A survey paper for COMP 530

with topic

Towards High-Dimensional Clustering

23 November 1999

Agustino Kurniawan
agustino@ust.hk

Nicolas Benech
benech@ust.hk

Tao Yufei
taoyf@ust.hk

Tian Feng
cstianf@ust.hk

Wang Jiying
cswangjy@ust.hk

Theocharis Malamatos
tmalamat@cs.ust.hk

Course: COMP 530
Fall Semester
Department of Computer Science
The Hong Kong University of Science and Technology

CONTENTS

1. Introduction

Clustering is used in many fields including data mining, statistical data analysis, compression, vector quantization and various business applications. The fundamental clustering problem is to group together data items which are similar to each other.

A clustering algorithm can be employed either as a stand-alone tool or as a preprocessing step for other algorithms which operate on the detected clusters. Cluster analysis plays an important role in solving many problems in pattern recognition and image processing. It is used for feature selection in numerous applications involving unsupervised learning, speech and speaker recognition, image segmentation, image registration and others. Clustering, for example, can help us also answer questions like the following: Is this virus similar to any other virus? Do these viruses form a group with certain common characteristics? Do young people vote differently than their parents? Which kind of customers is a product likely to attract based on its price, quality or uses?

1.1 Clustering Algorithm

A clustering is a type of classification imposed on a finite set of objects. If the objects are characterized as patterns, or points in a *d*-dimensional metric space, the proximity can be the distance between pairs of points, such as Euclidean distance. Unless a meaningful measure of distance, or proximity, between pairs of objects has been established, no meaningful cluster analysis is possible. Various measures have been defined with each having various advantages and disadvantages according to the problem/dataset applied.

There are two basic types of clustering algorithms: partitioning and hierarchical algorithms.

Partitioning algorithms construct a partition of a database DB of *n* objects into a set of *k* clusters where *k* is an input parameter. Each cluster is represented by the center of gravity of the cluster or by one of the objects of the cluster located near its center. Each object is assigned to the cluster with its representative closest to the considered object. Typically, partitioning algorithms start with an initial partition of DB and then use an iterative control strategy to optimize the clustering quality, e.g. the average distance of an object to its representative.

Hierarchical algorithms create a hierarchical decomposition of DB. The hierarchical decomposition is represented by a tree structure that splits DB into small subsets until each subset consists of only one object. In such a hierarchy, each node of the tree represents a cluster of DB. The tree can be build either from the leaves up to the root by merging clusters (agglomerative hierarchical clustering) or from the root down to the leaves by dividing clusters at each step. It is necessary that a termination condition is defined to indicate when the merge or division process should stop.

All traditional partitioning and hierarchical algorithms face some problems. For a partitioning algorithm, it is not easy to determine the "natural" number k of clusters in a database. Also these algorithms tend to split large clusters into smaller ones. On the other hand, a main problem with hierarchical algorithms is the difficulty to choose appropriate parameters for the termination condition. In addition tend to be more prone to outliers (see below).

An important chapter in clustering is the handling of outliers. Outliers (or noise) are points that do not belong to any of the clusters. Typically the neighborhoods of outliers are generally sparse compared to points in clusters and the distance of an outlier to the nearest cluster is

comparatively higher than the distance among points in bonafide clusters themselves. Outliers influence the decisions of the clustering algorithm in an unpredictable and subtle way and if they are not identified and eliminated from the data set usually result in completely wrong clustering results. There have been various techniques developed to limit their effects.

1.2 Clustering in High-Dimensional Databases

Most of recent research related to the task of clustering has been directed to achieving efficiency as larger and larger amounts of data are collected and stored in databases. What is problematic is that most traditional clustering algorithms do not scale well with the size and dimension of the data set. The application of clustering to large databases rises some additional/new requirements for the clustering algorithms:

1) Minimal domain knowledge to determine the parameters, since appropriate values are usually not well predicted when dealing with large databases.

2) Good efficiency, because there can be several hundreds of dimensions and more than one million records in one database. In such a case, the efficiency of most traditional algorithms decreases heavily. Note that the techniques for dimensionality reduction cannot be used in all problems or can be used up to a limited degree.

3) Clusters existing in different subspaces of all dimensions, because in such high-dimensional spaces, it maybe that not all dimensions are relevant to a given cluster. Examining every different subspace for clusters is not efficient as the number of different subspaces is dependent exponentially on the number of dimensions.

1.3 Outline of the paper

This survey paper tries to include clustering algorithms that satisfy to some degree at least one of the above requirements. Not all of them are suited for high-dimensional clustering. This is due to the fact that there are very few clustering algorithms especially designed for higher dimensions (e.g. for d >20) up to now.

In section 2, we present DBSCAN, a clustering algorithm which discovers clusters in a spatial database. It relies on a density-based notion of clusters. It requires only one input parameter and supports the user in determining an appropriate value for it. The OPTICS algorithm is explained in section 3. OPTICS can be considered as an extension to the DBSCAN that handles the global density issue and can be used as a basis for cluster analysis.

BIRCH is the algorithm of Section 4. It is one the first algorithms proposed for large datasets. It uses a preclustering stage to reduce the size of the data on which a traditional hierarchical clustering algorithm will successively run. In Section 5, an new hierarchical clustering algorithm is given named CURE which can recognize a wide variety of clusters by using multiple representatives for each cluster and which applies various techniques for its efficient use in large databases.

We present CLIQUE in section 6, a clustering algorithm that can identify dense clusters in subspaces of maximum dimensionality. It generates cluster descriptions in the form of DNF expressions that are minimized to be easy in comprehension. In section 7, we introduce PROCLUS, which provides an approach finding the clusters with different dimensions. This algorithm ends up with a partition of the original data set.

2 DBCSAN: Density-Based Spatial Clustering of Applications with Noise

The algorithm presented in [1] tries to recognize the clusters by taking advantage of the fact that within each cluster the typical density of points is considerably higher than outside of the cluster. Furthermore, the density within areas of noise is lower than the density in any of the clusters. This idea leads to a density-based approach for clustering. In this algorithm, we assume that the objects to be clustered are represented as ***points*** in the measurement space. We will formalize the notion of "density-based clusters" and "noise" in a database *D* of points in *k*-dimensional space *S*. Note of that, both the notion of clusters and the algorithm DBSCAN, work well in 2D or 3D Euclidean space as in some high dimensional feature space.

2.1 A Density Based Notion of Clusters

The choice of a distance function will determine the shape of the neighborhood, using *dist(p,q*) to denote the distance of two points p and q in the database space. For example, if we use $| x_2 - x_1 | + | y_2 - y_1 |$ to calculate the distance in 2D space, the shape of the neighborhood is a diamond; if we use the Euclidean distance, the shape is a cycle. Note, both the notion and the algorithm works with any distance function so that a specific function can be chosen for some given application.

Definition 1: (Eps-neighborhood of a point)

Let N_{Eps} (p) be the *Eps-neighborhood* of a point p, it is a set of points with the distance to p not more than Eps.

We can consider that there are two types of points in a cluster. One is the core point; another one is the border point (see Figure 1). In general, an Eps-neighborhood of a border point contains fewer points than an Eps-neighborhood of a core point.

Definition 2: (directly density-reachable)

A point p is *directly density-reachable* from a point q wrt.[1] Eps and MinPts, if p is inside N_{Eps} (q) and the cardinality of N_{Eps} (q) is larger than MinPts (core point condition).

[1] wrp. = with respect to

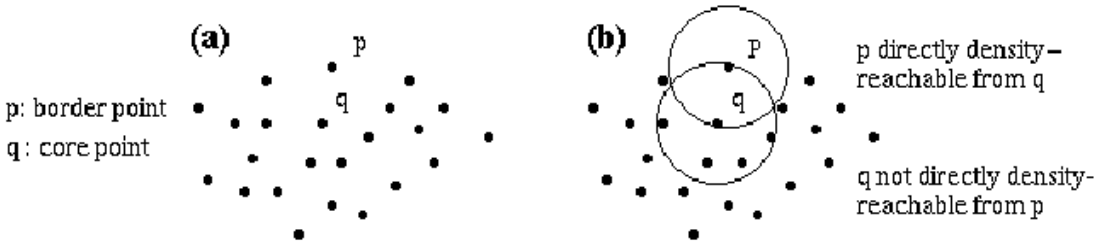


Figure 1: core points and border points

Definition 3: (density-reachable)

A point p is density-reachable from a point q with respect to Eps and MinPts, if there is a chain of points $p_1, \ldots p_n$, $p_1 = p$, $p_n = q$ such that p_{i+1} is directly density- reachable from p_i.

Density-reachability is a standard extension of direct density-reachability.

Definition 4: (density-connected)

A point p is *density-connected* to a point q wrt. Eps and MinPts if there is a point o such both, p and q are density-reachable from o wrt. Eps and MinPts.

Density-connectivity is symmetric. If a point a can be density-reachable from other points wrt. Eps and MinPts, it also can be density-connected to them.

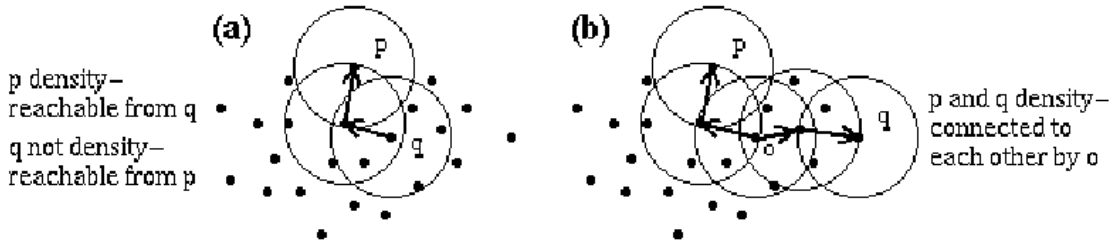


Figure 2: density-reachability and density-connectivity

Definition 5: (cluster)

Let D be a database of points. If a non-empty subset C of D satisfies the following conditions, it is a *density-based cluster* in D.

1) Every p, q in C: if p is in C and q is density-reachable from p wrt. Eps and MinPts, then q is also in C. (Maximum Condition)

2) Every p, q in C: p is density-connected to q wrt. Eps and MinPts.(Connecting Condition)

Definition 6: (noise)

Let C_1,..., C_k be the clusters of the database D wrt. Eps_i and $MinPts_i$, i = 1,…k, then the set of points not belonging any cluster C_i is defined as *noise*.

According to the notion of density-based clusters, we can get some observations.

(1) A cluster C wrt. Eps and MinPts contains at least MinPts points because it is defined as a set of density-connected points which have the maximal density-reachability.

(2) A cluster C wrt. Eps and MinPts is uniquely determined by any of its core points, since any point which can be density-reachable from a core point, can also be density-reachable from other core points.

2.2 The Algorithm

In this section, we present the algorithm DBSCAN (Density Based Spatial Clustering of Applications with Noise) which can discover the clusters and noise in a spatial database according to the definition 5 and 6. Ideally, we should in advance get the information of the appropriate parameters Eps and MinPts of each cluster. But this is not at all easy. However, we use a simple approach to determine the parameters. DBSCAN uses global values of Eps and MinPts for all the clusters. The density parameters of the thinnest cluster seem to be good approximation for these global parameter values.

The main idea of DBSCAN is a two-step approach to find one cluster. First, choose an arbitrary point from the database satisfying the core point condition. Second, retrieve all points that are density-reachable from the core point to find one cluster. Those points that dissatisfy the core condition and those points that are not density-reachable by any core points will be marked to be noise. This procedure is based on the fact that a cluster is uniquely determined by any of its core points.

For each of the n points of the database, we have at most one scan to find its Eps-neighborhood. The retrieval can be successfully performed by successive region query, while region query is supported efficiently by spatial index such as R-tree, R*-tree. Since the Eps-neighborhood is in practice small compared to the whole size of data space, the average run time complexity of a single region query is O(log n). Thus, the average run time complexity of DBSCAN is O(n * n) or O (n * log n) depending on whether the data space is indexed or not.

Since we use the same values for Eps and MinPts to all clusters, DBSCAN will probably merge two clusters of different density into one, if the two clusters are not "far" from each other. We can define the distance between two sets of points S and T, as dist(S,T) to be the minimum distance between points in S and points in T. So two sets of points of different density can be separate only if the distance between the two is larger than Eps.

2.3 Determining the Parameters Eps and MinPts

Since the whole data space is very large, it is very difficult to choose proper values for the parameters used in the algorithm. However, we develop a simple but heuristic approach to determine the parameters. The approach is based on the following observation. Let d be the distance of a point p to its k-th nearest neighbor, then the d-neighborhood of p contains exactly k+1 points for almost all points p. Furthermore, changing k for a point in a cluster does no result in large changes of d.

For given k we define a function k-dist from the database D to the real numbers, mapping each point to the distance from its k-th nearest neighbor. When sorting the points of the database in descending order of their k-dist value, the graph of this function gives some hints concerning the density distribution in the database. We call this graph the sorted k-dist graph. If we choose an arbitrary point p, set the parameter Eps to k-dist (p) and set the parameter MinPts to k, all points with an equal or smaller k-dist value will be core points. The threshold point is the first point in the first "valley" of the sorted k-dist graph (see figure 3). All points with a higher k-dist value (left of the threshold) are considered to be noise, all other points (right of the threshold) are assigned to some cluster.

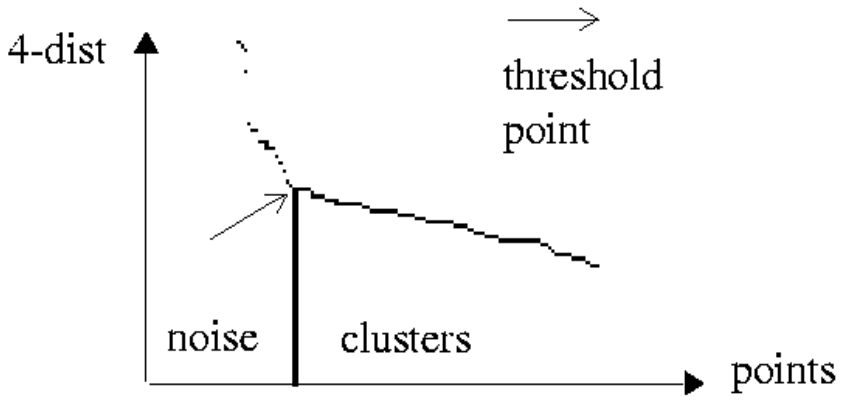


Figure 3: sorted 4-dist graph for a sample database

In general, it is very difficult to detect the first "valley" automatically, but it is relatively simple for a user to see this valley in a graphical representation. Therefore, we propose to follow an interactive approach for determining the threshold point.

DBSCAN needs two parameters, Eps and MinPts. However, the experiments by the authors indicate that the k-dist graphs for k>4 do not significantly differ from the 4-dist graph and, furthermore, they need considerably more computation. Therefore, we eliminate the parameter MinPts by setting it to 4 for all databases (for 2-dimensional data). The following interactive approach is proposed for determining the parameter Eps of DBSCAN:

1) The system computes and displays the 4-dist graph for the database.

2) If the user can estimate the percentage of noise, this percentage is entered and the system derives a proposal for the threshold point from it.

3) The user either accepts the proposed threshold or selects another point as the threshold point. The 4-dist value of the threshold point is used as the Eps value of DBSCAN.

2.4 Conclusion

We presented the clustering algorithm DBSCAN that relies on a density-based notion of clusters. It requires only one input parameter (Eps) and supports the user in determining an appropriate value for it. It has the advantage that it can identify clusters of arbitrary sizes. However the clusters identified may vary greatly according to the value of the input parameter. A global density value is not always sufficiently descriptive.

There are also some other open problems to be considered. One of them is that only point objects have been considered (for example, what if we are looking for clusters of polygons in space). Another is that this technique is not directly applicable in higher dimensions, as then an area can be dense in relation to a small number of dimensions but very sparse in relation to the whole space.

3 OPTICS for Cluster Analysis

Paper [2] introduces an algorithm called OPTICS, Ordering Points To Identify the Clustering Structure, which can be considered as a preprocessing step for other algorithms, such as DBSCAN algorithm. OPTICS algorithm creates an augmented ordering of the points object database representing its density-based clustering structure. Once this algorithm is implemented in the database, both traditional clustering information (e.g. representative points and arbitrary shaped clusters) and intrinsic clustering structure can be extracted.

There are several ways to represent the result. For medium size data sets, the cluster-ordering information can be represented graphically. Very large high-dimensional data sets require a visualization technique that can visualize more data items at the same time. For this reason, a pixel-oriented technique is used instead. The presentations are suitable for interactive exploration of the intrinsic clustering structure and give additional insight into the distribution and correlation of the data sets for the purpose of the cluster analysis.

The OPTICS algorithm does not produce a clustering of a data set explicitly, but can be described as giving ‘properties’ to the points object of the database which contains information equivalent to the density based clustering. The information given to each points object is in order and valued relatively based on its neighbor point object.

3.1 Clustering Algorithm Limitations

This algorithm is developed to overcome the limitations of many clustering algorithm such as determining input parameters values, sensibility to input parameters values and global density parameter issue. Many clustering algorithms require values for input parameters which are hard to determine, especially for real-world data sets containing high-dimensional objects which the domain knowledge are often not known in advance. In many cases, the algorithms are also very sensible to these parameter values. A slightly different parameter input often produces very different results. In addition, in high dimensional real-data sets, the intrinsic cluster structure often cannot be characterized by global density parameters.

One of the examples is the DBSCAN algorithm that works with global density parameter. A global density parameter to search clusters in the database has performance advantage over many density parameters in large databases, however some cluster structure can not be revealed. This can be illustrated in Figure 1.

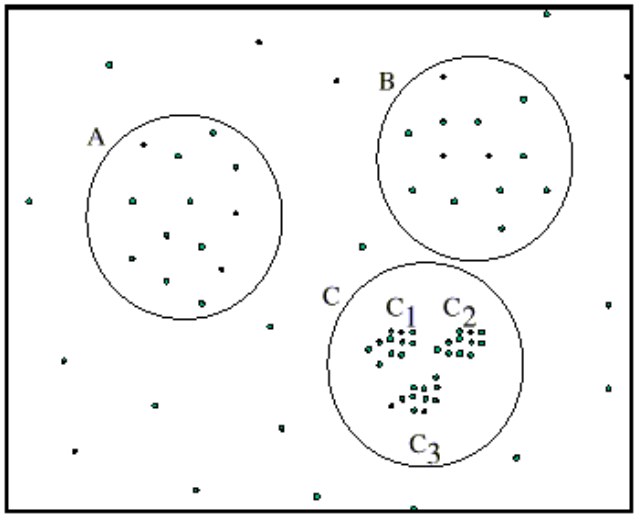


Figure 1: Example of Clusters with Different Density Parameters

The DBSCAN algorithm with global density parameter will not possibly detect the clusters A, B, C1, C2, and C3 simultaneously. A high Eps value will detect only the clusters A, B, and C, while a low Eps value will detect the clusters C1, C2, C3 which in this case the cluster sets A & B are considered as noise.

The existing clustering algorithm, such as Single-Link method, can be used to detect and analyze such clustering structure of small data sets. However, the results produced by hierarchical algorithms, such as dendrogram, are difficult to understand and analyze for more than a few hundred objects. Another alternative is using DBSCAN algorithm with different Eps parameters settings. However, there are too many possible parameter values that require a high cost to test and we may still miss some cluster levels.

3.2 OPTICS Algorithm

The OPTICS algorithm is based on density-based algorithm which is considered as an extended DBSCAN algorithm for an infinite number of distance parameters ε_I which are smaller than ε ($0 \le \varepsilon_I \le \varepsilon$). The basic principle of this algorithm is to give every point object a special ordering with respect to its density-based structure in the database. The properties / information given to each point object by this algorithm consist of two values, called the core-distance and a reachability-distance. How this algorithm works can be explained by how the core-distance and a reachability-distance information are given to each object.

The core-distance information of an object p can be described as in the following definition:

- Assume: p is an object in database D.

  ε is a distance value – you can think of it as a radius.

  Nε(p) is the ε-neighborhood of p.

  MinPts is a natural number that corresponds to the density.

  MintPts-distance(p) is the distance from p to its *MinPts'* neighbor.

- Core-distance of an object p =

$$\begin{cases} UNDEFINED, \text{ if } Card(N_{\varepsilon}(p)) < MinPts \\ MinPts\text{-}distance(p), \text{ otherwise} \end{cases}$$

We can say that the core-distance of an object p is the smallest distance (ε') between p and an object in its ε-neighborhood. With the conditions that p would be a core object with respect to (ε') value and the neighbor object is contained in Nε(p). Otherwise, the core-distance of an object p is UNDEFINED.

The reachability-distance value of an object p is a distance between p to another object, for example object o. It is described as the reachability-distance of object p with respect to object o, which can be explained as in the following:

- Assume: p and o are objects in database D.

  Nε(o) is the ε-neighborhood of o.

  MinPts is a natural number that correspond to density.

- Reachability-distance of p with respect to o =

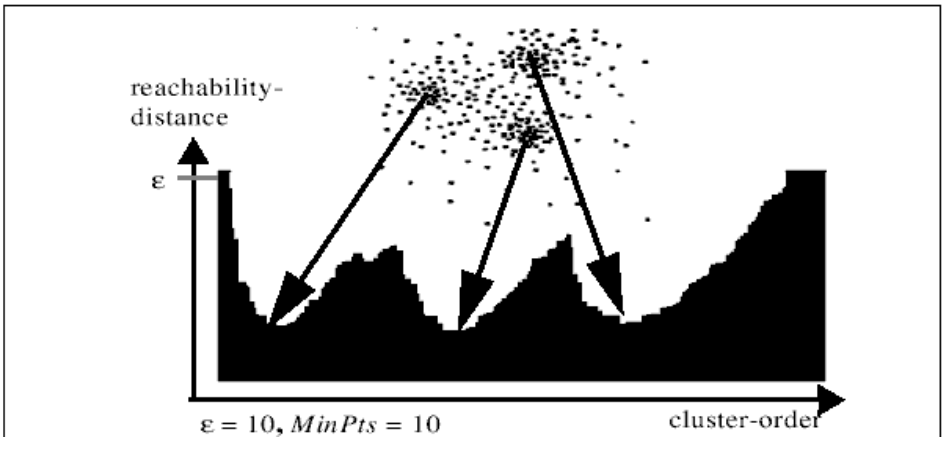
$$\begin{cases} UNDEFINED, \text{ if } \left|N_{\varepsilon}(o)\right| < MinPts \\ max(core\text{-}distance(o), distance(o,p)), \text{otherwise} \end{cases}$$

We can also say that the reachability-distance of an object p with respect to object o is either the core-distance(o) or the distance from o to p, which one is the biggest. With the condition that if o is a core object then p is directly density-reachable from o. In this case the reachability-distance value of object p can not be smaller than the core-distance of o, because for the smaller distances no object is directly density-reachable from o. If o is not a core object, then the reachability-distance of an object p with respect to object o is UNDEFINED.

OPTICS algorithm does not assign cluster membership, but store the order information in which the objects are processed. However, having the augmented cluster ordering of a database with respect to ε and MinPts, we can simply extract any density-based cluster from this order by scanning the cluster ordering and assigning cluster memberships depending on the reachability distance and the core distance of the objects. The scanning and the extracting process, which can be considered as an extended DBSCAN algorithm, will assign cluster memberships of each object.

3.3 **Visualization using Reachability-Plots**

After this algorithm creates an ordering points of a database and stores the values of each object, this information can be used to analyze the cluster. The clustering structure of the data sets can be represented and understood graphically. One example of the visual representation



of this cluster is a simple 2-D graph called reachability-plots. Basically, the reachability-plots will draw the reachability-distance value in the y-axis and the point ordering in the x-axis for the graph, as illustrated in Figure 2.

Figure 2: Visualization of the Cluster Ordering

In this graph, the cluster order represented in x-axis is independent from the dimension of the data sets. A cluster is represented by a 'valley' in the graph. A 'hill' in the graph implies that the reachability-distance of the points is 'far' meaning that the point must be either a noise or a point which are not part of the sub-cluster represented in the lower 'valley'. Having this representation, we can understand and analyze the clustering structure of the database.

The graph visualization depends on ε and MinPts parameters setting meaning that we can get different graph representation by giving different value of ε and MinPts. The fact that there are many possible representations of this cluster can be illustrated in the Figure 3. However, the clustering structure revealed from these different visualizations would be more or less the same and therefore it could be still recognized.

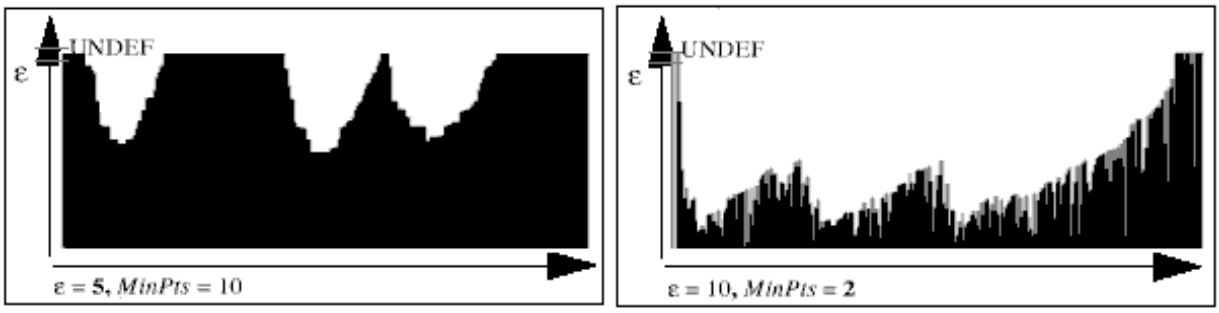


Figure 3: Visualization Variations from Different Parameters Setting

The value of the MinPts has to be relatively large in order to get a good visualization. Based on the experiments, a MinPts value between 10 and 20 will always give a good result, while the ε value will impact the 'height' of the graph which does not significantly effect the structure and smoothness of the graph surface.

3.4 Conclusions

Optics algorithm is suitable to be used as a tool for cluster analysis as it can extract traditional clustering information, such as representative points and arbitrary shaped clusters. In addition, Optics can also reveal the intrinsic and hierarchical clustering structure. In contrast with the DBSCAN method, Optics provide a solution to the global density issue by giving every point object the augmented cluster-ordering containing information which is equivalent to the density-based clustering that corresponds to a broad range of parameter settings.

The visualization technique proposed in this paper provides a good representation of the clustering structure, thus it can be used as a tool to get insight into the distribution of a data set. In addition, the visualization can also reveal hierarchical cluster with different sizes, densities and shapes.

However, some limitations exist in this algorithm. The visualization technique of this algorithm requires proper values in the parameter settings in order to get good results. The

experiments have been done to get a range of values that are considered as good values, but the usability of values may not be applicable to all types of data sets. Different number of objects, size of dimensions and densities may require different parameter settings that have to be defined properly.

Another issue with Optics implementations is related to the recent issue in developing effective and efficient indexing for high dimensional data. Without a well-developed indexing technique, OPTICS has to scan the whole database in n^2 in order to give the augmented order on every point, which make this algorithm unfeasible for a very large high dimensional data set.

Another challenge for this algorithm is how to devise a mechanism to handle dynamic update, such as deleting, adding and moving points in the database without reconstructing the whole ordering information. So, what needs to be further explored is techniques to provide an incremental update of the ordering information with a very low cost.

4 BIRCH

BIRCH (Balanced Iterative Reducing and Clustering using Hierarchies) [3] is a data clustering method. The authors, Tian Zhang, Raghu Ramakrishnan and Miron Livny demonstrate in their paper that this algorithm is suitable for very large databases. BIRCH can find a rather good clustering with a single scan of the data space.

4.1 BIRCH properties

The clustering decision is local. That means that it does not need to take into consideration the other clusters and instead uses features of the cluster. An excellent property is the exploitation of the non-uniform repartition of the points in the data space. Dense regions of points are considered as clusters and data points outside of clusters are considered as outliers (points that can be considered as noise and might be removed). It is an incremental method which does not require the entire dataset in advance. The latter is scanned only once thus reducing considerably I/O costs.

4.2 Background

Before we go further into the BIRCH algorithm, we need to define some properties we will use later.

Properties of a single cluster

Formal definition

Given N d-dimensional data points in a cluster $\{\vec{X}_i\}$ where i=1,2, ... , N, the centroid $\vec{X}_0$, radius R and diameter D are defined as follow:

$$\vec{X}_0 = \frac{\sum_{i=1}^{N} \vec{X}_i}{N}$$

$$R = \left[\frac{\sum_{i=1}^{N} \left(\vec{X}_i - \vec{X}_0\right)}{N} \right]^{\frac{1}{2}}$$

$$D = \left[\frac{\sum_{i=1}^{N} \sum_{j=1}^{N} (\vec{X}_i - \vec{X}_j)^2}{N(N-1)} \right]^{\frac{1}{2}}$$

Comprehensive definition

The centroid $\vec{X}_0$ is intuitive. R is the average distance from the member points of a cluster to its centroid and D is the average pairwise distance within a cluster. R and D are two different measures of the tightness of the cluster around the centroid.

We also need to define how to measure the closeness between two clusters.

Properties between two clusters: distances

Given a d-dimensional space and the centroids of two clusters: $\vec{X}_{01}$ and $\vec{X}_{02}$, *the centroid Euclidian distance* D_0 and *centroid Manhattan distance* D_1 are defined as follow:

$$D_0 = \sqrt{(\vec{X}_{0_1} - \vec{X}_{0_2})^2}$$

$$D_1 = \left|\vec{X}_{0_1} - \vec{X}_{0_2}\right| = \sum_i^d \left|\vec{X}_{0_1}{}^i - \vec{X}_{0_2}{}^i\right|$$

There are many other distances and from the following, the authors of BIRCH use preferably D_2 and D_4. Given N_1 d-dimensional data points in a cluster: $\{\vec{X}_i\}$ where i = 1,2,...,N_1, and N_2 data points int another cluster: $\{\vec{X}_j\}$ where j = N_1+1, N_1+2,..., N_1+N_2, the average inter-cluster distance D_2, average intra-cluster distance D_3 (which is in fact D of the merged cluster) and variance increase distance D_4 of the two clusters are defined as follow:

$$D_2 = \left(\frac{\sum_{i=1}^{N_1} \sum_{j=N_1+1}^{N_1+N_2} (\vec{X}_i - \vec{X}_j)^2}{N_1 N_2} \right)^{\frac{1}{2}}$$

$$D_3 = \left(\frac{\sum_{i=1}^{N_1+N_2} \sum_{j=1}^{N_1+N_2} (\vec{X}_i - \vec{X}_j)^2}{(N_1 + N_2)(N_1 + N_2 - 1)} \right)^{\frac{1}{2}}$$

$$D_4 = \left(\sum_{k=1}^{N_1+N_2} \left(\vec{X}_k - \frac{\sum_{l=1}^{N_1+N_2} \vec{X}_l}{N_1+N_2} \right)^2 - \sum_{i=1}^{N_1} \left(\vec{X}_i - \frac{\sum_{l=1}^{N_1} \vec{X}_l}{N_1} \right)^2 - \sum_{j=N_1+1}^{N_1+N_2} \left((\vec{X}_j - \frac{\sum_{l=N_1+1}^{N_1+N_2} \vec{X}_l}{N_2}) \right)^2 \right)^{\frac{1}{2}}$$

Clustering Features (CF)

Rough definition

A clustering feature (CF) is a set of three data from which can be extracted all information concerning a subcluster of data points.

Formal definition

Given N d-dimensional data points in a cluster: X_i where i=1, 2, … , N the CF entry of the cluster is defined as the triple:

$$CF = (N, \mathbf{LS}, SS) = (N, \sum_{i=1}^{N} \vec{X}_i, \sum_{i=1}^{N} \vec{X}_i^{\,2})$$

N is the number of data points in the cluster;

LS is the linear sum of the N data points;

SS is the square sum of the N data points.

CF Tree

Non-leaf nodes

A non-leaf node may contain up to B entries of the form $[CF_i, child_i]$ where $1 \leq i \leq B$. CF_i is the CF entry of the subcluster $child_i$ and $child_i$ is a pointer to the i^{th} child node.

Leaf nodes

Each leaf node may contain up to L CF entries. To provide a more efficient scan of all the leaf nodes, they are all chained with two pointers on the previous and next nodes. The CF entries must satisfy a threshold T condition: the diameter of the cluster has to be less than the threshold value.

One of the BIRCH requirements is that each node shall fit in a page of size P. The parameters B and L are thus determined by P, which can of course be adjusted for performance tuning.

Insertion of new data in the CF tree

This insertion algorithm is very similar to those used for B trees.

1. The first step is to identify the appropriate leaf. To achieve this, we descend the CF tree recursively by choosing the closest child node according to a metric distance (as described in the paragraph *properties between two clusters*). This might be the centroid

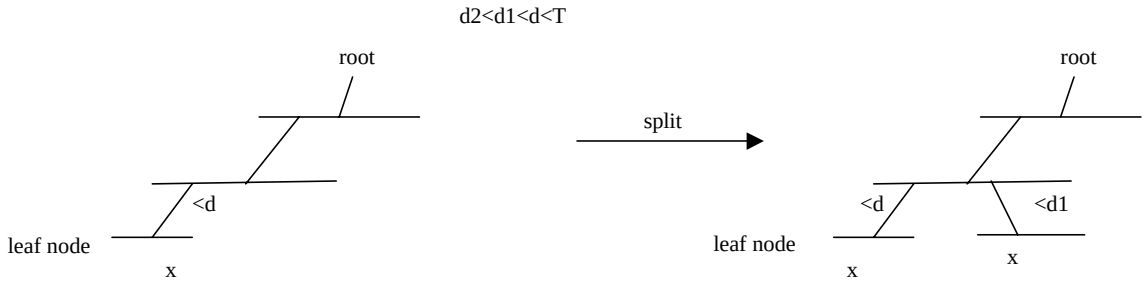
distance D_0, the Manhattan distance D_1 or another metric distance such as the average inter-cluster distance D_2, the average intra-cluster distance D_3 or the variance increase distance D_4.

2. Then upon reaching the leaf node, we find the closest leaf entry, say Li. We now compute the metric distance for the "cluster" L_i' = "cluster" L_i + new data. If this value still comply to the threshold condition we say that L_i can 'absorb' the new data (the data point or subcluster is contained in L_i) and we can update the CF entry of L_i consequently. If the threshold condition is not satisfied, then we need to add a new entry to the leaf. If there is space on the leaf node for this entry, i.e. if the number of current entries is less than L, we can jump to step 3. Otherwise we must split this leaf node.

3. We now have to modify the path to the leaf. Once the new data has been inserted, we have to update the CF information of each non-leaf entry on the path to the leaf, i.e. each non-leaf parent node of the lead up to the root of the CF tree. If step 2 didn't require a split we only need to update all the CF entries. Otherwise we need to insert a new non-leaf entry into the parent node of the new leaf. Since the parent node may also be full, the split propagation may go up to the root of the CF tree. In this case, the height of the tree would be increased by one.

4. Due to the page size, a lot of splits may occur, reducing the clustering quality. To try to reduce this problem we can find the node where the split propagation stops, take the two closest entries (still according to a metric distance and which should not be those corresponding to the split) and try to merge them. If the merged entries fit on a single page, this will free a node space and otherwise improve the distribution of entries in the closest two children.

The limitation of the page size produces undesirable side effects. It is easily understood that a node does not always correspond to a natural cluster. But less obvious is the fact that occasionally a same data point might be entered into two different leaf entries. Let's say we first enter x<d2 in the above example. After other processing, we re-enter x<d2. But this time, due to a split occurrence, one child leads to entries <d and another to entries <d1. Since the latter is better, x will be inserted here, being therefore inserted twice in the CF tree. The step 3 of the BIRCH algorithm (which is described in the next section) is here to remedy to this problem.



4.3 The BIRCH algorithm

The BIRCH algorithm consists of four phases, which are shown in Figure 1.

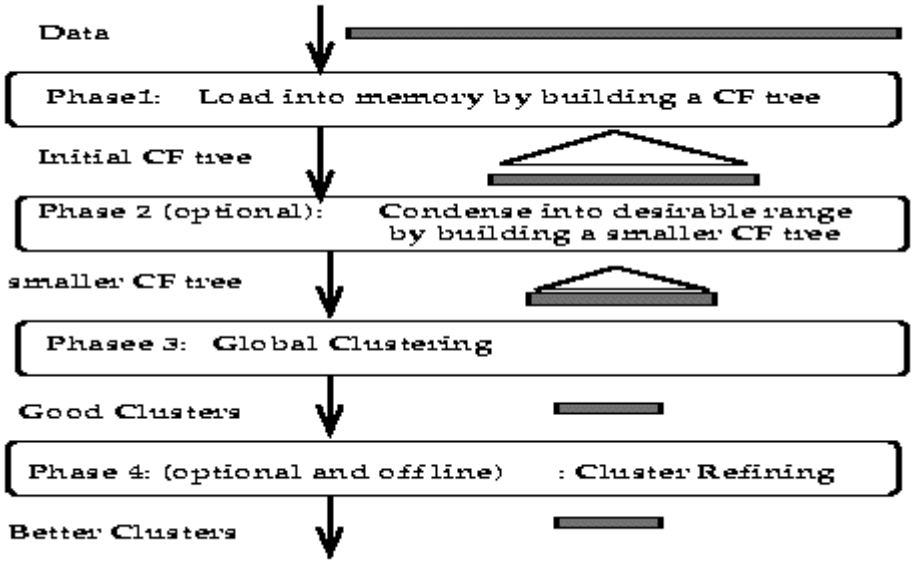


Figure 1: BIRCH overview

Phase 1

This phase initialize the threshold T value, scans all data linearly and insert each point into a CF tree. If it run out of memory before the end of the process, T is increased and the existing leaves are reinserted into the new CF tree. When this is done, the scanning process is resumed from where it was interrupted. An overview of phase 1 is given in the following diagram (Figure 2). The new CF tree is smaller than the previous one. The authors have proven that when rebuilding the tree with an increased value of T, the new tree will be non-larger than the previous tree. This is called the reducibility theorem.

After phase 1, other computations will be faster because no I/O operations will be needed and the clustering will be applied on subclusters and not on single data points.

Phase 2

This phase is optional. Its existence is based on experimentation. It builds a condensed CF tree by removing some outliers (data points that should be regarded as noise) and merging clusters.

Phase 3

This phase applies a global clustering over the leaves of the CF tree. This process takes $O(N^2)$. The reason is that skewed data input order added to the splitting triggered by the page size P leads to a heterogeneous pattern of the clustering data. This step aims to remedy to this problem and provides a better pattern distribution of the data.

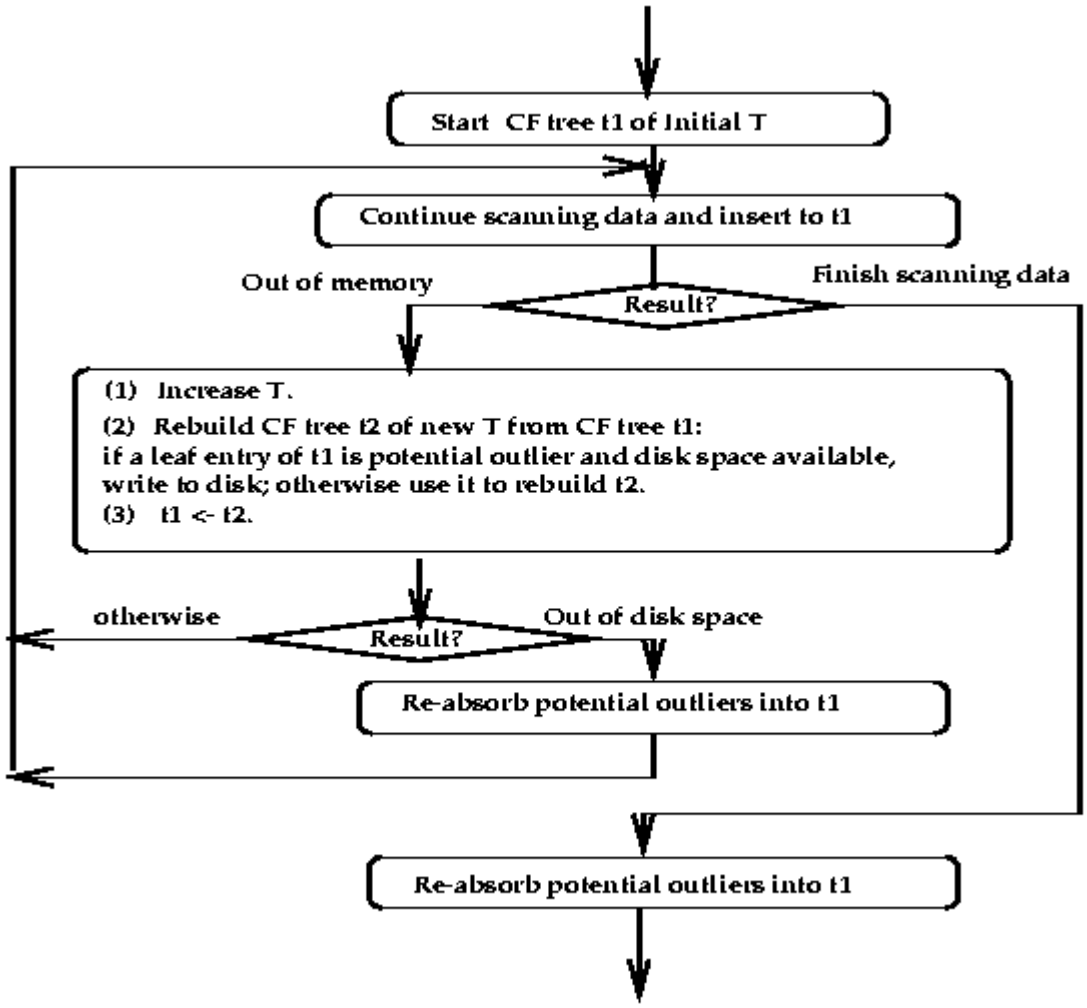


Figure 2: Control Flow of Phase 1

Phase 4

This phase is optional and consists of a further refinement and outliers elimination. It uses the centroids of the clusters outputted by phase 3 as seeds and redistributes the data points to its closest seed to obtain a new set of clusters.

4.4 Some further considerations

Choice of the threshold value T

The choice of the initial threshold value T_0 is very difficult. The goal is to reduce the number of rebuilds. Currently the authors used a heuristic approach so that by increasing T, approximately two leaf entries will be merged into one. Different heuristics provided by the authors are:

- choosing T proportionally to the data seen so far

- increase threshold based on some measure of the cluster volume like the average volume of a cluster or the average volume of a leaf
- make T greater than the distance between the two closest clusters on the most crowded leaf so that this two cluster will be merged
- multiply T by an expansion factor based on the distance between the two closest clusters on the most crowded leaf

Handling of outliers

The leaf entries are considered to be potential outliers if they have far fewer points than the average of the leaf. This is of course a heuristic. The threshold value is first increased thus allowing each leaf entry to ‘absorb’ more points. The potential outliers might also be written out to disk. When a certain disk occupation is reached the data is rescanned. The points that cannot be ‘absorbed’ have great chances to be outliers and can be removed.

An Application

An interesting application of BIRCH is the clustering of colors to characterize an image (filtering). For example it might be used for extracting object in an image. We would obtain satisfactory clusters corresponding to the sky, clouds, or trees.

4.5 Conclusion

BIRCH is a good and fast general clustering algorithm for very large databases. It minimizes the I/O costs with a single scan of the whole data set and the use of preclustering and throws out most of the outliers. However it suffers from a major problem which is how to find a good threshold value T. The heuristics can help but provide no guarantee. Also experiments by others [4] and [5] show that BIRCH works satisfactory only when the clusters can be represented well by separated hyperspheres of similar size and when the clusters are spanned in the whole space.

5 CURE – Clustering Using Representatives

5.1 Introduction

In [4], S. Guha, R. Rastogi and K. Shim present a hierarchical clustering algorithm (named CURE) together with additional features that allow it to be used for large databases. Compared to traditional clustering algorithms, CURE is more robust to outliers and is more successful in identifying clusters having non-spherical shapes (e.g. ellipsoidal) and wide variances in size. CURE achieves this by representing each cluster by a certain fixed number of points that are generated by selecting well scattered points from the cluster and then shrinking them toward the center of the cluster by a specified fraction. Having more than one representative point per cluster permits CURE to adjust better to the geometry of non-spherical shapes while the shrinking helps to dampen the effects of outliers. For use in large databases, CURE employs a combination of random sampling and partitioning. A random sample is selected from the data set, it is partitioned and then partial clusters are located in each of the partitions. After this, the partial clusters are clustered to give the final clusters for the whole space. The authors give experimental results over some example datasets that show that the quality and the speed of CURE is better than that of some well-known clustering algorithms (e.g. BIRCH). They also test its behaviour in relation to the values of various input parameters.

The steps under the CURE clustering scheme are described in the figure below:

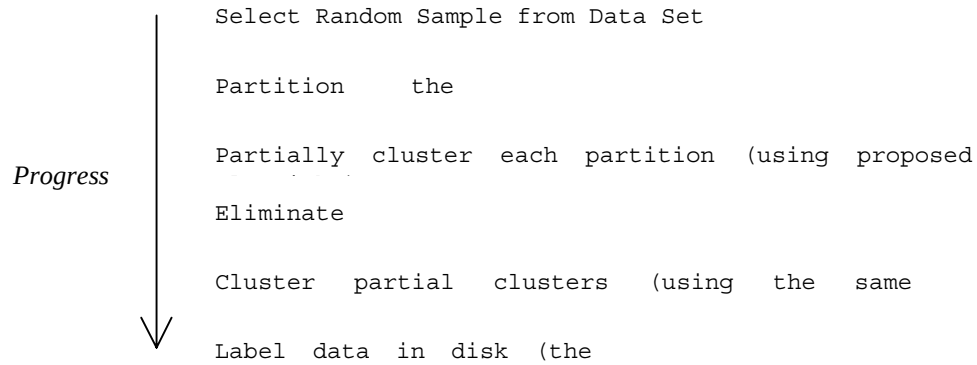


Figure 1: The CURE method

We mention that the elimination of outliers is basically the exclusion of the partial "clusters" which have a very small number of elements while the labeling step involves in the assignment of every element in the dataset to one of the clusters identified by the previous steps. CURE has linear storage requirements and its running time for data with low dimensions is $O(n^2)$ where n is the size of the random sample. This time complexity is typical of most hierarchical clustering algorithms.

5.2 The Algorithm

Problems of traditional clustering algorithms

We refer first to partitional clustering algorithms and then to hierarchical clustering algorithms. Partitional clustering algorithms may present problems when the clusters are close to each other, have different sizes between them or cannot be approximated well by spherical shapes. For example, the commonly used square-error method fails to select as clusters the three cycle disks in Figure 2.

This method tries to minimize the sum of the squares of the distances of every point from the center (mean) of the cluster it is assigned to. Selecting the centers of the three disks as the centers of the clusters results in, in this case, a big square-error as the numerous outer points of the large disk contribute large distance values. So, the method wrongly chooses to split the large cluster.

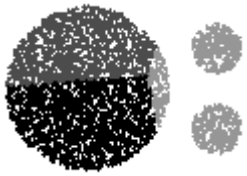


Figure 2: Size problem

Similar observations hold generally for the hierchical clustering algorithms as well. In agglomerative hierchical clustering, we begin with each point being a cluster by its own and then at each step we merge the pair of clusters closest together until only k clusters remain (in a bottom-up fashion). The particular problems of each algorithm depend on the measure used for counting the distance between clusters. Some commonly used measures are:

$$d_{mean}(C_i, C_j) = || m_i - m_j ||$$

$$d_{max}(C_i, C_j) = \max_{p \in Ci, p' \in Cj} || p - p' ||$$

$$d_{min}(C_i, C_j) = \min_{p \in Ci, p' \in Cj} || p - p' ||$$

where m_i, m_j the means of cluster C_i and C_j respectively. Methods using d_{mean} (centroid-based approach) and d_{max} usually make the same mistakes with the square-error method. Methods using d_{min} (all-points approach) can recognize better arbitrarly sized clusters but suffer from the "chaining effect". This means that they tend to consider as one two clusters that are connected by a narrow bridge of just few points.

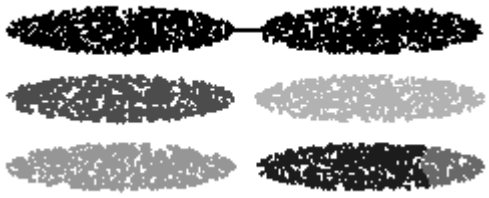


Figure 3: The chaining effect

CURE's approach

CURE employs a new agglomerative hierarchical clustering algorithm that adopts a middle ground between the centroid-based and the all-points extremes. In order to compute the

distance between a pair of clusters, for each cluster, *c* representative points are stored. These are determined by first choosing *c* well scattered points within the cluster, and then shrinking them toward the mean of the cluster by a fraction *a*. The distance between two clusters is then the distance between the closest pair of representative points – one belonging to each of the two clusters. The *c* representative points attempt to capture the physical shape and geometry of the cluster. Furthermore, shrinking the scattered points towards the mean by a factor *a* gets rid of surface abnormalities and mitigates the effects of outliers. The reason for this is that outliers typically will be further away from the cluster center, and as a result, the shrinking would cause outliers to move more towards the center while the remaining representative points would experience minimal shifts. The larger movements in the outliers would reduce their ability to cause the wrong clusters to be merged. Playing with the factor *a* can cause the algorithm to show preference to elongated or more compact clusters, whichever the case.

General Description

The input parameters to the algorithm are the input data set *S* containing *n* points in *d*-dimensional space and the desired number of clusters *k*. Starting with the individual points as individual clusters, at each step the closest pair of clusters is merged to form a new cluster. The process is repeated until there are only *k* remaining clusters. The distance of two clusters for this algorithm is defined as the shortest distance between any pair of representatives (not both of them belonging to the same cluster). Obviously, this kind of distance measure doesn't have metric properties. As for the distance between two points, any of the L_p metrics could be used (manhattan, euclidean etc).

Implementation Details

For every cluster *u* we keep the mean of it (*u.mean*), the set of its *c* representative points (*u.rep*) and the cluster which is closest to it (*u.closest*). The algorithm uses also a heap with the distances of any cluster *u* to its closest cluster (*u.closest*) (the minimum distance on the top) and a *k-d* tree of all the representatives points to help it to keep track of which cluster is closest to which while clusters are being merged between them.

The main loop goes as follows. The closest two clusters *u* and • (top element of the heap) are merged to form a new cluster *w*. The *k-d* tree is updated to contain the correct representatives. Then, the distance of *w* of every cluster *x* is computed to find out the closest cluster to the new cluster (*w.closest*). Also we have to check if the *w* is now the closest cluster to any of the rest clusters (*w* has different representatives than its "parents" *u* and •). We distinguish two cases. If a cluster *x* didn't have as the closest cluster *u* or • then it suffices to check just the distance between *w* and *x* and see if *w* is now closer to *x* than the *x*'s closest cluster. In the case that *x* had as a closest cluster one of *u*, • then the *k-d* tree is searched in order to locate the possibly nearest cluster. The search is limited within distance *dist*(*x*, *w*) as we know that *w* is a potential closest neighbour of *x*. We also mention that the heap is updated when, in any of the above cases, the distance between any two clusters is changed. The same loop continues until we are left with just *k* clusters.

During merging we have to compute the *c* representatives of the new cluster which should be well scattered. This is done as follows. The mean of *w* is computed based on the sizes and the means of the parent clusters. The point of *w* that is furthest from this mean becomes the first representative. The rest representatives are calculated iteratively by choosing the point of *w* that is the furthest from the so far selected representatives. The *c* representatives are then shrunk by a factor *a* towards the mean of *w*. As this procedure examines every point in the

cluster w it is very costly. Another alternative with cost only $O(1)$ is that the representatives be selected with the same procedure but altered to examine only the total $2c$ representatives of the parent clusters. Since the scattered points for w are chosen from the original scattered points for the parent clusters, they can be expected to be fairly well spread out.

Analysis of execution time

The initialization of the heap and the *k-d* tree are bounded by $O(n \log n)$ time. Each operation on the heap (insertion, deletion, value update) takes $O(\log n)$ time. Similarly the expected time of any operation on the *k-d* tree (insertion, deletion, identification of closest point) is $O(\log n)$. The main loop is executed $O(n)$ times. We mention the cost of every operation in each loop:

- The cost of merging two clusters is $O(n)$ (because of the c scans to locate the c well scattered points). If we use the alternative way to select the representatives it is just $O(1)$.
- The cost of updating the representatives in the *k-d* tree is on average $O(\log n)$ (deletion of the old ones, insertions of the new ones)
- The cost of finding the closest cluster to w as described in the paper is $O(n)$. It could be reduced using the *k-d* tree but without any overal difference.
- The cost of finding the new closest cluster for every cluster x and updating correspondingly the heap is more difficult to compute. Let's assume that a cluster can be the closest cluster for at most m clusters. The possible x's that had one of the parents of w (either u or •) as the closest cluster cannot then be more than $2m$. So, the *k-d* tree is probed for at most $2m$ times. Also we have $2m$ updates on the heap. Moreover, for the x's with closest clusters other than w parents, we can find at most m of them that will be closest now to w (that is plus m heap updates). Therefore in the worst case we have a cost of $O(n + m \log n)$ for this operation.

We conclude that in total the algorithm's execution time is $O(n^2 + n\, m \log n)$. It has been shown that the m increases very rapidly when the dimensions augment (becomes comparable to n). In addition, the cost of finding the nearest neighbour in a *k-d* tree has a constant which is at least exponential to the dimension d and which cannot be ignored for higher dimensions. So the above time reduces to $O(n^2)$ only when we have very few dimensions. The running time is also analogous to the number of representatives c used.

5.3 Adaptions for Large Data Sets

The at least quadratic time complexity doesn't permit the algorithm to be directly applied to large datasets. Using though some techniques we can achieve to have fast results for large datasets without wasting the quality of the original algorithm. Some of the techniques used are random sampling, appropriate selection of the sample size, partitioning and outlier elimination.

Random sampling

The idea is to apply CURE's clustering algorithm to a random sample drawn from the data set rather than the entire data set. Typically, the random sample will fit in the main-memory and will be much smaller than the original data set allowing significant reduction in the execution time. Random sampling can also improve the quality of clustering as only a very small

percentage of the outliers is expected to be selected for the input. However, random sampling due to its statistical nature may distort the original image of the clusters in the data set. In practice, this is rarely the case especially when the sample size chosen is large enough.

Sample size considerations

The authors using Chernoff bounds provide an equation giving the size the random sample should have as a function of the probability allowed of missing out an actual cluster. The other parameters in the equation are the size of the smallest cluster wanted, the size of the smallest cluster we want to identify as a percentage of the average cluster size, the minimum number of points from each cluster that we want be present in the sample and finally the number of clusters we want to identify. It turns out that under some logical assumptions the sample size is independent of the total number of points in the data set which suggests that CURE and similar algorithms are scalable. However, for identification of clusters that are close to each other, we should take care that the sample points in a cluster have a high probability of being uniformly distributed. Otherwise some points from different clusters may be found close in relation to the other points in the same cluster and wrongly be merged together. To avoid this we can select a sample size as if we were to find clusters consisting of subclusters of diameter r where r is the distance between the closest pair of clusters. We simply change accordingly the values of some parameters in the equation.

Partitioning

Apart from random sampling, another idea to speed up the algorithm is to partition the sample space into p partitions, each of size n/p. Each partition is partially clustered (independently of the others) until the final number of clusters in each partition reduces to n/pq where q some constant greater to 1. Alternatively, this "preclustering" stage could stop if the distance between the closest pair of clusters to be merged next increases above a certain threshold. Once the n/pq clusters for each partition have been generated, a second clustering pass on the n/q partial clusters for all the partitions (p partitions in total) begins until the number of clusters is reduced to k. An analysis shows that we can achieve an improvement factor $(q-1)/pq + 1/q^2$ over the algorithm without partitioning. Nevertheless, we cannot increase p and q uncontrollably. The number of clusters in each partition is good to be at least 2 or 3 times k in order to avoid "overclustering" the points in each partition during the first pass.

Partitioning can also reduce I/O time in the case the sample set is too large to fit in memory. For the first pass we only have to make sure that each partition fits in memory. For the second pass we can keep in main-memory the representatives points of the clusters instead of all the sample points. Also using the alternative merging method that computes the representatives of a new cluster from those of its parents, we manage not to use at all the non-representative points.

Labeling Data on Disk

Because not all the data are used for the clustering procedure the points that were not used in the sample need to be assigned to the clusters as these where identified in the sample. This is done by labelling each point as belonging to the cluster which has the representative point that is closest to the point examined. Note that other algorithms that use only one point to represent the cluster (centroid) are easy to be mistaken when the clusters have non-spherical shapes, non-uniform sizes and are close to each other.

Handling Outliers

Outliers are a major problem for clustering. CURE tries to cope with them at multiple steps. First random sampling filters out a majority of the outliers and increases the mean distance between them. Taking advantage of the observation that the outliers have generally large distances from other points so the "clusters" they form grow more slowly that the others, the algorithm chooses to eliminate the clusters with a small number of points (e.g. 1 or 2) after some clustering has been done (first phase). A good moment for this is usually when 1/3 of the initial number of clusters remains. In addition as a second phase, the algorithm tries towards the end of the algorithm to eliminate any clusters with small numbers of points. Usually these are outliers that got sampled in close vicinity so the first phase couldn't exclude them. If such small groups remain then they can invoke an error to be made that in most cases will totally disturb the final result.

5.4 Experimental Results

The performance of CURE was compared with that of BIRCH (preclustering and then centroid-based hierarchical clustering) and MST (an all-points hierarchical clustering with d_{min} as measure). BIRCH can recognize well spherical shapes with similar size while MST is better at clustering arbitrary shapes. It is, though, sensitive to outliers. CURE was used in the tests with all its adaptions for large datasets. It succeeded in all four 2-dimensional datasets in contrast to the other two algorithms. Some regulations in the parameters of the algorithm were necessary before. For example, when a approached zero the quality was similar to the MSP and when a approached one to BIRCH. For values in between (not extermal) there was a good balance. Also with sample sizes smaller than 2% of the size of data set the results were poor. In addition, the number of partitions p and the degree to be partially clustered q have had not to be very high.

The execution time of CURE in relation to BIRCH was in all cases tested better. The difference was much bigger when partitioning was used or for larger values of datasets as the sample size selected by CURE remained roughly the same and so was its running time.

5.5 Conclusions - Drawbacks

CURE clustering scheme provides a wealth of ideas that improve both quality and speed of the clustering. There are still shapes of clusters that will fail to recognize but these are only a minority of the most commonly met cluster geometries. However the agglomerative clustering approach and the use of *k-d* tree in order to locate the closest cluster to every cluster is expected to damage severely its performance even in moderate dimensions. No tests were presented to disprove this estimation. There is also the additional factor c in the execution time (not present, for example, in BIRCH) that will count especially in the labeling phase. Another problem common in most clustering algorithms is that its behaviour is dependent on parameters whose "good" values are difficult to know (e.g. the number of clusters to report). Nevertheless the tests show that CURE has a good stability at least for many of them. For the scalability of CURE although the theoretical analysis is reasonable the experiments committed were not enough to establish the maintenance of quality in real datasets.

Finally, one method we propose to improve the speed of the algorithm is to use a variable number of representatives for every cluster according to an estimation of its geometrical complexity.

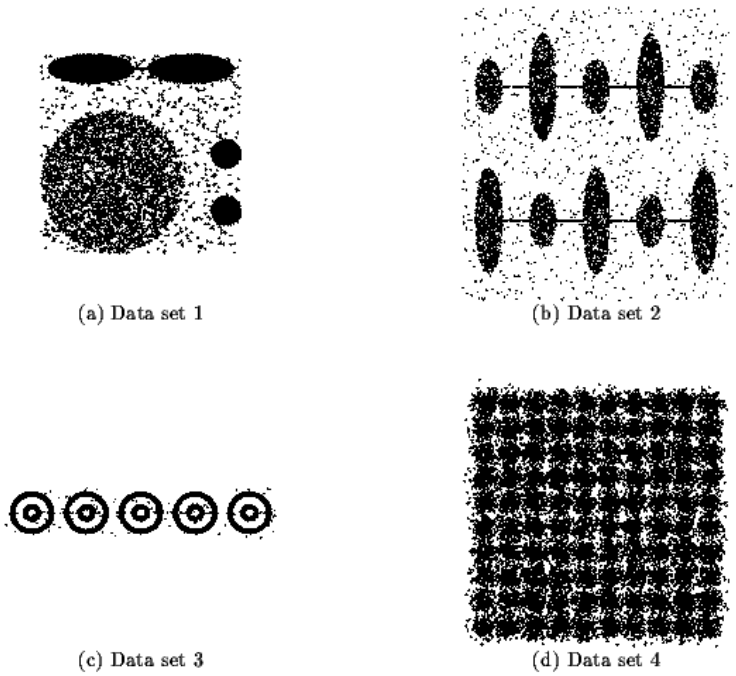


Figure 4: Datasets used for quality comparison

6 CLIQUE

6.1 Niche for CLIQUE

All previous algorithms have a common shortcoming, which is that given a database with n-dimensions, then with these algorithms you can only find the clusters in n-dimension. They are not effective in identifying clusters that exist in the subspaces of the original data space.

While with the development of data mining, special requirements have been put on clustering algorithms. Especially, now the database is in the trend of being high dimensional. An object (data record) typically has dozens of attributes and the domain for each attribute can be large. It is not meaningful to look for clusters in such a high dimensional space as the average density of points anywhere in the data space is likely to be quite low. Compounding this problem, many dimensions or combinations of dimensions can have noise or values that are uniformly distributed. So we need a proper clustering algorithm especially for these high dimensional database, which can find the clusters in the subspaces of the original space.

This problem is often tackled by two methods. The most common one is to require the user to specify the subspace (a subset of the dimensions) for cluster analysis. But, it's not a good way because maybe the subspaces guessed by the user have no interesting clusters at all. The second method is to apply a dimensionality reduction algorithm to the database. The existing techniques transform the original data space into a lower dimensional space by forming dimensions that are linear combinations of given attributes. However, although these techniques may succeed in reducing the dimensionality, they still have some drawbacks. First, the new dimension generated by linear combination is hard to interpret, making it hard to understand clusters in relation to the original data space. Second, these techniques are not effective in identifying clusters that may exist in subspaces of the original data space.

In the paper, Automatic Subspace clustering of High Dimensional Data for Data Mining Application [5], the authors propose a new technique called Clique, which eliminates the above problems.

6.2 Algorithm

CLIQUE, consists of the following 3 steps:

1. Identification of subspaces that contain clusters.
2. Identification of clusters.
3. Generation of minimal description for the clusters.

In this algorithm, there are two input parameters:

1. ξ: the number of intervals in every dimension for partition.
   (the units are obtained by partitioning every dimension into ξ intervals. And if different dimensions have different domains, then the length of intervals in different dimensions would be different too.)
2. τ: the threshold of density in each unit.
   (if in one unit, the density is greater than τ, then this unit is a dense unit.)

Identification of subspaces that contain clusters.

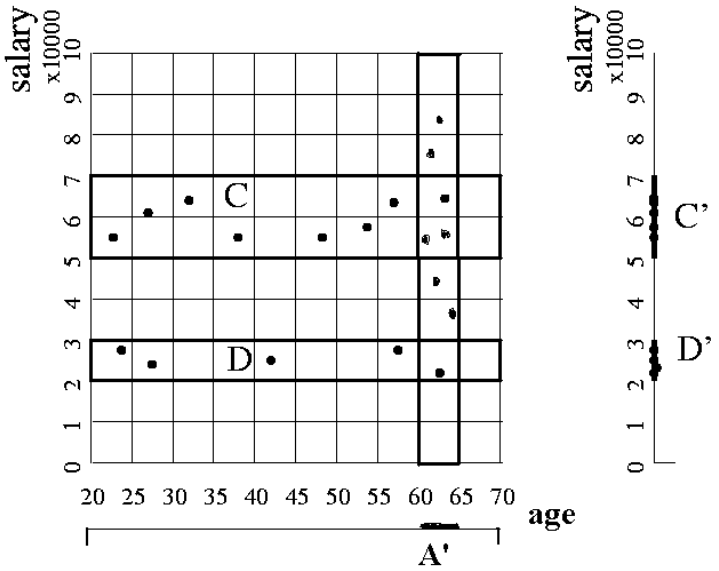
The tricky part in identifying subspaces that contain clusters lies in finding dense units in different subspaces.

- ***A bottom-up algorithm to find dense units***

We use a bottom-up algorithm that exploits the monotonicity of the clustering criterion with respect to prune the search space.

Lemma Monotonicity: If a collection of points S is a cluster in a k-dimensional space, then S is also part of a cluster in any (k-1)-dimensional projections of this space.

Algorithm: The algorithm proceeds level by level. It first determines 1-dimensional dense units by projecting the values in every dimension and scanning all units in all the dimensions. If the density of the unit is greater than the input parameter τ, then this unit is considered dense, and should be chosen. Having determined (k-1)-dimensional dense units, then from all the (k-1)-dimensional subspaces containing the dense units, every time choose 2 subspaces on the condition that the 2 subspaces share k-2 dimensions. Intersect the dense units in the 2 different (k-1)-dimensional subspaces by projecting over the common k-2 dimensions. In this way, we can find the candidate k-dimensional units. After that, the algorithm scans over the candidate units, to find those units that are really dense on k-dimensional space. Here we see



an example.

Figure 1: Identification of clusters in subspaces (projections)

of the original data space

In figure 1, in 1-dimension subspace, we find cluster A' on age axis, and two clusters C', D' on salary axis. A'=60<age<65, C'=5<salary<7, D'=2<salary<3. Now we go one dimension higher, what we need to do is to intersect A' and C', intersect A' and D'.

Time complexity: If a dense unit exists in k dimensions, then all of its projections in a subset of the k dimensions, that is, $O(2^k)$ different combinations are also dense. The algorithm makes k passes over the database. It follows that the running time is $O(c^k + mk)$ for a constant c.

- Making the bottom-up algorithm faster

Getting all the promising subspaces of k-dimension, we need to choose 2 of them, intersect the units in the two different subspaces, and then get the (k+1)-dimensional subspace candidate. Every time to choose 2 from such as N, is a combination, if N is very great, then the cost would be quite expensive.

To speed up the algorithm, in every step, having calculated the k-dimensional subspaces, before we transfer from k-dimensions to (k+1)-dimensions, we need to prune the pool of candidate dense units in order to eliminate the false hits

Assume we have the subspaces, $S_1, S_2, ...S_n$. Our pruning technique first groups together the dense units that lie in the same subspace. Then, for each subspace, it computes the fraction of the database that is covered by the dense units in it. Here, we refer to the number of points in the subspace as the coverage of subspace S_j.

We sort the subspaces in the descending order of their coverage. As we all know, we reduce the cost on the price that we may lose some important information about the database. So where to prune is a tradeoff between cost and information. Subspaces with large coverage are selected and the rest are pruned.

Finding clusters

After identifying the subspaces containing dense units, CLIQUE calculates the clusters that are formed. The problem is equivalent to finding connected components in a graph defined as follows: Graph vertices correspond to dense units, and there is an edge between two vertices if and only if the corresponding dense units have a common face.

Units corresponding to vertices in the same connected component of the graph are connected because there is a path of units that have a common face between them, therefore they are in the same cluster. On the other hand, units corresponding to vertices in different components cannot be connected, and therefore cannot be in the same cluster. We use a depth-first search algorithm to find the connected components of the graph.

Time complexity: If the total number of dense units in the subspace is n, the total number of data structure accesses is 2kn.

Generating minimal cluster description

The input to this step consists of disjoint sets of connected k-dimensional units in the same subspace. Each such set is a cluster and the goal is to generate a concise description for it. To generate a minimal description of each cluster, we would want to cover all the units comprising the cluster with the minimum number of regions such that all regions contain only connected units, then use DNF to describe the regions. For a cluster C in a k-dimensional

subspace S, a set Ŕ of regions in the same subspace S is a cover of C if every region R ∈ Ŕ is contained in at least one of the regions in Ŕ. Computing the optimal cover is known to be NP-hard, even in the 2-dimensional case. The optimal cover is the cover with the minimal number of rectangles.

CLIQUE uses a heuristic algorithm consisting of two steps. It greedily covers the cluster by a number of maximal rectangles (regions), and then discards the redundant rectangles to generate a minimal cover.

- Covering with maximal region

The input to this step is a set C of connected dense units in the same k-dimensional space S. The output will be a set Ŕ of maximal regions such that Ŕ is a cover of C. We present a greedy growth algorithm for this task.

Greedy growth: We begin with an arbitrary dense unit u1 ∈ C and greedily grow (as described below) a maximal region R1 that cover u1. We add R1 to Ŕ. Then we find another u2 ∈ C that is not yet covered by any of the maximal regions in Ŕ. We greedily grow a maximal region R2 that covers u2. We repeat this procedure until all units in C are covered by some maximal region in Ŕ.

To obtain a maximal region covering a dense unit, we start with u and grow it along dimension a1, both to the left and to the right of the unit. We grow u as much as possible in both directions, using connected dense units contained in C. The result is a rectangular region. We now grow this region along dimension a2, both to the left and to the right of the region. We again use only connected dense units from C, obtaining a possibly bigger rectangular region. This procedure is repeated for all the dimensions, yielding a maximal region covering u. The order in which dimensions are considered for growing a dense unit is randomly determined.

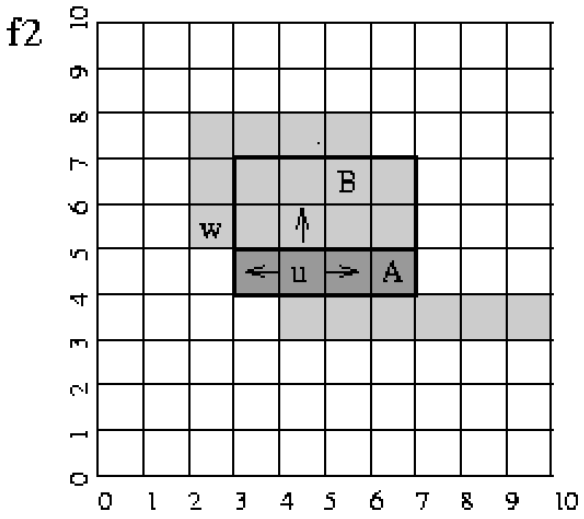


Figure 2: Illustration of the greedy growth algorithm

Figure 2 illustrates how the algorithm works. Here the dense units appear shaded. Starting from the dense unit u, first we grow along the horizontal dimension, finding rectangle A consisting of four dense units. Then A is extended in the vertical dimension. When it cannot be extended further, a maximal rectangle is obtained, in this case B. The next step is to find another maximal region starting from a dense unit not covered by B, for example w.

Time complexity: the greedy growth algorithm performs a total of $O(n^2)$ dense unit accesses. This bound is almost tight.

- Minimal Cover

The last step of CLIQUE takes as input a cover for each cluster and finds a minimal cover. Minimality is defined in terms of the number of maximal regions (rectangles) required to cover the cluster.

We propose the following greedy heuristic:

Removal heuristic: Remove from the cover the smallest (in number of units) maximal region which is redundant (i.e. every unit is also contained in some other maximal region). Break ties arbitrarily. Repeat the procedure until no maximal region can be removed.

Time complexity: The cost of sorting the regions is O(n log n). The scan requires |Ri| dense unit accesses for each region R_i. The total number of accesses for all regions is then $\sum | Ri | = O(n^2)$.

6.3 Performance Analysis

In the following, we will present the scalability of CLIQUE with the number of data records and the dimensionality of the data space respectively.

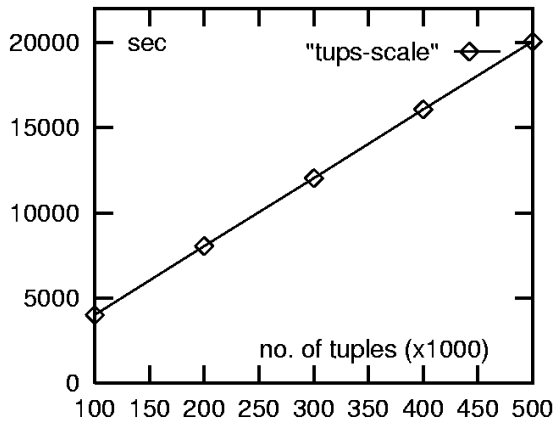


Figure 3: Scalability with the number of data records

Database size: Figure 3 shows the scalability as the size of the database is increased from 100,000 to 500,000 records. The data space had 50 dimensions and there were 5 clusters, each in a different 5-dimensional subspaces, and τ was set to 0.5%.

As expected, the running times scales linearly with the size of the database because of the number of dimensions in the database does not change.

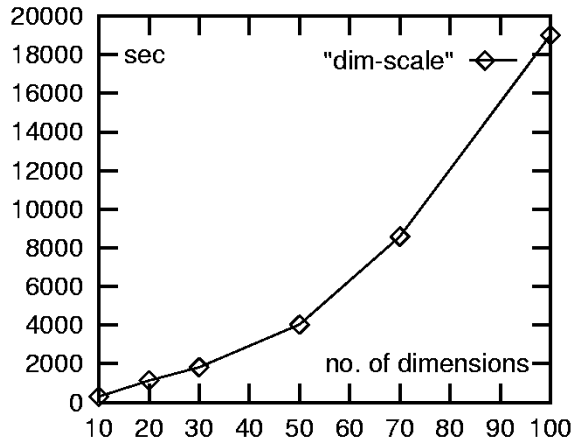


Figure 4: Scalability with the dimensionality of the data space

Dimensionality of the data space: Figure 4 shows the scalability as the dimensionality of the data space is increased from 10 to 100. The database had 100,000 records and there where 5 clusters, each in a different 5-dimensional subspace, and τ was set to 0.5%. Here, the figure seems not bad, but in fact, this is because we designed the data and forced the algorithm to stop when the dimension reaches 5. If we don't have any restriction, the cost would increase very quickly with the augment of dimensionality.

6.4 Conclusion

We introduced the problem of automatic subspace clustering, motivated by the needs of emerging data mining applications. The solution we propose, CLIQUE, has been designed to find clusters embedded in subspaces of high dimensional data without requiring the user to guess subspaces that might have interesting clusters. CLIQUE generates cluster descriptions in the form of DNF expressions that are minimized for ease of comprehension.

7 Projected Clusters in High Dimensional Spaces

There have been numerous researches done on the clustering problem, whose applications are many (e.g. in similarity search, customer segmentation, pattern recognition, trend analysis and classification). The problem of clustering data can be defined as follows: Given a set of points in multidimensional space, find a partition of the points into clusters so that the points within each cluster are close to each other. The word "partition", however, seems to have caused some dispute over different clustering algorithms. Some researchers' opinions are that a partition over a set S of points should be a set of clusters that are disjoint with each other. That is, any point in S should appear once and exactly once in all the clusters. In other words, the clusters shouldn't overlap. However, others hold different opinions and they think clusters with moderate overlaps should also be counted in as a clustering solution. Actually, in the applications which exist nowadays, both of the above two clustering algorithms find their respective positions. A supermarket classification, for example, will most probably need the first type of clustering if the manager demands for a list of categories in which no customer can belong to two of them. Data mining in sales records will prefer the second clustering algorithm as it will provide much more interesting information if the algorithm can take into account different combinations of the products. PROCLUS [6], the algorithm that we are going to introduce, will give another solution to divide the original set of points into disjoint clusters.

Besides the conception of partition, there is another interesting problem that finds its way in the clustering algorithms. Most traditional clustering algorithms will give a result of clusters that are built on the same dimensions (possible a subset of all the original dimensions). For example, let us assume the original data space has 3 dimensions x, y, and z. These algorithms may end up with several clusters, which are built upon the same sub-dimensions, say x and y. This, however, can sometimes lead to the inability to explore some useful clustering information. Let's take a look at the following set of 6 points, which reside in a data space with two dimensions x and y.

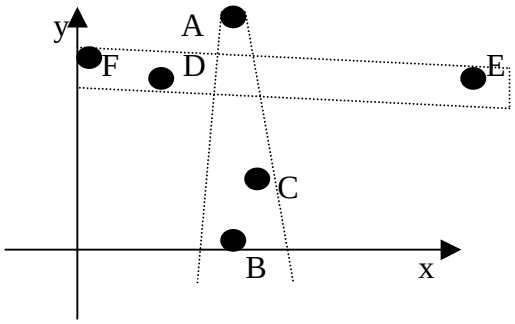


Figure 1: a set of points on a 2-dimension data space

Judging from the set, we can find that points A, B and C are clustered well along x dimension, while D, E and F are clustered well along y dimension. Sometimes user will wish to find both these 2 clusters, however, this is out of the scope of the traditional algorithms as they can only find those clusters that are built upon the same sub-dimensions. That is, if they find the cluster built along x dimension, they will have to ignore the one containing D, E and F. This problem has been addressed for the first time in the algorithm called CLIQUE, which works from lower to higher dimensionality subspaces and discovers "dense" regions in each subspace (for detail about this algorithm, please refer to the previous section). PROCLUS

will also address the problem, and will find clusters with different sub-dimensions. That is, if it is applied to the data set illustrated in figure 1, both clusters will be found, and the sub-dimensions for the cluster containing A, B and C is dimension x, while the sub-dimension for the other is dimension y. In the following paper, the author names *projected cluster* a cluster together with a subset of dimensions, upon which the cluster is built. This is also how the name of this algorithm comes into being, that is, the PROjected CLUSters.

In the following sub-sections, we will first introduce the outline of this algorithm before we look into the detail.

7.1 The idea behind PROCLUS

The problem of finding projected clusters is two-fold: to form groups (clusters) of points that are close to each other within each group, and to find the appropriate set of dimensions for each cluster such that the points in the cluster are close to each other along these dimensions. This will arouse two issues: how to find the points in each group and how to decide the suitable dimensions for each cluster. The idea of these two issues will be presented in this sub-section.

Suppose we are going to find K clusters among the original set of points, the idea is to start with K starting points, called *medoids* , which are chosen from the original set of points by following some requirements. One obvious requirement is that these K points should be sufficiently distant from each other to make it less possible for them to fall in each other's cluster. Having selected the K medoids, PROCLUS will proceed to build K clusters based on these medoids. The rationale of doing this is the locality principle. That is, a point will tend to fall into the cluster whose medoid is the closest to it. After we have applied this method to every point will we have divided the points into K "groups". How to decide the sub-dimensions for each cluster is less straightforward. Suppose we are going to decide the sub-dimensions for a cluster with p points. In order to illustrate more clearly, we assume the number of the dimensions in the whole data space is |D|. Our problem is to find a subset of these |D| dimensions for this cluster. Remember a cluster is a group of points that are close to each other along every dimension selected. Therefore, if the dimension Dj is selected, it means all the points in this cluster should be close to each other along the dimension Dj. Since we start from the medoids, which are the starting points, we can think instead that all the points are close to the medoid of this cluster along dimension Dj. Given this, we will decide the sub-dimensions in the following way. First compute the average distance from each point to the medoid along each dimension. Thus we will have |D| values h1,h2,..,h|D|. What we do next is to pick out those values that are small according to some criteria. If hj is picked, then we will add dimension j into the sub-dimensions of this cluster. The sub-dimensions picked out this way will be considered to be the space where the points in this cluster will be best clustered.

Now we will move to introduce PROCLUS more explicitly.

7.2 The Algorithm

Input and Output

The input to this algorithm is the number K of how many clusters a user wants to find, as well as the average cardinality L of all the dimensions built on all of these clusters. Suppose the cardinalities of each cluster is C1,C2,…,Ck, then

$$L = (C_1 + C_2 + ... + C_K) / K$$

There is no restriction on K and L except that KL should be an integer because C1+C2+…+Ck is apparently an integer.

The output of this algorithm is K clusters of points, as well as the specific dimensions for each cluster, and the cardinalities of the dimensions satisfy the above formula.

The whole algorithm is split into 3 phases, that is, the Initialization Phase, the Iterative Phase, and the Refinement Phase. The output of the Initialization Phase is a set of points that serve as the candidate medoids. The size of the set is larger than K to allow more combinations of trial medoids. The Iterative Phase will aim at deciding the specific dimensions for each cluster. Finally, the Refinement Phase will try to improve the quality of the clusters. Also, the *outliers*, which are the points that are scattered so sparsely that they fail falling in any of the clusters, are also handled in this phase.

Initialization Phase

As said before, this phase will yield a set of points which are the possible candidate medoids for the next phase. Since a medoid will be the surrogate center for a cluster, one of the requirements for choosing medoids will be that they are sufficiently far from each other. The reason for this is straightforward. If two medoids are close, they are likely to be in the same cluster rather than be in different ones. On the other hand, if we based on our selection on the distance calculation, it is most likely that we will pick out some outliers as our medoids because most outliers will be far away from the other points. Since there is no way for us to know at the initialization phase if a point is an outlier, what we can do is to try with some more combinations of the candidate medoids, and get the one which gives the best clustering performance. That is the reason why we will choose a little more than K medoids. Actually, the number we choose is proportional to K, as you will see below, BK, where B is a constant. The following is how we pick the BK points based on a greedy algorithm.

```
Greedy(set of points, Number of medoids: k)
{d(…,…) is some distance function}
begin
M={m1} {m1 is a random point of S}
{Compute distance between each point and medoid m1}
for each x in S-M
  dist(x)=d(x,mi)
for i=2 to k
begin
  {choose medoid mi to be as far away as possible from previous medoids}
  let mi in S-M be s.t. dist(mi)=max{dist(x)|x in S-M}
    M = M ∪ {m_i}
```

```
  {compute distance between each point and its closest medoid}
  for each x in S-M
     dist(x)=min{dist(x),d(x,mi)}
end
return M
end
```

Iterative Phase

The output of the last phase is a set V with BK points that are the candidate medoids. In this phase, we are going to decide K points in V to be the medoids and construct the clusters upon them. Since the whole combination of picking out K points from the BK points will be large and thus impossible to enumerate in real application, PROCLUS adopts the idea of neighboring algorithm. That is, the technique can be viewed as a restricted search on the complete graph with vertex set M. The current node in the graph represents the best set of medoids found so far. The algorithm tries to advance to another node in the graph as follows: It first determines the bad medoids in Mbest using a criterion which we will discuss later, and then replaces the bad medoids with random points from set V to obtain another vertex in the graph. If the clustering determined by the new vertex is better than the old one, then we let the new one be the present vertex Mbest. Otherwise, it sets the new vertex to a new one by replacing with the bad medoids with some new random points from the BK points. The above process continues for some certain times or within some certain time limit before it gives out the result. Obviously, PROCLUS can't guarantee the best solution, but the solution it returns is a good solution. We also mention that the first vertex that PROCLUS starts with is randomly generated from the set V.

When considering a vertex, PROCLUS tries to build clusters upon these K points. It should be pointed out that the points in each cluster are temporarily assigned in this phase and that these points are utilized only to compute the specific dimensions for each cluster. The real assignment of points will be done in the Refinement Phase, which will be discussed just after this phase. Our introduction will proceed in two steps. First we will talk about how PROCLUS assigns the points in the original set to form K clusters based on the medoids having been decided. Then we refer to the technique PROCLUS decides the specific dimensions for each cluster.

Given a set of k medoids M={m1,m2,…,mk}, PROCLUS evaluates the locality of the space near them in order to find the dimensions that matter most. More exactly, for each medoids mi let qi be the minimum distance from any other medoid to mi, i.e.

$$q_i = \min_{j \neq i} d(m_i, m_j)$$

If a point is within the distance qi from mi, then we add qi to Li, which is the set of points that are within the sphere with the center being mi and radius being qi. We check every point in the original data set like the above, and thus get the K sets L1,L2,…Lk. Note that these k sets aren't necessarily disjoint with each other, and nor are these k sets necessarily cover the whole data set.

Now, PROCLUS proceeds to determine the specific dimensions for each cluster. Remember that the total number of dimensions for all the clusters is KL. Based on the rationale

$$Y_i = (\sum_{j=1}^{d} X_{i,j}) / d$$

$$\sigma_i = \sqrt{(\sum_{j} (X_{i,j} - Y_i)^2) / (d-1)}$$

$$Z_{i,j} = \frac{X_{i,j} - Y_i}{\sigma_i}$$

introduced in subsection 7.1, PROCLUS selects the dimensions for the clusters as follows. We first compute the average distance along each dimension from the points in Li to mi. Let Xi,j denote this average distance along dimension j. To each medoid mi, we wish to associate those dimensions j for which the values Xi,j are as small as possible relative to statistical expectation, subject to the restriction that the total number of dimensions associated to medoids must be equal to KL. In order to avoid the dilemma that there are too few dimensions in some clusters, PROCLUS adds the restriction that there must be at least 2 dimensions. After computing all the Xi,j (i=1,…,K and j=1,…,|D|), PROCLUS makes the selection by the following statistical evaluation:

A negative value of Zi,j indicates that along dimension j the points in Li are more closely correlated to the medoid mi. PROCLUS first sorts all the Zi,j values in increasing order, then chooses the 2 smallest for each I(for a total of 2k values), and then greedily pick the remaining lowest K(L-2) values. With each medoid I we associate those dimensions j whose corresponding Zi,j value was chosen by the above algorithm. In the end, when we have picked all the KL values, we will have at the same time determined the specific dimensions for each cluster.

Here there is one thing that might lead to puzzlement. Since the points in each cluster in this phase are not the final assignment, how is it possible to compute the dimensions that utilize these points? Actually, though these points might not be the final points in each cluster, they must be at least the candidate points in the final assignment. In other words, a majority of these points will officially fall into the same cluster because they are close to the medoid (recall how we get the locality with a sphere for each medoid). Therefore, though the computation of the dimensions is not absolutely accurate, it's still a good one.

As we mentioned earlier, PROCLUS will compare the current solution with the one that has been considered the best so far. The evaluation of a solution in PROCLUS in done by computing the total average distance of points in different clusters to the centroid of Ci along each specific dimension j (a centroid of a cluster C is given by the following formula)

$$x_c = \sum_{i=1}^{t} x_i / t$$

Now it's time for PROCLUS to replace some bad medoids with some new random points, and commit the above computation again. PROCLUS chooses the bad medoids by considering the following 2 principles. First, the medoid of the cluster with the least number of points is bad. Second, the medoid of any cluster with less than (N/|k|)minDeviation points

is bad, where N is the total number of points and minDeviation is an adjustable constant in the range (0,1).

Refinement Phase

The output of the previous phase is K clusters {C1, C2,…,Ck}, as well as their respective specific dimensions. PROCLUS will proceed with two jobs. The first of which is to refine the dimensions by first discarding all of them and compute them again in a slight different way. This job is not of too much interest, so we will discuss no more here. The second job is to finally assign each point to the medoids relative to their settled dimensions. In order to do this, we need a function that can compute the distance between 2 points in a space described by a subset of dimensions of the original all dimensions. This function should have the power to minimize the effect of different number of dimensions. Because of this, PROCLUS computes the distance with the following function:

$$d_D(x_1,x_2)=(\sum_{i\in D}|x_{1,i}-x_{2,i}|)/|D|$$

(The D in the above function is a set of dimensions)

The author calls the distance computed by the above formula the *projected Manhattan segmental* distance. For each point in the whole data set, PROCLUS computes the projected Manhattan segmental distance for all the clusters, and assigns the point to the cluster with the smallest projected Manhattan segmental distance.

7.3 Conclusion

PROCLUS is good at discovering interesting patterns in subspaces of high dimensional data spaces. This is a generalization of feature selection, in that it allows the selection of different sets of dimensions for different subsets of the data. Projected clustering is general enough to allow us to deal with different correlations among various subsets of the input points.

PROCLUS guarantees a partition of the original set of data. To our understanding, this has its advantages and drawbacks. This feature will certainly make PROCLUS much more efficient than traditional algorithms in applications that emphasize a partition. However, there are also applications that require certain data to fall in more than one cluster and PROCLUS might be totally unable to achieve satisfactory results in this case. Therefore, we can conclude that PROCLUS will take the lead in partition-required applications while CLIQUE will outperform in others.

Apart from the inefficiency in finding non-partition clusters, PROCLUS still has the following drawback. As mentioned in the early part, the PROCLUS will take as input the average number of dimensions of all the clusters, which will determine, together with the other input, the number of clusters, the total number of dimensions from all the cluster. Since it is difficult for a user to decide the optimum parameters, PROCLUS may fail to return a good solution even in the partitioning-required applications. For example, for the data set in Figure 1, if the parameter inputted by the user is 2 for number of clusters, and 3 for the average dimension, PROCLUS would perform poorly.

Final Remarks

From our survey we can conclude that when the size of the dataset is small to medium and its dimensionality is low then there are a sufficient number of algorithms that achieve good and fast clustering. Nevertheless when the size of the dataset is very large, it has many dimensions and a high level of noise then no good and fast clustering algorithm exists. Lately, there are some algorithms (PROCLUS, CLIQUE) that try to issue these problems with promising ideas and results, but still a general solution is far away. A solution is further hardened by the fact that most clustering algorithms are sensitive to their input parameters. For different datasets different input parameter settings will give a satisfactory result. Finding the correct ones is not at all an easy task.

References

[1] Ester M., Kriegel H-P., Sander J., Xu X.: A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise, *Proc. 2nd Int. Conf. on Knowledge Discovery and Data Mining (KDD'96), Portland, OR, 1996,* pp. 226-231.

[2] Ankerst M., Breunig M. M., Kriegel H-P., Sander J.: OPTICS: Ordering Points To Identify the Clustering Structure, *Proc. ACM SIGMOD '99.*

[3] Tian Zhang, Raghu Ramakrishnan, Miron Livny: BIRCH: An Efficient Data Clustering Method for Very Large Databases, *SIGMOD Conf. 1996:* 103-114.

[4] Sudipto Guha, Rajeev Rastogi, Kyuseok Shim: CURE: An Efficient Clustering Algorithm for Large Databases, *SIGMOD Conference 1998:* 73-84.

[5] Rakesh Agrawal, Johannes Gehrke, Dimitrios Gunopulos, Prabhakar Raghavan: Automatic Subspace Clustering of High Dimensional Data for Data Mining Applications, *Proc. ACM SIGMOD '99.*

[6] Charu C. Aggarwal, Cecilia Procopiuc, Joel L. Wolf, Philip S. Yu, Jong Soo Park: Fast Algorithms for Projected Clustering, *Proc. ACM SIGMOD '99.*