Node Splitting Algorithms in Tree-Structured High-Dimensional Indexes for Similarity Search

Yongjian Fu
Department of Computer Science
University of Missouri-Rolla
Rolla, MO, 65409-0350
yongjian@umr.edu

Jui-Che Teng
Department of Computer Science
University of Missouri-Rolla
Rolla, MO, 65409-0350
jteng@umr.edu

S. R. Subramanya
Department of Computer Science
University of Missouri-Rolla
Rolla, MO, 65409-0350
subra@umr.edu

ABSTRACT

Most high-dimensional indexing structures proposed for similarity search in content-based retrieval systems are tree-structured. A high-dimensional tree-structured index is built by inserting feature vectors which represent multimedia objects into the index tree. If the node in which a vector is to be inserted is full, it has to be split, which eventually cause the index tree to grow. In this paper, several node splitting algorithms for tree-structured high-dimensional indexes are studied. The algorithms are classified into three categories: seed-based algorithms, sort-based algorithms, and clustering algorithms. The algorithms have been implemented and tested on an image database. Their relative performances in various situations are presented.

Keywords

High-dimensional index, node splitting, similarity search, image database, content-based retrieval

1. INTRODUCTION

A multimedia system manages many multimedia objects including digital images, graphics, videos, and audios. An interesting research problem in multimedia systems is the content-based retrieval of multimedia objects, for example the retrieval of images based on colors and shapes. As the contents of multimedia objects are usually represented by their features, an object is a vector in the feature space. Content-based retrievals are then mapped into searches in the feature space.

Similarity search refers to the search of similar objects to a query object in the feature space [15]. For example, given an image, its features are compared with those of other images in the database to find similar images in terms of color and shape. Similarity search has potential applications in medical image management, digital library, and GIS.

SAC 2002, Madrid, Spain

To facilitate similarity search, indexes may be built on the feature data. Such indexes are referred to as high-dimensional indexes because the dimensionality of the features can easily run up to tens. Several high-dimensional indexing structures for similarity search have been proposed including SS-tree [15], SS$^+$-tree [11], SR-tree [10], X-tree [2], and M-tree [5]. These structures extend previous multi-dimensional structures such as R-tree [8], R*-tree [1], and KD-tree [6], by enhancing nodes and entries with feature statistics specifically required by similarity search.

Most of the high-dimensional indexes proposed so far are tree-structured. An index tree is constructed by inserting objects represented by feature vectors into the tree. It grows when the nodes split. In this paper, we study the node splitting algorithms in high-dimensional indexes for similarity search.

The rest of the paper is organized as follows. In Section 2, related work and background in similarity search and high-dimensional indexing are described. Node splitting algorithms are discussed in Section 3. Experiments and explanations are presented in section 4. Section 5 concludes the paper.

2. SIMILARITY SEARCH AND HIGH-DIMENSIONAL INDEXING

A user of a multimedia system may search for objects by keywords and by contents. With the limitation of keyword searching, researchers have been studying contend-based retrieval for multimedia objects, especially images[7].

In order to perform content-based retrieval, the feature of objects are extracted and stored with the objects in a multimedia database. An object is then represented by its feature vector. A user may submit a query object, in which case the features of the query object are extracted, or specify the features of targeted objects. The features are compared with those of the objects in the database. Similar objects, which are determined by a similarity function, are returned as the result of the query. The similarity function is a metric defined on object features instead of objects themselves. Usually, the similarity function calculates the dissimilarity of two object, for example, the Euclidean distance between the two feature vectors of the two objects.

The queries can be classified into two classes: k nearist-neighbor (k-NN) queries and range queries. A k-NN query searches for k objects that are the most similar (closest in distance) to the query object. A range query finds all objects

which are with in a predefined similarity range to the query object. To process the queries, indexes may be built on the feature vectors and can be used in the similarity search.

One prominent characteristic of the features is their high dimensionality. Usually, many features are needed to represent an object. For example, the shape of an image may be represented by more than 10 features. The dimensionality of a feature vector can easily run up to tens, and sometimes a hundred, which poses challenges for earlier indexing structures, such as B^+-tree, R-tree [8], and KD-tree [6]. The data structures need to be space efficient and provide fast insertion and search algorithms.

Recently, many high-dimensional indexing structures have been proposed [2, 4, 5, 10, 11, 15]. Most high-dimensional indexes proposed are tree-structured [2, 5, 10, 11, 15]. Starting with an empty tree, an index is built by inserting feature vectors one by one into the tree. The indexing structures differ in their node structure, insertion algorithm, and search algorithm.

An indexing structure for similarity search, SS-tree, was introduced by White and Jain [15]. In an SS-tree, a node represents the bounding spheres of the vectors in the subtree rooted at the node, instead of the bounding rectangle of the vectors as in R-tree [8] and its variants [12, 1]. It has been found that SS-tree outperforms R-tree for similarity search in high-dimensional spaces.

One problem with SS-tree is the overlapping of the bounding spheres of the nodes, which gets worse when the dimensionality increases. Several approaches have been proposed to overcome the problem. SS^+-tree [11] extends SS-tree by introducing a clustering method, the k-means algorithm, in node splitting to reduce overlapping. SR-tree [10] combines SS-tree and R^*-tree such that an entry contains both bounding rectangle and bounding sphere of the vectors. X-tree [2] introduces supernode to avoid node splitting as much as possible. A supernode is a nonleaf node with extra entries to accommodate alienated entries in order to reduce overlapping among the nodes. M-tree [5] includes the distance between the parent node to each child in the entry. Based on the property of metric space, M-tree may improve the search speed by reducing the number of distance calculations.

The previous work has been focused on node structure. While node structure is essential for indexes, it is clear that the insertion algorithm plays a key role in the shaping of the overlapping among the nodes. In this paper, we study node splitting, an important part in the insertion algorithm, and its effect on indexes. A simple splitting strategy was used in SS-tree, SR-tree, and X-tree. Although node splitting was studied in SS^+-tree and M-tree, they did not consider many possible algorithms. To the best of our knowledge, there is not a comprehensive study on the effect of node splitting in high-dimensional indexing.

To illustrate our ideas, we will use SS-tree as the index structure in this paper. However, the node-splitting algorithms that we discuss can be applied to any tree-structured indexes. The details of the SS-tree are explained as follows.

In an SS-tree, each node consists of a set of entries and some information about itself including the number of children it has and its depth (height). In a nonleaf node, each entry refers to one of the node's children. The entry stores the total number, the centroid and radius of the bounding sphere, and the variance, of the vectors in the subtree rooted at the child, and a reference to the child. Each entry in a leaf node stores a single object, in which the centroid is the feature vector of the object and the reference is to the object.

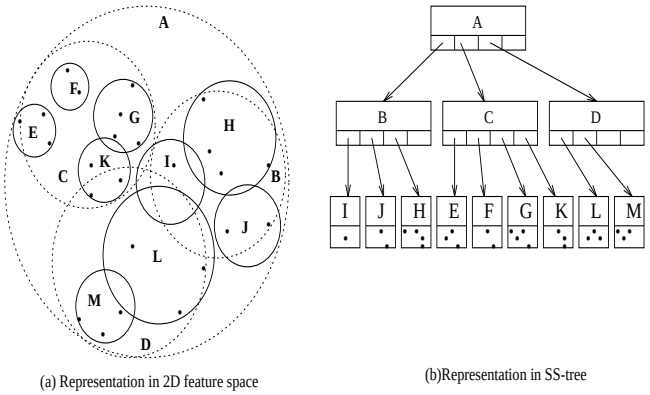

Figure 1: A simple SS-tree

A simple SS-tree for two-dimensional feature vectors is illustrated in Fig. 1. Fig. 1(a) shows the vectors in feature space, and Fig. 1(b) shows the data structures in the SS-tree to represent them.

An SS-tree is initialized with the root node and an empty child which is a leaf. When a vector is inserted into an SS-tree, it descends down from the root to a leaf node choosing the closest child along the path according to a (dis)similarity metric. If there is an empty entry in that leaf, the vector is put there and the information is updated along the path up to the root. Otherwise, the overflowing node is split and a new node is created. Some entries are moved to the new node whose representing entry will be re-inserted in a similar procedure.

The splitting strategy guides the distribution of entries among the two nodes, the overflowing node and the new node. In this paper, a set of splitting algorithms have been studies and tested on an image database. The algorithms are discussed in Section 3 and their performance is presented in Section 4.

There have been some studies on the usefulness of similarity search and high-dimensional indexes [3, 14, 13]. Weber et. al. [13] showed that the linear scan of the file containing features is more efficient than using any index when the dimensionality is high, but they assume the dimensions are independent and the data are evenly distributed. Bayer et. al. relaxed these conditions and showed that objects become so close in a high-dimensional space that the differences in distance become statistically insignificant. They found this effect could happen for a dimensionality of 10-15. However, as Weber and Zezula pointed out [14], the real data may not satisfy even the relaxed conditions. Besides, the indexes may benefit from parallalization and are helpful in approximate search.

3. NODE SPLITTING ALGORITHMS

As explained earlier, an SS-tree is built by inserting vectors one by one into an initially empty SS-tree. A vector is inserted into a leaf node which is decided by descending from the root and selecting the closest child node along the path. If the leaf node is full, it is split. A new leaf node is created. The vectors in the original leaf node and the vector being inserted are distributed among the two nodes.

The insertion algorithm for SS-tree is outlined below.

```
(1)   insert(r: node, p: entry)
          // insert p into a tree rooted at r
(2)       n = r        //start at the root
(3)       while n is not a leaf node or
              at the same level as p        // descending
(4)           determine a child of n, c, such that
                  dist(c, p) = min(dist(c_i, p)), c_i is a child of n
(5)           n ← c
(6)       if n has an empty entry
(7)           add p into n
(8)           modify the information in its ancestors
(9)       else
(10)          n' = split(n, p)        // n' is the new node
(11)          modify the information in n's ancestors
(12)          create an entry e to represent n'
(13)          insert(r, e)        //re-insert the new node
```

The algorithm takes an entry as an input parameter so that it works for both leaf entries and nonleaf entries. A leaf entry consists of a single vector. A nonleaf entry corresponds to a node, whose information is summarized in the entry, including the bounding sphere of the vectors in the tree rooted at the node. The distance between two entries e_1 and e_2, dist(e_1, e_2), is the distance between the centroids of the entries. Procedure split(n, p) takes a node n and an entry p, and returns a new node n'. The entries in n and the entry p are distributed between n and n', according to a splitting algorithm. The splitting algorithm partitions the entries into two groups, which are put into n and n', respectively.

There are several issues involved in node splitting as discussed below. Different decisions on these issues give different algorithms.

- Complexity

  There are 2^{B+1} possible partitions for $B+1$ entries. If a splitting algorithm considers all possible partitions, its complexity will be exponential.

- Objective

  The objective specifies what a splitting algorithm should try to achieve. Three objectives are investigated in our study: to minimize total radii, to minimize overlay, and to minimize square error. The objective also implies the criterion for comparing partitions.

- Balance

  In a balanced partition, each group has at least certain number of entries. The number is specified by the user in terms of percentage of total entries. A balanced partition is desirable because it keep the tree more balanced.

In this study, only algorithms with a polynomial complexity are examined. To make the partition balanced, a user specified threshold called *balance threshold*, T, is introduced, which specifies the percentage of entries each group should have. To avoid extremely unbalanced partition, each group is set to have at least $L = max(T * (B+1), 2)$ entries. This guarantees that a group has at least two entries, even when $T = 0\%$.

Eight splitting algorithms are examined in this study. The algorithms can be broadly classified into *seed-based algorithms*, *sort-based algorithms*, and *clustering algorithms*.

- In a seed-based algorithm, two entries are selected as representatives or seeds of the groups. Other entries are distributed based on their distances to the seeds. An entry is put into the same group as the closer seed. Obviously, the selection of the seeds is essential to the algorithm. Four selection methods are examined: random selection, two furtherest entries, two entries which minimize total radius, and two entries which minimize overlapping. They are explained later.

- In a sort-based algorithm, the entries are sorted on a dimension and divided into two groups according to a cutpoint on the dimension. The dimension with the maximum variance is selected as the sorting dimension. Two algorithms are developed which select cutpoints to minimize either total radius or overlapping.

- The well-known k-means clustering method is employed to find two group of entries. The k-means method tries to minimize square error. The basic k-means and a variant, the sequential k-means, are studied.

For simplicity, we assume the input of the split algorithm is a set of entries, instead of a node and an entry. The conversions between a node and a set of entries are straightforward. The input set consists of $B+1$ entries, B entries in the original node and the inserting entry, where B is the maximal number of entries in a node, called *branching factor*. The input set is split into two output sets as a result of the split algorithm. The output set that is closer to the parent node stays in the original node and the other is put into a new node which is then returned by the split algorithm.

3.1 Seed-Based Algorithms

In seed-based algorithms, two entries are selected as seeds and other entries are put into the same group as that of the closer seed. The procedure is outlined below.

Several seed selection criteria are possible, which leads to different algorithms. The followings are examined in our study.

1. Random selection (algorithm RANDOM)

   Two entries are randomly selected as seeds. After the distribution, assume the two groups, N_1 and N_2, have n_1 and n_2 entries ($n_1 < n_2$). If $n_1 < L$, move $L - n_1$ entries in N_2, which are closest to N_1, to N_1. This algorithm is the simplest among all the algorithms with a complexity of $O(B)$.

2. Two furthest apart entries (algorithm FURTHEST)

   The pair of entries who distance is the largest among all pairs are selected as seeds. The partition is balanced as in algorithm RANDOM. Since there are total $\frac{1}{2}B(B+1)$ pairs to be considered, the complexity of the algorithm is $O(B^2)$.

3. Two entries which minimize the total radii (algorithm MIN-R)

   In this algorithm, we consider all pairs of entries. For each pair, other entries are distributed with the pair

as the seeds. If the resulting two groups both have at least L entries, the radii of the resulting two groups are calculated. The pair with the smallest sum of radii is selected as the seeds. The complexity of the algorithm is $O(B^3)$ since for each pair, $B-1$ entries are distributed, and there are $\frac{1}{2}B(B+1)$ pairs to be considered,

4. Two entries which minimize overlapping (algorithm MIN-O)

   Like the previous algorithm, we consider all pairs of entries in this algorithm. For each pair, other entries are distributed with the pair as the seeds, and discarding the pair if a resulting group has less than L entries. Unlike the previous algorithm, the overlapping between the two resulting groups are calculated and the pair with the smallest overlap is selected as the seeds. The overlapping between two groups is estimated as the smallest distance of a point on the boundaries to the line between two centroids. This algorithm has a complexity of $O(B^3)$, same as the previous algorithm.

3.2 Sort-based Algorithms

Another kind of split algorithms sorts the entries according to their values on a dimension. The first t entries are put into a group and the rest is put into another. The cut point, t, is selected such that total radii or overlapping is minimized.

The algorithms are outlined as follows.

1. Decide the dimension with the maximum deviation, max_dev_dim
2. Sort the entries according to max_dev_dim
3. Determine a cut point c, $L \leq c \leq B+1-L$, such that the total radii of the two groups is minimized (algorithm SORT-MR) or the overlapping is minimized (algorithm SORT-MO). For each cut point c, the first c entries are put into the first group, and the rest into another.
4. Put the first c entries into a group and the rest into another.

Both algorithms have a complexity of $O(B\log(B))$ because of the sorting. A different sort-based algorithms is used in SS-tree, in which the cutpoint is selected to minimize the sum of variances in max_dev_dim.

3.3 Clustering Algorithms

The split can be viewed as the clustering of entries into two clusters, one in the original node and another in the new node. A suitable clustering method is the well-known k-means clustering method [9]. Given a set of data points, k-means groups them into k clusters such that the square error is minimized. The number of clusters, k, is specified by the user.

To split a node by k-means, k is set to 2. During clustering, the entries are assigned to one of the clusters whose centroid (mean) is closer to the entry. The centroids of the clusters are adjusted either after each iteration or when an entry changes membership. The first choice leads to the basic k-means method, while the second leads to the sequential k-means method.

The basic k-means method and the sequential k-means are outline as follows.

1. Basic k-means (algorithm K-MEANS)
   1. Randomly assign each entry to a cluster
   2. Compute the centroids of the clusters
   3. Assign each entry to the closer cluster
   4. Repeat step 2 and 3 until no entry changes membership or the number of iteration has reached a predefined threshold
   5. After the clustering, assume the two groups, N_1 and N_2, have n_1 and n_2 entries ($n_1 < n_2$)
   6. If $n_1 < L$
   7. select $L-n_1$ entries in N_2 which are closest entries to N_1
   8. move them to N_1
2. Sequential k-means (algorithm SK-MEANS)
   1. Randomly assign each entry to a cluster
   2. Compute the centroids of the clusters
   3. For each entry, if its current cluster is not closer, move the entry to the other cluster and recompute the centroids of the clusters
   4. Repeat step 2 and 3 until no entry changes membership or the number of iteration has reached a predefined threshold
   5. After the clustering, assume the two groups, N_1 and N_2, have n_1 and n_2 entries ($n_1 < n_2$)
   6. If $n_1 < L$
   7. select $L-n_1$ entries in N_2 which are closest entries to N_1
   8. move them to N_1

The complexity of k-means is $O(nkT)$ where n, k, T are the number of entries, the number of clusters, and the maximum number of iterations, respectively. Since $k=2$ in our case, the complexity of both algorithm is $O(BT)$.

4. PERFORMANCE STUDY

The splitting algorithms have been implemented and tested on an image database consisting of 800 images each containing a single object. Extensive experiments have been conducted to study the tree quality and running time of each algorithm. The tests are done on a Sun Ultra1 workstation with 64MB memory running Solaris 2.5. The image features used in the experiments are discussed in the next subsection, followed by results of the experiments.

The image features used in the experiments include 7 in moment invariants, 11 in geometry, and 20 in Fourier descriptors. Currently in our system, the features are computed off-line with a Matlab program and stored in a file. All the features are normalized and grouped together into a single vector.

These algorithms are compared in terms of total insertion time and resulting tree quality. The total insertion time is

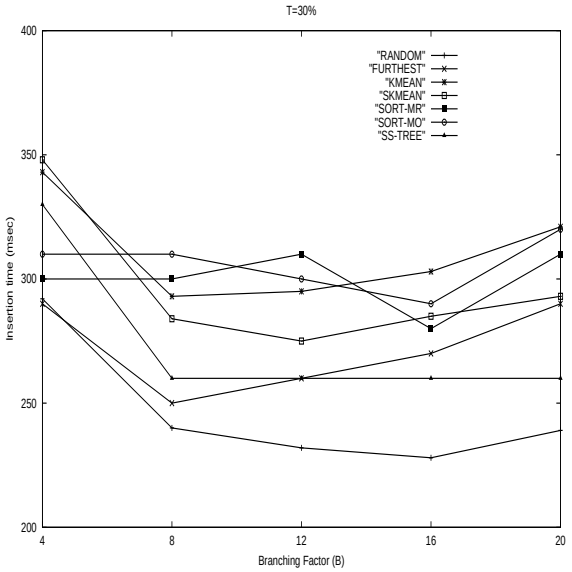

Figure 2: Insertion times for various B.

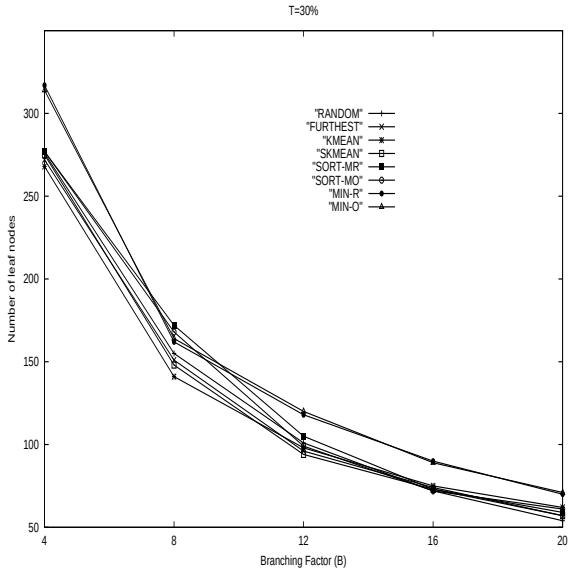

Figure 3: Number of leaf nodes for various B.

measured since node splitting is part of the insertion algorithm, whose speed is our concern. Besides, it is hard to measure only the time spent on splitting. To compare the quality of resulting trees, the trees are compared in terms of the number of leaf nodes and the average radius of leaf nodes. In addition, we search the trees for the nearest 20 neighbors of every image. The average number of nodes accessed during search is calculated for all images.

Two parameters which affect the performance of the algorithms, are the branching factor B and the balance threshold T. In the experiments, the values of B are from 4 to 20 and the values of T are from 0% to 40%. Different combinations of the two parameters are tested to examine their effects on the four performance measures: the insertion time, the number of leaf nodes, the average radius of leaf nodes, and the average number of accessed nodes. The iteration threshold I in the two k-means methods is set to 20.

The insertion time for the algorithms is shown in Figure 2. The balance threshold T is set to 30%. Other values of T give similar results. The times for algorithms MIN-O and MIN-R are much longer, from almost double ($B = 4$) to an order of magnitude ($B = 20$) comparing to others, so they are not shown in the figure. Generally, the algorithms do not make much difference on insert time. This is because node splitting is only a small part of the insertion algorithm. It is interesting to notice that when B is 4, the insertion is the slowest. This is because a small B causes more splitting, although splitting is faster.

Figure 3 shows the number of leaf nodes of the resulting trees. The number of leaf nodes basically decides the total number of nodes in a tree. The balance threshold T is 30%. The tree sizes, i.e., the total number of nodes in the trees, are comparable (usually within 10% difference) for all values of B. The two seed-based algorithms, MIN-O and MIN-R, tends to create larger trees than other algorithm.

The number of leaf nodes in resulting trees for various values of T is shown in Figure 4. The branching factor B is 20. When T is relatively large ($\geq$ 20%), the tree sizes, i.e., the total number of nodes in the trees, resulted from the algorithms are comparable (within 15% difference). When T is small ($\leq$ 10%), the splitting algorithm can make a significant difference in the size of resulting tree, in which the differences are from 20% to 40%. In such cases, the clustering algorithms BK-MEANS and SK-MEANS usually outperform others by creating smaller trees, with MIN-O and MIN-R to be worst.

However, a smaller tree generally causes a larger average radius of leaf nodes. This is because more entries are put into a node. This means the tree size should not be the only measure when comparing tree quality. The average radius of leaf nodes is examined as well.

Figure 5 shows the average radius of leaf nodes of the resulting trees. The average radius of leaf nodes for MIN-R and MIN-O is much bigger (> 1.0) than that of others, thus is not shown in the figure. The balance threshold T is 30%. When B is small (4 or 8), the difference in the average radius of leaf nodes is negligible. When B is large ($\geq$ 12), the two clustering algorithms, BK-MEANS and SK-MEANS, tends to have a tree with smaller average radius than other algorithms.

The average radius of leaf nodes in resulting trees for various values of T are shown in Figure 6. The branching factor B is 20. Except MIN-R and MIN-O, other algorithms perform more or less the same for different values of T. It is a little surprising to see that MIN-O and MIN-R create larger trees with larger average radius of leaf nodes. This reveals that the extensive search for best seeds in MIN-O and MIN-R is futile.

As an empirical comparison of the search performance of the trees, the average number of accessed nodes during a search is measured. Figure 7 shows the average number of nodes accessed in a search as a percentage of the total number of nodes in the resulting trees for various values of B. The balance threshold T is 30%. The best algorithms is obviously SK-MEANS which visits the fewest nodes for almost all values of B except when B is 20. Two algorithms,

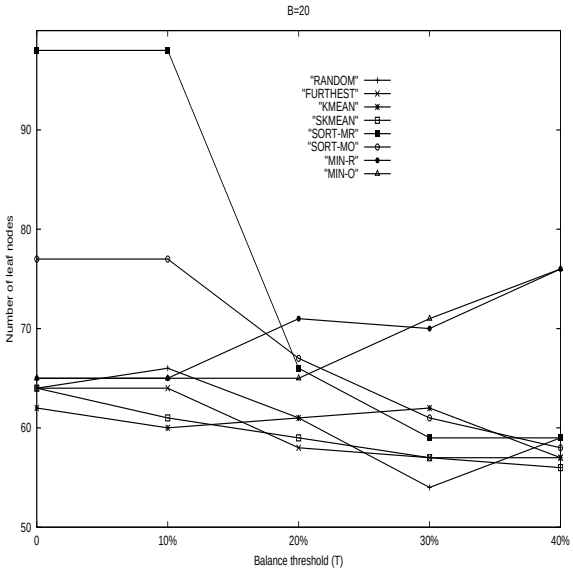

Figure 4: Number of leaf nodes for various T.

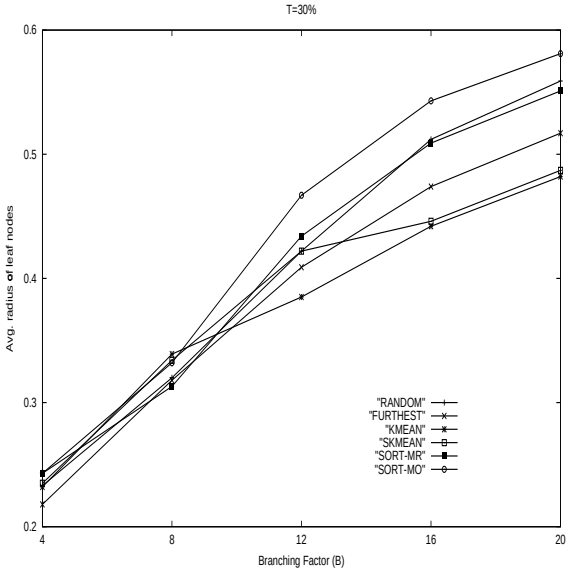

Figure 5: Average radius of leaf nodes for various B.

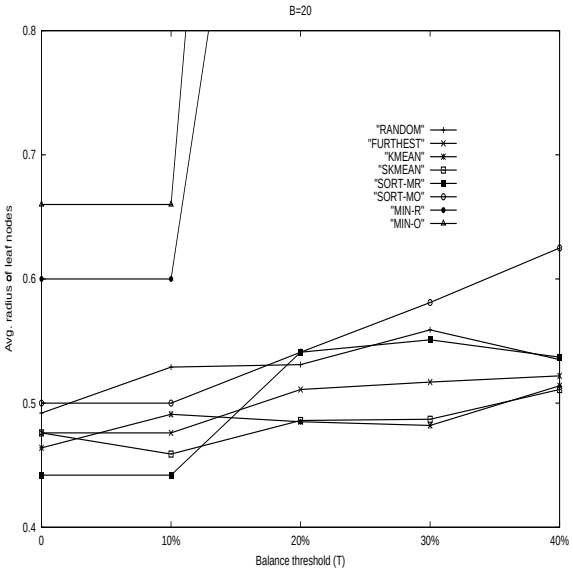

Figure 6: Average radius of leaf nodes for various T.

MIN-O and MIN-R, do not perform as well as others when B is 4.

Figure 8 shows the average number of nodes accessed in a search as a percentage of the total number of nodes in the resulting trees for various values of T. The branching factor B is 20. It seems that the relative performance of the algorithms is drastically affected by T. One stable performer is FURTHEST, which is the best or close to the best most of the time.

In summary, simple algorithms such as RANDOM, SORT-MR, and SORT-MO, produce satisfactory results. The clustering algorithms, BK-MEANS and SK-MEANS, produce the most stable results. In most cases, they are the best performer or very close to the best. In general, the algorithms generate better trees with a balanced partition. A suitable balance threshold T is from 20% to 30%.

It should also be pointed out that in almost all cases, three or more algorithms perform better than or as well as the SS-tree algorithm. In some cases, the improvement is quite impressive.

5. CONCLUSION

Node splitting in tree-structured high-dimensional indexes is studied. Eight node splitting algorithms are proposed and examined. The algorithms are implemented and tested in an image database. The experiments show that they can affect the quality of resulting trees, sometimes significantly. Simple algorithms are found to work well while clustering algorithms are the most robust.

We are conducting more experiments to verify the observations in the paper. For the future, we plan to test our algorithms on other tree-structured high-dimensional indexing structures besides SS-tree.

6. REFERENCES

[1] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *Proc. 1990*

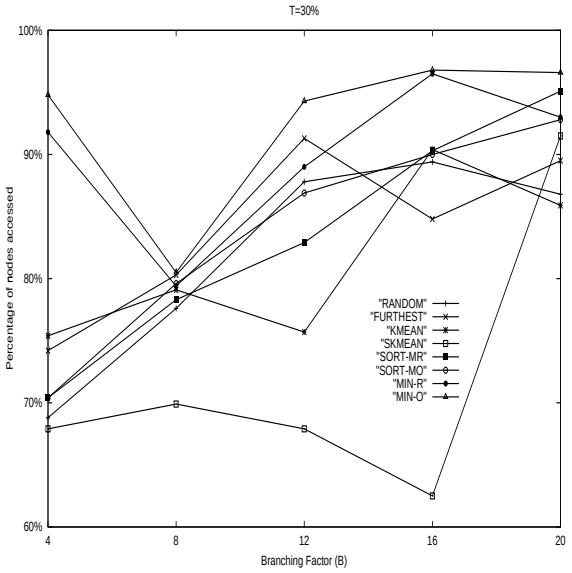

Figure 7: Percentage of nodes visited for various B.

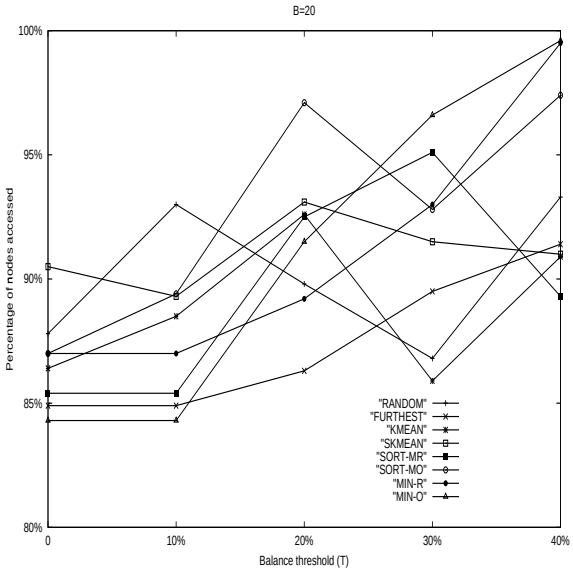

Figure 8: Percentage of nodes visited for various T.

ACM-SIGMOD Int. Conf. Management of Data, pages 322–331, Atlantic City, NJ, June 1990.

[2] S. Berchtold, D. Keim, and H. Kriegel. The X-tree: an index structure for high-dimensional data. In *Proc. 22nd VLDB*, pages 28–39, Mumbai, India,, 1996.

[3] K. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft. When is "nearest neighbour" meanful? In *Proc. ICDT*, pages 217–235, 1999.

[4] T. Bozkaya and M. Ozsoyoglu. Distance-based indexing for high-dimensional metric spaces. In *Proc. 1997 ACM-SIGMOD Int. Conf. Management of Data*, pages 357–368, Tucson, Arizona, May 1997.

[5] P. Ciaccia, M. Patella, and P. Zezula. M-tree: an efficient access method for similarity search in metric spaces. In *Proc. 23rd VLDB*, pages 426–435, Athens, Greece, 1997.

[6] J. H. Friedman, J. H. Bentley, and R. A. Finkel. An algorithm for finding best matches in logarithmic expected time. In *ACM Trans. on Mathematical Software*, pages 209–226, Vol. 3, Sept. 1977.

[7] V. Gudivada and V. Raghavan. Content-based image retrieval systems. In *IEEE Computer*, pages 18–22, Vol 28, No. 9, 1995.

[8] A. Guttman. R-tree: A dynamic index structure for spatial searching. In *Proc. 1984 ACM-SIGMOD Int. Conf. Management of Data*, June 1984.

[9] A. K. Jain and R. C. Dubes. *Algorithms for Clustering Data*. Printice Hall, 1988.

[10] N. Katayama and S. Satoh. The SR-tree: an indexing structure for high-dimensional nearest neighbor queries. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 369–380, Tucson, Arizona, 1997.

[11] R. Kurniawati, J. Jin, and J. Shepherd. The SS$^+$-tree: an improved indexing structure for similarity searches in a high-dimensional feature space. In *Proc.*, pages 110–120, SPIE Vol. 3022, 1997.

[12] T. Sellis, N. Roussopoulos, and C. Faloutsos. The R+-Tree: A dynamic index for multi-dimensional objects. In *Proc. 13th Int. Conf. Very Large Data Bases*, pages 3–11, Brighton, England, 1987.

[13] R. Weber, H. J. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *Proc. 24th VLDB*, pages 194–205, New York, USA, 1998.

[14] R. Weber and P. Zezula. Is similarity search useful for high-dimensional spaces? In *DEXA Workshop*, pages 146–147, Florence, Italy, 1999.

[15] D. White and R. Jain. Similarity indexing with the SS-tree. In *Proc. 12th IEEE Int. Conf. on Data Engineering*, pages 516–523, New Orleans, Louisiana, 1996.