:

— P_x, where the points are sorted by x-dimension;
— P_y, where the points are sorted by y-dimension.

This subsection will establish the last main result of this article:

THEOREM 5.2. *If P has been bi-dimensionally sorted, the exact DBSCAN problem (in 2D space) can be solved in $O(n)$ worst-case time.*

The theorem reveals that coordinate sorting is actually the "hardest" part of the 2D DBSCAN problem! This means that we can even beat the $\Omega(n \log n)$ time bound for this problem in scenarios where sorting can be done fast. The corollaries below state two such scenarios:

COROLLARY 5.3. *If each dimension has an integer domain of size at most n^c for an arbitrary positive constant c, the 2D DBSCAN problem can be solved in $O(n)$ worst-case time (even if P is not bi-dimensionally sorted).*

PROOF. [Kirkpatrick and Reisch 1984] showed that n integers drawn from a domain of size n^c (regardless of the constant $c \geq 1$) can be sorted in $O(n)$ time, by generalizing the idea of radix sort. Using their algorithm, P can be made bi-dimensionally sorted in $O(n)$ time. Then, the corollary follows from Theorem 5.2. □

The above corollary is important because, in real applications, (i) coordinates are always discrete (after digitalization), and (ii) when n is large (e.g., 10^6), the domain size of each dimension rarely exceeds n^2. The 2D DBSCAN problem can be settled in linear time in all such applications.

COROLLARY 5.4. *If each dimension has an integer domain, the 2D DBSCAN problem can be solved in $O(n \log \log n)$ worst-case time or $O(n\sqrt{\log \log n})$ expected time (even if P is not bi-dimensionally sorted).*

PROOF. [Andersson et al. 1998] gave a deterministic algorithm to sort n integers in $O(n \log \log n)$ worst-case time. [Han and Thorup 2002] gave a randomized algorithm to do so in $O(n\sqrt{\log \log n})$ expected time. Plugging these results into Theorem 5.2 yields the corollary. □

Next, we provide the details of our algorithm for Theorem 5.2. The general framework is still the 3-step process as shown in Section 5.1, but we will develop new methods to implement Steps 1 and 2 in linear time, utilizing the property that P is bi-dimensionally sorted. Step 3 is carried out in the same manner as in the Gunawan's algorithm (Section 2.2), which demands only $O(n)$ time.

5.2.1. Step 1. Recall that, for this step, Gunawan's algorithm places an arbitrary grid T (where each cell is a square with side length $\epsilon/\sqrt{2}$) in $\mathbb{R}^2$, and then proceeds as follows:

(1. 1) For each non-empty cell c of T, compute the set $P(c)$ of points in P that are covered by c.
(1. 2) For each non-empty cell c of T, identify all of its non-empty ϵ-neighbor cells c' (i.e., the minimum distance between c and c' is less than ϵ).
(1. 3) Perform a labeling process to determine whether each point in P is a core or non-core point.

Our approach differs from Gunawan's in Steps 1.1 and 1.2 (his solution to Step 1.3 takes only $O(n)$ time, and is thus sufficient for our purposes). Before continuing, note that Steps 1.1 and 1.2 can be done easily with hashing using $O(n)$ *expected* time, but our goal is to attain the same time complexity in the worst case.

Step 1.1. We say that a column of T (a *column* contains all the cells of T sharing the same projection on the x-dimension) is *non-empty* if it has at least one non-empty cell. We label the leftmost non-empty column as 1, and the 2nd leftmost non-empty column as 2, and so on. By scanning P_x once in ascending order of x-coordinate, we determine, for each point $p \in P$, the label of the non-empty column that contains p; the time required is $O(n)$.

Suppose that there are n_{col} non-empty columns. Next, for each $i \in [1, n_{col}]$, we generate a sorted list $P_y[i]$ that arranges, in ascending of y-coordinate, the points of P covered by (non-empty) column i. In other words, we aim to "distribute" P_y into n_{col} sorted lists, one for each non-empty column. This can be done in $O(n)$ time as follows. First, initialize all the n_{col} lists to be empty. Then, scan P_y in ascending order of y-coordinate; for each point p seen, append it to $P_y[i]$ where i is the label of the column containing p. The point ordering in P_y ensures that each $P_y[i]$ thus created is sorted on y-dimension.

Finally, for each $i \in [1, n_{col}]$, we generate the target set $P(c)$ for every non-empty cell c in column i, by simply scanning $P_y[i]$ once in order to divide it into sub-sequences, each of which includes all the points in a distinct cell (sorted by y-coordinate). The overall cost of Step 1.1 is therefore $O(n)$. As a side product, for every $i \in [1, n_{col}]$, we have also obtained a list L_i of all the non-empty cells in column i, sorted in bottom-up order.

Step 1.2. We do so by processing each non-empty column in turn. First, observe that if a cell is in column $i \in [1, n_{col}]$, all of its ϵ-neighbor cells must appear in columns $i-2, i-1, i, i+1$, and $i+2$ (see Figure 3c). Motivated by this, for each $j \in \{i-2, i-1, i, i+1, i+2\} \cap [1, n_{col}]$, we scan synchronously the cells of L_i and L_j in bottom-up order (if two cells are at the same row, break the tie by scanning first the one from L_i). When a cell $c \in L_i$ is encountered, we pinpoint the last cell $c_0 \in L_j$ that was scanned. Define:

— c_{-1} as the cell in L_j immediately *before* c_0;
— c_1 as the cell in L_j immediately *after* c_0;
— c_2 as the cell in L_j immediately after c_1;
— c_3 as the cell in L_j immediately after c_2;

The 5 cells[3] $c_{-1}, c_0, ..., c_3$ are the only ones that can be ϵ-neighbors of c in L_j. Checking which of them are indeed ϵ-neighbors of c takes $O(1)$ time. Hence, the synchronous

[3]If $c_0 = \emptyset$ (namely, no cell in L_j has been scanned), set c_1, c_2, c_3 to the lowest 3 cells in L_j.

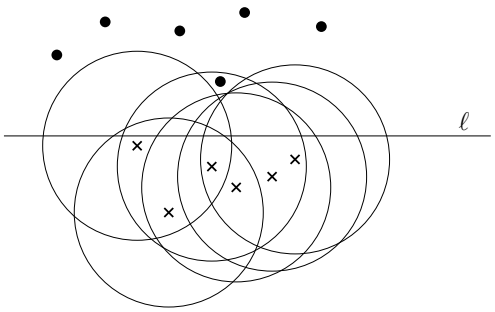


Fig. 10. USEC with line separation

scan of L_i and L_j costs $O(|L_i| + |L_j|)$ time. The total cost of Step 1.2 is, therefore, $O(n)$, noticing that each L_i ($i \in [1, n_{col}]$) will be scanned at most 5 times.

Remark. By slightly extending the above algorithm, for each non-empty cell c, we can store the points of $P(c)$ in two sorted lists:

— $P_x(c)$, where the points of $P(c)$ are sorted on x-dimension;
— $P_y(c)$, where the points are sorted on y-dimension.

To achieve this purpose, first observe that, at the end of Step 1.1, the sub-sequence obtained for each non-empty cell c is precisely $P_y(c)$. This allows us to know, for each point $p \in P$, the id of the non-empty cell covering it. After this, the $P_x(c)$ of all non-empty cells c can be obtained with just another scan of P_x: for each point p seen in P_x, append it to $P_x(c)$, where c is the cell containing p. The point ordering in P_x ensures that each $P_x(c)$ is sorted by x-coordinate, as desired. The additional time required is still $O(n)$.

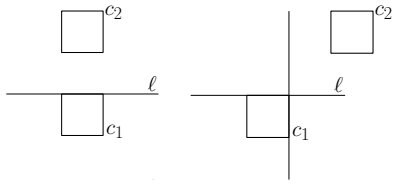
5.2.2. Step 2. For this step, Gunawan's algorithm generates a graph $G = (V, E)$ where each core cell in T corresponds to a distinct vertex in V. Between core cells (a.k.a., vertices) c_1 and c_2, an edge exists in E if and only if there is a core point p_1 in c_1 and a core point p_2 in c_2 such that $dist(p_1, p_2) \leq \epsilon$. Once G is available, Step 2 is accomplished in $O(n)$ time by computing the connected components of G. The performance bottleneck lies in the creation of G, to which Gunawan's solution takes $O(n \log n)$ time. We develop a new algorithm below that fulfills the purpose in $O(n)$ time.

USEC with Line Separation. Let us introduce a special variant of the USEC problem defined in Section 2.3, which stands at the core of our $O(n)$-time algorithm. Recall that in the 2D USEC problem, we are given a set S_{ball} of discs with the same radius ϵ, and a set S_{pt} of points, all in the data space $\mathbb{R}^2$. The objective is to determine whether any point in S_{pt} is covered by any disc in S_{ball}. In our special variant, there are two extra constraints:

— There is a horizontal line ℓ such that (i) all the centers of the discs in S_{ball} are on or below ℓ, and (ii) all the points in S_{pt} are on or above ℓ.
— The centers of the discs in S_{ball} have been sorted by x-dimension, and so are the points in S_{pt}.

Figure 10 illustrates an instance of the above *USEC with line separation problem* (where crosses indicate disc centers). The answer to this instance is *yes* (i.e., a point falls in a disc).

LEMMA 5.5. *The USEC with line separation problem can be settled in linear time, namely, with cost* $O(|S_{pt}| + |S_{ball}|)$.



(a) Case 1 (b) Case 2

Fig. 11. Deciding the existence of an edge by USEC with line separation

An algorithm for achieving the above lemma is implied in [Bose et al. 2007]. However, the description in [Bose et al. 2007] is rather brief, and does not provide the full details. In the appendix, we reconstruct their algorithm, and prove its correctness (such a proof was missing in [Bose et al. 2007]). Nonetheless, we believe that credits on the lemma should be attributed to [Bose et al. 2007]. The reader may also see [de Berg et al. 2015] for another account of the algorithm.

Generating G in $O(n)$ Time. We now return to our endeavor of finding an $O(n)$ time algorithm to generate G. The vertices of G, which are precisely the core cells, can obviously be collected in $O(n)$ time (there are at most n core cells). It remains to discuss the creation of the edges in G.

Now, focus on any two core cells c_1 and c_2 that are ϵ-neighbors of each other. Our mission is to determine whether there should be an edge between them. It turns out that this requires solving at most two instances of USEC with line separation. Following our earlier terminology, let $P(c_1)$ be the set of points of P that fall in c_1. Recall that we have already obtained two sorted lists of $P(c_1)$, that is, $P_x(c_1)$ and $P_y(c_1)$ that are sorted by x- and y-dimension, respectively. Define $P(c_2), P_x(c_2)$, and $P_y(c_2)$ similarly for c_2. Depending on the relative positions of c_1 and c_2, we proceed differently in the following two cases (which essentially have represented all possible cases by symmetry):

— Case 1: c_2 is in the same column as c_1, and is above c_1, as in Figure 11a. Imagine placing a disc centered at each point in $P(c_1)$. All these discs constitute S_{ball}. Set S_{pt} directly to $P(c_2)$. Together with the horizontal line ℓ shown, this defines an instance of USEC with line separation. There is an edge between c_1, c_2 if and only if the instance has a *yes* answer.
— Case 2: c_2 is to the northeast of c_1, as in Figure 11b. Define S_{ball} and S_{pt} in the same manner as before. They define an instance of USEC with line separation based on ℓ. There is an edge between c_1, c_2 if and only if the instance has a *yes* answer.

It is immediately clear from Lemma 5.5 that we can make the correct decision about the edge existence between c_1, c_2 using $O(|P(c_1)| + |P(c_2)|)$ time. Therefore, the total cost of generating all the edges in G is bounded by:

$$\sum_{\text{core cell } c_1} \left(\sum_{\epsilon\text{-neighbor } c_2 \text{ of } c_1} O(|P(c_1)| + |P(c_2)|) \right) = \sum_{\text{core cell } c_1} O(|P(c_1)|) = O(n)$$

where the first equality used the fact that each core cell has $O(1)$ ϵ-neighbors, and hence, can participate in only $O(1)$ instances of USEC with line separation.

6. DISCUSSION ON PRACTICAL EFFICIENCY

Besides our theoretical findings, we have developed a software prototype based on the proposed algorithms. Our implementation has evolved beyond that of [Gan and Tao

2015] by incorporating new heuristics (note also that [Gan and Tao 2015] focused on $d \geq 3$). Next, we will explain the most crucial heuristics adopted which apply to all of our algorithms (since they are based on the same grid-based framework). Then, we will discuss when the original DBSCAN algorithm of [Ester et al. 1996] is or is not expected to work well in practice. Finally, a qualitative comparison of the precise and ρ-approximate DBSCAN algorithms will be presented.

Heuristics. The three most effective heuristics in our implementation can be summarized as follows:

— Recall that our ρ-approximate algorithm imposes a grid T on $\mathbb{R}^d$. We manage all the non-empty cells in a (main memory) R-tree which is constructed by bulkloading. This R-tree allows us to efficiently find, for any cell c, all its ϵ-neighbor non-empty cells c'. Recall that such an operation is useful in a number of scenarios: (i) in the labeling process when a point p falls in a cell covering less than $MinPts$ points, (ii) in deciding the edges of c in G, and (iii) assigning a non-core point in c to appropriate clusters.

— For every non-empty cell c, we store all its ϵ-neighbor non-empty cells in a list, after they have been computed for the first time. As each list has length $O(1)$, the total space of all the lists is $O(n)$ (recall that at most n non-empty cells exist). The lists allow us to avoid re-computing ϵ-neighbor non-empty cells of c.

— Theoretically speaking, we achieve $O(n)$ expected time by first generating the edges of G and then computing its connected components (CC). In reality, it is faster not to produce the edges, but instead, maintain the CCs using a union-find structure [Tarjan 1979].

  Specifically, whenever an edge between non-empty cells c and c' is found, we perform a "union" operation using c and c' on the structure. After all the edges have been processed like this, the final CCs can be easily determined by issuing a "find" operation on every non-empty cell. In theory, this approach entails $O(n \cdot \alpha(n))$ time, where $\alpha(n)$ is the inverse of the Ackermann which is extremely slow growing such that $\alpha(n)$ is very small for all practical n.

  An advantage of this approach is that, it avoids a large amount of edge detection that was needed in [Gan and Tao 2015]. Before, such detection was performed for each pair of non-empty cells c and c' that were ϵ-neighbors of each other. Now, we can safely skip the detection if these cells are already found to be in the same CC.

Characteristics of the KDD'96 Algorithm. As mentioned in Section 1.1, the running time of the algorithm in [Ester et al. 1996] is determined by the total cost of n region queries, each of which retrieves $B(p, \epsilon)$ for each $p \in P$. Our hardness result in Theorem 3.1 implies that, even if *each* $B(p, \epsilon)$ returns just p itself, the cost of all n queries must still sum up to $\Omega(n^{4/3})$ for a hard dataset.

As reasonably argued by [Ester et al. 1996], on practical data, the cost of a region query $B(p, \epsilon)$ depends on how many points are in $B(p, \epsilon)$. The KDD'96 algorithm may have acceptable efficiency when ϵ is small such that the total number of points returned by all the region queries is near linear.

Such a value of ϵ, however, may not exist when the clusters have varying densities. Consider the example in Figure 12 where there are three clusters. Suppose that $MinPts = 4$. To discover the sparsest cluster on the left, ϵ needs to be at least the radius of the circles illustrated. For each point p from the right (i.e., the densest) cluster, however, the $B(p, \epsilon)$ under such an ϵ covers a big fraction of the cluster. On

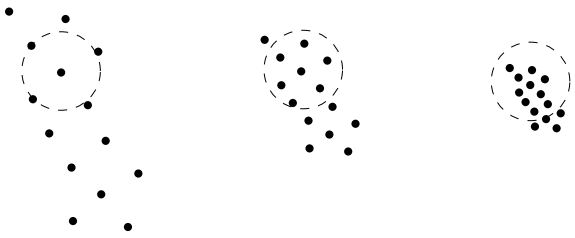


Fig. 12. A small ϵ for the left cluster is large for the other two clusters

this dataset, therefore, the algorithm of [Ester et al. 1996] either does not discover all three clusters, or must do so with expensive cost.

A Comparison. The preceding discussion suggests that the relative superiority between the KDD'96 algorithm and our proposed ρ-approximate algorithm depends primarily on two factors: (i) whether the cluster densities are similar or varying, and (ii) whether the value of ϵ is small or large. For a dataset with varying-density clusters, our algorithm is expected to perform better because, as explained, a good ϵ that finds all clusters must be relatively large for the dense clusters, forcing the KDD'96 algorithm to entail high cost on those clusters.

For a dataset with similar-density clusters, the KDD'96 algorithm can be faster when ϵ is sufficiently small. In fact, our empirical experience indicates a pattern: when the ρ-approximate algorithm is slower, the grid T it imposes on $\mathbb{R}^d$ has $\Omega(n)$ non-empty cells—more specifically, we observe that the cutoff threshold is roughly $n/\sqrt{2}$ cells, regardless of d. This makes sense because, in such a case, most non-empty cells have very few points (e.g., one or two), thus the extra overhead of creating and processing the grid no longer pays off.

The above observations will be verified in the next section.

7. EXPERIMENTS

The philosophy of the following experiments differs from that in the short version [Gan and Tao 2015]. Specifically, [Gan and Tao 2015] treated DBSCAN clustering as a computer science problem, and aimed to demonstrate the quadratic nature of the previous DBSCAN algorithms for $d \geq 3$. In this work, we regard DBSCAN as an *application*, and will focus on parameter values that are more important in practice.

All the experiments were run on a machine equipped with 3.4GHz CPU and 16 GB memory. The operating system was Linux (Ubuntu 14.04). All the programs were coded in C++, and compiled using g++ with -o3 turned on.

Section 7.1 describes the datasets in our experimentation, after which Section 7.2 seeks parameter values that lead to meaningful clusters on those data. The evaluation of the proposed techniques will then proceed in three parts. First, Section 7.3 assesses the clustering precision of ρ-approximate DBSCAN. Section 7.4 demonstrates the efficiency gain achieved by our approximation algorithm compared to exact DBSCAN in dimensionality $d \geq 3$. Finally, Section 7.5 examines the performance of exact DBSCAN algorithms for $d = 2$.

7.1. Datasets

In all datasets, the underlying data space had a normalized integer domain of $[0, 10^5]$ for every dimension. We deployed both synthetic and real datasets whose details are explained next.

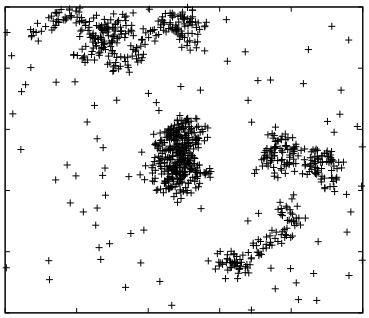


Fig. 13. A 2D seed spreader dataset

Synthetic: Seed Spreader (SS). A synthetic dataset was generated in a "random walk with restart" fashion. First, fix the dimensionality d, take the target cardinality n, a *restart probability* $\rho_{restart}$, and a *noise percentage* ρ_{noise}. Then, we simulate a *seed spreader* that moves about in the space, and spits out data points around its current location. The spreader carries a *local counter* such that whenever the counter reaches 0, the spreader moves a distance of r_{shift} towards a random direction, after which the counter is reset to c_{reset}. The spreader works in *steps*. In each step, (i) with probability $\rho_{restart}$, the spreader *restarts*, by jumping to a random location in the data space, and resetting its counter to c_{reset}; (ii) no matter if a restart has happened, the spreader produces a point uniformly at random in the ball centered at its current location with radius $r_{vincinity}$, after which the local counter decreases by 1. Intuitively, every time a restart happens, the spreader begins to generate a new cluster. In the first step, a restart is forced so as to put the spreader at a random location. We repeat in total $n(1-\rho_{noise})$ steps, which generate the same number of points. Finally, we add $n \cdot \rho_{noise}$ noise points, each of which is uniformly distributed in the whole space.

Figure 13 shows a small 2D dataset which was generated with $n = 1000$ and 4 restarts; the dataset will be used for visualization. The other experiments used larger datasets created with $c_{reset} = 100$, $\rho_{noise} = 1/10^4$, $\rho_{restart} = 10/(n(1-\rho_{noise}))$. In expectation, around 10 restarts occur in the generation of a dataset. The values of $r_{vincinity}$ and r_{shift} were set in two different ways to produce clusters with either similar or varying densities:

— *Similar-density* dataset: Namely, the clusters have roughly the same density. Such a dataset was obtained by fixing $r_{vincinity} = 100$ and $r_{shift} = 50d$.

— *Varying-density* dataset: Namely, the clusters have different densities. Such a dataset was obtained by setting $r_{vincinity} = 100 \cdot ((i \mod 10) + 1)$ and $r_{shift} = r_{vincinity} \cdot d/2$, where i equals the number of restarts that have taken place (at

Table I. Parameter values (defaults are in bold)

parameter	values
n (synthetic)	100k, 0.5m, 1m, **2m**, 5m, 10m
d (synthetic)	2, 3, **5**, 7
ϵ	from 100 (or 40 for $d = 2$) to 5000 (each dataset has its own default)
$MinPts$	10, 20, 40, 60, 100 (each dataset has its own default)
ρ	**0.001**, 0.01, 0.02, ..., 0.1

Table II. Cluster quality under different $(MinPts, \epsilon)$: SS similar density

	$MinPts = 10$			$MinPts = 100$		
ϵ	CV Index	# clusters	# noise pts	CV Index	# clusters	# noise pts
40	0.978	10	325	0.555	230	309224
60	0.994	9	197	0.577	72	33489
80	0.994	9	197	0.994	9	506
100	0.994	9	197	0.994	9	197
200	0.994	9	197	0.994	9	197

(a) *SS-simden-2D*

	$MinPts = 10$			$MinPts = 100$		
ϵ	CV Index	# clusters	# noise pts	CV Index	# clusters	# noise pts
100	0.996	14	200	0.205	240	467
200	0.996	14	200	0.996	14	200
400	0.996	14	200	0.996	14	200
800	0.996	14	200	0.996	14	200
1000	0.996	14	200	0.996	14	200

(b) *SS-simden-3D*

	$MinPts = 10$			$MinPts = 100$		
ϵ	CV Index	# clusters	# noise pts	CV Index	# clusters	# noise pts
100	0.102	4721	219	0.583	19057	632
200	0.996	13	200	0.996	13	241
400	0.996	13	200	0.996	13	200
800	0.996	13	200	0.996	13	200
1000	0.996	13	200	0.996	13	200

(c) *SS-simden-5D*

	$MinPts = 10$			$MinPts = 100$		
ϵ	CV Index	# clusters	# noise pts	CV Index	# clusters	# noise pts
100	0.588	19824	215	0.705	19822	1000
200	0.403	14988	215	0.403	14976	998
400	0.992	17	200	0.992	17	200
800	0.984	17	200	0.984	17	200
1000	0.980	17	200	0.980	17	200

(d) *SS-simden-7D*

the beginning $i = 0$). Note that the "modulo 10" ensures that there are at most 10 different cluster densities.

The value of n ranged from 100k to 10 million, while d from 2 to 7; see Table I. Hereafter, by *SS-simden-*d*D*, we refer to a d-dimensional similar-density dataset (the default cardinality is 2m), while by *SS-varden-*d*D*, we refer to a d-dimensional varying-density dataset (same default on cardinality).

Real. Three real datasets were employed in our experimentation:

— The first one, *PAMAP2*, is a 4-dimensional dataset with cardinality 3,850,505, obtained by taking the first 4 principle components of a PCA on the PAMAP2 database [Reiss and Stricker 2012] from the UCI machine learning archive [Bache and Lichman 2013].

— The second one, *Farm*, is a 5-dimensional dataset with cardinality 3,627,086, which contains the VZ-features [Varma and Zisserman 2003] of a satellite image of a farm in Saudi Arabia[4]. It is worth noting that VZ-feature clustering is a common approach to perform color segmentation of an image [Varma and Zisserman 2003].

— The third one, *Household*, is a 7-dimensional dataset with cardinality 2,049,280, which includes all the attributes of the Household database again from the UCI archive [Bache and Lichman 2013] except the temporal columns *date* and *time*. Points in the original database with missing coordinates were removed.

[4]*http://www.satimagingcorp.com/gallery/ikonos/ikonos-tadco-farms-saudi-arabia*

Table III. Cluster quality under different $(MinPts, \epsilon)$: SS varying density

	$MinPts = 10$			$MinPts = 100$		
ϵ	CV Index	# clusters	# noise pts	CV Index	# clusters	# noise pts
100	0.480	1294	50904	0.457	164	774095
200	0.574	70	2830	0.584	153	250018
400	0.946	6	161	0.836	21	18383
800	0.904	6	154	0.939	6	154
1000	0.887	6	153	0.905	6	153

(a) *SS-varden-2D*

	$MinPts = 10$			$MinPts = 100$		
ϵ	CV Index	# clusters	# noise pts	CV Index	# clusters	# noise pts
100	0.321	1031	577830	0.055	114	1358330
200	0.698	1989	317759	0.403	100	600273
400	0.864	573	23860	0.751	91	383122
800	0.917	11	195	0.908	91	50711
1000	0.904	11	194	0.884	27	236

(b) *SS-varden-3D*

	$MinPts = 10$			$MinPts = 100$		
ϵ	CV Index	# clusters	# noise pts	CV Index	# clusters	# noise pts
400	0.244	5880	267914	0.523	10160	568393
800	0.755	286	200	0.858	4540	432
1000	0.952	12	200	0.903	1667	357
2000	0.980	8	200	0.980	8	200
3000	0.980	8	200	0.980	8	200

(c) *SS-varden-5D*

	$MinPts = 10$			$MinPts = 100$		
ϵ	CV Index	# clusters	# noise pts	CV Index	# clusters	# noise pts
400	0.423	7646	801947	0.450	6550	837575
800	0.780	9224	10167	0.686	5050	425229
1000	0.804	7897	200	0.860	8054	506
2000	0.781	1045	200	0.781	1044	400
3000	0.949	13	200	0.949	13	200
4000	0.949	13	200	0.949	13	200

(d) *SS-varden-7D*

7.2. Characteristics of the Datasets

This subsection aims to study the clusters in each dataset under different parameters, and thereby, decide the values of $MinPts$ and ϵ suitable for the subsequent efficiency experiments.

Clustering Validation Index. We resorted to a method called *clustering validation* (CV) [Moulavi et al. 2014] whose objective is to quantify the quality of clustering using a real value. In general, a set of good density-based clusters should have two properties: first, the points in a cluster should be "tightly" connected; second, any two points belonging to different clusters should have a large distance. To quantify the first property for a cluster C, we compute a *Euclidean minimum spanning tree* (EMST) on the set of core points in C, and then, define $DSC(C)$ as the maximum weight of the edges in the EMST. "DSC" stands for *density sparseness of a cluster*, a term used by [Moulavi et al. 2014]. Intuitively, the EMST is a "backbone" of C such that if C is tightly connected, $DSC(C)$ ought to be small. Note that the border points of C are excluded because they are not required to have a dense vicinity. To quantify the second property, define $DSPC(C_i, C_j)$ between two clusters C_i and C_j as

$$\min\{dist(p_1, p_2) \mid p_1 \in C_1 \text{ and } p_2 \in C_2 \text{ are core points}\}$$

where "DSPC" stands for *density separation for a pair of clusters* [Moulavi et al. 2014]. Let $\mathscr{C} = \{C_1, C_2, ..., C_t\}$ (where $t \geq 2$) be a set of clusters returned by an algorithm. For

each C_i, we define (following [Moulavi et al. 2014]):

$$V_{\mathscr{C}}(C_i) = \frac{(\min_{1\leq j\leq t, j\neq i} DSPC(C_i, C_j)) - DSC(C_i)}{\max\left\{DSC(C_i), \min_{1\leq j\leq t, j\neq i} DSPC(C_i, C_j)\right\}}$$

Then, the *CV index* of $\mathscr{C}$ is calculated as in [Moulavi et al. 2014]:

$$\sum_{i=1}^{t} \frac{|C_i|}{n} V_{\mathscr{C}}(C_i)$$

where n is the size of the dataset. A higher validity index indicates better quality of $\mathscr{C}$.

[Moulavi et al. 2014] computed $DSC(C_i)$ and $DSPC(C_i, C_j)$ differently, but their approach requires $O(n^2)$ time which is intolerably long for the values of n considered here. Our proposition follows the same rationale, admits faster implementation (EMST is well studied [Agarwal et al. 1991; Arya and Mount 2016]), and worked well in our experiments as shown below.

Influence of *MinPts* and ϵ on DBSCAN Clusters. For each dataset, we examined the quality of its clusters under different combinations of $MinPts$ and ϵ. For $MinPts$, we inspected values 10 and 100, while for ϵ, we inspected a wide range starting from $\epsilon = 40$ and 100 for $d = 2$ and $d \geq 3$, respectively. Only two values of $MinPts$ were considered because (i) either 10 or 100 worked well on the synthetic and real data deployed, and (ii) the number of combinations was already huge.

Table II presents some key statistics for *SS-simden-d**D*** datasets with $d = 2, 3, 5$ and 7, while Table III shows the same statistics for *SS-varden-d**D***. Remember that the cardinality here is n = 2m, implying that there should be around 200 noise points. The number of intended clusters should not exceed the number of restarts whose expectation is 10. But the former number can be smaller, because the seed spreader may not necessarily create a new cluster after a restart, if it happens to jump into the region of a cluster already generated.

Both $MinPts = 10$ and 100, when coupled with an appropriate ϵ, were able to discover all the intended clusters—observe that the CV index stabilizes soon as ϵ increases. We set 10 as the default for $MinPts$ on the synthetic datasets, as it produced better clusters than 100 under most values of ϵ. Notice that, for varying-density datasets, ϵ needed to be larger to ensure good clustering quality (compared to similar-density datasets). This is due to the reason explained in Section 6 (c.f. Figure 12). The bold ϵ values in Tables II and III were chosen as the default for the corresponding datasets (they were essentially the smallest that gave good clusters).

Figure 14 plots the OPTICS diagrams[5] for *SS-simden-5D* and *SS-varden-5D*, obtained with $MinPts = 10$. In an OPTICS diagram [Ankerst et al. 1999], the data points are arranged into a sequence as given along the x-axis. The diagram shows the area beneath a function $f(x) : [1, n] \rightarrow \mathbb{R}$, where $f(x)$ can be understood roughly as follows: if p is the x-th point in the sequence, then $f(x)$ is the smallest ϵ value which (together with the chosen $MinPts$) puts p into some cluster—in other words, p remains as a noise point for $\epsilon < f(x)$. A higher/lower $f(x)$ indicates that p is in a denser/sparser area. The ordering of the sequence conveys important information: each "valley"—a subsequence of points between two "walls"–corresponds to a cluster. Furthermore, the points of this valley will remain in a cluster under any ϵ greater than the maximum $f(x)$ value of those points.

[5]The OPTICS algorithm [Ankerst et al. 1999] requires a parameter called $maxEps$, which was set to 10000 in our expeirments.

Table IV. Cluster quality under different $(MinPts, \epsilon)$: real data

	$MinPts = 10$			$MinPts = 100$		
ϵ	CV Index	# clusters	# noise pts	CV Index	# clusters	# noise pts
100	0.174	6585	2578125	0.103	478	3369657
200	0.222	17622	1890108	0.210	818	2800524
400	0.092	11408	620932	0.226	1129	2396808
500	0.059	5907	406247	0.233	1238	2097941
800	0.037	3121	215925	0.099	756	949167
1000	0.032	2530	159570	0.078	483	594075
2000	0.033	549	28901	0.126	237	209236
3000	0.237	110	5840	0.302	100	75723
4000	0.106	30	1673	0.492	31	24595
5000	0.490	9	673	0.506	12	9060

(a) *PAMAP2*

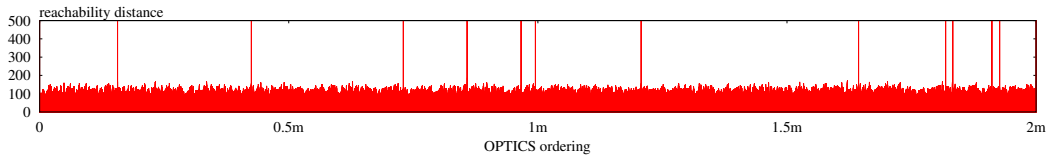
	$MinPts = 10$			$MinPts = 100$		
ϵ	CV Index	# clusters	# noise pts	CV Index	# clusters	# noise pts
100	0.002	925	3542419	0.001	3	3621494
200	0.005	3296	2473933	0.008	21	3404402
400	0.006	1420	1153340	0.191	13	1840989
700	0.004	962	514949	0.364	28	1039114
800	0.004	994	410432	0.198	18	859002
1000	0.005	689	273723	0.295	15	594462
2000	0.002	217	46616	0.120	13	181628
3000	0.001	55	15096	0.131	6	62746
4000	0.058	35	8100	0.764	3	24791
5000	0.024	27	5298	0.157	6	12890

(b) *Farm*

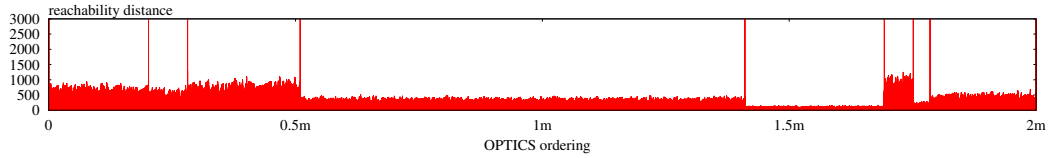
	$MinPts = 10$			$MinPts = 100$		
ϵ	CV Index	# clusters	# noise pts	CV Index	# clusters	# noise pts
100	0.057	3342	1702377	0.026	54	1944226
200	0.114	5036	1314498	0.074	87	1829873
400	0.085	4802	911088	0.088	165	1598323
800	0.048	2148	490634	0.257	47	974566
1000	0.045	1800	404306	0.227	55	829398
2000	0.129	601	139483	0.416	28	327508
3000	0.074	447	73757	0.241	48	193502
4000	0.007	195	34585	0.565	10	112231
5000	0.015	131	18059	0.649	8	68943

(c) *Household*

Figure 14a has 13 valleys, matching the 13 clusters found by $\epsilon = 200$. Notice that the points in these valleys have roughly the same $f(x)$ values (i.e., similar density).

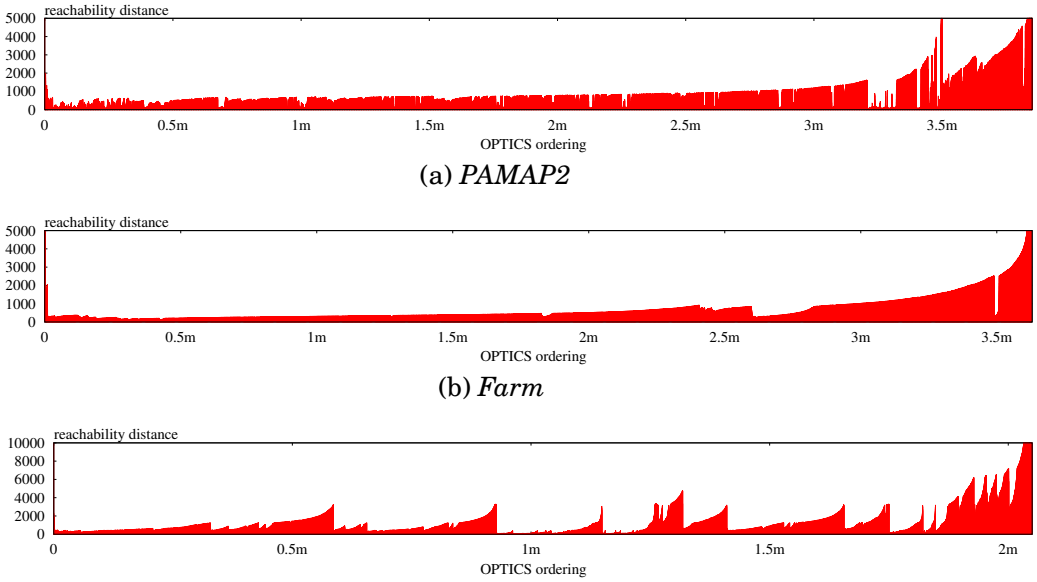


(a) *SS-simden-5D*



(b) *SS-varden-5D*

Fig. 14. Optics diagrams for 5D synthetic data



(a) *PAMAP2*

(b) *Farm*

(c) *Household*

Fig. 15. Optics diagrams for real datasets

Figure 14b, on the other hand, has 8 valleys, namely, the 8 clusters found by $\epsilon = 2000$. Points in various valleys can have very different $f(x)$ values (i.e., varying density). The OPTICS diagrams for the other synthetic datasets are omitted because they illustrate analogous observations about the composition of clusters.

Next, we turned to the real datasets. Table IV gives the statistics for *PAMAP2*, *Farm*, and *Household*. The CV indexes are much lower (than those of synthetic data), indicating that the clusters in these datasets are less obvious. For further analysis, we chose $MinPts = 100$ as the default (because it worked much better than $MinPts = 10$), using which Figure 15 presents the OPTICS diagrams for the real datasets, while Table V details the sizes (unit: 1000) of the 10 largest clusters under each ϵ value in Table IV. By combining all these data, we make the following observations:

— *PAMAP2:* From Figure 15a, we can see that this dataset contains numerous "tiny valleys", which explains the large number of clusters as shown in Table IV(a). An interesting ϵ value would be 500, which discovers most of those valleys. Notice from Table IV(a) that the CV index is relatively high at $\epsilon = 500$. It is worth mentioning that, although $\epsilon = 4000$ and 5000 have even higher CV indexes, almost all the valleys disappear at these ϵ values, leaving only 2 major clusters one of which contains over 90% of the points.

— *Farm:* There are two clusters in the dataset. The first one is the valley between 2.6m and 2.8m on the x-axis of Figure 15b, and the second one is the small dagger-shape valley at 3.5m. The best value of ϵ that discovers both clusters lies around 700—they are the 2nd and 4th largest clusters at the row of $\epsilon = 700$ in Table V(b). Once again, there exist some large values ϵ such as 4000 that give high CV indexes, but assign almost all the points (over 99% for $\epsilon = 4000$) into one cluster.

— *Household:* This is the "most clustered" real dataset of the three. It is evident that $\epsilon = 2000$ is an interesting value: it has a relatively high CV index (see Table IV(c)),

Table V. Sizes of the 10 largest clusters: real data (unit: 10^3)

ϵ	1st	2nd	3rd	4th	5th	6th	7th	8th	9th	10th
100	18.8	16.9	11.9	10.2	9.72	8.10	7.00	5.57	5.54	5.54
200	25.9	21.6	20.0	18.9	18.8	18.2	18.2	17.4	16.9	15.5
400	66.9	54.9	39.1	35.2	29.1	28.1	23.8	21.7	20.0	19.4
500	124	114	55.5	53.4	47.0	42.9	41.2	29.3	29.2	20.0
800	2219	41.5	37.3	26.7	20.5	19.2	19.0	17.4	15.3	13.9
1000	2794	116	20.5	16.4	13.1	9.12	9.08	8.69	7.65	7.10
2000	3409	78.0	18.2	13.3	9.65	9.57	6.60	6.60	5.11	5.04
3000	3470	239	18.5	11.8	2.03	1.90	1.83	1.21	1.20	1.09
4000	3495	315	2.03	1.86	1.84	0.965	0.786	0.735	0.698	0.687
5000	3497	339	1.85	0.977	0.553	0.551	0.328	0.328	0.217	0.216

(a) *PAMAP2*

ϵ	1st	2nd	3rd	4th	5th	6th	7th	8th	9th	10th
100	4.10	1.34	0.150							
200	195	18.0	3.93	0.868	0.717	0.647	0.529	0.393	0.391	0.303
400	1604	129	37.3	11.1	1.75	0.713	0.408	0.327	0.265	0.226
700	2282	218	44.1	17.0	10.4	4.27	3.59	1.12	1.09	0.863
800	2358	381	17.4	6.28	1.34	0.921	0.859	0.545	0.528	0.414
1000	3009	18.0	1.47	0.740	0.718	0.446	0.422	0.332	0.287	0.214
2000	3418	18.8	2.79	1.88	1.45	0.386	0.374	0.230	0.186	0.165
3000	3562	0.951	0.681	0.350	0.190	0.177				
4000	3600	1.08	0.470							
5000	3611	1.18	0.537	0.273	0.130	0.114				

(b) *Farm*

ϵ	1st	2nd	3rd	4th	5th	6th	7th	8th	9th	10th
100	37.8	19.8	16.2	12.3	3.84	3.31	2.33	0.529	0.525	0.505
200	47.8	30.9	24.8	24.5	20.3	15.2	7.79	6.91	5.82	4.39
400	52.6	39.9	34.2	31.9	27.3	25.9	21.1	18.8	15.9	15.8
800	274	193	117	97.3	70.2	48.0	33.9	33.6	27.0	24.8
1000	294	198	158	99.7	94.5	81.7	51.6	30.8	27.2	25.0
2000	560	320	222	220	110	75.2	71.4	68.2	25.1	25.0
3000	586	337	243	221	111	91.8	85.0	69.7	26.6	26.1
4000	1312	575	17.0	10.9	9.98	7.07	3.71	0.381	0.197	0.100
5000	1918	22.2	14.9	13.3	11.2	0.299	0.101	0.100		

(c) *Household*

Note: interesting clusters are underlined.

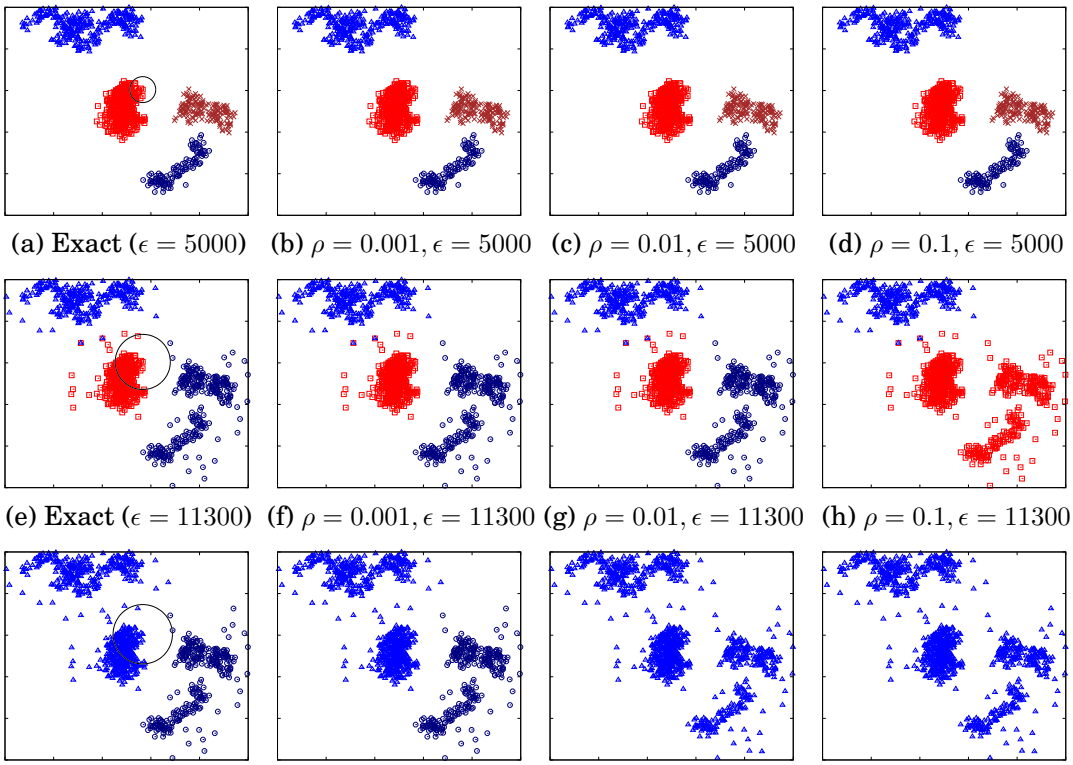
and discovers most of the important valleys in Figure 15, whose clusters are quite sizable as shown in Table V(c).

Based on the above discussion, we set the default ϵ of each real dataset to the bold values in Table V.

7.3. Approximation Quality

In this subsection, we evaluate the quality of the clusters returned by the proposed ρ-approximate DBSCAN algorithm.

2D Visualization. To show directly the effects of approximation, we take the 2D dataset in Figure 13 as the input (note that the cardinality was deliberately chosen to be small to facilitate visualization), and fixed $MinPts = 20$. Figure 16a demonstrates the 4 clusters found by exact DBSCAN with $\epsilon = 5000$ (which is the radius of the circle shown). The points of each cluster are depicted with the same color and marker. Figures 16b, 16c, and 16d present the clusters found by our ρ-approximate DBSCAN when ρ equals 0.001, 0.01, and 0.1, respectively. In all cases, ρ-approximate DBSCAN returned exactly the same clusters as DBSCAN.



(a) Exact ($\epsilon = 5000$) (b) $\rho = 0.001, \epsilon = 5000$ (c) $\rho = 0.01, \epsilon = 5000$ (d) $\rho = 0.1, \epsilon = 5000$

(e) Exact ($\epsilon = 11300$) (f) $\rho = 0.001, \epsilon = 11300$ (g) $\rho = 0.01, \epsilon = 11300$ (h) $\rho = 0.1, \epsilon = 11300$

(i) Exact ($\epsilon = 12200$) (j) $\rho = 0.001, \epsilon = 12200$ (k) $\rho = 0.01, \epsilon = 12200$ (l) $\rho = 0.1, \epsilon = 12200$

Fig. 16. Comparison of the clusters found by exact DBSCAN and ρ-approximate DBSCAN

Making things more interesting, in Figure 16e, we increased ϵ to 11300 (again, ϵ is the radius of the circle shown). This time, DBSCAN found 3 clusters (note that 2 clusters in Figure 16a have merged). Figures 16f, 16g, and 16h give the clusters of ρ-approximate DBSCAN for $\rho = 0.001$, 0.01, and 0.1, respectively. Once again, the clusters of $\rho = 0.001$ and 0.01 are exactly the same as DBSCAN. However, 0.1-approximate DBSCAN returned only 2 clusters. This can be understood by observing that the circle in Figure 16e almost touched a point from a different cluster. In fact, it will, once ϵ increases by 10%, which explains why 0.1-approximate DBSCAN produced different results.

Then we pushed ϵ even further to 12200 so that DBSCAN yielded 2 clusters as shown in Figure 16i. Figures 16j, 16k, and 16l illustrate the clusters of ρ-approximate DBSCAN for $\rho = 0.001$, 0.01, and 0.1, respectively. Here, both $\rho = 0.01$ and 0.1 had given up, but $\rho = 0.001$ still churned out exactly the same clusters as DBSCAN.

Surprised by $\rho = 0.01$ not working, we examined the reason behind its failure. It turned out that 12200 was extremely close to the "boundary ϵ" for DBSCAN to output 2 clusters. Specifically, as soon as ϵ grew up to 12203, the exact DBSCAN would return only a single cluster. Actually, this can be seen from Figure 16i—note how close the circle is to the point from the right cluster! In other words, 12200 is in fact an "unstable" value for ϵ.

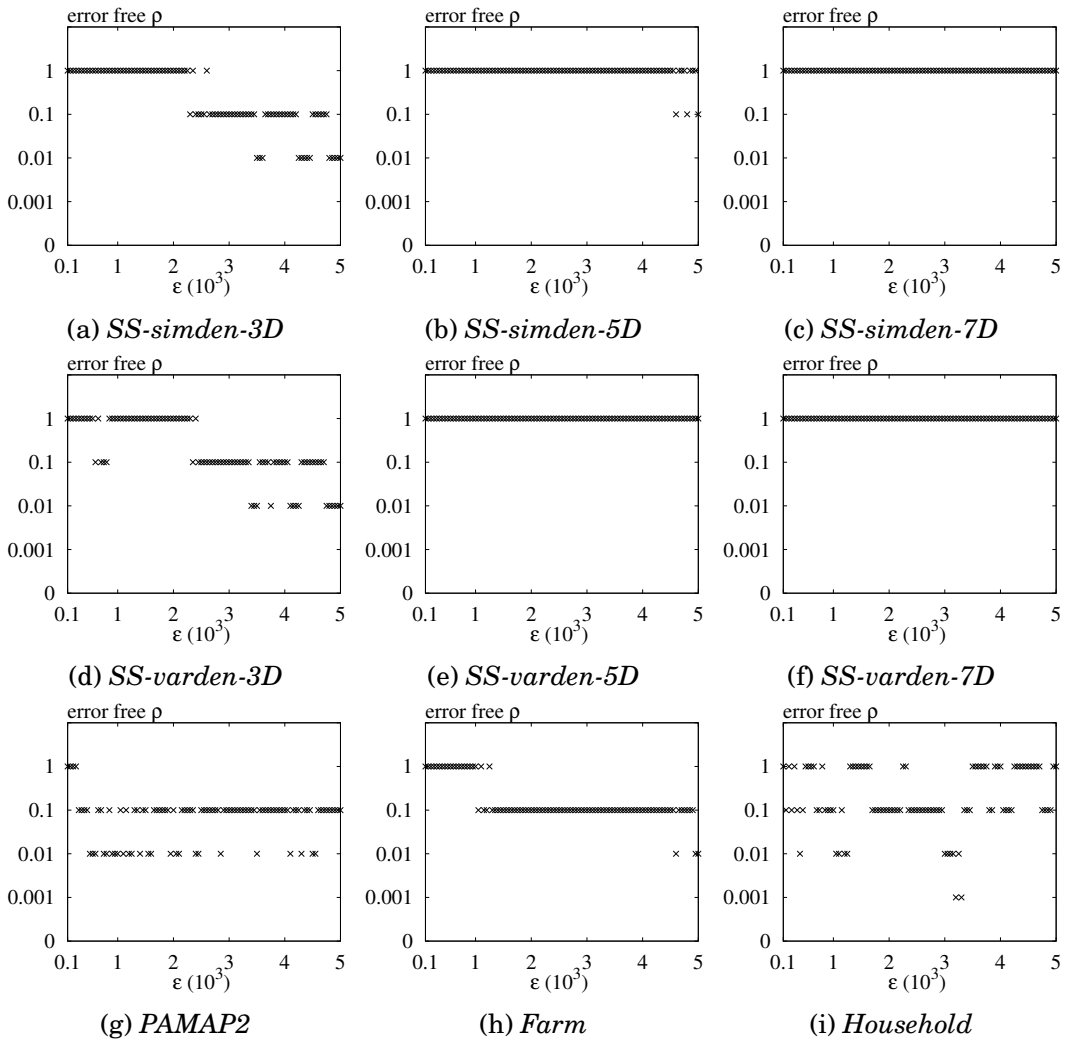


Fig. 17. Largest ρ in $\{0.001, 0.01, 0.1, 1\}$ for our ρ-approximate DBSCAN algorithm to return the same results as precise DBSCAN

Dimensionalities $d \geq 3$. We deployed the same methodology to study the approximation quality in higher dimensional space. Specifically, for a dataset and a value of ϵ, we varied ρ among $0.001, 0.01, 0.1$ and 1 to identify the highest *error-free* ρ under which our ρ-approximate algorithm returned exactly the same result as precise DBSCAN. Figure 17 plots the highest error-free ρ for various datasets when ϵ grew from 100 to 5000. For example, by the fact that in Figure 17a the (highest) error-free ρ is 1 at $\epsilon = 100$, one should understand that our approximate algorithm also returned the exact clusters at $\rho = 0.001, 0.01$, and 0.1 at this ϵ. Notice that in nearly all the cases, 0.01-approximation already guaranteed the precise results.

As shown in the next subsection, our current implementation was fast enough on all the tested datasets even when ρ was set to 0.001. We therefore recommend this value for practical use, which was also the default ρ in the following experiments.

Recall that, by the sandwich theorem (Theorem 4.3), the result of 0.001-approximate DBSCAN must fall between the results of DBSCAN with ϵ and 1.001ϵ, respectively. Hence, if 0.001-approximate DBSCAN differs from DBSCAN in the outcome, it means that the (exact) DBSCAN clusters must have changed within the parameter range $[\epsilon, 1.001\epsilon]$.

7.4. Computational Efficiency for $d \geq 3$

We now proceed to inspect the running time of DBSCAN clustering in dimensionality $d \geq 3$ using four algorithms:

— *KDD96* [Ester et al. 1996]: the original DBSCAN algorithm in [Ester et al. 1996], which deployed a memory R-tree whose leaf capacity was 12 and internal fanout was 4 (the same values were used in the R-trees deployed by the other methods as well).
— *CIT08* [Mahran and Mahar 2008]: the state of the art of exact DBSCAN, namely, the fastest existing algorithm able to produce the same DBSCAN result as *KDD96*.
— *SkLearn (http://scikit-learn.org/stable)*: the DBSCAN implementation in the popular machine learning tool-kit *scikit-learn*. One should note that, judging from its website, *SkLearn* was implemented in Cython with its wrapper in Python.
— *OurExact*: the exact DBSCAN algorithm we developed in Theorem 3.3, except that we did not use the BCP algorithm in Lemma 2.5; instead, we indexed the core points of each cell with an R-tree, and solved the BCP problem between two cells by repetitive nearest neighbor search [Hjaltason and Samet 1999] using the R-tree.
— *OurApprox*: the ρ-approximate DBSCAN algorithm we proposed in Theorem 4.6. Our implementation has improved the one in the short version [Gan and Tao 2015] by incorporating new heuristics (see Section 6). In some experiments, we will also include the results of the old implementation—referred to as *OurApprox-SIG*—to demonstrate the effectiveness of those heuristics.

Each parameter was set to its default value unless otherwise stated. Remember that the default values of $MinPts$ and ϵ may be different for various datasets; see Section 7.2.

Influence of ϵ. The first experiment aimed to understand the behavior of each method under the influence of ϵ. Figure 18 plots the running time as a function of ϵ, when this parameter varied from 100 to 5000 (we refer the reader to [Gan and Tao 2015] for running time comparison under $\epsilon > 5000$).

KDD96 and *CIT08* retrieve, for each data point p, all the points in $B(p, \epsilon)$. As discussed in Section 6, these methods may be efficient when ϵ is small, but their performance deteriorates rapidly as ϵ increases. This can be observed from the results in Figure 18. *OurExact* and *OurApprox* (particularly the latter) offered either competitive or significantly better efficiency at a vast majority of ϵ values. Such a property is useful in tuning this crucial parameter in reality. Specifically, it enables a user to try out a large number of values in a wide spectrum, without having to worry about the possibly prohibitive cost—note that *KDD96* and *CIT08* demanded over 1000 seconds at many values of ϵ that have been found to be interesting in Section 7.2.

The performance of *OurApprox-SIG* is reported in the first synthetic dataset *SS-simden-3D* and the first real dataset *PAMAP2*. There are two main observations here. First, the proposed heuristics allowed the new implementation to outperform the one at SIGMOD quite significantly. Second, the improvement *diminished* as ϵ increased. This happens because for a larger ϵ, the side length of a cell (in the grid

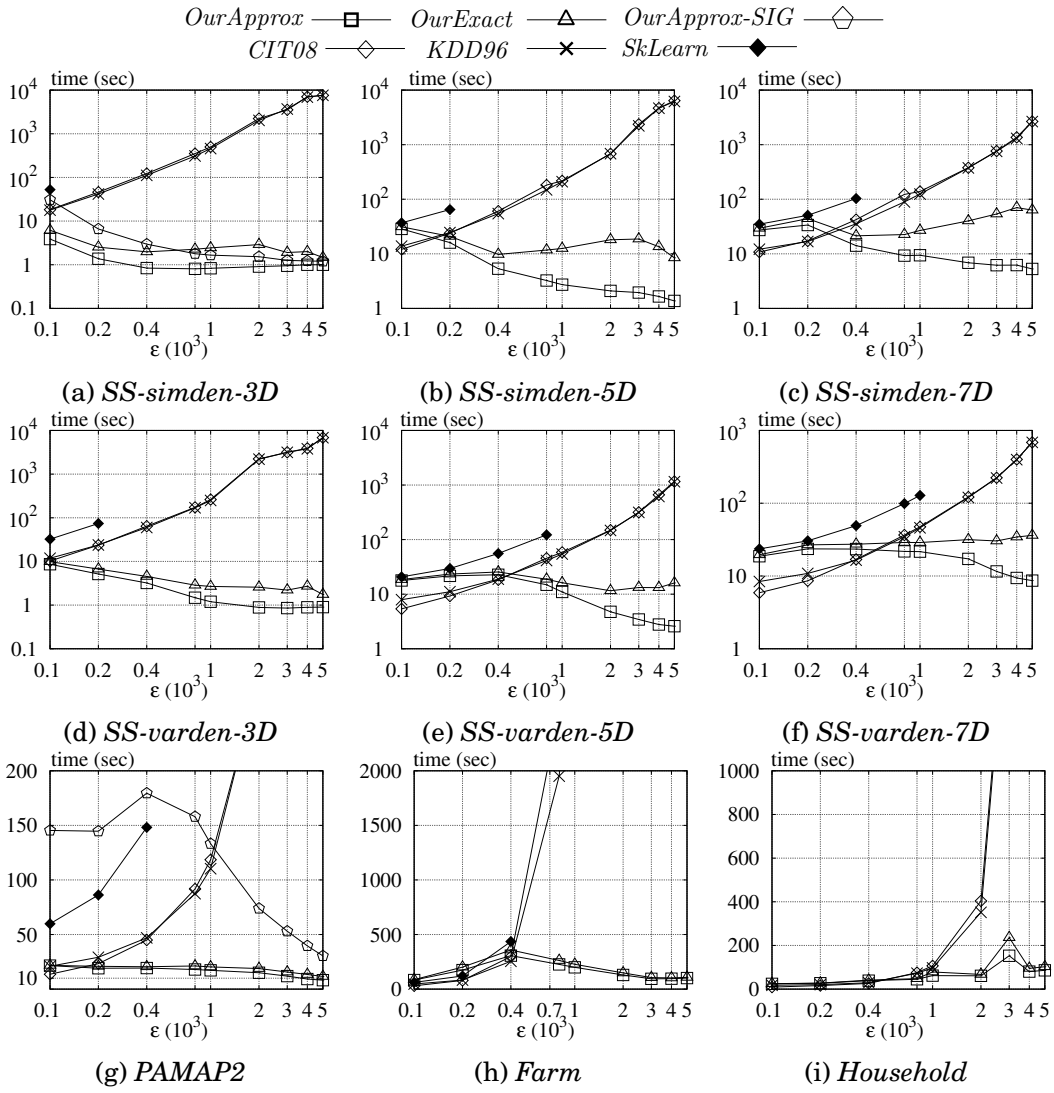


Fig. 18. Running time vs. ϵ ($d \geq 3$)

imposed by our algorithm) increases, which *decreases* the number of non-empty cells. In that scenario, the graph G (see Section 4.4) has only a small number of edges, thus making even a less delicate implementation (such as *OurApprox-SIG*) reasonably fast. In other words, the importance of the heuristics is reflected chiefly in *small* ϵ. To avoid clattering the diagrams, *OurApprox-SIG* is omitted from the other datasets, but similar patterns were observed.

Scalability with n. The next experiment examined how each method scales with the number n objects. For this purpose, we used synthetic SS datasets by varying n from 100k to 10m, using the default ϵ and $MinPts$ values in Tables II and III. The results are presented in Figure 19—note that the y-axis is in log scale. If *SkLearn* does not have a result at a value of n, it ran out of memory on our machine (same convention adopted

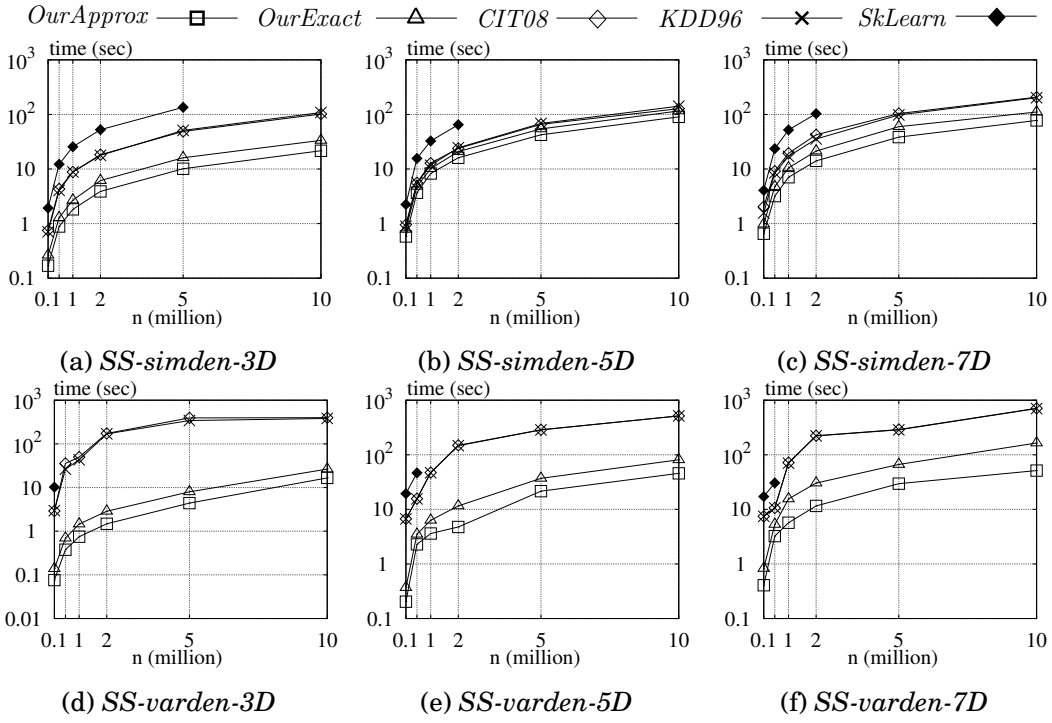


Fig. 19. Running time vs. n ($d \geq 3$)

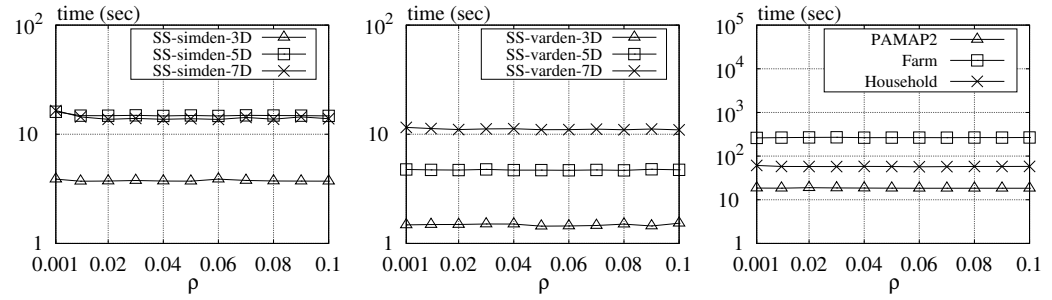


Fig. 20. Running time vs. ρ ($d \geq 3$)

in the rest of the evaluation). *KDD96* and *CIT08* had competitive performance on similar-density datasets, but they were considerably slower (by a factor over an ordge of magnitude) than *OurApprox* and *OurExact* on varying-density data, confirming the analysis in Section 6.

Influence of ρ. Figure 20 shows the running time of *OurApprox* as ρ changed from 0.001 to 0.1. The algorithm did not appear sensitive to this parameter. This, at first glance, may look surprising, because the theoretical analysis in Section 4.3 implies that the running time should contain a multiplicative term $(1/\rho)^{d-1}$, as is the worst-case cost of an approximate range count query, which is in turn for detecting whether two cells in G (i.e., the grid our algorithm imposes) have an edge. There

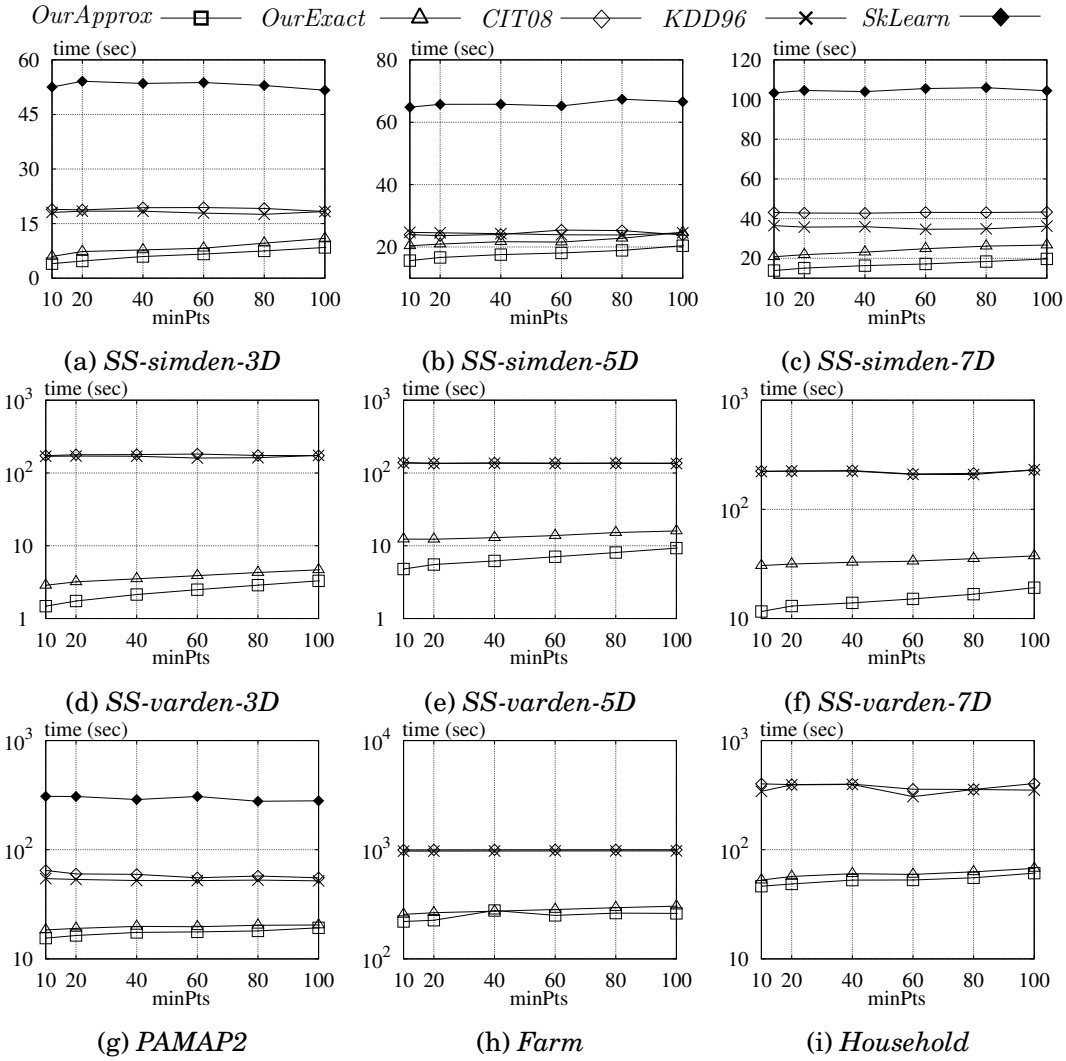
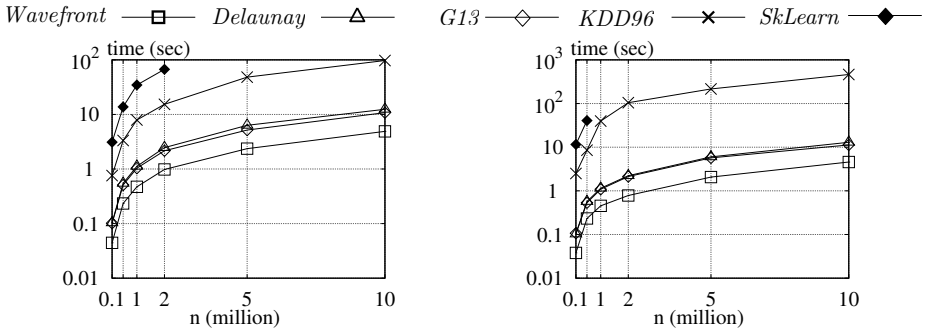


Fig. 21. Running time vs. $MinPts$ ($d \geq 3$)

are primarily two reasons why such a dramatic blow-up was not observed. First, the union-find heuristic explained in Section 6 significantly reduces the number of edges that need to be detected: once two cells are found to be in the same connected component, it is unnecessary to detect their edges. Second, even when an edge detection is indeed required, its cost is unlikely to reach $(1/\rho)^{d-1}$ because our algorithm for answering an approximate range count query often terminates without exploring the underlying quad-tree completely—recall that the algorithm can terminate as soon as it is certain whether the query answer is zero.

Influence of *MinPts*. The last set of experiments in this section measured the running time of each method when $MinPts$ increased from 10 to 100. The results are given in Figure 21. The impact of this parameter was limited, and did not change any of the observations made earlier.



(a) *SS-simden-2D* (b) *SS-varden-2D*

Fig. 22. Running time vs. n ($d = 2$)

7.5. Computational Efficiency for $d = 2$

In this subsection, we will focus on exact DBSCAN in 2D space, and compare the following algorithms:

— *KDD96* and *SkLearn*: As introduced at the beginning of Section 7.4.

— *G13* [Gunawan 2013]: The $O(n \log n)$ time algorithm by Gunawan, as reviewed in Section 2.2.

— *Delaunay*: Our algorithm as explained in Section 5.1, which runs in $O(n \log n)$ time.

— *Wavefront*: Our algorithm as in Theorem 5.2, assuming that the dataset has been bi-dimensionally sorted—recall that this is required to ensure the linear-time complexity of the algorithm.

Once again, each parameter was set to its default value (see Table I and Section 7.2) unless otherwise stated. All the experiments in this subsection were based on SS similar- and varying-density datasets.

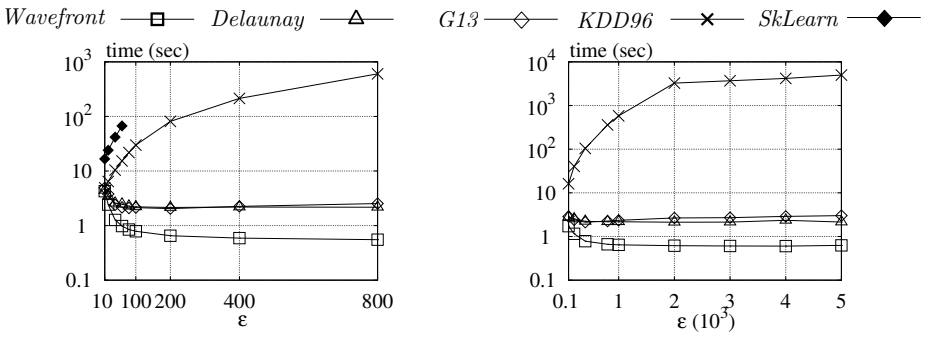
Results. In the experiment of Figure 22, we measured the running time of each algorithm as the cardinality n escalated from 100k to 10m. *Wavefront* consistently outperformed all the other methods, while *Delaunay* was observed to be comparable to *G13*. It is worth pointing out the vast difference between the running time here and that shown in Figure 19 for $d \geq 3$ (one can feel the difficulty gap of the DBSCAN problem between $d = 2$ and $d \geq 3$).

Next, we compared the running time of the five algorithms by varying ϵ. As shown in Figure 23, the cost of *Wavefront*, *Delaunay*, and *G13* actually improved as ϵ grew, whereas *KDD96* and *SkLearn* worsened. *Wavefront* was the overall winner by a wide margin.

Finally, we inspected the influence of $MinPts$ on the running time. The results are presented in Figure 24. In general, for a larger $MinPts$, *Wavefront*, *Delaunay*, and *G13* require a higher cost in labeling the data points as core or non-core points. The influence, however, is contained by the fact that this parameter is set as a small constant compared to the dataset size. The relative superiority of all the methods remained the same.

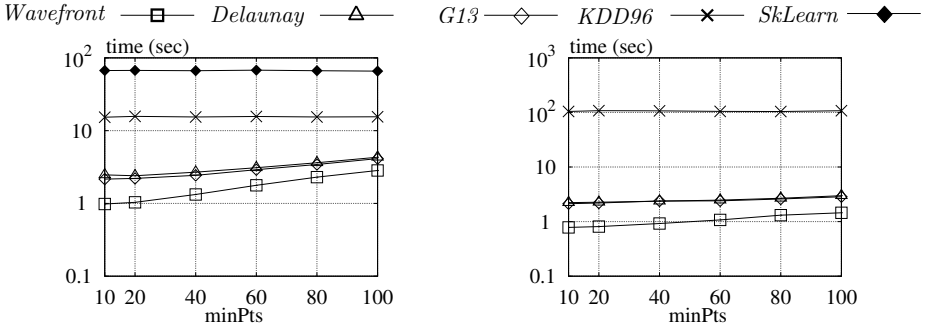
8. CONCLUSIONS

DBSCAN is an effective technique for density-based clustering, which is very extensively applied in data mining, machine learning, and databases. However, currently there has not been clear understanding on its theoretical computational



(a) *SS-simden-2D* (b) *SS-varden-2D*

Fig. 23. Running time vs. ϵ ($d = 2$)



(a) *SS-simden-2D* (b) *SS-varden-2D*

Fig. 24. Running time vs. $MinPts$ ($d = 2$)

hardness. All the existing algorithms suffer from a time complexity that is quadratic to the dataset size n when the dimensionality d is at least 3.

In this article, we show that, unless very significant breakthroughs (ones widely believed to be impossible) can be made in theoretical computer science, the DBSCAN problem requires $\Omega(n^{4/3})$ time to solve for $d \geq 3$ under the Euclidean distance. This excludes the possibility of finding an algorithm of near-linear running time, thus motivating the idea of computing approximate clusters. Towards that direction, we propose ρ-approximate DBSCAN, and prove both theoretical and experimentally that the new method has excellent guarantees both in the quality of cluster approximation and computational efficiency.

The exact DBSCAN problem in dimensionality $d = 2$ is known to be solvable in $O(n \log n)$ time. This article further enhances that understanding by showing how to settle the problem in $O(n)$ time, provided that the data points have already been pre-sorted on each dimension. In other words, coordinating sorting is in fact the hardest component of the 2D DBSCAN problem. The result immediately implies that, when all the coordinates are integers, the problem can be solved in $O(n \log \log n)$ time deterministically, or $O(n\sqrt{\log \log n})$ expected time randomly.

We close the article with a respectful remark. The objective of the article, as well as its short version [Gan and Tao 2015], is to understand the computational complexity of DBSCAN and how to bring down the complexity with approximation. The intention has never, by any means, been to argue against the significance of

DBSCAN—in contrary, there is no doubt that DBSCAN has proved to be a highly successful technique. In fact, even though many *algorithmic* aspects about this technique have been resolved in this article, from the *data mining* perspective, how to choose between exact DBSCAN (even implemented just as in the KDD96 algorithm) and our approximate DBSCAN is far from being conclusive. There are, for sure, datasets where a small ϵ value suffices, in which case exact DBSCAN may finish even faster than the approximate version. However, selecting the right parameters is seldom trivial in reality, and often requires multiple iterations of "trial and error". The proposed approximate algorithm has the advantage of being reasonably fast regardless of the parameters. This allows users to inspect the clusters under numerous parameter values in a (much) more efficient manner. With this said, we feel that the exact and approximate versions can comfortably co-exist with each other: the approximate algorithm serves nicely as a "filtering step" for the exact algorithm.

REFERENCES

Pankaj K. Agarwal, Herbert Edelsbrunner, and Otfried Schwarzkopf. 1991. Euclidean Minimum Spanning Trees and Bichromatic Closest Pairs. *Discrete & Computational Geometry* 6 (1991), 407–422.

Arne Andersson, Torben Hagerup, Stefan Nilsson, and Rajeev Raman. 1998. Sorting in Linear Time? *Journal of Computer and System Sciences (JCSS)* 57, 1 (1998), 74–93.

Mihael Ankerst, Markus M. Breunig, Hans-Peter Kriegel, and Jörg Sander. 1999. OPTICS: Ordering Points To Identify the Clustering Structure. In *Proceedings of ACM Management of Data (SIGMOD)*. 49–60.

Sunil Arya and David M. Mount. 2000. Approximate range searching. *Computational Geometry* 17, 3-4 (2000), 135–152.

Sunil Arya and David M. Mount. 2016. A Fast and Simple Algorithm for Computing Approximate Euclidean Minimum Spanning Trees. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 1220–1233.

K. Bache and M. Lichman. 2013. UCI Machine Learning Repository. (2013). http://archive.ics.uci.edu/ml

Christian Böhm, Karin Kailing, Peer Kröger, and Arthur Zimek. 2004. Computing Clusters of Correlation Connected Objects. In *Proceedings of ACM Management of Data (SIGMOD)*. 455–466.

B Borah and D K Bhattacharyya. 2004. An improved sampling-based DBSCAN for large spatial databases. In *Proceedings of Intelligent Sensing and Information Processing*. 92–96.

Prosenjit Bose, Anil Maheshwari, Pat Morin, Jason Morrison, Michiel H. M. Smid, and Jan Vahrenhold. 2007. Space-efficient geometric divide-and-conquer algorithms. *Computational Geometry* 37, 3 (2007), 209–227.

Vineet Chaoji, Mohammad Al Hasan, Saeed Salem, and Mohammed J. Zaki. 2008. SPARCL: Efficient and Effective Shape-Based Clustering. In *Proceedings of International Conference on Management of Data (ICDM)*. 93–102.

Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. 2008. *Computational Geometry: Algorithms and Applications* (3 ed.). Springer-Verlag.

Mark de Berg, Constantinos Tsirogiannis, and B. T. Wilkinson. 2015. Fast computation of categorical richness on raster data sets and related problems. 18:1–18:10.

Jeff Erickson. 1995. On the relative complexities of some geometric problems. In *Proceedings of the Canadian Conference on Computational Geometry (CCCG)*. 85–90.

Jeff Erickson. 1996. New Lower Bounds for Hopcroft's Problem. *Discrete & Computational Geometry* 16, 4 (1996), 389–418.

Martin Ester. 2013. Density-Based Clustering. In *Data Clustering: Algorithms and Applications*. 111–126.

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 ACM Knowledge Discovery and Data Mining (SIGKDD)*. 226–231.

Junhao Gan and Yufei Tao. 2015. DBSCAN Revisited: Mis-Claim, Un-Fixability, and Approximation. In *Proceedings of ACM Management of Data (SIGMOD)*. 519–530.

Ade Gunawan. 2013. *A Faster Algorithm for DBSCAN*. Master's thesis. Technische University Eindhoven.

Jiawei Han, Micheline Kamber, and Jian Pei. 2012. *Data Mining: Concepts and Techniques*. Morgan Kaufmann.

Yijie Han and Mikkel Thorup. 2002. Integer Sorting in 0(n sqrt (log log n)) Expected Time and Linear Space. In *Proceedings of Annual IEEE Symposium on Foundations of Computer Science (FOCS)*. 135–144.

G. R. Hjaltason and H. Samet. 1999. Distance Browsing in Spatial Databases. *ACM Transactions on Database Systems (TODS)* 24, 2 (1999), 265–318.

David G. Kirkpatrick and Stefan Reisch. 1984. Upper Bounds for Sorting Integers on Random Access Machines. *Theoretical Computer Science* 28 (1984), 263–276.

Matthias Klusch, Stefano Lodi, and Gianluca Moro. 2003. Distributed Clustering Based on Sampling Local Density Estimates. In *Proceedings of the International Joint Conference of Artificial Intelligence (IJCAI)*. 485–490.

Zhenhui Li, Bolin Ding, Jiawei Han, and Roland Kays. 2010. Swarm: Mining Relaxed Temporal Moving Object Clusters. *Proceedings of the VLDB Endowment (PVLDB)* 3, 1 (2010), 723–734.

Bing Liu. 2006. A Fast Density-Based Clustering Algorithm for Large Databases. In *Proceedings of International Conference on Machine Learning and Cybernetics*. 996–1000.

Eric Hsueh-Chan Lu, Vincent S. Tseng, and Philip S. Yu. 2011. Mining Cluster-Based Temporal Mobile Sequential Patterns in Location-Based Service Environments. *IEEE Transactions on Knowledge and Data Engineering (TKDE)* 23, 6 (2011), 914–927.

Shaaban Mahran and Khaled Mahar. 2008. Using grid for accelerating density-based clustering. In *Proceedings of IEEE International Conference on Computer and Information Technology (CIT)*. 35–40.

Jirí Matousek. 1993. Range Searching with Efficient Hiearchical Cutting. *Discrete & Computational Geometry* 10 (1993), 157–182.

Boriana L. Milenova and Marcos M. Campos. 2002. O-Cluster: Scalable Clustering of Large High Dimensional Data Sets. In *Proceedings of International Conference on Management of Data (ICDM)*. 290–297.

Davoud Moulavi, Pablo A. Jaskowiak, Ricardo J. G. B. Campello, Arthur Zimek, and Jörg Sander. 2014. Density-Based Clustering Validation. In *International Conference on Data Mining*. 839–847.

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 *Conference on High Performance Computing Networking, Storage and Analysis*. 62.

Tao Pei, A-Xing Zhu, Chenghu Zhou, Baolin Li, and Chengzhi Qin. 2006. A new approach to the nearest-neighbour method to discover cluster features in overlaid spatial point processes. *International Journal of Geographical Information Science* 20, 2 (2006), 153–168.

Attila Reiss and Didier Stricker. 2012. Introducing a New Benchmarked Dataset for Activity Monitoring. In *International Symposium on Wearable Computers*. 108–109.

S. Roy and D. K. Bhattacharyya. 2005. An Approach to Find Embedded Clusters Using Density Based Techniques. In *Proceedings of Distributed Computing and Internet Technology*. 523–535.

Gholamhosein Sheikholeslami, Surojit Chatterjee, and Aidong Zhang. 2000. WaveCluster: A Wavelet Based Clustering Approach for Spatial Data in Very Large Databases. *The VLDB Journal* 8, 3-4 (2000), 289–304.

Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. 2006. *Introduction to Data Mining*. Pearson.

Robert Endre Tarjan. 1979. A Class of Algorithms which Require Nonlinear Time to Maintain Disjoint Sets. *Journal of Computer and System Sciences (JCSS)* 18, 2 (1979), 110–127.

Cheng-Fa Tsai and Chien-Tsung Wu. 2009. GF-DBSCAN: A New Efficient and Effective Data Clustering Technique for Large Databases. In *Proceedings of International Conference on Multimedia Systems and Signal Processing*. 231–236.

Manik Varma and Andrew Zisserman. 2003. Texture Classification: Are Filter Banks Necessary?. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 691–698.

Wei Wang, Jiong Yang, and Richard R. Muntz. 1997. STING: A Statistical Information Grid Approach to Spatial Data Mining. In *Proceedings of Very Large Data Bases (VLDB)*. 186–195.

Ji-Rong Wen, Jian-Yun Nie, and HongJiang Zhang. 2002. Query clustering using user logs. *ACM Transactions on Information Systems (TOIS)* 20, 1 (2002), 59–81.

A. APPENDIX: SOLVING USEC WITH LINE SEPARATION (PROOF OF LEMMA 5.5)

We consider that all the discs in S_{ball} intersect ℓ (any disc completely below ℓ can be safely discarded), and that all discs are distinct (otherwise, simply remove the redundant ones).

For each disc $s \in S_{ball}$, we define its portion on or above ℓ as its *active region* (because only this region may contain points of S_{pt}). Also, we use the term *upper arc* to refer to the portion of the boundary of s that is strictly above ℓ. See Figure 25 for an illustration of these notions (the upper arc is in bold). Note that, as the center of s is on or below ℓ, the active region and upper arc of s are at most a semi-disc and a semi-circle, respectively. The following is a basic geometric fact:

PROPOSITION A.1. *The upper arcs of any two discs in S_{ball} can have at most one intersection point.*

Define the *coverage region*—denoted by U—of S_{ball} as the union of the active regions of all the discs in S_{ball}. Figure 26a demonstrates U for the example of Figure 10. Evidently, the answer of the USEC instance is *yes* if and only if S_{pt} has at least a point falling in U.

We use the term *wavefront* to refer to the part of the boundary of U that is strictly above ℓ; see the solid curve in Figure 26b. A disc in S_{ball} is said to be *contributing*, if it defines an arc on the wavefront. In Figure 26b, for instance, the wavefront is defined by 3 contributing discs, which are shown in bold and labeled as s_1, s_3, s_6 in Figure 26a.

It is rudimentary to verify the next three facts:

PROPOSITION A.2. *U equals the union of the active regions of the contributing discs in S_{ball}.*

PROPOSITION A.3. *Every contributing disc defines exactly one arc on the wavefront.*

PROPOSITION A.4. *The wavefront is x-monotone, namely, no vertical line can intersect it at two points.*

A.1. Computing the Wavefront in Linear Time

Utilizing the property that the centers of the discs in S_{ball} have been sorted by x-dimension, next we explain how to compute the wavefront in $O(|S_{ball}|)$ time.

Label the discs in S_{ball} as $s_1, s_2, s_3, ...$, in ascending order of their centers' x-coordinates. Let U_i ($1 \leq i \leq |S_{ball}|$) be the coverage region that unions the active regions of the first i discs. Apparently, $U_1 \subseteq U_2 \subseteq U_3 \subseteq ...$, and $U_{|S_{ball}|}$ is exactly U. Define W_i to be the wavefront of U_i, namely, the portion of the boundary of U_i strictly above ℓ. Our algorithm captures W_i in a linked list $\mathcal{L}(W_i)$, which arranges the defining discs (of W_i) in left-to-right order of the arcs (on W_i) they define (e.g., in Figure 11, $\mathcal{L}(W_6)$ lists s_1, s_3, s_6 in this order). By Proposition A.3, every disc appears in $\mathcal{L}(W_i)$ at most once. Our goal is to compute $W_{|S_{ball}|}$, which is sufficient for deriving U according to Proposition A.2.

It is straightforward to obtain W_1 from s_1 in constant time. In general, provided that W_{i-1} ($i \geq 2$) is ready, we obtain W_i in three steps:

(1) Check if s_i defines any arc on W_i.
(2) If the answer is *no*, set $W_i = W_{i-1}$.
(3) Otherwise, derive W_i from W_{i-1} using s_i.

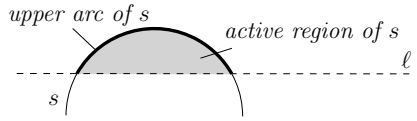
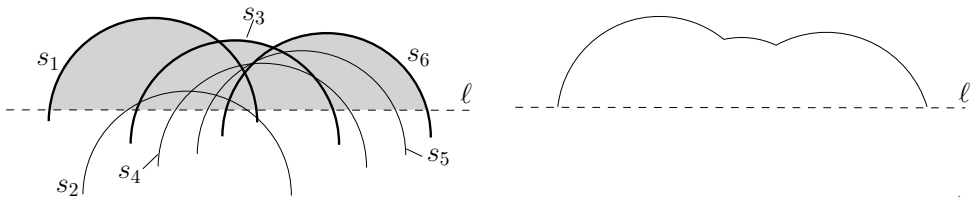


Fig. 25. Illustration of active region and upper arc



(a) Coverage region U (the shaded area) (b) Wavefront (solid curve)

Fig. 26. Deciding the existence of an edge by USEC with line separation

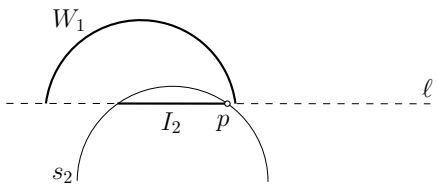


Fig. 27. Illustration of the Step-1 algorithm in Section A.1

Next, we describe how to implement Steps 1 and 3.

Step 1. We perform this step in constant time as follows. Compute the intersection of s_i and ℓ. The intersection is an interval on ℓ, denoted as I_i. Let s_{last} be the rightmost defining disc of W_{i-1} (i.e., the last disc in $\mathcal{L}(W_{i-1})$). If the right endpoint of I_i lies in s_{last}, return *no* (that is, s_i does not define any arc on W_i); otherwise, return *yes*.

As an example, consider the processing of s_2 in Figure 26a. At this moment, W_1 is as shown in Figure 27, and includes a single arc contributed by s_1. Point p is the right endpoint of I_2. As p falls in s_1 (= s_{last}), we declare that s_2 does not define any arc on W_2 (which therefore equals W_1).

The lemma below proves the correctness of our strategy in general:

LEMMA A.5. *Our Step-1 algorithm always makes the correct decision.*

PROOF. Consider first the case where the right endpoint p of I_i is covered by s_{last}. Let I_{last} be the intersection between s_{last} and ℓ. By the facts that (i) the x-coordinate of the center of s_i is larger than or equal to that of the center of s_{last}, and (ii) s_i and s_{last} have the same radius, it must hold that I_i is contained in I_{last}. This implies that the active region of s_i must be contained in that of s_{last} (otherwise, the upper arc of s_i needs to go out of s_{last} and then back in, and hence, must intersect the upper arc of s_{last} at 2 points, violating Proposition A.1). This means that s_i cannot define any arc on W_i; hence, our *no* decision in this case is correct.

Now consider the case where p is not covered by s_{last}. This implies that p is not covered by U_{i-1}, meaning that I_i must define an arc on W_i (because at least p needs to appear in U_i). Our *yes* decision is therefore correct. □

Step 3. We derive $\mathcal{L}(W_i)$ by possibly removing several discs at the end of $\mathcal{L}(W_{i-1})$, and then eventually appending s_i. Specifically:

(3. 1) Set $\mathcal{L}(W_i)$ to $\mathcal{L}(W_{i-1})$.
(3. 2) Set s_{last} to the last disc in $\mathcal{L}(W_i)$.
(3. 3) If the arc on W_{i-1} defined by s_{last} is contained in s_i, remove s_{last} from $\mathcal{L}(W_i)$ and repeat from Step 3.2.
(3. 4) Otherwise, append s_i to the end of $\mathcal{L}(W_i)$ and finish.

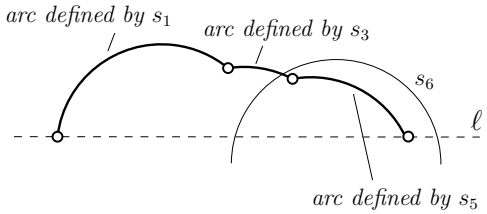


Fig. 28. Illustration of the Step-3 algorithm in Section A.1

To illustrate, consider the processing of s_6 in Figure 26a. At this moment, the wavefront W_5 is as shown in Figure 28, where the arcs are defined by s_1, s_3, and s_5, respectively. Our Step-3 algorithm starts by setting $\mathcal{L}(W_6)$ to $\mathcal{L}(W_5)$, which lists s_1, s_3, s_5 in this order. Currently, $s_{last} = s_5$. As the arc on W_5 defined by s_5 is covered by s_6 (see Figure 28), we remove s_5 from $\mathcal{L}(W_6)$, after which s_{last} becomes s_3. As the arc on W_5 defined by s_3 is not contained in s_6, the algorithm terminates by adding s_6 to the end of $\mathcal{L}(W_6)$, which now lists s_1, s_3, s_6 in this order.

Now we prove the correctness of our algorithm:

LEMMA A.6. *Our Step-3 algorithm always obtains the correct $\mathcal{L}(W_i)$.*

PROOF. If the arc on W_{i-1} defined by s_{last} is covered by s_i, the upper arc of s_{last} must be covered by the union of the discs in $\{s_1, s_2, ..., s_i\} \setminus \{s_{last}\}$. Therefore, s_{last} is not a defining disc of W_i and should be removed.

Otherwise, s_{last} must be retained. Furthermore, in this case, s_i cannot touch the arc on W_{i-1} defined by any of the discs that are before s_{last} in $\mathcal{L}(W_i)$. All those discs, therefore, should also be retained.

Finally, by Lemma A.5 and the fact that the execution is at Step 3, we know that s_i defines an arc on W_i, and thus, should be added to $\mathcal{L}(W_i)$. □

Running Time. It remains to bound the cost of our wavefront computation algorithm. Step 1 obviously takes $O(|S_{ball}|)$ time in total. Step 2 demands $\sum_{i=1}^{n} O(1 + x_i)$ time, where x_i is the number of discs deleted at Step 3.3 when processing disc s_i. The summation evaluates to $O(|S_{ball}|)$, noticing that $\sum_{i=1}^{n} x_i \leq n$ because every disc can be deleted at most once.

A.2. Solving the USEC Problem

Recall that the USEC instance has a *yes* answer if and only if a point of S_{pt} is on or below the wavefront. Proposition A.4 suggests a simple planesweep strategy to determine the answer. Specifically, imagine sweeping a vertical line from left to right, and at any moment, remember the (only) arc of the wavefront intersecting the sweeping line. Whenever a point $p \in S_{pt}$ is swept by the line, check whether it falls below the arc mentioned earlier. Because (i) the arcs of the wavefront have been listed from left to right, and (ii) the points of S_{pt} have been sorted on x-dimension, the planesweep can be easily implemented in $O(|S_{ball}|+|S_{pt}|)$ time, by scanning the arcs in the wavefront and the points of S_{pt} synchronously in ascending order of x-coordinate. This concludes the proof of Lemma 5.5, and also that of Theorem 5.2.