
Data Mining and Knowledge Discovery, 7, 123–152, 2003

Using Self-Similarity to Cluster Large Data Sets

DANIEL BARBARÁ dbarbara@gmu.edu
ISE Department, MSN 4A4, George Mason University, Fairfax, Virginia, 22030, USA

PING CHEN Chenp@zeus.dt.uh.edu
Computer and Mathematical Science Department, University of Houston-Downtown, One Main Street, Houston, TX 77002, USA

Editors: Fayyad, Mannila, Ramakrishnan

Received September 8, 2000; Revised November 2, 2001

Abstract. Clustering is a widely used knowledge discovery technique. It helps uncovering structures in data that were not previously known. The clustering of large data sets has received a lot of attention in recent years, however, clustering is a still a challenging task since many published algorithms fail to do well in scaling with the size of the data set and the number of dimensions that describe the points, or in finding arbitrary shapes of clusters, or dealing effectively with the presence of noise. In this paper, we present a new clustering algorithm, based in self-similarity properties of the data sets. Self-similarity is the property of being invariant with respect to the scale used to look at the data set. While fractals are self-similar at every scale used to look at them, many data sets exhibit self-similarity over a range of scales. Self-similarity can be measured using the fractal dimension. The new algorithm which we call Fractal Clustering (FC) places points incrementally in the cluster for which the change in the fractal dimension after adding the point is the least. This is a very natural way of clustering points, since points in the same cluster have a great degree of self-similarity among them (and much less self-similarity with respect to points in other clusters). FC requires one scan of the data, is suspendable at will, providing the best answer possible at that point, and is incremental. We show via experiments that FC effectively deals with large data sets, high-dimensionality and noise and is capable of recognizing clusters of arbitrary shape.

Keywords: clustering, self-similarity, scalability

1. Introduction

Clustering is one of the most widely used techniques in data mining. It is used to reveal structure in data that can be extremely useful to the analyst. The problem of clustering is to partition a data set consisting of n points embedded in a d-dimensional space into k sets or clusters, in such a way that the data points within a cluster are more similar among them than to data points in other clusters. A precise definition of clusters does not exist. Rather, a set of functional definitions have been adopted. A cluster has been defined (Backer, 1995) as a set of entities which are alike (and different from entities in other clusters), an aggregation of points such that the distance between any point in the cluster is less than the distance to points in other clusters, and as a connected region with a relatively high density of points. Our method adopts the first definition (likeness of points) and uses a fractal property to define similarity between points.

The area of clustering has received an enormous attention as of late in the database community. The latest techniques try to address pitfalls in the traditional clustering algorithms (for a good coverage of traditional algorithms see Jain and Dubes (1988)). These pitfalls range from the fact that traditional algorithms favor clusters with spherical shapes (as in the case of the clustering techniques that use centroid-based approaches), are very sensitive to outliers (as in the case of all-points approach to clustering, where all the points within a cluster are used as representative of the cluster), or are not scalable to large data sets (as is the case with all traditional approaches).

New approaches need to satisfy the data mining desiderata (Bradley et al., 1998):

- Require at most one scan of the data.
- Have on-line behavior: provide the best answer possible at any given time and be suspendable at will.
- Be incremental by incorporating additional data efficiently.

In this paper we propose a clustering algorithm that follows this desiderata, while providing a very natural way of defining clusters that is not restricted to spherical shapes (or any other type of shape). This algorithm is based on self-similarity (namely, a property exhibited by self-similar data sets, i.e., the fractal dimension) and clusters points in such a way that data points in the same cluster are more *self-affine* among themselves than to points in other clusters.

This paper is organized as follows. Section 2 offers a brief introduction to the fractal concepts we need to explain the algorithm. Section 3 describes our technique. Section 4 summarizes experimental results that we have obtained using our technique. Finally, Section 5 offers conclusions and guidelines for future work.

2. Fractal dimension

Nature is filled with examples of phenomena that exhibit seemingly chaotic behavior, such as air turbulence, forest fires and the like. However, under this behavior it is almost always possible to find *self-similarity*, i.e. an invariance with respect to the scale used. The structures that exhibit self-similarity over every scale are known as *fractals* (Mandelbrot, 1983). On the other hand, many data sets, that are not fractal, exhibit self-similarity over a range of scales.

Fractals have been used in numerous disciplines (for a good coverage of the topic of fractals and their applications see Schroeder (1991)). In the database area, fractals have been successfully used to analyze R-trees (Faloutsos and Kamel, 1997), Quadtrees (Faloutsos and Gaede, 1996), model distributions of data (Faloutsos et al., 1996) and selectivity estimation (Belussi and Faloutsos, 1995).

Self-similarity can be measured using the *fractal dimension*. Loosely speaking, the fractal dimension measures the number of dimensions "filled" by the object represented by the data set. In truth, there exists an infinite family of fractal dimensions. By embedding the data set in an n-dimensional grid which cells have sides of size r, we can count the frequency with which data points fall into the i-th cell, p_i, and compute D_q, the generalized fractal

dimension (Grassberger, 1983; Grassberger and Procaccia, 1983), as shown in Eq. (1).

$$D_q = \begin{cases} \dfrac{\partial \log \sum_i p_i \log p_i}{\partial \log r} & \text{for } q = 1 \\ \dfrac{1}{q-1} \dfrac{\partial \log \sum_i p_i^q}{\partial \log r} & \text{otherwise} \end{cases} \tag{1}$$

Among the dimensions described by Eq. (1), the *Hausdorff fractal dimension* ($q = 0$), the *Information Dimension* ($\lim_{q \to 1} D_q$), and the *Correlation dimension* ($q = 2$) are widely used. The Information and Correlation dimensions are particularly useful for data mining, since the numerator of D_1 is Shannon's entropy, and D_2 measures the probability that two points chosen at random will be within a certain distance of each other. Changes in the Information dimension mean changes in the entropy and therefore point to changes in trends. Equally, changes in the Correlation dimension mean changes in the distribution of points in the data set.

The traditional way to compute fractal dimensions is by means of the box-counting plot. For a set of N points, each of D dimensions, one divides the space in grid cells of size r (hypercubes of dimension D). If $N(r)$ is the number of cells occupied by points in the data set, the plot of $N(r)$ versus r in log-log scales is called the *box-counting plot*. The negative value of the slope of that plot corresponds to the Hausdorff fractal dimension D_0. Similar procedures are followed to compute other dimensions, as described in Liebovitch and Toth (1989).

To clarify the concept of box-counting, let us consider the famous example of George Cantor's dust, constructed in the following manner. Starting with the closed unit interval [0,1] (a straight-line segment of length 1), we erase the open middle third interval $(\frac{1}{3}, \frac{2}{3})$ and repeat the process on the remaining two segments, recursively. Figure 1 illustrates the procedure. The "dust" has a length measure of zero and yet contains an uncountable number of points. The Hausdorff dimension can be computed the following way: it is easy to see that for the set obtained after n iterations, we are left with $N = 2^n$ pieces, each of length $r = (\frac{1}{3})^n$. So, using a unidimensional box size with $r = (\frac{1}{3})^n$, we find 2^n of the boxes populated with points. If, instead, we use a box size twice as big, i.e., $r = 2(\frac{1}{3})^n$, we get 2^{n-1} populated boxes and so on. The log-log plot of box population vs. r renders a line with slope $D_0 = -\log 2 / \log 3 = -0.63\ldots$. The value 0.63 is precisely the fractal dimension of the Cantor's dust data set.

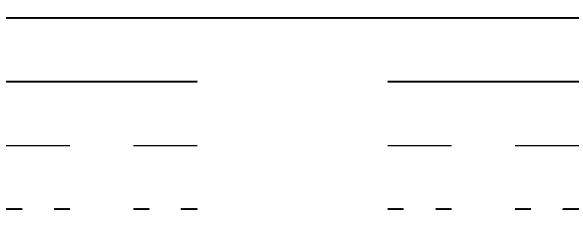

Figure 1. The construction of the Cantor dust. The final set has fractal (Hausdorff) dimension 0.63.

Figure 2. A "hybrid" Cantor dust set. The final set has fractal (Hausdorff) dimension larger than that of the the rightmost set (which is the Cantor dust set of figure 1).

In what follows of this section we present a motivating example that illustrates how the fractal dimension can be a powerful way for driving a clustering algorithm. Figure 2 shows the effect of superimposing two different Cantor dust sets. After erasing the open middle interval which results of dividing the original line in three intervals, the leftmost interval gets divided in 9 intervals, and only the alternative ones survive (5 in total). The rightmost interval gets divided in three, as before, erasing the open middle interval. The result is that if one considers grid cells of size $\frac{1}{3\times 9^n}$ at the n-th iteration, the number of occupied cells turns out to be $5^n + 6^n$. The slope of the log-log plot for this set is $D_0' = \lim_{n\to\infty}(\log(5^n + 6^n))/\log(3 \times 9^n)$. It is easy to show that $D_0' > D_0^r$, where $D_0^r = \log 2/\log 3$ is the fractal dimension of the rightmost part of the data set (the Cantor dust of figure 1). Therefore, one could say that the inclusion of the leftmost part of the data set produces a change in the fractal dimension and this subset is therefore "anomalous" with respect to the rightmost subset (or vice-versa). From the clustering point of view, for a human being it is easy to recognize the two Cantor sets as two different clusters. And, in fact, an algorithm that exploits the fractal dimension (as the one presented in this paper) will indeed separate these two sets as different clusters. Any point in the right Cantor set would change the fractal dimension of the left Cantor set if included in the left cluster (and viceversa). This fact is exploited by our algorithm (as we shall explain later) to place the points accordingly.

To further motivate the algorithm, let us consider two of the clusters in figure 7: the right-top ring and the left-bottom (square-like) ring. Figure 3 shows two log-log plots of number of occupied boxes against grid size. The first is obtained by using the points of the left-bottom ring (except one point). The slope of the plot (in its linear region) is equal to 1.57981, which is the fractal dimension of this object. The second plot, obtained by adding to the data set of points on the left-bottom ring the point (93.285928, 71.373638)—which naturally corresponds to this cluster—almost coincides with the first plot, with a slope (in its linear part) of 1.57919. On the other hand, the other plots are obtained by the data set of points in the right-top ring, and by adding to that data set the point (93.285928, 71.373638). The first plot exhibits a slope in its linear portion of 1.08081 (the fractal dimension of the data set of points in the right-top ring); the second plot has a slope of 1.18069 (the fractal dimension after adding the above-mentioned point). While the change in the fractal dimension brought about the point (93.285928, 71.373638) in the bottom-left cluster is 0.00062, the change in the right-top ring data set is 0.09988, more than 3 orders of magnitude bigger than the first change. Our algorithm would proceed to place point (93.285928, 71.373638) in the left-bottom ring, based on these changes.

Figure 3 also illustrates another important point. The "ring" used for the box counting algorithm is not a pure mathematical fractal set, as the Cantor Dust (figure 1), or the

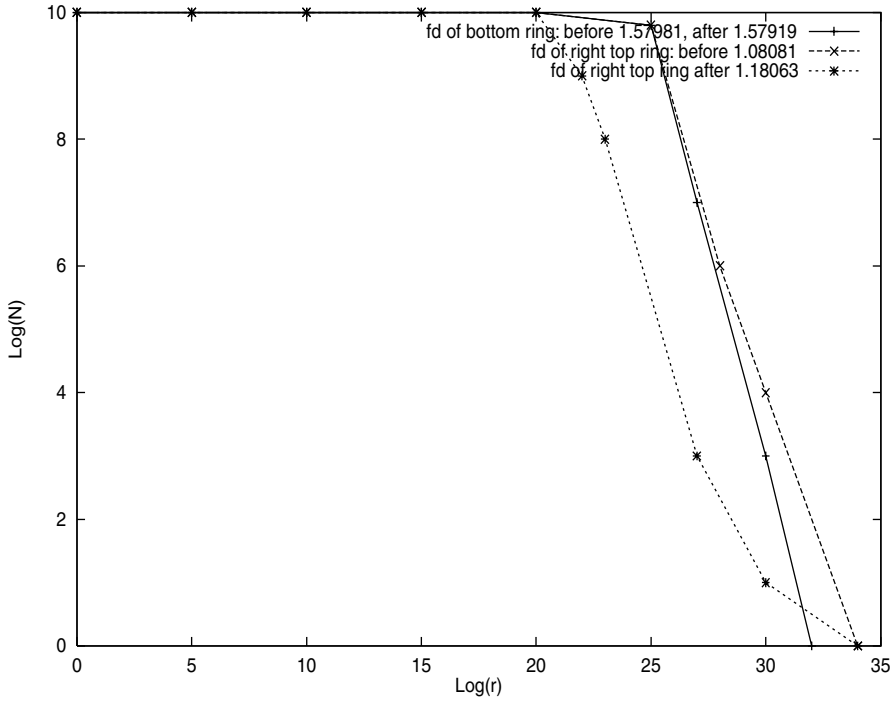

Figure 3. The box-counting plots of the bottom-left ring data set of figure 7, before and after the point (93.285928, 71.373638) has been added to the data set. The difference in the slopes of the linear region of the plots is the "fractal impact" (0.00062). (The two plots are so similar that lie almost on top of each other.) The box-counting plots of the top-right ring data set of figure 7, before and after the point (93.285928, 71.373638) has been added to the data set. The difference in the slopes of the linear region of the plots is the "fractal impact" (0.09988), much bigger than the corresponding impact shown in in the ring data set.

Sierpinski Triangle (Mandelbrot, 1983) are. Yet, this data set exhibits a fractal dimension (or more precisely a linear behavior in the log-log box counting plot) through a (relatively) large range of grid sizes. This fact serves to illustrate the point that our algorithm does not depend on the clusters being "pure" fractals, but rather to have a measurable dimension (i.e., their box count plot has to exhibit linearity over a range of grid sizes). Since we base our definition of cluster in the self-similarity of points within the cluster, this is an easy constraint to meet.

3. Clustering using the fractal dimension

Incremental clustering using the fractal dimension, abbreviated as Fractal Clustering, or FC, is a form of grid-based clustering (where the space is divided in cells by a grid; other techniques that use grid-based clustering are STING (Wang et al., 1997), WaveCluster (Sheikholeslami et al., 1998) and Hierarchical Grid Clustering (Schikuta, 1996)). The

main idea behind FC is to group points in a cluster in such a way that none of the points in the cluster changes the cluster's fractal dimension radically. FC also combines connectness, closeness and data points position information to pursue high clustering quality.

Our algorithm takes a first step of initializing a set of clusters, and then, incrementally adds points to that set. In what follows, we describe the initialization and incremental steps.

3.1. FC initialization step

In clustering algorithms the quality of initial clusters is extremely important, and has direct effect on the final clustering quality. Obviously, before we can apply the main concept of our technique, i.e., adding points incrementally to existing clusters, based on how they affect the clusters' fractal dimension, some initial clusters are needed. In other words, we need to "bootstrap" our algorithm via an initialization procedure that finds a set of clusters, each with sufficient points so its fractal dimension can be computed. We present in this section two choices for the initialization algorithm. The first choice uses fractal concepts to cluster a sample of points taken from the data set. The second uses a more traditional distance-based procedure. Obviously, there are many choices for the initialization step, we present only the results of these two here, but we are considering and experimenting with other options. In any case, if the wrong decisions are made at this step, we will be able to correct them later by reshaping the clusters dynamically.

3.1.1. Initialization algorithm 1. In this first algorithm, the process of initialization is made easy by the fact that we are able to convert a problem of clustering a set of multidimensional data points (which is a subset of the original data set) into a much simpler problem of clustering 1-dimensional points. The problem is further simplified by the fact that the set of data points that we use for the initialization step fits in memory. Figure 4 shows the pseudo-code of the initialization step. Notice that lines 3 and 4 of the code map the points of the initial set into unidimensional values, by computing the effect that each point has in the fractal dimension of the rest of the set (we could have computed the difference between the fractal dimension of S and that of S minus a point, but the result would have been the same). Line 5 of the code deserves further explanation: in order to cluster the set of Fd_i values, we can use any known algorithm. For instance, we could feed the fractal dimension values Fd_i, and a value k to a K-means implementation (Fukunaga, 1990; Selim and Ismail,

1.Given an initial set S of points $\{p_1, \cdots, p_M\}$ that fit in main memory
(obtained by sampling the data set).
2. For each $i = 1, \cdots, M$
    3. Define group $G_i = S - \{p_i\}$
    4. Calculate the fractal dimension of the set G_i, Fd_i.
5. Cluster the set of Fd_i values,
(The resulting clusters are the initial clusters.)

Figure 4. First initialization algorithm.

1.Given an initial set S of points $\{p_1, \cdots, p_M\}$ that fit in main memory
(obtained by sampling the data set), and a distance threshold κ.
(Initially $\kappa = \kappa_0$.)
2. Mark all points as unclustered, and make $k = 0$
3. Choose a point P (at random) out of the set of unclustered points
    4. Mark P as belonging to cluster C_k
    5. Starting at P and in a recursive, depth-first fashion,
        call $P' = NEAR(P, \kappa)$
    6. If P' is not NULL
        7. Mark P' as belonging to cluster C_k
    8. else backtrack to the previous point in the search.
    9. Update $\hat{d}$, the average distance between pairs of points in C_k
    10. Make $\kappa = \kappa_0 \times \hat{d}$
11. If there are still unclustered points, make $k = k + 1$ and go to 3.

$NEAR(P, \kappa)$
Find the nearest neighbor of P such that $dist(P', P) \leq \kappa$.
If no such P' can be found return NULL
Otherwise return P'.

Figure 5. Second initialization algorithm.

1984). Alternatively, we can let a hierarchical clustering algorithm (e.g., CURE (Guha et al., 1998)) cluster the sequence of Fd_i values.

Although, in principle, any of the dimensions in the family described by Eq. (1) can be used in line 4 of the initialization step, we have found that the best results are achieved by using D_2, i.e., the correlation dimension.

3.1.2. Initialization algorithm 2. In this second choice, we use a more classic distance-based approach. We try to cluster the initial sample of points by taking points at random and finding recursively points that are close to them. When no more close points can be found, a new cluster is initialized, choosing another point at random out of the set of points that have not been clustered yet. Figure 5 presents the pseudo code for this algorithm.

Given a sample of points (which fits in main memory) and a distance threshold κ (Line 1), the algorithm proceeds to build clusters by picking a random yet unclustered point and recursively finding the nearest neighbor in such a way that the distance between the point and the neighbor is less than κ. The neighbor is then included in the cluster, and the search for nearest neighbors continues in depth-first fashion, until no more points can be added to clusters. Notice that we do not try to restrict the clusters by constraining it, as it is customarily done, to have a diameter less than or equal to a certain threshold, since we want to be able to find arbitrarily shaped clusters.

The threshold, κ is the product of a predefined, static parameter κ_0, and $\hat{d}$, the average distance between pairs of points already in the cluster. Thus, κ is a dynamic threshold that gets updated as points are included in the cluster. Intuitively, we are trying to restrict the membership to an existing cluster to points whose minimum distance to a member of the cluster is similar to the average distance between points already in the cluster.

1.Given a batch S of points brought to main memory:
2.For each point $p \in S$:
  3.For $i = 1, \cdots, k$:
    4. Let $C'_i = C_i \bigcup \{p\}$
    5. Compute $F_d(C'_i)$
    6.Find $\hat{i} = min_i(|F_d(C'_i) - F_d(C_i)|)$
      7.If $|F_d(C'_{\hat{i}}) - F_d(C_{\hat{i}})| > \tau$
        8.Discard p as noise
      9.else
        10. Place p in cluster $C_{\hat{i}}$

Figure 6. The incremental step for FC.

3.2. Incremental step

After we get the initial clusters, we can proceed to cluster the rest of the data set. Each cluster found by the initialization step is represented by a set of boxes (cells in a grid). Each box in the set records its population of points. Let k be the number of clusters found in the initialization step, and $C = \{C_1, C_2, \ldots, C_k\}$ where C_i is the set of boxes that represent cluster i. Let $F_d(C_i)$ be the fractal dimension of cluster i.

The incremental step brings a new set of points to main memory and proceeds to take each point and add it to each cluster, computing its new fractal dimension. The pseudo-code of this step is shown in figure 6. Line 5 computes the fractal dimension for each modified cluster (adding the point to it). Line 6 finds the proper cluster to place the point (the one for which the change in fractal dimension is minimal). We call the value $|F_d(C'_i - F_d(C_i)|$ the *Fractal Impact* of the point being clustered over cluster i. The quantity $\min_i |F_d(C'_i - F_d(C_i)|$ is the *Minimum Fractal Impact* of the point. Line 7 is used to discriminate "noise." If the Minimum Fractal Impact of the point is bigger than a threshold τ, then the point is simply rejected as noise (Line 8). Otherwise, it is included in that cluster. We choose to use the Hausdorff dimension, D_0, for the fractal dimension computation of Line 5 in the incremental step. We chose D_0 since it can be computed faster than the other dimensions and it proves robust enough for the task.

To compute the fractal dimension of the clusters every time a new point is added to them, we keep the cluster information using a series of grid representations, or layers. In each layer, boxes (i.e., grids) have a size that is smaller than in the previous layer. The sizes of the boxes are computed in the following way. For the first layer (largest boxes), we divide the cardinality of each dimension in the data set by 2, for the next layer, we divide the cardinality of each dimension by 4 and so on. Accordingly, we get $2^D, 2^{2D}, \ldots, 2^{LD}$ D-dimensional boxes in each layer, where D is the dimensionality of the data set, and L the maximum layer we will store. Then, the information kept is not the actual location of points in the boxes, but rather, the number of points in each box. It is important to remark that the number of boxes in layer L can grow considerably, specially for high-dimensionality data sets. However, we need only to save boxes for which there is any population of points, i.e., empty boxes are not needed. The number of populated boxes at that level is, in practical data sets, considerably smaller (that is precisely why clusters are formed, in the first place).

Let us denote by B the number of populated boxes in level L. Notice that, B is likely to remain very stable throughout passes over the incremental step.

Every time a point is assigned to a cluster, we register that fact in a table, adding a row that maps the cluster membership to the point identifier (rows of this table are periodically saved to disk, each cluster into a file, freeing the space for new rows). The array of layers is used to drive the computation of the fractal dimension of the cluster, using a box-counting algorithm. In particular, we chose to use FD3 (Sarraille and DiFalco, http://tori.postech.ac.kr/softwares/), an implementation of a box counting algorithm based on the ideas described in Liebovitch and Toth (1989).

3.3. Reshaping clusters in mid-flight

It is possible that the number and form of the clusters may change after having processed a set of data points using the step of figure 6. This may occur because the data used in the initialization step does not accurately reflect the true distribution of the overall data set or because we are clustering an incoming stream of data, whose distribution changes over time. There are two basic operations that can be performed: splitting a cluster and merging two or more clusters into one.

A good indication that a cluster may need to be split is given by how much the fractal dimension of the cluster has changed since its inception during the initialization step. (This information is easy to keep and does not occupy much space.) A large change may indicate that the points inside the cluster do not belong together. (Notice that these points were included in that cluster because it was the *best choice* at the time, i.e., it was the cluster for which the points caused the least amount of change on the fractal dimension; but this does not mean this cluster is an ideal choice for the points.)

Once the decision of splitting a cluster has been made, the actual procedure is simple. Using the box (finest resolution layer, i.e., the lst layer of boxes) population we can run the initialization step. That will define how many clusters (if more than one) are needed to represent the set of points. Notice that up to that point, there is no need to re-process the actual points that compose the splitting cluster (i.e., no need to bring them to memory). This is true since the initialization step can be run over the box descriptions directly (the box populations represent an approximation of the real set of points, but this approximation is good enough for the purpose). On the other hand, after the new set of clusters has been decided upon, we need to relabel the points and a pass over that portion of the data set is needed (we assume that the points belonging to the splitting cluster can be retrieved from disk without looking at the entire data set: this can be easily accomplish by keeping each cluster in a separate file).

Merging clusters is even simpler. As an indication of the need to merge two clusters, we keep the minimum distance between clusters, defined by the distance between two points P_1 and P_2, such that P_1 belongs to the first cluster and P_2 to the second, and P_1 and P_2 are the closest pair of such points. When this minimum distance is smaller than a threshold, it is time to consider merging the two clusters. The threshold used is the minimum of the $\kappa = \kappa_0 \times \hat{d}$ for each of the two clusters. (Recall that $\hat{d}$ is the average pairwise distance in the cluster.) The merging can be done by using box population at the highest level of resolution

(smallest box size), for all the clusters that are deemed as too close. To actually decide whether the clusters ought to be merged or not, we perform the initialization algorithm 2, using the center of the populated boxes (at the highest resolution layer) as "points." Notice that it is not necessary to bring previously examined points back to memory, since the relabeling can simply be done by equating the labels of the merged clusters at the end. In this sense, merging does not affect the "one-pass" property of fractal clustering (as splitting does, although only for the points belonging to the splitting cluster).

3.4. Complexity of the algorithm

We assume that the cost of computing the fractal dimension of a set of n points is $O(n \log(n))$, as it is the case for the software (FD3, Sarraille and DiFalco, http://tori.postech.ac.kr/softwares/) that we have chosen for our experiments.

For the first initialization algorithm, the complexity is $O(M^2 \log(M))$, where M is the size of the sample of points. This follows from the fact that for each point in the sample, we need to compute the fractal dimension of the rest of the sample set (minus the point), incurring a cost of $O(M \log(M))$ per point. The incremental step is executed $O(N)$ times, where N is the size of the data set. The complexity of the incremental step is $O(n \log(n))$ where n is the number of points involved in the computation of the fractal dimension. Now, since we do not use the point information, but rather the box population to drive the computation of the fractal dimension, we can claim that n is $O(B)$ (the number of populated boxes in the highest layer). Three facts are worth noticing. First, the box population tends to be small for all cases of data sets with real clusters in them. Otherwise, the points would be uniformly distributed all over the space and no clusters would exist. Secondly, even though the number of boxes increases with the dimensionality, it is likely that less boxes would be really populated in high-dimensional cases. Thirdly, the number of boxes can be controlled in two ways: by increasing the size of the basic grid (finest level), or by using the memory reduction techniques discussed in Section 3.6. As proven in the experimental section, doing this does not dramatically affect the quality of the clustering. All these factors make it possible to assume that $B \ll N$. It then follows that the incremental part of FC will take time linear with respect to the size of the data set.

For small data sets, the first initialization algorithm time becomes dominant in FC. However, for large data sets, i.e., when $M \ll N$, the cost of the incremental step dominates, making FC linear in the size of the data set.

The second initialization algorithm exhibits different complexity. A naive implementation of *NEAR* will do a pass over the sample data every time it tries to find the nearest neighbor of a given point, given a complexity $O(M)$, and therefore a worst-case total complexity $O(M^2)$. Clever data structures for the nearest neighbor problem have been studied (Samet, 1990) and can be used to improve the running time of the initialization algorithm. In practice, however, we have found that running a naive implementation of *NEAR* is still much more time-efficient than running the first initialization algorithm. Following the same reasoning we used for the first initialization algorithm, we can conclude that for large data sets, the cost of the incremental step dominates, making FC linear in the size of the data set.

3.5. *Confidence bounds*

One question we need to settle is how to determine if we are placing points as outliers correctly. A point is deemed an outlier in the test of Line 7, in figure 6, when the Minimum Fractal Impact of the point exceeds a threshold τ. To add confidence to the stability of the clusters that are defined by this step, we can use the Chernoff bound (Chernoff, 1952) and the concept of adaptive sampling (Domingo et al., 1998, 2000; Domingos and Hulten, 2000; Lipton and Naughton, 1995; Lipton et al., 1993), to find the minimum number of points that must be successfully clustered after the initialization algorithm in order to guarantee with a high probability that our clustering decisions are correct. We present these bounds in this section.

Consider the situation immediately after the initial clusters have been found, and we start clustering points using FC. Let us define a random variable X_i, whose value is 1 if the i-th point to be clustered by FC has a Minimum Fractal Impact which is less than τ, and 0 otherwise. Using Chernoff's inequality one can bound the expectation of the sum of the X_i's, $X = \sum_i^n X_i$, which is another random variable whose expected value is np, where $p = Pr[X_i = 1]$, and n is the number of points clustered. The bound is shown in Eq. (2), where ϵ is a small constant.

$$Pr\left[\frac{X}{n} > (1+\epsilon)p\right] \leq \exp\left(-\frac{pn\epsilon^2}{3}\right) \tag{2}$$

Notice that we really do not know p, but rather have an estimated value of it, namely $\hat{p}$, given by the number of times that X_i is 1 divided by n. (i.e., the number of times we can successfully cluster a point divided by the total number of times we try.) In order that the estimated value of p, $\hat{p}$ obeys Eq. (3), which bounds the estimate close to the real value with an arbitrarily large probability (controlled by δ), one needs to use a sample of n points, with n satisfying the inequality shown in Eq. (4).

$$Pr[\|\hat{p} - p\|] > 1 - \delta \tag{3}$$

$$n > \frac{3}{p\epsilon^2} \ln\left(\frac{2}{\delta}\right) \tag{4}$$

By using adaptive sampling, one can keep bringing points to cluster until obtaining at least a number of successful events (points whose minimum fractal impact is less than τ) equal to s. It can be proven that in adaptive sampling (Watanabe, 2000), one needs to have s bound by the inequality shown in Eq. (5), in order for Eq. (3) to hold. Moreover, with probability greater than $1 - \delta/2$, the sample size (number of points processed) n, would be bound by the inequality of Eq. (6). (Notice that the bound of Eq. (6) and that of Eq. (4) are very close; The difference is that the bound of Eq. (6) is achieved without *knowing* p in advance.)

$$s > \frac{3(1+\epsilon)}{\epsilon^2} \ln\left(\frac{2}{\delta}\right) \tag{5}$$

$$n \leq \frac{3(1+\epsilon)}{(1-\epsilon)\epsilon^2 p} \ln\left(\frac{2}{\delta}\right) \tag{6}$$

Therefore, after seeing s positive results, while processing n points where n is bounded by Eq. (6) one can be confident that the clusters will be stable and the probability of successfully

clustering a point is the expected value of the random variable X divided by n (the total number of points that we attempted to cluster).

3.6. Memory management

Our algorithm is very space-efficient, by the virtue of requiring memory just to hold the boxes population at any given time during its execution. This fact makes FC scale very well with the size of the set. Notice that if the initialization sample is a good representative of the rest of the data, the initial clusters are going to remain intact (just containing large populations in the boxes). In that case, the memory used during the entire clustering task remains stable.

However, there are cases in which we will have demands beyond the available memory. Mainly, there are two cases where this can happen. If the sample is not a good representative (or the data changes with time in an incoming stream) we will be forced to change the number and structure of the clusters (as explained in Section 3.3), possibly requiring more space. The other case arises when we deal with high dimensional sets, where the number of boxes needed to describe the space may exceed the available memory.

For these cases, we have devised a series of memory reduction techniques that aim to achieve reasonable trade-offs between the memory used and the performance of the algorithm, both in terms of its running time and the quality of the uncovered clusters.

3.6.1. Memory reduction technique 1. In this technique, we cache boxes in memory, while keeping others swapped out to the disk, replacing the ones in memory on demand. Our experience shows that the boxes of smallest size consume 75% of all memory. So, we share the cache only amongst the smallest boxes, keeping the other layers always in memory. Of course, we cluster the boxes in pages, and use the pages as a caching unit. This reduction technique affects the running time but not the clustering quality.

3.6.2. Memory reduction technique 2. A way of requiring less memory is to ignore boxes with very few points. While this method can, in principle, affect the quality of clusters, it may actually be a good way to eliminate noise from the data set.

4. Experimental results

In this section we will show the results of using FC to cluster a series of data sets. Each data set aims to test how well FC does in each of the issues we have discussed in the Section 3. For each one of the experiments we have used a value of $\tau = 0.03$ (the threshold used to decide if a point is noise or it really belongs to a cluster), and a $\kappa = 3$ (for the initialization algorithm 2), unless otherwise specified in the results. We performed the experiments in a Sun Ultra2 with 500 Mb. of RAM, running Solaris 2.5. In spite of the large memory of the machine, we do not bring the entire data set to memory at once, so we can test the scalability of FC, regardless of the size of main memory. We used a buffer of fixed size (10 Kbytes) to incrementally bring chunks of each data set we experimented with. That, plus the memory reserved to keep the box counts (whose size depends on the dimensionality of the data set) is the only memory that FC uses for processing. Also, unless we specify it otherwise, the merging and splitting mechanisms are always in place to be triggered as required. When

using the first initialization algorithm, we have used K-means to cluster the unidimensional vector of effects. In each of the experiments, the points are distributed equally among the clusters (i.e., each cluster has the same number of points). After we run FC, for each cluster found, we count the number of points that were placed in that cluster and that also belonged there. The accuracy of FC is then measured for each cluster as the percentage of points correctly placed there. (We know, for each data set, the membership of each point; in one of the data sets we spread the space with outliers: in that case, the outliers are considered as belonging to an extra "cluster.")

4.1. Scalability

In this subsection we show experimental results of running time and cluster quality using a range of data sets of increasing sizes and a high-dimensional data set.

First, we use data sets whose distribution follows the one shown in figure 7 for scalability experiments. We use a complex set of clusters in this experiment, in order to show how FC can deal with arbitrarily-shaped clusters. (Not only do we have a square-shaped cluster, but also one of the clusters resides inside of another one.) We vary the total number of points in the data set to measure the performance of our clustering algorithm. In every case, we pick a sample of 600 points to run the initialization step. The results are summarized in figure 8. The first number in the column "time" is the running time when using the first initialization algorithm, while the second number corresponds to the usage of the second initialization algorithm. As it can be seen, the running time (when using the first initialization algorithm) increases with the size of the data set in a almost linear fashion, going from 53 seconds

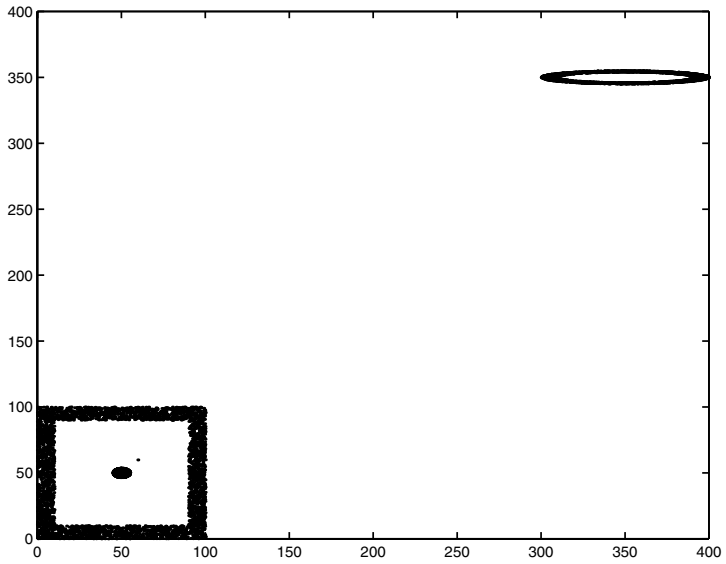

Figure 7. Three-cluster data set used for scalability experiments. Cluster 1 is the circle on the bottom left, cluster 2 is the oval in the upper right, and cluster 3 is the square on the bottom left, encircling cluster 1.

N	time	clusters found	assigned to	Coming from cluster1	cluster2	cluster3	accuracy
30,000	53s.	1	10,001	10,000	0	1	100 %
	12s.	2	10,000	0	10,000	0	100 %
		3	9,999	0	0	9,999	99.99%
300,000	91s	1	100,018	100,000	0	18	100%
	56s.	2	100,000	0	100,000	0	100%
		3	99,982	0	0	99,982	99.98 %
3,000,000	526s	1	1,000,004	1,000,000	0	4	100%
	485s.	2	1,000,000	0	1,000,000	0	100%
		3	999,996	0	0	999,996	99.96%
30,000,000	5,028s.	1	10,000,039	10,000,000	0	39	100%
	4,987s.	2	10,000,000	0	10,000,000	0	100%
		3	9,999,961	0	0	9,999,961	99.99%

Figure 8. Results of using FC in a data set (of several sizes) whose composition is shown in figure 7. The table shows the data set size (N), the running time for FC (time), and the composition for each cluster found in terms of points assigned to the cluster (points in cluster) and their provenance, i.e., whether they actually belong to cluster1, cluster2 or cluster3. Finally, the accuracy column shows the percentage of points that were correctly put in each cluster. The memory usage is constant for the entire range of data sets and equal to 64 Kbytes.

(48 seconds for initialization and 5 seconds for the incremental step) in the case of the 30,000 points data set to 5,028 seconds for the set with 30 million points. When using the second initialization algorithm, the running time goes from 12 sec. (7 seconds for initialization and 5 for the incremental step) for the 30,000 points data set to 4,987 sec. (4,980 sec. for the incremental step and 7 seconds for initialization) in the case of 30 million points. The incremental step grows linearly from 5 seconds in the 30,000 points case to 4,980 seconds in the 30 million point data set. The memory taken by our algorithm is constant for the entire range of data sets (64 Kbytes). Finally, the figure shows the composition of the clusters found by the algorithm, indicating the number of points and their procedence: whether they actually belong to cluster1, cluster2 or cluster3. (Since we know to which one of the rings of figure 7 each point really belongs). Some points of cluster 3 (less than 0.01%) are misplaced. The quality of the clusters found by using the two initialization algorithms is very similar, so we only report it once.

4.2. Quality of the clusters

In any incremental clustering algorithm, the sample and the order of processing have an influence over the quality of the clusters found. To investigate the influence of the sample (and therefore the processing order), we performed the experiments reported in the previous section with different samples. The results reported in figure 8 correspond to the worst performance among all the samples. For instance, in the case of 300,000 points, the worst accuracy (found in the recovery of Cluster 3), is 99.98%. For some samples, that figure reached almost 100%. So, the variability of the results is small with respect to the sample. This, along with the fact that the merging and splitting policies can correct initial mistakes, make FC a very stable algorithm.

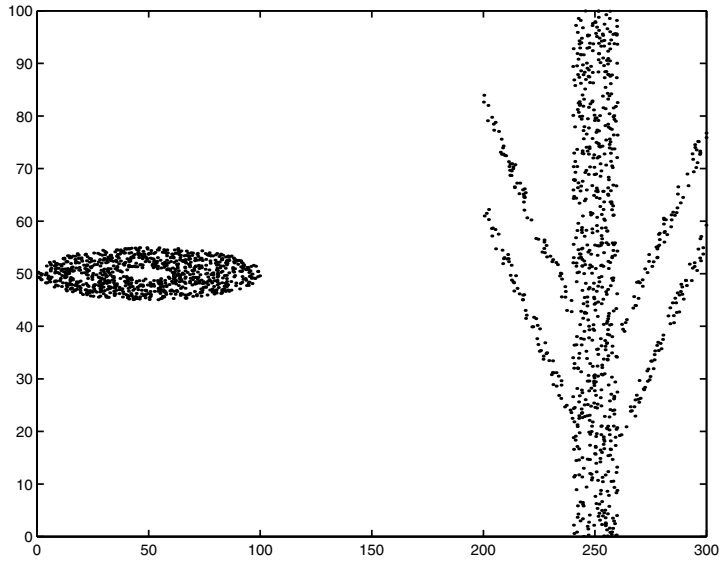

Figure 9. A complex data set used for measuring the quality of clusters obtained by the fractal dimension algorithm.

				clusters	points in
Alg.	N	time	mem.	found	cluster
1	300,000	175 s.	64 Kb.	1	150,000
				2	150,000
2	300,000	194 s.	64Kb.	1	150,000
				2	98203
				3	13,818
				4	11,961
				5	13,074
				6	12,944
merg.	300,000	211 s.	64 Kb.	1	150,000
				2	150,000

Figure 10. Results of using FC in the data set of figure 9. The first set of values (algorithm 1) shows the results of using initialization algorithm 1; the second set (algorithm 2) shows them for the initialization algorithm 2; the third set (merg.) shows the results when using algorithm 2 and merging.

To show how FC is capable of cluster arbitrary shapes, we used the algorithm in a data set of 300,000 points shown in figure 9. The data set is organized in two clusters: a tree-like structure and a ring cluster. (Each cluster contains 150,000 points.) With the first initialization algorithm, the initialization step took 136 seconds. (Which is the dominating factor, in this case, for the running time of 175 sec.) The second initialization algorithm ran for 32 sec., resulting in a total time of 196 seconds. Figure 10, shows how FC places every point in the correct cluster when using the first initialization algorithm. The second

initialization algorithm however, divides the tree-like cluster into 5 clusters (one for the "trunk" and 4 additional ones for each "branch"). This also explains the extra time taken by the incremental step in the second experiment (162 seconds) when compared to the first (39 seconds). Since each new point has to be considered for placement in six clusters, instead of two, the time spent computing the change in the fractal dimension is considerably greater. To further test the merging technique, we conducted the experiment (when using the second initialization algorithm) with and without the clusters generated by the second initialization algorithm. When using merging, the overall running time was 211 seconds (49 seconds more than the running time obtained without performing the merging), and as a result, five clusters got merged into a single one (the tree-like structure), as reported in figure 10.

To test FC ability to cluster spherical shapes, we tried a data set of three 3-dimensional spheres. The sphere centers are located at (50,50,50), (50,150,50), and (150,50,50) respectively. All the spheres have the same radius, 31.62. Points inside the spheres are generated by using a uniform distribution. Each sphere contains 100,000 points. FC is able to correctly cluster each one of the points, with an execution time of 26 seconds. For comparison, CURE used 358,394.5 seconds to cluster this data set, and placed 38,736 points in the first cluster, 81,321 in the second, and 89,011 in the third one, regarding 90,932 as outliers. BIRCH fares much better, finding three clusters with 99,983, 99,971 and 99,971 respectively. The remaining 75 points are declared as outliers. The clusters found by BIRCH are centered at (50.0143, 149.991, 50.0393), (149.993, 49.9897, 50.0309), and (50.0607, 49.9266, 50.0044) respectively (very close to the real centers). BIRCH took 46 seconds to cluster this data set.

4.3. Resistance to noise

To show the resistance of the algorithm to noise (outliers), we used the data set shown in figure 11, similar to that shown in figure 7, but with a set of points that act as noise. The data set used has 300,000 points with 5% "noisy" points, i.e., 285,000 points belong into the clusters, while 15,000 points are outliers. We conducted experiments for several values of τ, to show the sensitivity of the algorithm to this parameter. Figure 12 shows the results of these experiments. First, the results show that the placement of the points in the clusters is very stable along a wide range of values of τ. All the cluster points are placed correctly when τ ranges from 0.0001 to 0.05 (more than two orders of magnitude). The outliers are filtered in a proportion that ranges from 96.8% to 87.44% for τ ranging from 0.0001 to 0.01. For higher values of τ (0.05) a more marked deterioration of the filtering performance is obtained (only 69.86% of the outliers are filtered). All these argue for small values of τ, and shows that the performance is very good for a large range of the values. The behavior of the percentage of filtered noise vs. τ is shown in the graph of figure 13.

4.4. Sensitivity with respect to κ

The parameter κ is used in the second initialization algorithm to find the initial set of clusters. In this experiment, designed to test the sensitivity of the algorithm to κ, we used the data set illustrated in figure 7, with 300,000 points, equally distributed among the three clusters.

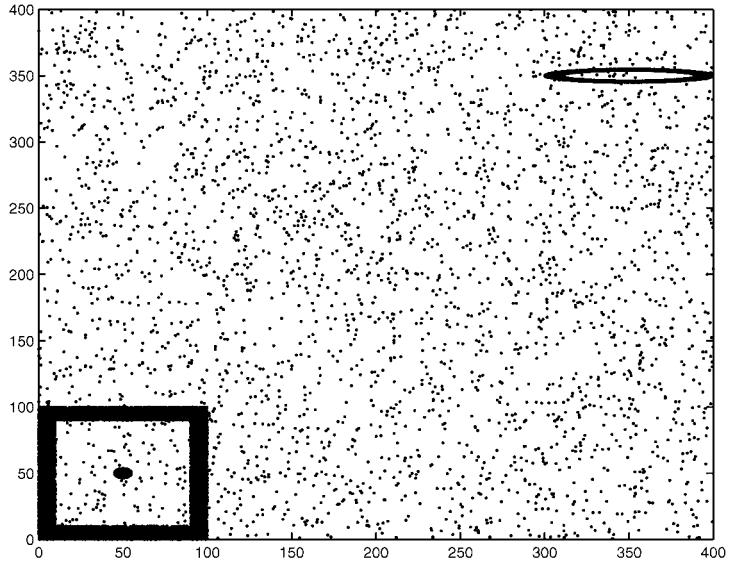

Figure 11. A noisy data set with three clusters and 5% of noise.

τ	time	memory	clusters found	points in cluster	Coming from cluster1	cluster2	cluster3	outliers
0.0001	96s.	64 Kb.	1	95,000	95,000	0	0	0
			2	95,113	0	95,000	0	113
			3	95,367	0	0	95,000	367
			outliers	14,520	0	0	0	14,520
0.003	96s.	64 Kb.	1	95,000	95,000	0	0	0
			2	95,113	0	95,000	0	113
			3	95,426	0	0	95,000	426
			outliers	14,461	0	0	0	14,461
0.007	96s.	64 Kb.	1	95,000	95,000	0	0	0
			2	95,218	0	95,000	0	218
			3	95,632	0	0	95,000	632
			outliers	13,150	0	0	0	13,150
0.01	96s.	64 Kb.	1	95,000	95,000	0	0	0
			2	95,218	0	95,000	0	218
			3	95,667	0	0	95,000	667
			outliers	13,115	0	0	0	13,115
0.05	96s.	64 Kb.	1	96,932	95,000	0	0	1,932
			2	95,946	0	95,000	0	946
			3	96,643	0	0	95,000	1,643
			outliers	10,479	0	0	0	10,479

Figure 12. Results of using FC on the noisy data set of figure 11. The number of points in the dataset is 300,000, with 15,000 of them being noise, and the rest equally distributed in the three clusters.

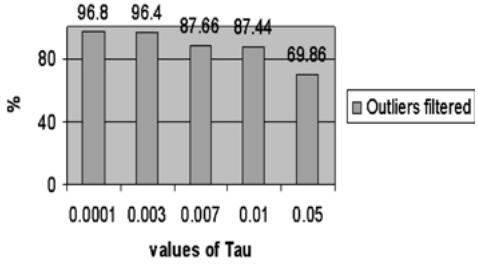

Figure 13. Percentage of filtered noise vs. τ for the experiments on the noisy data set of figure 11.

κ	time	clusters found	assigned to	Coming from cluster1	cluster2	cluster3
2.2	24 s.	1	100,000	100,000	0	0
		2	38,306	0	38,306	0
		3	50,139	0	50,139	0
		4	100,000	0	0	100,000
		5	11,555	0	11,555	0
2.5	34 s.	1	100,000	100,000	0	0
		2	40,491	0	40,491	0
		3	59,509	0	59,509	0
		4	100,000	0	0	100,000
3.0	24 s.	1	100,000	100,000	0	0
		2	100,000	0	100,000	0
		3	100,000	0	0	100,000
100.0	24 s.	1	100,000	100,000	0	0
		2	100,000	0	100,000	0
		3	100,000	0	0	100,000
200.0	24 s.	1	100,000	100,000	0	0
		2	100,000	0	100,000	0
		3	100,000	0	0	100,000
1,000.0	25 s	1	292,360	100,000	100,000	92,360
		2	7,640	0	0	7,640

Figure 14. Results of the experiment to test the sensitivity of FC with respect to κ. The table shows the number of clusters found in each case, and their composition. The results shown were obtained without using merge or split. Merge and split recovered the original clusters in all the cases.

The results are shown in figure 14. The results shown are obtained without using split or merge, to show the variability in the number of clusters as κ changes. However, with the merge and split mechanisms in place, the three original clusters are always recovered. For $\kappa = 2.2$, clusters 2, 3, and 5 are merged into cluster 2, increasing the processing time by 47 seconds. For $\kappa = 2.5$, clusters 2 and 3 are merged into cluster 2, adding 45 seconds of processing time. For $\kappa = 1{,}000$, merge and split restore the original three clusters adding 40 seconds of processing time.

Dimensions	N	time	mem.	clusters cluster	points in assigned to	Coming from Cluster 1	Coming from Cluster 2
2	300,000	71 s.	14.6 KB.	1	148,769	148,074	695
				2	151,231	1,925	149,305
4	300,000	85s	34.12 KB.	1	150,000	150,000	0
				2	150,000	0	150,000
6	300,000	131s.	2 MB.	1	150,391	149,999	392
				2	149,609	1	149,608
10	300,000	280s.	2 MB.	1	150,000	150,000	0
				2	150,000	0	150,000

Figure 15. Results of using FC in a data set with different number of dimensions.

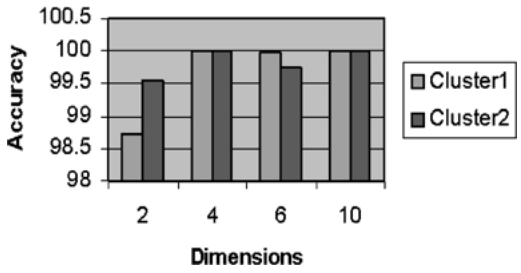

Figure 16. Accuracy of FC as a function of the number of dimensions for the data set of the figure 15.

4.5. Scalability with respect to number of dimensions

Figure 15 shows the results of an experiment in which we vary the number of dimensions for a two-cluster data set (the "shape" of the data set is similar to that of figure 25, but, of course, we cannot draw it for more than 2 dimensions). The results show that the algorithm scales well with dimensionality. Particularly noticeable is the economy of memory usage. Also, the quality of the found clusters remains good throughout the range of dimensions tested. Figure 16 shows the accuracy of FC (based on the results shown in figure 15) as the number of dimensions change.

4.6. Evaluation of memory reduction techniques

In this section we evaluate our memory reduction techniques. For this experiment, we use a 10-dimension data set with 200,000 points. There are two clusters, each one of them a ring with 100,000 points.

Figure 17 shows the results of clustering this data set, first without any memory reduction technique, and then with techniques 1 (caching boxes) and 2 (ignoring boxes with low population). We observed no reduction in the quality of the clusters obtained in each of the two memory reduction techniques. On the other hand, while the time taken by technique 2 is essentially the same as the one observed when we did not use memory reduction,

Technique	N	time	memory	clusters found	assigned to	Coming from cluster1	Coming from cluster2
no red.	200,000	64s.	2 Mb.	1	100,000	100,000	0
				2	100,000	0	100,000
Tech.1	200,000	589s.	4 Kb.	1	100,000	100,000	0
				2	100,000	0	100,000
Tech.2	200,000	65s.	3.5 Kb.	1	100,000	100,000	0
				2	100,000	0	100,000

Figure 17. Results of applying memory reduction techniques. The accuracy of FC is in each case 100%.

technique 1 substantially increases the running time (820% over the time taken with no memory reduction). This is expected, due to the extra I/O that the technique has to incur in. Both techniques achieve a considerable reduction of the memory usage: technique 1 reduces the memory from 2 Mb to 4 Kb (a decrease of 99.80%), while technique 2 reduces the memory from 2 Mb to 3.5 Kb. (a decrease of 99.82%). Technique 2 is likely to become more useful as the dimensionality increases, since there will be more and more boxes that are very sparsely populated in that case.

4.7. Splitting clusters

In order to further test the usefulness of our splitting technique, using the data set of figure 7 containing 30,000 points we performed the following experiment. For the initial sample, we selected points belonging only to two of the three clusters. Then, we monitored when the change of the fractal dimension for the clusters with respect to the original fractal dimension (of the initial clusters) exceeded 0.1. At that point we took a sample of each cluster and ran the initialization algorithm again. The results obtained were similar in quality to those reported in figure 8. The running time increased from 56 seconds to 120 seconds.

4.8. Real data set 1

We performed an experiment using our fractal clustering algorithm to cluster points in a real data set. The data set used was a picture of the world map in black and white, where the black pixels represent land and the white pixels water. In other words, the representation of each data point is three dimensional: its two coordinates and the attribute with values "white" or "black." The data set contains 3,319,530 pixels or points. With the second initialization algorithm the running time was 589 sec (558 seconds for the incremental step and 31 seconds for the initialization). We show in figures18–22 the clusters that were found. As it can be noticed, the quality of the clusters is extremely good: Cluster 0 (figure 18) spans the European, Asian and African continents, Cluster 1 (figure 19 corresponds to the North American continent (with a few outliers spread around Central America), Cluster 2 (figure 20) corresponds to the South American continent, Cluster 3 (figure 21) corresponds to Australia, and finally Cluster 4 (figure 22) shows Antarctica. The first initialization algorithm takes 1,008 sec. to run (387 sec. for initialization and 621 sec. for the incremental

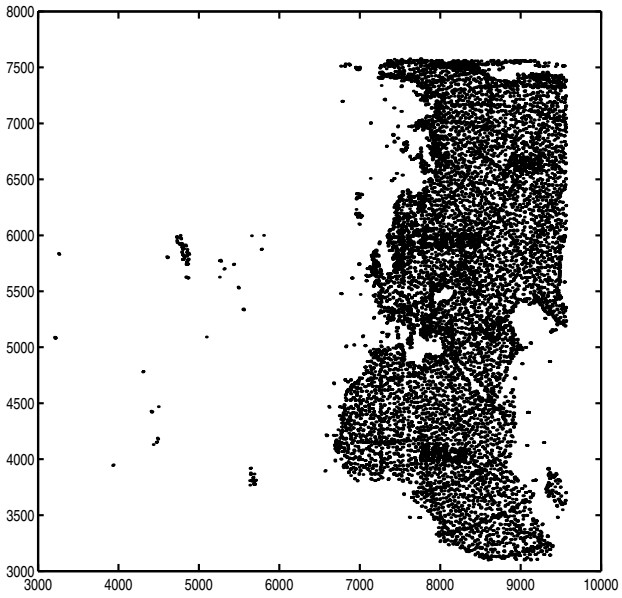

Figure 18. Cluster 0.

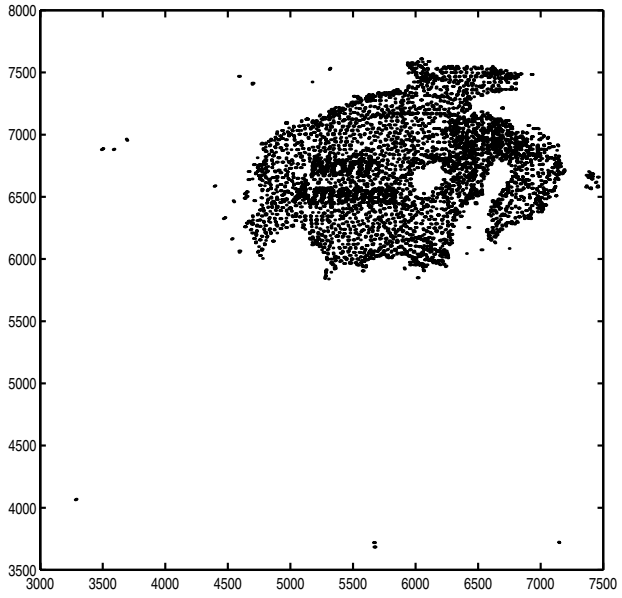

Figure 19. Cluster 1.

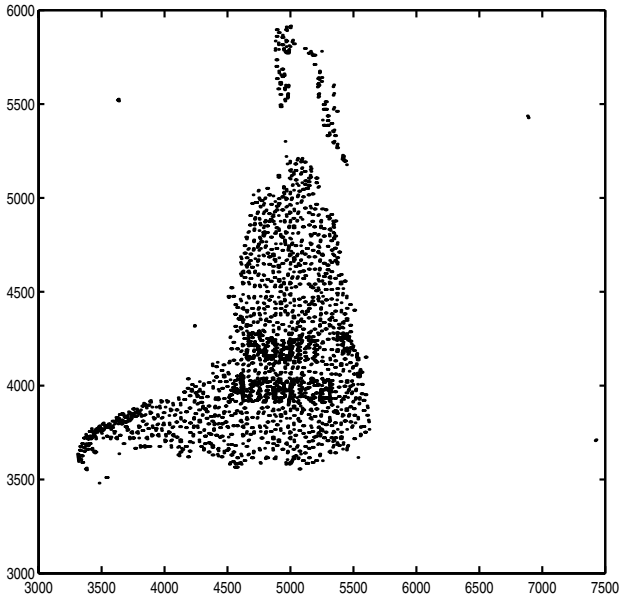

Figure 20. Cluster 2.

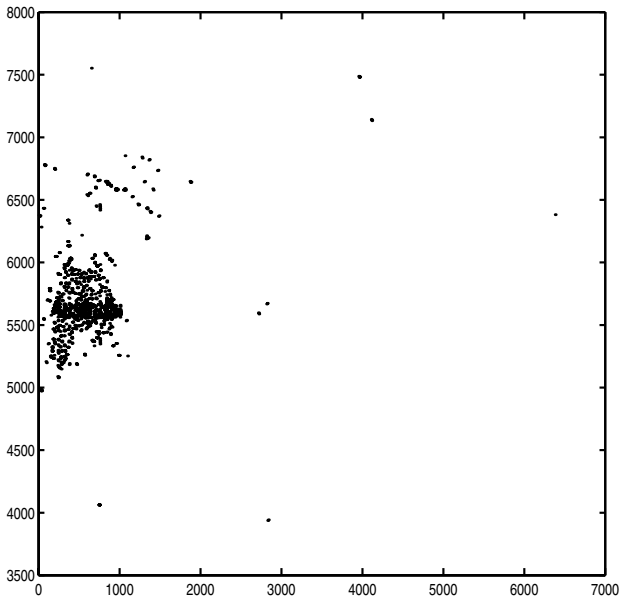

Figure 21. Cluster 3.

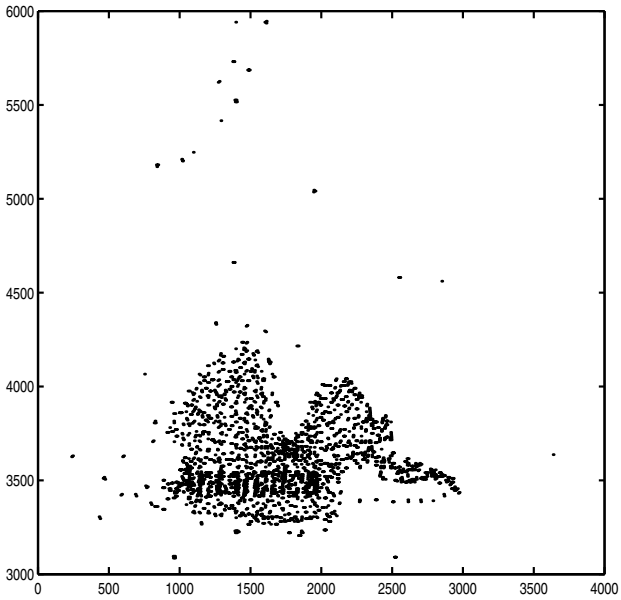

Figure 22. Cluster 4.

step). Unfortunately, the quality of the clusters are not near as good as the ones shown for the second initialization algorithm.

4.9. Real data set 2

We have tried FC with another real dataset. This one represents the global population distribution in the year 1990 (Yi-Fan Li, 1990). The dataset is organized in grid cells of one by one degrees. So, each cell becomes a three dimensional data point, containing its coordinates (center of the cell), and its population. After applying initialization algorithm 2 and FC to it, we obtained the clusters shown in figures 23 and 24. The clusters formed along the main land areas: American (North, Central and South-) continent and Eurasia. Recall that in initialization algorithm 2 we do not set the number of clusters initially, but they are automatically found by the algorithm (so, naturally our technique found 2 clusters in this data set). FC took 89 seconds to cluster the 12,857 points of the data set (87 seconds were spent in the initialization algorithm, using 3,214 points and 2 seconds in the incremental step, to cluster the remaining points). We also tried to cluster this data set with BIRCH and CURE. BIRCH output was also two clusters, both of them circles, centered at coordinates (27.0824, 63.5497) and (16.7907, −83.2713) and with radii 53.8442 and 41.0408. Due to the "spherical" nature of these clusters, 1,026 points were regarded by BIRCH as outliers. With CURE, we also found two clusters, one with 2,260, and the other with 6,064, while 4,533 points were considered outliers (more than 35% of the data points).

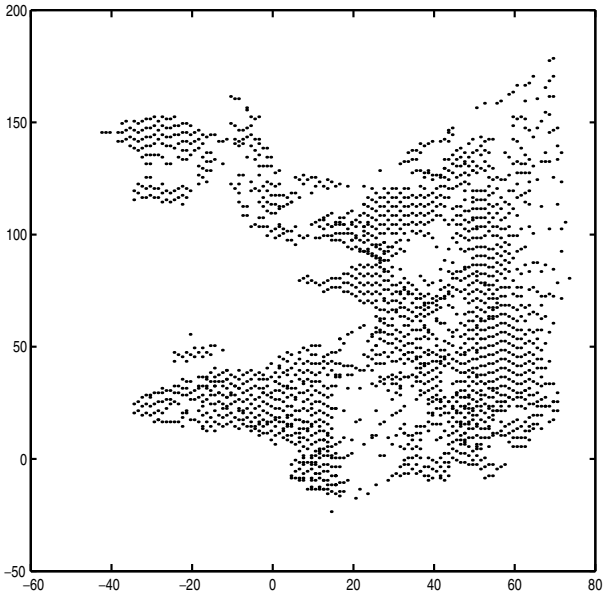

Figure 23. Cluster 1 in the population distribution data set.

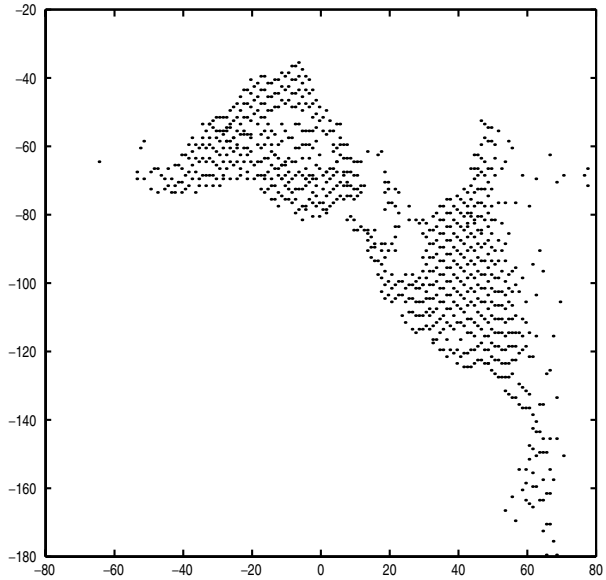

Figure 24. Cluster 2 in the population distribution data set.

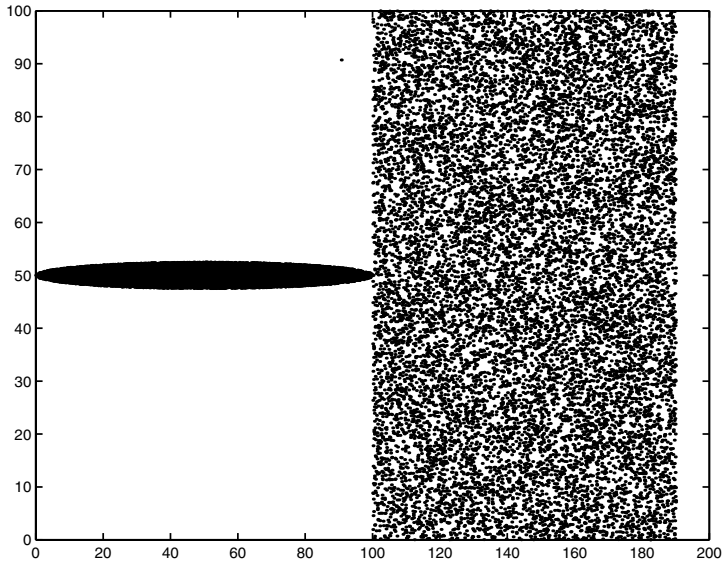

Figure 25. A data set used to compare FC, BIRCH, and CURE.

4.10. Comparison of FC with CURE and BIRCH

We used FC, CURE, and BIRCH to cluster the data set in figure 25. We use the first initialization algorithm for FC, since its running time is the least favorable for FC. (Quality results are similar when using the second initialization algorithm.) Figure 26 shows how CURE and FC perform for this data set for two different sizes (300 and 3,000 points). CURE declares 104 (34.6%) and 1,025 (34.16%) of the points as outliers, while FC is able to place all the points correctly. The running time of CURE and BIRCH are, for these data sets, smaller than that for FC (simply because the later is dominated by the initialization time). But as the data sets grow in size, we noticed that CURE's and BIRCH's running time increases more rapidly than that of FC.

Figure 27 shows the results of comparing BIRCH and FC using the data set of figure 25, but using a larger number of points than in the previous experiment. (CURE was not used in this experiment since its running time was too big.) We can see that FC outperforms BIRCH in quality, and although BIRCH is faster for the smaller data set of 30,000 points, its running time surpasses that of FC for the 3 million point data set.

Finally, we also tried the data set of figure 9 with 20,000 points on CURE, to compare its results with those reported in figure 10. Figure 28 shows the comparison. CURE declares 6,226 (31.13%) points as outliers, placing the rest in the right clusters, while FC places all the points correctly. For this larger set, FC's running time clearly outperforms CURE's. The running time reported in the table for our algorithm corresponds to using the first initialization algorithm. The second initialization algorithm results in similar quality and running times of 0.14 sec. for the 300 point set and 14 s. for the 3,000 point data set.

N	Algorithm	time	clusters found	points in cluster	Coming from cluster1	Coming from cluster2	accuracy
300	CURE	0.26s.	1	69	69	0	46 %
			2	127	0	127	84.6%
			outliers	104	-	-	-
	FC	49s.	1	150	150	0	100%
			2	150	0	150	100 %
	BIRCH	0.34s.	1	72	72	0	48%
			2	192	150	42	100%
			outliers	36	-	-	-
3,000	CURE	22s.	1	1,626	1,146	480	76.4 %
			2	349	0	349	23.26%
			outliers	1,025	-	-	-
	FC	49s.	1	1,486	1,481	5	99.7%
			2	1,514	19	1,496	98.8 %
	BIRCH	0.98s.	1	941	941	0	62.7%
			2	1964	1500	464	100%
			outliers	95	-	-	-

Figure 26. Results of using FC, BIRCH and CURE in the data set of figure 25.

N	Algorithm	time	clusters found	points in cluster	Coming from cluster1	Coming from cluster2	accuracy
30,000	FC	49s.	1	14,819	14,802	17	98.68 %
			2	15,181	197	14,984	99.89 %
	BIRCH	5s.	1	11,962	11,962	0	79.74%
			2	17,968	2,968	15,000	100 %
			outliers	70	-	-	-
3,000,000	FC	401s.	1	1,482,228	1,480,564	1,664	98.70 %
			2	1,517,772	19,434	1,498,338	99.88 %
	BIRCH	544s.	1	1,324,351	1,324,251	0	88.28%
			2	1,665,626	1,500,000	165,626	100 %
			outliers	10,023	-	-	-

Figure 27. Results of using FC and BIRCH in the data set of figure 25.

Algorithm	time	clusters found	points in cluster	Coming from cluster1	Coming from cluster2	accuracy
CURE	2,520s	1	4,310	4,310	0	43.10 %
		2	9,464	0	9,464	94.64 %
		outliers	6,266	-	-	-
FC	52s.	1	10,000	10,000	0	100 %
		2	10,000	0	10,000	100 %
BIRCH	3.89 s.	1	9,654	9,654	0	96.5 %
		2	9,987	0	9,987	98.7%
		outliers	359	-	-	-

Figure 28. Results of using FC and CURE in the data set of figure 9.

5. Related work

The topic of clustering has received ample attention in the research community throughout the years. A recent tutorial (Hinneburg and Keim, 1999), classifies clustering methods in three varieties:

5.1. Model- and optimization-based approaches

Most of the initial approaches devised for clustering belong to this class. The methods usually start with an initial partition and use an iterative strategy to optimize a function.

The K-Means algorithm (Fukunaga, 1990; Selim and Ismail, 1984) uses k initial prototypes of center of gravity of the clusters, and then iteratively assigns the data points to the nearest prototype and shifts the prototypes towards the mean of the clusters obtained. K-means was initially devised as an in-memory algorithm, but variants of it have been recently devised for large datasets that do not fit in main memory (Bradley et al., 1998).

The Expectation Maximization algorithm (Lauritzen, 1995) follows a similar approach to K-Means, by iterating on the process of estimating the parameters of k Gaussian distributions, optimizing the probability that the mixture of Gaussian distributions fits the data.

CLARANS (Ng and Han, 1994), is the first method designed for large data sets. CLARANS k medoids, assigns points to the nearest medoid and follows an iterative procedure to optimize a distance function. CLARANS has, however, a large computational complexity (Wang et al., 1997) ($O(kN^2)$, where N is the size of the data set) and due to its randomized approach in selecting medoids, the quality of the results cannot be guaranteed. Moreover CLARANS does not perform well for complex cluster structures.

5.2. Density-based methods

DBSCAN (Ester et al., 1996), uses a density-based notion of clusters: for each cluster, the neighborhood of a given radius must contain at least a minimum number of points (in other words, the density of the cluster must exceed a threshold). DBSCAN can discover clusters of arbitrary shape, uses an R^*—tree to achieve better performance and has complexity $O(N \log N)$.

STING (Statistical INformation Grid-based method) (Wang et al., 1997), divides the space in rectangular cells using a hierarchical structure and proceeds to store the statistical parameters (mean, variance, type of distribution). Then the clusters are determined as the density-connected components of the grid. The complexity of the method is $O(N)$, but it fails to find some arbitrary shaped clusters.

Hierarchical Grid Clustering (Schikuta, 1996), also organizes the space as a grid, sorting the cells according to their density and scanning and merging adjacent cells to form a hierarchy.

CURE (Guha et al., 1998) is also a hierarchical clustering algorithm that tries to overcome the problems found in traditional hierarchical algorithms when they attempt to merge clusters: at one extreme the clusters are represented by a single point (or centroid), at the other by the complete set of points that are members of the cluster. Both extremes fail to

deal well with non-spherical clusters. CURE adopts a middle ground approach, choosing a fixed-size set of points to represent clusters.

WaveCluster (Sheikholeslami et al., 1998), is also a grid-based approach with complexity $O(N)$, that treats points as multidimensional feature vectors, quantizes the feature space, and then assigns the points to the corresponding quantized units. WaveCluster proceeds to apply wavelet transform on the feature space, find the connected components in the subbands of the transformed space, calling them clusters and finally assign the points to their corresponding clusters.

5.3. Hybrid methods

BIRCH (Zhang et al., 1996), builds a hierarchical data structure, the CF-tree—a height-balanced tree—to incrementally cluster incoming points. BIRCH tries to come up with the best clusters with available main memory, while minimizing the amount of I/O required. Results can be improved by allowing several passes over the data set, but in principle one pass suffices to get a clustering, so the complexity of the algorithm is $O(N)$. Since each node in the tree has predefined limited capacity, the clusters do not always correspond to natural shapes. (In Sheikholeslami et al. (1998), it is reported that BIRCH does not perform well for non-spherical clusters.) The algorithm is also sensitive to the ordering of the data.

COBWEB (Fisher, 1996) is an incremental clustering technique that falls under the class of conceptual clustering algorithms (intended to discover understandable patterns in data). COBWEB uses the category utility function (Gluck and Corter, 1985) to guide classification and tree formation. There are some similarities between our incremental algorithm and that of COBWEB.

6. Conclusions

In this paper we presented a new clustering algorithm based on the usage of the fractal dimension. This algorithm clusters points according to the effect they have on the fractal dimension of the clusters that have been found so far. The algorithm is, by design, incremental and its complexity is $O(N)$, where N is the size of the data set (thus, it requires only one pass over the data, with the exception of cases in which splitting of clusters needs to be done).

Our experiments have proven that the algorithm has very desirable properties. It is resistant to noise, capable of finding clusters of arbitrary shape and capable of dealing with points of high dimensionality.

We have also shown that our FC algorithm compares favorably with other algorithms such as CURE and BIRCH, obtaining better clustering quality. As the data sets get bigger, FC is able to outperform BIRCH and CURE in running time as well.

Acknowledgments

We like to thank Vipin Kumar and Eui-Hong (Sam) Han for lending us their implementation of CURE. We also like to thank Raghu Ramakrishnan and Venkatesh Ganti for

their BIRCH code, and for spending long hours helping us in the use of BIRCH for our experiments.

This work has been supported by NSF grant IIS-9732113.

References

Backer, E. 1995. Computer-Assisted Reasoning in Cluster Analysis. Prentice Hall.

Belussi, A. and Faloutsos, C. 1995. Estimating the selectivity of spatial queries using the 'Correlation' fractal dimension. In Proceedings of the International Conference on Very Large Data Bases, pp. 299–310.

Bradley, P.S., Fayyad, U., and Reina, C. 1998. Scaling clustering algorithms to large databases. In Proceedings of the Fourth International Conference on Knowledge Discovery and Data Mining, New York City.

Bradley, P.S., Fayyad, U., and Reina, C. 1998. Scaling clustering algorithms to large databases (extended abstract). In Proceedings of the ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery.

Carbon Dioxide Information Analysis Center. Contributor: Yi-Fan, Li. 1990. Global population distribution. URL http://cdiac.esd.ornl.gov/ftp/db1016/.

Chernoff, H. 1952. A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations. Annals of Mathematical Statistics, 23:493–509.

Domingo, C., Gavaldá, R., and Watanabe, O. 1998. Practical algorithms for online selection. In Proceedings of the first International Conference on Discovery Science.

Domingo, C., Gavaldá, R., and Watanabe, O. 2000. Adaptive sampling algorithms for scaling up knowledge discovery algorithms. Discovery Science, 1999:172–183.

Domingos, P. and Hulten, G. 2000. Mining high-speed data streams. In Proceedings of the 2000 Conference on Knowledge Discovery and Data Mining, pp. 71–80.

Ester, M., Kriegel, J.P., Sander, J., and Su, X. 1996. A density-based algorithm for discovering clusters in large spatial databases with noise. In Proceedings of the 2nd International Conference on Knowledge Discovery and Data Mining, pp. 226–231.

Faloutsos, C. and Gaede, V. 1996. Analysis of the Z-ordering method using the Hausdorff fractal dimension. In Proceedings of the International Conference on Very Large Data Bases, pp. 40–50.

Faloutsos, C. and Kamel, I. 1997. Relaxing the uniformity and independence assumptions, using the concept of fractal dimensions. Journal of Computer and System Sciences, 55(2):229–240.

Faloutsos, C., Matias, Y., and Silberschatz, A. 1996. Modeling skewed distributions using multifractals and the '80-20 law'. In Proceedings of the International Conference on Very Large Data Bases, pp. 307–317.

Fisher, D.H. 1996. Iterative optimization and simplification of hierarchical clusterings. Journal of AI Research, 4:147–180.

Fukunaga, K. 1990. Introduction to Statistical Pattern Recognition. San Diego, California: Academic Press.

Gluck, M.A. and Corter, J.E. 1985. Information, uncertainty, and the utility of categories. In Proceedings of the Seventh Annual Conference of the Cognitive Science Society, Irvine, CA.

Grassberger, P. 1983. Generalized dimensions of strange attractors. Physics Letters, 97A:227–230.

Grassberger, P. and Procaccia, I. 1983. Characterization of strange attractors. Physical Review Letters, 50(5):346–349.

Guha, S., Rastogi, R., and Shim, K. 1998. CURE: An efficient clustering algorithm for large databases. In Proceedings of the ACM SIGMOD Conference on Management of Data, Seattle, Washington, pp. 73–84.

Hinneburg, A. and Keim, D. 1999. Clustering techniques for large data sets: From the past to the future. Tutorial Notes for ACM SIGKDD International Conference on Knowledge Discovery and Data Mining.

Jain, A. and Dubes, R.C. 1988. Algorithms for Clustering Data. Englewood Cliffs, New Jersey: Prentice Hall.

Lauritzen, S.L. 1995. The EM algorithm for graphical association models with missing data. Computational Statistics and Data Analysis, 19:101–201.

Liebovitch, L.S. and Toth, T. 1989. A fast algorithm to determine fractal dimensions by box counting. Physics Letters, A141:386–390.

Lipton, R.J. and Naughton, J.F. 1995. Query size estimation by adaptive sampling. Journal of Computer Systems Science, 51:18–25.

Lipton, R.J., Naughton, J.F., Schneider, D.A., and Seshadri, S. 1993. Efficient sampling strategies for relational database operations. Theoretical Computer Science, 116:195–226.

Mandelbrot, B.B. 1983. The Fractal Geometry of Nature. New York: Freeman.

Ng, R.T. and Han, J. 1994. Efficient and effective clustering methods for spatial data mining. In Proceedings of the 20th Very Large Data Bases Conference, pp. 144–155.

Samet, H. 1990. Applications of Spatial Data Structures. Addison-Wesley.

Sarraille, J. and DiFalco, P. FD3. http://tori.postech.ac.kr/softwares/.

Schikuta, E. 1996. Grid clustering: An efficient hierarchical method for very large data sets. In Proceedings of the 13th Conference on Pattern Recognition, IEEE Computer Society Press, pp. 101–105.

Schroeder, M. 1991. Fractals, Chaos, Power Laws: Minutes from an Infinite Paradise. New York: W.H. Freeman.

Selim, S.Z. and Ismail, M.A. 1984. K-means-type Algorithms: A generalized convergence theorem and characterization of local optimality. IEEE Transactions on Pattern Analysis and Machine Intelligence, PAMI-6(1).

Sheikholeslami, G., Chatterjee, S., and Zhang, A. 1998. WaveCluster: A multi-resolution clustering approach for very large spatial databases. In Proceedings of the 24th Very Large Data Bases Conference, pp. 428–439.

Wang, W., Yand, J., and Muntz, R. 1997. STING: A statistical information grid approach to spatial data mining. In Proceedings of the 23rd Very Large Data Bases Conference, pp. 186–195.

Watanabe, O. 2000. Simple sampling techniques for discovery science. IEICE Transactions on Information and Systems.

Zhang, T., Ramakrishnan, R., and Livny, M. 1996. BIRCH: A efficient data clustering method for very large databases. In Proceedings of the ACM SIGMOD Conference on Management of Data, Montreal, Canada, pp. 103–114.