International Journal of Computational Geometry & Applications


# GEOMETRIC ALGORITHMS FOR DENSITY-BASED DATA CLUSTERING

DANNY Z. CHEN*
*Department of Computer Science and Engineering, University of Notre Dame, Notre Dame, Indiana 46556, USA*
*chen@cse.nd.edu*

MICHIEL SMID†
*School of Computer Science, Carleton University Ottawa, Ontario, Canada K1S 5B6*
*michiel@scs.carleton.ca*

BIN XU*‡
*Department of Computer Science and Engineering, University of Notre Dame, Notre Dame, Indiana 46556, USA*
*bxu@cse.nd.edu*

Received (received date)
Revised (revised date)
Communicated by (Name)

Data clustering is a fundamental problem arising in many practical applications. In this paper, we present new geometric approximation and exact algorithms for the density-based data clustering problem in $d$-dimensional space $\mathbb{R}^d$ (for any constant integer $d \geq 2$). Previously known algorithms for this problem are efficient only when the specified range around each input point, called the $\delta$-neighborhood, contains on average a constant number of input points. Different distributions of the input data points have significant impact on the efficiency of these algorithms. In the worst case when the data points are highly clustered, these algorithms run in quadratic time, although such situations might not occur very frequently on real data. By using computational geometry techniques, we develop faster approximation and exact algorithms for the density-based data clustering problem in $\mathbb{R}^d$. In particular, our approximation algorithm based on the $\epsilon$-fuzzy distance function takes $O(n \log n)$ time for any given fixed value $\epsilon > 0$, and our exact algorithms take sub-quadratic time. The running times and output quality of our algorithms do not depend on any particular data distribution. We believe that our fast approximation algorithm is of considerable practical importance, while our sub-quadratic exact algorithms are more of theoretical interest. We implemented our approximation algorithm and the experimental results show that our approximation algorithm is efficient on arbitrary

*The work of these authors was supported in part by Lockheed Martin Corporation and by the National Science Foundation under Grant CCR-9988468.
†The work of this author was supported in part by NSERC.
‡Corresponding author.

input point sets.

*Keywords*: Density-based clustering; geometric algorithms; approximation algorithms; range search; nearest neighbor search; BBD trees; graph traversal.

## 1. Introduction

Data clustering is a fundamental problem that arises in many applications (e.g., data mining, information retrieval, pattern recognition, biomedical informatics, and statistics). The main objective of data clustering is to partition a given data set into clusters (i.e., subsets) based on certain criteria. Significant challenges to clustering are that the size of the input data is often very large, and that little *a priori* knowledge about the structure of the data is known. For example, huge sets of spatial data can be generated by satellite images, medical instruments, video cameras, etc. Data clustering research can be classified as density-based (e.g., see ref.[15,16,18,20,25,26]) and non-density-based (e.g., minimum diameter, $k$-center, $k$-median, and minimum sum[3,7,11,12,13,14,19,21,23,24]).

In this paper, we consider the density-based data clustering problem for spatial data. This problem can be defined as follows[16]. Suppose we are given a set $S$ of $n$ points in the $d$-D space $\mathbb{R}^d$ (for any constant integer $d \geq 2$) and two parameters $\delta > 0$ and $\tau > 1$. For any point $p$ in $\mathbb{R}^d$, we denote by $N_\delta(p)$ the sphere centered at $p$ and having radius $\delta$ in $\mathbb{R}^d$ (based on some given distance function). This sphere $N_\delta(p)$ is called the *$\delta$-neighborhood* of $p$.

(1) If for a point $p \in S$, there are at least $\tau$ points of $S$ (including $p$) in the sphere $N_\delta(p)$, i.e., $|S \cap N_\delta(p)| \geq \tau$, then all points of $S \cap N_\delta(p)$ belong to the same cluster of $S$.
(2) For two subsets $C_1$ and $C_2$ of $S$, if each of $C_1$ and $C_2$ belongs to a cluster and if $C_1 \cap C_2 \neq \emptyset$, then $C_1 \cup C_2$ belongs to the same cluster.
(3) A *cluster* of $S$ is a maximal set satisfying the two conditions above.
(4) All points of $S$ that do not belong to any cluster are called *noise*.
(5) The *density-based clustering (DBC) problem* is to find all clusters of $S$, and all noise of $S$.

Note that the shape of a cluster depends on the given distance function.

Intuitively, one can view the DBC problem as finding galaxies and isolated stars in $\mathbb{R}^3$, and cities and rural residences in $\mathbb{R}^2$, based on a certain neighborhood density of the data points.

A generalization of the DBC problem is the *weighted* DBC (WDBC) problem: Each point of $S$ is associated with a real-valued weight, and the sum of weights of the points in $S \cap N_\delta(p)$ (instead of its size $|S \cap N_\delta(p)|$) is used to decide whether a point $p \in S$ is dense. The WDBC problem also arises in applications. For example, in clustering image data (e.g., image segmentation), each voxel is associated with a brightness value (its weight), and a clustering based on the sum of weights of the voxels in a specified neighborhood may be meaningful.

It is easy to solve the DBC problem in $O(n^2)$ time. Jain and Dubes[20] introduced a density-based method to identify clusters among points in $\mathbb{R}^d$. A heuristic algorithm for determining the parameters $\delta$ and $\tau$ was given in[16]. Commonly used DBC approaches in the data mining community in general first represent the input point set $S$ by a certain data structure, and then search the $\delta$-neighborhood of each point of $S$ to form clusters. For example, the DBSCAN (Density Based Spatial Clustering of Applications with Noise) algorithms[15,16,17,18,25] are based on the R*-tree[8], whereas the FDC (Fast Density-based Clustering) algorithm[26] is based on the $k$-D tree[9,10]. These DBC algorithms[15,16,17,18,25,26] are efficient when each $\delta$-neighborhood contains on average a constant number of points in $S$. However, different distributions of the input data points have significant impact on the efficiency of these algorithms. This is because they search the $\delta$-neighborhood of each input point without deleting input points in the neighborhood searched (thus some input points may be visited many times). In the worst case when the data points are highly clustered, these algorithms run in quadratic time, although such situations might not occur very frequently on real data. Note that for many database applications, since their data are of enormous sizes and need not be well distributed, $\Theta(n^2)$ time algorithms are highly impractical.

In this paper, we present faster approximation and exact algorithms for the DBC problem in $\mathbb{R}^d$ ($d \geq 2$). The running times and output quality of our algorithms do not depend on any particular data distribution. To achieve nearly linear time DBC algorithms, we consider developing approximate solutions. Our approximation is in the following sense[4]. Let $\epsilon > 0$ be any given constant. For a point $p \in \mathbb{R}^d$, the boundary of our $\delta$-neighborhood $N_\delta(p)$ is *fuzzy* (called *$\epsilon$-fuzzy*), i.e., each point of $S$ whose distance to $p$ is within the range $[(1-\epsilon)\delta, (1+\epsilon)\delta]$ is arbitrarily considered to be in or out $N_\delta(p)$. On the other hand, points whose distances to $p$ are less than $(1-\epsilon)\delta$ (resp., larger than $(1+\epsilon)\delta$) are guaranteed to be in $N_\delta(p)$ (resp., not in $N_\delta(p)$). Studying approximations of this kind is useful because in many practical situations, the input data is imprecise or an approximate solution obtained in a small amount of time is sufficient. Our main results are summarized below.

- An $O(n^{2(1-1/(d+2))}\text{polylog}(n))$ time exact DBC algorithm for the Euclidean distance metric.
- An $O(kn \log n)$ time exact DBC algorithm, if $d = 2$, for the convex distance function based on a regular $k$-gon.
- An $O(n \log n + n(1/\epsilon)^{d-1})$ time approximate DBC algorithm for the $\epsilon$-fuzzy distance function.
- We have implemented our approximate DBC algorithm and tested it for various clustered point sets. The experimental results show that our approximation algorithm is efficient and does not depend on any particular data distribution (as indicated by our theoretical analysis).

We think our fast approximation algorithm is of considerable practical impor-

tance, while our sub-quadratic exact algorithms are more of theoretical interest.

We formulate the DBC problem as computing special connected components of a certain geometric graph $G$. This graph $G$ may have $\Theta(n^2)$ edges. Therefore, in order to obtain a sub-quadratic time DBC algorithm, we represent $G$ implicitly, and manage to find the special connected components by examining only $O(n)$ edges of $G$. This is achieved by using techniques from computational geometry such as dynamic range search data structures.

Since the WDBC problem can be solved with only minor modifications to our DBC algorithms, in the rest of the paper, we'll discuss only the solutions for the DBC problem.

The rest of this paper is organized as follows. We present an overview of the general algorithmic steps in Section 2, and the efficient exact and approximation theoretical implementations of the general algorithm in Section 3. Some modifications to the approximate DBC approach are given in Section 4 to simplify our approximate DBC algorithm and its implementation. The experimental results of the approximate DBC algorithm based on these modifications are shown in Section 5.

## 2. Main Ideas and Algorithm Overview

Let $S \subseteq \mathbb{R}^d$ be the input point set, and let $S_\delta(p)$ denote the point set $S \cap N_\delta(p)$ for any point $p$ in $\mathbb{R}^d$. A point $p \in S$ is called a *dense* point if $|S_\delta(p)| \geq \tau$; otherwise, $p$ is called a *sparse point*. Dense and sparse points play different roles in our algorithms.

One of our main ideas is to formulate the DBC problem as that of computing special connected components of an undirected graph $G$, which is defined as follows. Each vertex $v_p$ of $G$ corresponds to exactly one point $p$ of $S$. An edge connects two distinct vertices $v_p$ and $v_q$ in $G$ if and only if (i) the distance $dist(p, q)$ between the two corresponding points $p$ and $q$ of $S$ (based on a given distance metric *dist*) is less than or equal to $\delta$, and (ii) $p$ or $q$ is a dense point. We say that a connected component of $G$ is *non-trivial* if it contains a vertex $v_p$ such that the corresponding point $p \in S$ is a dense point. In the rest of this paper, we let $p$ denote both a point in $S$ and its corresponding vertex in $G$. The following lemma is obvious.

**Lemma 1.** *Each cluster of $S$ corresponds to exactly one non-trivial connected component of $G$.*

Hence, our goal is to compute all non-trivial connected components of $G$. But, storing and finding $G$'s connected components in a straightforward manner will take $O(n^2)$ time since $G$ can have $\Theta(n^2)$ edges. Our second idea is to avoid looking at all edges of $G$ when computing its non-trivial connected components. In fact, we seek to examine only $O(n)$ edges of $G$ (i.e., $O(n)$ pairs of points of $S$) in this computation. To do that, we assume for the rest of this section that we have some data structures that can be used to obtain information on the (dynamically changing) subset of $S$ in the $\delta$-neighborhood $N_\delta(p)$ of any point $p \in S$. In Section 3, we will show

how these data structures can actually be theoretically implemented exactly and approximately.

Basically, we find each non-trivial connected component in $G$ by a breadth-first search traversal of $G$, starting at some dense point. In particular, whenever a point $p \in S$ is included in a cluster, $p$ is removed from $S$ (and hence it is removed from all relevant data structures storing $p$), so that no subsequent search in $G$ will visit the vertex $p$ again. To achieve this, some care must be taken. Below is an overview of our general algorithm. (The details on how to actually carry out the key steps of this algorithm will be shown in Section 3.)

(1) Compute the subsets of all dense and sparse points in $S$, denoted by $D$ and $H$, respectively.
(2) Build a dynamic data structure $T_S$ (resp., $T_D$) for the point set $S$ (resp., $D$).
    /* Initially, $T_S$ contains all input points, and $T_D$ contains only the dense points. */
(3) While $T_D \neq \emptyset$, do the following: /* There is at least one more cluster to be found. */
    (a) Initialize an empty queue $DQ$, and get a new cluster ID.
    (b) Delete an arbitrary point $p$ from $T_D$, put $p$ into $DQ$, and delete $p$ from $T_S$.
    (c) While $DQ \neq \emptyset$, do the following:
        /* This `while` loop finds the cluster containing all (dense) points in $DQ$. */
        i. Remove the first point $p$ from $DQ$, and assign the cluster ID to $p$.
        ii. Find the set $CS_\delta(p)$ of all points of $S_\delta(p)$ that are currently in $T_S$, and delete these points from $T_S$. /* $CS_\delta(p)$ may contain both dense and sparse points. */
        iii. While $CS_\delta(p) \neq \emptyset$, do the following:
            A. Remove a point $q$ from $CS_\delta(p)$, and assign the cluster ID to $q$.
            /* Each point $q$ in $S_\delta(p)$ belongs to the same cluster as the dense point $p$. */
            B. If $q$ is a dense point, then put $q$ at the end of $DQ$, and delete $q$ from $T_D$. Otherwise ($q$ is a sparse point), find the set $DS_\delta(q)$ of all dense points of $S_\delta(q)$ currently in $T_D$, delete these points from $T_D$, and put these points at the end of $DQ$.
            /* All dense points of $S_\delta(q)$ belong to the same cluster as the sparse point $q$. */
(4) All points of $S$ that are still in $T_S$ are noise.

We now show the correctness of this clustering algorithm. First, we argue that once a point $p$ of $S$ is visited by the breadth-first search procedure, $p$ will not be visited again in the rest of the search. This can be seen easily since whenever a point of $S$ is reported in either $T_S$ or $T_D$, it is immediately deleted from all the corresponding data structures. This implies that only $O(n)$ edges of $G$ (i.e., $O(n)$ pairs of points of $S$) are visited by the algorithm.

Next, we argue that each cluster of $S$ (i.e., each non-trivial connected component of $G$) is obtained correctly by the above breadth-first search procedure (contained in the outermost `while` loop of Step 3). Note that there are two types of edges in $G$: (1) those connecting two dense points, and (2) those connecting a dense point and a sparse point. When a dense point $p$ is taken from the queue $DQ$ (called the *dense-point queue*), the edges of both types connecting $p$ and all other unvisited points in $S_\delta(p)$ are visited; this is done inside the innermost `while` loop of Step 3. After a sparse point $q$ in a cluster is visited, the type (2) edges connecting $q$ and all unvisited dense points in $S_\delta(q)$ are visited, putting these dense points into the queue $DQ$ (also inside the innermost `while` loop of Step 3). In an inductive fashion, the breadth-first search procedure explores the connectivity of $G$. Hence, the non-trivial connected component of $G$ being searched is found correctly, as stated by the next lemma.

**Lemma 2.** *All non-trivial connected components of the graph $G$ are correctly computed by the above clustering algorithm. Furthermore, the algorithm only visits $O(n)$ edges of $G$.*

The astute reader may wonder whether Lemma 2 still holds if the above algorithm uses only one dynamic data structure $T_S$ (i.e., $T_D$ is not used). In this case, it appears difficult for this algorithm to visit only $O(n)$ pairs of points of $S$. The reason is that since $T_D$ is not available, when we need to identify the dense points in $S_\delta(q)$ for a sparse point $q$ (in the innermost `while` loop of Step 3), we must examine all unvisited (dense and sparse) points of $S$ in $S_\delta(q)$. The number of sparse points thus examined can be large and such sparse points may stay in $T_S$ after this examination (because they need not belong to the same cluster as $q$). This may lead to an $\Theta(n^2)$ time algorithm. However, an actual implementation may combine $T_S$ and $T_D$ into a single data structure (e.g., to save memory), as long as such a data structure supports the functionalities of both $T_S$ and $T_D$.

Also, observe that both the data structures $T_S$ and $T_D$ used in our algorithm need not be fully dynamic. What we essentially need are two types of operations on $T_S$ and $T_D$: (1) searching for the points of $S$ in $N_\delta(p)$ for a point $p \in S$ (i.e., the point set $S_\delta(p)$), and (2) deleting points from the two data structures. Each query point is a point in $S$, and the insertion operation is not necessary. We should point out that this fact will be extensively explored by the version of approximation algorithm used in our programming implementation in Sections 4 and 5.

## 3. Efficient Theoretical Implementations of the Clustering Algorithm

In the previous section, we have given a high-level overview of our clustering algorithm. To obtain fast theoretical implementations for the exact and approximation algorithms, we must carefully choose the data structures that support the search and deletion operations.

### 3.1. *Computing the Dense Points*

In Step 1 of the clustering algorithm, the set $S$ is partitioned into $D$ (the dense points) and $H$ (the sparse points). To implement this step, we need to compute, for each point $p$ of $S$, the number of points in $S \cap N_\delta(p)$. As we will see below, the efficiency of a solution to this problem heavily depends on the metric used and on whether we compute the size of $N_\delta(p)$ exactly or approximately.

We first consider the Euclidean metric. This case can be solved by well known techniques in computational geometry[2]. By projecting the $d$-dimensional space to $(d+1)$-dimensional space, we can reduce the problem of computing the dense points of $S$ to the problem of answering $n$ half-space counting queries in the $(d+1)$-dimensional set $S'$, where $S'$ consists of the projection points of $S$.[2]

Given any parameter $m$, $n \leq m \leq n^{d+1}$, the points of $S'$ can be preprocessed in $O(m)$ time, such that any half-space counting query can be answered in $O((n/m^{1/(d+1)})\text{polylog}(n))$ time; see Agarwal and Erickson[2]. Therefore, the dense points in $S$ can be computed in $O(m+(n^2/m^{1/(d+1)})\text{polylog}(n))$ time. If we choose $m = n^{2(d+1)/(d+2)}$, then the running time becomes $O(n^{2(1-1/(d+2))}\text{polylog}(n))$, which is (slightly) sub-quadratic in $n$.

Let us now assume that the dimension $d$ is equal to two, and let us use a convex distance function based on a regular $k$-gon, where $k \geq 3$ is an integer. If $k$ is sufficiently large, then this metric approximates the Euclidean metric arbitrarily closely. Let $K$ be the regular $k$-gon of radius $\delta$ having its center at the origin. For any $p \in S$, let $K_p := K + p$, i.e., $K_p$ is the regular $k$-gon obtained by translating $K$ so that its center is at $p$. We want to compute, for each $p \in S$, the number of points of $S$ that are in $K_p$. By partitioning $K$ into $k-2$ triangles $K^1, K^2, \ldots, K^{k-2}$, we can reduce this to computing, for each $p \in S$ and each $1 \leq i \leq k-2$, the number of points of $S$ that are in the translated triangle $K_p^i$. We will solve this problem for each $i$ independently.

We fix $i$ with $1 \leq i \leq k-2$. Let $p \in S$, let $T_p := K_p^i$, and let $a_p$, $b_p$, and $c_p$ be the vertices of $T_p$, sorted in counterclockwise order. Let $h_p^{ab}$ be the half-plane consisting of all points of $\mathbb{R}^2$ that are to the right of the directed line through $a_p$ and $b_p$. Define $h_p^{bc}$ and $h_p^{ca}$ analogously, and let $n_p^{ab} := |h_p^{ab} \cap S|$, $n_p^{bc} := |h_p^{bc} \cap S|$, and $n_p^{ca} := |h_p^{ca} \cap S|$. Let $m_p^c$ be the number of points of $S$ that are in the wedge with apex $c_p$ that is bounded by the lines through $\overline{a_p c_p}$ and $\overline{b_p c_p}$ and that does not contain $T_p$. Define $m_p^a$ and $m_p^b$ in an analogous way. Then the number of points in $S \cap T_p$ is equal to

$$n - n_p^{ab} - n_p^{bc} - n_p^{ca} + m_p^a + m_p^b + m_p^c. \tag{1}$$

Hence, our problem becomes that of computing each of the terms in (1) for each point of $S$. By sorting the points of $S$ in the direction orthogonal to the line through $a_p$ and $b_p$, the values $n_p^{ab}$, for all $p \in S$, are obtained in $O(n \log n)$ time. All values $n_p^{bc}$ and $n_p^{ca}$ can be obtained in the same amount of time. Let us see how all values $m_p^c$, $p \in S$, can be computed. Let $\ell_p$ and $\ell'_p$ be the bounding lines of the wedge defining $m_p^c$. By sweeping over all points of $S$ in the direction orthogonal to $\ell_p$, and

by maintaining all points visited so far in a balanced binary search tree sorted in the direction orthogonal to $\ell'_p$, we obtain all values $m^c_p$, $p \in S$, in $O(n \log n)$ time.

To summarize, using the convex distance function based on a regular $k$-gon, all dense points of $S$ can be computed in $O(kn \log n)$ time. This method is likely to be extended to higher dimensions by approximating the Euclidean hyper-sphere of radius $\delta$ by an appropriate $k$-facet polytope.

Observe that in this solution, we approximate the Euclidean circle $C$ of radius $\delta$ by a regular $k$-gon $K$. Hence, we ignore points of $S$ that are in $C_p \setminus K_p$. In our next solution, we use a more "fuzzy" approach to approximate $|C_p \cap S|$ in $\mathbb{R}^d$ ($d \geq 2$), which is due to Arya and Mount[4]: Any point of $S$ that is within Euclidean distance $(1-\epsilon)\delta$ of $p$ is guaranteed to be counted; any point of $S$ whose Euclidean distance to $p$ is more than $(1+\epsilon)\delta$ is guaranteed not to be counted; any other point of $S$ may or may not be counted. Here, $\epsilon > 0$ is a given fixed real number. Arya and Mount[4] have shown how to preprocess $S$ in $O(n \log n)$ time, such that an approximate range counting query can be answered in $O(\log n + (1/\epsilon)^{d-1})$ time. This leads to an algorithm that computes all points of $S$ that are "approximately" dense, in $O(n \log n + n(1/\epsilon)^{d-1})$ time.

### 3.2. *Steps 2 and 3*

To implement Steps 2 and 3 of our clustering algorithm, we need a data structure for the following problem. Let $Z$ be a set of $n$ points in $\mathbb{R}^d$. (In the clustering algorithm, $Z$ is either the entire set $S$ or the dense point set $D$.) We want to process an on-line operation sequence, in which each operation is

- a query of the form "given a point $p$ of $Z$, report all points of $Z$ that are contained in the $\delta$-neighborhood $N_\delta(p)$," or
- a deletion of a point from $Z$.

Let $q_1, q_2, \ldots, q_k$ be all query points, and let $A_i$ be the output for query point $q_i$, $i \geq 1$. The sequence of operations has the properties that $k \leq n$ and that the sets $A_i$, $i \geq 1$, are pairwise disjoint. Hence, $\sum_i |A_i| \leq n$.

Let us again consider this problem first for the Euclidean metric. As we have seen in Section 3.1, a query can be reduced to a half-space reporting query in $\mathbb{R}^{d+1}$. Agarwal, Eppstein, and Matoušek[1] have shown that, for any parameter $m$, $n \leq m \leq n^{\lfloor (d+1)/2 \rfloor}$, a data structure can be built in $O(m^{1+\nu})$ time such that any query can be answered in $O((n/m^{\lfloor (d+1)/2 \rfloor}) \log n + |A_i|)$ time, and any deletion can be done in $O(m^{1+\nu}/n)$ time. Here, $\nu$ is an arbitrarily small positive real constant. Using this result, the entire sequence of operations takes $O(m^{1+\nu} + (n^2/m^{\lfloor (d+1)/2 \rfloor}) \log n)$ time. If we choose $m = n^{2(1-1/\lfloor (d+3)/2 \rfloor)}$, then the entire running time for implementing Steps 2 and 3 is $O(n^{2(1-1/(d+2))}\text{polylog}(n))$, which is the time for computing the dense points (see Section 3.1).

Next we assume that $d = 2$ and use the convex distance function based on the regular $k$-gon $K$. Klein *et al.*[22] have designed a data structure that can be built

in $O(kn \log n)$ time, that supports each query in $O(k \log n + |A_i|)$ time and each deletion in $O(k \log n)$ time. (Recall that in a query, we need to report all points of $Z$ that are contained in a translate of $K$.) Hence, for this metric, Steps 2 and 3 can be implemented in $O(kn \log n)$ time.

Finally, if we use the "fuzzy" notion of neighborhood in $\mathbb{R}^d$, $d \geq 2$ (see Section 3.1), we can use the data structure of Arya and Mount[4]. This data structure answers each query in $O(\log n + (1/\epsilon)^{d-1} + |A_i|)$ time, and supports each deletion in $O(\log n)$ time. (The latter claim can be proved by using the same techniques that were used to dynamize the approximate nearest neighbor data structure of Arya *et al.*[6].) This leads to an $O(n \log n + n(1/\epsilon)^{d-1})$ time implementation of Steps 2 and 3.

### 3.3. *The Final Time Bounds*

By combining the results obtained above, we have the following theorem.

**Theorem 1.** *Let $S$ be a set of $n$ points in $\mathbb{R}^d$, and let $\delta > 0$ and $\tau > 1$ be two parameters. The density-based clustering problem for $S$ (based on $\delta$ and $\tau$) can be solved in*

*(1)* $O(n^{2(1-1/(d+2))}\text{polylog}(n))$ *time for the Euclidean metric.*
*(2)* $O(kn \log n)$ *time, if $d = 2$, for the convex distance function based on a regular $k$-gon.*
*(3)* $O(n \log n + n(1/\epsilon)^{d-1})$ *time for the $\epsilon$-fuzzy distance function, for any real constant $\epsilon > 0$.*

## 4. Refinements to the Approximate DBC Approach

This section presents some refinements to the approximate DBC approach given in Section 3. There are two reasons for making these refinements: (i) It simplifies the approximate DBC algorithm considerably and makes it easier for our implementation to use existing geometric computing software such as Arya and Mount's ANN library[5] (instead of starting from scratch), hence greatly reducing our programming efforts; (ii) it reduces the execution time and memory usage in the actual implementation (by some constant factors), which is especially meaningful when the data sets are very large. The approximate DBC approach given in this section is based on Arya and Mount's BBD tree data structure for approximate range search in $\mathbb{R}^d$ for $d \geq 2$.[4] A similar data structure was used for approximate nearest neighbor queries in $\mathbb{R}^d$.[6] However, our approximate DBC algorithm needs the approximate range search data structure to support dynamic deletion operations (as shown in Section 3), but the BBD tree data structures in ref.[4,6] are only for static settings (i.e., the input point set $S$ remains unchanged throughout). Thus, we need to modify the BBD tree in ref.[4] to accommodate deletions (fortunately, no insertion is needed). Since a BBD tree data structure has been implemented in the ANN library by Arya and Mount[5], our modified BBD tree for approximate range search with deletions

can be implemented by using and extending some of the programming structures in ref.[5]. Also, we seek to make our implementation as practically efficient as possible, both in the execution time and memory usage (even constant factor improvements are very useful when $S$ is large).

Hence, we make the following key refinements: 1) Using only one approximate range search data structure, a BBD tree $T$, for storing both the input point set $S$ and dense point set $D$, instead of two data structures $T_S$ and $T_D$ as discussed in Section 3 (i.e., $T$ is a combination of $T_S$ and $T_D$); 2) a fast deletion operation on $T$. Note that $T$ does not store the dense points twice, but merely marks each point of $S$ as dense or sparse. Section 4.1 describes the structure of our BBD tree $T$. Section 4.2 presents and analyzes the search and deletion operations on $T$. The final approximate DBC algorithm with these refinements is given in Section 4.3.

### 4.1. *Approximate Range Search Tree T*

Our BBD tree $T$ has the same structure as that for approximate range search in ref.[4], with a constant bucket size. In addition, we need to include more information in $T$ for our refinements.

We denote the subtree of $T$ rooted at a node $v$ by $subtree(v)$, the root of a subtree $T'$ of $T$ by $root(T')$, and the height of $T'$ by $height(T')$. Note that each node $v$ in $T$ corresponds to a subregion (denoted by $cell(v)$) in the partition of the space $\mathbb{R}^d$ based on the input point set $S$, as shown in ref.[4], $v$ has several other fields: (a) $all\_points\_number(v)$: the total number of (dense and sparse) points of $S$ that are currently stored as "present" in $subtree(v)$; (b) $dense\_points\_number(v)$: the total number of dense points in $S$ that are currently stored as "present" in $subtree(v)$; (c) $parent(v)$: a pointer to the parent node of $v$ (if any) in $T$. These fields are useful for both the search and deletion operations on $T$.

Step 1 of our approximate DBC algorithm (as shown in Section 3.1) simply uses the algorithm for approximate range *counting queries* in ref.[4] to identify approximately dense and sparse points of $S$. We assume this is already done. Then by using a procedure similar to the one for Lemma 4 of ref.[4], our approximate range search BBD tree $T$ can be constructed in $O(n \log n)$ time and $O(n)$ space. Also, note that by using the field $all\_points\_number(root(T))$ (resp., $dense\_points\_number(root(T))$), it is easy to check in $O(1)$ time whether $T$ contains any point (resp., dense point) of $S$ after many deletions.

### 4.2. *Search and Deletion Operations on T*

We need $T$ to support the following key operations on the dense point set $D$: (I) Reporting all dense points in the $\delta$-neighborhood $N_\delta(q)$ of a point $q \in S$ ($q$ may be a sparse point) that are currently "present" in $T$; (II) deleting a given dense point $p$ from $T$. We also need from $T$ similar operations on the entire input point set $S$. Due to their similarity, we only discuss these operations of $T$ on the dense point set $D$.

Based on the approximate range search algorithm in ref.[4], we have the following procedure for approximately reporting the "present" dense points within a spherical range $R(p)$ of radius $\delta$ and centered at a point $p$ in $\mathbb{R}^d$ (i.e., $R(p) = N_\delta(p)$). Let $R\_outer(p)$ denote the spherical range of radius $(1+\epsilon)\delta$, and $R\_inner(p)$ denote the spherical range of radius $(1-\epsilon)\delta$, both centered at $p$.

**Procedure** Range_Search_for_Dense_Points($v$, $p$)
*Input:* A node $v$ in the BBD tree $T$, and a point $p$ for the center of $R(p)$.
*Output:* All "present" dense points in $T$ that are in the approximate range of $p$.
begin
if $dense\_points\_number(v) = 0$ then return $\emptyset$;
if $cell(v) \cap R\_inner(p) = \emptyset$ then return $\emptyset$;
if $cell(v) \subseteq R\_outer(p)$ then
if $all\_points\_number(v) = dense\_points\_number(v)$ then
return all "present" points in $subtree(v)$;
else /* $all\_points\_number(v) > dense\_points\_number(v) > 0$ */
if $v$ is a leaf node then return all "present" dense points in $cell(v)$;
else return Range_Search_for_Dense_Points($left\_child(v), p$)
$\cup$ Range_Search_for_Dense_Points($right\_child(v), p$);
if $v$ is a leaf node then return all "present" dense points in $cell(v)$ that are in $R(p)$;
else return Range_Search_for_Dense_Points($left\_child(v), p$)
$\cup$ Range_Search_for_Dense_Points($right\_child(v), p$);
end

Like the approximate range search algorithm in ref.[4], the above approximate range reporting procedure recursively searches the BBD tree $T$ starting from its root, avoiding the nodes whose cells are outside the inner range $R\_inner(p)$ and only considering the nodes whose cells are either inside the outer range $R\_outer(p)$ or whose cells are neither inside $R\_outer(p)$ nor outside $R\_inner(p)$. For a node $v$ whose cell $cell(v)$ intersects $R\_inner(p)$, the procedure searches $subtree(v)$ to find all "present" dense points in $subtree(v)$, by using the fields $dense\_points\_number$ of appropriate nodes in $subtree(v)$. The recursive search at a node $v$ stops if $dense\_points\_number(v) = 0$. Thus, the search never wastes time on any subtree of $T$ that contains no "present" dense points.

Checking whether $T$ contains any "present" dense point and reporting such a point (e.g., for testing the condition of the outermost `while` loop of Step 3 in Section 2) can be done by tracing an arbitrary root-to-leaf path in $T$ whose nodes all have a value $dense\_points\_number > 0$.

The deletion of a dense point $p$ from $T$ is simple. Suppose $p$ is given (by a reporting procedure) for deletion. Let $leaf(p)$ be the leaf node of $T$ whose cell contains $p$. We then mark $p$ as "deleted" in $T$, and trace the path from $leaf(p)$ to $root(T)$ in $T$, adjusting the values of $all\_points\_number(v)$ and $dense\_points\_number(v)$ for each node $v$ on that path, by following the pointer $parent(v)$.

The next lemma analyzes our approximate range reporting and deletion operations on $T$.

**Lemma 3.** *On the BBD tree $T$ that is set up to store $n$ input points in $\mathbb{R}^d$, each deletion operation takes $O(\log n)$ time, and each approximate range reporting operation takes $O(m \log n + m(1/\epsilon)^{d-1})$ time for any given fixed value $\epsilon > 0$, where $m$ is the number of points that it reports.*

**Proof.** It is easy to see the correctness of the deletion operation, since it only involves marking a given point as "deleted" in $T$ and changing appropriately the values of *all_points_number*$(v)$ and *dense_points_number*$(v)$ for each node $v$ on a corresponding leaf-to-root path in $T$ (i.e., the topological structure of $T$ remains unchanged). Since the set of points in $T$ that our approximate range reporting operation reports (with deletions) is a subset of the set of points that the approximate range reporting operation in ref.[4] would report (without deletions), the correctness of our approximate range reporting operation on $T$ follows from that of its counterpart in ref.[4] and the suitable use of the fields *all_points_number*$(v)$ and *dense_points_number*$(v)$ in Procedure Range_Search_for_Dense_Points($v$, $p$).

Now we analyze the time bounds of these operations. Since once a point $p \in S$ is given, deleting $p$ from $T$ essentially involves tracing a leaf-to-root path in $T$, a deletion operation takes $O(height(T)) = O(\log n)$ time. (The fact that $height(T) = O(\log n)$ follows from Lemma 1 of ref.[4].) For an approximate range reporting operation on $T$ that reports $m$ points, because reporting each such point takes at most $O(\log n + (1/\epsilon)^{d-1})$ time (i.e., the time bound of an approximate range *counting* query in ref.[4]), the total time for reporting all $m$ points is $O(m \log n + m(1/\epsilon)^{d-1})$. □

Note that our approximate range search data structure $T$ is far from being fully dynamic and may not be optimal. In particular, the time bounds of its operations all depend on the size $n = |S|$ of the entire input point set $S$ rather than the set of the *current* points stored in $T$. Nevertheless, this data structure is sufficient to yielding an $O(n \log n + n(1/\epsilon)^{d-1})$ time approximate DBC algorithm in $\mathbb{R}^d$, as shown in the next section, and is relatively simple to implement.

### 4.3. *The Resulting Approximate DBC Algorithm*

Using the approximate range search BBD tree $T$, we obtain an $O(n \log n + n(1/\epsilon)^{d-1})$ time approximate DBC algorithm in $\mathbb{R}^d$ that is relatively simple to implement. Below we discuss this algorithm based on the steps as given in Section 2.

Step 1 is simply carried out by performing $n$ *counting queries* on the approximate range search data structure in ref.[4], one for each point of $S$. This takes altogether $O(n \log n + n(1/\epsilon)^{d-1})$ time. This step classifies approximately the points of $S$ into dense and sparse points.

Step 2 builds the approximate range search BBD tree $T$ by using an algorithm

similar to the data structure construction algorithm in ref.[4], in $O(n\log n)$ time and $O(n)$ space.

Step 3 is carried out by mainly performing approximate range reporting operations and deletion operations on $T$. Each point of $S$, once found by a reporting operation, is deleted from $T$. Since every point of $S$ (dense or sparse) is reported and deleted exactly once in $T$, at most $n$ approximate range reporting operations and $n$ deletion operations on $T$ are performed. By Lemma 3, the total time of this step is $O(n \log n + n(1/\epsilon)^{d-1})$.

**Theorem 2.** *The approximate DBC algorithm for $n$ input points in $\mathbb{R}^d$ given in this section takes $O(n \log n + n(1/\epsilon)^{d-1})$ time and $O(n)$ space for any given fixed value $\epsilon > 0$.*

## 5. Experimental Results for the Approximate DBC Algorithm

To show the good performance of the approximate DBC algorithm presented in Section 4, we implemented it in C++ on a Sun Sparc 20 workstation running Solaris. Our implementation is based on a careful extension of the programming structure of the ANN library created by Arya and Mount[5]. This is because the data structures used in ref.[5] for approximate nearest neighbor queries and our approximate DBC algorithm in Section 4 are both hinged on similar BBD trees. In fact, the availability of the ANN library significantly helped our programming efforts in this study. On the other hand, our implementation work is still quite substantial (with altogether over ten thousand lines of code).

Our experimental study has several goals. The first goal is to verify that the good efficiency of our approximation algorithm does not depends on a certain specific distribution. The second goal is to find out, in various density-based clustering settings, how the execution time of our approximate DBC algorithm varies as a function of each of several parameters: (1) the input data size $n = |S|$, (2) the accuracy factor $\epsilon$, and (3) the input data dimension $d$. The effects of the radius $\delta$ of the $\delta$-neighborhood $N_\delta(\cdot)$ with respect to these parameters are also examined. Our third goal is to develop software for our approximate DBC algorithm, and make it available to density-based clustering applications. Besides, we know the theoretical upper time bound for our approximate DBC algorithm is $O(2^d n \log n + n(3\sqrt{d}/\epsilon)^{d-1})$ which appears to be quite high for various practical settings. Our fourth goal thus is to compare the experimental results with this theoretical upper time bound to determine the practical efficiency of our approximate DBC algorithm.

### 5.1. *Execution Time and Different Data Distributions*

In order to verify that different distributions of the data sets do not have significant impact on the efficiency of our approximation algorithm, we use both real data sets and synthetic data sets of different distributions in our experiments. The data size varies from 0.5M to 4M in dimension 2.

For the real data, we used the graph data for SEQUIA 2000 storage benchmark, which contains 201,659 stream segments (arcs comprising 3,922,986 points), extracted from the US Geological Survey's Digital Line Graph hydrographic data for California. We extracted 0.5M, 1M, ..., and 3.5M points from the graph data set to form data sets of different sizes. To obtain a 4M real data set, we extracted 173,041 points randomly from the graph data set, and combined to the graph data set.

We used six different distributions for the synthetic data sets. Arya and Mount[4,5] discussed methods for generating point sets of various distributions. We use some of these methods here (see ref.[4,5] for more details on the methods of generating data sets).

(1) Uniform: Each coordinate of a point is chosen uniformly from the interval $[-1, 1]$.
(2) Gaussian: Each coordinate is chosen from the Gaussian distribution with the standard deviation of 0.05 and mean of 0.
(3) Clustered Gaussian: 10 "core" points were first chosen from the uniform distribution (in interval [0, 1]) in the unit hypercube, and then many points based on a Gaussian distribution with a standard deviation of 0.05 centered around each core point were generated in the unit hypercube.
(4) Laplacian: Each coordinate is chosen from the Laplacian distribution with mean 0 and standard deviation 1.
(5) Two Correlated Distributions: The two correlated distributions were formed by grouping the output of autoregressive sources into vectors of length $d$. The following recurrence is used to generate successive output:

$$X_i = \rho X_{i-1} + W_i,$$

where the $W_i$'s are a sequence of zero-mean, independent, identically distributed random variables. The recurrence depends on three parameters, the choice of $X_1$, the choice of $W_i$, and the value of the correlation coefficient $\rho$. We choose 0.05 for $\rho$. The values of $X_1$ and $W_i$ are defined below. The first component $X_1$ is selected from the corresponding uncorrelated distribution (either Gaussian or Laplacian) and the remaining components $X_i$ are generated by the above equation. The values of $W_i$'s are chosen by the marginal density of the $X_j$'s, which are normal with the current standard deviation.

   (a) Correlated Gaussian: Both $X_1$ and $W_i$ are chosen from the Gaussian distribution with the standard deviation of 0.05 and mean of 0.
   (b) Correlated Laplacian: Both $X_1$ and $W_i$ are chosen from the Laplacian distribution with the mean of 0 and standard deviation of 1.

Figure 1 illustrates the cluster searching times vs. the data sizes with an $\epsilon$ of 0.05. We can see that the differences of the searching times among the data sets with different distributions are very small for different data sizes. For each of

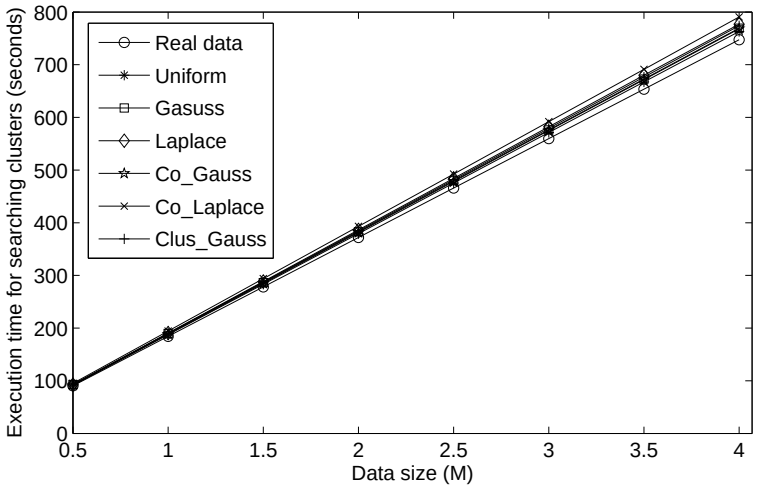


Fig. 1. Relation between the execution time of our approximate DBC algorithm and the data size, for 2-D data sets with different distributions.

the data sizes, searching clusters with correlated Laplacian distribution takes more time than other distributions, while the real data takes less time than all others. Let $T_{cl}$ denote the searching time of a correlated Laplacian data set, and $T_u$ the searching time of a uniform data set. We define the percentage of time difference as: $(T_{cl} - T_u)/T_u \times 100$. For different data sizes, 0.5M, 1M, ..., and 4M, the percentages of time difference are 4.97, 5.44, 5.59, 5.66, 5.71, 5.74, 5.76, and 5.77, respectively. The average of these percentages is 5.58, which is quite small. Our above and other many experiments have shown that different distributions of input data sets do not have significant impact on the efficiency of our algorithm.

In all the experiments below, we use data sets of clustered Gaussian distribution as described above. Each curve in Figures 2 to 6 below represents the average of 20 experiments whose data sets were generated with different seeds for the random number generator that we used.

### 5.2. *Execution Time and Accuracy*

We considered the values of $\epsilon$ varying from 0 to 0.5, the values of the radius $\delta$ of the $\delta$-neighborhood varying from 1/2 to 1/128, and data sets of size 100K. Figure 2 and Figure 3 give the execution time of our approximate DBC algorithm as a function of $\epsilon$ with some fixed values of $\delta$, for 2-D data sets.

Figure 2 and Figure 3 show some clear patterns. As $\epsilon$ decreases from 0.05 to 0, there are significant increases in execution times for larger ranges (i.e., the bigger values of $\delta$), and not so significant increases for smaller ranges (e.g., the execution time increases by 325.3% with a radius of 1/2, and only by 11.9% with a radius of 1/128). The increases of the execution times are rather slow when $\epsilon$ varies from

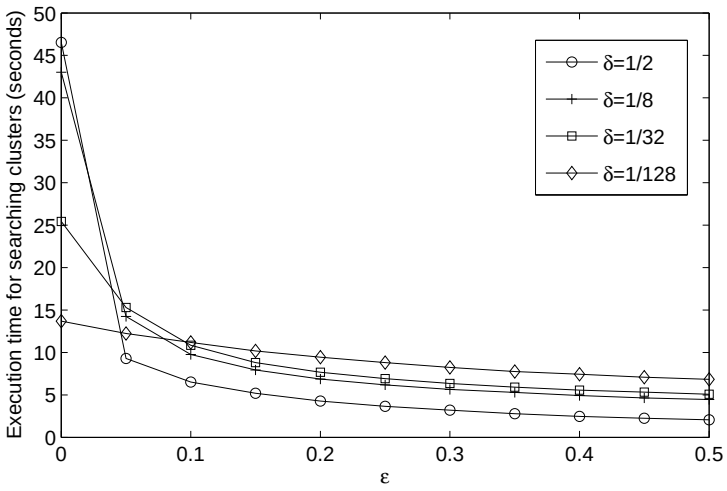


Fig. 2. Relation between the execution time of our approximate DBC algorithm and $\epsilon$, for 2-D data sets of size 100K.

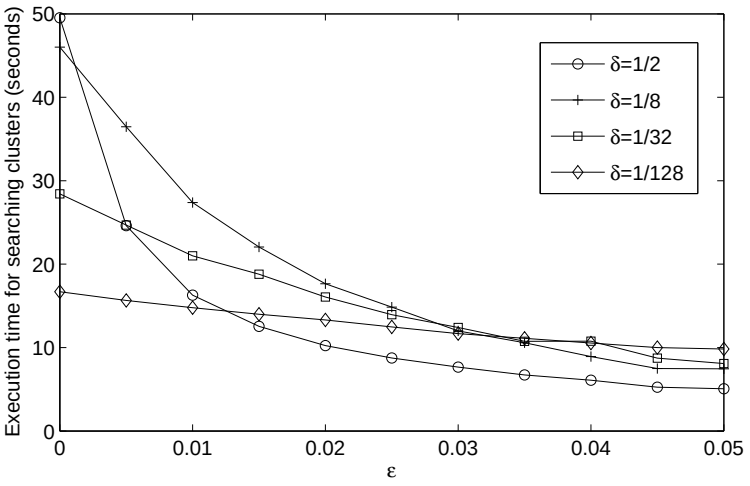


Fig. 3. Relation between the execution time of our approximate DBC algorithm and $\epsilon$, for 2-D data sets of size 100K.

0.5 to 0.05, and the trends tend to be "flat" (e.g., as $\epsilon$ decreases from 0.5 to 0.05, which is a "long" interval, the execution time increases by 144.1% with a radius of 1/2, and by 52.1% with a radius of 1/128).

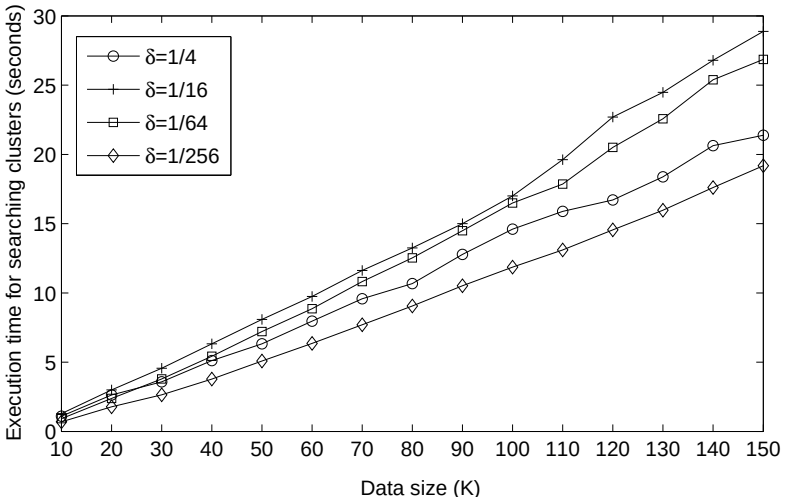


Fig. 4. Relation between the execution time of our approximate DBC algorithm and the data size, for 2-D data sets.

### 5.3. *Execution Time and Data Size*

We considered input data sets of sizes varying from 10K to 150K, and the values of $\delta$ varying from 1/4 to 1/256, and $\epsilon$ of 0.05. Figure 4 shows some of the results for 2-D data sets.

In Figure 4, the four curves for different values of $\delta$ all indicate that the execution times increase almost linearly with respect to the increases of data sizes. Further, the execution times increase very slowly as data sizes increase (with a small positive slope).

### 5.4. *Execution Time and Data Dimension*

A key issue is how the execution time of our approximate DBC algorithm depends on the dimensionality of the input data set. Since the dimension value $d$ acts as the power parameter in the theoretical upper time bound, $d$ is also a crucial factor to the execution time.

We considered dimensions varying from 2 to 30, the values of $\delta$ ranging from 1/32 to 1/256, $\epsilon$ of 0.05, and data sets of size 100K. As shown in Figure 5, for the values of $\delta$ = 1/32, 1/64, 1/128, and 1/256, when the dimension $d$ changes from 2 to 30, the execution time increases 18.4, 15.0, 14.5, and 14.6 times, respectively. However, the theoretical time bound increases $8.86 \times 10^{70}$ times!

In Section 5.2 and Section 5.3, the theoretical upper time bound is not high since the dimensions of the input data sets are all 2, which is low. However, when $\epsilon$ is small (e.g., $\epsilon = 0.05$), the theoretical upper time bound will increase dramatically with the increase of the dimension (as huge as $8.86 \times 10^{70}$ times in the above example).

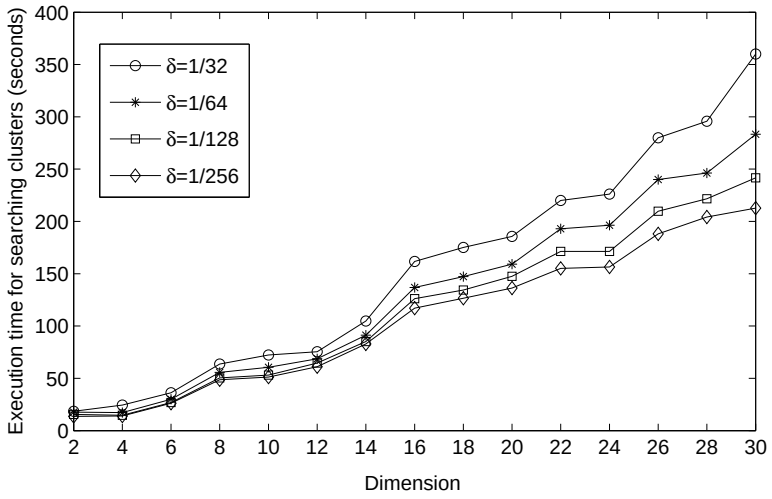


Fig. 5. Relation between the execution time of our approximate DBC algorithm and the dimension, for data sets of size 100K.

Our experimental results show that the real execution time of our algorithm is much smaller than the theoretical upper time bound with the increase of the dimension when $\epsilon$ is small. Hence, our approximation algorithm appears to be practical for real clustering applications.

### 5.5. *Approximate Range Search*

As mentioned in Section 3, we use a "fuzzy" approach in the approximate range search[4]. Since our approximate DBC algorithm is based on this technique, the output quality of our approximate DBC algorithm depends on the quality of the approximate range search. In the following experiments, we examine the quality of our approximate DBC algorithm by evaluating the approximate range search algorithm.

We considered the values of $\epsilon$ varying from 0.005 to 0.05, the values of the radius $\delta$ of the $\delta$-neighborhood varying from 1/64 to 1/512, and 2-D data sets of size 100K. For every point $p$ in a data set, we compute the range search error rate $ER_p$ in the following way: Suppose $S_0$ is the point set of $S_\delta(p)$ when $\epsilon = 0$, and $S_\epsilon$ is the point set of $S_\delta(p)$ with approximation factor $\epsilon$. $ER_p$ represents the number of points which are in $S_0$ but not in $S_\epsilon$ and in $S_\epsilon$ but not in $S_0$. The average range search error rate is the average of all the $ER_p$ values of the points in the data set.

Figure 6 gives the average range search error rates as a function of $\epsilon$ with some fixed values of radius, for 2-D data sets. As shown in Figure 6, the average error rates are very small. The average error rates are below 2% when $\epsilon = 0.005$, and between 10.3% and 5.6% when $\epsilon = 0.05$ with radii from 1/64 to 1/512. With these small average range search error rates, our approximate DBC algorithm will produce

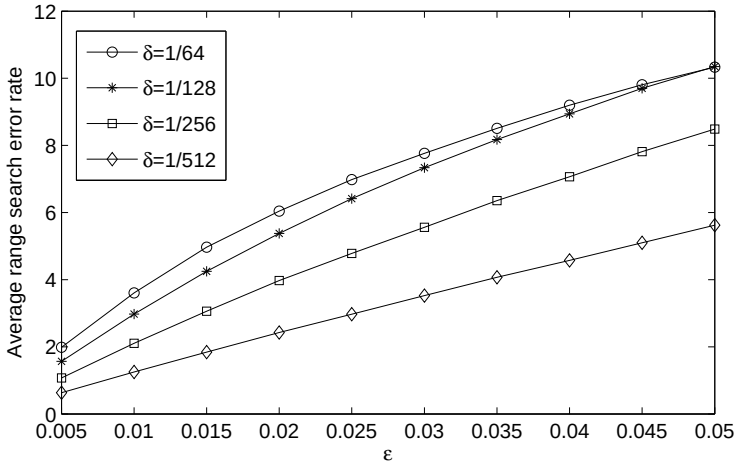


Fig. 6. Average range search error rates as a function of $\epsilon$ in our approximate DBC algorithm, for 2-D data sets of size 100K.

results with high quality.

### 5.6. *Discussion of the Experimental Results*

In the above experiments, we used clustered input data sets instead of uniformly distributed ones. One can see that comparing to the theoretical upper time bound, the observed execution time for searching clusters is not so sensitive to the dimension when $\epsilon$ is small (e.g., $\epsilon \leq 0.05$).

Below we show two examples of output produced by our approximate DBC algorithm. Figure 7(a) gives a set of 2000 input points in $\mathbb{R}^2$, generated as described at the beginning of this section. Figure 7(b) is a space partition based on the input points used by our BBD tree $T$. Using our approximate DBC algorithm with $\epsilon = 0.1$, $\delta = 0.14$, and $\tau = 45$, we obtain the output in Figure 7(c), which removes all noise points and shows the clusters clearly. Figure 8 shows an example of 100K points.

From the experimental results shown in Section 5.2, we know that by changing the value of $\epsilon$, output solutions with different accuracy can be obtained, and the execution time of our approximate DBC algorithm becomes smaller if we use bigger $\epsilon$. Note that different levels of accuracy may be needed for different real clustering applications, and fast execution time is a key requirement to many such applications. Therefore, we can achieve a good trade-off between the execution time and output quality based on the requirements of a specific application, so that we not only can solve a given clustering problem with sufficient accuracy but also complete the computation within a required time period. Hence in real applications, our approximate DBC algorithm based on the approximate range search data structure is likely to be efficient and practical.

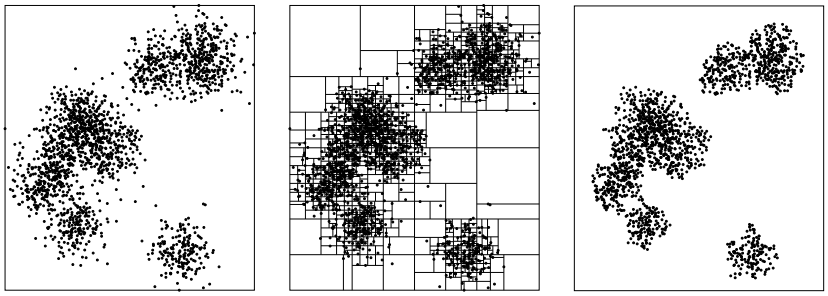


Fig. 7. An output example produced by our approximate DBC algorithm on 2000 points in $\mathbb{R}^2$: (a) The input data set, (b) the space partition used by our BBD tree $T$, and (3) the output clusters.

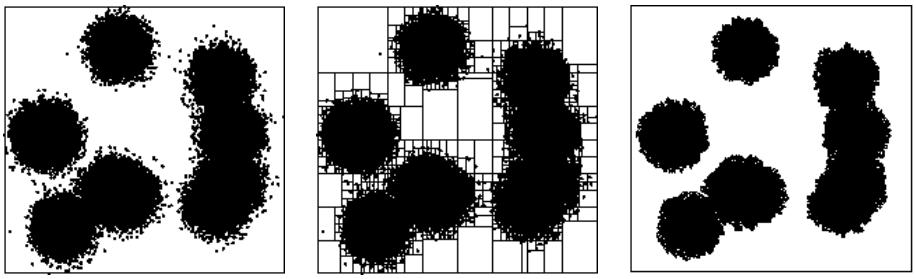


Fig. 8. An output example produced by our approximate DBC algorithm on 100K points in $\mathbb{R}^2$: (a) The input data set, (b) the space partition used by our BBD tree $T$, and (3) the output clusters.

## References

1. P.K. Agarwal, D. Eppstein, and J. Matoušek, Dynamic half-space reporting, geometric optimization, and minimum spanning trees, *Proc. 33rd Annual IEEE Symp. Found. Comput. Sci.*, 1992, pp. 80-89.
2. P.K. Agarwal and J. Erickson, Geometric range searching and its relatives. In: B. Chazelle, J. E. Goodman and R. Pollack (Eds.), *Advances in Discrete and Computational Geometry*, American Mathematical Society, Providence, RI, 1999, pp. 1–56.
3. P. K. Agarwal and C. M. Procopiuc, Exact and approximation algorithms for clustering, *Proc. 9th ACM-SIAM Symp. Discrete Algorithms*, 1998, pp. 658–667.
4. S. Arya and D.M. Mount, Approximate range searching, *Comput. Geom. Theory Appl.*, 17 (2000), pp. 135–152.
5. S. Arya and D.M. Mount, ANN: A library for approximate nearest neighbor searching, *2nd CGC Workshop on Computational Geometry*, 1997. Also, see http://www.cs.umd.edu/∼mount/.
6. S. Arya, D.M. Mount, N.S. Netanyahu, R. Silverman, and A. Wu, An optimal algorithm for approximate nearest neighbor searching in fixed dimensions, *J. ACM*, 45 (1998), pp. 891–923.

7. Y. Bartal, M. Charikar, and D. Raz, Approximating min-sum $k$-clustering in metric spaces, *Proc. 33rd Annual ACM Symp. on Theory of Computing*, 2001, pp. 11-22.
8. N. Beckmann, H.P. Kriegel, R. Schneider, and B. Seeger, The R*tree: An efficient and robust access method for points and rectangles, *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 1990, pp. 322-331.
9. J.L. Bentley, Multidimensional binary search trees used for associative searching, *Commun. of ACM*, 18 (9) (1975), pp. 509-517.
10. J.L. Bentley, $K$-d trees for semidynamic point sets, *Proc. 6th Annual ACM Symp. Comput. Geom.*, 1990, pp. 187-197.
11. A. Borodin, R. Ostrovsky, and Y. Rabani, Subquadratic approximation algorithms for clustering problems in high dimensional spaces, *Proc. 31st Annual ACM Symp. on Theory of Computing*, 1999, pp. 435–444.
12. M. Charikar and S. Guha, Improved combinatorial algorithms for facility location and $k$-median problems, *Proc. 40th IEEE Annual IEEE Symp. Found. Comput. Sci.*, 1999, pp. 378–388.
13. M. Charikar, S. Guha, $\acute{E}$. Tardos, and D. Shmoys, A constant-factor approximation algorithm for the $k$-median problem, *Proc. 31st Annual ACM Symp. on Theory of Computing*, 1999, pp. 1–10.
14. P. Drineas, A. Frieze, R. Kannan, S. Vempala, and V. Vinay, Clustering in large graphs and matrices, *Proc. 10th ACM-SIAM Symp. Discrete Algorithms*, 1999.
15. M. Ester, H.-P. Kriegel, J. Sander, M. Wimmer, and X. Xu, Incremental clustering for mining in a data warehousing environment, *Proc. 24th Int. Conf. on Very Large Databases*, 1998, pp. 323-333.
16. M. Ester, H.-P. Kriegel, J. Sander, and X. Xu, A density-based algorithm for discovering clusters in large spatial databases with noise, *Proc. 2nd Int. Conf. on Knowledge Discovery and Data Mining*, 1996, pp. 226-231.
17. M. Ester, H.-P. Kriegel, and X. Xu. A database interface for clustering in large spatial databases, *Proc. 1st Int. Conf. on Knowledge Discovery and Data Mining*, 1995, pp. 94-99.
18. M. Ester, H.-P. Kriegel, and X. Xu, Knowledge discovery in large spatial databases: Focusing techniques for efficient class identification, *Proc. 4th Int. Symp. On Large Spatial Databases*, *Lecture Notes in Computer Science*, Vol. 951, Springer, 1995, pp. 67-82.
19. P. Indyk, A sublinear time approximation scheme for clustering in metric spaces, *Proc. 40th Annu. IEEE Symp. Found. Comput. Sci.*, 1999, pp. 154–159 .
20. A.K. Jain and R.C. Dubes, *Algorithms for Clustering Data*, Prentice Hall, 1988.
21. R. Kannan, S. Vempala, and A. Vetta, On clusterings — good, bad and spectral, *Proc. 41st IEEE Annual IEEE Symp. Found. Comput. Sci.*, 2000.
22. R. Klein, O. Nurmi, T. Ottmann, and D. Wood, A dynamic fixed windowing problem, *Algorithmica*, 4 (1989), pp. 535-550.
23. N. Mishra, D. Oblinger, and L. Pitt, Sublinear time approximate clustering, *Proc. 12th ACM-SIAM Symp. Discrete Algorithms*, 2001.
24. R. Ostrovsky and Y. Rabani, Polynomial time approximation schemes for geometric $k$-clustering, *Proc. 41st IEEE Annual IEEE Symp. Found. Comput. Sci.*, 2000.
25. J. Sander, M. Ester, H.-P. Kriegel, and X. Xu, Density-based clustering in spatial databases: The algorithm GDBSCAN and its application, *Data Mining and Knowledge Discovery*, 2 (2) (1998), pp. 169-194.
26. B. Zhou, D.W. Cheung, and B. Kao, A fast algorithm for density-based clustering in large database, *Proc. 3rd Pacific-Asia Conf. on Methodologies for Knowledge Discovery and Data Mining*, 1999, pp. 338-349.