A faster algorithm for DBSCAN

Gunawan, A.

Award date:
2013

TECHNISCHE UNIVERSITEIT EINDHOVEN

Department of Mathematics and Computer Science

MASTER THESIS

A Faster Algorithm for DBSCAN

Ade Gunawan

Supervisors:
prof. dr. M.T. de Berg

Eindhoven, March 2013

Abstract

Clustering is the task of partitioning a set of objects into groups (called clusters) so that the objects in the same cluster are more similar to each other than to those in other clusters. This master thesis focus on improving the running time of DBSCAN, a density-based clustering algorithm. In density-based clustering, the clusters are defined by using a density threshold which is usually defined by the user. DBSCAN is widely used for clustering set of points because it can detect clusters with arbitrary shape and does not need to know the number of clusters beforehand. It only needs two parameters and runs fairly fast as it only requires linear number of range queries on the data. Theoretically, its worst-case running time complexity is $O(n^2)$. When a suitable indexing structure is used (e.g. R*-tree), its performance will be significantly improved and it is claimed that *in practice* it runs in $O(n \log n)$ time.

No known algorithm can compute clustering consistent with original DBSCAN algorithm and whose running time is $O(n \log n)$. Our main contribution is introducing a faster algorithm for DBSCAN which theoretically runs in $O(n \log n)$ time in the worst case. We experimentally investigate the performance of a simplified version of this algorithm and compare it to the original algorithm. The experimental results show that the new algorithm outperforms the original DBSCAN algorithm.

Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

Data mining is the process of discovering interesting patterns in large data sets. The discovery of these patters can help to understand the data and thus it can help to make decisions or do predictions based on the data. A very common task in data mining is clustering. Clustering is the task of partitioning a set of objects into groups (called clusters) so that the objects in the same cluster are more similar to each other than to those in other clusters. Most of the work is on clustering set of points and this is also what we focus on

One of the most widely used and most cited clustering algorithm is DBSCAN. DBSCAN[6] is a density-based clustering algorithm. The idea behind such algorithms is that the clusters are defined as areas of higher density separated by lower density areas. The main strength of density-based clustering algorithms compared to other type of clustering is that they can detect clusters with arbitrary shape and do not need to know the number of clusters beforehand. DBSCAN only requires two parameters which determine when a region is considered to be dense enough to belong to a cluster. It also runs fairly fast as it only requires linear number of range queries on the data. Theoretically, its worst-case running time complexity is $O(n^2)$. When a suitable indexing structure is used (e.g. R*-tree), its performance will be significantly improved and it is claimed that *in practice* it runs in $O(n \log n)$ time.

Many improvements have been proposed to have a better performance on DBSCAN algorithm. IDBSCAN[2] and FDBSCAN[7] try to make a sampling-based improvement with the intention to reduce the number of range queries. On the other hand, GF-DBSCAN[11] and GriDBSCAN[8] try to do different approach by partitioning the data into grid. In Chapter 2, we will describe these algorithms in more detail. We will show that although IDBSCAN, FDBSCAN, and GF-DBSCAN are faster than the original DBSCAN, their results are different from DBSCAN's, mainly on the border points. GriDBSCAN's result is the same as DBSCAN's but it will not improve the theoretical running time of the original DBSCAN since it uses the same method for range queries. Thus, there is no known algorithm that can compute exactly the same clustering as the original DBSCAN algorithm and whose running time is $O(n \log n)$ in the worst case.

In this project, we introduce a faster algorithm for DBSCAN which theoretically runs in $O(n \log n)$ time. This algorithm always produces the same result for a given set of points (and given parameter settings) in contrast to the original algorithm which is non-deterministic for border points. We experimentally investigate the performance of a simplified version of this algorithm and compare it to the original algorithm. The experimental results show that the new algorithm outperforms the original DBSCAN algorithm. The experiments also show that

the greater the value of ε, the faster the new algorithm is and the smaller the value of $minPts$, the faster the new algorithm is.

Chapter 2

Clustering Algorithms

Clustering algorithm can be categorized based on their cluster model. In this section, we will give a general overview of three different clustering models.

2.1 Connectivity-based clustering

Connectivity-based clustering tries to connect objects to form clusters based on their distance. This model usually represent its result by using a dendrogram, i.e., a tree that iteratively splits the data into smaller subsets until each subset consists of only one object. For this reason, this model is often called hierarchical clustering. The dendrogram can either be created from the leaves up to the root (agglomerative approach) or from the root down to the leaves (divisive approach) by merging or dividing clusters at each step. Examples of algorithms which based on this model are SLINK[10] (which use minimum distance as distance between clusters) and CLINK[4] (which use maximum distance as distance between clusters).

The advantage of this model is that it is easy to understand. However, the results are not always easy to use as it provides a hierarchy which still needs to be further processed to find appropriate clusters. Another weakness of this model is that it does not have a notion of noise. This weakness makes this model not robust to outliers. Also, the fastest known running time complexity that can be achieved using this model is $O(n^2)$[10][4] which is slow for large data. If we use minimum distance as distance between clusters, there is a way to improve this bound by creating a minimum spanning tree, giving $O(n \log n)$ as the new bound.

2.2 Centroid-based clustering

Centroid-based clustering tries to construct a partition of the data into a set of k clusters for a given parameter k. The partitioning algorithm usually starts with an initial partition of the data and then uses an iterative strategy to optimize an objective function. Each cluster is represented by the gravity center of the cluster (k-means algorithms) or by one of the objects of the cluster located near its center (k-medoid algorithms). Consequently, partitioning algorithms use a two-step procedure. First, determine k representatives minimizing the objective function. Second, assign each object to the cluster with its representative closest to the considered object. Because of the second step, the resulting partition will be similar to Voronoi diagram where each cluster will be contained in one Voronoi cell.

This model has two major weaknesses. First, we need define the number of clusters k beforehand. Second, since the clusters are basically Voronoi cells, the shape of the clusters has to be convex.

2.3 Density-based clustering

In density-based clustering, clusters are defined as areas of higher density separated by lower density areas. The main strengths of density-based clustering algorithm compared to other types of clustering is that it has a notion of noise, can detect clusters with arbitrary shape, and does not need to know the number of clusters beforehand. Two of the most popular density-based clustering algorithms are DBSCAN[6] and OPTICS[1]. OPTICS is an improvement of DBSCAN which addresses one weakness of DBSCAN: the problem of detecting clusters in data with large differences in densities. However, unlike DBSCAN which produces a clear clustering result by assigning each point (except noise) to a cluster id, OPTICS produces its result as an ordering of points. This ordering is usually further processed to produce a reachability-plot (a kind of dendrogram). Then, the user needs to manually decide which parts of the plot are the clusters by defining a distance threshold for each potential cluster. In this thesis, we focus only on DBSCAN.

2.3.1 DBSCAN

One of the most widely used and most cited clustering algorithm is DBSCAN[6]. It requires two parameters (ε and $minPts$) and works by distinguishing core points, border points, and noise.

2.3.1.1 Definitions

Definition 1. *The* ε-neighborhood *of a point p, denoted by $N_\varepsilon(p)$, is defined by $N_\varepsilon(p) = \{q \in D | dist(p,q) \leq \varepsilon\}$, where D is a set of points and $dist(p,q)$ is a distance function e.g. Euclidian distance, between p and q.*

Definition 2. *A point p is a* core point *if $|N_\varepsilon(p)| \geq minPts$*

Definition 3. *A point p is* directly density-reachable *from a point q with respect to ε and $minPts$ if $p \in N_\varepsilon(q)$ and q is a core point.*

Definition 4. *A point p is a* border point *if p is directly density-reachable from a core point q and $|N_\varepsilon(p)| < minPts$*

Definition 5. *A point p is* density-reachable *from a point q with respect to ε and $minPts$ if there is a chain of points $p_1, ..., p_n$, with $p_1 = q$ and $p_n = p$ such that p_{i+1} is directly density-reachable from p_i.*

Definition 6. *A point p is* density-connected *to a point q with respect to ε and $minPts$ if there is a point o such that both, p and q are density-reachable from o.*

Definition 7. *Let D be a set of points. A* cluster *C with respect to ε and $minPts$ is a non-empty subset of D satisfying the following conditions:*

1. *$\forall p, q$: if $p \in C$ and q is density-reachable from p with respect to ε and $minPts$, then $q \in C$.*
2. *$\forall p, q \in C : p$ is density-connected to q with respect to ε and $minPts$.*

Definition 8. *A point p is a* noise *if it is neither a core point nor a border point. This implies that noise does not belong to any clusters.*

2.3.1.2 Algorithm

First, DBSCAN starts with an arbitrary point p and retrieves all points in its ε-neighborhood. If the number of points in the ε-neighborhood is sufficient (with respect to $minPts$), a new cluster is started. This cluster is then expanded until it retrieves all points which are *density-reachable* from p. If the number of points in the ε-neighborhood is not sufficient, p will be marked as noise. Note that this point might later be found in a sufficiently sized ε-environment of a different point and hence be made part of a cluster. The full algorithm for DBSCAN is shown in Algorithm 1. The function $RangeQuery(p, \varepsilon)$ reports all points $q \in D$ such that $dist(p, q) \leq \varepsilon$.

Algorithm 1 DBSCAN(D, ε, $minPts$)

```
1: {D is a set of unclassified points}
2: {ε is the maximum distance}
3: {minPts is the minimum points to form a cluster}
4: Initialize cluster id C = 0
5: for each unclassified point p ∈ D do
6:    N_ε(p) = RangeQuery(p, ε)
7:    if |N_ε(p)| ≥ minPts then
8:       Set p's cluster id to C
9:       ExpandCluster(p, N_ε(p), C, ε, minPts)
10:      C ← C + 1
11:   else
12:      Label p as noise
```

Algorithm 2 $ExpandCluster(p, NeighborPts, C, \varepsilon, minPts)$

```
1: for each point q ∈ NeighborPts do
2:    if q is unclassified then
3:       N_ε(q) = RangeQuery(q, ε)
4:       if |N_ε(p)| ≥ minPts then
5:          NeighborPts = NeighborPts ∪ N_ε(q)
6:    if q does not belong to any cluster then
7:       Set q's cluster id to C
```

The running time of DBSCAN depends on how many times the function $RangeQuery(p, \varepsilon)$ is called. We can see that DBSCAN calls this function exactly once for each point. Therefore,

the total running time complexity for DBSCAN is $O(n^2)$ using the most naive way to do a range query (by simply checking the distance of all other $n-1$ points in D). In [6], it is claimed that range queries can be supported efficiently by using spacial access methods such as R*-tree. By using it, the average running time complexity can be reduced to $O(n \log n)$. They claimed this to be true because the height of the tree is in $O(\log n)$, and a query with a small query range (such as ε-neighborhood) has to traverse only a limited number of paths in the tree. However, there is no theoretical guarantee that this holds since we cannot be sure that every range query will traverse a limited number of paths in the tree especially if ε is large. This fact is the main motivation for this project. Another fact is that we noticed that DBSCAN can be non-deterministic for border points, since there can be a border point which is *directly density-reachable* from more than one cluster. In this case, that point will be assigned to the cluster which encounters it first. Since DBSCAN starts with an arbitrary point, the result of DBSCAN on this kind of border points can be different depending on the order in which the points in D are given.

Many improvements have been proposed to have a better performance on DBSCAN algorithm. We will describe four interesting improvements which have been done to DBSCAN. These improvements can be divided into two categories: sampling-based improvements and partition-based improvements. Sampling-based improvement is a method which tries to reduce the number of range queries by skipping some queries from specific points. IDBSCAN[2] and FDBSCAN[7] use this kind of improvement. Partition-based improvement is a method which partitions the data into groups (using a grid) and tries to process each group separately. GF-DBSCAN[11] and GriDBSCAN[8] use this kind of improvement.

2.3.2 IDBSCAN

IDBSCAN is motivated by the fact that one can speed up DBSCAN by using two approaches: by reducing the query time or by reducing the number of queries. Since no good progress can be made for the first approach, IDBSCAN attempted to exploit the second approach.

DBSCAN expands a cluster when it finds a core point p. It expands the cluster by carrying out range queries for every point contained in $N_\varepsilon(p)$. However, the neighborhoods of the point contained in $N_\varepsilon(p)$ will intersect with each other. Suppose $q \in N_\varepsilon(p)$, if $N_\varepsilon(q)$ is covered by the neighborhoods of other point in $N_\varepsilon(p)$, then the range query for q can be omitted. Figure 2.1 shows an example of this case. In Figure 2.1, $N_\varepsilon(y)$ is completely covered by $N_\varepsilon(a)$, $N_\varepsilon(b)$, and $N_\varepsilon(c)$. Therefore, executing a range query on y is not needed and can simply be omitted.

IDBSCAN wants to exploit this fact by sampling some representatives rather than take all of the points in a core point's neighborhood. It selects the representatives by first dividing the core point's neighborhood into eight parts (in 2D case) as shown in Figure 2.2 We have eight distinct points marked on the circle as $A, B, ..., H$. These points are called $MarkedBoundaryObjects$ (MBO). For each MBO, find the nearest point and add it as representative. Since we can select the same point for different MBOs, the total number of representatives will be at most eight. IDBSCAN basically will only process these representatives and ignore all other points inside the core point's neighborhood. For example, in Figure 2.1, the points a, b, and c will be the representatives and the range query for y will be omitted.

However selecting representatives like this can produce some cases in which the resulting cluster of IDBSCAN and the original DBSCAN are different. First, it should be clear that all the representatives must be core points. If one of them is not a core point, the points inside its ε-neighborhood will not be expanded, giving us a wrong clustering result. Even though

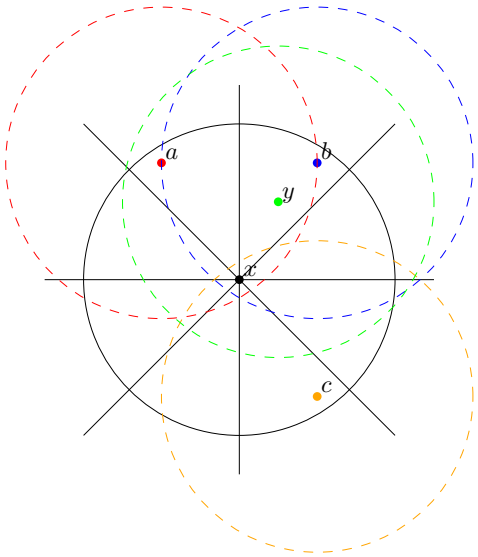


Figure 2.1: Intersecting neighborhoods

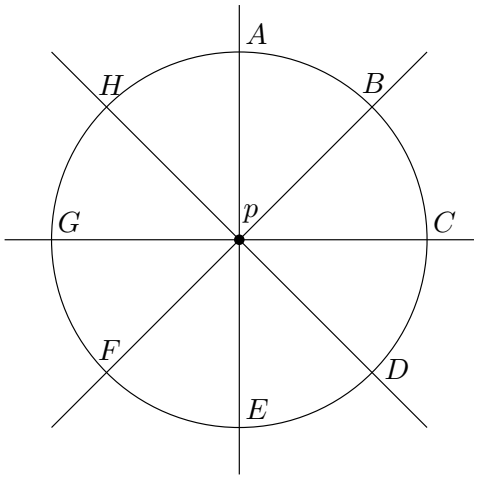


Figure 2.2: Circle with eight distinct points

we can ensure that all of the representatives are core points, there is a counterexample which shows that the resulting cluster of IDBSCAN and the original DBSCAN are different. An example is shown in Figure 2.3. In this figure, the points a, y, and b are in $N_\varepsilon(x)$. IDBSCAN will select only a and b as representatives and omit the range query for y. This causes a and b to not expand the cluster to z which is outside of $N_\varepsilon(a)$ and $N_\varepsilon(b)$. Eventually, z will be chosen as a starting point to expand a new cluster but after finding that $N_\varepsilon(z)$ only contains one point (y), z will be marked as noise. Of course this is not correct. By the definitions stated in Section 2.3.1.1, z should be a border point because it is *directly-density-reachable* from y. Therefore, we can conclude that selecting representatives as IDBSCAN does is incorrect, that is, it will produce different clusters from the original DBSCAN's result.

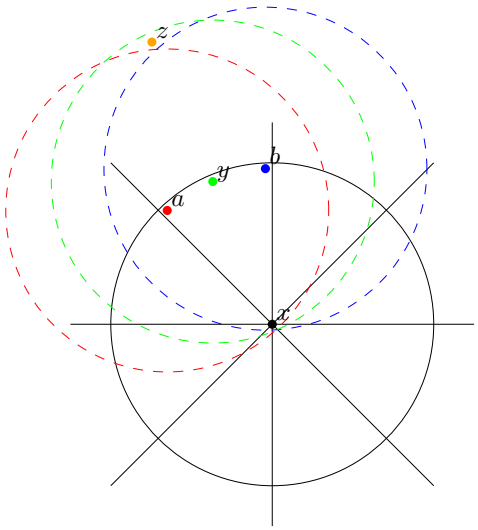


Figure 2.3: A counterexample for IDBSCAN, $minPts = 2$

2.3.3 FDBSCAN

FDBSCAN has the same motivation as IDBSCAN, namely that, the neighborhoods of points in the same ε-neighborhood will intersect with one another. Therefore, it is not necessary to execute a range query for every points. The difference is on how it selects the representatives. In contrast to IDBSCAN, which is still selecting representatives inside the ε-neighborhood, FDBSCAN tries to omit more range queries by selecting representatives outside of the ε-neighborhood.

FDBSCAN first sorts the points by a certain coordinate (e.g. x-coordinate in 2D). Then, it selects the first point in the order and performs a range query. If the ε-neighborhood does not contain at least $minPts$ points, mark that point as noise. If the ε-neighborhood contains at least $minPts$ points, then mark all of those points as one cluster and omit all range queries for those points. Then, move to the next point in order. If the point already belongs to a cluster, skip it and move to the next point. If not, then do a range query and check whether it intersects with other cluster and merge if necessary.

However, there is also some counterexamples for this algorithm. One of those is shown in Figure 2.4. In this counterexample, the points are sorted in x-coordinate from left to right. FDBSCAN will first do a range query on x and find only one point (a). Since $minPts = 3$, x will be marked as noise. The algorithm then continues to the next point y and performs a range query on it. Since $|N_\varepsilon(a)| \geq minPts$, the points y, a, b, c, and d will be assigned in one cluster giving the end result of one cluster of four points and one noise point. This is obviously not correct since the original DBSCAN will detect x not as a noise but as a border point instead, giving the end result of one cluster of five points.

2.3.4 GF-DBSCAN

GF-DBSCAN tries to improve the original DBSCAN by not only reducing the number of queries but also reducing the range query time. It reduces the number of queries by doing something similar to what FDBSCAN does. The only difference is that GF-DBSCAN does not sort the points. However, this will not avoid the incorrectness that happens to FDBSCAN.

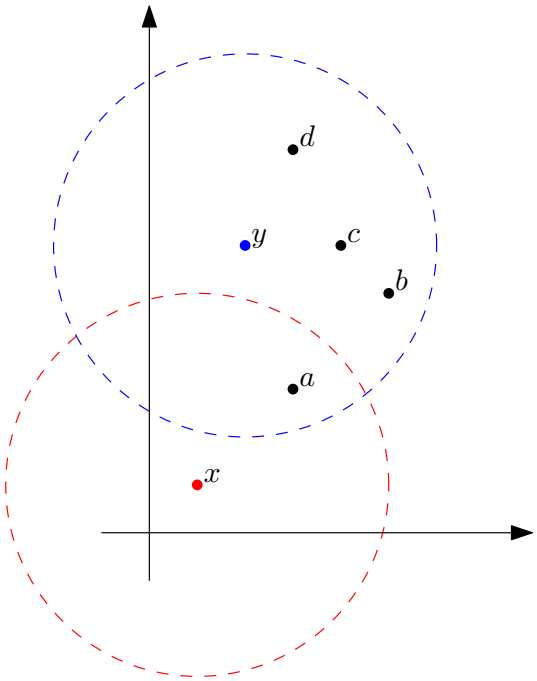


Figure 2.4: A counterexample for FDBSCAN, $minPts = 3$

Therefore, by similar counterexample, we can prove that GF-DBSCAN can produce different result from the original DBSCAN's result.

They also introduce a method to reduce the range query time by partitioning the data into grid. Each cell in that grid will have size ε such that to do one range query, it only needs to access a constant number of neighboring cells (nine for 2D as shown in Figure 2.5). This

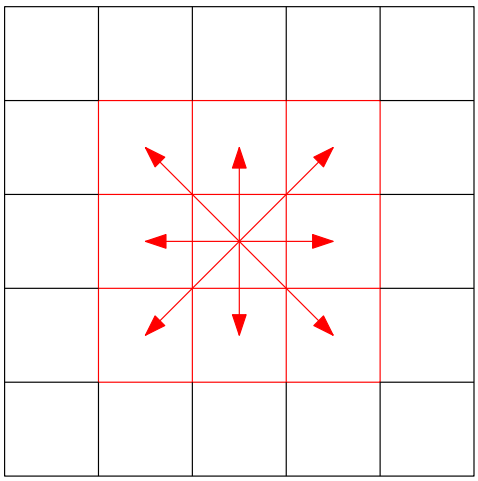


Figure 2.5: Range query on neighboring cells

grid-based method is somewhat similar to our proposed algorithm in Chapter 3 although the grid size and global algorithm are different.

2.3.5 GriDBSCAN

GriDBSCAN is motivated by EnhancedDBSCAN[5] which tries to improve DBSCAN by using a divide-and-conquer approach. The basic idea is to partition the data into groups by using CLARANS[9], apply DBSCAN to each group, and merge the results. GriDBSCAN tries to improve the partitioning and merging step by constructing a grid which partitions the data. The grid size w is left for user to decide but they recommend that $w > 2\varepsilon$ to have a good performance.

The interesting thing that GriDBSCAN does is that they define one group as one cell plus points around the cell in the ε-enclosure of the cell (Figure 2.6). This implies that some

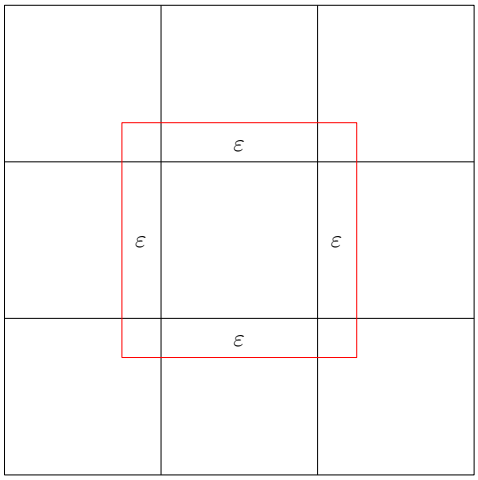


Figure 2.6: ε-enclosure of a cell

points will be included in multiple groups and hence may be assigned to multiple clusters. In this case, if the point is a core point, then all of the clusters assigned to it have to be merged. In other words, one core point is sufficient to merge all clusters to which it was assigned. In [8], they prove that GriDBSCAN will produce the same result as the original DBSCAN's result. They also stated that GriDBSCAN has a theoretical speedup of $C/2^{2d}$, where C is the number of cells and d is the dimensionality. The only drawback of this method is that it adds one more input parameter which should be decided by the user.

Chapter 3

The New Algorithm

In this chapter, we introduce a new algorithm for DBSCAN in $\mathbf{R}^2$ which has a theoretical running time complexity of $O(n \log n)$. This algorithm can be divided into four main steps. First, partition the data using a grid. Second, determine all core points. Third, merge *density-connected* core points into clusters. Fourth, determine border points and noise.

3.1 Partitioning

This step is basically constructing a grid and assigning each data point to the grid cell it lies in. Each cell's diagonal length is ε which implies that the grid cell will have width $= \varepsilon/\sqrt{2}$. The motivation behind this cell size is that if the number of points inside a cell is greater than $minPts$, we will be able to directly mark all of those points as core points. This is true because the maximum distance that any two points can have inside a cell is ε, so if there are more than $minPts$ points inside the cells, the ε-neighborhood of any point inside that cell will contain at least $minPts$ points.

Before constructing the grid, we have to find four values, those are: minimum x-coordinate, maximum x-coordinate, minimum y-coordinate, and maximum y-coordinate. From these values, we can determine the number of rows and columns for our grid. Then, we can compute, for each point, the cell which it belongs to. In the case of a point on the border of a cell, we assign it in the top-right cell as shown in Figure 3.1, where x will be assigned to cell A and y will be assigned to cell B. If at least one point is exactly on the top-most border or right-most border of the grid, we will add an extra row / column to prevent it to be assigned to the outside of the grid. For example, in Figure 3.1, an extra row is added because of point p and an extra column is added because of point q. An algorithm to construct this is shown in Algorithm 3.

Lemma 3.1.1. *Algorithm 3 runs in $O(n + nCells)$ time in the worst case.*

Proof. Line 4 can be done in linear time by simply scanning P. Similarly, line 8-10 can also be done in linear time. The running time for Line 7 is $O(nCells)$. Therefore, the worst-case running time complexity for Algorithm 3 is $O(n + nCells)$. □

In the case of a very small ε and/or a very sparse data, i.e. very large $maxX - minX$ and $maxY - minY$, $nCells$ can be very large. This will obviously make the algorithm less efficient than it should be. We want to avoid this problem by limiting the number of cells to at most $O(n)$.

Algorithm 3 ConstructGrid(P, ε,)

1: $\{P$ is a set of points$\}$
2: $\{\varepsilon$ is the maximum distance$\}$
3: $cellWidth \leftarrow \varepsilon/\sqrt{2}$
4: Initialize $minX$, $maxX$, $minY$, and $maxY$ from P
5: $nRows \leftarrow \lfloor \frac{maxX - minX}{cellWidth} + 1 \rfloor$
6: $nCols \leftarrow \lfloor \frac{maxY - minY}{cellWidth} + 1 \rfloor$
7: Initialize G as an empty grid with size $nCells = nRows \times nCols$
8: **for** each point $p \in P$ **do**
9: Add point p to $G \left[\lfloor \frac{p.x - minX}{cellWidth} + 1 \rfloor \right] \left[\lfloor \frac{p.y - minY}{cellWidth} + 1 \rfloor \right]$
10: **return** G

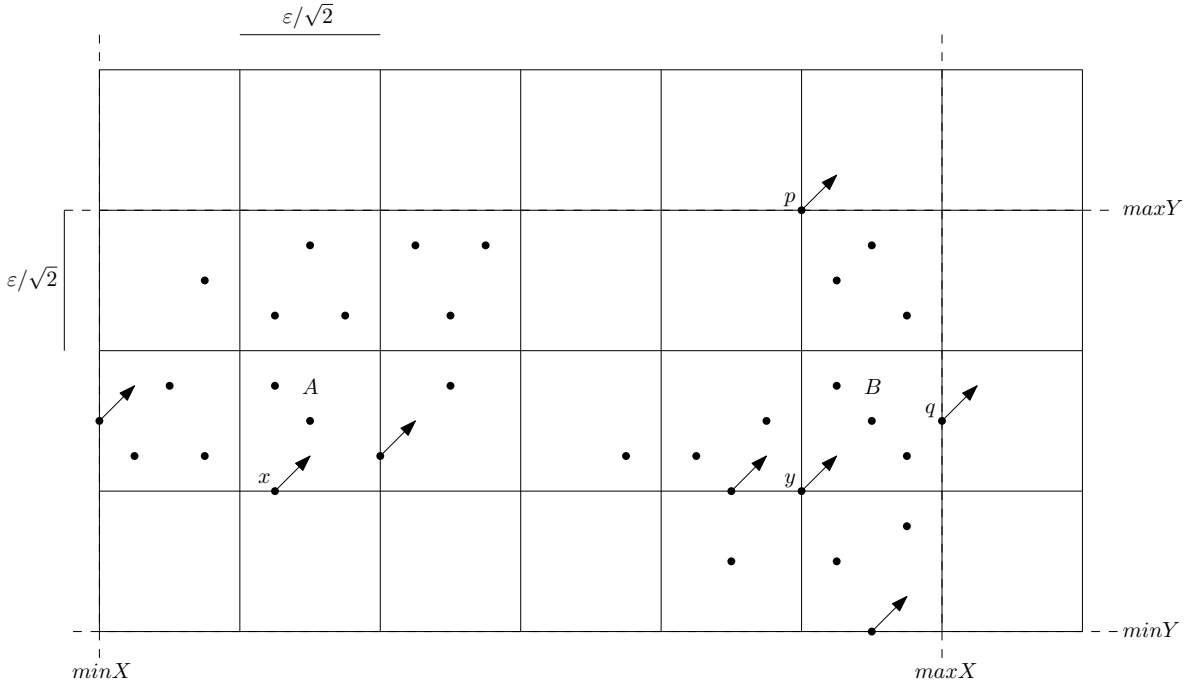


Figure 3.1: An example of the grid on a data set

3.1.1 Hash table

In a regular grid, we can have a very large number of cells because there can be many empty cells (i.e. cells without any point). We want to avoid this problem by using a hash table. First, we can define the *key* of point p as follows:

$$p.key = \lfloor \frac{p.x - minX}{cellWidth} + 1 \rfloor \cdot (nCols + 1) + \lfloor \frac{p.y - minY}{cellWidth} + 1 \rfloor$$

Note that it is simply $i \cdot (nCols + 1) + j$ if we know that a point p lies in cell $G[i, j]$. Now, we initialize a hash table $H[0..n]$ and define a hash function $h : \{0..nCells\} \rightarrow \{0..n\}$. This function should map the *key* that we defined before to the memory location of the cell. By

using a good hash function h, we can ensure that we will have Insert and Search in $O(1)$ time.

3.1.2 Sorting-based algorithm

The second approach that we want to introduce is creating a non-regular "grid" as shown in Figure 3.2. To build this, first, we need to sort all the points in x-coordinate. Then, we scan the points and take the first strip whose width is $\varepsilon/\sqrt{2}$. Sort this strip in y-coordinate and take the first box whose height is $\varepsilon/\sqrt{2}$. Find the next point and make another box from it until there is no point left in the strip. Do the same for the next strips until there are no more data. The full algorithm is shown in Algorithm 4. Note that in the implementation, we also compute and store the information of the boundary for each box. We can do this while scanning the points. With this kind of partitioning, we can ensure that each box will contain at least one point. Therefore, the number of boxes will be bounded by the number of points.

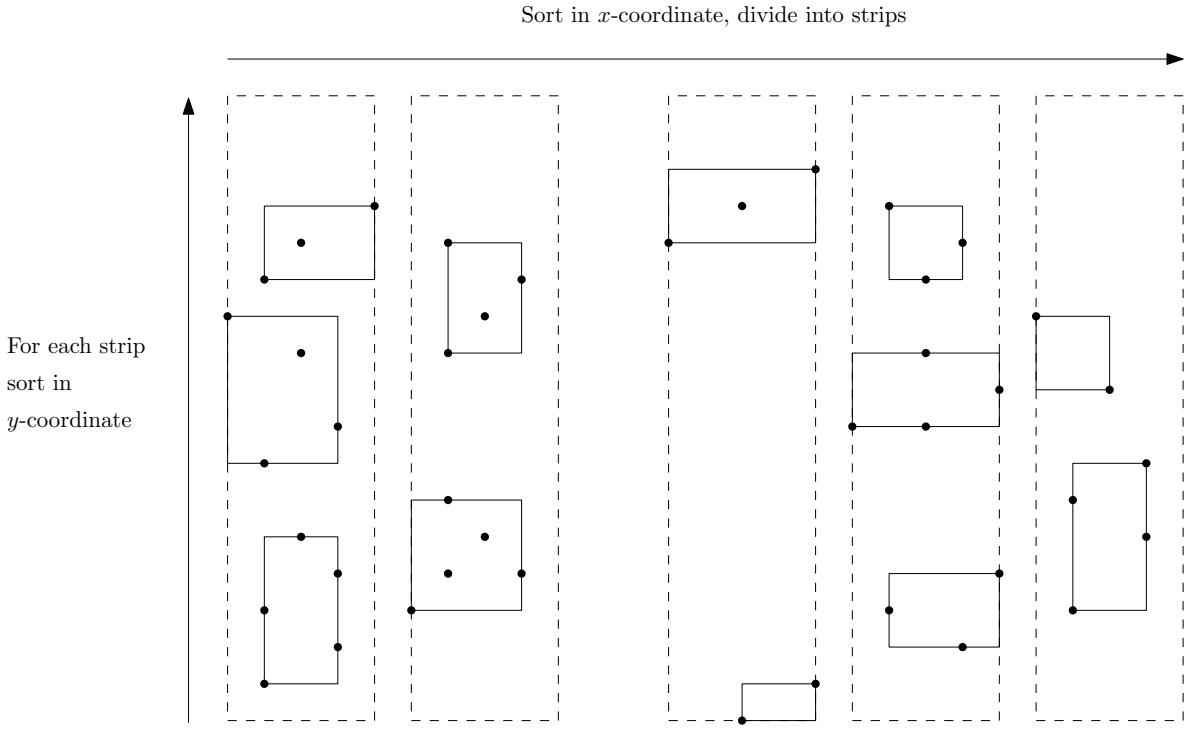


Figure 3.2: Non-regular grid

Lemma 3.1.2. *The running time complexity for Algorithm 4 is $O(n \log n)$.*

Proof. First, It is clear that line 5 takes $O(n \log n)$ time since it is a simple sorting. Now, we only need to compute the running time for line 8-13 (sorting strips in y-coordinate). Let $Strip$ be a set of strips and $Strip_i$ is the i-th strip consisting of $|Strip_i|$ points, we know that the total running time for sorting all strips is $O(\sum_{i=1}^{|Strip|} |Strip_i| \log |Strip_i|)$. Therefore, the

Algorithm 4 ConstructBoxes(P, ε)

```
1: {P is a set of points}
2: {ε is the maximum distance}
3: cellWidth ← ε/√2
4: Initialize G as an empty set of boxes
5: Sort P in x-coordinate
6: Initialize empty set of points strip
7: Add the first point P_1 to strip
8: for i ← 2 to |P| do
9:    q ← first point of strip
10:   if P_i.x > q.x + cellWidth then
11:      AddStripToGrid(G, strip, cellWidth)
12:      Remove all points from strip
13:   Add point P_i to strip
14: AddStripToGrid(G, strip, cellWidth)
15: return G
```

Algorithm 5 AddStripToGrid(G, *strip*, $cellWidth$,)

```
1: {G is a grid}
2: {strip is a set of points}
3: {ε is the maximum distance}
4: Sort strip in y-coordinate
5: Initialize empty set of points Box
6: Add the first point of strip to Box
7: for i ← 2 to |P| do
8:    q ← j-th point of strip
9:    if q.y > Box_1.y + cellWidth then
10:      Add Box to G
11:      Remove all points from Box
12:   Add q to Box
13: Add Box to G
```

total running time for Algorithm 4 is:

$$
\begin{aligned}
O(n\log n) + O(\sum_{i=1}^{|Strip|} |Strip_i|\log|Strip_i|) &\leq O(n\log n) + O(\sum_{i=1}^{|Strip|} |Strip_i|\log n) \\
&\leq O(n\log n) + O(\log n \cdot \sum_{i=1}^{|Strip|} |Strip_i|) \\
&\leq O(n\log n) + O(\log n \cdot n) \\
&\leq O(n\log n)
\end{aligned}
$$

□

3.2 Determining core points

In this step, we are going to iterate through all non-empty cells. Note that if we use Algorithm 3 when partitioning, it will gives us some empty cells inside the grid. To get the list of all non-empty cells, we can simply create a new list, scan all the cells and add only the non-empty cells to the list. This process takes $O(n + nCells)$ which is asymptotically the same as the Algorithm 3 itself.

To find all the core points inside a cell, first, we have to check whether the number of points inside that cell is greater than $minPts$. If so, mark all those points as core points. If not, we have to check each point inside that cell and determine whether it is a core point or not. To determine whether a point p is a core point or not, we need to check some of the points inside the *neighboring cells* $\mathcal{N}_\varepsilon(c)$ as shown in Figure 3.3.

Definition 9. *The set of* neighboring cells *of a cell c, denoted by $\mathcal{N}_\varepsilon(c)$, is defined by $\mathcal{N}_\varepsilon(c) = \{c' : c' \text{is a grid cell such that } dist(c, c') \leq \varepsilon\}$. Note that $c \in \mathcal{N}_\varepsilon(c)$.*

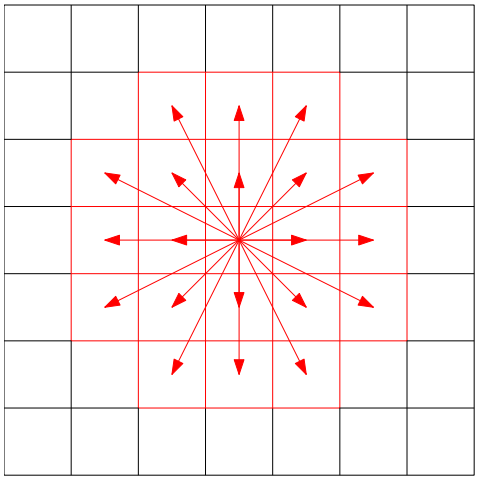


Figure 3.3: Neighboring cells $\mathcal{N}_\varepsilon(c)$ for range query

We can see that for the 2 dimensional regular grid, we need to check at most 21 cells. Note that for the "grid" that we introduced in Section 3.1.2, we can still ensure that only $O(1)$ boxes need to be checked. We compute the distance between each point in those cells to p. This terminates either when all of the points inside of those cells has already been checked or we have found $minPts$ points that have distance less than or equal to ε. The full algorithm is shown in Algorithm 6. Note that $|c|$ denotes the number of points inside a cell c.

Lemma 3.2.1. *The running time complexity for Algorithm 6 is $O(minPts \cdot n)$.*

Proof. The running time of the algorithm can be expressed as follows. Let C_1 be the set of cells c with $|c| > minPts$, and let C_2 be the set of cells with $c \leq MinPoints$. Note that for each cell c in C_1 we spend only $O(|c|)$ time. For each c in C_2 we spend $\sum_{c' \in \mathcal{N}_\varepsilon(c)} O(|c'|)$ time for each point in c, so $\sum_{c' \in \mathcal{N}_\varepsilon(c)} O(|c||c'|)$ in total. Hence, the total running time is

$$\sum_{c \in C_1} O(|c|) + \sum_{c \in C_2} \sum_{c' \in \mathcal{N}_\varepsilon(c)} O(|c||c'|) \leq O(n) + O(minPts) \cdot \sum_{c \in C_2} \sum_{c' \in \mathcal{N}_\varepsilon(c)} O(|c'|)$$

Algorithm 6 DetermineCorePoints(G, ε, $minPts$)

```
1: {G is a grid of data with cell width = ε/√2}
2: {ε is the maximum distance}
3: {minPts is the minimum points to form a cluster}
4: for each non-empty cell c ∈ G do
5:   if |c| > minPts then
6:     Mark all points p ∈ c as core points.
7:   else
8:     for each point p ∈ c do
9:       nPoints ← 0
10:      for each cell nc ∈ N_ε(c) do
11:        for each point q ∈ nc do
12:          if dist(p, q) ≤ ε then
13:            nPoints ← nPoints + 1
14:            if nPoint ≥ minPts then
15:              Mark p as a core point
16:              break
17:        if nPoint ≥ minPts then
18:          break
```

Now, consider a fixed cell c'. Note that c' is only counted for cells c such that $c' \in \mathcal{N}_\varepsilon(c)$ which is equivalent to $c \in \mathcal{N}_\varepsilon(c')$. Hence we can reformulate the total running time as follows:

$$\begin{aligned} O(n) + O(minPts) \cdot \sum_{c \in C_2} \sum_{c' \in \mathcal{N}_\varepsilon(c)} O(|c'|) &\leq O(n) + O(minPts) \cdot \sum_{c' \in C_1 \cup C_2} \sum_{c \in \mathcal{N}_\varepsilon(c')} O(|c'|) \\ &\leq O(n) + O(minPts) \cdot \sum_{c' \in C_1 \cup C_2} O(|c'|) \cdot |\mathcal{N}_\varepsilon(c)| \\ &\leq O(n) + O(minPts) \cdot O(n) \cdot 21 \\ &\leq O(minPts \cdot n) \end{aligned}$$

□

3.3 Merging clusters

This step is done based on the fact that if the distance between two core points in two different cells is at most ε, these two points belong to the same cluster. A simple example is shown in Figure 3.4. This figure shows two *neighboring cells*, each contains four points. Since $minPts = 3$, we know that all of these points are core points. If the distance between a and b is at most ε, we can conclude that all of these eight points belong to the same cluster. Otherwise, we conclude that there are two clusters with four points each.

To describe how we connect all the cells that have at least one core point inside, we can consider these cells as nodes in a graph. Now, we want to connect every two nodes which should be in one cluster. More precisely, we add an edge between the nodes for cells c and c' if and only if there are core points $q \in c$ and $q' \in c'$ such that $dist(q, q') \leq \varepsilon$. The final clusters (of core points) then correspond to the connected components of this graph.

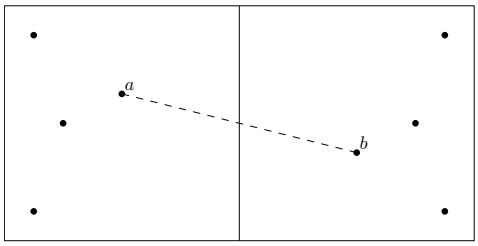


Figure 3.4: Merging step example, $minPts = 3$

Lemma 3.3.1. *The running time complexity for merging cluster is $O(n \log n)$.*

Proof. Iterating through all non-empty cells takes $O(n)$ time. When checking whether a cell should be connected to another cell or not, we can use a Voronoi diagram so that every time we want to find the nearest point from a neighboring cell's points, it only takes $O(\log n)$ time giving us the total of $O(n \log n)$ time. After this, the connected components can be found in linear time using a standard depth-first search[3]. □

3.4 Determining border points and noise

The last step is to determine border points and noise. This is done by iterating through all non-core points. A non-core point p is a border point of cluster c if there exists a at least one core point in $N_\varepsilon(p)$ and the closest core point to p belongs to a cluster c. All the points which are not either core or border point are therefore noise. The full algorithm is shown in 7.

Algorithm 7 DetermineBorderPoint(G, ε)

```
1: {G is a grid of data with cell width = ε/√2}
2: {ε is the maximum distance}
3: for each non-empty cells c ∈ G do
4:   for each point p ∈ c do
5:     if p is not a core point then
6:       q ← NULL
7:       for each cell c' ∈ N_ε(c) do
8:         tempPoint ← NearestCorePoint(p, c')
9:         if dist(p, tempPoint) ≤ dist(p, q) then
10:          q ← tempPoint
11:      if q ≠ NULL then
12:        Assign point p to q's cluster
13:      else
14:        Mark p as noise
```

Lemma 3.4.1. *The running time complexity for Algorithm 7 is $O(minPts \cdot n)$.*

Proof. Algorithm 7 is similar to Algorithm 6 but it only checks points in neighboring cells for non-core points instead of all points. Hence Algorithm 7 cannot cost more than Algorithm 6. □

3.5 Putting it all together

By combining Lemma 3.1.2, 3.2.1, 3.3.1 3.4.1, we get the following result.

Theorem 3.5.1. *There is an algorithm for clustering a set of n points in the plane that runs in $O(minPts \cdot n + n \log n)$ time and produces a clustering consistent with DBSCAN, where $minPts$ is the parameter of DBSCAN specifying the minimum number of points in the neighbor of a core point.*

Chapter 4

Experiments

In the previous chapter we described an algorithm for DBSCAN with $O(n \log n)$ worst-case running time. In this chapter we will experimentally compare our new algorithm, more precisely, the one using a regular grid, to two implementations of the original DBSCAN algorithm. The first implementation is the straightforward $O(n^2)$ algorithm. The second is an implementation which we call "original with grid". This algorithm is simply the same as the original DBSCAN algorithm but now we use a grid to perform a range query. More precisely, for a point $p \in c$, the procedure $RangeQuery(p, \varepsilon)$ checks all the points inside cells $c' \in \mathcal{N}_\varepsilon(c)$. We implemented this instead of an R-tree because we believe that this outperforms an R-tree in this case. The reason is that unlike an R-tree, our grid-based range search is targeted to range searching with a disc of radius ε, so only $O(1)$ cells visited.

Another thing which we want to point out is that we did not implement Voronoi diagrams for merging clusters (Section 3.3). The reason behind this is that building a Voronoi diagram itself takes significant time. Also, although this implies that this process runs in $O(n^2)$ time but in practice it runs in $O(n)$ time because we have $O(1)$ of points inside most cells.

We implemented all of the algorithms using Java programming language and NetBeans as IDE. All the experiments were done on a computer with this specification:

- Processor: Intel Core i5 CPU M540 @2.53GHz
- RAM: 4 GB
- Operating system: Windows 7 32-bit

4.1 Data sets

We generate five types of random data sets for our experiments.

1. Uniform fill
   Created by simply generating uniformly random points inside a bounding box of size $\sqrt{m} \times \sqrt{m}$, where $m = \cdot n$ and n is the total number of points. This gives us $(\varepsilon/0.9)^2/2$ expected points inside a cell. Using $n = 1000000$ and $\varepsilon = 2$, we can get an insight to the data by showing the distribution of number of points inside cell as shown in Figure 4.1. We can see that the number of empty cells is not too many (more precisely 10.84%). The maximum number of points inside a cell can vary from 12 to 15 but the number of cells containing this many points is very small. Note that we only show the results until number of points equals 8 because for larger numbers, the number of cells is so

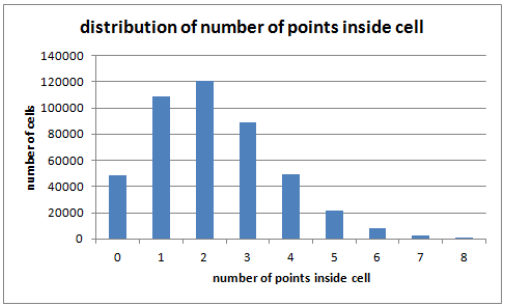


Figure 4.1: Distribution of number of points inside cell for uniform fill data

small that we cannot see the bar. Typically, the total number of cells with more than 8 points is less than 0.06%.

2. Uniform discs
   Let n be the total number of points and $m = 0.9 \cdot n$. We generate m points uniformly at random inside five discs and $n - m$ points as noise inside a bounding box of size approximately $5\sqrt{m} \times 5\sqrt{m}$. The discs have diameter of $\sqrt{m}$, thus making their area $\frac{1}{4} \cdot \pi\sqrt{m} \cdot \sqrt{m}$. Note that the discs do not intersect with each other. Using $n = 1000000$ and $\varepsilon = 4$ we get a distribution shown in Figure 4.2. The number of empty cells is very large (82.12%, not shown in the figure) because now we have larger area (25 times the uniform fill's area). The number of cells containing one point is relatively large because there are many cell with only noise inside (remember that we add 10% noise points). The maximum number of points inside a cell varies from 12 to 13 (not shown in the figure) but the number of cells containing this many points is very small. As before, we only show the results until number of points equals 7 because for larger numbers, the number of cells is so small that we cannot see the bar. Typically, the total number of cells with more than 7 points is less than 0.03%.

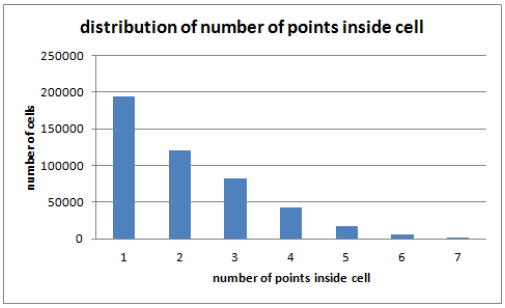
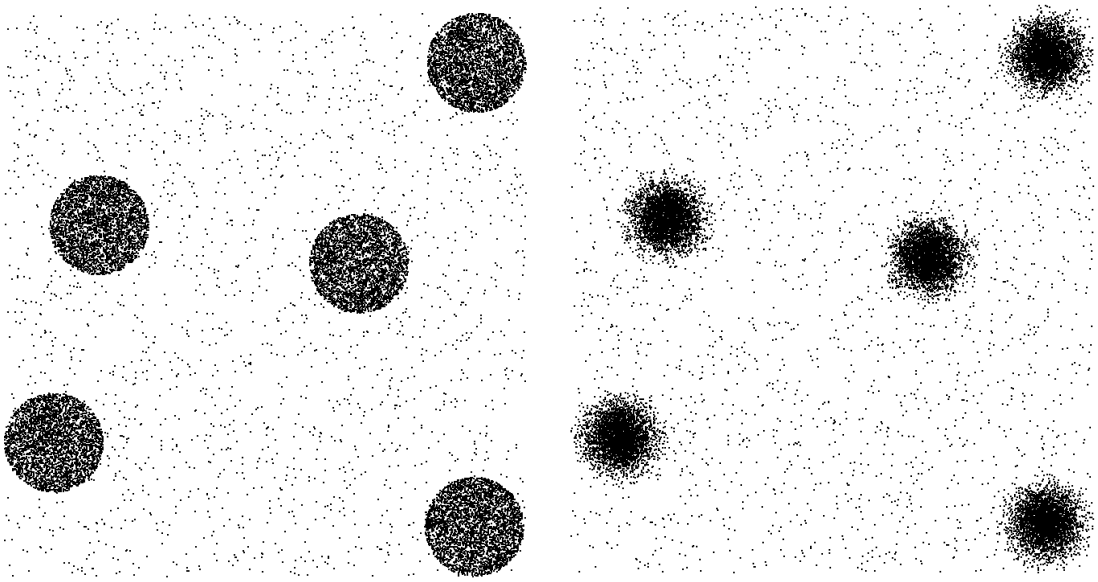


Figure 4.2: Distribution of number of points inside cell for uniform discs data

3. Gaussian discs
   Generated similar to uniform discs as described previously but instead of uniformly generated random points inside discs, we use Gaussian distribution with mean = center of the disc and standard deviation = 1/6. A visual comparison between uniform and Gaussian discs data is shown in Figure 4.3. Using $n = 1000000$ and $\varepsilon = 6$ we get a distribution shown in Figure 4.4. The number of empty cells is very large (82.14%, not



(a) uniform (b) Gaussian

Figure 4.3: Example of discs data set

shown in the figure) because of the larger area that we have. The maximum number of points inside a cell can vary from 37 to 41 (not shown in the figure) but the number of cells containing this many points is very small. The maximum number of points is very large compared to the uniform discs data because we have a very dense part in the center of the Gaussian disc. As before, we only show the results until number of points equals 25 because for larger numbers, the number of cells is so small that we cannot see the bar. Typically, the total number of cells with more than 25 points is less than 0.07%.

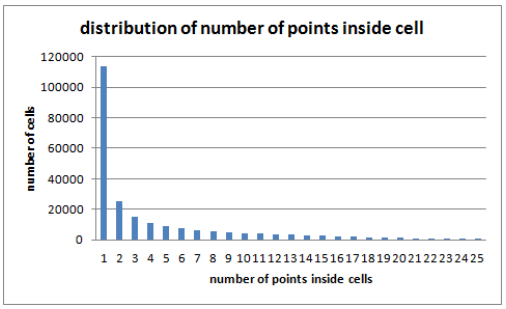


Figure 4.4: Distribution of number of points inside cell for Gaussian discs data

4. Uniform Strips
Let n be the total number of points and $m = 0.9 \cdot n$. We generate m points uniformly at random inside five strips and $n - m$ points as noise inside a bounding box of size approximately $10\sqrt{m} \times 10\sqrt{m}$. The size of the strips is $5\sqrt{m} \times \frac{1}{5}\sqrt{m}$. Note that the strips may intersect with each other. Using $n = 1000000$ and $\varepsilon = 5$ we get a distribution shown in Figure 4.5. The number of empty cells is very large (93.96%, not shown in

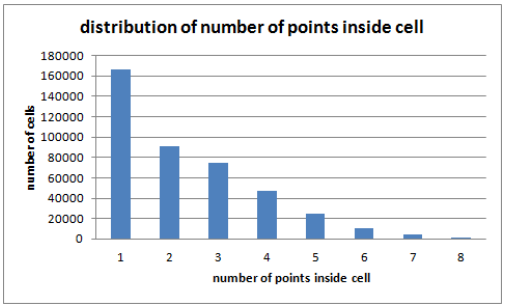
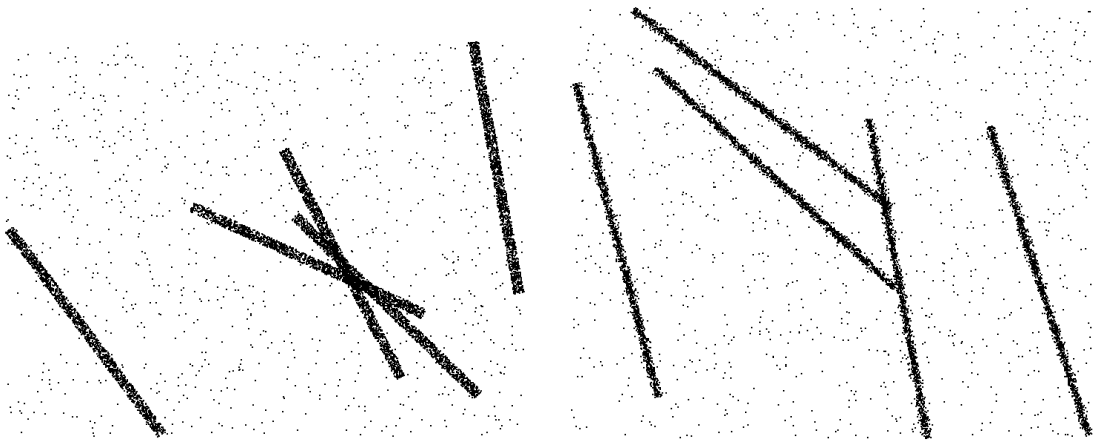


Figure 4.5: Distribution of number of points inside cell for uniform strips data

the figure) because of the larger area that we have (100 times the uniform fill data). The maximum number of points inside a cell can vary from 15 to 16 (not shown in the figure) but the number of cells containing this many points is very small. As before, we only show the results until number of points equals 8 because for larger numbers, the number of cells is so small that we cannot see the bar. Typically, the total number of cells with more than 8 points is less than 0.02%.

5. Gaussian Strips
   Generated similar to uniform strips but the points will be generated by a Gaussian distribution of standard deviation equal to 1/6 along the width of the strips. Note that, along the length of the strips, they are still uniformly distributed. A visual comparison between uniform and Gaussian discs data is shown in Figure 4.6. Using $n = 1000000$



(a) uniform (b) Gaussian

Figure 4.6: Example of strips data set

and $\varepsilon = 6$ we get a distribution shown in Figure 4.7. The number of empty cells is very large (94.13%, not shown in the figure) because of the larger area that we have. The maximum number of points inside a cell can vary from 26 to 32 (not shown in the figure) but the number of cells containing this many points is very small. As before, we only show the results until number of points equals 14 because for larger numbers, the number of cells is so small that we cannot see the bar. Typically, the total number of

cells with more than 14 points is less than 0.03%.

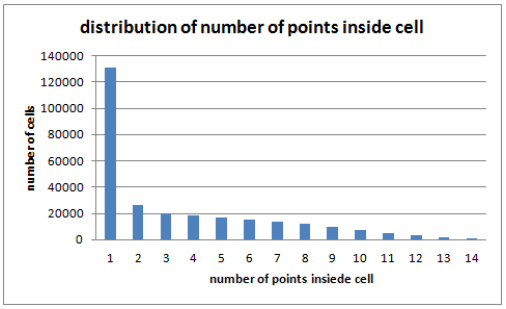
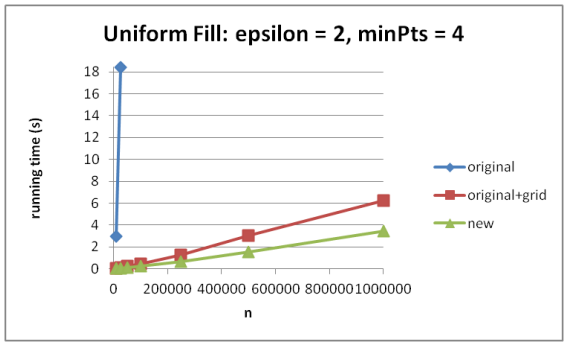


Figure 4.7: Distribution of number of points inside cell for Gaussian strips data

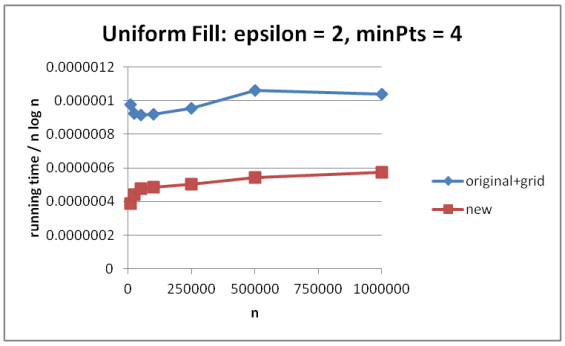
4.2 Results and discussions

4.2.1 Dependency of the running time on n

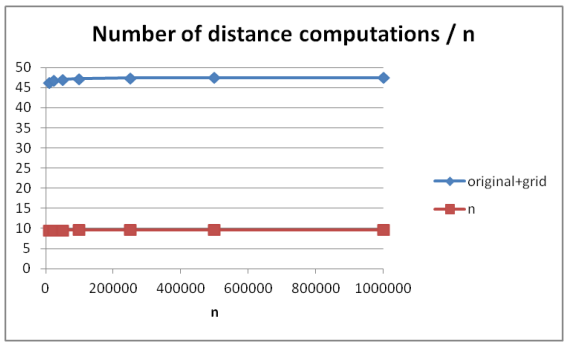
For this experiment, we run the three algorithms on data sets of size from $n = 10,000$ until $n = 1,000,000$ while keeping the value of ε and $minPts$ fixed. For uniform fill data, we have chosen the value for ε and $minPts$ such that all of the points will be detected as a single cluster. From Figure 4.8a, we can see that the new algorithm outperforms both the



(a) running time



(b) running time / $n \log n$

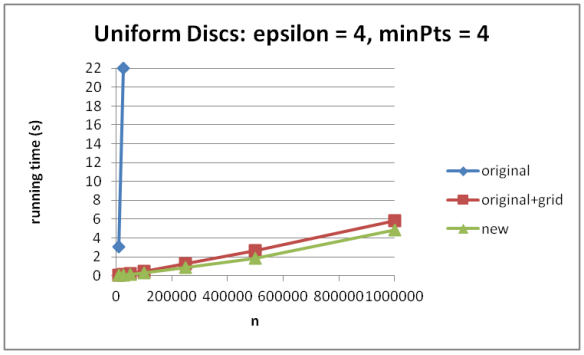


(c) number of distance computations / n

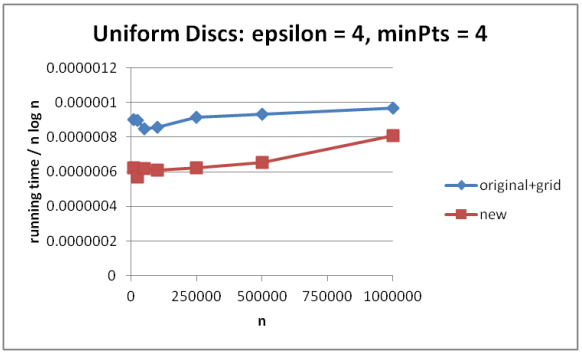
Figure 4.8: Uniform fill, variable n

original algorithm with grid and without grid. Since the original DBSCAN algorithm is very slow, we decided not to do experiments on larger data sets (but only up to 25,000 points). From Figure 4.8b, we can see that the new algorithm is approximately 2 times faster than the original algorithm with grid. The running time for both algorithms seems to be slightly higher than what should be an $n \log n$ curve. However, when we compute the number of distance computations instead of running time (Figure 4.8c), we found that it is linear to the number of points except for the beginning of the curve. This happens for all the data sets that we tested. From figure 4.8c, we can see that the number of distance computations for the new algorithm is approximately 20% of the original with grid's. This shows that the new algorithm saves a lot of distance computations. Remember that it will skip some distance computations if it finds a grid cell with more than $minPts$ points inside. Also, when checking points in the neighboring cells, it stops after finding at least $minPts$ points whose distance is at most ε. The different behavior between running time and the number of distance computations may be caused by caching behavior of the operating system but we do not try to confirm this.

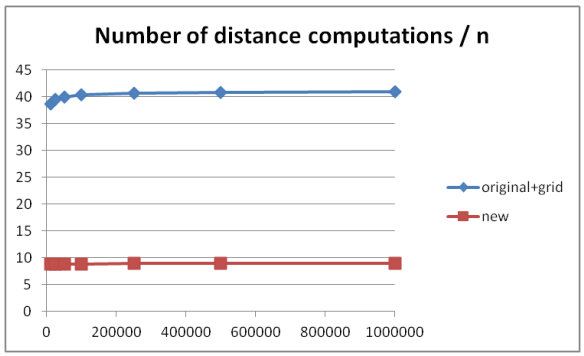
For the discs data (both uniform and Gaussian), we have chosen the value for ε and $minPts$ such that the algorithm provides the "correct" clustering result, i.e. a disc is detected as a cluster and noise is detected as noise. After experimenting with uniform discs data, we have a similar result as the one for the uniform fill data. Although the ratio between the running time of the new algorithm and original with grid is now lower (1.4 times faster instead of 2).



(a) running time



(b) running time / $n \log n$

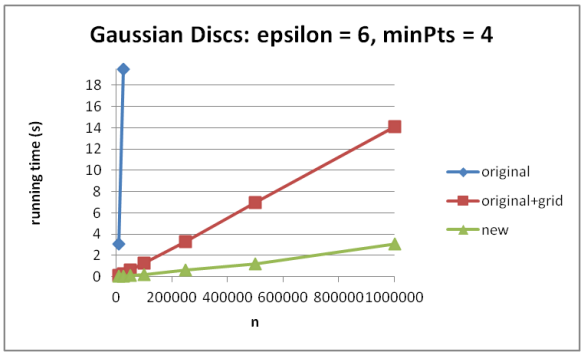


(c) number of distance computations / n

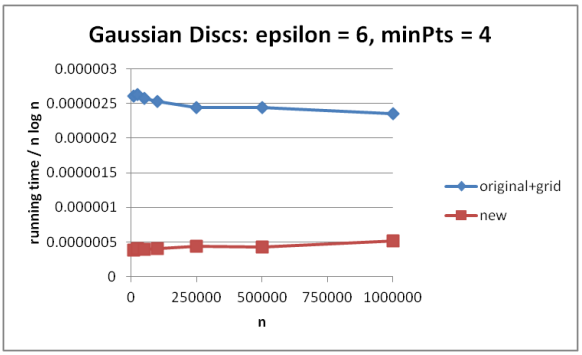
Figure 4.9: Uniform discs, variable n

For the Gaussian discs data, the new algorithm outperforms the original with grid by a

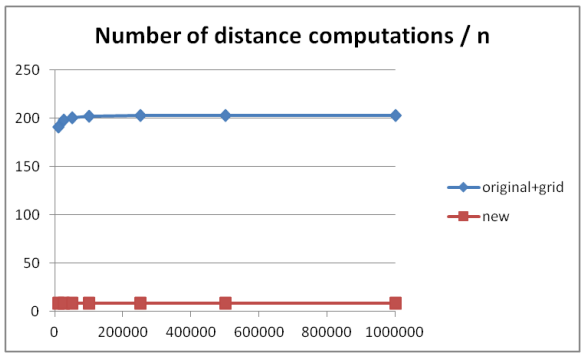
factor of 6 as we can see in Figure 4.10c. This happens because, in Gaussian distribution,



(a) running time



(b) running time / $n \log n$



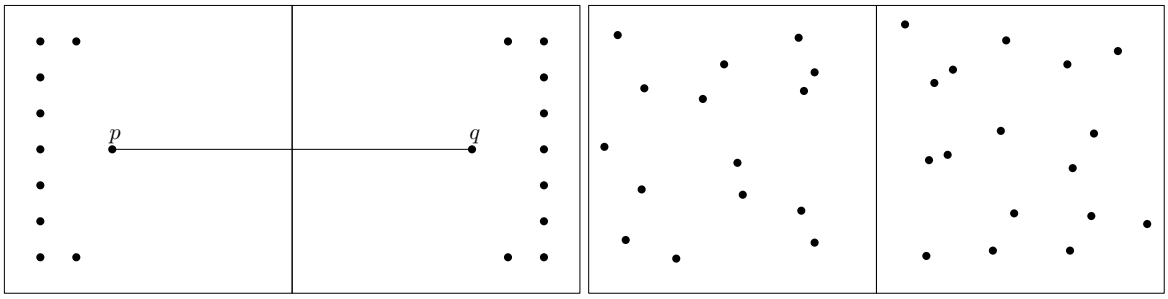
(c) number of distance computations / n

Figure 4.10: Gaussian discs, variable n

we have a very dense region of points in the center of the discs which makes many cells have more than $minPts$ points inside it. From figure 4.10c, we can see that the number of distance computations for the new algorithm is approximately 4% of the original with grid's. We can also see that, for large n, the number of distance computations is linear to the size of n. This shows that even though we do not use the Voronoi diagram for merging clusters and the number of points inside a cell can be very large (caused by the dense part in the center of the disc), the number of distance computations will not be $O(n^2)$. This happens because the algorithm stops immediately after finding a pair of points whose distance is at most ε. To better describe this, consider two extreme cases as shown in Figure 4.11. In Figure 4.11a, there will be many distance computations done before actually finding points p and q. This case has a very small chance of happening on our data set. On the other hand, there will be many pair of cells similar to the ones shown in Figure 4.11b. In this case, many pairs of points have distance at most ε. Thus, the algorithm has a very high chance to find one of these pairs before doing too many distance computations.

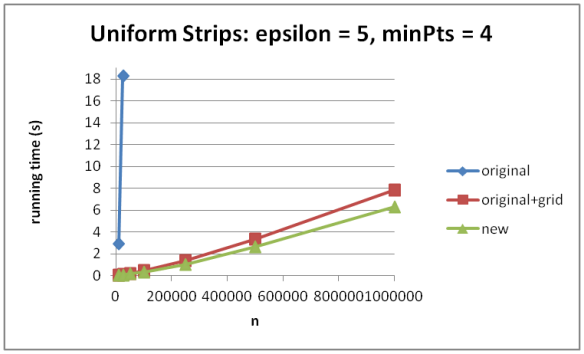
For the strips data (both uniform and Gaussian), we have chosen the value for ε and $minPts$ such that the algorithm provides the "correct" clustering result, i.e. a strip is detected as a cluster and noise is detected as noise. From Figure 4.12b, we can see that the new algorithm is still faster than the original with grid although it is now only 1.3 times faster.

For the Gaussian strips data, The new algorithm is 2.5 times faster than the original with

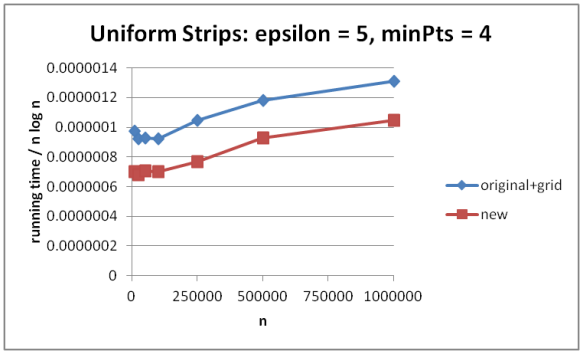


(a) bad case (b) good case

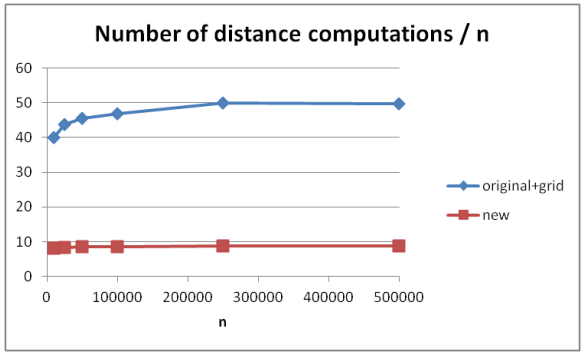
Figure 4.11: Two extreme cases of merging two cells



(a) running time



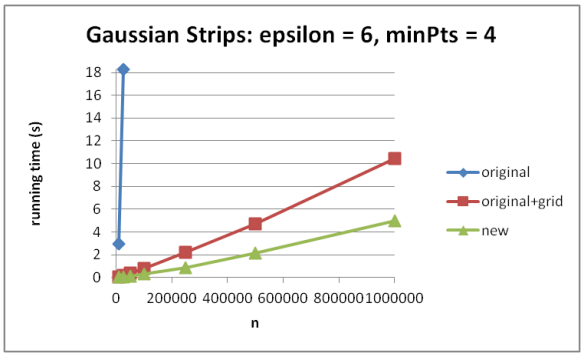
(b) running time / $n \log n$



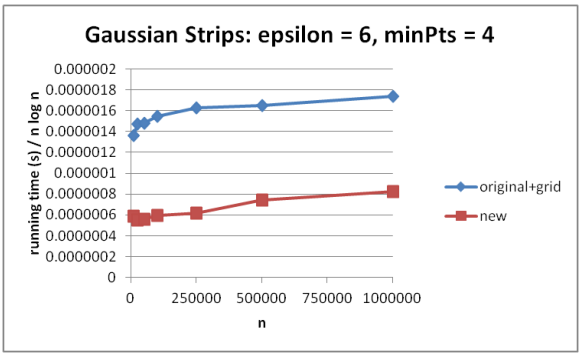
(c) number of distance computations / n

Figure 4.12: Uniform strips, variable n

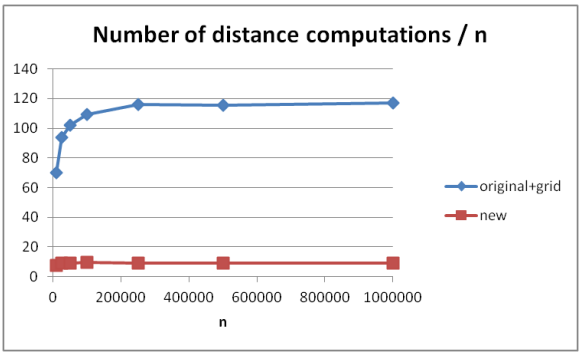
grid. An interesting fact is for $n < 250000$, the number of distance computations is not linear to the number of points as shown in Figure 4.13c. The cause of this behavior remains to be investigated.



(a) running time



(b) running time / $n \log n$

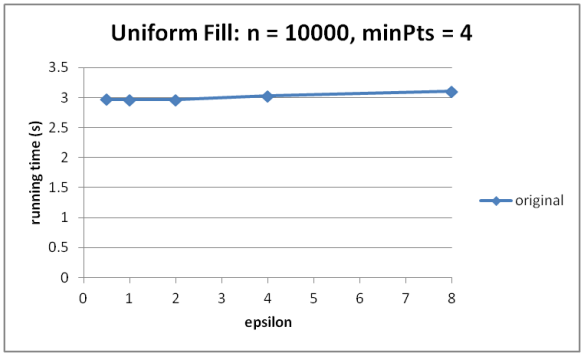


(c) number of distance computations / n

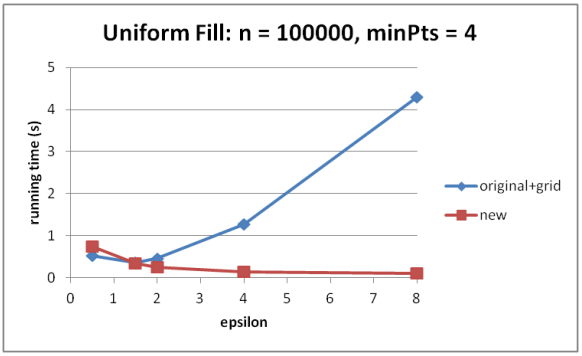
Figure 4.13: Gaussian strips, variable n

4.2.2 Dependency of the running time on ε

Now, we want to see how the algorithms perform if we change the value of ε. For uniform fill data set, the result of the experiment is shown in Figure 4.14. Figure 4.14a basically shows



(a) original



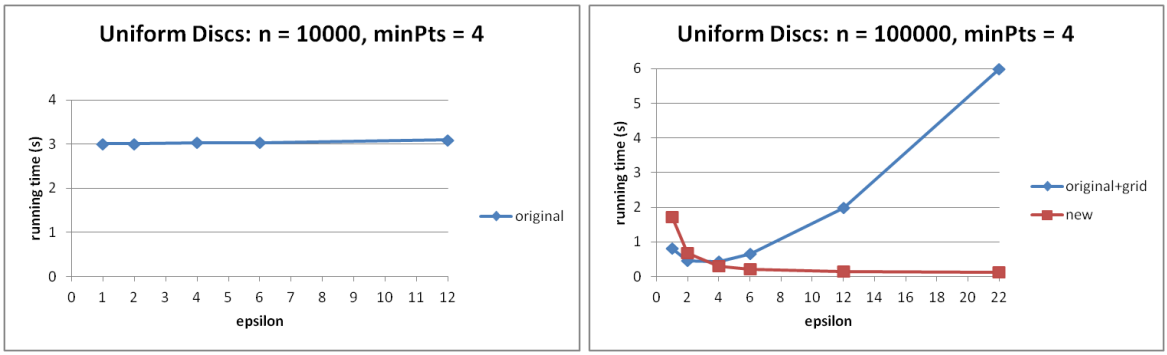
(b) improved

Figure 4.14: Uniform fill, variable ε

that the original DBSCAN algorithm is not significantly affected by the change of ε. This happens because for each point, the original algorithm's range query does a complete scan

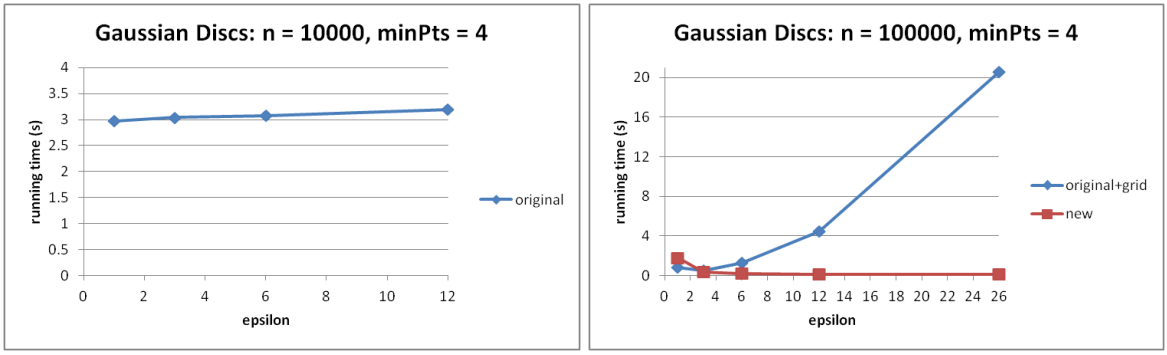
of the whole data set regardless of ε. Figure 4.14b is more interesting because it shows that the new algorithm and the original algorithm with grid are very sensitive to ε. Generally, the original with grid will benefit from having small value of ε because larger value of ε implies more points inside grid's cell and this caused the range query to be more expensive. In contrast, the new algorithm will benefit from having a large value of ε because more points inside grid cell implies higher chance that it will find a grid with more than $minPts$ points inside. The running time of both is similar if the value of ε is approximately 1.5. However, for $\varepsilon = 1.5$, multiple small clusters will be detected by the algorithm which is not something that we want. For $\varepsilon \geq 2$, we will have our desired result, i.e. all points are detected as one cluster. For this value of ε, the new algorithm outperforms the original with grid. One recommendation that we can provide from this experiment is when you use the new algorithm for clustering, try a large value of ε first, then decrease it until you find a suitable result. This will save a lot of time compared to trying small value of ε first.

For the other data sets (discs and strips), we have similar result as the uniform fill experiment's result. The results are shown in Figure 4.15, 4.16, 4.17, and 4.18.



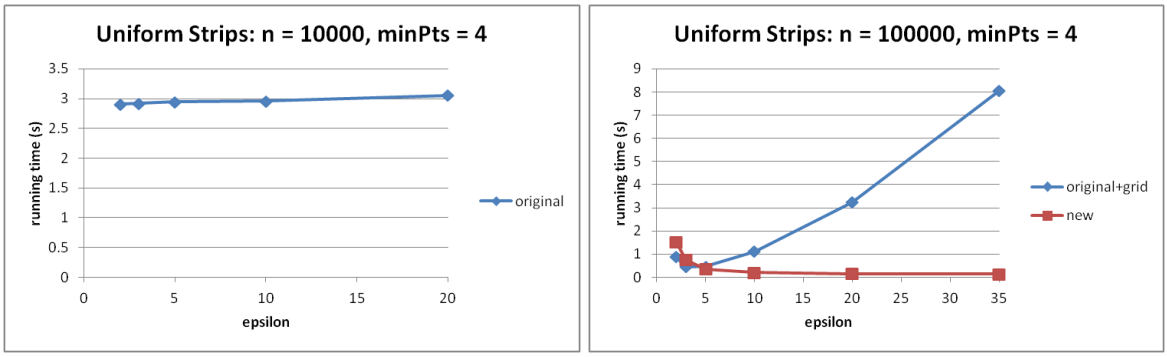
(a) original (b) improved

Figure 4.15: Uniform discs, variable ε



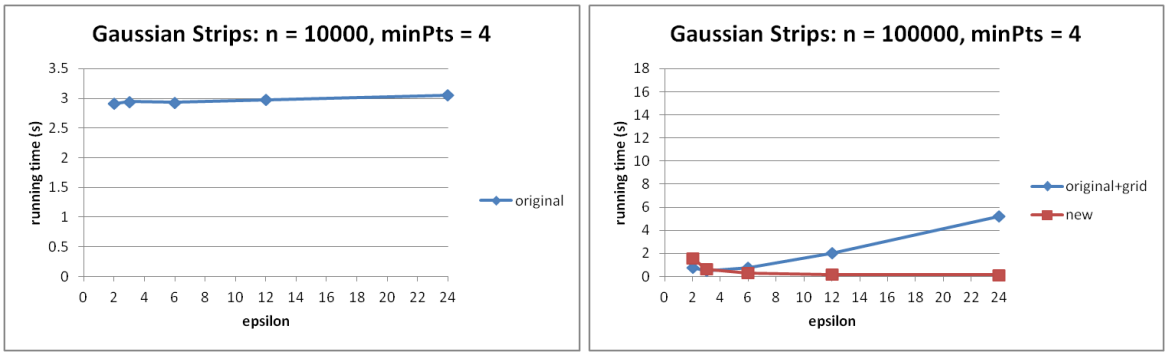
(a) original (b) improved

Figure 4.16: Gaussian discs, variable ε



(a) original (b) improved

Figure 4.17: Uniform strips, variable ε

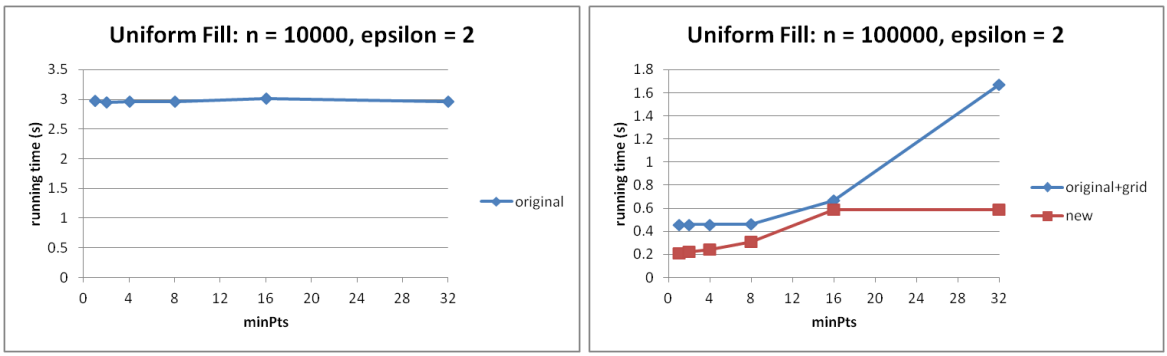


(a) original (b) improved

Figure 4.18: Gaussian strips, variable ε

4.2.3 Dependency of the running time on *minPts*

In the next experiment, we want to see how the algorithms perform if we change the value of *minPts*. For uniform fill data set, the result of the experiment is shown in Figure 4.19. Similar

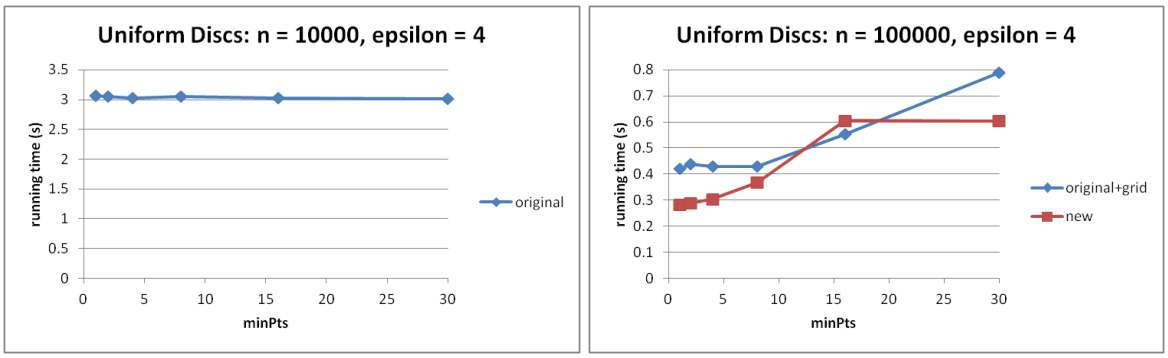


(a) original (b) improved

Figure 4.19: Uniform fill, variable *minPts*

to Figure 4.14a, Figure 4.19a shows that the original DBSCAN algorithm is not sensitive to the value of $minPts$. Figure 4.19b shows that the original with grid is quite stable as for $minPts \leq 8$. For $8 < minPts \leq 32$ it will be slower as the value of $minPts$ grows. We cannot find the exact reason for this behavior but we observe that for $minPts \leq 8$, the result of the algorithm will be one big cluster but for $8 < minPts \leq 32$, the number of the resulting clusters grows and the size of the clusters shrink. The new algorithm will also be slower as the value of $minPts$ grows but stable after $minPts = 16$. One of the advantages of the new algorithm is that we can skip some distance computations when the number of points inside a cell is more than $minPts$. As $minPts$ grows, the number of such cells will be less until it reaches 0. Also, remember when checking points in the neighboring cells, the new algorithm stops after finding at least $minPts$ points whose distance is at most ε. If $minPts$ is too large, the algorithm will not be too sensitive to the change of $minPts$.

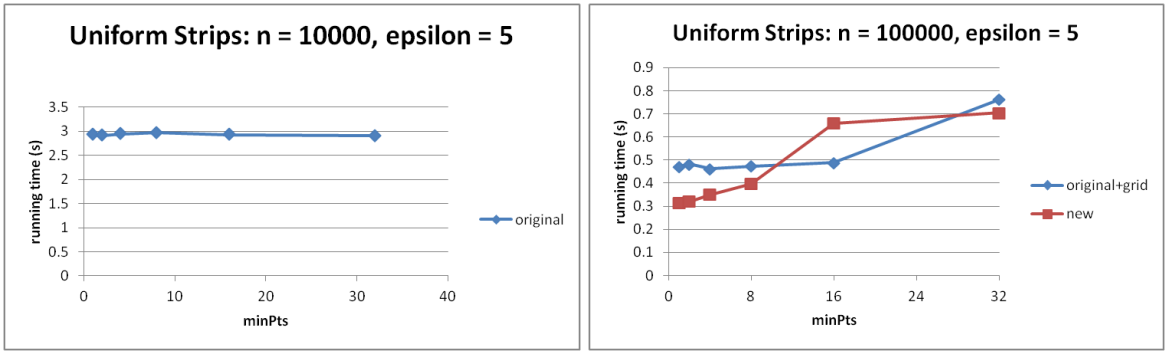
For uniform discs and strips data sets, the result is basically similar to the uniform fill experiment's result denoted by similar shape of the curve. The only difference is that there is a part where the new algorithm is slower than the original with grid, i.e. in Figure 4.20b from $minPts \approx 12$ to $minPts \approx 18$ This happens because the speedup that the new algorithm on the discs data is lower compared to the uniform fill data (2 times faster for uniform fill and 1.4 faster for uniform discs).



(a) original (b) improved

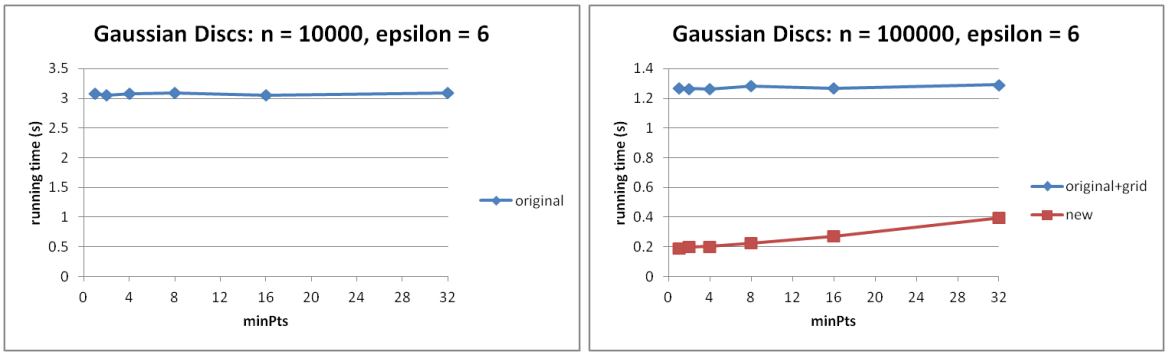
Figure 4.20: Uniform discs, variable $minPts$

For the Gaussian discs and strips data sets, the results are different than the result we obtained from the uniformly distributed data sets. We can see, from Figure 4.22b, that the original algorithm with grid is now not sensitive to $minPts$ since unlike the uniform fill / discs / strips data, the number of resulting clusters does not increase. This happens because the dense part of the discs will still be detected as one cluster even though the value $minPts$ is large. For the new algorithm, it will continue increasing because when checking points in the neighboring cells, it stops after finding at least $minPts$ points whose distance is at most ε. One recommendation that we can provide from this experiments is when you use the new algorithm for clustering, try a small value of $minPts$ first, then increase it until you find a suitable result.



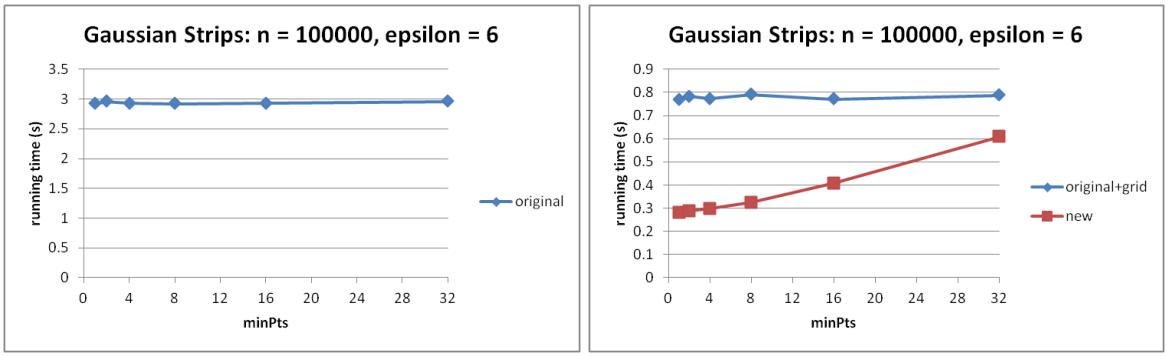
(a) original (b) improved

Figure 4.21: Uniform strips, variable *minPts*



(a) original (b) improved

Figure 4.22: Gaussian discs, variable *minPts*



(a) original (b) improved

Figure 4.23: Gaussian strips, variable *minPts*

Chapter 5

Conclusions

We introduce an algorithm for clustering a set of n points in the plane that runs in $O(minPts \cdot n + n \log n)$ time and produces a clustering consistent with DBSCAN, where $minPts$ is the parameter of DBSCAN specifying the minimum number of points in the neighbor of a core point. We performed experiments comparing the running time of new algorithm with the "original DBSCAN with grid". The results of these experiments show that the new algorithm outperforms the "original with grid". The experiments also show that the greater the value of ε, the faster the new algorithm is. Therefore, when clustering, it will be more efficient to try a large value of ε first, then decrease it until we find a suitable result. The experiments also show that the smaller the value of $minPts$, the faster the new algorithm is. Therefore, when clustering, it will be more efficient to try a small value of $minPts$ first, then increase it until we find a suitable result.

Future research can be done considering the fact that the new algorithm can still be improved. For example, we can merge some cells right after finding the core points which can save us some distance computations. Another future work that can be done is implementing the sorting based "grid" that we introduced and do some experiments on it. Adapting the algorithm to higher dimensional space should also be investigated.

Appendix A

Tables of experiments result

A.1 Dependency of the running time on n

n	ε	$minPts$	original	original+grid	new
10000	2	4	2.9626	0.039	0.0156
25000	2	4	18.4506	0.1015	0.0483
50000	2	4		0.2152	0.1124
100000	2	4		0.4587	0.2435
250000	2	4		1.2871	0.6804
500000	2	4		3.0236	1.5414
1000000	2	4		6.2277	3.4443

Table A.1: Uniform fill, variable n

n	ε	$minPts$	original	original+grid	new
10000	4	4	3.0267	0.036	0.0249
25000	4	4	22.0524	0.0984	0.0626
50000	4	4		0.1996	0.1452
100000	4	4		0.4292	0.3043
250000	4	4		1.237	0.8411
500000	4	4		2.6599	1.8645
1000000	4	4		5.8189	4.8553

Table A.2: Uniform strips, variable n

n	ε	$minPts$	original	original+grid	new
10000	6	4	3.0778	0.1044	0.0155
25000	6	4	19.5345	0.2887	0.0451
50000	6	4		0.6039	0.0934
100000	6	4		1.2635	0.2016
250000	6	4		3.2978	0.5865
500000	6	4		6.9512	1.234
1000000	6	4		14.1149	3.089

Table A.3: Gaussian discs, variable n

n	ε	$minPts$	original	original+grid	new
10000	5	4	2.9453	0.0389	0.028
25000	5	4	18.3116	0.1014	0.0749
50000	5	4		0.2182	0.1655
100000	5	4		0.4616	0.3509
250000	5	4		1.4103	1.0371
500000	5	4		3.3711	2.6498
1000000	5	4		7.8466	6.2836

Table A.4: Uniform strips, variable n

n	ε	$minPts$	original	original+grid	new
10000	6	4	2.9326	0.0546	0.0234
25000	6	4	18.3014	0.1623	0.0609
50000	6	4		0.3479	0.1306
100000	6	4		0.7723	0.2978
250000	6	4		2.1949	0.833
500000	6	4		4.6957	2.1125
1000000	6	4		10.4179	4.9592

Table A.5: Gaussian strips, variable n

A.2 Dependency of the running time on ε

n	ε	$minPts$	original+grid	new
100000	0.5	4	0.5178	0.7441
100000	1.5	4	0.3605	0.3448
100000	2	4	0.4587	0.2435
100000	4	4	1.2698	0.1403
100000	8	4	4.298	0.1061
100000	16	4	16.0636	0.0887

Table A.6: Uniform fill, variable ε

n	ε	$minPts$	original+grid	new
100000	1	4	0.8081	1.7226
100000	2	4	0.4463	0.6784
100000	4	4	0.4292	0.3043
100000	6	4	0.6535	0.2137
100000	12	4	1.9798	0.1466
100000	22	4	5.9872	0.114

Table A.7: Uniform discs, variable ε

n	ε	$minPts$	original+grid	new
100000	1	4	0.822	1.7816
100000	3	4	0.5116	0.3683
100000	6	4	1.2635	0.2016
100000	12	4	4.4771	0.1357
100000	26	4	20.5829	0.1015

Table A.8: Gaussian discs, variable ε

n	ε	$minPts$	original+grid	new
100000	2	4	0.8715	1.5289
100000	3	4	0.4455	0.7487
100000	5	4	0.4616	0.3509
100000	10	4	1.1231	0.1998
100000	20	4	3.2397	0.1451
100000	35	4	8.0477	0.1358

Table A.9: Uniform strips, variable ε

n	ε	$minPts$	original+grid	new
100000	2	4	0.7614	1.5387
100000	3	4	0.4901	0.6322
100000	6	4	0.7723	0.2978
100000	12	4	2.0077	0.1826
100000	24	4	5.2148	0.1342
100000	60	4	16.4514	0.1216

Table A.10: Gaussian strips, variable ε

A.3 Dependency of the running time on $minPts$

n	ε	$minPts$	original+grid	new
100000	2	1	0.4555	0.2122
100000	2	2	0.4585	0.2235
100000	2	4	0.4587	0.2435
100000	2	8	0.4618	0.3089
100000	2	16	0.6661	0.5853
100000	2	32	1.6721	0.5879

Table A.11: Uniform fill, variable $minPts$

n	ε	$minPts$	original+grid	new
100000	4	1	0.4211	0.2825
100000	4	2	0.4384	0.2873
100000	4	4	0.4292	0.3043
100000	4	8	0.4286	0.368
100000	4	16	0.5539	0.6052
100000	4	30	0.7894	0.6038

Table A.12: Uniform discs, variable $minPts$

n	ε	$minPts$	original+grid	new
100000	6	1	1.2668	0.1887
100000	6	2	1.2653	0.1981
100000	6	4	1.2635	0.2016
100000	6	8	1.2824	0.2247
100000	6	16	1.2682	0.2697
100000	6	32	1.2915	0.3933

Table A.13: Gaussian discs, variable $minPts$

n	ε	$minPts$	original+grid	new
100000	5	1	0.4707	0.3152
100000	5	2	0.4818	0.3198
100000	5	4	0.4616	0.3509
100000	5	8	0.4744	0.3979
100000	5	16	0.4882	0.6599
100000	5	32	0.763	0.7037

Table A.14: Uniform strips, variable $minPts$

n	ε	$minPts$	original+grid	new
100000	6	1	0.7692	0.2823
100000	6	2	0.7818	0.2886
100000	6	4	0.7723	0.2978
100000	6	8	0.791	0.3244
100000	6	16	0.7708	0.4088
100000	6	32	0.7875	0.6085

Table A.15: Gaussian strips, variable $minPts$

Appendix B

Visualizations of clustering results

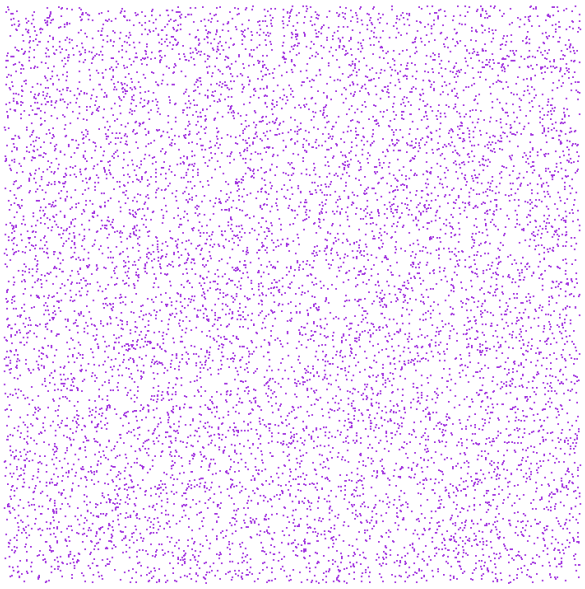


Figure B.1: Clustering result on uniform fill data

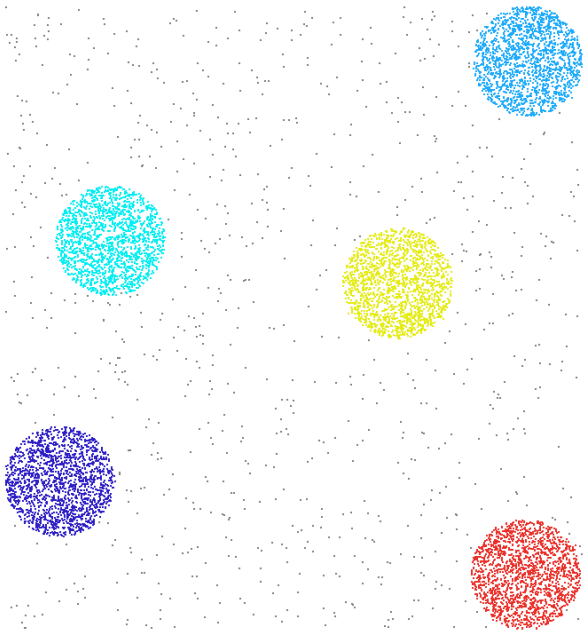


Figure B.2: Clustering result on uniform discs data

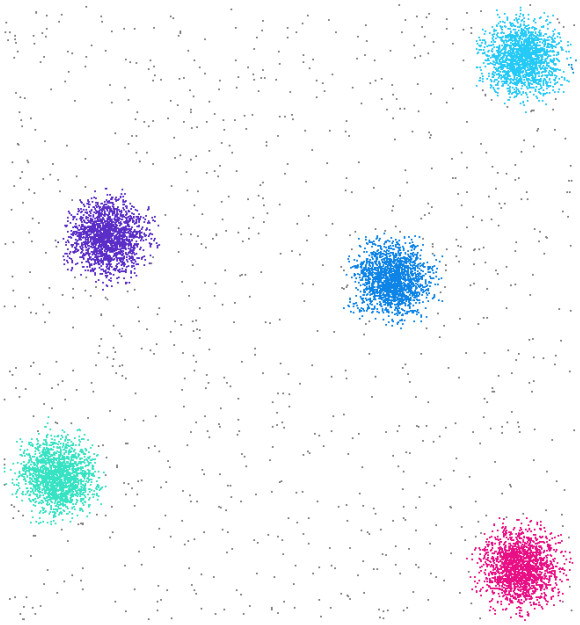


Figure B.3: Clustering result on Gaussian discs data

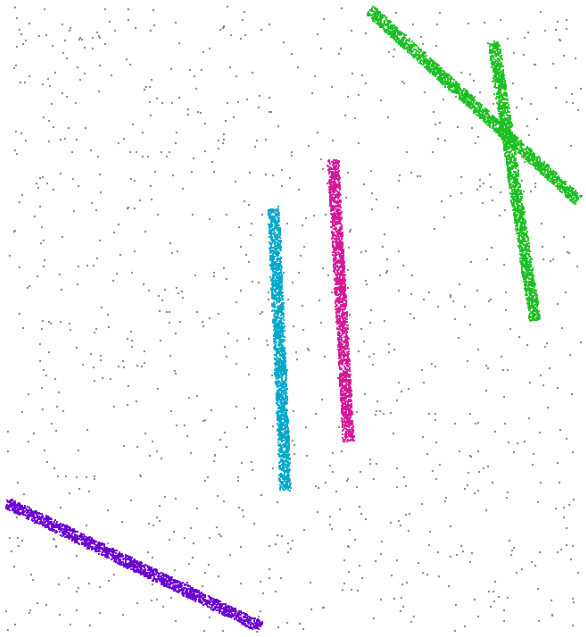


Figure B.4: Clustering result on uniform strips data

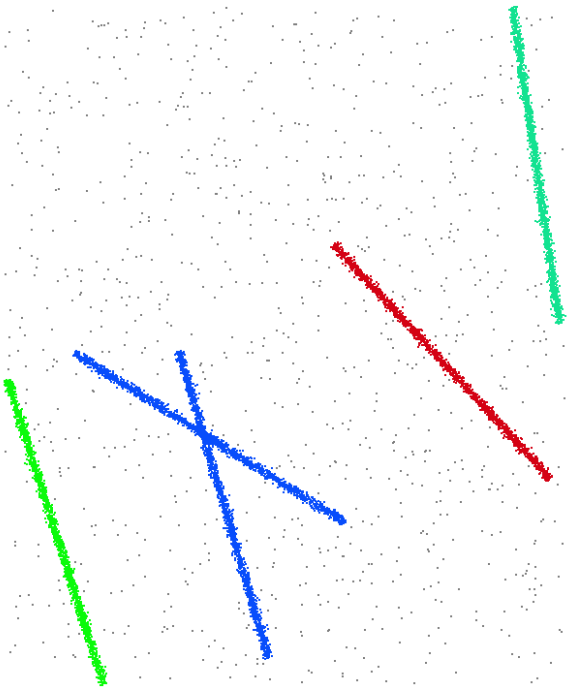


Figure B.5: Clustering result on Gaussian strips data

Bibliography

[1] M. Ankerst, M. M. Breunig, H.-P. Kriegel, and J. Sander. Optics: Ordering points to identify the clustering structure. In *SIGMOD Conference*, pages 49–60, 1999.

[2] B. Borah and D. Bhattacharyya. An improved sampling-based DBSCAN for large spatial databases. In *Intelligent Sensing and Information Processing, 2004. Proceedings of International Conference*, pages 92 – 96, 2004.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms (3. ed.)*. MIT Press, 2009.

[4] D. Defays. An efficient algorithm for a complete link method. *Comput. J.*, 20(4):364–366, 1977.

[5] Y. El-Sonbaty, M. A. Ismail, and M. Farouk. An efficient density based clustering algorithm for large databases. In *ICTAI*, pages 673–677, 2004.

[6] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *KDD*, pages 226–231, 1996.

[7] B. Liu. A fast density-based clustering algorithm for large databases. In *Machine Learning and Cybernetics, 2006. Proceedings of International Conference*, pages 996 –1000, 2006.

[8] S. Mahran and K. Mahar. Using grid for accelerating density-based clustering. In *CIT*, pages 35–40, 2008.

[9] R. T. Ng and J. Han. Efficient and effective clustering methods for spatial data mining. In *VLDB*, pages 144–155, 1994.

[10] R. Sibson. Slink: An optimally efficient algorithm for the single-link cluster method. *Comput. J.*, 16(1):30–34, 1973.

[11] C.-F. Tsai and C.-T. Wu. Gf-dbscan: a new efficient and effective data clustering technique for large databases. In *Proceedings of the 9th WSEAS international conference on Multimedia systems & signal processing*, pages 231–236, 2009.