AN EFFICIENT INDEX STRUCTURE FOR HIGH DIMENSIONAL IMAGE DATA

Myung Keun Shin[1], Soon Young Huh[1],
Seok Hee Lee[2], Jae Soo Yoo[2], Ki Hyoung Jo[2] ,
and Jang Sun Lee[3]

[1] *KAIST Graduate School of Management, Seoul, South Korea, 130-012*
[2] *Department of Computer & Communication Engineering,*
Chungbuk National University, San 48, Gaesin Dong,
Heungduk Ku, Cheongju, Chungbuk, South Korea, 361-763
[3] *Internet Service Department, Computer and Software Technology Lab.*
Electronics and Telecommunications Research Institute
Kajong-Dong, Yusong-Gu, Taejon, 305-350, Korea

Abstract. The existing multi-dimensional index structures are not adequate for indexing higher-dimensional data sets. Although conceptually they can be extended to higher dimensionalities, they usually require time and space that grow exponentially with the dimensionality. In this paper, we analyze the existing index structures and derive some requirements of an index structure for content-based image retrieval. We also propose a new structure, called CIR(Content-based Image Retrieval)-tree, for indexing large amount of point data in high dimensional space that satisfies the requirements. In order to justify the performance of the proposed structure, we compare the proposed structure with the existing index structures in various environments. We show, through experiments, that our proposed structure outperforms the existing structures in terms of retrieval time and storage overhead.

1. Introduction

Many recent applications such as image databases, medical databases, GIS and CAD/CAM require enhanced indexing for content-based image retrieval. Content-based image retrieval is to query large on-line databases using the content of images as the basis of queries. The Content examples include the color, texture, and shape of image objects and regions. In the applications that need content-based retrieval, indexing of high-dimensional data has become increasingly important for fast retrieval. For example, in image databases, the image objects are usually mapped to feature vectors in some high-dimensional space. The queries are processed against a database that consists of feature vectors. The index structures for the content-based retrieval also efficiently need to process similarity queries that are related to some measure of similarity between feature vectors.

There are several index structures for high dimensional data such as SS-tree [5], TV-tree [11], X-tree [20] and SR-tree [16]. The SS-tree was proposed as an index structure to efficiently support similarity search. The idea of TV-tree comes from the observation that, in most high-dimensional data sets, a small number of the dimensions bears most of the information. The main idea of the X-tree is to avoid the overlap of bounding boxes in the directory by using a new organization of the directory that is optimized for high-dimensional space. The SR-tree is an extension of the R*-tree [15] and the SS-tree [5]. The SR-tree uses both bounding spheres and bounding rectangles to improve the performance on nearest neighbor queries. However, they are not suitable for an indexing structure for content-based retrieval, because they usually require time and space that grow exponentially with the dimensionality [18], although conceptually they can be extended to higher dimensionalities. The Pyramid-Technique [21] was pro-

posed based on a special partitioning strategy to break the so-called curse of dimensionality. It is suitable for high-dimensional range queries.

In this paper, we derive some design requirements of an index structure for content-based image retrieval. We also propose a new structure, called CIR (Content-based Image Retrieval)-tree, for indexing large amounts of point data in high dimensional space that satisfies the derived requirements. We performed extensive experiments with a synthetic uniformly distributed data as well as a real data. The relationships among various performance parameters are thoroughly investigated. We show through performance comparison based on experiments that regardless of data distribution, the CIR-tree significantly improves performance in both of the retrieval time and the storage overhead over TV-tree, X-tree and Pyramid-Technique.

The remainder of this paper is organized as follows. In section 2, we describe related work. In section 3, we present a few requirements of an index structure for content-based retrieval. In section 4, we propose a new indexing structure that satisfies the requirements. Section 5 performs experiments to show that the proposed index structure outperforms existing index structures. Finally, conclusions are described in section 6.

2 Related Work

The R-tree [8] and its most successful variant, the R*-tree [15] have been used most often for indexing high dimensional data in the database literature. The R-tree is a height-balanced tree corresponding to the hierarchy of nested rectangles. The rectangle of an internal node is determined by the minimum bounding rectangle of those of its children. The rectangle of a leaf node is determined by the minimum bounding rectangle of the data entries contained in that leaf. Therefore, the rectangle of the root node corresponds to the minimum bounding rectangle of the whole data entries, while the rectangle of an internal node corresponds to the minimum bounding rectangle of the data entries contained in its lower leaves.

The R*-tree [15] has two major enhancements over the R-tree. First, rather than considering the area only, it minimizes margin and overlap of each enclosing rectangle in the internal nodes. Second, the R*-tree introduces the notion of forced reinsert to make the shape of the tree less dependent on the order of insertion. However, the R-tree and the R*-tree explode exponentially with the dimensionality, eventually reduce to sequential scanning.

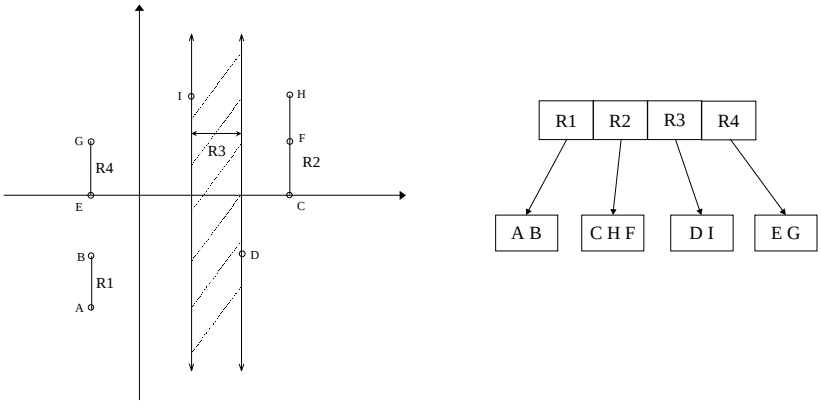


Fig. 1. An example of TV-tree. In the root level, region R3 uses only one dimension for discrimination. But other regions use two dimensions for discrimination.

The TV-tree [11] is a method in the database literature that was proposed specifically for indexing high-dimensional data. The basis of the TV-tree is to use dynamically contracting and

extending feature vectors. That is, it uses as little features as possible that are necessary to discriminate the objects. An example of TV-tree is given in Fig. 1. The points designated from A to I denote data points (only the first two dimensions are shown). In the root level, region R3 uses only one dimension for discrimination. But other regions use two dimensions for discrimination.

However, the TV-tree would not handle overlap properly. To solve the overlap problem of the TV tree, our proposed CIR-tree will suggest an improved ChooseSubtree algorithm that chooses the most appropriate node for inserting an image object. The detail of the algorithm is described in section 4.3.1. Also the CIR-tree adapts the supernode concept in Split algorithm to alleviate the overlap problem.

The X-tree [20] was proposed as an index structure to avoid splits that would result in a high degree of overlap in the directory. To do this, the X-tree uses a split algorithm that minimizes overlap and additionally uses the concept of supernodes. Supernodes are large directory nodes of variable size(a multiple of the usual block size). Supernodes are created during insertion to avoid splits in the directory that would result in highly overlapped structure. The X-tree uses the notion of maximum overlap value (*MaxO*) to decide whether it splits a node or extends a node to a supernode. Most insertion algorithm split a node into two in case there occurs an overflow. But, if the overlap of the two split nodes is larger than the *MaxO*, the X-tree extends the original node into a supernode instead of splitting it. The suggested value of the *MaxO* in [20] is 20%.

Due to the fact that the overlap is increasing with the dimension, the number and size of supernodes increase with the dimension [20]. Fig. 2 shows three examples of X- tree with different dimensionalities.

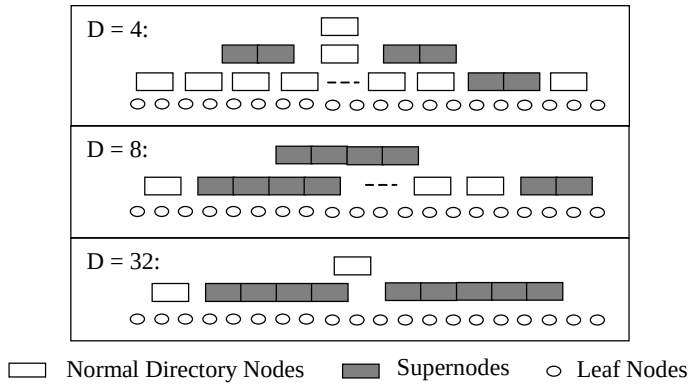


Fig. 2. Various shapes of the X-tree in different dimensions

Although the overlap was reduced in the directory, the X-tree loses the efficiency of hierarchical structure. In Fig. 2, when the number of dimensions D is 32, the structure of the X-tree looks linear because of large supernodes. However, because our CIR-tree uses smaller feature vector than the X-tree in the directory, the size of supernodes will be decreased. Of course the total size of the directory will be decreased.

3 Design requirements for high dimensional index structures

Under the condition that the features of images have been extracted, we analyze the properties of the previously proposed high dimensional index structures and present desired design requirements for high dimensional index structures. The design requirements are as follows.

The index structure must deal with high dimensional features efficiently.

Index structures for a content-based image retrieval system must deal with high dimensional image features. Most existing multi-dimensional index structures are not adequate for handling high dimensionality. When going to higher dimensions, they become extremely inefficient because the number of nodes increases exponentially. When the index structure is constructed, the number of nodes should not increase exponentially as the number of dimension increases.

The overlap between directory regions must be minimized.

In general, an overlap means a region that is covered by more than one directory area. As the amount of data and the height of a tree increase, the overlap area increases remarkably with growing dimensionality of data. Usually, since the overlap increases the number of paths to be traversed, it produces bad effects on processing queries. As a result, the new index structure should provide an algorithm to minimize the overlap.

Storage utilization must be optimized.

Higher storage utilization generally reduces the query cost since the height of the tree would be kept low. Eventually, query types with large query regions are more likely to be influenced since the concentration of regions in several nodes will have a stronger effect if the number of found keys is high.

The index structure must be appropriate to similarity retrieval.

Unlike conventional database systems, a content-based image retrieval system processes queries based on similarity since images are not atomic symbol and unformatted data. Therefore the index structure must process similarity queries efficiently.

The index structure must employ a similarity measure that can evaluate well similarity between high-dimensional features.

In content-based retrieval system, image features are expressed as points in multi-dimensional space. We use the Euclidean distance between two point objects as a similarity. In general, since the dimensions of image features are independent with one another and different in respect of relativity and distribution, measuring the similarity between two objects with just Euclidean distance measure suffers limits on exactness. As a result, another similarity measure must be employed.

The index structure must process various query types efficiently.

An index structure has to be able to process various query types such as exact match query, partial match query, range query and k-nearest neighbor search query. We want a structure uniformly good at every query rather than very good at some queries but poor at other queries.

An index structure for content-based image retrieval system has to deal with high-dimensional features dynamically.

Though there are certain applications having archival nature, i.e., insertions are less frequent and updates/deletions are seldom necessary, the content-based image retrieval system in practice requires a dynamic information storage structure.

4. CIR(Content-based Image Retrieval)-tree

4.1. Characteristics

Various index structures for high dimensional data sets have been proposed. However, most of them have the dimensionality problem, as surveyed in the previous sections, eventually loosing the efficiency as an index structure. TV-tree and X-tree are the index structures proposed to support efficient query processing of high-dimensional data. It is true that they are more adequate index structure for high-dimensionality than existing index structures such as R-tree and its variants. As we mentioned in the section 2, however, they suffer from processing image data with a large number of features.

We propose a new high dimensional index structure, called CIR-tree, in order to alleviate the problem. The proposed CIR-tree satisfies the design requirements mentioned in section 3. The idea of CIR-tree came from the insights of these two structures, that is, the main characteristics of the X-tree and the TV-tree. We applied the main idea of both tree structures to CIR-tree in order to solve the dimensionality problem, and enhance the reinsert algorithm. For the nodes that are close to the root node, we use just a few dimensions so that we can store more branches and obtain a high fan out. On the other hand, we use more and more dimensions as descending tree so that we can see more discrimination. In the CIR-tree, it is assumed that feature vectors for data objects are ordered in ascending order by its importance, and the importance can be obtained by employing various conversion functions [11].

Like other index structures, CIR-tree represents data with hierarchical structure. A node in one level has its children nodes. This constitutes a hierarchical structure starting from a root node to leaf nodes. An internal node includes the MBRs of its children nodes, and a leaf node has feature vectors. The CIR-tree alleviates disadvantages of the index structures of R-tree group. According to experimental evaluation of overlap in the R*-tree directories, overlap increases to about 90% for high dimensionality larger than 5 [20]. The increase of overlap deteriorates the performance of index structure remarkably. The overlap can be increased when a node is split or a record is inserted. The CIR-tree uses supernode concept of X-tree to reduce the number of node splits and a split algorithm to avoid overlap when overflow occurs. That is, the CIR-tree avoids overlap whenever it is possible without allowing the tree to degenerate. Otherwise, the CIR-tree uses extended variable size directory nodes, so-called supernodes. Therefore the structure of the CIR-tree is the mixture type of the linear array structure for representing supernode and the hierarchical structure of the R-tree.

The CIR-tree uses forced reinsert operations that re-group entries between neighboring nodes and thus decrease the overlap. The CIR-tree uses the concept of weighted center, or the average coordinate of each entry, to enhance the reinsert algorithm. The use of the weighted center significantly improves the clustering effect of nodes in the CIR-tree. Therefore the CIR-tree has a chance to construct a condensed tree structure and to decrease the overlap between neighboring nodes.

In general, the existing index structures use Euclidean distance as a similarity measure on retrieval. However, the Euclidean distance is not appropriate as a distance measure for high dimensional data because of its exactness limit. To alleviate such a problem, CIR-tree uses the weighted Euclidean distance such as Equation #1. The weighted Euclidean distance processes various kinds of similarity queries more efficiently than the Euclidean distance.

$$D(X, Y) = \sqrt{(x - y)^T \, diag(w)(x - y)} \qquad \text{(Equation \#1)}$$

where, x and y are feature vectors and w is a vector representing relative weight.

4.2. Structure of the CIR-tree

The structure of the CIR-tree is similar to TV-tree except for supernodes. Each node consists of pointers to child branches, and a MBR represents a child node. The MBR is a minimum region containing all descendants of that branch, and has the feature vector as much as necessary for discrimination.

The data structures of MBR are as follows:

```
struct     MBR {   Feature    inactive,
   Feature    lower,
   Feature    upper };

struct Feature {   float feature_value[];
   int   no_of_dimensions };
```

where `Feature` denote 'feature vector'.

A directory node contains the MBRs that represent minimum bounding region of all their descendents. The data structure is as follows.

```
struct Branch_node {     int   no_of_element;
        list   of(MBR) };
```

A leaf node includes actual feature vectors. The structure of the leaf node is as follows.

```
struct Leaf_node   {     int   no_of_element;
        list of(Feature) };
```

A supernode is created when splitting a directory node. We will discuss the conditions of creating supernodes in section 4.3.2, when we describe the split algorithm. The structure of supernodes is represented as a continuous array of nodes

4.3. Algorithms in CIR-tree

4.3.1. Insertion algorithm

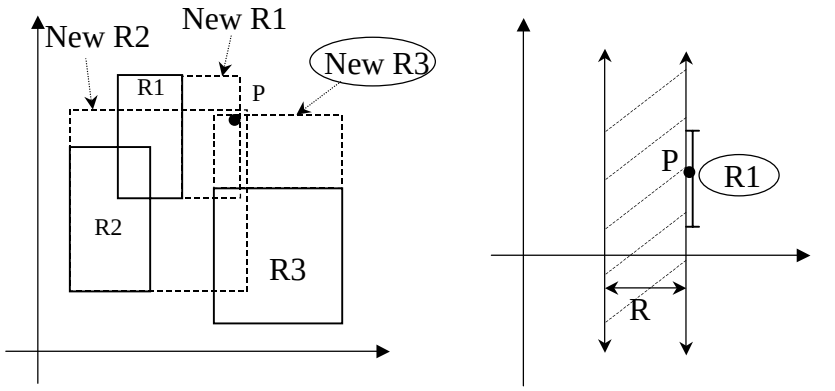
To insert a new object, we should find the branch at each level that is most suitable to hold the new object, and then insert the new object to the chosen leaf node. If overflow occurs at this time, we can cope with it by reinserting some entries in the node or splitting the node. After inserting, splitting, and reinserting a node, we update the MBRs of affected nodes.

The insertion algorithm calls ChooseSubtree algorithm first. ChooseSubtree is very important to make well-clustered tree structure. However, the TV-tree overlooks the clustering of data. But in the CIR-tree, the second criteria shown below clusters similar object together. Eventually, this reduces the overlap and significantly improves retrieval performance. The algorithm ChooseSubtree uses the following criteria, in descending priority:

Select the MBR that has minimum number of new pairs of overlapping MBR within the node. An example is in Fig. 3 (a).

Select the MBR that uses more dimensions for discrimination. Fig. 3 (b) shows only the first two dimensions. R1 and R2 are overlapped. R1 uses two dimensions for discrimination and R2 uses one dimension for it. The R2 may have more regions in the direction of the next dimension. When inserting the point P, R1 is selected and R2 is not because R2 uses more dimensions. Using more dimensions means that similar object are clustered in the small region.

Select the MBR whose center is close to a new object.



(a) R3 is selected because extending R1 or R2 will lead to a new pair of overlapping regions.

(b) R1 is selected because R1 uses two dimensions for discrimination, but R2 uses only first dimension.

Fig. 3. Illustration of the criteria of ChooseSubtree algorithm

When overflow occurs during insertion, the CIR-tree first performs a more enhanced reinsert algorithm than the R*-tree. We will discuss the enhanced algorithm in the following section. If another overflow occurs during reinsertion, the algorithm returns fail. Then it tries to split the node. If the area of overlap within the node exceeds certain predetermined threshold value in the split algorithm, the node is extended to supernode. The detail of split algorithm will be explained in section 4.3.3. The pseudo code for insertion algorithm is as follows.

Algorithm Insertion

```
1. ChooseSubtree() // choose the best branch to follow,
   // descend the tree until the leaf
   // node is reached
2. Insert a new object into the leaf node.
3. if(node overflows)
4.     Call Reinsert
5.     if(Reinsert fail)
6.         Call Split
5.         if( the split routine returns supernode)
6.              Extend the leaf node to supernode
7.         else
8.              Insert the MBRs of two split nodes
                into parent node
9. UpdateTree()   // update the MBRs of the parent node
```

4.3.2. Reinsert algorithm

In most tree structures for high dimensional data, including R-tree, R*-tree and TV-tree, different insertion orderings of a set of records results in different trees. For this reason data entries inserted during the early growth of the structure may have introduced bounding rectangles, which cause a bad retrieval performance in current situation. As a result, the trees suffer from the deterioration of tree performance.

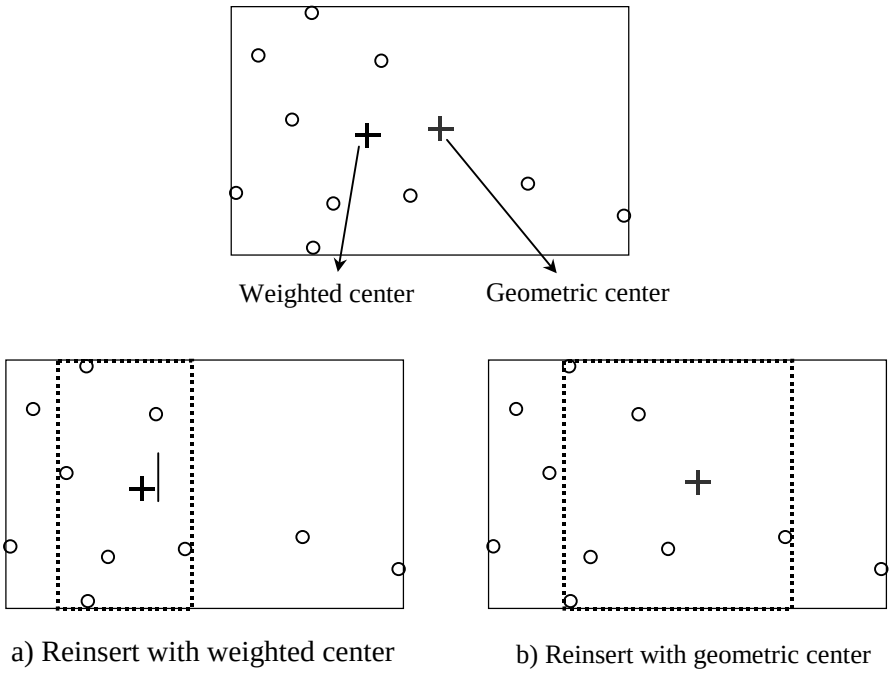


Fig. 4. Comparison of a weighted center with a geometric center where p = 40%. Reinsertion with weighted center makes smaller bounding rectangle.

The R-tree, the R*-tree and the TV-tree force the entries to be reinserted during the insertion routine. But the X-tree does not perform reinsertion. Therefore the X-tree has more overlapped bounding rectangles, and as a result, it makes large supernodes that may decrease the retrieval performance.

If overflow occurs, p entries farthest from the center of the node are deleted and they are reinserted from the top level. This provides a possibility of eliminating dissimilar entries from the node so that it accomplishes more efficient clustering. The parameter p can be varied in performance tuning stage. The experimentally suggested value of p in [15] is 30% of the maximum number of entries in a node.

The R*-tree and TV-tree use geometric center to find the farthest entries. The CIR-tree uses weighted center. The weighted center $\vec{c}$ of a node N is defined as:

$$\vec{c} = \frac{\sum_{i=1}^{n} \vec{e_i}}{n}$$

Where $\vec{e_i}$ denotes the center vector of the entry and n denotes the number of entries in a node N.

If we use the weighted center as a center of the node, the center of a node is moved toward a place where, in its vicinity, the entries are more densely. Figure 4 shows the effect of the weighted center when p = 40%. By using the weighted center, we can get smaller, or well-clustered MBRs after deleting the farthest entries. In addition, the smaller MBRs would decrease the overlap. The computation cost for weighted center is linear in the number of entries and in the number of dimensions. It is not a great burden in the whole algorithm.

```
Algorithm Reinsert
1. delete p% of entries from a node
2. insert them from the top level
3. if overflow occurs during insertion
4. return fail
5. else
6. return success
```

4.3.3 Split algorithm

The purpose of splitting is to divide the set of MBRs of vectors into two groups in order to facilitate upcoming operations and provide high space usability. The creation and extension of a supernode occur if there is no possibility to find a suitable hierarchical structure. In other words, if the dividing of the MBRs does result in large overlap split, we does not split the node but create a supernode of twice block size, or appending a block size if the current node is a supernode.

When splitting a node, we sort the entries by the first dimension, then look for the best break point in the sorted entries where the overlap of the two split MBRs gets the minimum. Of course, both of the two split nodes have larger size than minimum fill factor. If the overlap exceeds the *MaxO* value, the directory node would be extended to a supernode. As mentioned in section 2, we set the *MaxO* value to 20%.

```
Algorithm Split
1. find the first dimension with which overlap free split
   is Possible.
2. if the dimension found
3. do split
4. return the MBRs of two split nodes
5. else    // overlap free split is impossible
6. split with the first active dimension
7. If(overlap_ratio > 0.2) return supernode
8. else return the MBRs of two split nodes
9. end
```

4.3.4. Search algorithm

In this algorithm, the search starts from root node. It examines whether there is intersection between entries in the node and the search area or not. If the intersection exists, we traverse the child nodes of the entries. Because MBRs are allowed to overlap, multiple branches can be traversed. The following is the pseudo-code of search algorithm.

```
Algorithm Search
1. If(accessed node == Leaf node)
Evaluate the similarity of the query and the
        entries in the node.
```

```
3. Return the objects satisfying the query according
        to similarity.
4. else // for directory nodes
5. Select all MBRs including the query for
        active dimensions.
6. Call the search algorithm recursively with
        child node that the selected MBR points to.
7. end
```

4.3.5. Nearest Neighbor Search algorithm

We used the Hjaltason and Samet's algorithm [7] known as optimal. This algorithm uses the *MINDIST* values as a metric to prune the node from the search list. The *MINDIST* is an Euclidean distance between the query point and the nearest edge of the rectangle. Since the CIR-tree and the TV-tree do not use full dimensions to compute the *MINDIST* value, the effectiveness of pruning is degraded. To compensate this effect, the *MINDIST* value is re-scaled as

$$MINDIST' = MINDIST \times \frac{\#\ of\ full\ Dimension}{\#\ of\ active\ Dimension}$$

Of course this may not retrieve exact *k* nearest neighbors. But in the reinsertion algorithm, we use the distance as a metric to find the farthest entries, yielding well-clustered node. This makes the modified MINDIST valid as a metric for pruning. In our experiment on 10 nearest neighbor queries, only one or two records in the tail were different with the result of the X-tree. In the case of the TV-tree, that did not make well-clustered index structure as the CIR-tree, about 4 or 5 records are different with the result of the X-tree. This is because the choosing subtree and the reinsertion algorithm is not efficient as the CIR-tree. The following is the pseudo-code of nearest neighbor search algorithm.

```
Algorithm NN_Search
1. initialize SearchList with the child nodes of the root
2. sort SearchList by MINDIST
3. while (SearchList is not empty)
4.      if (top of SearchList is a leaf)
5.      find nearest point NNP
6.      if(NNP is closer than NN)
7.           prune SearchList with NNP
8.           let NNP be the new NN
9. else
10.          replace top of SearchList with its child nodes
11.     endif
12.     sort SearchList by MINDIST
13. endwhile
```

4.3.6. Deletion

Deletion is quite simple unless underflow occurs. In this case, the remaining entries of the node will be deleted and reinserted. The underflow may propagate to upper level.

4.3.7. The properties of CIR-tree

The proposed CIR-tree uses a variable number of dimensions when constructing tree to support high-dimensional data efficiently. For nodes that are close to the root node, we use just a few dimensions to store more data in a node. This tree provides higher fan out in the top levels, so the height of the tree becomes lower. In that result, the number of disk accesses reduces, similarity retrieval becomes easier, and the efficiency of storage space increases. It processes various query types more effectively, and facilitates deletion and insertion process as well. Also, the CIR-tree uses weighted Euclidean distance for more exactly evaluating similarity between a query and an object. Using supernode, it minimizes overlap, so it reduces the factors that deteriorate retrieval performance. However, since it employs weighted Euclidean distance, in order to give weight to each feature we need to get advice from domain experts.

5. Experiments

We show the characteristics of proposed CIR-tree by comparing its performance with those of TV-tree, X-tree and Pyramid-Technique. In this experiment, we used SUN SPARC station 20 with 128Megabytes of main memory and 6 Gigabytes of hard disk. All simulation programs were implemented with ANSI C++ and compiled with GNU C++ compiler. We used the TV-tree, the X-tree and the Pyramid-Technique programs without modifying the program sources that were implemented by the authors of the references [11, 20, 21]. The size of each block in these experiments is 4Kbytes. As a synthetic data set, we generated 2,000,000 uniformly distributed floating point numbers between 0.0 and 10.0, and then we grouped them with desired dimensions to make the data points. The dimension was varied from D=4 with 500,000 data points up to D=18 with 111,111 data points.

5.1 Insertion performance

Because the CIR-tree uses the reinsertion technique in insertion algorithm, the insertion cost of the CIR-tree is higher than those of the X-tree and the Pyramid-Technique which do not use the reinsertion technique. The Pyramid-Technique always spends less time than the CIR-tree, but as the size of the supernode grows, the X-tree tends to spend more time than the CIR-tree.

5.2 Retrieval performance

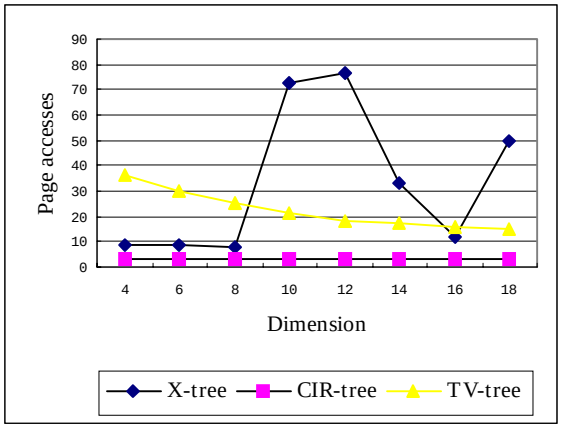


Fig. 5. The number of page accesses for an exact match query

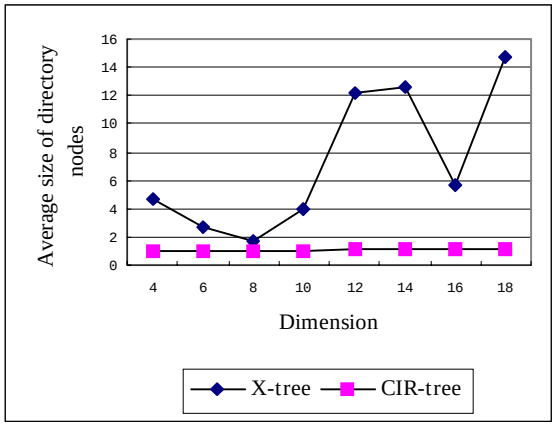


Fig. 6. An average size of directory nodes

Fig. 5 and Fig. 6 show the comparison result of three index structures for uniformly distributed data. We have applied 50,000 exact match queries for each dimension. Note that, to count the page accesses, the access to supernode of size *s* was counted as *s* page accesses. As shown in Fig. 5, the CIR-tree outperforms other index trees. Since the number of data points is decreased with increasing number of dimensions, page accesses of the TV-tree with dimension are reduced. The retrieval performance of the X-tree depends on the size of supernodes. For large supernodes, for example D=12, the increase of page accesses of the X-tree is significant. Because the CIR-tree maintains small size of directory, the number and the size of supernodes are smaller than those of the X-tree. Eventually the CIR-tree always provides better performance for each dimension.

We also performed the range queries and the nearest neighbor queries with the TV-tree, the X-tree and the CIR-tree. For the range query we first generated 5,000 center vector using random number generation, then made range two bounding rectangles; for upper bound vector we added one to each dimension of the center vector, for lower bound vector we subtract one from the center vector. We extended the TV-tree source with the functions that process the range queries and the neighbor queries. The original source of the TV-tree supports point query only. The result of the range queries is presented in Fig. 7. Similar to the result shown in Fig. 5, the page accesses of the X-tree depends on the size of supernodes. And the page accesses of the CIR-tree is kept stable.

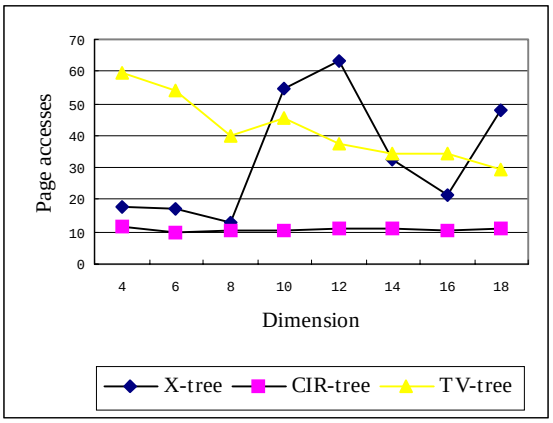


Fig. 7. Range query

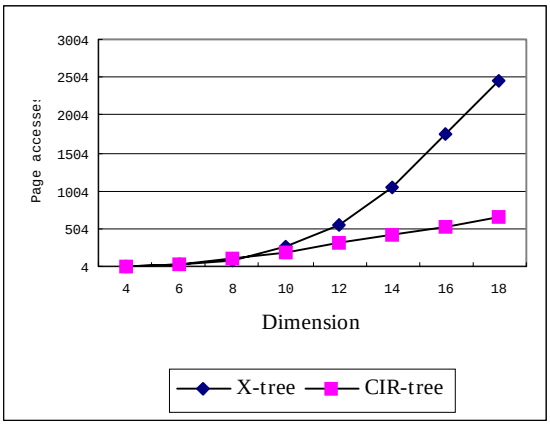
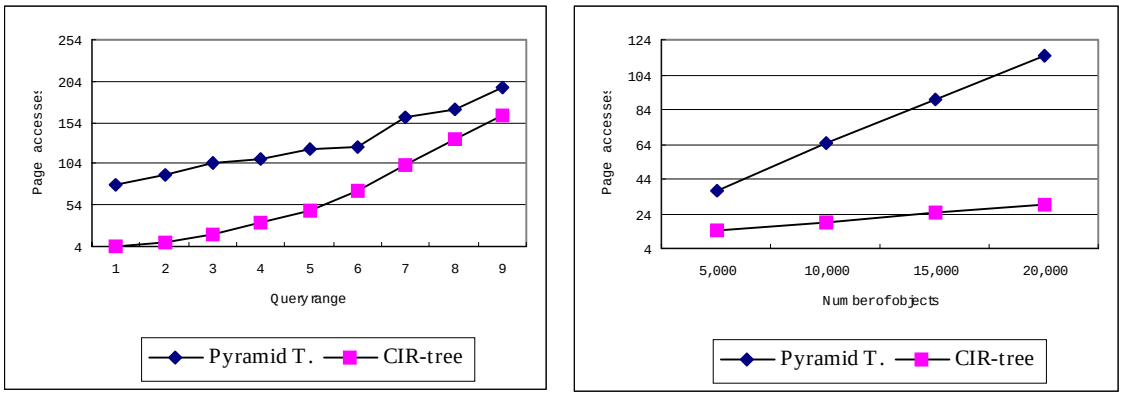


Fig. 8. 10 nearest neighbor query

Fig. 8 shows the results of 10 nearest neighbor queries. Because the result of the *k* nearest neighbor queries with the TV-tree was meaningless, we exclude the page accesses of the TV-tree in the figure. The page accesses of the X-tree is increased exponentially with dimension. But the page accesses of the CIR-tree is increased linearly with dimension.

Finally, we compared the CIR-tree with the Pyramid-Technique by using a real data set. We used the letter image recognition data in [2] as the real data set. It was 20,000 points of 17 dimensional data (1 category and 16 numeric features). The category was one of the 26 capital letters in the English alphabet and the numeric features were scaled to fit into a range of integer values from 0 to 15. Fig. 9 is the experimental results with the real data that the CIR-tree shows better performance than the Pyramid-Technique in range query and nearest neighbor query. Note that, in Fig. 9 (a), the difference of the number of the page accesses of the two methods gets smaller as the range increases. The CIR-tree seems better than the Pyramid-Technique in large ranges, but we cannot definitely say that it is superior to the Pyramid-Technique because the Pyramid-Technique has simple node structure than the CIR-tree. In other words, although

the Pyramid-Technique needs more page accesses than the CIR-tree, it may spend less CPU time than the CIR-tree because of its simple node structure.



(a) Range query (b) 10 Nearest neighbor query

Fig. 9. Comparison with Pyramid Technique by using the real data

5.3. Storage space

Fig. 10 shows an experimental result of each index structure in terms of storage space. Due to the fact that the CIR-tree and X-tree create similar numbers of leaf nodes, the comparison of the number of leaf nodes is meaningless. So we only compared the number of directory nodes. The figure shows that the space usage of the X-tree increases with the number of dimensions, but the space usage of the CIR-tree is kept stable. This is because the CIR-tree stores small number of features in the directory node for all dimensions. The CIR-tree creates small number of nodes and the tree uses the storage space effectively. As a result, the performance comparison in terms of storage space shows that the storage overhead of the CIR-tree is much less than that of the X-tree.

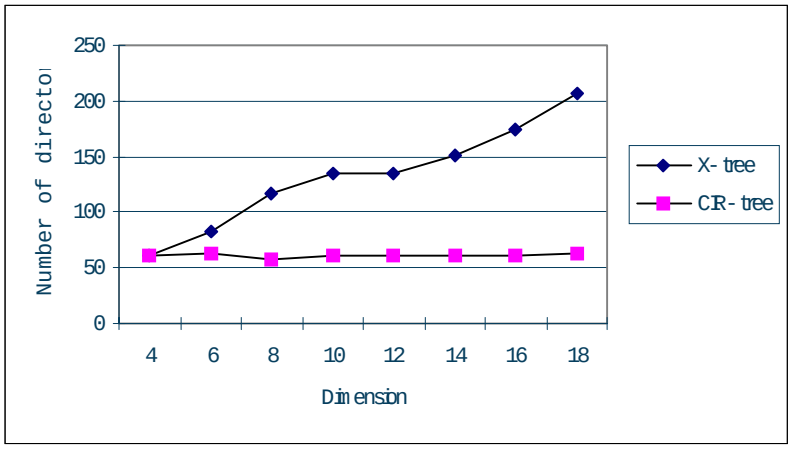


Fig. 10. Number of directory nodes depending on the dimensionality

6. Conclusion

In this paper, we have analyzed existing index structure for high dimensional data and proposed several desired design requirements that the new index structure must have. We have also proposed new efficient index structure, called CIR-tree, which followed the design requirements. Since the proposed CIR-tree used fewer dimensions at upper levels, it was able to store more data at a node. This method produced high fan out at upper levels. As a result, the height of the tree become lower, solving the dimensionality problems that we mentioned several times before. This supported high dimensional data more efficiently, and diminished disk accesses and improved the disk storage utilization. By using the weighted center, proposed in this paper, in the reinsert algorithm, the CIR-tree produced well-clustered structure with less overlap. The CIR-tree has also used weighted Euclidean distance measure to overcome the exactness problem of Euclidean distance and used supernode in order to minimize overlap.

We have compared the proposed CIR-tree with the TV-tree, the X-tree and the Pyramid-Technique through various experiments to manifest the superiority of our tree. The experiments have showed that the CIR-tree outperformed the TV-tree, the X-tree and the Pyramid-Technique in terms of retrieval speed and space requirements. But the CIR-tree needs further investigation to improve nearest neighbor query performance.

Acknowledgment

This work was supported in part by Korea Science and Engineering Foundation (KOSEF. 1999-1-303-007-3) and the Ministry of Information & Communication of Korea("Support Project of University foundation research<'99> " supervised by IIT.

References

1. B. Furht, S. W. Smoliar and H. Zhang, "Video and Image Processing in Multimedia Systems," Kluwer Academic Publishers, 1995.
2 Blake, C.L. & Merz, C.J. (1998). UCI Repository of machine learning databases, [http://www.ics.uci.edu/~mlearn/MLRepository.html]. Irvine, CA: University of California, Department of Information and Computer Science.
3. C. Faloutsos, R. Barber, M. Flickner, J. Hafner, W. Niblack, D. Petkovic and W. Equiz., "Efficient and Effective Querying by Image Content," Journal of Intelligent Information System(JIIS), 3(3):231-262, July 1994.
4. Charles E. Jacobs, Adam Finkelstein, David H. Salesin., "Fast Multiresolution Image Query." Proc. ACM SIGGRAPH, New York, 1995.
5. D. A. White and R. Jain, "Similarity Indexing with the SS-tree," Proc. 12th Int. Conf. On Data Engineering, New Orleans, pp. 516-523, 1996.
6. D. A. White and R. Jain, "Similarity Indexing: Algorithms and Performance," Proc. SPIE: Storage and Retrieval for Image and Video Databases IV, Vol. 2670, pp.62-75, 1996.
7. Gisli R. Hjaltason and Hanan samet, "Ranking in spatial Databases," Proc. 4th Symposium on Spatial Databases, Portland, Maine, Aug. 1995, pp.83-95.
8. Guttman A., "R-trees: A Dynamic Index Structure for spatial Searching," Proc. 7th Int. Conf. on Data Engineering, 1991, pp.520-527.
9. J. K. Wu, A. Desai Narasimhalu, B. M. Mehtre, C. P. Lam and Y. J. Gao, "CORE: a contentbased retrieval engine for multimedia systems.," ACM Multimedia Systems, 3:25-41, 1995.
J. T. Robinson, "The K-D-B-Tree: A Search Structure for Large Multidimensional Dynamic Indexes," ACM SIGMOD, pp. 10-18, Apr. 1981.
11. K. I. Lin, H. Jagadish and C. Faloutsos, "The TV-tree: An Index Structure for High Dimensional Data", VLDB Journal, Vol. 3, pp.517-542, 1994.

12. Lomet D., "A Review of Recent Work on Multi-attribute Access Methods," ACM SIGMOD RECORD, Vol. 21, No. 3, pp. 56-63, Sept. 1992.
13. M. J. Swain and D. H. Ballard, "Color indexing," International Journal of Computer Vision," 7(1):11-32, 1991.
14. Myron Flickner, Harpreet Sawhney, Wayne Niblack, Jonathan Ashley, Qian Huang, Byron Dom, Monika Gorkani, Jim Hafner, Denis Lee, Dragutin Petkovic, David Steele and Peter Yanker, "Query by Image and Video Content: The QBIC System," IEEE Computer, 28(9), 1995.
15. 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, pp.322-331, May 1990.
16. Norio Katayama and Shin'ichi satoh, "The SR-tree: An Index Structure for High-Dimensional Nearest Neighbor Queries," Proc. ACM SIGMOD Int. Conf. On Management of Data, Tucson, Arizona, pp. 369-380, 1997.
17. P. M. Kelly, T. M. Cannon and D. R. Hush., "Query by image example: the CANDID approach," Proc. SPIE Storage and Retrieval for Image and Video Database III, 2420:238-248, 1995.
18. Roger Weber, Hans-Jorg Schek and Stephen Blott, "A Quantitative Analysis and Performance Study for Similarity-Search Methods in High-Dimensional Spaces," Proc. 24th VLDB Conf., New York, USA, pp 194-205, 1998.
19. Roussopoulos N., Kelley S., Vincent F., "Nearest Neighbor Queries," Proc. ACM SIGMOD Int. Conf. On Management of Data, San Jose, CA, pp. 71-79, 1995.
20. S. Berchtold, D. A. Keim and H.-P. Kriegel, "The X-tree: An Index Structure for High-Dimensional Data," Proc. 22nd VLDB Conf., Bombay, India, pp. 28-39, Sep. 1996
21. S. Berchtold, Christian Bohm and Hans-Peter Kriegel, "The Pyramid-Technique: Towards Breaking the Curse of Dimensionality," Proc. ACM SIGMOD Int. Conf. on Management of Data, Seattle, WA, pp. 142-153, 1998.
22. W. E. Mackay and G. Davenport, "Virtual video editing in interactive multimedia applications," Communications of the ACM, 32:802-810, July 1989.
23. W. Niblack, R. Barber, W. Equitz, M. Flickner, E. Glasman, D. Petkovic, P. Yanker, C. Faloutsos and G. Taubin, "The QBIC project: Querying image by content using color, texture and shape." Proc. SPIE Storage and Retrieval for Image and Video Databases, pp. 173-187, Feb. 1993.
24. Y. Alp Aslandogan, Chuck Their, Clement T. Yu, Chengwen Liu and Krishnakumar R. Nair, "Design, Implementation and Evaluation of SCORE (a System for Content based Retrieval of pictures)," Proc. 11th Int. Conf. of Data Engineering, pp. 280-287, 1995.