Fuzzy Clustering for Content-based Indexing in Multimedia Database

By

Ho-Yin YUE

Supervised By

Prof. Kwong-Sak LEUNG & Prof. Irwin Kuo-Chin KING

A Thesis Submitted in Partial Fulfillment
of the Requirements for the Degree of
Master of Philosophy
in
Computer Science and Engineering

June, 2001

Fuzzy Clustering for Content-based Indexing in Multimedia Database

submitted by

Ho-Yin YUE

for the degree of Master of Philosophy

at the Chinese University of Hong Kong

Abstract

In this information age, how to manage information is one of the important issues in our daily life. In a content-based retrieval database, contents or features of the database objects are used for retrieval. Typically, these data exist in natural clusters. However, many of the currently indexing methods omit this data clusters information in the construction of the indexing structure which leads to performance degradation.

To improve the retrieval performance, we (1) use *Sequential Fuzzy Competitive Clustering (SFCC)*, a fast and noise resistant fuzzy clustering algorithm, to obtain the natural clusters information and (2) use the result of SFCC clustering to construct a good indexing structure (SFCC-b-tree) for effective nearest-neighbor search. SFCC-b-tree uses a hierarchical clustering approach to transform the feature space into a sequence of nested clusters. These nested clusters are then further converted into an indexing tree for data retrieval.

Our experimental results show that: (1) SFCC is faster than other tested clustering methods to locate natural clusters for indexing. (2) FCC and SFCC are more accurate and noise resistance than other tested clustering methods. (3) SFCC-b-tree is efficient for 100% nearest-neighbor search and it is faster

than other indexing methods in both building time and searching time. Moreover, we prove worked out a efficiency formula for SFCC-b-tree. We can make use of it to predict the searching efficiency of SFCC-b-tree and compare it with other indexing methods for a given set of parameters.

To let people stop moving forward is not "depression"; but "give-up". To let people keep going is not "hope"; is "mind".

Violet, ARMS Vol. 15

Acknowledgment

I would like to express my gratitude to my supervisor, Professor Kwong-Sak LEUNG and Professor Irwin Kuo-Chin KING for their academic guidance, critiques of my ideas, emotional support, and encouragement. My research could not have been finished reasonably without insightful advice from them.

Moreover, I would like to sincerely thank my examining committee, Professor Kin-Hong LEE, and Kin-Hong WONG for their comments and useful suggestions on this thesis.

Thanks to my friend T. K. Lau for his support on the background knowledge. Without his support, I may still know nothing in this filed.

Thanks to all the other friends I have made at the Chinese University of Hong Kong for making my stay at CUHK an enjoyable period of time.

Finally, I am deeply grateful to my family, especially my parents and brother for their love, support, and patience during the past two years. This thesis would not have been possible without their support and confidence.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this information age, people need to manipulate a lot of multimedia data objects, such as images, sounds, articles, and videos in their daily life. However, as usually, when the size of database is huge, people find it impossible to index the database by human and need to develop some automatic methods for indexing and retrieving the multimedia data objects from the database.

In a traditional database, although people use keywords or text descriptors for indexing and retrieval, but they are usually ineffective and imprecise. It even poses difficulties for the end users especially for those without special training. The main difficulties are:

1. **Lack of Standards:** Different users may use different words to describe a same multimedia data object for retrieval.

2. **Lack of Descriptive Power:** Even when standardized vocabulary is used, it is still hard to depict the object clearly and precisely.

3. **Lack of Automatic Keywords Extraction Methods:** There is no efficient keywords extraction algorithm to extract meaningful keywords from multimedia data objects.

Image database are those databases that store image as their data. It is a special kind of multimedia database. Here, we use this to demonstrate the

above difficulties and introduce some alternative to deal with these difficulties.

Assume that, we want to search an image with an old person standing at the right hand side of the image. In this case, we may use *old people* or *elderly people* as the keyword for this retrieval. This shows that, even for a very simple query, different users may use different keywords for a same query. Even for the same user, it is likely to happen that he may use different keywords at different times.

Even when only the keyword *elderly people* is used. How to define the *fuzzy term*, right hand side, is still a problem we need to face. Because different person may have different definitions on the adjective (or Fuzzy Term) *right*. So, it is hard to define the meaning of *right* or other fuzzy terms in a common standard.

Furthermore, it is hard to extract those keywords from a multimedia database automatically. For the above example, there is no efficient and automatic methods to extract the high-level keywords "elderly people" and "old people" from the image. As a result, many image database systems that use keywords as query in retrieval still need human to extract these keywords. To improve the efficiency and accuracy, we need a new kind of database which is especially designed for multimedia data indexing and retrieval.

Rather than using keywords in queries, databases that support content-based retrieval use the content in the multimedia object itself as the queries for retrieval. These contents (or features) may be color, texture, sketch and shape for image databases. On the other hand, it can be frequency range, sound quality and intensity for sound databases. In a typical content-based retrieval task, objects with features similar to the query will be the retrieval

results.

For example, we may retrieve an image with a red sun at the right hand side of the image by just sketch a red circle on the right hand side of the image.

Many content-based retrieval multimedia database systems have been developed in the past few years. For example, Montage [2] allows users input histogram and sketch the query out for retrieval. Query by Image Content (QBIC) [3] allows users input color, texture, and shape of the database objects as the query. Photobook [4] makes use of semantics-preserving image compression to support search based on three image content descriptions: appearance, 2-D shape, and textural properties. VisualSEEK [5] is a content-based image and vedio retrieval system for World Wide Web. It uses color contents and spatial layout of color regions of images for retrieval. There are still many other multimedia database systems support content-based query for retrieval include Chabot [6], VLMSYS [7], ART MUSEUM [8], KMeD [9, 10], and CORE [11]. Although the above database systems use different approaches for multimedia management, most of them have shown that they are efficient for retrieval.

In a typical multimedia database system, all the database objects have to be pre-analyzed and then organized in a special way before retrieval. The main steps are:

1. **Feature Extraction:** Features are extracted from the database objects. The definition of features are usually pre-defined, such as color histogram and texture. These features are usually stored in the form of real-valued multi-dimensional vectors.

2. **Indexing:** The database may then organize the extracted features by using an indexing structure for retrieval.

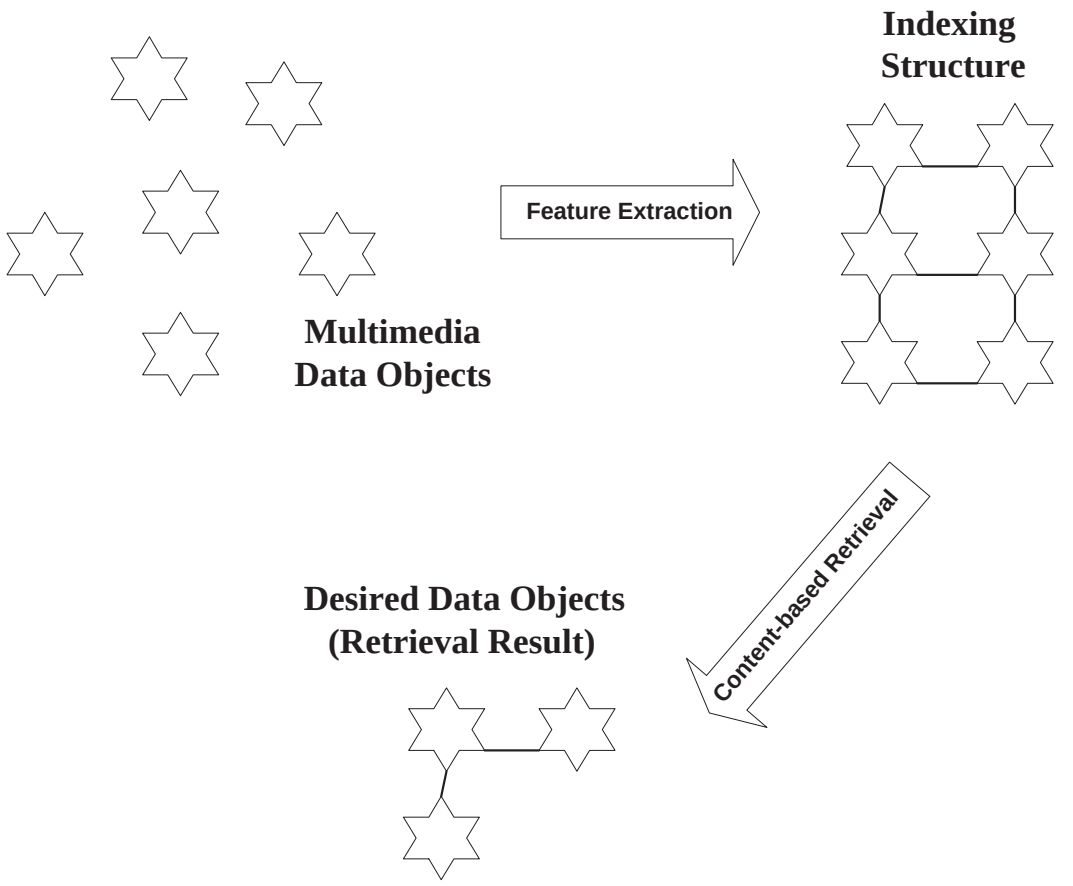


Figure 1.1: The flow of indexing and retrieval in a content-based retrieval multimedia database.

3. **Retrieval:** Content-based retrieval can be performed on the indexing structure efficiently and effectively.

In summary, Figure 1.1 shows the flow of the process.

Many multimedia database systems use multi-dimensional vectors to represent the data objects. So, it can support similar search easily. By applying a suitable distance function to the feature vectors as the similarity measurement, the database objects can then be ranked according to the distance (or similarity) between the query and database objects. The top ranked objects are then retrieved as the result for similar search. Nearest-neighbor search is a

typical kind of similar searching. As a result, if the database objects are represented in the form of multi-dimensional vectors, a nearest-neighbor is simply a multi-dimensional point (or vector) too. Then, the result of the query is the objects with features which are the neighbors of the query point. Using nearest-neighbor search, we can retrieve similar data objects easily.

For the multimedia databases with nearest-neighbor retrieval, a good indexing method is a key component for efficient and accurate retrieval. Nowadays, alphanumeric data indexing techniques are already well-developed such as [12, 13]. However, these database systems are designed for one-dimensional vectors. When the dimension of the vector increases, these database systems seem not to be very efficient. Therefore, people have begun to develop new indexing methods for content-based retrieval in databases such as R-tree [14], R+-tree [15], R*-tree [16], SR-tree [17], Quad-tree [18], k-d tree [19], VP-tree [20], MVP-tree [20] and some other methods [21, 22].

1.1 Problem Definition

Typically, natural data objects usually form clusters and these objects can be approximated using mixture of Gaussian distributions in the feature vector space. For nearest-neighbor search, a group of objects with similar features (or in the same natural cluster) will often be retrieved as the result of a query. From this observation, if we can first calculate the natural clusters from the feature space and build an indexing structure based on these cluster information, nearest-neighbor search will become more efficient and effective.

There are many existing algorithms for getting the natural cluster information. However most of them have a very high computational complexity. So,

it is impossible to use them in multimedia databases, which are usually huge in size.

Most of the existing indexing methods usually generate partitions for the feature vector space which lead to indexing structures for efficient retrieval in many cases, but as the main concern for them is how to build the indexing structure as balanced as possible, most of them seem to fail to retrieve similar database objects when a nearest-neighbor query lies on the partition boundary. One of the reasons is that these methods do not look at the distribution of the features to find natural clusters. So that features in the same natural cluster may be partitioned into several different partitions. As a result, the performance of nearest-neighbor searches for these methods will be degraded.

Thus, the problems we are facing are:

1. To find an efficient clustering method to locate natural clusters from the input feature vector set,
2. To build a good indexing structure based on the clusters for efficient and effective retrieval, and
3. To develop a good searching method based on the indexing structure for increasing the retrieval performance.

1.2 Contributions

The main contributions of our work for solving the problems defined in the last section are shown as follows.

1. We develop two clustering methods, Fuzzy Competitive Clustering (FCC) [23, 24] and Sequential Fuzzy Competitive Clustering (SFCC) to get the natural cluster information from the input feature vector set. FCC and

SFCC are unsupervised heuristic algorithms for clustering. They provide a good approximate of the cluster prototype with cluster information on each dimension. From the experimental results, we find SFCC is computational efficient. Therefore, we make use of SFCC to calculate the natural clusters for indexing and retrieval.

2. We build indexing structures based on natural cluster information in a hierarchical approach. The hierarchical approach transforms a feature space into a sequence of nested clusters and builds a hierarchical binary indexing tree (SFCC-b-tree) based on the clusters. We then apply a overlap checking technique (Section 4.2.5) on the indexing structure for efficient data retrieval. In short, we make use of the information of natural clusters for efficient and effective indexing and retrieval.

3. According to the experimental results of SFCC-b-tree, we use linear regression method to work out a formula to describe the relationship between the searching parameters and the searching efficiency. We can then make use of this formula to find out the estimated efficiency value for a given set of parameters. Besides, we can generalize the formula to other indexing methods for comparing their efficiency with a given set of parameters.

Our experimental results show that:

1. FCC and SFCC get better cluster prototypes than k-means, competitive learning, and rival penalized competitive learning.

2. SFCC is faster than k-means clustering algorithm and most of the off-line clustering algorithms.

3. SFCC-b-tree is faster and needs less instance access than VP-tree to produce 100% nearest-neighbor search results.

1.3 Thesis Organization

We organize the rest of the thesis as follows. First, we present the technical details and problems on multimedia database indexing, and clustering methods for both the fuzzy and non-fuzzy ones in Chapter 2. Then, we present our proposed fuzzy clustering algorithms in Chapter 3. We cluster the input data with two different approaches. Chapter 4 shows the hierarchical approach to build the binary indexing tree. Several experiments and discussions are presented in this chapter. Chapter 5 shows a case study on a real life problem of our indexing algorithm. Finally, we show how to work out the efficiency formula from the experimental results and have a brief summary of our proposed methods together with some suggested future directions in Chapter 6.

Chapter 2

Literature Review

We divide this chapter into three parts. The first part concentrates on *Content-based Retrieval Multimedia Database*. We give some backgrounds of this kind of databases and then in the second section, we present some problems found in the existing content-based indexing methods. In the third section, we present some *clustering methods* for both the fuzzy and non-fuzzy ones.

2.1 Content-based Retrieval, Background and Indexing Problem

In this section, we first give some technical backgrounds of the content-based retrieval multimedia databases: *Feature Extraction*, *Nearest-neighbor Search*, and *Content-based Indexing*. We then present some problems found in the existing content-based indexing methods.

2.1.1 Feature Extraction

Feature Extraction is one of the most important subjects in content-based retrieval multimedia databases. Feature extraction means extracting some useful features from the object. In content-based retrieval multimedia database, users may want to retrieve database objects similar to a query in terms of some kinds of features. Therefore, when a multimedia data object is inserted into the database, the useful features of the object will be extracted and transformed into feature vectors. The database then organizes the feature vectors for content-based retrieval.
The definition of feature extraction is:

Definition 2.1 (Feature Extraction) *Let* $DB = \{I_i\}_{i=1}^{n}$ *be a set of database objects. With a set of feature parameters* $\theta = \{\theta_i\}_{i=1}^{m}$*, a feature extraction function* f *is defined as:*

$$f : I \times \theta \rightarrow \mathcal{R}^d \ ,$$

which extracts a real-valued d-dimensional vector.

We use a simple example here to explain the above definition. Let $DB = \{I_1, \ldots, I_{10}\}$ be a set of 10 images and $\theta = \{\theta_1\}$ be the image feature parameter set where θ_1 indicates the number of top colors considered for extraction. $f(I_5, 2)$ will return a real-value vector based on the top two colors in the image I_5.

Many features can be used for feature extraction, such as, *color*, *texture*, and *shape*. Here are some examples for images.

1. **Color:** The color histogram is built and transformed to the feature vector [25].

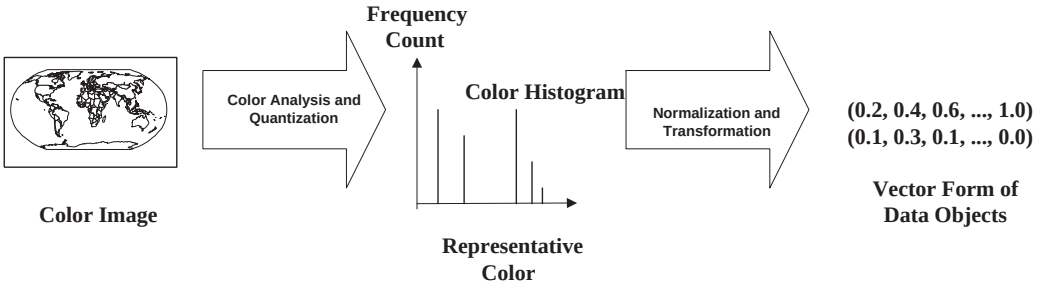


Figure 2.1: Feature extraction of a color image using color histogram.

2. **Texture:** There are some statistical methods use to analyze the texture information of an image [26]. Some researchers use Gabor filter for image scaling and orientation in texture analysis [27, 28].

Here is an example to illustrate the detail of the feature extraction using color histogram (Figure 2.1). Given an image, all the pixels in the image are quantized into n representative colors according to its pixel color. By calculating the frequency of each representative color, an n-bucket color histogram is formed. For fair comparison to other color histogram, the sum of the frequencies is normalized to 1. After normalization, the histogram is transformed into an n-dimensional feature vector for indexing and retrieval.

2.1.2 Nearest-neighbor Search

By using feature vector, content-based retrieval multimedia database allows users to perform similar search. In a typical similar search query, those database objects with similar features to the query will be retrieved as the result. Nearest-neighbor (NN) is one of the common similar searching techniques used in the MMDBs for content-based retrieval.

Nearest-neighbor search usually makes use of a distance function for similarity measurement. The distance function usually takes two feature vectors as

input and outputs a real value as the similarity measurement. In most cases, the smaller of the distance value, the more the similarity between each of the input features.

The definition of the distance function is:

Definition 2.2 (Distance Function) *A typical distance function D is defined as follows:*

$$D : F \times F \rightarrow \mathcal{R}$$

, satisfying:

1. $D(x, y) \geq 0$,
2. $D(x, y) = D(y, x)$,
3. $D(x, y) = 0$ *iff* $x = y$, *and*
4. $D(x, y) + D(y, z) \geq D(x, z)$.

where x, y, and $z \in F$ and F is a feature vector set.

One of the widely used distance function is the L_2-norm (Euclidean distance). It is defined as: $D(x, y) = \|x - y\| = \sqrt{\sum_{i=1}^{d}(x_i - y_i)^2}$.

Having the distance function, nearest-neighbor search in multimedia databases is a retrieval of database objects with features nearest to a query under the feature space with a given distance function. There are two main kinds of nearest-neighbor search, they are the *range-neighbor search* and *k nearest-neighbor search*.

The definition for these two kinds of search are:

Definition 2.3 (Range Nearest-neighbor Search) *Given a set of N features $X = \{x_i\}_{i=1}^{N}$, a range nearest-neighbor query $\hat{x}$ returns the set P of*

features:

$$P = \{x | x \in X \text{ and } 0 \leq D(x, \hat{x}) \leq \epsilon\}, \tag{2.1}$$

where ϵ is a pre-defined positive real number and D is a distance function.

Definition 2.4 (k Nearest-neighbor Search) *Given a set of N features $X = \{x_i\}_{i=1}^{N}$, a k nearest-neighbor query $\hat{x}$ returns the set $P \subseteq X$ satisfying:*

1. *$|P| = k$ for $1 \leq k \leq N$, and*
2. *$D(\hat{x}, x) \leq D(\hat{x}, y)$ for $y \in X - P$.*

where D is a distance function.

The main difference between range NN search and k-NN search is their inputs and retrieval results. In range NN search, we use the query point and a small positive number ϵ as the input. Then it gives the objects with features located inside the query hyper-sphere with radius ϵ as the results (Figure 2.2(a)). On the other hand, k-NN search takes the query point and a positive integer k as the inputs. It gives the objects with features which are the top k nearest neighbors to the query as the results (Figure 2.2(b)).

Many different algorithms for nearest-neighbor search have been proposed. Table 2.1 summarizes some of the searching algorithms.

2.1.3 Content-based Indexing Methods

In this section, we discuss some of the indexing methods. Most of the indexing methods can be classified into two main classes: *rectangle-based indexing* and *partition-based indexing*.

Algorithm	**Data**	**Metric**	**Result**
Burkhard and Keller (1973) [29]: Some approaches to best-match file searching	1,000 randomly generated registers of a file using 30-bits keys	Hamming distance	$\sim$ 700 average distance computations ($\sim$ 70 %)
Fukunaga and Narendra (1975) [30]: A branch-and-bound algorithm for computing K-nearest neighbors based on a hierarchical indexing structure	1,000 2-D uniform samples data	Euclidean distance	$\sim$ 580 average distance computations ($\sim$ 58 %)
Feustel and Shapiro (1982) [31]: The nearest-neighbor problem in an abstract metric space	29 randomly generated 5-vertices directed graphs	Graph-isomorphism-based discrete-valued distance	$\sim$ 3 average distance computations ($\sim$ 10 %)
Kamgar and Kanal (1985) [32]: An improved branch-and-bound algorithm for computing k-nearest neighbors based on a hierarchical indexing structure	1,000 2-D samples uniform sample data	Euclidean distance	$\sim$ 165 average distance computations ($\sim$ 16.5 %)
Roussopoulos et al. (1995) [33]: Nearest neighbor queries for R-tree	1K, 4K, 16K, 64K, and 256K synthetic uniformly distributed data sets	MINDIST and MINMAXDIST distances	The no. of nearest neighbors increased the no. of pages accessed grew in a linear ratio
Nene and Nayer (1997) [34]: A simple algorithm for nearest-neighbor search in high dimensions	30,000 and 100,000 high dimensional uniform and normal distribution samples	Euclidean distance	$\sim$ 20 % of search time used than exhaustive search for 30,000 10-D data and $\sim$ 40 % of search time used than exhaustive search for 30,000 25-D data

Table 2.1: Searching performance of some nearest-neighbor search algorithms. [1]

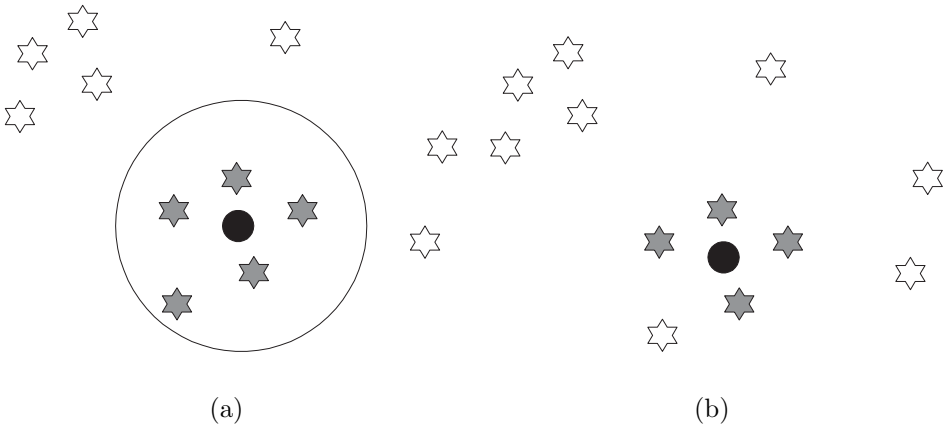


Figure 2.2: (a) Range nearest-neighbor search in 2-D. (b) k nearest-neighbor search in 2-D ($k = 4$).

Rectangle-based Indexing

Rectangle-based indexing methods use rectangles to organize the features into groups for indexing. *R-tree*, *R+-tree*, *R*-tree*, and *SR-tree* are some classical examples of rectangle-based indexing methods.

R-tree

R-tree [14] is the generalization version of the B-tree [12, 13] for multi-dimensional data indexing. It uses rectangles to partition the data into groups. The partition process proceeds hierarchically until all the leaf nodes contain a number of instances within a pre-defined range.

- **Properties:** R-tree is a balanced tree with *Leaf Node* and *Non-leaf Node* only. Let M and $m \leq M$ be the maximum and the minimum number of entries that a node can contain respectively. Then, every leaf node except the root node must contain a number of entries between m and M. Also, every non-leaf node except the root has between m and M children. The root node has at least two children unless it is a leaf node.

- **Insertion:** R-tree is built by inserting the data objects one by one. It does not consider the global data distribution in tree building. Figure 2.3 shows an example of R-tree. Starting from the root node with a minimum bounding rectangle (MBR) which is the smallest rectangle containing all the data objects for the node. Data will continue to be inserted into the node until overflow. When the node has overflowed, a splitting algorithm is applied to partition the corresponding rectangle into several smaller rectangles for child nodes.

- **Deletion:** Apart from insertion, deletion is also a major operation of R-tree. After deleting a data object from a node, a merging algorithm is applied if the deleted node contains less than m objects.

- **Searching:** The searching algorithm for R-tree is not very difficult. Given a query, all the nodes in R-tree with MBRs that overlapped with the query rectangle are examined in order to find the result of the query. R-tree works fine in many cases. However when the query lies on the overlapping area of two or more minimum bounding rectangles or the degree of overlapping between those minimum bounding rectangles is high, the efficiency is very low for R-tree. Because all the involved rectangles have to be examined in order to find out the result of query which reduces the efficiency of the retrieval in such cases. Therefore, it is better to decrease the overlapping area as much as possible to make the retrieval more efficient.

R+-tree

R+-tree is [15] is a variation of R-tree. It tries to prevent the high overlapping of MBRs by modifying the searching and updating algorithms. According to the experimental results shows in [15], R+-tree has a better searching performance when compared with R-tree. Also, it is more efficient for indexing point

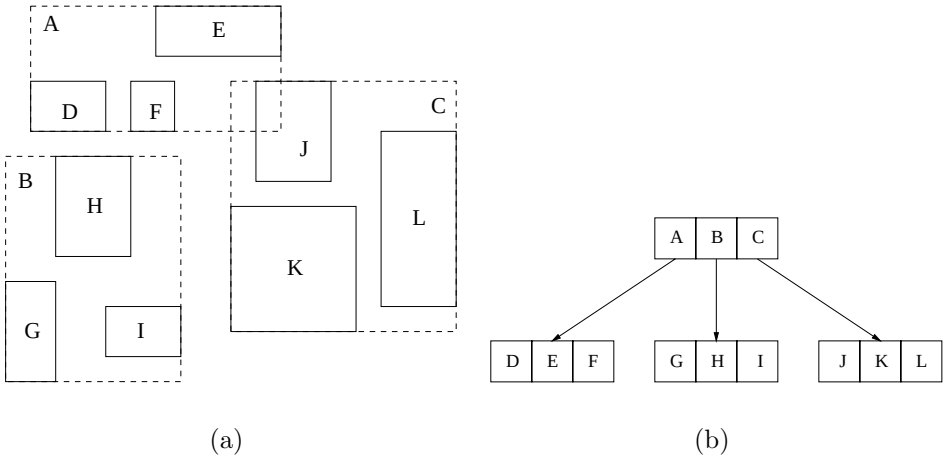


Figure 2.3: (a) An input data set partitioned by using minimum bounding rectangles. (b) The corresponding R-tree structure.

data and point queries than R-tree.

R*-tree

R*-tree [16] is another variation of R-tree. The authors of R*-tree showed in [16] that overlapping-region-technique does not imply bad searching performance. They also find that in order to get a better searching performance, some essential points should be considered:

1. The area covered by a MBR should be minimized.
2. The overlap between MBRs should be minimized.
3. The margin of a MBR should be minimized.
4. Storage utilization should be optimized.

Therefore, the authors modify the splitting algorithms used in R-tree to increase the searching performance by reducing the area of MBR, margin, and overlap of rectangles. Moreover, the storage utilization is higher than R-tree.

In short, from the experimental results in [16], R*-tree outperforms the other R-tree variants.

SR-tree

SR-tree [17] is the extension of R*-tree [16] and the SS-tree [22]. It stands for Sphere/Rectangle-tree. As its name suggests, it makes use of both rectangles and spheres for indexing. It improves the performance on nearest-neighbor search by reducing both the volume and the diameter of regions compared with the R*-tree and SS-tree. According to the experimental results in [17], SR-tree performs much better than R*-tree especially in the high dimension vector space.

Partition-based Indexing

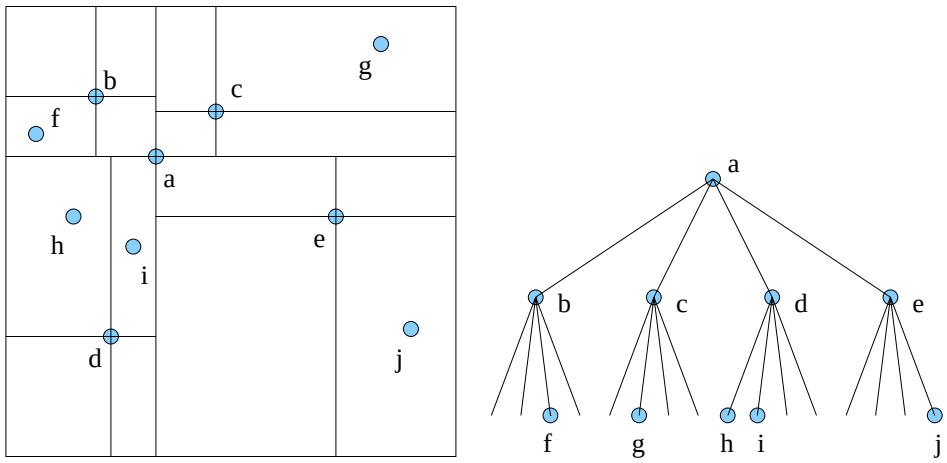
Partition-based indexing methods use lines or curves to partition the feature vector space into partitions for indexing. *Quad-tree*, *k-d tree*, *VP-tree*, and *MVP-tree* are some classical examples for partition-based indexing methods.

Quad-tree

Quad-tree [18] is one of the first indexing methods developed for multi-dimensional data. It is the generalization version of binary tree.

- **Properties:** Quad-tree divides the feature vector space into partitions according to the direction of the data points. Using two-dimensional space as an example, each non-leaf node in Quad-tree has four child nodes representing its four directions NE, SE, SW, NW. Figure 2.4 shows an example of Quad-tree.

- **Insertion:** Same as typical binary tree, Quad-tree are built by inserting data objects one by one into the tree. When a new data object is inserted, its corresponding direction to the root node is determined and



(a) (b)

Figure 2.4: (a) An input 2-D data set for quad-tree. (b) The corresponding quad-tree structure.

the corresponding child node will be selected for further processing until the leaf node level is reached.

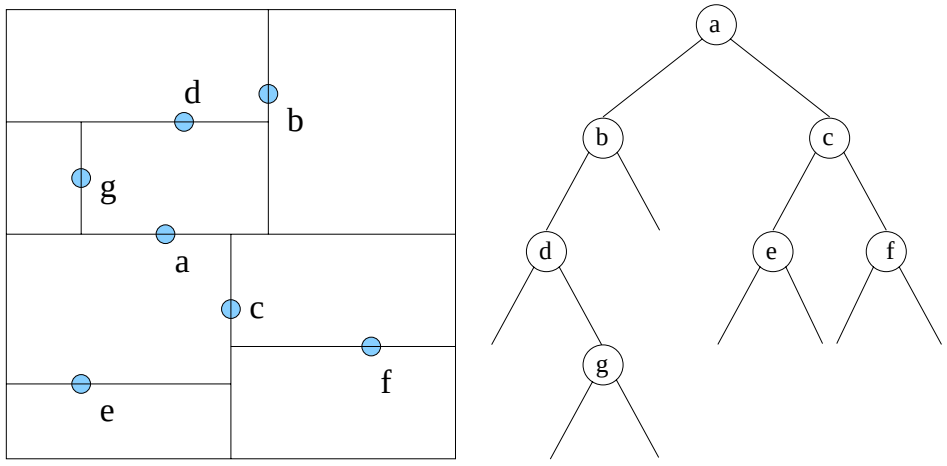
- **Searching:** Searching in quad-tree is based on the direction of the query to each non-leaf node. Assume the vector space is in k-dimension. Given a query, it compares all the k coordinates to the node and determines which child node (or direction) is examined next. This process will then repeat until the target leaf node is found.

The insertion algorithm yields an $O(n \log n)$ performance in the 2-D case. So, it is an efficient algorithm for 2-D vector space.

k-d tree

k-d tree [19] is also a kind multi-dimensional binary search tree where k denotes the dimensionality of the search space.

- **Properties/Searching:** The searching algorithm is similar to quad-tree



(a) (b)

Figure 2.5: (a) An input data set for k-d tree. (b) The corresponding k-d tree structure.

except we just need to compare one different attribute value at each level of tree. For example, in 2-D space, we are comparing the x coordinates at even levels while we are comparing the y coordinates at odd levels.

- **Insertion:** k-d tree are built by inserting data objects one by one into the tree. When a new data object is inserted, it first compares the pre-selected attribute between itself and the node to determine a child node for further processing. The process will continue until a leaf node is reached. Then, the data object will be inserted and it partitions the space associated with the leaf node into sub-spaces for two child nodes according to a suitable attribute value. Figure 2.5 shows an example of k-d tree.

Because it only needs to consider part of attributes for a query at each node, k-d tree is relatively more efficient than quad-tree for indexing and retrieval.

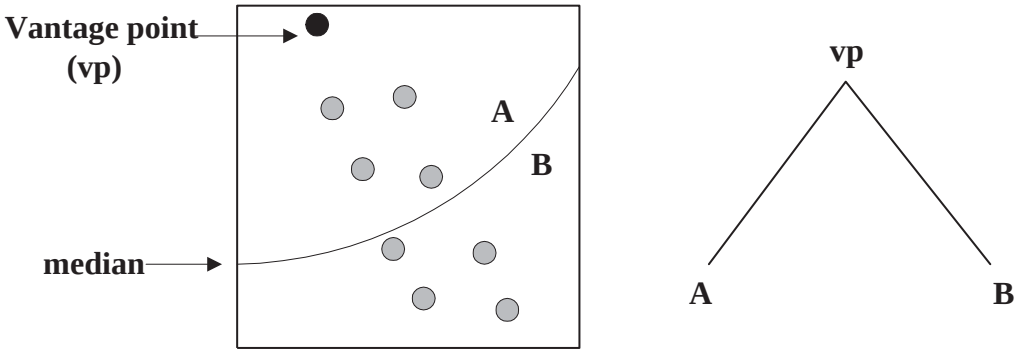


Figure 2.6: A simple VP-tree for the data set on the left.

VP-tree

Vantage point tree (VP-tree) [35, 20] is an indexing method for multi-dimensional nearest-neighbor search.

- **Properties:** VP-tree is also a partition-based indexing method. The difference between VP-tree and k-d tree is VP-tree partitions the feature space based on the distance between the feature vectors and a calculated vantage point.

- **Building Indexing Tree:** VP-tree divides the vector space according to the distance between data points and the vantage point. According to the median of these distance, the whole feature space is divided into two partitions, it is those with distance smaller than the median distance and distance larger than median distance. This process will continue in the sub-partitions individually, until an indexing tree structure are built based on the resultant vector sets. Figure 2.6 shows an example of VP-tree.

- **Searching:** For searching in VP-tree, the query first calculates its distance from the vantage point associated with the root node. Then we compare the distance with the median of the node to determine which child node is going to be examined next. The process repeats until the

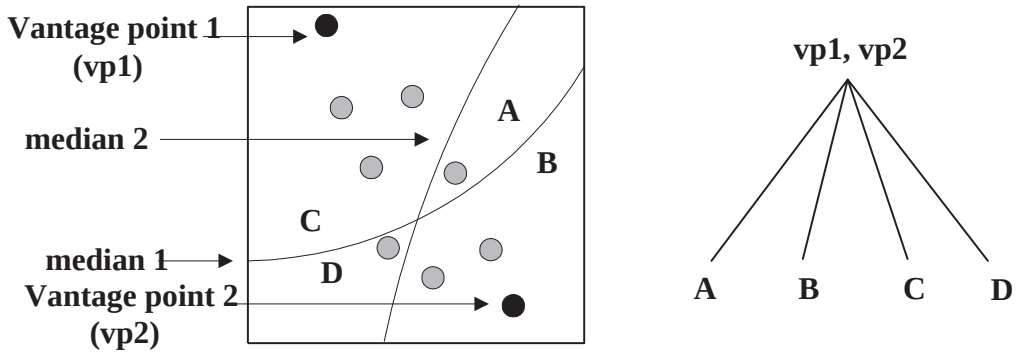


Figure 2.7: A simple MVP-tree for the data set on the left.

target leaf-node is found. The data objects associated with the leaf-node satisfying the searching criteria of the query will be retrieved as the result.

The experimental results in [20] show that VP-tree preforms better than k-d tree.

MVP-tree

Multi-vantage point tree (MVP-tree) [36] is a variation of VP-tree. MVP-tree also uses vantage point and median to partition the vector space. However, MVP-tree uses two or more vantage points to partition the feature space. So, MVP-tree is not a binary tree.

- **Building Indexing Tree:** MVP-tree uses two or more vantage points to partition the feature space. Use MVP-tree with 2 vantage points as an example, Figure 2.7 shows the partitions of one of the tree levels.

- **Properties:** As MVP-tree has more than one vantage point. Every non-leaf node has more than two child nodes. According to the experimental results shown in [36], MVP-tree outperforms VP-tree in high dimensional data.

2.2 Indexing Problems

In a typical multimedia database, data objects usually form clusters. We use document database as an example.

In a document, usually it contains "keywords". Keywords mean those words that can describe the document. Also, those similar document may contain similar keywords. For example, document about vehicle may contain words like car, speed, and so on. However, those documents about food may contain fish, meat, and so on.

As a result, similar document forms natural clusters. Usually, these kinds of clusters can be approximated by a Gaussian distribution.

On the other hand, we know that for nearest-neighbor search, a group of similar data instances is often retrieved together as the results of the query. In other words, instances in the same natural cluster will be retrieved as a group of results. Therefore, if we can first calculate the natural clusters from the feature vector space and then build the indexing structure according to these natural clusters information, nearest-neighbor search on the structure will become more efficient and effective.

As mentioned in the previous chapter, most of the indexing structures do not consider the natural clusters information. They may partition a natural cluster into several different nodes.

Consequently, they seem to work fine for many cases in general, but fail to retrieve similar database objects when the nearest-neighbor query lies on the partition boundary. We call this the boundary problem.

For example, those rectangle-based indexing methods like R-tree and R+-tree are based on the input sequence of the data objects. So, they do not pay any attention on the natural clusters information. For those partition-based indexing methods like VP-tree, they partition the feature space based on the median distances from the data objects to the vantage point. However, it does not consider the natural clusters information as well. As a result, the performance of the nearest-neighbor retrievals for these methods is reduced by the boundary problem.

2.3 Data Clustering Methods for Indexing

We propose to use an efficient clustering algorithm for content-based indexing to the indexing problems mentioned in the last section. Assume there exist natural clusters in the data set, we can locate these clusters and build an indexing structure based on these information.

Although the term of clustering is different in different fields, it refers to an automatic unsupervised classification method in data analysis.

There are many different types of clustering algorithms, each has its own advantages and disadvantages, from k-means clustering algorithm to Fuzzy c means algorithm [37]. However, almost all the common clustering algorithms [38, 39, 40, 41, 42, 43, 44] nowadays can divide into two groups. They are "probabilistic clustering" [45] and "possibilistic clustering" [46]. In the later part of this paper, we will have a brief discussion on these two main classes of clustering algorithm.

In 1960's, Zadeh proposed the "fuzzy set theory" [47]. In the fuzzy set

theory, the degree that an instance belonging to a class is indicated by the membership, which ranges between zero to one. Later on, researchers started to apply fuzzy set theory in clustering and a number of fuzzy clustering algorithms have been proposed [38, 39, 40, 41, 42, 43, 44].

The most famous fuzzy clustering method is the fuzzy c means clustering [39, 40, 41]. However, it is better to state that c means clustering does not mean a specific algorithm but a class of algorithms. Recently, new methods of fuzzy c means have been proposed. However, their main ideas are very similar to the traditional c means clustering.

2.3.1 Probabilistic Clustering

The main property of "probabilistic clustering" is that they all obey the constraint:

$$\sum_{i=1}^{i=c} u_{ik} = 1; k = 1, ..., n \ , \tag{2.2}$$

where, u_{ik} is the membership value of instance k towards concept i, c is the number of clusters, and n is the number of instances.

Here we illustrate some of the probabilistic clustering algorithms.

Fuzzy c Means Clustering (FCM)

Fuzzy c Means clustering [37] algorithm is the most common fuzzy clustering algorithm. The basic idea of FCM is very similar to k-Means algorithm. It assumes that the number of clusters c, is known a *priori*, and tries to minimize the cost function:

$$J_{fcm} = \sum_{i=1}^{i=c} \sum_{k=1}^{k=n} u_{ik}^{m} d_{ik}^{2} \ , \tag{2.3}$$

subject to the n probabilistic constraints,

$$\sum_{i=1}^{i=c} u_{ik} = 1; k = 1, ..., n \ , \tag{2.4}$$

where, u_{ik} is the membership value of instance k towards concept i, and

$$d_{ik} = \left\| x_k - v_i \right\| \ . \tag{2.5}$$

As $m = 1$ FCM converges in theory to the traditional k-means solution [48]. So, m is usually set larger than 1. Usually, $1.5 < m < 2.0$.

The conditions for local Minimum for the cost function in Equation 2.3 are derived using Lagrangian multipliers [49] and the results are:

$$u_{ik} = \left(\sum_{j=1}^{j=c} \left(\frac{d_{ik}}{d_{jk}} \right)^{1/m-1} \right)^{-1} \quad \forall i, k \ , \tag{2.6}$$

and

$$v_i = \frac{\sum_{k=1}^{k=n} \left(u_{ik}^m x_k \right)}{\sum_{k=1}^{k=n} \left(u_{ik}^m \right)} \quad \forall i \ . \tag{2.7}$$

Minimization of J_{fcm} is performed by a fixed-point iteration scheme known as the Alternating Optimization (AO) technique [50]. Figure 2.8 shows an example of using FCM for clustering. Those dots in the figure represent the input data objects and crosses stand for the cluster centers.

However, it is needless to say that, FCM is a wonderful method because FCM introduce a "strange number" m. However, no one knows the physical meaning of the parameter m. Also, there is no idea about how to make an optimal choice of the parameter m. On the other hand, from the mathematical point of view, the value m in Equation 2.3 is unnatural and unnecessary.

There is another great problem on FCM and many other Probabilistic Clustering algorithms. This is the problem of "Outliers" [51].

Outliers are vectors, or called data point, in the data domain which are so distant from the rest of the other vectors in the data set. As a result, it would be unreasonable to assign them high membership values to any of the c clusters. The constraint in Equation 2.4 does not permit all the c memberships to assume value lower than $1/c$. For an outlier x_k, all the ratios d_{ik}/d_{jk} will often be close to unity, resulting in all the c membership values close to $1/c$. Because FCM and many other Probabilistic Clustering algorithms use the membership value as a weighting to calculate the cluster centroid. These unreasonable high membership values often cause improper positioning of the centroids. In fact, if an outlier is very distant, one of the centroids may position itself at the outlier's position.

The problems caused by the outliers are referred to as "noise sensitivity". Researchers have developed many "noise resistant" clustering algorithm, and a noise resistant clustering algorithm should have the following properties:

(1) An outlier should have low membership value to all the c clusters.
(2) Centroids generated by the algorithm on a noisy set should not deviate significantly from those generated for the noiseless set, obtained by removing the outliers.

The Noise Cluster Approach

In 1990's, Dave [52] proposed the noise cluster approach to solve the problem of noise sensitivity. In the noise cluster approach, we define a class of outliers, called the *noise cluster*. An extra centroid is used as a "representative" (prototype) for all the outliers.

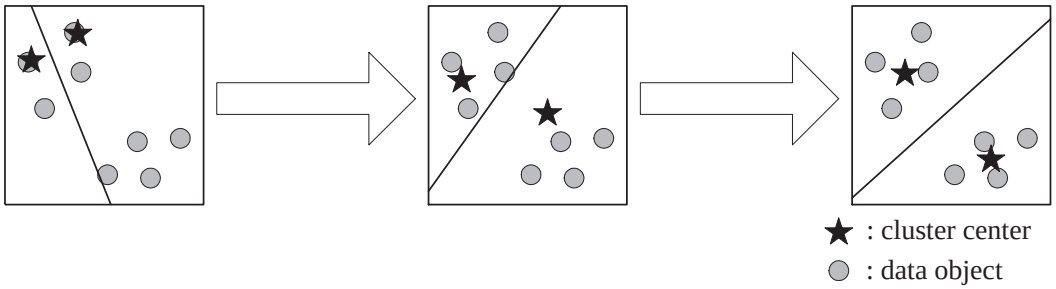


Figure 2.8: An example of fuzzy c means clustering.

Dave also proposed that this prototype has a constant distance δ from all the vectors in the data domain. The memberships of the vectors in the data set to the noise cluster are defined as:

$$u_{*j} = 1 - \sum_{i=1}^{i=c} u_{ij} \ , \tag{2.8}$$

and the Dave's objective function J_{NC} is given by

$$J_{NC} = \sum_{i=1}^{i=c}\sum_{k=1}^{k=n} u_{ik}^m d_{ik}^2 + \sum_{k=1}^{k=n} \delta^2 \left(1 - \sum_{i=1}^{i=c} u_{ik}\right)^m \ . \tag{2.9}$$

The conditions for local minima of J_{NC} are given by

$$v_i = \frac{\sum_{k=1}^{k=n}\left(u_{ik}^m x_k\right)}{\sum_{k=1}^{k=n}\left(u_{ik}^m\right)}, \quad \forall i \ , \tag{2.10}$$

and

$$u_{ij} = \left(\sum_{k=1}^{k=c}\left(\frac{d_{ij}^2}{d_{kj}^2}\right)^{\frac{1}{m-1}} + \left(\frac{d_{ij}^2}{\delta^2}\right)^{\frac{1}{m-1}}\right)^{-1} \ . \tag{2.11}$$

The clusters (or partitions) are generated by Alternating Optimization. The noise cluster approach works fine if the appropriate value of δ is given,

however, there is no consistent method to find a good value of δ for a given data set at present time. [53]

Competitive Learning (CL)

Competitive Learning [54, 55, 56] is another famous learning algorithm in clustering. Applications using CL had applied in marketing [57, 58], image processing [59, 60], and even in education [61].

Before having a short discussion on this learning algorithm, we better state out the concepts behind this algorithm first. Actually, this algorithm has the same concept with k-means algorithm [62, 63, 64]. It is to minimize the total distance from each data point to the cluster center it belongs to. Recall what we did in k-means algorithm, in every loop, we first check which cluster does the data point belonging to. After this, we update the cluster center according to those data points assigned to that cluster. As a result, the cluster center moves towards to these data points and converge at the center of these data points.

From the above observation, people try to develop another clustering algorithm, called "Competitive Learning", which have lower complexity. In competitive learning, rather than considering ALL the N data point, it only takes one data point at each time. The selected data points then attract the nearest cluster centers toward itself. As a result, the effect is similar with those in $k-$means algorithm.

Assuming the centroid of i^{th} cluster is m_i, with a small value of learning rate α, the updated centroid m_{inew} in each loop is given from the old value of centroid m_{iold} by:

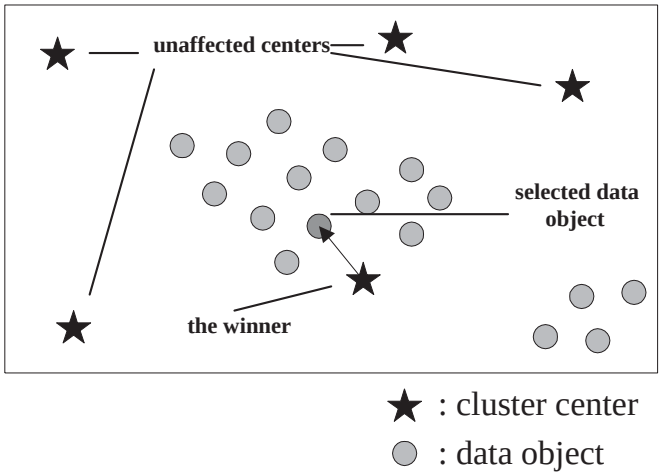


Figure 2.9: An example of competitive learning clustering.

$$m_{inew} = \begin{cases} m_{iold} + \alpha \times d_{ij} & \text{if } d_{ij} \text{ is the smallest ,} \\ m_{iold} & \text{otherwise .} \end{cases} \tag{2.12}$$

Figure 2.9 shows an example of Competitive Learning.

By now, we may find that competitive learning is actually a special type of probabilistic clustering. It is a Winner-Takes-All (WTA) version of probabilistic clustering algorithms. However, it gives a lower complexity and usually a shorter convergence time when compared with other types of probabilistic clustering algorithms.

Also, competitive learning is seem to be a noise resistance algorithm. Because although outliers force the centroid goes toward itself, the effect of this move decrease as the number of loop. As a result, if we can ensure the algorithm runs enough of loops, we can even omit the effect of the outliers. As competitive learning has such an advantage it is widely applied in many different field [57, 58, 59, 60, 61].

However, there are two main problems on the competitive learning. The first is called "died unit". Imagine in a normal two clusters problem. In this problem, everything go seem to be right and perfectly. However, in the initialization state, we define the two centroids as follow:

Centroid 1:
Almost at the mean of these two clusters.

Centroid 2:
Very far away from these two clusters. The distance is enough to make this centroid never be the winner.

Then after we start to run the competitive learning, you may find that both the centroids seem to be state at the initial positions rather then moves toward the center of these two clusters. It is because in competitive learning, only the winning centroid will be updated. As a result, if there exist some centroid that always lose in the competitive, the centroid will then never update and it is called the "died unit"

Another problem is also about the initialization of centroids. Consider there are two clusters in the data set. One of the cluster in the data set has a large variance and large number of instance, another cluster is the reverse of this one, it has a small variance and small number of instances. Then we initialize both the centroids around the mean of the "big cluster".

After the learning, you may also find that rather then getting the result of two clusters with one big and one small, the result tell you that the centroid of these two cluster are very near and the centroid of these cluster is almost

same as the centroid of the "big cluster".

It is because although the "small cluster" attracts the cluster centroid toward itself, its effect decrease as number of loops and also the attraction for the "small cluster" is too small when compare with the "big" one. As a result, centroids may trapped in a big cluster and give wrong solution in clustering.

Rival Penalize Competitive Learning (RPCL)

RPCL [65] is actually a more general case of CL. The idea comes from the weakness on resource distribution of CL. In CL, a cluster center can keep in starvation if it is far away from the data. It can also always be the winner. As a result, bad result may occur from such a clustering method.

RPCL solve this problem by adding two factors into the traditional CL. The first one is called the "frequency sensitive", and the second one is called the "Rival Penalty". Frequency sensitive means a cluster center will be prohibited to win more if it wins too much times. As a result, starvation will not occur and every cluster center have a chance to win. However, using frequency sensitive alone is not enough, it is because frequency sensitive ensures every cluster center have a chance to win. Assume there are three clusters in the data but we think the number of cluster should be four. If frequency sensitive is used, the data may be shared by all the cluster centers and give a wrong result. Rival Penalty is a method to take the redundancy cluster center away from the data. The main idea is that it gives penalty to the rival (or second winner). As a result, those cannot get a cluster becoming a rival at all times, and finally gets out from the data.

$$m_{inew} = \begin{cases} m_{iold} + \frac{win\_number_i}{total\_trial}\alpha d_{ij} & \text{if } d_{ij} < d_{ik} \forall k, \\ m_{iold} - \frac{win\_number_i}{total\_trial}\beta d_{ij} & \text{if } d_{ij} \text{ is the 2nd smallest,} \\ m_{iold} & \text{otherwise.} \end{cases} \tag{2.13}$$

2.3.2 Possibilistic Clustering

Different from "probabilistic clustering", "Possibilistic clustering" does not obey the constraint shown in Equation 2.2.

On the other hand, a probabilistic clustering method such as FCM gives inter-cluster information but no intra-cluster information. Consider the follows four membership in a probabilistic clustering method, $u_{ik}, u_{ij}, u_{rk}, u_{rj}$. where u_{ik} stands for the membership value of k^{th} instance for i^{th} cluster.

Suppose $u_{ik} > u_{rk}$, then the instance x_k belongs to i^{th} cluster with a higher probability than to the r^{th} cluster. This kind of information is called the inter-cluster [66] information. However, no conclusion can be drawn from the values u_{ij} and u_{ik} as to which of the instance among x_k and x_j belongs to the i^{th} cluster to a greater degree, that is, no inter-cluster information.

Possibilistic clustering is the reverse of probabilistic clustering, it tries to generate memberships interpreted as typicalities. If typicality $t_{ik} = \alpha$, then x_k belongs to the i^{th} cluster with a possibility α. Suppose the typicalities generated from a possibilistic clustering method are, $t_{ik}, t_{ij}, t_{rk}, and t_{rj}$. If $t_{ij} > t_{ik}$, then we conclude that x_j belongs to the i^{th} cluster with a greater degree than x_k. However, no conclusions can be drawn from the values t_{ij} and t_{rj} as to the relative degree to which x_j belongs to the i^{th} and r^{th} clusters.

The Possibilistic c means (PCM) algorithm [67, 68] is one of the most

famous possibilistic clustering algorithm. It tries to minimizes the objective function:

$$J_{PCM} = \sum_{i=1}^{i=c}\sum_{k=1}^{k=n} t_{ik}^{m} d_{ik}^{2} + \sum_{i=1}^{i=c} \eta_i \sum_{k=1}^{k=n} \Big(1 - t_{ik}\Big)^{m}. \tag{2.14}$$

Here, η is a measure of the radius of the i^{th} cluster and is called the "bandwidth parameter". The conditions for local minima for the cost function (Equation 2.14) is given by:

$$t_{ik} = \left(1 + \left(\frac{d_{ik}^{2}}{\eta_i}\right)^{\frac{1}{m-1}}\right)^{-1} \quad \forall i, k, \tag{2.15}$$

and

$$v_i = \frac{\sum_{k=1}^{k=n}\left(t_{ik}^{m} x_k\right)}{\sum_{k=1}^{k=n}\left(t_{ik}^{m}\right)} \quad \forall i. \tag{2.16}$$

The minimization is an AO on Equation 2.15 and Equation 2.16. There has several methods for choosing the value η [67].

One great problem for PCM is the objective function for PCM (Equation 2.14) can break down into a sum of c signal objective function. As a result, the centroids do not "affect" each other during the optimization process. This property often leads to "coincident clusters" [69]. Another problem for PCM is the result of PCM heavily depends on the initialization. The authors of [67] suggest to use FCM to initialize PCM. However, if an outlier is distant, PCM will not be able to recover from the "bad" initial partition generated by FCM.

Chapter 3

Fuzzy Clustering Algorithms

In this chapter, we introduce two fuzzy clustering algorithms, *Fuzzy Competitive Clustering* [23, 24] and *Sequential Fuzzy Competitive Clustering*. Both the proposed algorithms are members of possibilistic clustering. Which means they do not need to obey the constraint shown in Equation 2.2. After a description on both the algorithms, we briefly explain the differences between FCC, SFCC, and traditional clustering algorithms. We then conduct experiments on these two clustering algorithms and some other clustering algorithms. At the end of this chapter, we will make a comparison on their properties and performance and state why SFCC is the most suitable clustering algorithm to be used in multimedia indexing.

The aim for the proposed clustering algorithms is to generate clusters for multimedia databases. A typical multimedia database usually consists of the following properties:

1. High Dimensionality.
2. Huge Amount of Sample Points.
3. Contain Natural Clusters.
4. Many Noise in the Data.

So, a clustering algorithm suits for multimedia database should be able to perform well under databases with the above properties. In the remaining of the chapter, we perform a series of experiments, and conclude that SFCC is the most suitable clustering algorithm among all the tested algorithms.

3.1 Fuzzy Competitive Clustering

Fuzzy Competitive Clustering (FCC) is an extension of traditional statistical-based clustering algorithm. The main difference between FCC and traditional statistical-based algorithm is that in traditional competitive learning, the measurement in the competition step is the absolute distance; however, in FCC the measurement is the fuzzy membership value, which is a relative distance measurement. We show in the later experiments that, it is more flexible and robust when compared with those algorithms using absolute distance as a measurement.
The algorithm of FCC is outlined as follows:

(Step 0) Initialization: Every cluster in FCC is described by a fuzzy prototype [70, 71]. In the initialization step, we randomly pick k points as the initial cluster prototype centers and every prototype has the same variance in each dimension as the initial variance of the cluster prototypes.

(Step 1) Competition: Calculate the fuzzy membership value for each data instance to each cluster prototype. The membership value u_{ik} for data instance x_k to cluster i is calculated from the equation:

$$u_{ik} = \frac{\sum_{j=1}^{a} u_{jik}}{a} \ . \tag{3.1}$$

where a is the the number of attribute and u_{jik} is the membership value of data instance x_k to cluster i in the j^{th} dimension.

u_{jik} can be any fuzzy membership function. In our experiment, we use the *crisp* function as the fuzzy membership function and it is defined as:

$$u_{jik} = \frac{\sigma_{ji} + 1}{\sigma_{ji} + d(i,k) + 1} \ , \tag{3.2}$$

where σ_{ji} is the variance of i^{th} cluster prototype in the j^{th} dimension. $d(i,k)$ is the distance between instance x_k and the i^{th} cluster center.

We use the crisp function as the membership function because it has the following property:

$$u_{jik} \leq 0.5 \quad \text{if} \ \ d(i,k) \geq \sigma_{ji}. \tag{3.3}$$

When we try to convert a fuzzy membership value into a boolean value. It is common that we set the threshold as 0.5. So, by using crisp function, the variance becomes the boundary between belonging (TRUE) or not belonging (FALSE) to the cluster. This feature gives an easy way to understand the cluster properties. So, crisp function is used here instead of other functions.

After the calculation of fuzzy membership values, we increase the weighting, w_{ik}, of the k^{th} instance towards i^{th} cluster if the membership value of this data instance is the largest towards this cluster. The weighting is changed according to the following equation:

$$w_{ik} = \begin{cases} u_{ik} + \eta\Big(1 - u_{ik}\Big) & \text{if } u_{ik} \text{ is the largest,} \\ u_{ik} & \text{otherwise.} \end{cases} \tag{3.4}$$

where, η, $0 \leq \eta \leq 1$, is the learning rate.

The above process of w_{ik} is similar to the normalization process in traditional clustering algorithms. If u_{ik} is the the largest, then its weighting, w_{ik}

should be the largest among w_{ik} for all k. Through this kind of *normalization like* process, we let each cluster prototype interact with each other to prevent generating *coincident clusters* [69]. Coincident clusters mean those clusters having very similar cluster prototype and we cannot find a clear difference between them.

(Step 2) Updating Cluster Fuzzy Prototypes: We update the cluster prototype by

$$m_i^{'} = \frac{\sum_{k=1}^{n} w_{ik}^2 x_k}{\sum_{k=1}^{k=n} w_{ik}^2} \ , \tag{3.5}$$

$$\sigma_i^{'} = \frac{\sum_{k=1}^{n} w_{ik} \left\| x_i - m_i \right\|}{\sum_{k=1}^{k=n} w_{ik}} \ . \tag{3.6}$$

where, $m_i^{'}$ is the new cluster centroid of the i^{th} cluster and $\sigma_i^{'}$ is the new variance vector of the i^{th} cluster. A variance vector stores the variance of each dimension for a cluster.

Steps 1 and 2 are iterated until the iteration converges or the number of iterations reaches a pre-specified value. The final cluster prototypes are the results of the FCC.

After FCC, every data point is assigned a membership value to a cluster according to the Equation 3.1. If hard-cut boundary is used, the data point belongs to the cluster that gives the highest membership value.

3.2 Sequential Fuzzy Competitive Clustering

In this section, we will introduce an efficient fuzzy clustering algorithm, sequential fuzzy competitive clustering (SFCC). In Chapter 4, we show how we

make use of SFCC for content based indexing in order to lessen the boundary problems mentioned in Chapter 2.

SFCC is an online clustering algorithm, which means that we do not need to get the whole training data set before clustering. Actually, as the number of data objects exist in database become larger and larger, it becomes impossible to use offline clustering algorithms for such a database. So, off-line clustering algorithm would be a suitable clustering algorithm for huge database.

The algorithm of SFCC is outlined as follows:

(Step 0) Initialization: Every cluster in SFCC is described by a fuzzy prototype. In the initialization, we randomly pick k points as the k initial cluster prototype centers and every prototype has the same variance in each dimension as the initial variance of the cluster prototypes.

(Step 1) Competition: Randomly pick a data instance from the training data set, and calculate its fuzzy membership value for this instance to each cluster prototype. The membership value of an instance to a cluster is calculated by:

$$u_{ik} = \lambda_i^{\frac{1}{\beta T}} (\prod_{j=1}^{a} u_{jik})^{1/a} \ . \tag{3.7}$$

where a is the the number of attribute, u_{jik} is the membership value of data instance x_k to cluster i in j^{th} dimension, T is the number of iterations so far, and β is a constant used to control the adaptive rate. λ_i is a value used to make sure that all the cluster prototypes have a chance to become the winner cluster. It is defined as:

$$\lambda_i = 1 - (\frac{number\ of\ winning\ by\ cluster_i}{number\ of\ total\ loops}) \ . \tag{3.8}$$

Similar to FCC, u_{jik} can be any fuzzy membership function. In our experiment, we use the *crisp* function as the fuzzy membership function and it is defined as:

$$u_{jik} = \frac{\sigma_{ji} + 1}{\sigma_{ji} + d_j(i,k) + 1} \ . \tag{3.9}$$

where σ_{ji} is the variance of the i^{th} cluster prototype in the j^{th} dimension. $d_j(i,k)$ is the distance between instance x_k and the i^{th} cluster center in the j^{th} dimension.

After we have calculated the membership value, we find the **winner** cluster. The winner cluster is the cluster with the highest membership value.

(Step 2) Updates: After we found the winner and rival clusters, we update these cluster prototypes by:

$$m_w^{'} = m_w + \alpha u_{iw}(x_k - m_w) \ , \tag{3.10}$$

$$\sigma_{iw}^{'} = \sigma_{iw} + \alpha u_{iw}(d_j(i,k) - \sigma_{iw}) \ . \tag{3.11}$$

where m_w is the cluster centers of the winner cluster. σ_{iw} is the variance of the winner cluster in i^{th} dimension. α is the learning rate.

Steps 1 and 2 are iterated until the iteration converges or the number of iterations reaches a pre-specified value. The final cluster prototypes are the results of the SFCC.

After SFCC, we use the geometry mean of the membership value of the pre-defined fuzzy membership function in each of the dimension as the final

membership value of a data point. In our case, it is:

$$u_{ik} = (\prod_{j=1}^{a} u_{jik})^{1/a} \ . \tag{3.12}$$

where a is the the number of attribute, u_{jik} is the membership value of data instance x_k to cluster i in j^{th} dimension.

If hard-cut boundary is used, the data point is belonging to the cluster that gives the highest membership value.

3.3 Experiments

In this section, we perform some experiments to examine the performance of FCC and SFCC. The experiments focus on the clustering performance in seven respects, they are:

- **Experiment 1:** Data set with different number of samples.
- **Experiment 2:** Data set on different dimensionality.
- **Experiment 3:** Data set with different number of natural clusters inside.
- **Experiment 4:** Data set with different noise level.
- **Experiment 5:** Clusters with different geometry size.
- **Experiment 6:** Clusters with different number of data instances.
- **Experiment 7:** Performance on real data set.

Among all of the above experiments, the first three experiments are aimed to examine the time complexity of the proposed clustering algorithm. After that, the fourth to seventh experiments are aimed to examine the clustering

accuracy of the proposed clustering algorithms.

All the clustering algorithms used here are coded in MATLAB, and conducted on an Ultra Sparc 5 machine.

After each of the experiments, we compare the results of FCC and SFCC with some other common clustering algorithms. These algorithms include, *competitive learning (CL)*, *rival penalized competitive learning (RPCL)*, *fuzzy c means clustering (FCM)* and *k-means clustering (KM)*.

3.3.1 Experiment 1: Data set with different number of samples

In experiment 1, we test the clustering algorithms with data set having different number of sample points. We want to know the time complexity under these situations.

Motivation

Usually, the time complexity of a typical clustering algorithm grows exponentially with the number of data samples. However, multimedia databases usually have a huge amount of sample points. In such a case, we need a clustering algorithm in linear time complexity with the number of sample points.

In this experiment, we test our algorithms under the data set with different number of sample points to check if they are linear time complexity with number of sample points.

Experimental Setting

In experiment 1, we test our method with synthetic data sets in the Gaussian mixture distribution, whose probability density function can be written as follows:

$$p(\vec{x}) = \sum_{j=1}^{n} \alpha_j G(\vec{x}, \vec{m}_j, \Sigma_{\mathcal{X}_j}), \tag{3.13}$$

where n is the number of mixtures. Each weight, $\alpha_j \geq 0$ and $\sum_{j=1}^{n} \alpha_j = 1$, and each $G(\vec{x}, \vec{m}_j, \Sigma_{\mathcal{X}_j})$ is a single Gaussian function with the mean, $\vec{m}_j$ and the covariance matrix, $\Sigma_{\mathcal{X}_j}$.

In our experiments, we use the equal weight for each Gaussian mixture as follows:

$$\alpha_1 = \alpha_2 = \cdots = \alpha_n = \frac{1}{n}. \tag{3.14}$$

We also use a diagonal matrix as the covariance matrix of each Gaussian function:

$$\Sigma_{\mathcal{X}_i} = \begin{bmatrix} \sigma_i & 0 & \cdots & 0 \\ 0 & \sigma_i & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \sigma_i \end{bmatrix}. \tag{3.15}$$

We set σ_i and n as random variables with range from 1 to 10 for generating the testing data set.

We randomly generate 3 different data sets in each of the 2-D, 3-D, 5-D, and 10-D cases with the above setup and a fixed number of instances for each of the data sets. Then we apply the clustering algorithms on them to obtain the results.

Experiment Results

After the clustering, we mark down the time it needs to finish clustering. The results are summarized in Table 3.1-Table 3.4, Figure 3.1, and Figure 3.2.

From the experimental results, we conclude that Fuzzy c-means and Fuzzy Competitive Clustering have the highest complexity when the number of data instances is large. k-means algorithm seem to have an exponential increase in computational time to the number of instances. However, as it takes little number of iterations, the computational time is still small for k-means algorithm.

For those competitive-based clustering algorithms, Competitive Learning, Rival Penalized Competitive Learning, and Sequential Fuzzy Competitive Clustering, they seem to be less sensitive to the number of instances. So, we conclude that competitive based clustering algorithms are more suitable for database with huge amount of samples inside.

Under high dimensional data, the computational time for Competitive Learning and Rival Penalized Competitive Learning become more unpredictable. This shows the evidence that the computational time for them are not very related on the number of instances. Actually, we can prove in the later experiments that the computational time for them are mainly related on the distribution and the dimensionality of the data set.

3.3.2 Experiment 2: Data set on different dimensionality

In experiment 2, we test the clustering algorithms under data sets with different dimensionality.

	300	**1500**	**3000**	**6000**	**9000**
FCM	20.5591	50.2018	79.1989	171.9868	334.5710
KM	0.4884	2.6896	3.2299	7.5796	8.4775
CL	0.3598	0.4066	0.4249	0.6029	0.7944
RPCL	**0.2114**	**0.4034**	**0.5344**	**0.5483**	**0.7024**
FCC	7.3169	29.0409	53.5846	151.9438	270.2041
SFCC	0.9213	0.9914	1.0025	1.1132	1.2411

Table 3.1: The average time used (in second) for clustering data set in two dimensions with different number of data instances in Experiment 1.

	300	**1500**	**3000**	**6000**	**9000**
FCM	19.9911	94.8020	147.2157	422.6909	504.0983
KM	0.6086	2.7982	5.7552	8.2314	10.4216
CL	**0.2823**	1.1953	1.3771	2.5114	2.4085
RPCL	0.5299	**1.1115**	**0.9497**	**1.3662**	**1.5093**
FCC	7.1366	33.7620	62.7409	181.1739	367.9844
SFCC	1.2321	1.5268	1.5768	1.7702	1.8312

Table 3.2: The average time used (in second) for clustering data set in three dimensions with different number of data instances in Experiment 1.

	300	**1500**	**3000**	**6000**	**9000**
FCM	13.6461	131.2552	348.2287	860.7690	1526.1280
KM	1.7986	6.8853	12.3620	25.2378	32.5269
CL	**1.2405**	5.7927	4.8121	6.9584	13.3065
RPCL	1.7119	4.7748	3.8774	9.9629	2.7767
FCC	5.3039	36.7250	108.3956	214.4625	527.4522
SFCC	1.8356	**1.9488**	**2.0623**	**2.2554**	**2.5156**

Table 3.3: The average time used (in second) for clustering data set in five dimensions with different number of data instances in Experiment 1.

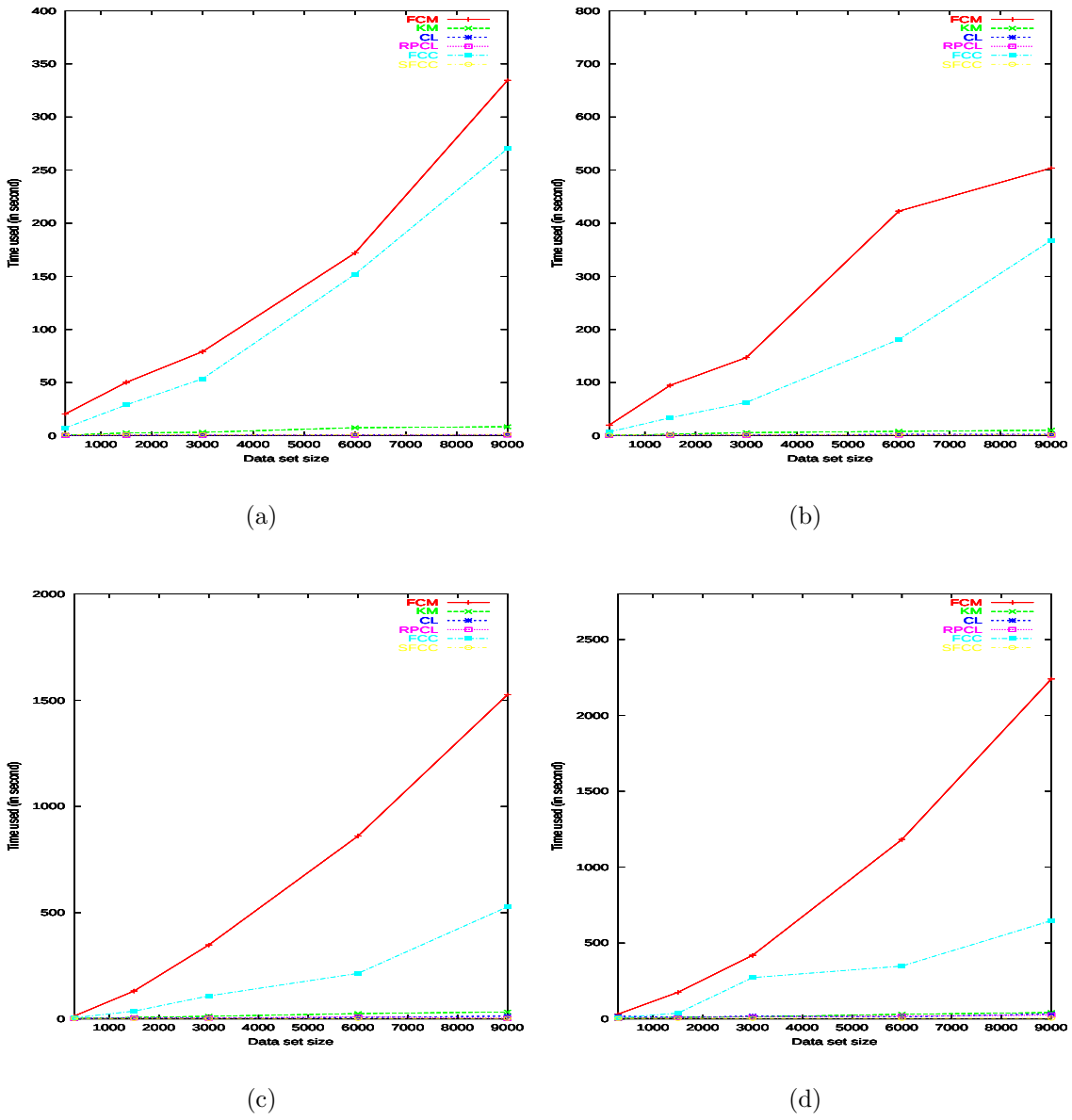


Figure 3.1: Results of Experiment 1. (a), (b), (c), and (d), the time needed for clustering different number of data instances under 2-D, 3-D, 5-D, and 10-D respectively.

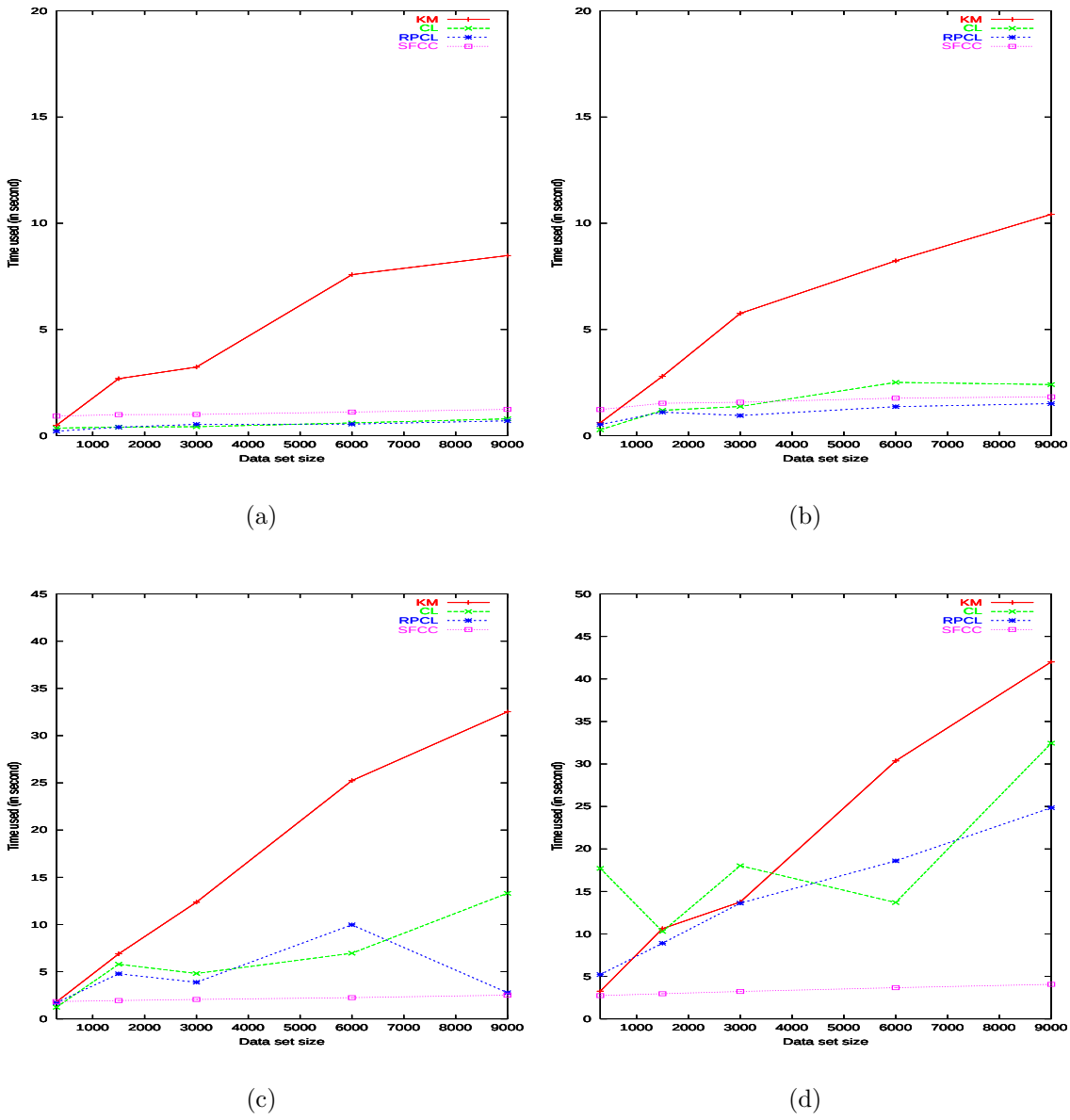


Figure 3.2: Results of Experiment 1. (a), (b), (c), and (d), the time needed for clustering different number of data instances under 2-D, 3-D, 5-D, and 10-D respectively.

	300	**1500**	**3000**	**6000**	**9000**
FCM	31.6585	175.1086	419.2682	1182.5000	2240.1000
KM	3.2584	10.6250	13.7535	30.3643	42.0067
CL	17.7092	10.2882	18.0297	13.6982	32.4412
RPCL	5.2192	8.9104	13.6194	18.5913	24.8490
FCC	7.1643	38.1214	272.2166	347.8854	648.0155
SFCC	**2.7462**	**2.9621**	**3.2310**	**3.6732**	**4.0695**

Table 3.4: The average time used (in second) for clustering data set in ten dimensions with different number of data instances in Experiment 1.

Motivation

Dimensionality [72, 73] of the data set is another issue that affects the complexity very much. Many clustering algorithms have an exponential computational time that varies with the dimensionality of the data set. However, in a typical multimedia database, data are existed in high dimension. As a result, we need to have a clustering algorithm that spends acceptable computational time under high dimensional data.

Experiment Setting

Similar to Experiment 1, we use synthetic data sets in the Gaussian mixture distribution to test our clustering algorithms in Experiment 2.

In Experiment 2, we randomly generate 3 different data sets with 1,500, 3,000, 6,000, and 9,000 data instances under dimensionality of 2-D, 3-D, 5-D and 10-D. Then, we apply the clustering algorithms on the data sets and obtain the results.

Experimental Results

After the clustering, again, we mark down the time it needs to finish clustering. The results are summarized in Table 3.5, Table 3.6, Table 3.7, Table 3.8, Figure 3.3, and Figure 3.4.

From the results of Experiment 2, we conclude that the computational time for all the clustering algorithms increase as the dimensionality of the data sets increase.

Among all of the tested clustering algorithms, those non-competitive based clustering algorithms, including Fuzzy c-means, k-means, and Fuzzy Competitive Clustering algorithm, are more sensitive to the data dimensionality than those competitive based clustering algorithms.

Non-competitive based clustering algorithms seem to be more sensitive to data dimensionality because they need to calculate all the distance pairs between cluster center and data point in every iteration. On the other hand, given k clusters, competitive based clustering algorithm just need to calculate k distance pairs in each iteration. Because the computational complexity for each distance calculation increase with the dimensionality, the overall computational time of the clustering algorithm increase with the dimensionality.

Among all the tested clustering algorithms, our proposed algorithm, Sequential Fuzzy Competitive Clustering seems to be the least sensitive one. The reason is because in SFCC, it uses membership value instead of the norm-distance. According to Section 3.2, we know the computational complexity of membership value is smaller than norm-distance; hence, SFCC seems to be less sensitive to the increase of dimensionality.

	2-D	**3-D**	**5-D**	**10-D**
FCM	50.2018	94.8020	131.2552	175.1086
KM	2.6896	2.7982	6.8853	10.6250
CL	0.4066	1.1953	5.7927	10.2882
RPCL	**0.4034**	**1.1115**	4.7748	8.9104
FCC	29.0409	33.7620	36.7250	38.1214
SFCC	0.9914	1.5268	**1.9488**	**2.9621**

Table 3.5: The average time used (in second) for clustering data set with 1,500 data instances under different dimensionality in Experiment 2.

	2-D	**3-D**	**5-D**	**10-D**
FCM	79.1989	147.2157	348.2287	419.2682
KM	3.2299	5.7552	12.3620	13.7535
CL	**0.4249**	1.3771	4.8121	18.0297
RPCL	0.5344	**0.9497**	3.8774	13.6194
FCC	53.5846	62.7409	108.3960	272.2166
SFCC	1.0025	1.5768	**2.0623**	**3.2310**

Table 3.6: The average time used (in second) for clustering data set with 3,000 data instances under different dimensionality in Experiment 2.

	2-D	**3-D**	**5-D**	**10-D**
FCM	171.9868	422.6909	860.7690	1182.5000
KM	7.5796	8.2314	25.2378	30.3643
CL	0.6029	2.5114	6.9584	13.6982
RPCL	**0.5483**	**1.3662**	9.9629	18.5913
FCC	151.9438	181.1739	214.4625	347.8854
SFCC	1.1132	1.7702	**2.2554**	**3.6732**

Table 3.7: The average time used (in second) for clustering data set with 6,000 data instances under different dimensionality in Experiment 2.

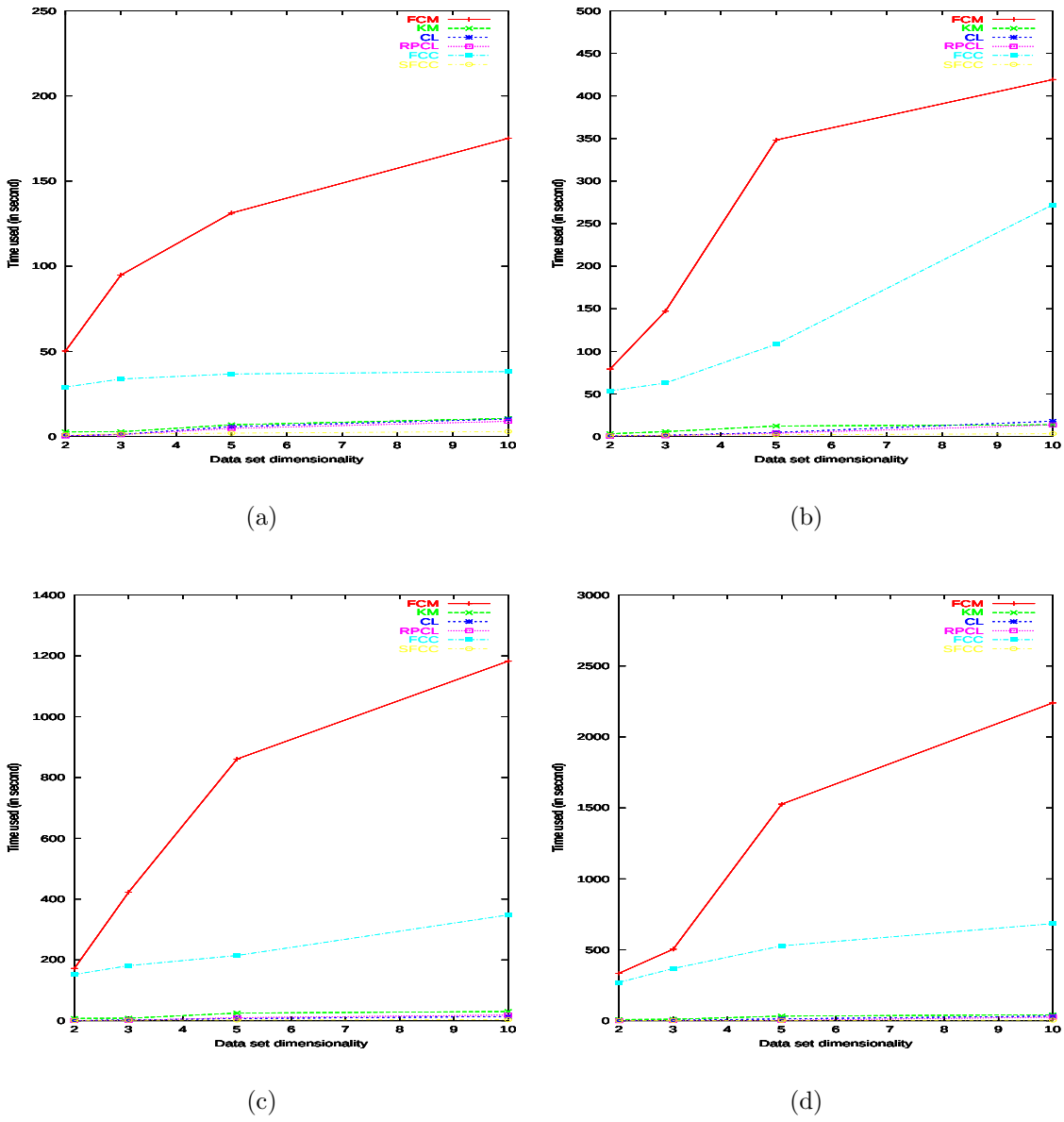


Figure 3.3: Results of Experiment 2. (a), (b), (c), and (d), the time needed for clustering data under different dimensionality with number of instances equal to 1,500, 3,000, 6,000, and 9,000 respectively.

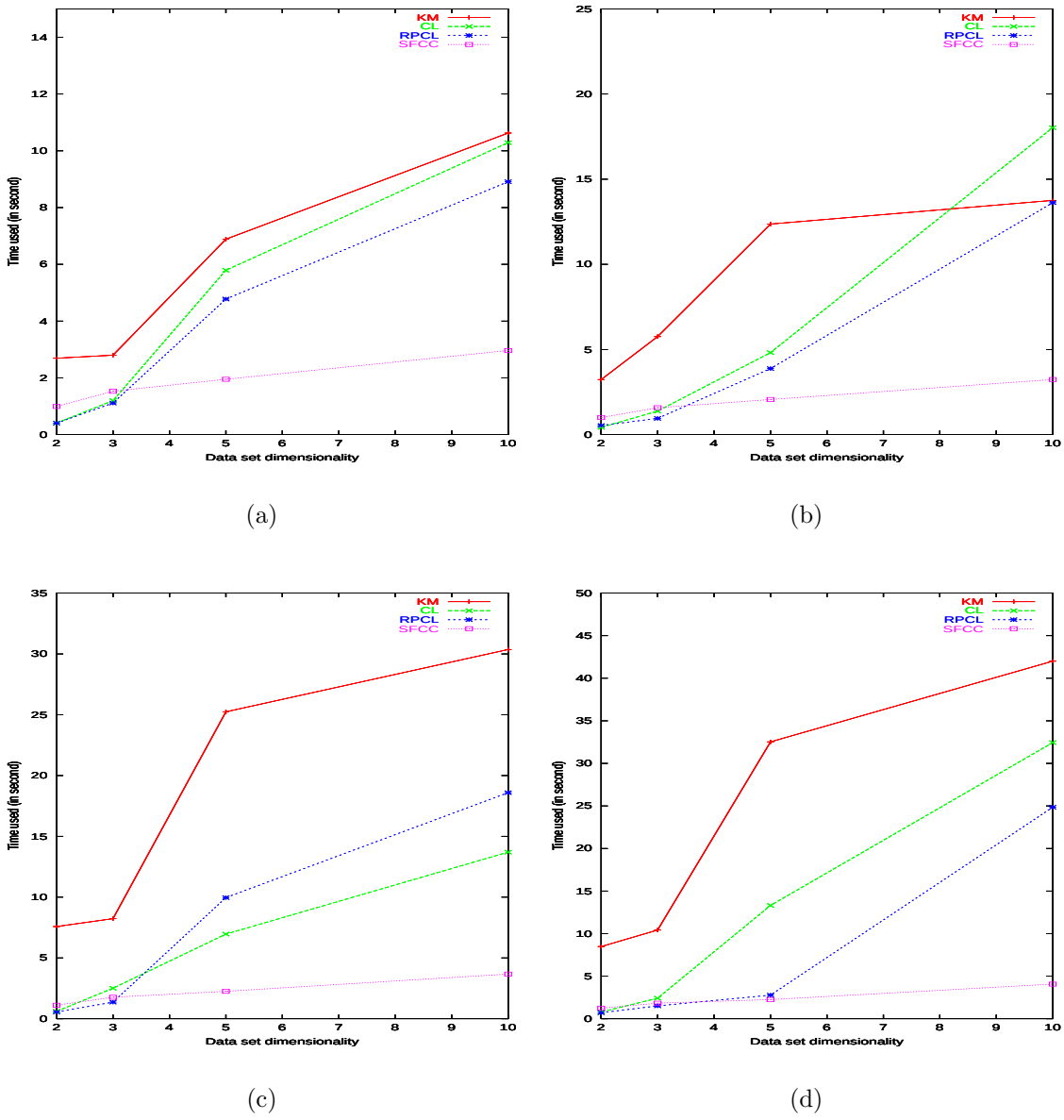


Figure 3.4: Results of Experiment 2. (a), (b), (c), and (d), the time needed for clustering data under different dimensionality with number of instances equal to 1,500, 3,000, 6,000, and 9,000 respectively.

	2-D	**3-D**	**5-D**	**10-D**
FCM	334.5710	504.0983	1526.1280	2240.1000
KM	8.4775	10.4216	32.5269	42.0067
CL	0.7944	2.4085	13.3065	32.4412
RPCL	**0.7024**	**1.5093**	2.7767	24.8490
FCC	270.2041	367.9844	527.4522	684.0155
SFCC	1.2411	1.8312	**2.2554**	**4.0695**

Table 3.8: The average time used (in second) for clustering data set with 9,000 data instances under different dimensionality in Experiment 2.

3.3.3 Experiment 3: Data set with different number of natural clusters inside

Motivation

Besides data set size and dimensionality, the number of clusters that the clustering algorithms need to find is also another important factor affecting the time complexity of the clustering algorithms. In Experiment 3, we test our clustering algorithms under data sets with different number of natural clusters inside. Within each of the data sets, suppose it has k clusters, we cluster the data set exactly into k clusters to check the time complexity of the clustering algorithms.

Experiment Setting

Similar to Experiment 1, data sets used here are synthetic data sets in the Gaussian mixture distribution. The dimensionality of these data sets are fixed to 10 and the number of data points in each of the data sets are fixed to 5,000.

The number of natural clusters in these data sets vary from 5 to 25. In each of the test cases, assumes there are k natural clusters inside, we then cluster the data set into k clusters and then mark down the time used to complete

the clustering.

Experimental Results

The experimental results are summarized in Table 3.9 and Figure 3.5.

In our experimental results, we found that as the number of clusters increases, the execution time also increases. In each iteration, all the clustering algorithms need to calculate the distance pairs between each cluster center and data instances. So, when the number of candidate clusters increases, the distance pairs needed to calculate increase also. This increase in distance calculation makes the overall computation time increases.

Those non-competitive based clustering algorithms seem to have a sharper increase in computation time as well as the number of candidate clusters. Suppose, there exist n data instances in the data set. When we increase one more candidate cluster in non-competitive based clustering, it increases n distance calculations in each of the iterations. On the other hand, competitive based clustering algorithms such as, CL, RPCL, and SFCC, just increase one more distance calculation in each of the iterations. So, competitive based clustering algorithms are less sensitive than non-competitive based clustering algorithms to the number of candidate clusters.

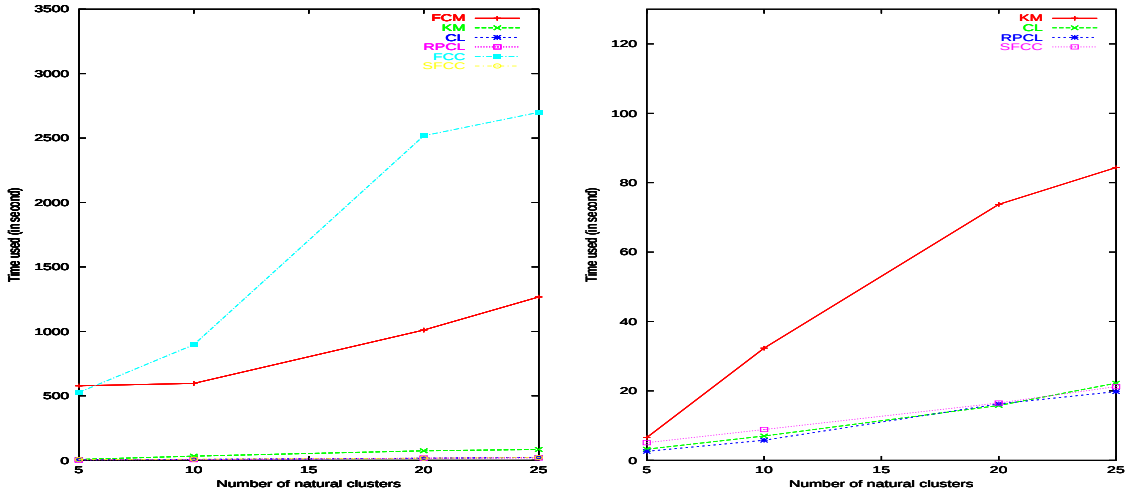
3.3.4 Experiment 4: Data set with different noise level

Motivation

From Experiment 1 to Experiment 3, we are concerning the computational complexity of the proposed clustering algorithms. However, start from Experiment 4, we are focusing on the clustering accuracy.

	5 clusters	**10 clusters**	**20 clusters**	**25 clusters**
FCM	578.2570	596.8290	1010.3300	1267.4
KM	6.5468	32.3214	73.7572	84.3761
CL	3.2040	7.0011	**15.7868**	22.2145
RPCL	**2.5823**	**5.7845**	16.2451	**19.7950**
FCC	526.8512	896.7460	2518.2349	2699.6831
SFCC	5.0724	8.8657	16.5210	21.2301

Table 3.9: The average time used (in second) for clustering 10-D data set with 5,000 data instances and different number of clusters in Experiment 3.



(a) (b)

Figure 3.5: Results of Experiment 3. (a), and (b), the time needed for clustering 10 dimensional data with 5,000 data instances with different number of natural clusters inside.

In a typical database, data are not noiseless. For most of the clustering algorithms, clustering accuracy decreases as the noise level of the data. Even worse that, some of the clustering algorithms trapped in local optimal solution when the noise level is high. For a multimedia database, the noise level is even higher than normal database does. So, clustering algorithms deal with multimedia database should able to overcome this situation.

In this experiment, we examine our clustering algorithms on data sets with different noise level. The aim of this is to prove that our proposed algorithms work well under noisy data.

Experiment Setting

Similar to Experiment 1, data sets in this Experiment are synthetic data sets in the Gaussian mixture distribution. Here are the setting for the data sets used in Experiment 4:

1. The data sets are 10 dimensional data.
2. The total number of data instances are 1,000.
3. There are total 4 clusters.
4. The number of instances in all data sets are the same.
5. σ_i used for each cluster is fixed at 0.3.
6. Norm distance between any two clusters is 2.
7. We generate random noise from 3% to 20% of the total number of data instances.

After we cluster the data set, we mark down the execution time and the error percentage for each of the data sets as the experimental results.

Experimental Results

We summarize the experimental results in Table 3.10, Table 3.11, and Figure 3.6.

From the experimental results, we conclude the experiment in the following respects:

- **Computational Complexity**

  The computational complexity seems not very sensitive to the noise level. According to the computational complexity in big-Oh notation, none of the tested clustering algorithms have count the term "*noise level*" in. So, it is expected that the noise level of the data set does very little effect on the computational complexity.

  However, when the noise level of the data set increase to some extent, the overall running time for clustering slightly decrease. This is because the noise we generated are random noise. When the number of random noise increase, the data set appears more likely to an uniform distribution. On the other hand, all the tested algorithms having the properties that it stops when all the clusters center are standing at the mean of the cluster's data. Uniform distributed data are just fulfilling this requirement. So, the number of total iterations for clustering decrease when the random noise level increase to some extent. This decrease in number of total iterations finally reflects on decrease in the overall running time.

- **Clustering Accuracy**

  As what we expected, the accuracy decrease when the noise level increase. When the noise level increase, the cluster's characteristic decrease, or in

other words, covered by the noise. As a result, it is more difficult for a clustering algorithm to locate the real underlying cluster characteristic.

Among all the tested clustering algorithms, we found that SFCC algorithm are the least sensitive one to noise. Refer to Section 3.2, the attracting power for a data point to a cluster prototype is proportional to the membership value of this data point towards the cluster prototype. For distant data point, its membership is small. Thus, the attracting power for a distant data point is small too. This feature enables cluster prototypes in SFCC to prevent from being attracted from noise or outliers.

On the other hand, in all the other tested algorithms and most of the traditional clustering algorithms, the attracting power for a data point to a cluster prototype (or cluster center) does not decrease as the distance between them increase. As a result, those outliers can pull the cluster prototypes with the same attracting power as those cluster's data do. This is why other clustering algorithms much more suffer from the noise data than SFCC does. As FCM use the distance ratio between data points and cluster prototypes to define the attracting power, it is the most sensitive clustering algorithm among all of the tested algorithms. Because the ratio is equally divided into k candidate clusters, which is a very high value. Section 3.4.1 gives more details on this problem.

3.3.5 Experiment 5: Clusters with different geometry size

Clusters used in Experiment 5 having a large different in variance (σ_i). It means they have a large difference in their geometry size. It is aimed to see

	3 %	**5 %**	**10 %**	**15 %**	**20 %**
FCM	27.0217	26.7168	26.3275	27.1133	26.2898
KM	4.6742	4.3863	3.2213	2.7421	2.4890
CL	1.6739	1.6848	**1.6109**	1.3986	**1.3827**
RPCL	**1.6109**	**1.3986**	1.6848	**1.3872**	1.6763
FCC	25.8411	27.1746	27.7135	26.9207	25.9584
SFCC	3.5654	3.5702	3.5731	3.5811	3.5802

Table 3.10: The average time used (in second) for clustering 10-D data set with 1,000 data instances and different percentage of noise data in Experiment 4.

	3 %	**5 %**	**10 %**	**15 %**	**20 %**
FCM	0.037	0.044	0.290	0.285	0.342
KM	**0.030**	**0.043**	0.119	0.140	0.214
CL	0.033	0.119	0.151	0.161	0.234
RPCL	0.039	0.164	0.207	0.279	0.255
FCC	0.042	0.067	0.070	0.112	0.142
SFCC	0.042	0.047	**0.060**	**0.062**	**0.082**

Table 3.11: The average error (in percentage) for clustering 10-D data set with 1,000 data instances and different percentage of noise data in Experiment 4.

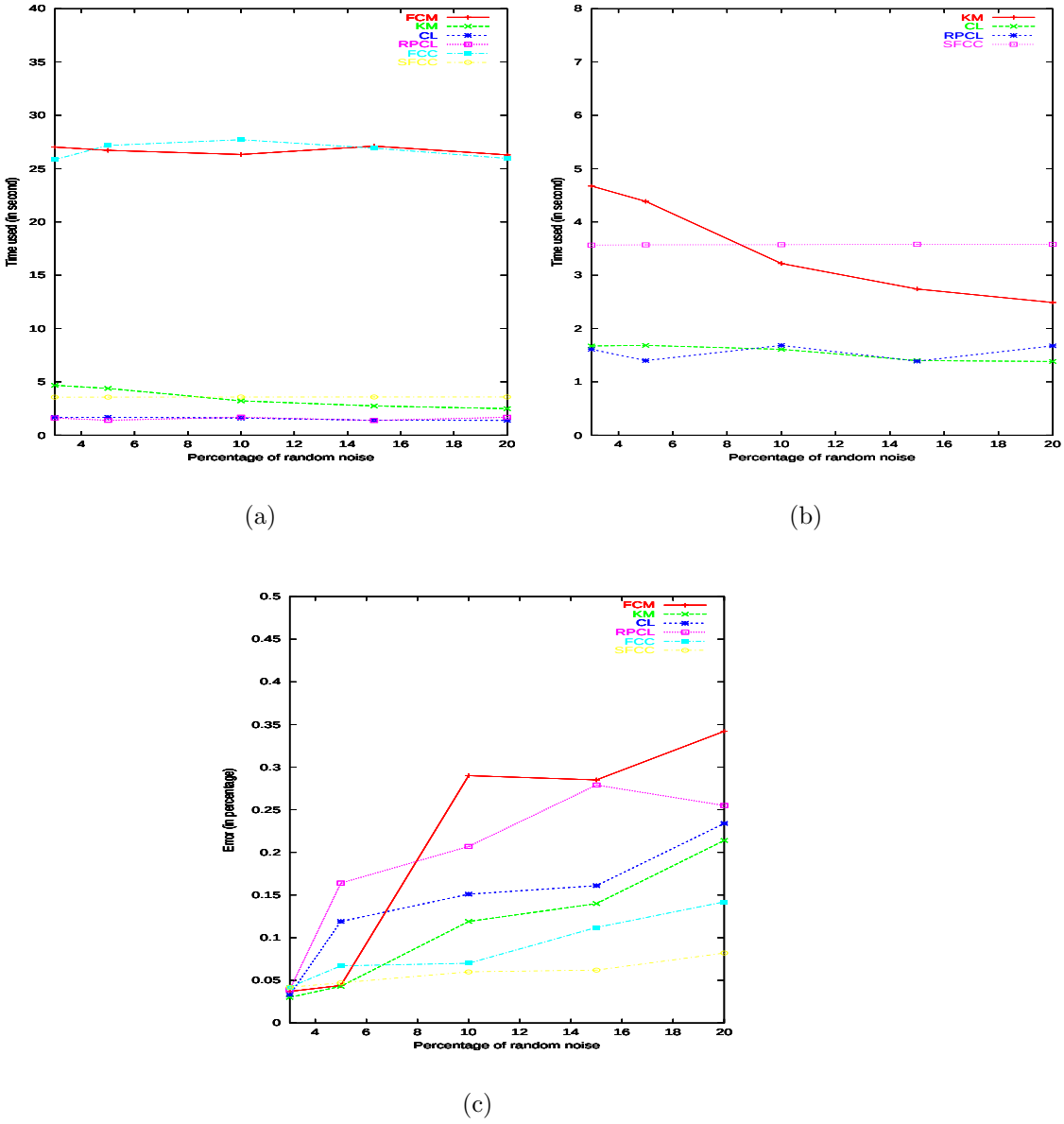


Figure 3.6: Results of Experiment 4. (a), and (b), the time needed for clustering 10 dimensional data with 1,000 data instances with differnt percentage of noise data. (c), the error percentage under the above setting.

the behaviors of FCC and SFCC under this kind of data sets.

Motivation

In traditional clustering algorithms, they assume all the clusters are in the shape of hyper-sphere. Within these hyper-spheres, they all have the same radius. As a result, to calculate the cluster label for a particular data point, it is the same to calculate the distance between this datum and all the cluster centers, the one with the smallest distance is the cluster that the datum belongs to. However, we know that clusters are not always with the same radius, or even not in the shape of hyper-sphere. Thus, traditional clustering algorithms seem to be weak in facing this kind of problems.

In this experiment, we want to know the performance of FCC and SFCC under data sets with large variance difference between the clusters inside.

Experiment Setting

Similar to Experiment 1, data sets in this experiment are synthetic data sets in Gaussian mixture distribution. Here are the setting for the data sets used in Experiment 5:

1. The data sets are 10 dimensional data.
2. The total number of data instances are 1,000.
3. There are total 2 clusters.
4. The number of instances in all the data sets are the same.
5. σ_i used for the smaller cluster is fixed at 0.2.
6. Norm distance between any two clusters is 1.

7. We define the ratio of σ_i between the larger cluster and smaller cluster is α, and it varies from 2 to 10 in this experiment.

After we cluster the data sets, we mark down the execution time and the error percentage for each of the data sets as the experimental results.

Experimental Results

We test the FCC and SFCC in the same way as Experiment 4. The experimental results are summarized in Table 3.12, Table 3.13, and Figure 3.7.

From the experimental results, we conclude the experiment in the following respects:

- **Computational Complexity**

  As α increase, we found that the overall computational time slightly increases. Actually, the computational complexity for each iteration in all the clustering algorithms is not related to the α value. So, the overall computational time is theoretically unrelated to α. However, as the data distribution seems to be more complex when α is large, all the clustering algorithms need more iterations in order to converge. So, the overall computational time slightly increases as α.

- **Clustering Accuracy**

  The clustering accuracy for all the clustering algorithms decrease as α increase. it is very easy to understand why α affect the clustering accuracy like this. As α increase, the complexity of the data distribution increase too. As a result, the clustering accuracy decreases.

  Among all the algorithms being tested, we found that FCC and SFCC perform the best under small α value. However, when α becomes larger

	2	**4**	**6**	**8**	**10**
FCM	31.6569	31.7875	30.9754	34.2668	34.4200
KM	2.2530	2.5643	3.1370	3.5130	3.8254
CL	1.5389	1.7966	2.0512	3.0904	3.2359
RPCL	**0.7963**	**1.6341**	**1.6036**	**1.8794**	3.0973
FCC	10.2384	11.5259	11.3135	15.2393	15.1950
SFCC	2.0912	2.1036	2.0755	2.0986	**2.1061**

Table 3.12: The average time used (in second) for clustering 10-D data set with 1,000 data instances and different α value in Experiment 5.

	2	**4**	**6**	**8**	**10**
FCM	0.003	0.063	0.109	0.142	**0.152**
KM	0.003	0.103	0.163	0.201	0.237
CL	0.009	0.037	0.120	0.167	0.198
RPCL	0.013	0.071	0.131	0.198	0.229
FCC	0.002	0.038	**0.056**	0.110	0.320
SFCC	**0.001**	**0.014**	0.097	**0.107**	0.282

Table 3.13: The average error (in percentage) for clustering 10-D data set with 1,000 data instances and and different α value in Experiment 5.

and larger, the error percentage increases faster and faster. When α becomes extremely high, FCC and SFCC even have the highest error percentage.

According to Section 3.4.1, FCC and SFCC are not needed to obey Equation 3.16. As a result, when the relative distance between two cluster centers are too small, unreasonable high membership value may assign to data. This unreasonable high membership value finally reflects on the decrease of clustering accuracy. So, when the relative distance between two cluster centers is too small, the error percentage would even higher than those of traditional clustering algorithms.

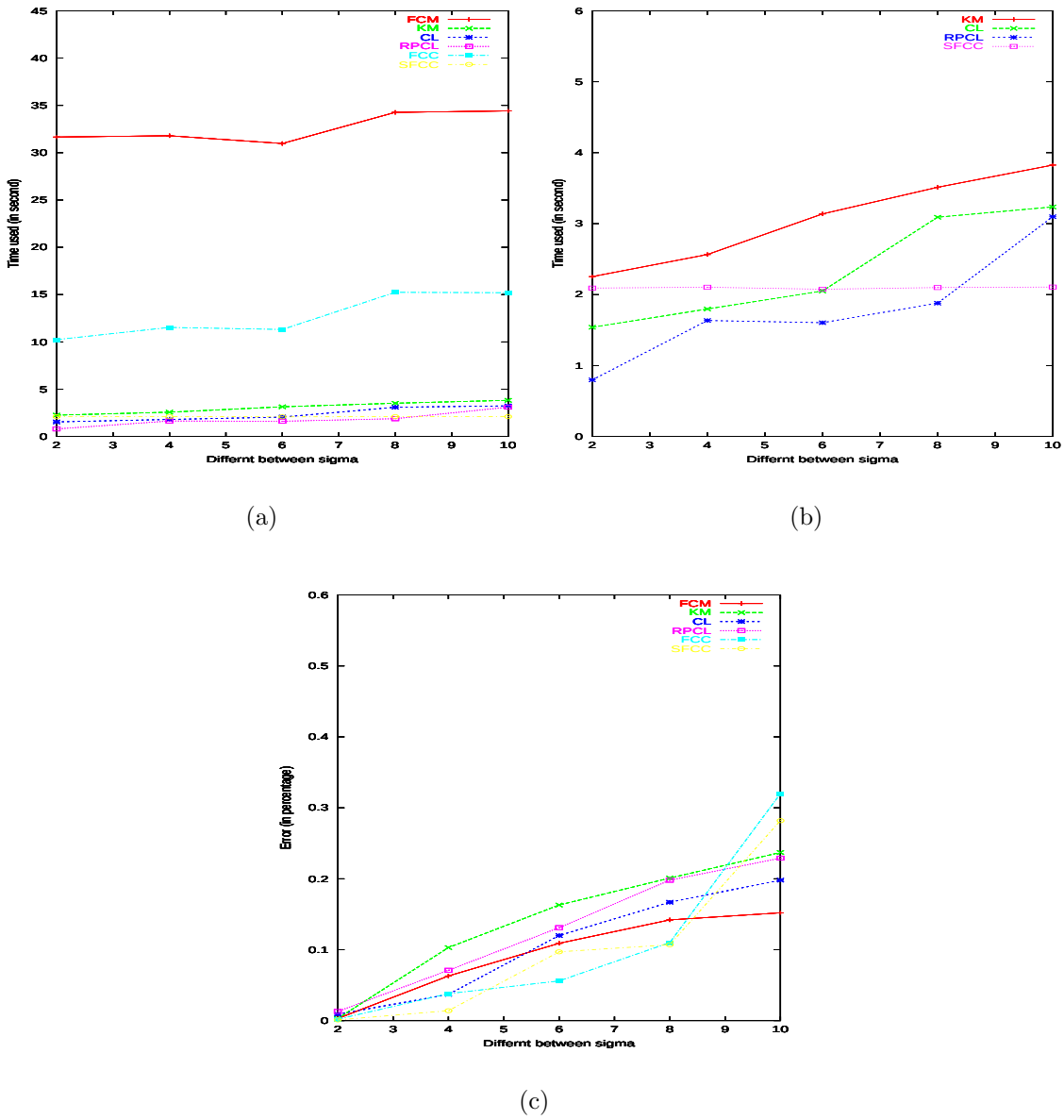


Figure 3.7: Results of Experiment 5. (a), and (b), the time needed for clustering 10 dimensional data with 1,000 data instances and different α value. (c), the error percentage at different α value under the above setting.

3.3.6 Experiment 6: Clusters with different number of data instances

In all the above experiments, the number of data instances are the same in all the clusters. In this experiment, we are now examining the proposed algorithms under data sets with different number of data instances in different cluster.

Motivation

For a typical real data set, usually, we do not have any idea on how much data instances should be included in a particular cluster. So, it is unacceptable for an algorithm that can perform well only when all the clusters having the same number of data instances. In this experiment, we are try to show that our proposed clustering algorithms not only work fine for clusters having the same number of data instances, but it also works well when the number is not equal.

Experiment Setting

Similar to experiment 1, data sets in this Experiment are synthetic data sets in Gaussian mixture distribution. Here are the setting for the data sets used in Experiment 6:

1. The data sets are 10 dimensional data.
2. The total number of data instances are 1,000.
3. There are total 2 clusters.
4. σ_i are fixed to 0.2 for all the clusters.
5. The number of instances in the cluster with less data points is fixed to 100.

6. Norm distance between any two clusters is 1.

7. The ratio of the number of data points between two clusters is defined as β.

8. We vary the value of β from 3 to 6 in Experiment 6.

After we cluster the data set, we mark down the execution time and the error percentage for each of the data sets as the experimental results.

Experimental Results

We test the FCC and SFCC in the same way as Experiment 4. The experimental results are summarized in Table 3.14, Table 3.15, and Figure 3.8.

From the experimental results, we conclude the experiment in the following respects:

- **Computational Complexity**

  The running time of all the tested algorithms seems to be unrelated to β. Because the computational complexity for each iteration is not related to β. So, the computational time in each iteration keeps constant no matter how β varies. Also, the value of β does not affect the complexity of the data distribution. So, the total number of iteration is also unrelated to β. Combining these two factors, we conclude that the overall Computational Complexity is unrelated to the value of β. So, the overall execution time is unrelated to the value of β as well.

- **Clustering Accuracy**

  From Figure 3.8, we found that the error percentage of all the tested clustering algorithms increase as β increases. As we known, cluster centers tend to be staying at the mean of data points. So, if the data points

	3	**4**	**5**	**6**
FCM	11.8113	12.7360	11.9885	11.5392
KM	1.2475	1.5076	1.2575	1.9727
CL	1.0863	**1.1717**	1.2930	1.1023
RPCL	**0.9263**	1.3347	**1.0168**	**1.0504**
FCC	11.1114	11.4747	11.1365	11.3311
SFCC	3.5834	3.5980	3.6164	3.6274

Table 3.14: The average time used (in second) for clustering 10-D data set with 1,000 data instances and different β value in Experiment 6.

in each cluster are different, cluster centers would shift to those clusters with large number of data instances. This misplacing in cluster center results in the clustering error. Larger the difference (β), more serious in clusters misplacing, and thus higher error percentage.

Among all the clustering algorithms, k-means and SFCC seem to be the least sensitive to β. For the k-means algorithm, as it uses hard-cut approach in calculation, it is hard for cluster centers in k-means algorithm being attracted by data instance which does not belong to its cluster.

In the case of SFCC, the attracting power for a data point to a cluster prototype is proportional to the membership value of this data point towards the cluster prototype. For a distant data point, its membership is small. Thus, the attracting power for a distant data point is small too. As a result, this small attracting power brings the similar effect as hard-cut in k-means algorithm.

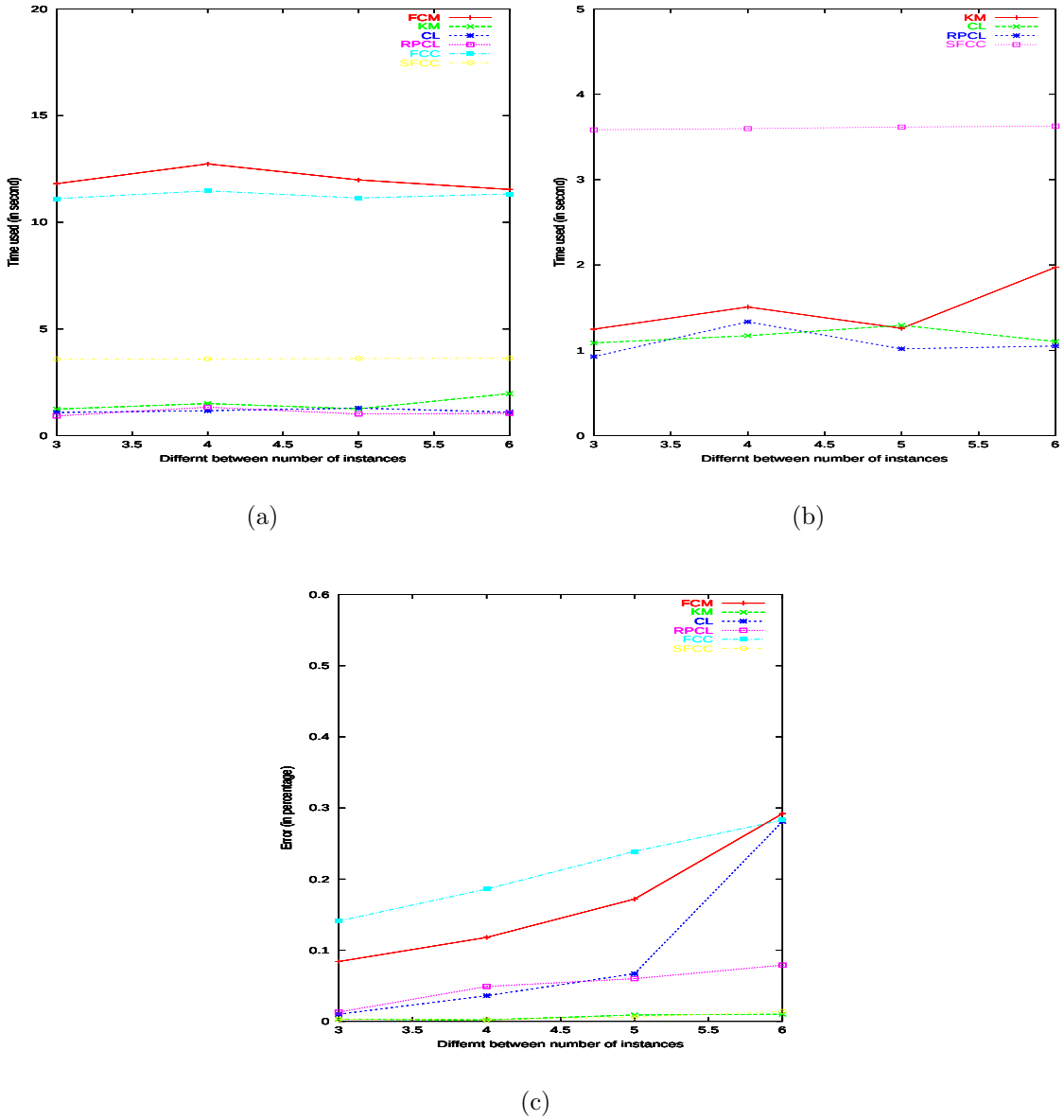


Figure 3.8: Results of Experiment 6. (a), and (b), the time needed for clustering 10 dimensional data with 1,000 data instances and different β value. (c), the error percentage at different β value under the above setting.

	3	**4**	**5**	**6**
FCM	0.084	0.118	0.172	0.292
KM	**0.003**	**0.002**	0.009	**0.010**
CL	0.010	0.036	0.067	0.281
RPCL	0.013	0.049	0.060	0.079
FCC	0.141	0.186	0.239	0.283
SFCC	**0.003**	**0.002**	**0.006**	0.014

Table 3.15: The average error (in percentage) for clustering 10-D data set with 1,000 data instances and and different β value in Experiment 6.

3.3.7 Experiment 7: Performance on real data set

Motivation

So far from Experiment 1, we only deal with synthetic data sets. However, it does not imply that the proposed clustering algorithms also work fine in real data. In this experiment, we use a very famous real data set, iris data set [74] to examine our proposed algorithms. We want to show our proposed algorithms also work fine in real data sets.

Experiment Setting

We use iris data set in this experiment. In iris data set:

1. There are total 150 data instances equally divided into 3 clusters.
2. There are total 4 attributes in the data set.
3. There exist overlapping between different clusters

Same as other experiments, we perform clustering on this data set and mark down the running time and error percentage.

FCM	KM	CL	RPCL	FCC	SFCC
3.2167	1.0390	0.8419	**0.7531**	1.9428	1.5500

Table 3.16: The average execution time (in second) for clustering iris data set with 150 data instances and 4 attributes in Experiment 7.

FCM	KM	CL	RPCL	FCC	SFCC
0.040	0.057	0.107	0.140	0.100	0.054

Table 3.17: The average error (in percentage) for clustering iris data set with 150 data instances and 4 attributes in Experiment 7.

Experimental Results

Our experiment shows that SFCC performing well under real data too. It gets the best balance between execution time and clustering accuracy. It is at the second position according to both speed and accuracy. However, other algorithms may have a great tradeoff between speed and accuracy. For example, Fuzzy c-means algorithm gets the lowest error percentage, however it takes the longest running time. Similarly, Competitive Learning takes the shortest running time but the highest error percentage. So, we conclude that SFCC works fine under typical real data set.

3.4 Discussion

3.4.1 Differences Between FCC, SFCC, and Others Clustering Algorithms

One of the great problem on Probabilistic Clustering algorithms is *Outliers*. Outliers are vectors, or called data points, in the data domain which are so distant from the rest of the other vectors in the data set, that it would be unreasonable to assign them high membership values to any of the c clusters.

Every Probabilistic Clustering algorithm obeys the constraint:

$$\sum_{i=1}^{c} u_{ik} = constant; \quad k = 1, ..., n \ . \tag{3.16}$$

where, u_{ik} is the membership value of the k^{th} instance to the i^{th} cluster.

However, the above constraint does not permit all the c memberships to assume value lower than $1/c$. For an outlier x_k, all the ratios d_{ik}/d_{jk} will often be close to unity, resulting that all the c membership values close to $1/c$. Because FCM and many other Probabilistic Clustering algorithms use the membership value as a weighting to calculate the cluster centroid. This unreasonable high membership value often causes improper positioning of the centroids. In fact, if an outlier is very distant, one of the centroids might position itself at the outlier's position.

On the other hand, possibilistic clustering also raise another problem. Use Possibilistic c means (PCM) algorithm [67, 68] as an example. The objective function for PCM can break down into a sum of c single objective functions. As a result, the centroids do not *affect* each other during the optimization process. This properties often leads to coincident clusters. Another problem for PCM is that the result of PCM depends heavily on initialization. The authors of [67] suggest to use FCM to initialize PCM. However, if an outlier is distant, PCM will not able to recover from the *bad* initial partition generated by FCM.

However, FCC and SFCC do not have the above problems. It is because in FCC and SFCC, we do not have the constraint as in Equation 3.16. Hence, we can assign a small value of u_{ik} to an instance, if it is needed to do so. Also, as every cluster prototype interacts with each other, FCC and SFCC would not generate *coincident clusters* like PCM. As we use fuzzy prototype to describe the cluster in FCC and SFCC, they can be used to find the information of the

	FCC	SFCC	FCM	KM	CL	RPCL
Fuzzy	Yes	Yes	Yes	No	No	No
Inter-cluster data	Yes	Yes	Yes	Yes	Yes	Yes
Intra-cluster data	Yes	Yes	No	No	No	No
Noise-Resistant	Yes	Yes	No	No	Yes	Yes
$\sum_{i=1}^{c} u_{ik} = 1$	Not Needed	-	Yes	Yes	-	-
Complexity in each iteration	$O(nkd)$	$O(kd)$	$O(nkd)$	$O(nkd)$	$O(kd)$	$O(kd)$
Knowledge on each dimension	Yes	Yes	No	No	No	No

Table 3.18: Comparison on the properties between *FCC*, *SFCC*, and several traditional clustering algorithms.

cluster in each dimension.

We summarize the feature of these clustering algorithms in Table 3.18. In the table, n is the number of total instances, k is the number of candidate clusters, and d is the dimensionality of the data.

3.4.2 Why SFCC?

In this section, we explain why SFCC is a fuzzy clustering algorithm suitable for multimedia database.

In a typical database, it usually consists of a huge amount of data instances in very a high dimension. So, the computational complexity of the algorithm should not be exponential with the number of instances or the dimensionality of the data. According to our experimental results, all the non-competitive based clustering algorithms having an exponential increase in computation complexity to the number of instances and dimensionality of the data. On the

other hand, competitive based algorithms give an acceptable complexity under huge amount of data instances and high dimensionality. This is the first point makes SFCC suitable for clustering in multimedia database.

In our experimental results, they also shows that SFCC having a better clustering accuracy than Competitive Learning and Rival Penalized Competitive Learning under noisy data and data in complex distribution. Noisy data in complex distribution is what a typical property in multimedia database.

In conclusion, SFCC is more suitable than other traditional clustering algorithms in clustering of multimedia database.

Chapter 4

Hierarchical Indexing based on Natural Clusters Information

In this chapter, we present a hierarchical indexing approach. We call the indexing structure generated from this approach *Sequential Fuzzy Competitive Clustering Binary Tree* (SFCC-b-tree). In the rest of this chapter, we will first present the advantages of hierarchical indexing approach and the details of SFCC-b-tree. Then, it is followed by a description on the searching method for the SFCC-b-tree and experiments on its performance analysis.

4.1 The Hierarchical Approach

In non-hierarchical indexing structure, usually there is no clear relationship between each level or node. This behavior makes us difficult to update the indexing structure and perform 100% nearest-neighbor search. In our indexing structure, we use a hierarchical approach such that relationship can be found in nodes between two levels. This feature enables us to update the indexing structure easily. Moreover, we can backtracking in the hierarchical structure in order to perform the 100% nearest-neighbor search.

The hierarchical approach divides the vector space into a sequence of nested

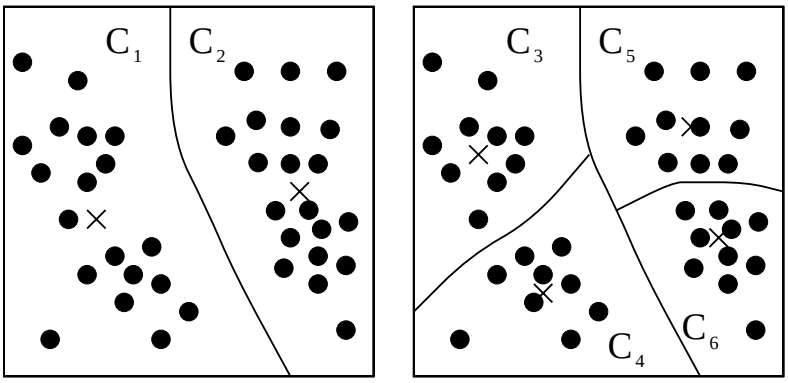


Figure 4.1: Cluster $C3$ and $C4$ are the sub-cluster of $C1$. Cluster $C5$ and $C6$ are the sub-cluster of $C2$. There is no overlapping area between those clusters in the same level. $C1$ and $C2$ in level one. $C3$, $C4$, $C5$, and $C6$ in level two.

clusters. Every cluster in the sequence is a subset of its parent cluster. Also, those clusters having the same parent cluster do not have any overlapping with each of the other. Figure 4.1 so the idea of the hierarchical nested clusters.

The hierarchical clustering approach can also be represented as follows. Let the feature vector set with n vectors be X, where

$$X = \{x_i\}_{i=1}^{n} \ .$$

A cluster, C, is a subset of vectors within X. The hierarchical approach breaks X into a sequence of clusters $C_1, C_2, C_3, ..., C_m$ satisfying the following:

$$C_i \cap C_j = \emptyset, \quad 1 \leq i, j \leq m, i \neq j \ ,$$

and

$$C_1 \cup C_2 \cup \ldots \cup C_m = X \ .$$

A cluster Y is nested into cluster Z if and only if every component exists in cluster Y can be found in cluster Z as well. A hierarchical clustering is a

sequence of clusters in which each cluster is nested into the previous cluster in the sequence.

After the hierarchical clustering, we use a mapping function to map the generated clusters into a binary tree structure. After the binary tree structure is generated, it supports nearest-neighbor search. At the top level, a nearest-neighbor query q is compared to the clusters in the immediate lower level. The cluster that q having the highest similarity will be selected. The elements in the selected cluster will be the result of the query q if they satisfy the criteria of being the nearest-neighbor search. Otherwise, the search and comparison will proceed to the lower levels until that we get a final result. On more details about the nearest-neighbor search, Section 4.2.5 present how to make use of a branch-and-bound algorithm to speed up the searching.

4.2 The Sequential Fuzzy Competitive Clustering Binary Tree (SFCC-b-tree)

In this section, we introduce the hierarchical SFCC binary tree. Also, we outline the procedure of how to use SFCC clustering algorithm to build the SFCC-b-tree.

Given a set of data set, we can use SFCC to generate two non-overlapping clusters within the data set. With these non-overlapping clusters (or subsets), we can generate the nested clusters structure very easily. The basic idea is that we apply SFCC on the data set and divide it into two non-overlapping clusters each time and then continue to do SFCC clustering on each of the subset hierarchically until each of the final sub-clusters contains a number of elements which less than a pre-specified number of elements. After we get all

these clusters, we are able to build the SFCC-b-tree easily.

4.2.1 Data Structure of SFCC-b-tree

In the SFCC-b-tree, there are two kinds of tree node. These two kinds of tree node are:

- The Leaf Node, and
- The Non-leaf Node.

The underlying definition of them is:

Definition 4.1 (Tree Node) *A Tree Node contains the following entries:*

1. ***Cluster Prototype:*** *The cluster prototype which is generated from SFCC clustering algorithm. This represents the distribution of the data instances within this subtree.*
2. ***Boundary Table:*** *It is the smallest hyper-cube (SHC) that encloses all the data instances in the subtree. The reason for a tree node includes this entry is we can make use of this hyper-cube to perform overlapping check. The nearest-neighbor search result can only exist in a tree node if there are overlapping between the hyper-cube of the query and the hyper-cube of the tree node.*
3. ***Number of Instances in this subtree:*** *Same as its name, it is the total number of instances exist in the subtree. The reason to include this entry in the tree node is not for speeding up the searching time. However, it is a convenience to keep the total number of instances in database maintenance, so we also include this entry in the tree node.*

4. ***Left Child Pointer:*** *If the tree node is a non-leaf node, then it is a pointer in tree node type which points to the left child of the tree node. If the tree node is a leaf node, then it is a NULL pointer.*

5. ***Right Child Pointer:*** *Having similar setting with the Left Child Pointer. However, Right Child Pointer is a pointer pointing to the right child of the tree node.*

6. ***Leaf pointer:*** *It is a pointer to indicate whether a tree node is a leaf node or not. For a leaf node, it is a pointer pointing to the data array. A data array contains a cluster of at most M data instances calculated by SFCC. M is the maximum number of data instances in a leaf node. For a non-leaf node, Leaf pointer is a NULL pointer.*

Based on the Definition 4.1, a SFCC-b-tree satisfies the following properties:

Property 4.1 Each leaf node contains between 1 and M data point(s).

Property 4.2 Each non-leaf node has two children.

Property 4.3 According to [75], it is easy to calculate N_i, the total number of instances in the subtree with root node i from its child nodes. Suppose node i has two child nodes $i+1$ and $i+2$, then N_i is given by:

$$N_i = N_{i+1} + N_{i+2} \ .$$

4.2.2 Tree Building of SFCC-b-Tree

After an introduction of the data structure of SFCC-b-tree, we present the algorithm for building the hierarchical binary tree by using SFCC clustering algorithm.

Given a set of data, we perform top-down SFCC clustering and build a SFCC-b-tree based on the clusters. The basic idea is that we apply SFCC to cluster the data set into two sub-clusters each time and then continue to do SFCC clustering hierarchically to each of the sub-clusters until each of the final sub-clusters contains less than a pre-specified number of data points. With these SFCC clusters, we can build a binary tree structure.

Algorithm 4.1 BuildTree(D, P, M)

▷ *Input: A set of data objects D, a SFCC-b-tree node P (P is empty at the first time), and the maximum node size M*

▷ *Output: A SFCC-b-tree*

```
1  if D's size is greater than M then do
2      create a non-leaf node Q
3      add Q as a child node of P if any
4      use SFCC to cluster D into two sub-sets D1 and D2
5      BuildTree(D1, Q, M)
6      BuildTree(D2, Q, M)
7      return Q
8  else
9      create a leaf node L for D
10     add L as a child node of P if any
11     return L
12 end if
13 calculate the node information of D and store it in the corresponding entry of P
```

4.2.3 Insertion of SFCC-b-tree

In this section, we present the idea of how to insert data instances into an already built SFCC-b-tree.

The basic idea of updating the SFCC-b-tree is to first find out where should the instance located. Then we either add or delete it from the tree. After this, we update the information of the node if it is needed.

Algorithm 4.2 Insert(T, P, M)

▷ *Input: A SFCC-b-tree T, a data instance want to insert into the SFCC-b-tree P, and the maximum node size M*

▷ *Output: An updated SFCC-b-tree*

1 $N \leftarrow$ the root node of T
2 **while** N is not a leaf node **do**
3 $N \leftarrow$ the node that gives the highest membership value to data instance P among its child nodes if any
4 **end while**
5 associate P to N
6 **if** N's size is larger than M **then do**
7 split the node N into 2 sub-nodes by using SFCC
8 **end if**
9 update the information of N's ancestors if necessary

The performance of the indexing tree for searching may be reduced after some individual data point insertions. The more the insertions, the worse the performance. The reason is that the insertion algorithm dose not fully consider the overall distribution of the inserted data point and the original data so that it cannot guarantee to keep the natural clusters. The searching performance will then be worse. As a result, we may have to rebuild the indexing structure after a certain amount of data points have been inserted.

4.2.4 Deletion of SFCC-b-Tree

The basic idea of deletion is similar to insertion. Algorithm 4.3 shows the algorithm for deleting an instance from a already built SFCC-b-tree.

Algorithm 4.3 Delete(T, P, M)

▷ *Input: A SFCC-b-tree T, a data instance want to delete from the SFCC-b-tree P, and the maximum node size M*

▷ *Output: An updated SFCC-b-tree*

```
1  N ← the root node of T
2  while N is not a leaf node do
3      N ← the node that gives the highest membership value to data instance P
       among its child nodes if any
4  end while
5  if P is associated with N then do
6      remove P from node N
7      update the information of N's ancestors if necessary
8      if the size of N's parent node less than M then do
9          merge all N parent node's child nodes
10     end if
11 end if
```

4.2.5 Searching in SFCC-b-Tree

In this section, we present the k-nearest-neighbor (k-NN) search algorithm for SFCC-b-tree. We also present the pruning rules used in nearest-neighbor search.

In our proposed indexing structure, tree nodes in the same tree level do not overlap with each other. Indexing structure having this property is very suitable to use the branch-and-bound algorithm proposed in [32] to compute

the k nearest neighbors to a given query. So, we apply a modified branch-and-bound algorithm for k-NN search in SFCC-b-tree.

The basic idea of the modified branch-and-bound algorithm consists of two stages. First, we divide the feature set into disjoint subsets hierarchically (by SFCC in our method). Then we order it in a tree structure. In the second stage, we test the node with the pruning rules and search it in a suitable order.

In SFCC-b-tree, every tree node is actually represents a natural cluster in the database. As a consequence, a data instance is more likely to locate in the node which gives the data instance a higher membership value. Follow this idea, we should search the data query from the tree node in the sequence from high membership value to low membership value, according to the membership value that the tree node gives the data query.

However, no matter what the order we used in k-NN search, if you search all the tree nodes before getting the results, the efficiency should be too low to accept. So, we apply a pruning rule in our branch-and-bound algorithm to prune out those nodes which are impossible to contain any query result in it. The rule is:

Rule 4.1 (General Exclusion Rule) *Given a tree node T, and a k-NN query q in d dimension, T does not contain any possible k-NN search solution if* $\forall\ j = 1, 2, \ldots, d$*:*

$$q_j - b \nleqslant Tc_j + Tb_j \nleqslant q_j + b \ ,$$

and

$$q_j - b \nleqslant Tc_j - Tb_j \nleqslant q_j + b \ .$$

where, q_j is the coordinate of q in dimension j, b is the boundary distance for k-NN search, Tc_j is the cluster center of T in dimension j, and Tb_j is half of the length of the smallest hyper-cube (Section 4.2.1) in dimension j get from the boundary table (Section 4.2.1) of tree node T.

The proof of Rule 4.1 can be easily shown in simple logic. Given any two hyper-cube, they can only have overlapping if and only if at least they have overlapping in any one of the dimension. Follow this idea, we can get the Rule 4.1.

Make use of Rule 4.1, we propose an algorithm to check if there overlap between tree node T and the query hyper-cube with the query q as the center and length $2b$, b is the k-NN search boundary distance, in each dimension.

Algorithm 4.4 OverlapTest(T, q, b)

▷ *Input: A SFCC-b-tree node T, a query point q in d dimenion, the k-NN search boundary distance b*

▷ *Output: A boolean value, it is TRUE if there exist overlap, FALSE if there does not exist overlap.*

▷ *Internal variable: Integer i, q_j is the coordinate of q in dimension j, Tc_j is the cluster center of T in dimension j, and Tb_j is half of the length of the smallest hyper-cube*

```
1  for i=1 to d do
2      if (q_j − b ⩽ Tc_j + Tb_j ⩽ q_j + b) or (q_j − b ⩽ Tc_j − Tb_j ⩽ q_j + b) then
       do
3          return TRUE
4      end if
5  end for
6  return FALSE
```

Traditional pruning algorithm uses L_2-distance as a measurement, as a result, the calculation involve all the d dimensions. In our pruning algorithm,

we only need to deal with all the d dimensions in the worst case. In average, we only need to run $d/2$ number of loops to get the boolean result. So, our proposed overlap check algorithm is more efficient on average.

Having the overlap checking algorithm, we make use of it and develop a searching algorithm for k-NN search. The algorithm are shown in Algorithm 4.5.

Algorithm 4.5 SFCC-knnSearch(Q, P, b)

▷ *Input: A query point Q, a SFCC-b-tree node P (P is the rootnode at the first time), the query boundary square b (b has infinite length at the first time)*

▷ *Output: The set of results for knn similar search R*

```
1  if P and b do not have overlapping then do
2      return R
3  end if
4  if P is a non-leaf node then do
5      Calculate the membership value for Q towards child node D1 and D2 of P
6      if D1 has higher membership value then do
7          SFCC-knnSearch(Q, D1, b)
8          SFCC-knnSearch(Q, D2, b)
9      else
10         SFCC-knnSearch(Q, D2, b)
11         SFCC-knnSearch(Q, D1, b)
       end if
12 else
13     Perform linear knn search within the leaf node
14     Update R and b
15     return R and b
16 end if
```

Algorithm 4.5 is a depth first based high probability first searching algorithm. In the proposed k-NN search algorithm, every tree node is first checked by the overlap check algorithm. If there exists possible k-NN search result, its child node that gives higher membership value to the query is further examined. This process continue until it reaches a leaf node. As the data number in a leaf node is not very large, we perform a linear search in the leaf node, mark down those possible results, and backtrack one level. This process continues until all the subtrees are either pruned or searched.

4.3 Experiments

In this section, we perform a series of experiments to examine the performance of our proposed indexing structure. In our experiments, we use different kinds of data together with different parameters in order to measure the efficiency of the SFCC-b-tree with modified branch-and-bound algorithm for 100% k-nearest-neighbor search.

4.3.1 Experimental Setting

We conducted 4 different experiments to measure the efficiency of the SFCC-b-tree indexing structure for 100% nearest-neighbor search. All the experiments were conducted on an Ultra Sparc 5 machine and both the SFCC-b-tree and VP-tree used for comparison was implemented in C++.

Before we have a description on every experiment, we define a new measurement here. Because we are now doing 100% k-NN search, so traditional measurement such as recall and precision cannot be used here, as what they measure is accuracy but not efficiency.

In a nearest-neighbor retrieval, the most time-consuming part is to calculate the distances between a query and the feature vectors. Therefore, the efficiency of an indexing structure is almost proportional to the number of these distance computations. On the other hand, the efficiency of our indexing structure is defined based on the efficiency of the linear search because it has the worst efficiency in searching among other methods. Our searching efficiency is defined as:

Definition 4.2 (Efficiency Measurement)

$$efficiency = 1 - \frac{\text{\# of distance computations for the checked method}}{\text{\# of distance computations in linear search}}. \quad (4.1)$$

In the above definition, the efficiency of the linear search is 0 because it needs to compute the distance between every feature vector and the query. Also, the total number of distance computations for linear search is equal to the size of the data set. So, we can convert Equation 4.1 into:

$$efficiency = 1 - \frac{\text{\# of distance computations for the checked method}}{\text{size of the data set}}. \quad (4.2)$$

Here, we use an example to illustrate what is the practical meaning for this efficiency. For example, if the searching efficiency of a method is 0.7, then the methods needs only approximately 30% of the searching time needed by the linear search for retrieval. Our experiments are focus on the following respects:

- **Experiment 8:** Test for different leaf node sizes.
- **Experiment 9:** Test for different numbers of dimensions.
- **Experiment 10:** Test for different sizes of data sets.
- **Experiment 11:** Test for different data distributions.

We want to find out how the above parameters affect the overall efficiency of our indexing method. In each of the experiment, we first build a SFCC-b-tree in batch mode for each of the testing data set and then perform k-nearest neighbor searches to calculate the efficiency with Equation 4.2. Finally, we have a brief discussion and conclusion after each experiment.

4.3.2 Experiment 8: Test for different leaf node sizes.

In Experiment 8, we test our indexing structure with different leaf node sizes.

Motivation

In SFCC-b-tree, every leaf-node should contain less than a pre-defined number of data instances. However, as we perform linear search in the leaf node. It means that the larger the leaf-node size, the higher the overhead while perform linear search within a leaf-node. So, the overall efficiency is related to this pre-defined value.

In this experiment, we want to find out the relationship between the leaf-node sizes and the overall efficiency of the SFCC-b-tree.

Experimental Setting

In this experiment, similar to Experiment 1, we use synthetic data sets in the Gaussian mixture distribution to test our proposed indexing method. After indexing, we perform 100% k-nearest neighbor retrieval.

In our SFCC-b-tree, in order to find out the most suitable leaf-node size, we test it with several different leaf node size. The leaf node size varies from 1% to 20% of the total data size. Then, we mark down three figures from the experiments. They are: Indexing Structure Construction Time, Searching

Node Size	100, 200, 500, 1000, and 2000.
Size of Data Set	10,000.
Data Type	Clustered data with 100 Gaussian mixtures.
Dimensionality	2, 5, 10, 20.
Number of Database Objects Retrieved	10, 20, 50, 100, 500, and 1,000.

Table 4.1: Details of the parameters in Experiment 8.

Time, and Searching Efficiency.

In order to make a comparison with other indexing structure, we use the same setting to build a VP-tree and measure its figures for reference. Table 4.1 shows the details of the parameters setting in Experiment 8.

Experiment Results

Table 4.2 and 4.3 shows the building time of the indexing structures. Table 4.4 and 4.5 shows the searching time for 100% k-NN search. Figure 4.2 and 4.3 summarizes the experimental results.

From the experimental results, we found that:

1. The smaller the node size, the better the efficiency.
2. The smaller the node size, the longer the construction time.
3. The larger the number of database objects retrieved, the worse the efficiency.

From the experimental results, we find that smaller leaf-node size leads to better performance. When the leaf-node size is small, the SFCC-b-tree having a higher resolution power, it helps to prune out those impossible nodes more

efficiently. As a result, the efficiency for indexing structure having small leaf-node size is better.

It seem to be a tradeoff that the indexing structure construction time is higher for those indexing structure with small leaf-node size. However, as the construction phase is a pre-process and only does once. So, we would prefer to use a smaller leaf-node size. From the experiment, it finds that 1% (size=100) to 2% (size=200) of the total data set size is a suitable leaf-node size of SFCC-b-tree.

When, compared with VP-tree, SFCC-b-tree always have a better performance than VP-tree, no matter in efficiency or searching time. As VP-tree does not consider the natural cluster information, it is expected that SFCC-b-tree have a higher performance than VP-tree.

However, the construction time for SFCC-b-tree is higher than VP-tree when the leaf-node size is extremely small, for example, 1% of the total data set size. It is because similar to many competitive based clustering algorithms, SFCC converges slower in small amount of data with loosely structure. So, when the leaf-node size is too small, SFCC needs a long time to converge. This slow converge rate in clustering leads to the long construction time for SFCC-b-tree with small leaf-node size.

4.3.3 Experiment 9: Test for different dimensionality.

In Experiment 9, we want to test the performance of SFCC-b-tree under different dimensionality.

Node Size	**100**	**200**	**500**	**1,000**	**2,000**
2-D	152.83	77.62	31.54	15.40	6.34
5-D	317.83	157.04	61.23	35.37	13.82
10-D	523.51	261.30	97.76	43.15	18.60
20-D	924.18	468.29	164.87	83.91	37.66

Table 4.2: The average time used (in second) for building SFCC-b-tree with different leaf-node size in Experiment 8.

Node Size	**100**	**200**	**500**	**1,000**	**2,000**
2-D	224.56	210.63	206.53	202.06	192.11
5-D	316.89	287.50	285.04	285.19	256.89
10-D	422.99	422.05	409.17	395.08	384.23
20-D	669.11	659.43	649.79	632.90	592.04

Table 4.3: The average time used (in second) for building VP-tree with different leaf-node size in Experiment 8.

Node Size		**100**	**200**	**500**	**1,000**	**2,000**
20-D	**k = 10**	0.01	0.05	0.09	0.10	0.11
	k = 20	0.01	0.08	0.10	0.11	0.13
	k = 50	0.05	0.09	0.11	0.12	0.14
	k = 100	0.08	0.12	0.13	0.14	0.18
	k = 500	0.19	0.25	0.26	0.27	0.27
	k = 1,000	0.48	0.59	0.62	0.73	0.81

Table 4.4: The average time used (in second) for searching the k-nearest neighbors in SFCC-b-tree with different leaf-node size in Experiment 8.

Node Size		**100**	**200**	**500**	**1,000**	**2,000**
20-D	**k = 10**	3.11	3.15	3.20	3.24	3.59
	k = 20	3.17	3.20	3.22	3.24	3.25
	k = 50	3.16	3.22	3.23	3.27	3.27
	k = 100	3.18	3.23	3.25	3.30	3.31
	k = 500	3.23	3.42	3.78	3.92	3.98
	k = 1,000	5.04	5.13	5.23	5.36	5.39

Table 4.5: The average time used (in second) for searching the k-nearest neighbors in VP-tree with different leaf-node size in Experiment 8.

Node Size		**100**	**200**	**500**	**1,000**	**2,000**
2-D	**k = 10**	0.9695	0.9469	0.8938	0.8835	0.8766
	k = 20	0.9620	0.9393	0.8866	0.8844	0.8674
	k = 50	0.9461	0.9259	0.8780	0.8716	0.8628
	k = 100	0.9291	0.9063	0.8757	0.8741	0.8599
	k = 500	0.8557	0.8466	0.8415	0.8388	0.8342
	k = 1,000	0.7790	0.7891	0.7673	0.7650	0.7501
5-D	**k = 10**	0.8956	0.8845	0.8763	0.8748	0.8639
	k = 20	0.8881	0.8825	0.8761	0.8734	0.8608
	k = 50	0.8824	0.8758	0.8726	0.8712	0.8597
	k = 100	0.8766	0.8741	0.8710	0.8696	0.8581
	k = 500	0.8420	0.8405	0.8392	0.8361	0.8328
	k = 1,000	0.7634	0.7618	0.7586	0.7535	0.7475
10-D	**k = 10**	0.8515	0.8390	0.8366	0.8281	0.8073
	k = 20	0.8419	0.8369	0.8335	0.8250	0.7928
	k = 50	0.8414	0.8358	0.8320	0.8244	0.7928
	k = 100	0.8409	0.8302	0.8298	0.8240	0.7871
	k = 500	0.7624	0.7605	0.7603	0.7602	0.7545
	k = 1,000	0.5150	0.5145	0.5143	0.5140	0.5070
20-D	**k = 10**	0.6184	0.5778	0.5774	0.5761	0.5382
	k = 20	0.6032	0.5679	0.5677	0.5670	0.5304
	k = 50	0.6026	0.5679	0.5676	0.5639	0.5301
	k = 100	0.5826	0.5580	0.5673	0.5520	0.5220
	k = 500	0.5187	0.5094	0.5092	0.5066	0.4811
	k = 1,000	0.3376	0.3375	0.3368	0.3348	0.3277

Table 4.6: The average efficiency for perform 100% k-NN search in SFCC-b-tree with different leaf-node size in Experiment 8.

Node Size		**100**	**200**	**500**	**1,000**	**2,000**
2-D	**k = 10**	0.8148	0.7767	0.7126	0.6375	0.5000
	k = 20	0.8032	0.7610	0.6876	0.6187	0.4625
	k = 50	0.7719	0.7329	0.6627	0.5875	0.4585
	k = 100	0.7313	0.6985	0.6283	0.5625	0.4375
	k = 500	0.5912	0.5532	0.5358	0.5063	0.3880
	k = 1,000	0.5405	0.5163	0.4845	0.4619	0.3750
5-D	**k = 10**	0.6533	0.6406	0.6460	0.6203	0.4250
	k = 20	0.6336	0.6234	0.6209	0.6093	0.4125
	k = 50	0.6125	0.5938	0.5822	0.5844	0.4000
	k = 100	0.6000	0.5767	0.5666	0.5594	0.3375
	k = 500	0.5407	0.5266	0.5203	0.4750	0.2750
	k = 1,000	0.5062	0.5000	0.4735	0.3375	0.2459
10-D	**k = 10**	0.6015	0.5766	0.5406	0.4875	0.4000
	k = 20	0.5718	0.5500	0.5398	0.4749	0.3946
	k = 50	0.5030	0.4734	0.4343	0.4000	0.3375
	k = 100	0.4961	0.4641	0.4249	0.3937	0.3250
	k = 500	0.4766	0.4282	0.3906	0.3687	0.3125
	k = 1,000	0.3008	0.3000	0.2979	0.2625	0.2250
20-D	**k = 10**	0.3828	0.3577	0.3260	0.2937	0.2899
	k = 20	0.3750	0.3469	0.3258	0.2875	0.2790
	k = 50	0.3523	0.3296	0.3250	0.2850	0.2715
	k = 100	0.3297	0.3250	0.3125	0.2750	0.2680
	k = 500	0.3078	0.3031	0.3000	0.2726	0.2662
	k = 1,000	0.2883	0.2782	0.2344	0.2125	0.2000

Table 4.7: The average efficiency for perform 100% k-NN search in VP-tree with different leaf-node size in Experiment 8.

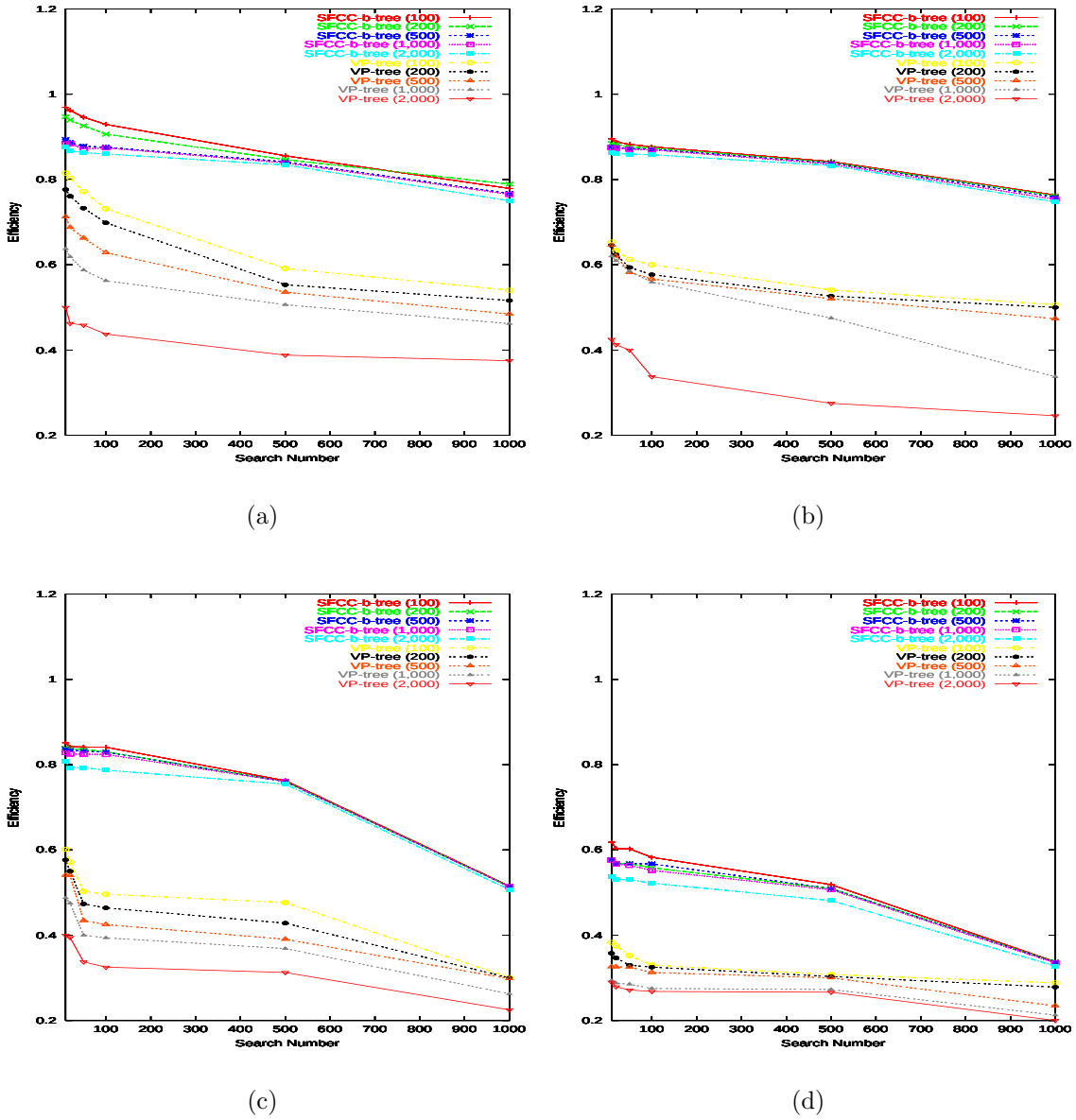


Figure 4.2: Results of Experiment 8. (a), (b), (c), and (d) are the average efficiency for perform 100% k-NN search with different leaf-node size under 2-D, 5-D, 10-D, and 20-D respectively in Experiment 8.

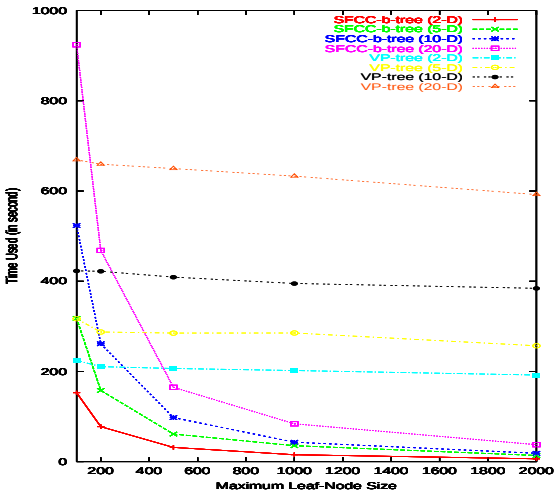


Figure 4.3: Time used (in second) to build the indexing structure with different leaf-node size.

Motivation

Usually, the dimensionality of the data set in multimedia database is high and not fixed. However, many existing indexing structures are not quite applicable in high dimensional data. As a result, they perform not very well in multimedia database.

SFCC-b-tree is aimed to deal with multimedia database, so, it is expected to work under high dimensional data. In this experiment, we want to find out whether SFCC-b-tree works well under high dimensional data and how the dimensionality of data affect its performance.

Experimental Setting

In Experiment 9, we use data with different dimensionality together with several other parameters to test the efficiency of SFCC-b-tree. Table 4.8 shows the details of the parameters. After indexing, we perform 100 different 100% k-nearest neighbors retrieval for SFCC-b-tree and VP-tree.

Dimensionality	2, 5, 10, 20.
Number of Database Objects Retrieved	10, 20, 50, 100, 500, and 1,000.
Node Size	100, 200, 500, 1000, and 2000.
Size of Data Set	10,000.
Data Type	Clustered data with 100 Gaussian mixtures.

Table 4.8: Details of the parameters in Experiment 9.

Experiment Results

We use some tables and figures to show the experiment results. Figure 4.4 shows the average efficiency of the indexing structures under different dimensionality and Figure 4.5 shows the construction time for them.

After the experiment, here are the observations:

1. The higher the dimensionality, the worse the efficiency.
2. The higher the dimensionality, the longer the construction time.
3. The larger the number of database objects retrieved, the worse the efficiency.

From the experimental results, we find that under low dimensions, SFCC-b-tree works very good. The searching efficiency of SFCC-b-tree under high dimensional data (20-D) is also acceptable. In 20-D data set, the efficiency of SFCC-b-tree is still higher than 0.5 for 100-nearest neighbors search.

On the other hand, given the same leaf-node size, the building time for SFCC-b-tree is shorter than VP-tree does. So, we conclude that SFCC-b-tree performs better than VP-tree in high dimensional data.

Dimensionality	**2**	**5**	**10**	**20**
m = 100	152.83	317.83	523.51	924.18
m = 200	77.62	157.04	261.30	468.29
m = 500	31.54	61.23	97.76	164.87
m = 1,000	15.40	35.37	43.15	83.91
m = 2,000	6.34	13.82	18.60	37.66

Table 4.9: The average time used (in second) for building SFCC-b-tree with different dimensionality and leaf-node size, m, in Experiment 9.

Dimensionality	**2**	**5**	**10**	**20**
m = 100	224.56	316.89	422.99	669.11
m = 200	210.63	287.50	422.05	659.43
m = 500	206.53	285.04	409.17	649.79
m = 1,000	202.06	285.19	395.08	632.90
m = 2,000	192.11	256.89	384.23	592.04

Table 4.10: The average time used (in second) for building VP-tree with different dimensionality and leaf-node size, m, in Experiment 9.

The reason for SFCC-b-tree performs better is we do not use L_2-distance in the construction phase. So, it does not suffer from the problem of high dimensionality which is stated in [76].

4.3.4 Experiment 10: Test for different sizes of data sets.

In Experiment 10, we test the performance of SFCC-b-tree with different sizes of data sets. The aim of this experiment is to find out if our method is suitable for huge database.

Dimensionality		**2**	**5**	**10**	**20**
m = 200	**k = 10**	0.01	0.05	0.07	0.05
	k = 20	0.01	0.08	0.10	0.08
	k = 50	0.02	0.09	0.11	0.09
	k = 100	0.08	0.12	0.13	0.12
	k = 500	0.10	0.25	0.26	0.25
	k = 1,000	0.38	0.42	0.49	0.59

Table 4.11: The average time used (in second) for searching the k-nearest neighbors in SFCC-b-tree with different dimensionality and leaf-node size, 200, in Experiment 9.

Dimensionality		**2**	**5**	**10**	**20**
m = 200	**k = 10**	0.55	0.85	1.67	3.15
	k = 20	0.74	0.88	1.67	3.20
	k = 50	0.89	0.92	1.70	3.22
	k = 100	0.91	1.02	1.75	3.23
	k = 500	1.57	2.33	3.04	3.42
	k = 1,000	3.04	4.53	5.10	5.13

Table 4.12: The average time used (in second) for searching the k-nearest neighbors in VP-tree with different dimensionality and leaf-node size, 200, in Experiment 9.

Dimensionality		**2**	**5**	**10**	**20**
m = 100	**k = 10**	0.9695	0.8956	0.8515	0.6184
	k = 20	0.9620	0.8881	0.8419	0.6032
	k = 50	0.9461	0.8824	0.8414	0.6026
	k = 100	0.9291	0.8766	0.8409	0.5826
	k = 500	0.8557	0.8420	0.7624	0.5187
	k = 1,000	0.7790	0.7634	0.5150	0.3376
m = 200	**k = 10**	0.9469	0.8845	0.8390	0.5778
	k = 20	0.9393	0.8825	0.8369	0.5679
	k = 50	0.9259	0.8758	0.8358	0.5679
	k = 100	0.9063	0.8741	0.8302	0.5580
	k = 500	0.8466	0.8405	0.7605	0.5094
	k = 1,000	0.7891	0.7618	0.5145	0.3375
m = 500	**k = 10**	0.8938	0.8763	0.8366	0.5774
	k = 20	0.8866	0.8761	0.8335	0.5677
	k = 50	0.8780	0.8726	0.8320	0.5676
	k = 100	0.8757	0.8710	0.8298	0.5673
	k = 500	0.8415	0.8392	0.7603	0.5092
	k = 1,000	0.7673	0.7586	0.5143	0.3368
m = 1,000	**k = 10**	0.8835	0.8748	0.8281	0.5761
	k = 20	0.8844	0.8734	0.8250	0.5670
	k = 50	0.8716	0.8712	0.8244	0.5639
	k = 100	0.8741	0.8696	0.8240	0.5520
	k = 500	0.8388	0.8361	0.7602	0.5066
	k = 1,000	0.7650	0.7535	0.5140	0.3348
m = 2,000	**k = 10**	0.8766	0.8639	0.8073	0.5382
	k = 20	0.8674	0.8608	0.7928	0.5382
	k = 50	0.8628	0.8597	0.7928	0.5301
	k = 100	0.8599	0.8581	0.7871	0.5220
	k = 500	0.8342	0.8328	0.7545	0.4811
	k = 1,000	0.7501	0.7475	0.5070	0.3277

Table 4.13: The average time used (in second) for perform k-NN search in SFCC-b-tree with different dimensionality and leaf-node size, m, in Experiment 9.

Dimensionality		**2**	**5**	**10**	**20**
m = 100	**k = 10**	0.8148	0.6533	0.6015	0.3828
	k = 20	0.8032	0.6336	0.5718	0.3750
	k = 50	0.7719	0.6125	0.5030	0.3523
	k = 100	0.7313	0.6000	0.4961	0.3297
	k = 500	0.5912	0.5407	0.4766	0.3078
	k = 1,000	0.5405	0.5062	0.3008	0.2883
m = 200	**k = 10**	0.7767	0.6406	0.5766	0.3577
	k = 20	0.7610	0.6234	0.5500	0.3469
	k = 50	0.7329	0.5938	0.4734	0.3296
	k = 100	0.6985	0.5767	0.4641	0.3250
	k = 500	0.5532	0.5266	0.4282	0.3031
	k = 1,000	0.5163	0.5000	0.3000	0.2782
m = 500	**k = 10**	0.7126	0.6460	0.5406	0.3260
	k = 20	0.6876	0.6209	0.5398	0.3258
	k = 50	0.6627	0.5822	0.4343	0.3250
	k = 100	0.6283	0.5666	0.4249	0.3125
	k = 500	0.5358	0.5203	0.3906	0.3000
	k = 1,000	0.4845	0.4735	0.2979	0.2344
m = 1,000	**k = 10**	0.6375	0.6203	0.4875	0.2937
	k = 20	0.6187	0.6093	0.4749	0.2875
	k = 50	0.5875	0.5844	0.4000	0.2850
	k = 100	0.5625	0.5594	0.3937	0.2750
	k = 500	0.5063	0.4750	0.3687	0.2726
	k = 1,000	0.4619	0.3375	0.2625	0.2125
m = 2,000	**k = 10**	0.5000	0.4250	0.4000	0.2899
	k = 20	0.4625	0.4125	0.3946	0.2790
	k = 50	0.4585	0.4000	0.3375	0.2715
	k = 100	0.4375	0.3375	0.3250	0.2680
	k = 500	0.3880	0.2750	0.3125	0.2662
	k = 1,000	0.3750	0.2459	0.2250	0.2000

Table 4.14: The average time used (in second) for perform k-NN search in VP-tree with different dimensionality and leaf-node size, m, in Experiment 9.

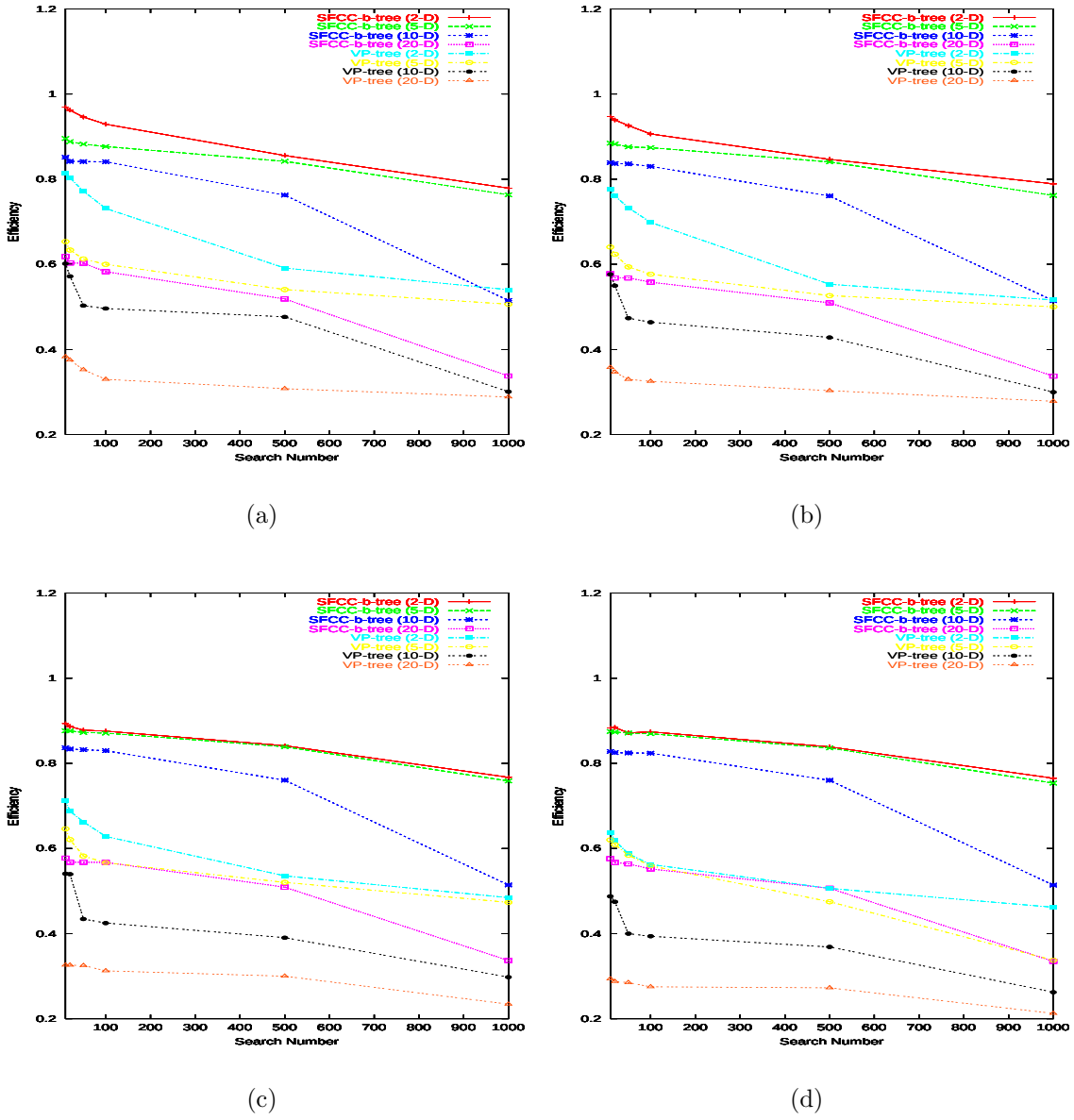


Figure 4.4: Results of Experiment 9. (a), (b), (c), and (d) are the average efficiency for perform 100% k-NN search under different dimensionality with leaf-node size 100, 200, 500, and 1,000 respectively in Experiment 9.

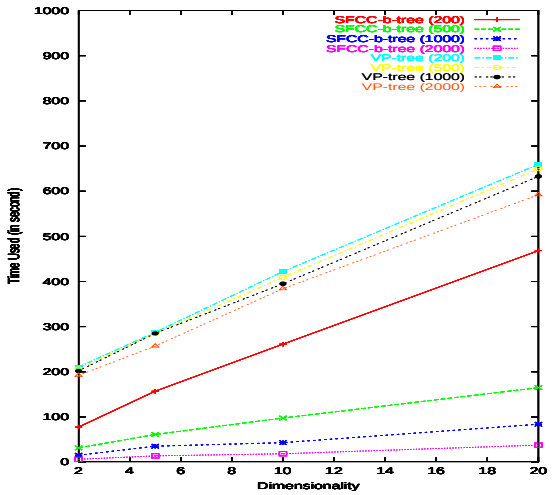


Figure 4.5: Time used (in second) to build the indexing structure with different dimensionality.

Motivation

A typical multimedia database usually exists in a huge size. So, those indexing structures deal with multimedia database should be able to work fine under huge data sets.

In this experiment, we want to find out if our method suitable for those large data sets. If it is suitable for large data sets, it is also suitable for multimedia databases.

Experimental Setting

Similar to others experiments, we test the efficiency of 100% nearest-neighbor retrieval for the indexing structures. In this experiment, we use different sizes of data sets together with other parameters which are shown in Table 4.15.

From Experiment 8, we find that 2% (200) of the total data set size is a suitable value of the leaf-node size. So, we fix the leaf-node size to 200 in this

Size of Data Set	1,000, 20,000, and 50,000.
Dimensionality	10.
Number of Database Objects Retrieved	10, 20, 50, 100, 500, and 1,000.
Node Size	200.
Data Type	Clustered data with 100 Gaussian mixtures.

Table 4.15: Details of the parameters in Experiment 10.

experiment.

Experiment Results

Again, we use some tables and figures to show the experiment results. Figure 4.6 shows the average efficiency of the indexing structures under different dimensionality and Figure 4.7 shows the construction time for them.

After the experiment, here are the observations:

1. The larger the data set size, the better the efficiency.
2. The larger the data set size, the longer the construction time.
3. The larger the number of database objects retrieved, the worse the efficiency.

From the experimental results, we find that searching efficiency increase with the size of data set. The reason for this can refer to Experiment 8. We know from Experiment 8 that smaller leaf-node size leads to better efficiency. In other words, if we keep the ratio of (data set size / leaf-node size) constant, increase the data set size has the same effect to decrease the leaf-node size. So, it is not surprising that the searching efficiency increase with the size of data set.

Data Set Size		**1,000**	**20,000**	**50,000**
m = 200	**SFCC-b-tree**	20.07	952.23	2050.67
	VP-tree	3.74	1679.73	8145.10

Table 4.16: The average time used (in second) for building indexing structure with different data set size for 10-D data set in Experiment 10.

Data Set Size		**1,000**	**20,000**	**50,000**
m = 200	**k = 10**	0.01	0.02	0.10
	k = 20	0.01	0.04	0.11
	k = 50	0.02	0.04	0.11
	k = 100	0.05	0.06	0.18
	k = 500	0.06	0.21	0.33
	k = 1,000	0.18	0.45	0.64

Table 4.17: The average time used (in second) for searching the k-nearest neighbors in SFCC-b-tree with different data set size in Experiment 10.

According to the construction time, VP-tree is more sensitive to the number of data size than SFCC-b-tree. It is because for VP-tree, it needs to check all the data instances in a tree node when building new child nodes. However, as we show in Experiment 1, SFCC is a competitive based clustering algorithm, and it is not sensitive to the number of samples. As a result, the construction time for SFCC-b-tree is less than VP-tree.

Data Set Size		**1,000**	**20,000**	**50,000**
m = 200	**k = 10**	0.171	3.258	8.273
	k = 20	0.171	3.347	8.290
	k = 50	0.189	3.371	8.336
	k = 100	0.196	3.469	8.427
	k = 500	0.354	5.497	11.226
	k = 1,000	0.431	8.933	18.432

Table 4.18: The average time used (in second) for searching the k-nearest neighbors in VP-tree with different data set size in Experiment 10.

Data Set Size		**1,000**	**20,000**	**50,000**
m = 200	**k = 10**	0.805	0.845	0.863
	k = 20	0.799	0.844	0.862
	k = 50	0.743	0.824	0.862
	k = 100	0.679	0.795	0.851
	k = 500	0.188	0.674	0.762
	k = 1,000	0.000	0.648	0.739

Table 4.19: The average efficiency for searching the k-nearest neighbors in SFCC-b-tree with different data set size in Experiment 10.

Data Set Size		**1,000**	**20,000**	**50,000**
m = 200	**k = 10**	0.487	0.512	0.739
	k = 20	0.485	0.491	0.735
	k = 50	0.437	0.442	0.730
	k = 100	0.325	0.420	0.691
	k = 500	0.037	0.384	0.657
	k = 1,000	0.000	0.372	0.640

Table 4.20: The average efficiency for searching the k-nearest neighbors in VP-tree with different data set size in Experiment 10.

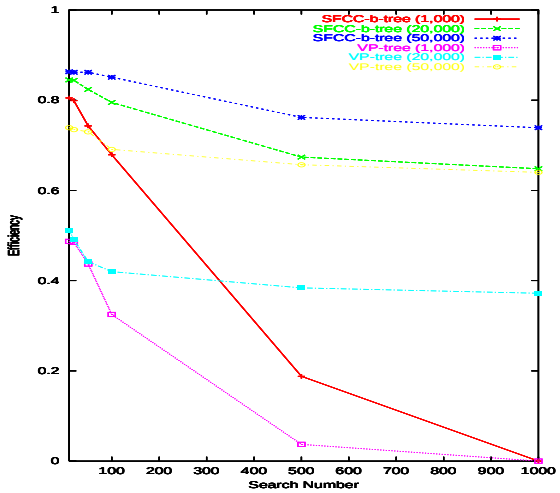


Figure 4.6: The average efficiency for perform 100% k-NN search with different data set sizes.

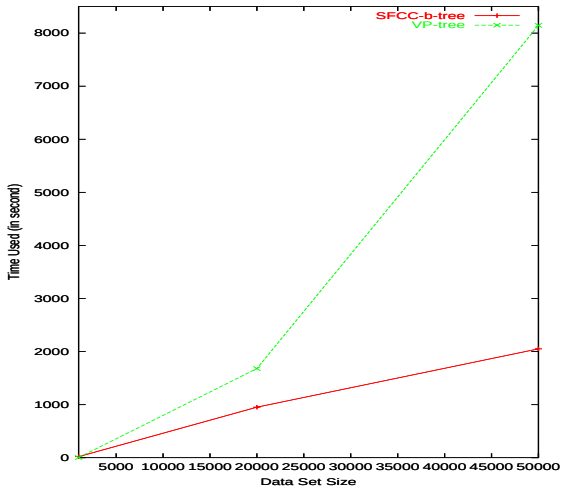


Figure 4.7: Time used (in second) to build the indexing structure with different data set sizes.

4.3.5 Experiment 11: Test for different data distributions.

In Experiment 11, we test our indexing method for different distributions.

Motivation

In our indexing method, we have an assumption that there exist natural clusters in the data. However, we want to make sure that our indexing method works fine when this assumption is weak.

Experimental Setting

In this experiment, we use clustered data with different numbers of Gaussian mixtures together with an uniform data set to test the indexing methods. Table 4.21 shows the setting of the parameters in this experiment. For each setting of parameters, we perform 100% k-nearest neighbor retrieval and the average results are used for analysis.

Data Type	Clustered data with 10 and 100 Gaussian mixtures together with uniform data set.
Size of Data Set	10,000.
Dimensionality	10.
Number of Database Objects Retrieved	10, 20, 50, 100, 500, and 1,000.
Node Size	200.

Table 4.21: Details of the parameters in Experiment 11.

Experiment Results

We use some tables and figures to show the experiment results. Table 4.22 shows the indexing structure construction time, Table 4.23 and Table 3.3 show the searching time for k-NN search, and Figure 4.8 shows the average efficiency of the indexing structures in Experiment 11.

After the experiment, here are the observations:

1. The more the Gaussian mixtures, the worse the efficiency.
2. The number of Gaussian mixtures seem unrelated to the indexing structure construction time.
3. The larger the number of database objects retrieved, the worse the efficiency.

From the experimental results, we find that more the Gaussian mixtures, worse the efficiency. It is because when the the number of Gaussian mixtures increase, our assumption in SFCC-b-tree becomes weak. Also, as the number of small clusters increases, the indexing structure may have more nodes. This worsen the searching performance because more decisions have to be made for determining whether the node is going to be examined.

Number of Gaussian Mixtures		**10**	**100**	**Uniform**
m = 200	**SFCC-b-tree**	264.54	261.30	263.91
	VP-tree	423.65	422.05	422.05

Table 4.22: The average time used (in second) for building indexing structure with data set having different number of Gaussian mixtures in Experiment 11.

Number of Gaussian Mixtures		**10**	**100**	**Uniform**
m = 200	**k = 10**	0.01	0.05	0.06
	k = 20	0.01	0.08	0.07
	k = 50	0.02	0.09	0.11
	k = 100	0.03	0.12	0.15
	k = 500	0.08	0.25	0.38
	k = 1,000	0.12	0.59	0.51

Table 4.23: The average time used (in second) for searching the k-nearest neighbors in SFCC-b-tree with data set having different number of Gaussian mixtures in Experiment 11.

Number of Gaussian Mixtures		**10**	**100**	**Uniform**
m = 200	**k = 10**	3.12	3.15	3.21
	k = 20	3.15	3.20	3.24
	k = 50	3.21	3.22	3.33
	k = 100	3.20	3.23	3.38
	k = 500	3.36	3.42	3.78
	k = 1,000	5.05	5.13	5.26

Table 4.24: The average time used (in second) for searching the k-nearest neighbors in VP-tree with data set having different number of Gaussian mixtures in Experiment 11.

Number of Gaussian Mixtures		**10**	**100**	**Uniform**
m = 200	**k = 10**	0.9811	0.8390	0.8073
	k = 20	0.9543	0.8369	0.7951
	k = 50	0.9379	0.8358	0.6643
	k = 100	0.9210	0.8302	0.5464
	k = 500	0.8462	0.7605	0.2148
	k = 1,000	0.7764	0.5145	0.0429

Table 4.25: The average efficiency for searching the k-nearest neighbors in SFCC-b-tree with data set having different number of Gaussian mixtures in Experiment 11.

Number of Gaussian Mixtures		**10**	**100**	**Uniform**
m = 200	**k = 10**	0.6403	0.5766	0.4518
	k = 20	0.6204	0.5500	0.3946
	k = 50	0.5844	0.4734	0.2812
	k = 100	0.5512	0.4641	0.2631
	k = 500	0.4750	0.4282	0.1682
	k = 1,000	0.3312	0.3000	0.0371

Table 4.26: The average efficiency for searching the k-nearest neighbors in VP-tree with data set having different number of Gaussian mixtures in Experiment 11.

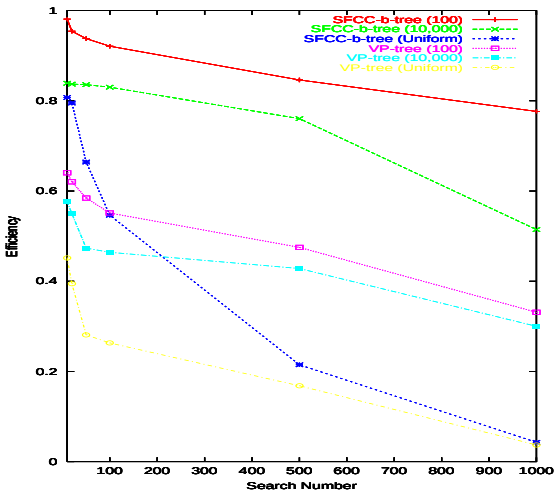


Figure 4.8: The average efficiency for performing 100% k-NN search with data set having different number of Gaussian mixtures in Experiment 11.

4.4 Summary

In this section, we have a summary on the performance of SFCC-b-tree.

After a series of experiment, we find that our indexing method has a good searching performance in general. Also, we have shown that our algorithm outperforms VP-tree in indexing and retrieval. From the experimental results, we find that in order to have a better searching efficiency, the SFCC-b-tree should have a small leaf-node size. 2% of the total data set size seem to be a good leaf-node size for SFCC-b-tree.

Also, as the relationship between each level is very clear in SFCC-b-tree. It enables us to update the SFCC-b-tree very easily.

Chapter 5

A Case Study on SFCC-b-tree

In this chapter, we have a case study on the proposed indexing method. We divide this chapter into five sections. In the first section, we explain the motivation of the case study and why we choose "web document database" for the case study. In the second section, we explain how we build up the database. Then, in the third section, we illustrate how we do the data pre-processing. It includes data cleaning, feature extraction, and others. In the fourth section of this chapter, we build the indexing structure and test its performance. The experimental results and explanation are included in this section too. Finally, we have a conclusion in the fifth section.

5.1 Introduction

We use web document database in our case study. Based on our assumption, data form natural clusters. Therefore, if we first extract the natural clusters information and use these information to build the indexing structure for data retrieval, the data retrieval will become more efficient and effective.

However, all the experiments in Chapter 4 are based on synthetic data sets. So, we want to examine our proposed algorithm in real life data set. If the

experimental results agree with our assumptions, then it is save to use the proposed indexing algorithm on other real life applications too.

We believe that there exist natural clusters in a web document database. In a typical web document, it usually contains keywords which can describe the document. For an example, those web document about "snow" may contain keywords like cold, white, and so on. Similarly, those document about "fire" may contain another set of keywords like hot, danger, and so on. As a result, we can assume that natural clusters exist in a web document database.

In the next section, we illustrate how we get the web document to form a database.

5.2 Data Collection

We have collected over 10,000 web documents from the internet. We get the documents in the following ways:

1. **Set the Start Page:**
   We first set our school web page (http://www.cse.cuhk.edu.hk/index.html) as the start page. Then we download this web page as the first web document in our web document database.

2. **Find Hyper-linkages in the Web Page:**
   After we set a start page, the second step is finding out all the hyper-linkages that exist in that web page. A hyper-linkage in web page means a virtual linkage between two web page. Users are able to go from one web page to another through the hyper-linkage.

All the hyper-linkages being found are stored in a queue. For example, if we find n hyper-linkages in the web page, then the queue will contain n elements, with each of the element representing a hyper-linkage.

3. **Download Another Web Page from the Queue:**
   After we find all the hyper-linkages within a web page, we pick a new page from the front of the queue and start to download this web page. At the same time, we find if there are any new hyper-linkages in this page. If there exist hyper-linkage, the new linkage will put at the end of the queue for downloading later.

   Then, we loop back to Step 2, until either the queue is empty or the total number of web page excess a pre-defined value.

By doing this, the web page is downloaded in a beneath first order.

5.3 Data Pre-processing

After we get all the web documents (or web pages), we perform data pre-processing. The aim of data pre-processing is to clean up those dummy data and extract those useful information from the web document. It consists of four different steps in data pre-processing, they are:

1. **Removing Stop Words:**
   In a web document, it usually contains many stop words. Stop words means those very common terms, variants of a given term, and the use of different terms with similar meanings.

   Before we pass the web document for Stemming. We first check all the words in the web document with a list or stop words (or stop list). Then,

all the stop words will be eliminated from the document.

We need to delete those stop words because in any measurements depending on the word frequencies, they tend to diminish the impact of frequency differences among less common words. Also, these words carry little meaning by themselves, therefore, they may result in a large amount of unproductive processing if left them in the document. So, we need to delete those stop words before further pre-processing.

2. **Stemming:**
   After removing those stop words, we then perform stemming. We need to stem the document because given a word, it may occur in many different forms. For example, compute, computed, computing, and various other words all have the same basic form and all deal with a set of closely related concepts. So, if a user use such a word as the query, the system may not be able to relate those words to one concept. This is clearly undesirable.

   Stemming is a way to deal with the above problem. Stemming strips off word endings, reducing them to a common core or stem. In the above example, the stem might be "comput". For a given document, stemming brings together the various forms of the word, resulting in a higher frequency count and thus in greater significance for the term.

3. **Keywords Extraction:**
   After stemming, we extract those keywords from the document. In our case study, we have a keywords list with a total of 5,000 words and phase in both Chinese and English. Then, we compare those words left after stemming with that in keywords list to make a frequency histogram of the keywords.

However, we do not use all the 5,000 keywords to build a 5,000-D database. Because it seems to be too large for a normal indexing structure. So, we use those keywords with the top 100 frequency count. As a result, our web document database is a 100-D database.

Also, after keywords extraction, we only keep those document contain those 100 keywords that mention above. By now, our database is a 100-D database with 7,328 web documents inside.

4. **Normalization:**
   Given two documents with different number of words, it appears that the keywords frequency for the longer document is higher than those for the shorter one. So, a long document seems to be more related to any topic if we just use the keywords frequency count to describe a document, and this is clearly undesirable.

   So, we need to normalize the frequency count for each document in the database. For example, assume the frequency count for the words "apple", "boy", and "cat" being appeared in a document is 10, 20, and 30 respectively. Then, after normalization, their value will become 10/(10 + 20 + 30) = 0.167, 20 / 60 = 0.333, and 0.500 respectively.

After normalization, the web document database is ready and we index it by SFCC-b-tree and VP-tree. After the indexing, we perform k-NN search on them. The Experimental results are shown in the next section.

Data Type	Real life web document.
Size of Data Set	7,328.
Dimensionality	100.
Number of Database Objects Retrieved	10, 20, 50, 100, 500, and 1,000.
Node Size	500.

Table 5.1: Details of the parameters in Chapter 5.

SFCC-b-tree	447.35
VP-tree	1,058.37

Table 5.2: The time used (in second) for building indexing structure with web document database in Chapter 5.

5.4 Experimental Results

In this section, we show the experimental results of the case study. After that, we discuss and conclude our experimental results. Table 5.1 summarize the web document database we use in the case study.

We have built the SFCC-b-tree and VP-tree for indexing. After then, we perform 100 different k-NN search to examine the performance of them. Table 5.2 shows the building time for these indexing structures. Table 5.3 and Figure 5.1 show the searching time for k-NN search. Table 5.4 and Figure 5.2 show the searching efficiency for the k-NN search.

From the experiment, we find that SFCC-b-tree needs a shorter building time than VP-tree. This result agrees with those show in Chapter 4. The searching time and efficiency for SFCC-b-tree is also better than VP-tree when the number of data objects retrieved is not very large.

		SFCC-b-tree	VP-tree
m = 500	**k = 10**	0.24	10.11
	k = 20	0.32	10.10
	k = 50	0.38	10.15
	k = 100	0.41	10.25
	k = 500	0.43	10.71
	k = 1,000	0.55	11.29

Table 5.3: The average time used (in second) for searching the k-nearest neighbors with web document database in Chapter 5.

		SFCC-b-tree	VP-tree
m = 500	**k = 10**	0.28	0.18
	k = 20	0.24	0.17
	k = 50	0.20	0.17
	k = 100	0.16	0.16
	k = 500	0.03	0.13
	k = 1,000	0.02	0.12

Table 5.4: The average efficiency for searching the k-nearest neighbors with web document database in Chapter 5.

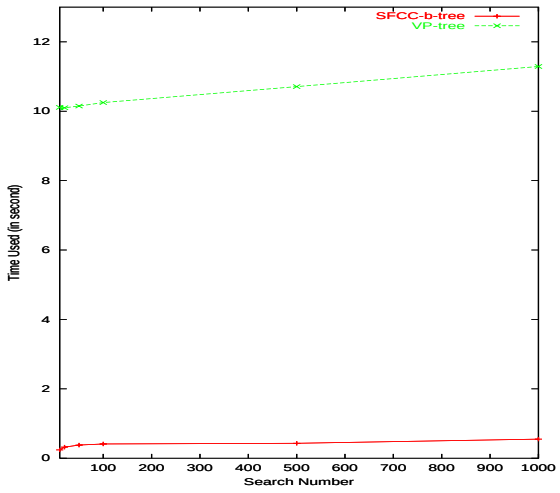


Figure 5.1: The average time used (in second) for searching the k-nearest neighbors with web document database in Chapter 5.

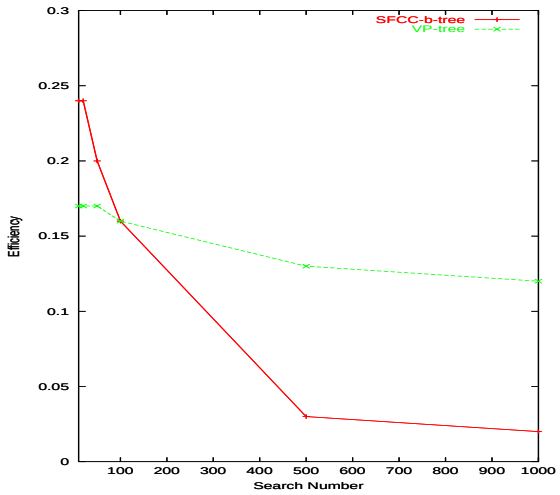


Figure 5.2: The average efficiency for searching the k-nearest neighbors with web document database in Chapter 5.

5.5 Summary

From the case study, we find the performance of SFCC-b-tree agrees with our assumption. This means that there exist natural clusters in the web document database. However, in the case study, we find that when the number of data objects retrieved is large, the efficiency of SFCC-b-tree drops quickly. It is because SFCC-b-tree uses natural clusters information to build the indexing structure. However, it does not ensure that a balance indexing tree will give. As a consequence, when the number of data objects retrieved is too large, the efficiency of SFCC-b-tree drops quickly.

However, in a typical k-NN search, the number of data objects retrieved is not very large, usually less than 50. Therefore, we can conclude that SFCC-b-tree performs well in both synthetic and real data.

Chapter 6

Conclusion

6.1 An Efficiency Formula

In this section, we try to use a formula to describe the searching efficiency. After we have this formula, we can predict the performance before we build the indexing structure and optimize the performance according to the predicted value.

6.1.1 Motivation

In real life databases, their sizes are usually very huge. As a result, it is better to build the indexing structure once only. On the other hand, we also want the indexing structure has a good performance. Therefore, we need an efficiency formula to predict the efficiency of the indexing structure given a set of parameters.

Having this efficiency formula, we can:

1. Predict the efficiency of the indexing structure before we build it out. This save a lot of time and resources.

k_{11}	k_{12}	k_{21}	k_{22}	k_{31}	k_{32}	k_{41}	k_{42}	K
-0.0001	1.088	-0.2244	0.333	-0.0001	0.960	0.1830	0.126	0.6761

Table 6.1: The values of constants for Equation 6.1.

2. Compare the efficiency between different indexing structures without really building it out. This enables us to choose a suitable indexing structure when we index a given database.

6.1.2 Regression Model

We use the data from Chapter 4 to perform the regression. From the experimental results, we know that the searching efficiency is related to:

1. Dimensionality (D),
2. Data set size (S),
3. Maximum leaf-node size (M), and
4. Number of data objects retrieved (R).

So, the efficiency formula is related to these parameters. Also, as we do not know the degree of the efficiency formula, we assume the formula is in the form:

$$efficiency = k_{11}R^{k_{12}} + k_{21}D^{k_{22}} + k_{31}M^{k_{32}} + k_{41}S^{k_{42}} + K. \qquad (6.1)$$

where k_{11}, k_{12}, k_{21}, k_{22}, k_{31}, k_{32}, k_{41}, k_{42}, and K are real-valued constant.

After having this model, we use regression tool to find those constants and their values are listed in Table 6.1.

Real Efficiency	**Predicted Efficiency**	**Difference**
0.28	0.16	0.12
0.24	0.16	0.08
0.20	0.15	0.05
0.16	0.14	0.02
0.03	0.07	0.04
0.02	-0.02	0.04

Table 6.2: The differences between the real efficiency and the predicted efficiency.

So, Equation 6.1 becomes:

$$efficiency = 0.1830S^{0.126} - 0.0001R^{1.088} - 0.2244D^{0.333} - 0.0001M^{0.960} + 0.6761. \tag{6.2}$$

We examine the error of the efficiency formula by comparing the efficiency from the case study in Chapter 5 with the predicted efficiency from the efficiency formula. The results are listed in Table 6.2.

From Table 6.2, we find that the average difference between the real efficiency and the predicted efficiency is 0.06. It means that if we use the efficiency formula to predict the efficiency of an unseen database.

6.1.3 Discussion

From the equation, we can estimate the efficiency by giving the parameter values. Apart from this, we can also find out the relationship between these parameters and the efficiency easily.

First, we know from Equation 6.2 that the efficiency is actually a sum of five terms. They are:

1. $0.1830S^{0.126}$,

2. $-0.0001R^{1.088}$,
3. $-0.2244D^{0.333}$,
4. $-0.0001M^{0.960}$, and
5. 0.6761.

So, we plot their value in Figure 6.1 to find their effect to efficiency under different values.

From Figure 6.1, we we can also find out the relationship between these parameters and the efficiency easily. Here are the relationships.

1. **Dimensionality (D):** As the values of $-0.2244D^{0.333}$ are always negative, it causes degradation to the efficiency. Also, it has the largest magnitude among all the other functions. Therefore, dimensionality causes the major degradation to the efficiency of SFCC-b-tree.
2. **Data set size (S):** The values of $0.1830S^{0.126}$ are positive and increase as S. Therefore, the larger the data set size, the higher the efficiency. Also, it has the second largest magnitude among all the other functions. Therefore, data set size causes a great contribution to the efficiency of SFCC-b-tree.
3. **Maximum leaf-node size (M):** The values of $-0.0001M^{0.960}$ are always negative and decrease as M. So, the larger the maximum leaf-node size, the lower the efficiency. The magnitude of $-0.0001M^{0.960}$ is not very large when compared with other functions, such as those about D and S. So, the effect of M on the efficiency of SFCC-b-tree is not very large.

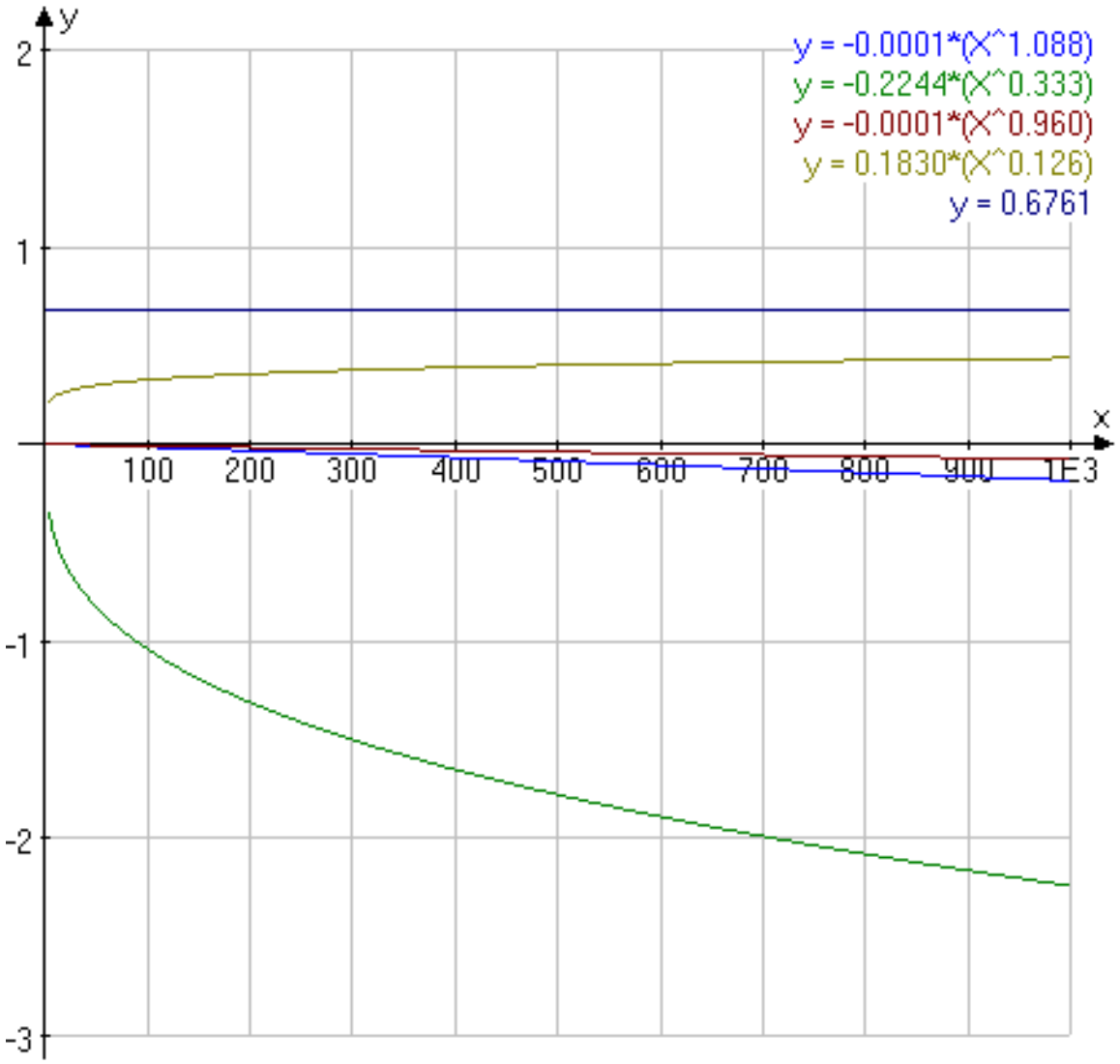


Figure 6.1: Plotting of equation $y = ax^b$ with different values of a and b.

4. **Number of data objects retrieved (R):** The values of $-0.0001R^{1.088}$ are always negative and decrease as M. So, the larger the maximum leaf-node size, the lower the efficiency. Also, The magnitude of $-0.0001R^{1.088}$ is not very large when compared with other functions, such as those about D and S. So, the effect of R on the efficiency of SFCC-b-tree is not very large.

6.2 Future Directions

In this section, we suggest some possible future directions for SFCC-b-tree.

1. **Hybrid Method for SFCC-b-tree:**

   (a) A more balanced tree: SFCC-b-tree used the natural clusters information to build the indexing structure. However, whether the tree balance is not a consideration of SFCC-b-tree. As a result, SFCC-b-tree may perform worse than those indexing structures having considered this issue, when the number of objects retrieved is extremely large. So, it is better to use a hybrid model to build the indexing tree in order to make a more balanced tree.

   (b) Smaller leaf node size: Also, SFCC-b-tree is not very efficient when the data set size is small (size ≤ 100). So, we suggest to use another suitable clustering algorithm for small data sets. As a result, we can use a smaller leaf node size in the tree and it will increase the overall efficiency of retrieval.

2. **The Relationship Formula:** Although the model we are using now is already very complex, the relationship formula presented in Section 6.1 is still based on some assumptions. We do not know that what the real

model of the relationship formula is. Therefore, more research is needed to find out a better model for the relationship formula.

6.3 Conclusion

In this thesis, we have presented two fuzzy clustering algorithms, FCC and SFCC. Also, we have used SFCC to build up an indexing structure, SFCC-b-tree, in a hierarchical approach. We have also analyzed our methods by using some experiments and a case study.

From the experiments and case study, it is concluded that: (1) FCC and SFCC outperform many other clustering algorithms. (2) SFCC-b-tree is efficient to produce 100% nearest-neighbor search results and it outperforms VP-tree for indexing and retrieveal in many aspects. (3) SFCC-b-tree works fine in high dimensional data (100-D data), and in multimedia data.

According to the experimental results, we also work out a efficiency formula for predicting the searching efficiency of SFCC-b-tree.

Bibliography

[1] T. K. Lau, "Rival Penalized Competitive Learning for Content-based Indexing", Master's thesis, The Chinese University of Hong Kong, Hong Kong, 1998.

[2] T. K. Lau and I. King, "Montage: An Image Database for the Fashion, Textile, and Clothing Industry in Hong Kong", in *Proceedings of the Third Asian Conference on Computer Vision (ACCV'98)*, volume 1, pages 410–417, 1998.

[3] C. Faloutsos et al., "Efficient and effective querying by image content", in *Journal of Intelligient Information Systems*, volume 3, pages 231–262, 1994.

[4] A. Pentland, R. W. Picard, and S. Sclaroff, "Photobook: Content-Based Manipulation of Image Databases", in *International Journal of Computer Vision*, volume 18, pages 223–254, 1996.

[5] J. Smith and S. F. Chang, "VisualSEEk: a fully automated content-based image query system", in *ACM multimedia - international conference - 1996*, pages 87–98, 1996.

[6] V. Ogle and M. Stonebraker, "Chabot: Retrieval from a Relational Database of Images", in *Computer*, volume 28, pages 40–48, 1995.

[7] A. Gupta, T. Weymouth, and R.Jain, "Semantic queries with pictures: The VIMSYS model", in *Proceedings 17th VLDB*, pages 69–79, 1991.

[8] K. Hirata and T. Kato, "Query by visual example - content based image retrieval", in *Advances in Database Technology EDBT'92, Third International Conference on Extending Database Technology*, pages 56–71, 1992.

[9] W. W. Chu, A. F. Cardenas, and R. K. Taira, "KMeD: A knowledge-based Multimedia Medical Distributed Database System", in *Information Systems*, volume 20, pages 75–96, 1995.

[10] W. W. Chu, I. T. Ieong, R. K. Taira, and C. M. Breant, "A temporal evolutionary object-oriented data model and its query language for medial image management", in *Proceedings of 18th VLDB Conference*, pages 53–64, 1992.

[11] J. K. Wu, A. D. Narasimhalu, B. M. Mehtre, C. P. Lam, and Y. P. Gao, "CORE: A content-based retrieval engine for multimedia information systems", in *ACM Multimedia Systems*, volume 3, pages 25–41, 1995.

[12] R. Bayer, "Symmetric Binary B-trees: Data Structure and Maintenance Algorithms", in *Acta Informatica*, volume 1, pages 290–306, 1972.

[13] D. Comer, "The Ubiquitous B-tree", in *ACM Computing Surveys*, volume 11, pages 121–137, 1979.

[14] A. Guttman, "R-trees: A Dynamic Index Structure for Spatial Searching", in *ACM SIGMOD*, volume 14, pages 47–57, 1984.

[15] T. Sellis, N. Roussopoulos, and C. Faloutsos, "The R^+-tree: a dynamic index for multidimensional objects", in *Proceedings of the 13th VLDB Conference*, pages 507–518, 1987.

[16] N. Beckmann, H. P. Kriegel, R. Schneider, and B. Seeger, "The R*-tree: an efficient and robust access method for points and rectangles", in *ACM SIGMOD Record*, volume 19, pages 322–331, 1990.

[17] N. Katayama and S. Satoh, "The SR-tree: an index structure for high-dimensional nearest neighbor queries", in *SIGMOD Record*, volume 26, pages 369–380, 1997.

[18] R. A. Finkel and J. L. Bentley, "Quad Trees: A Data Structure for Retrieval on Composite Keys", in *Acta Informatica*, volume 4, pages 1–9, 1974.

[19] J. L. Bentley, "Multidimensional Binary Search Trees Used for Associative Searching", in *Communications of the ACM*, volume 18, pages 509–517, 1975.

[20] P. N. Yianilos, "Data structures and algorithms for nearest neighbor search in general metric spaces", in *Proceedings of the 4th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 311–321, 1993.

[21] S. Berchtold, D. A. Keim, and H. P. Kriegel, "The X-tree: An Index Structure for High-Dimensional Data", in *Proceedings on the 22th VLDB Conference*, pages 28–39, 1996.

[22] D. A. White and R. Jain, "Similarity Indexing with the SS-tree", in *Proceedings of the 12th International Conference on Data Engineering*, pages 516–523, 1996.

[23] H. Y. Yue, I. King, and K. S. Leung, "Fuzzy Clustering Method for Content-based Indexing", in *2001 WSES FSFS International Conference, Fuzzy Sets and Fuzzy Systems*, volume 1, pages 5411–5419, 2001.

[24] H. Y. Yue, I. King, and K. S. Leung, *Advances in Fuzzy Systems and Evolutionary Computation*, volume 1, chapter "Fuzzy Clustering Method

for Content-based Indexing", pages 138–143, World Scientific Engineering Society, 2001.

[25] Y. H. Gong, C. H. Chuan, and X. Y. Guo, "Image Indexing and Retrieval Based on Color Histograms", in *Multimedia Tools and Applications*, volume 2, pages 133–156, 1996.

[26] F. Tomita and S. Tsuji, *"Computer analysis of visual textures"*, Kluwer Academic Publishers, Boston, 1990.

[27] W. Y. Ma and B. S. Manjunath, "Texture-based Pattern Retrieval from Image Databases", in *Multimedia Tools and Applications*, volume 2, pages 35–51, 1996.

[28] B. S. Manjunath and W. Y. Ma, "Texture Features for Browsing and Retrieval of Image Data", in *IEEE Transactions on Pattern Analysis and Machine Intelligence*, volume 18, pages 837–842, 1996.

[29] W. A. Burkhard and R. M. Keller, "Some approaches to best-match file searching", in *Communications of the ACM*, volume 16, pages 230–236, 1973.

[30] K. Fukunaga and P. M. Narendra, "A branch and bound algorithm for computing K-nearest neighbors", in *IEEE Transactions on Computers*, volume 24, pages 750–753, 1975.

[31] C. D. Feustel and L. G. Shapiro, "The nearest neighbor problem in an abstract metric space", in *Pattern Recognition Letters*, volume 1, pages 125–128, 1982.

[32] B. Kamgar and L. N. Kanal, "An improved branch and bound algorithm for computing k-nearest neighbors", in *Pattern Recognition Letters*, volume 3, pages 7–12, 1985.

[33] N. Roussopoulos, S. Kelley, and F. Vincent, "Nearest Neighbour Queries", in *ACM SIGMOD*, volume 24, pages 71–79, 1995.

[34] S. A. Nene and S. K. Nayar, "A Simple Algorithm for Nearest Neighbor Search in High Dimensions", in *IEEE Transactions on Pattern Analysis and Machine Intelligence*, volume 19, pages 989–1003, 1997.

[35] T. C. Chiueh, "Content-Based Image Indexing", in *Proceedings of the 20th VLDB Conference*, pages 582–593, 1994.

[36] T. Bozkaya and M. Ozsoyoglu, "Distance-Based Indexing for High-dimensional Metric Spaces", in *SIGMOD Record*, volume 26, pages 357–368, 1997.

[37] J. C. Bezdek, *"Pattern Recognition with Fuzzy Objective Function Algorithms"*, Plenum Press, New York, 1987.

[38] E. Backer, "Cluster Analysis by Optimal Decomposition of Induced Fuzzy Sets", in *Delft, The Netherlands: Delft University Press*, 1978.

[39] F. Sets, "Pattern Recognition with Fuzzy Objective Function Algorithms", in *Plenum Pres*, pages 112–127, New York, 1981.

[40] J. Dunn, "A fuzzy relative of the isodata process and its use in detecting compact well-separated clusters", in *Journal of Cybernetics*, page 3:32, 1974.

[41] J. Dunn, "Well separated clusters and optimal fuzzy-partitions", in *Journal of Cybernetics*, pages 4:95–104, 1974.

[42] E. Ruspini, "A new approach to clustering", in *Information and Control*, pages 15(1):22–32, 1969.

[43] L. Zadeh, "Similarity relations and fuzzy orderings", in *Information Sciences*, volume 3, pages 177–200, 1970.

[44] S. Ovchinnikov, "r fuzzy and p fuzzy, Fuzzy Sets and Systems", in *Similarity relations, fuzzy partitions, and fuzzy orderings*, pages 40:107–126, 1991.

[45] R. Dav and E. Krishnapuram, "Robust clustering method: a unified view", in *IEEE Transactions on Fuzzy Systems*, volume 5, pages 270–293, 1997.

[46] R. Krishnapuram and J. Keller, "A Possibilistic Approach to Clustering", in *IEEE Transactions on Fuzzy Systems*, volume 1, pages 98–110, 1993.

[47] L. Zadeh, "Fuzzy Sets", in *Information and Control*, volume 8, pages 338–353, 1965.

[48] J. MacQueen, "Some methods for classification and analysis of multivariate observations", in *Proceedings of the 5th Berkeley Symposium on Mathematical Statistics and Probability*, pages 281–297, 1988.

[49] L. E. Scales, *"Introduction to Non-Linear Optimization"*, Springer-Verlag, New York, 1985.

[50] N. Pal, K. Pal, and J. Bezdek, "A mixed c-means clustering model", in *Fuzzy Systems, 1997., Proceedings of the Sixth IEEE International Conferenc*, volume 1, pages 11–21, 1997.

[51] K. K. Chintalapudi and M. Kam, "A noise-resistant fuzzy c means algorithm for clustering", in *Fuzzy Systems Proceedings, 1998. IEEE World Congress on Computational Intelligence., The 1998 IEEE International Conference*, volume 2, pages 1458–1463, 1998.

[52] R. Dave, "Characterization and Detection of Noise in Clustering", in *Pattern Recognition Letters*, volume 12, pages 657–664, 1991.

[53] Y. Ohashi, "Fuzzy Clustering and Robust Estimation", in 9^{th} *Meet. SAS Users Grp. Int.*, 1984.

[54] H. Yin, R. Lengelle, and P. Gaillard, "Inverse-step competitive learning", in *Neural Networks, 1991. 1991 IEEE International Joint Conference on 1991*, volume 1, pages 839–844, 1991.

[55] X. Yong, Y. Guangqun, C. Hexin, and D. Yisong, "A new competitive learning algorithm for vector quantization based on the neuron winning probability", in *Intelligent Processing Systems, 1997. ICIPS '97. 1997 IEEE International Conference*, volume 1, pages 485–488, 1997.

[56] B. Kosko, "Stochastic competitive learning", in *Neural Networks, 1990., 1990 IJCNN International Joint Conference on 1990*, volume 2, pages 215–226, 1990.

[57] C. Sun, X. Yu, and X. Feng, "Adaptive clustering of stock prices data using cascaded competitive learning neural networks", in *Systems, Man and Cybernetics, 1996., IEEE International Conference on Volume: 3, 1996*, volume 3, pages 2359–2363, 1996.

[58] L. Xu and Y. M. Cheung, "Adaptive supervised learning decision networks for traders and portfolios", in *Computational Intelligence for Financial Engineering (CIFEr), 1997., Proceedings of the IEEE/IAFE 1997*, pages 206–212, 1997.

[59] D. Kim and D. Cho, "Texture segmentation using competitive learning algorithm with pyramid approach", in *Advanced Robotics, 1997. ICAR '97. Proceedings., 8th International Conference on 1997*, pages 851–856, 1997.

[60] B. Watkins and M. Tummala, "Classification vector quantization of image data using competitive learning", in *Image Processing, 1994. Proceedings. ICIP-94., IEEE International Conference*, volume 3, pages 922–925, 1994.

[61] D. McNeill and H. Car, "Competitive learning algorithms in adaptive educational toys", in *Artificial Neural Networks, Fifth International Conference on (Conf. Publ. No. 440) , 1997*, pages 205–209, 1997.

[62] D. Pollard, "A central limit theorem for k-means clustering", in *Annals of Probability*, pages 10:919–926, 1982.

[63] D. Pollard, "Strong consistency of k–means clustering", in *The Annals of Statistics*, volume 9, pages 135–140, 1981.

[64] B. Juang and L. Rabiner, "The segmental k-means algorithm for estimating parameters of hidden Markov models", in *IEEE Transactions on Acoustics, Speech, and Signal Processing*, volume 38, pages 1639–1641, 1990.

[65] L. Xu, "Rival penalized competitive learning, finite mixture, and multisets clustering", in *IJCNN'98*, pages 2525–2530, 1998.

[66] K. Chintalapudi and M. Kam, "A noise-resistant fuzzy c means algorithm for clustering", in *The 1998 IEEE International Conference*, volume 2, pages 1458–1463, IEEE World Congress on Computational Intelligence., 1998.

[67] R. Krishnapuram and J. Keller, "A Possibilistic Approach to Clustering", in *IEEE Transactions on Fuzzy System*, volume 1, pages 98–110, 1993.

[68] R. Krishnapuram and J. Keller, "The possibilistic C-means algorithm: insights and recommendations", in *Fuzzy Systems, IEEE Transaction*, volume 4 3, pages 385–393, 1996.

[69] M. Barni, V. Cappellini, and A. Mecocci, "Comments on a possibilistic approach to clustering", in *IEEE Transactions Fuzzy System*, volume 4, pages 393–396, 1996.

[70] J. Zhang and L. Zhang, "Learning fuzzy concept prototypes using genetic algorithms", in *Fuzzy Systems Conference Proceedings, 1999. FUZZ-IEEE '99. 1999 IEEE International*, volume 3, pages 1790–1795, 1999.

[71] E. Smith and D. Osherson, "Conceptual combination with prototype concepts", in *Cognitive Science 8(4)*, pages 337–361, 1984.

[72] A. Hinneburg and D. A. Keim, "Optimal Grid-Clustering: Towards Breaking the Curse of Dimensionality in High-Dimensional Clustering", in *The VLDB Journal*, pages 506–517, 1999.

[73] K. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft, "nearest neighbour", in *ICDT conference*, pages 217–235, 1999.

[74] C. Blake and C. Merz, "UCI Repository of machine learning databases", 1998.

[75] T. Zhang, R. Ramakrishnan, and M. Livny, "BIRCH: A New Data Clustering Algorithm and Its Applications", in *Data Mining and Knowledge Discovery*, volume 1, pages 141–182, 1997.

[76] A. Fu, Y. L. Cheung, and Y. S. Moon, "Dynamic VP-Tree Indexing for N-Nearest Neighbor Search Given Pair-Wise Distances", in *VLDB Journal, Springer*, volume 9, Issue 2, pages 154–173, 2000.