Using Natural Clusters Information to Build Fuzzy Indexing Structure

H.Y. Yue, I. King and K.S. Leung

Department of Computer Science and Engineering
The Chinese University of Hong Kong
Shatin, New Territories, Hong Kong
Email: {hyyue, king, ksleung}@cse.cuhk.edu.hk

Abstract

Efficient and accurate information retrieval is one of the main issues in multimedia databases. However, the key for this is how to build an efficient indexing structure. In this paper, we demonstrate how to use a fuzzy clustering algorithm, Sequential Fuzzy Competitive Clustering (SFCC), to get the natural clusters information from the data. Then use the information to build an efficient indexing structure, SFCC-binary tree (SFCC-b-tree). We will show in the experimental results that SFCC-b-tree performs better that VP-tree in most of the cases.

1 introduction

Efficient and accurate information retrieval is one of the main issues in multimedia databases. The content-based retrieval lets users to specify queries by features (or contents) such as *color*, *texture*, *sketch* and *shape* to retrieve database objects with features similar to the queries.

Many content-based retrieval multimedia database systems have been developed in the past few years. For example, Montage [1] is an image database for managing and retrieving that supports content-based retrieval by *color histogram*, *sketch*, *texture*, and *shape*. Query by Image Content (QBIC) [2] allows queries on databases based on color, texture, and shape of database objects.

1.1 Nearest-neighbor Searching

By using feature vectors, the content-based retrieval multimedia databases support similar searching. Applying a suitable distance function to the feature vectors as the similarity measurement, the database objects will be ranked according to a query. The top ranked objects are then retrieved as the result for similar retrieval. Nearest-neighbor search is a typical kind of similar searching. In the feature vector space or the real space, a query can be seen as a multi-dimensional point (or vector). Thus, the retrieved objects of this query are the nearest points around the query point.

An efficient nearest-neighbor search requires an indexing structure, which generates partitions for the feature vector space. Based on this indexing structure, only the objects in one or a few of partitions instead of in the whole feature vector space need to be visited during a nearest-neighbor search. Thus, the key issue of an indexing method is how to partition the feature space [3].

1.2 Boundary Problem

Typically, data form natural clusters. Also, the results of nearest-neighbor search and the query point lies in the same natural cluster. However, most of the indexing methods for content-based retrieval in databases such as R*-tree [4], SR-tree [5], k-d tree [6], VP-tree [7], MVP-tree [8] does not consider the natural clusters in the feature space and often divide the objects in the same natural cluster into several different nodes. As a result, these existing indexing methods usually confront the problem of performance degradation when the queries lie near the generated partition boundaries.

We build an indexing structure which partitions feature vector space based on the natural clusters to decrease the influence of the boundary problem. In this indexing structure, the partitioning boundaries approximately are the natural clustering boundaries. The nearest-neighbor search on this indexing structure will become more efficient.

In the next section, we will introduce a noise resistance clustering algorithm, *Sequential Fuzzy Competitive Clustering (SFCC)*. In Section 3, we will describe about how to build an efficient indexing structure, *SFCC Binary Tree (FCC-b-tree)* by using *SFCC*. In the same section, we will demonstrate how to make use of *Minimum Boundary Rectangle (MBR)*, which is the smallest rectangle containing all the data objects for the node, to perform nearest-neighbor search efficiently. In Section 4,

some experimental results will be shown. Then we will come to our discussion and conclusion part in Section 5 and Section 6 respectively.

2 Fuzzy Clustering Methods for Indexing

In this section, We will introduce an efficient fuzzy clustering algorithm for content based indexing in order to lessen the boundary problems mentioned in the previous section.

2.1 Sequential Fuzzy Competitive Clustering (SFCC)

The sequential is an online clustering algorithm. Which means that we do not need to get the whole training data set before clustering. On the other hand, as the number of data objects exist in database become larger and larger, it becomes impossible to use off-line clustering algorithms for such a database.

The algorithm of SFCC is outlined as follows.

(Step 0) Initialization: Every cluster in SFCC is describe by a fuzzy prototype. In the initialization, we randomly pick k points as the k initial cluster prototype centers and every prototype have 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:

$$u_{ik} = \lambda_i \times (\prod_{j=1}^{a} u_{jik})^{1/a} \ , \qquad (1)$$

where a is the the number of attribute and u_{jik} is the membership value of data instance x_k to cluster i in j^{th} dimension. λ_i is a value used to prevent starvation. It is defined as:

$$\lambda_i = 1 - (\frac{\text{number of winning by cluster}_i}{\text{number of total loops}}) \qquad (2)$$

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

$$u_{jik} = \frac{\sigma_{ji} + 1}{\sigma_{ji} + d_j(i,k) + 1} \ , \qquad (3)$$

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

After 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:

$$m_w^{'} = m_w + \alpha \times u_{iw} \times (x_k - m_w) \ , \qquad (4)$$

$$\sigma_{iw}^{'} = \sigma_{iw} + \alpha \times u_{iw} \times (d_j(i,k) - \sigma_{iw}) \ , \qquad (5)$$

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 prespecified value. The final cluster prototypes are the results of the SFCC.

3 SFCC-b-tree

3.1 Generating Top-Down Indexing Structure

After the SFCC clustering is finished, the next step is to build an indexing structure. We realize the indexing structure using a top-down hierarchical clustering approach, which clusters the next level of feature vectors only based on the subset of data points partitioned in the previous level. Thus, the hierarchical clustering approach transforms a feature vector space into a sequence of nested partitions.

In this approach, the clusters at the same level do not overlap each other. Therefore, we can use a tree to describe the relationships between the different clustering levels. Especially, if we set a restriction that each set of data points in a level can only be partitioned into at most two subsets in the next clustering level, this tree becomes a binary tree. After clustering, all the feature vectors are in one cluster at the root level, level 0, and there are at most 2^i subtrees (clusters) at level i.

We can use the binary tree as indexing structure for nearest-neighbor search. The basic idea is that, at the root node of the binary tree, a query vector $\vec{q}$ is compared to the fuzzy prototypes of the clusters in the immediate lower level. The child node corresponding to the cluster with $\vec{q}$ having the highest membership value is selected. The elements in the selected cluster will be the result of the query if they satisfy the criteria of the nearest-neighbor search. Otherwise, the search will proceed to the lower levels.

Next, we outline the procedure of building a binary indexing tree using SFCC clustering. We name the tree SFCC-b-tree, SFCC Binary Tree.

3.2 Building SFCC-b-tree

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. There are two kinds of nodes in the tree: *leaf node* and *non-leaf node*.

All nodes contain all the information for the cluster represented by that node. It includes:

1. Total number of instances in that subtree.
2. Cluster prototype of that cluster node.
3. Minimum boundary rectangle table (MBR), table for that node.

However, for a non-leaf node, it also contains two pointers pointing to its the child nodes. While for a leaf node, it also contains a cluster of at most M data points calculated by SFCC clustering. M is the maximum number of data in a leaf node.

The algorithm for building the hierarchical binary tree by using SFCC clustering is shown belows:

Algorithm 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
```

3.3 Update of SFCC-b-tree

Considering the update of SFCC-b-tree, we next design two operations which insert or delete a single feature vector to or from the SFCC-b-tree without re-clustering. Next, we show the two update operations.

3.3.1 Insertion

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

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.

3.3.2 Deletion

Apart from insertion, we can also delete an individual feature vector from a SFCC-b-tree.

The deletion algorithm makes the searching performance worse. When the number of deletions increases, the searching performance will decrease because node merging will change the original indexing structure. Sometimes, the resultant indexing tree will give better searching results especially when the number of deletions is relatively small. It is because only a few points are removed from the indexing tree and it does not affect the natural clusters and the indexing structure any more.

Next, we show how to realize retrieval using nearest nearest-neighbor search based on the SFCC-b-tree indexing structure.

3.4 Similar Searching in SFCC-b-tree

In SFCC-b-tree, we use a depth first search algorithm with node pruning to perform similar search. In the information tag of each node, we have a minimum boundary rectangle table, which is the smallest rectangle containing all the data objects for the node. Also, for each query point, we use a boundary square to represent the query. In similar search, datum can be a possible result of a query only if the distance between the datum and query point is smaller then a boundary. In similar range search, it is a pre-defined value while it is the k^{th} smallest distance in k-nearest neighbor search, where k is the number of result data points needed.

As a result, query results will exist in a tree node P if and only if there exist overlapping between the query boundary square and the minimum boundary rectangle. Two d-dimensional hyper-cube exist

overlapping, if and only if the following holds:

Rule 3.1 (General Overlapping Rule) *Given two n-dimensional hyper-cube P and Q exist overlapping, if and only if P and Q have overlapping on all the n dimension.*

With Rule 3.1, we are able to preform similar search easily. Here is the searching algorithm for SFCC-b-tree:

Algorithm 2 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
```

With the $k$nn similar search algorithm, it is very easy to chage it into range similar search algorithm by make b be a hyper-square with constant length.

4 Experimental Results

In this section, we are going to define the *efficiency* for the SFCC-b-tree indexing method and demonstrate it through a set of experiments.

Definition 4.1 (Efficiency Measurement) *Let, x is the number of instances reached for the checked method, y is the number of instances reached in linear search, and z is the size of data.*

$$\begin{aligned} efficiency &= 1 - \frac{x}{y} \\ &= 1 - \frac{x}{z}. \end{aligned} \tag{6}$$

We test our indexing method and compares it with VP-tree in the following experiments:

1. Data in different dimensions.
2. Different number of total data.
3. Different leaf node size.

4.1 Experiment 1: Data in different dimensions

In this section, we will compare the performance of SFCC-b-tree and VP-tree data in different dimension. Data set used in this experiment have a total size of 10,000 data points and 10 clusters with different number of data points in Gaussian distribution.

Both the SFCC-b-tree and VP-tree have a leaf node size of 200. After indexing structure are generated, we performs 100 queries of $k$nn search to them with k equal to 100.

4.1.1 Experiments Results

After a series of data retrieval, we calculate their retrieval performance and found the following:

Table 1: Building time comparison between SFCC-b-tree and VP-tree in different dimension. (in second)

Dimension	SFCC-b-tree	VP-tree
2	77.62	210.63
5	157.04	287.50
10	261.30	422.05
20	468.29	659.43

Table 2: Running time comparison between SFCC-b-tree and VP-tree with k equal to 100 in $k$nn search and different dimension. (in second)

Dimension	SFCC-b-tree	VP-tree
2	0.03	0.55
5	0.06	1.02
10	0.10	1.75
20	0.14	3.23

Table 3: Efficiency comparison between SFCC-b-tree and VP-tree with k equal to 100 in $k$nn search and different dimension.

Dimension	SFCC-b-tree	VP-tree
2	0.906	0.699
5	0.874	0.577
10	0.830	0.464
20	0.558	0.325

The results are summarized in Figure 1.

4.2 Experiment 2: Different number of total data.

The dimension of the data set is fixed to 10 and the leaf node size is set to 200 in this experiment. Data set used in this experiment have a total size of various from 1000 to 100,000 data points and 10 clusters with different number of data points in Gaussian distribution. After indexing structure are generated, we performs 100 queries of $k$nn search to them with k equal to 100.

4.2.1 Experiments Results

After a series of data retrieval, we calculate their retrieval performance and found the following:

Table 4: Building time comparison between SFCC-b-tree and VP-tree in different data set size. (in second)

Data set size	SFCC-b-tree	VP-tree
1000	10.42	3.74
10,000	261.30	422.05
20,000	952.23	1679.73
50,000	2050.67	8145.08

Table 5: Running time comparison between SFCC-b-tree and VP-tree with k equal to 100 in $k$nn search and different data set size. (in second)

Data set size	SFCC-b-tree	VP-tree
1000	0.01	0.19
10,000	0.10	1.75
20,000	0.05	3.46
50,000	0.18	8.42

Table 6: Efficiency comparison between SFCC-b-tree and VP-tree with k equal to 100 in $k$nn search and different data set size.

Data set size	SFCC-b-tree	VP-tree
1000	0.679	0.325
10,000	0.830	0.464
20,000	0.844	0.420
50,000	0.851	0.691

The results are summarized in Figure 2.

4.3 Experiment 3: Different leaf node size

The dimension of the data set is fixed to 10 and the data set used in this experiment have a total size of 10,000 and 10 clusters with different number of data points in Gaussian distribution. Both the SFCC-b-tree and VP-tree have a leaf node size various from 100 to 2,000 in experiment 3. After indexing structure are generated, we performs 100 queries of $k$nn search to them with k equal to 100.

4.3.1 Experiments Results

After a series of data retrieval, we calculate their retrieval performance and found the following:

Table 7: Building time comparison between SFCC-b-tree and VP-tree in different dimension leaf node size. (in second)

Node size	SFCC-b-tree	VP-tree
100	523.51	422.99
200	261.30	422.05
500	97.76	409.17
1000	43.15	395.08
2000	18.60	384.23

Table 8: Running time comparison between SFCC-b-tree and VP-tree with k equal to 100 in $k$nn search and different leaf node size. (in second)

Node size	SFCC-b-tree	VP-tree
100	0.09	1.65
200	0.10	1.75
500	0.14	1.88
1000	0.15	1.96
2000	0.17	2.21

Table 9: Efficiency comparison between SFCC-b-tree and VP-tree with k equal to 100 in $k$nn search and different leaf node size.

Node size	SFCC-b-tree	VP-tree
100	0.896	0.496
200	0.830	0.464
500	0.830	0.425
1000	0.824	0.394
2000	0.787	0.325

The results are summarized in Figure 3.

5 Discussion

After three experiments, we have the following conclusions.

- **About Building Time**
  The building time increase with:
  1. Higher the dimension.
  2. Larger the data set.
  3. Smaller the leaf node size.
- **About Searching Time**
  1. Higher the dimension, lower the efficiency.
  2. Larger the data set, higher the efficiency.
  3. Smaller the leaf node size, higher the efficiency.
- **SFCC-b-tree performs better than VP-tree in all the test cases.**

The results about the building time is very typically. Higher dimension and larger data set means higher complexity, so it needs longer time to build the indexing structure. In the experiments for searching time, it shows that larger the data set, higher the performance. It also shows that smaller the leaf node size, higher the performance. It is

mainly because they increase the data set size / leaf node size ratio. On the other hand, as VP-tree does not consider natural cluster information in building time. The performance is worse than SFCC-b-tree.

6 Conclusion

In this paper, we have demonstrate how to use SFCC to build indexing structure and show that SFCC-b-tree having a better performance than VP-tree. We use MBR to prune those nodes do not contain possible results for a query. However, it is possible to use some other shape to define the boundary and it may be the future works for this project.

References

[1] I. King, A. Fu, L. Chan, and L. Xu, "Montage: An image database for the hong kong's textile, fashion, and clothing industry," 1995. http://www.cse.cuhk.edu.hk/∼miplab.

[2] W. Niblack, R. Barber, W. Equitz, M. Flickner, E. Glasman, D. Petkovic, P. Yanker, C. Faloutsos, and G. Taubin, "The qbic project: querying images by content using color, texture, and shape," in *Proceedings of the SPIE - The International Society for Optical Engineering*, vol. 1908, pp. 173–187, 1993.

[3] I. King and T. K. Lau, "Comparison of several partitioning methods for information retrieval in image databases," in *Proceedings of the 1997 International Symposium on Multimedia Information Processing (ISMIP'97)*, pp. 215–220, 1997.

[4] N. Beckmann, H. P. Kriegel, R. Schneider, and B. Seeger, "The r*-tree: an efficient and robust access method for points and rectangles," *ACM SIGMOD Record*, vol. 19, no. 2, pp. 322–331, 1990.

[5] N. Katayama and S. Satoh, "The sr-tree: an index structure for high-dimensional nearest neighbor queries," *SIGMOD Record*, vol. 26, pp. 369–380, June 1997.

[6] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Communications of the ACM*, vol. 18, no. 9, pp. 509–517, 1975.

[7] T. C. Chiueh, "Content-based image indexing," in *Proceedings of the 20th VLDB Conference*, pp. 582–593, Sept. 1994.

[8] T. Bozkaya and M. Ozsoyoglu, "Distance-based indexing for high-dimensional metric spaces," *SIGMOD Record*, vol. 26, pp. 357–368, June 1997.

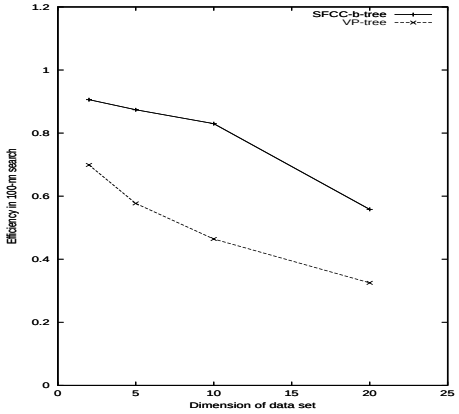

Figure 1: Efficiency comparison between SFCC-b-tree and VP-tree with k equal to 100 in $k$nn search and different dimension.

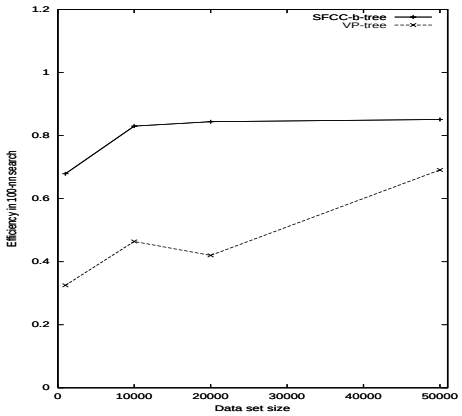

Figure 2: Efficiency comparison between SFCC-b-tree and VP-tree with k equal to 100 in $k$nn search and different data set size.

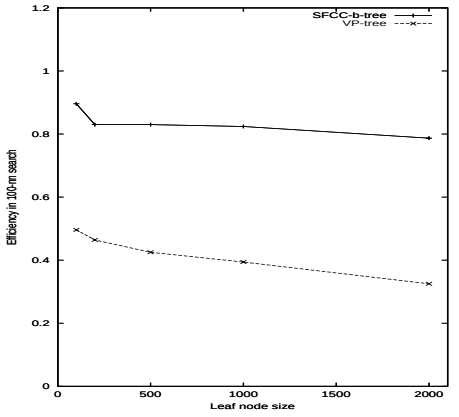

Figure 3: Efficiency comparison between SFCC-b-tree and VP-tree with k equal to 100 in $k$nn search and different leaf node size.