Improving High-Dimensional Indexing with Heuristics for Content-Based Image Retrieval

Yongjian Fu and Jui-Che Teng

Department of Computer Science
University of Missouri-Rolla
{yongjian,jteng}@umr.edu

Abstract. Most high-dimensional indexing structures proposed for similarity query in content-based image retrieval (CBIR) systems are tree-structured. The quality of a high-dimensional tree-structured index is mainly determined by its insertion algorithm. Our approach focuses on an important phase in insertion, that is, the tree descending phase, when the tree is explored to find a host node to accommodate the vector to be inserted. We propose to integrate a heuristic algorithm in tree descending in order to find a better host node and thus improve the quality of the resulting index. A heuristic criteria for child selection has been developed, which takes into account both the similarity-based distance and the radius-increasing of the potential host node. Our approach has been implemented and tested on an image database. Our experiments show that the proposed approach can improve the quality of high-dimensional indices without much run-time overhead.

1 Introduction

Visual data management has attracted increasing attention as more and more digital images, videos, and graphics, have become available on-line. One focus in the visual data management is the content-based image retrieval (CBIR), that is, the access of images based on their visual features such as color, shape, texture, spatial relationship, and so on [5]. The CBIR technology has wide applications in medical image management, multimedia systems, digital library, GIS, and many other systems.

In order to perform CBIR, images are processed and their visual features are extracted and stored with the images in an image database. Therefore an image can be represented as a feature vector in a feature space. A user then may give a query image and look for similar images. The features of the query image are extracted and are compared with those of other images in the database. Similar images which are decided based on some similarity metrics, such as the distance between the feature vectors of the images, are returned as the result of the query.

Such kind of queries are termed as similarity queries [12]. Unlike traditional database queries where an exact match is sought, similarity queries search for similar images to a query and the search is called similarity search. Like in traditional database systems, indexing structures may be built on the data to

facilitate similarity queries. An index will group images in the database with similar features to speed up similarity search.

Several high-dimensional indexing structures for similarity query have been proposed including SS-tree [12], SS$^+$-tree [10], SR-tree [8], X-tree [2], and M-tree [3]. These structures extend previous multidimensional structures such as R-tree [6], R*-tree [1], and KD-tree [4], by enhancing nodes and entries with feature statistics specifically required by similarity search.

The biggest problem with these structures is that there is overlapping among the nodes which would affect search efficiency and accuracy as well as storage cost. Different approaches have been attempted to reduce the overlapping, but most of them focus on data structure and node splitting strategy.

In this paper, the problem of overlapping is examined and a different approach is proposed to solve the problem. We focus on the insertion algorithm of the indexing structures because that is where the overlapping is introduced. We propose to integrate a heuristic algorithm, beam search, in the insertion algorithm in order to improve the quality of the indexing and thus improve similarity search performance.

The approach has been implemented and tested on an image database. Our experiments show that the integration of beam search in insertion can improve the quality of indexing and query without incurring much overhead.

The rest of the paper is organized as follows. In Section 2, related work and background in similarity query and similarity search are described. Our approach is introduced in Section 3. Experiments and explanations are presented in section 4. Related issues are discussed in Section 5 which also gives some directions for future work. Section 6 concludes the paper.

2 Similarity Query and Similarity Search

The modules involved in CBIR are illustrated in Fig. 1. When a user sends a query through the user interface, the query processing module will search for images satisfying the query condition and return them through the user interface. The query processing may need help from an index built on image features which are extracted and stored with the images in the image database.

There are three modes in which users may interact with a CBIR system: browsing, query-by-example, and query-by-attributes.

- In browsing mode, the user scans through the images sequentially to find interesting images.
- In query-by-example mode, the user gives an image and asks for images similar to it. The given image is called the query image which can be an image from the database or a sketch the user draws.
- In query-by-attributes mode, the user specifies the visual attributes, such as color distribution, of the images interested.

Of course, a user can switch mode in search for the target images. For example, one may first browse through the images to find an image that is close to what one is looking for and use that as a query image.

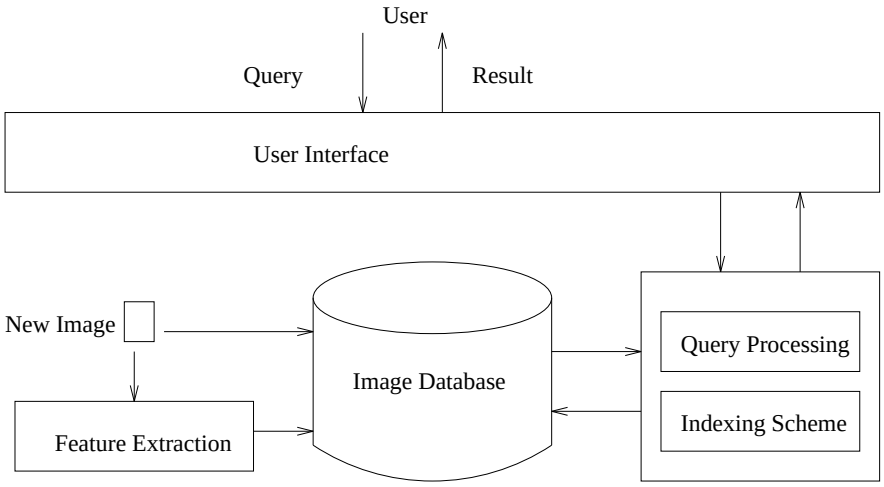


Fig. 1. Modules in content-based image retrieval

In query-by-example mode, the features of the query image are extracted using image processing techniques and submitted to the query processor as a vector. In query-by-attributes mode, a vector is given by the user. Therefore, to the query processor, these two modes are identical. In this paper, we assume the feature vector is already obtained. The browsing mode does not need indexing and thus is not discussed in this paper.

For the purpose of indexing and query processing in CBIR, images are often represented by their visual features. Each image is represented as a vector in the multidimensional vector space where each dimension represents a feature. Indexing structures are built on these vectors to facilitate the query processing. The design of indexing structures in CBIR systems is greatly influenced by the following two characteristics.

One prominent characteristic of image features is their high dimensionality. The dimensionality of feature vectors can easily run up to tens, and sometimes hundreds, which poses challenges for earlier indexing structures, such as B^+-tree, R-tree [6], and KD-tree [4]. The data structures need to be space efficient and provide fast insertion and search algorithms.

Another characteristic of CBIR systems is the fuzzy nature of queries. Usually, users cannot give a precise description of what they are looking for, or they have only a rough idea of the images they are looking for, but do not know exactly. In other words, they are interested in approximations instead of exact matches.

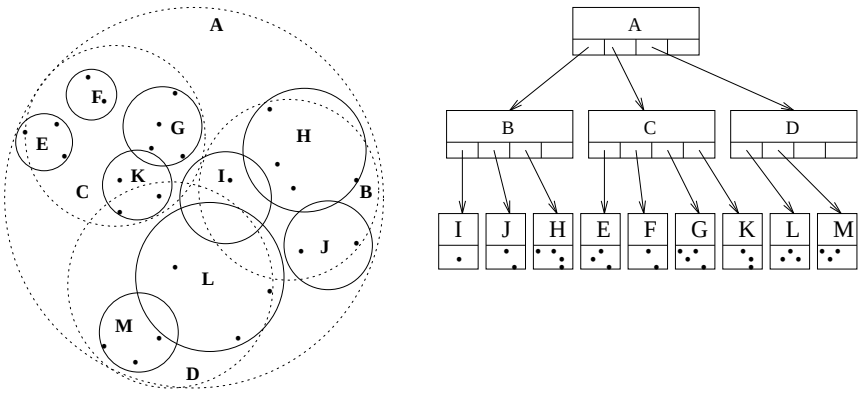
Most high-dimensional indexing structures for CBIR are tree-structured. Starting with an empty tree, an index is built on the vectors by inserting vectors one by one. When a user gives a query image, its features are extracted and searched with the help of the index. Images that are similar to the query

image based on some (dis)similarity metrics on the vectors are returned to the user as results. 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 [12]. 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 image, in which the centroid is the feature vector of the image and the reference is to the image. In their implementation, a node is represented as an array of entries which is represented by a structure called `SSElem` [12].

```
struct SSElem {
   SSElemPtr child_array_ptr;// pointer to child
   int immed_children;       // number of children in array
   int total_children;       // number of children in subtree
   int height;               // height above leaf
   int update_count;         // counter for update
   float radius;             // radius of bounding sphere
   float variance;           // sum of squared dist. from centroid
   float centroid[DIM];      // vector or mean
   char data[DATA_SIZE];     // closest immediate child
   };
```

A simple SS-tree for two-dimensional feature vectors is illustrated in Fig. 2. Fig. 2(a) shows the vectors in feature space, and Fig. 2(b) shows the data structures in the SS-tree to represent them.



(a) Representation in 2D feature space

(b)Representation in SS-tree

Fig. 2. A simple SS-tree

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.

One problem with SS-tree is the overlapping of the bounding spheres of nodes, which gets worse when the dimensionality increases. Several approaches have been proposed to overcome the problem. SS$^+$-tree [10] extends SS-tree by introducing a clustering method, the k-means algorithm, in node splitting to reduce overlapping. SR-tree [8] 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 [3] 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.

In this paper, we take a different approach to reduce the overlapping. If we look at the insertion algorithm of SS-tree, it consists of two phases: tree descending and node splitting. In tree descending, we traverse from the root to a leaf node, called *host*. Node splitting takes place when an entry needs to be added to a full node, i.e, a node whose entries are all used. Previous algorithms such as SS$^+$-tree [10] and M-tree [3] focus on splitting strategies. We look at the tree descending in the insertion algorithm in this paper.

It is obvious that SS-tree follows a single path in tree descending, i.e, starting from the root, the best child of the current node is followed. The best child is the one with the smallest weighted Euclidean distance between the new vector and the centroid of the child. This is illustrated in Fig. 3 where the dotted line denotes the path SS-tree follows.

However, this simple method may miss the best host. This is illustrated in Fig. 4(a), where A and B are nonleaf nodes of the root, $A1, A2, A3, A4$ are leaf nodes and children of A, and $B1, B2$ are leaf nodes and children of B. The new vector, p, will choose node B at the higher level and $B1$ as the host, even though $A3$ is the closest leaf node to p and thus a better host.

A heuristic algorithm, beam search, is introduced in tree descending to search for the best host and thus to reduce the overlapping. Instead of choosing a single child for descending, m children are followed. According to the similarity metric, the best m child nodes of the root are explored. The best m children of these nodes are then selected and the process continues until the leaf nodes are reached.

Moreover, we examine another factor that affects the tree descending, i.e., the criterion for choosing children to explore. It is not enough to use the distance between the new vector and the child as the only clue. An example is given in Fig. 4(b), where the new vector, p, is closer to $A1$, but would better be inserted

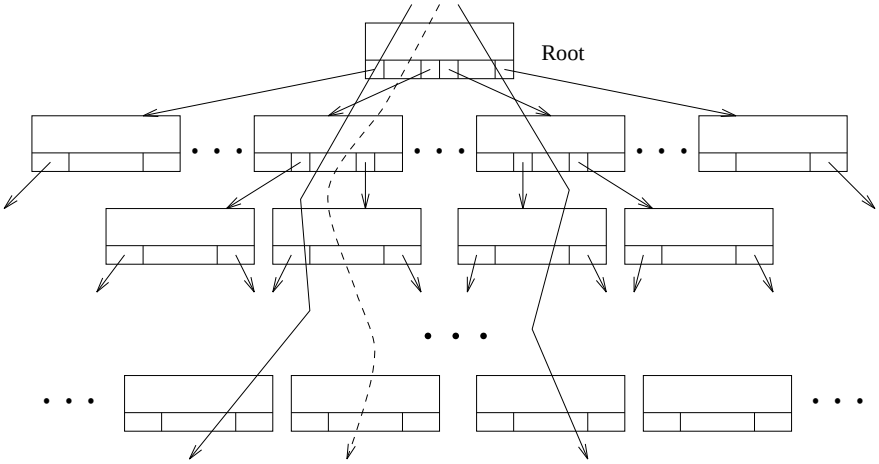


Fig. 3. Tree descending

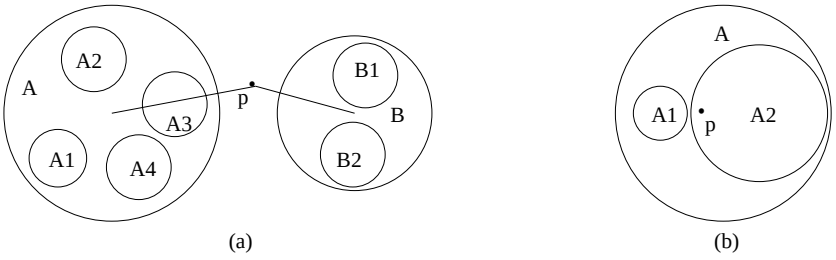


Fig. 4. Insertion of new vector

into $A2$ which causes less overlapping. We propose a weighted average of two measures as the child selection criterion in tree descending which subsumes the simple criteria used in previous algorithms. Our approach is explained in detail in the next section.

3 Building High Quality Indices

It is clear that the insertion algorithm plays a key role in the shaping of the overlapping among the nodes. In order to reduce overlapping, several approaches have looked into the node splitting strategies in insertion [10, 3]. As pointed out earlier, another important phase in insertion is tree descending. The tree descending process can be viewed as the search for the best host of the vector in the indexing tree.

A lot of tree searching algorithms are available, including depth-first, breadth-first, best-first, A*, iterative deepening, as summarized in [14]. They all find the best host eventually. However, in the worst case, these algorithms will search the entire tree, which is prohibitive when the tree size is large.

On the other extreme, SS-tree and many others follow a single path in descending whose time complexity is linear to the height of the tree. But they may easily miss the best host as discussed earlier.

In our approach, the beam search algorithm is adapted as a compromise between time complexity and indexing quality. Beam search algorithm can be briefly explained as follows. Detailed algorithm can be found in [14]. A priority queue which can hold m elements is initialized with only the initial state, where m is given by users. All of the elements in the queue are popped and states immediately reachable from these elements are pushed into the queue. The procedure continues until the queue is empty or the goal is reached.

The beam search algorithm is integrated into the insertion algorithm. At each level, we explore the best m nodes based on a selection criterion. Initially, m children of the root are selected. The best m children of the root's children are then explored. The process stops when we reach the leaf nodes. Obviously, this gives us a greater chance to find a better host than in SS-tree, while still keeping the time complexity linear to the height of the tree. The beam search based algorithm is illustrated in Fig. 3 where the solid lines denote the paths explored by the algorithm when m is two.

The selection criterion is a weighted average of two measures. For a vector p, the cost function for inserting p into a node n , $f(n,p)$, is defined as:

$$f(n,p) = w_1 * d(n,p) + w_2 * \Delta r(n,p) \tag{1}$$

where w_1, w_2 are adjustable weights, $d(n,p)$ is the weighted Euclidean distance from n to p, and $\Delta r(n,p)$ is the increase of radius of n if p is inserted. The intuitive idea is that a host should be as close to p and increase its radius as small as possible. Therefore, the smaller $f(n,p)$ is, the better it is to insert p into n.

This criterion generalizes the ones used in SS-tree and M-tree. If w_1 and w_2 are set to 1 and 0, respectively, it is the same as in SS-tree. M-tree uses a criterion in which w_1 is 1 and w_2 is ∞. The use of the criterion gives us more flexibility in picking hosts and thus tuning the algorithm. When $w_1 > w_2$, it will generate tighter nodes. On the other hand, a larger w_2 will result in less overlapping of nodes.

Obviously, SS-tree can be viewed as a special case of our approach where m, w_1, and w_2 are set to 1, 1, and 0, respectively. While other methods extend SS-tree in splitting strategy, we generalize it in tree descending. Moreover, our method can be applied to other high-dimensional tree-structured indices as well.

The node and entry structures implemented in our experiments are shown in Fig. 5. Unlike the implementation in [12], we clearly distinguish node structure and entry structure. This adds flexibility and readability in coding. Moreover, the separation reduces the storage cost because the information of a node, including height and number of immediate children, can be stored just once in the node, rather than repeated in all of its entries. Besides, a pointer to parent node is added which is useful for updating information in ancestors when a vector is inserted.

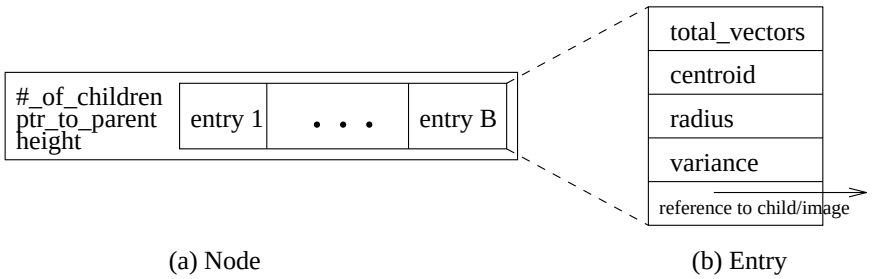


Fig. 5. Node and entry structures

3.1 Algorithm for Insertion

The insertion algorithm is outlined below. The tree is initialized with the root node and an empty leaf child of the root. The value of m used in beam search is predefined by a user. A split strategy similar to that of SS-tree is used in which entries are partitioned to minimize the variance in the dimension which has the biggest variance before splitting.

```
(1)  insert(r: node, p: entry)        // insert p into a tree rooted at r
(2)      initialize a priority queue, Q, and an array N[m]
(3)      for each child n of r do
(4)          add (n, f(n,p)) into Q
(5)      while Q is not empty          // descending
```

(6) $N[1] \leftarrow \text{pop}(Q)$
(7) if $N[1]$ is a leaf or at the same level as p //found the host
(8) $host \leftarrow N[1]$
(9) break
(10) $k \leftarrow 1$ //number of nodes to explore
(11) while Q is not empty and $k < m$
(12) $k \leftarrow k + 1$
(13) $N[k] \leftarrow \text{pop}(Q)$
(14) empty Q
(15) for $i = 1$ to k do
(16) add all child nodes of $N[i]$ into Q
(17) if $host$ has an empty entry
(18) add p into $host$ and modify bounding spheres
(19) else
(20) split $host$, create a new node s
(21) move some entries to s, modify bounding spheres
(22) create an entry e to represent s, insert(r, e) //re-insert new node

The priority queue Q holds the nodes with the smallest $f(n,p)$'s. Obviously, the descending process from step 2 to 12 has a time complexity of $O(mBh)$ where B is the maximal number of entries in a node called *branching factor*, and h is the height of the tree. It is achievable since the priority queue can be built after all elements are added.

3.2 Algorithm for Similarity Search

Given a query image, its features are extracted and sent to the search algorithm to find similar images. There are two kinds of queries, range query and nearest neighbor query. In a range query, users are looking for all images within certain similarity range of the query image, usually measured by the distance of their feature vectors. In a nearest neighbor query, users are looking for k most similar images, called nearest neighbors, to the query image.

Actually, these two kinds of queries can be unified into a single query type and handled by a general algorithm. Given an integer k and a range d, the algorithm finds k closest neighbors with in the distance d to the query image. When k is infinite, this becomes a range query. When d is infinite, this is a nearest neighbor query.

An algorithm is presented below which is a revised version of the one in [13] with some minor errors fixed. Starting at the root node, it progressively goes down the tree, pruning the nodes which do not contain answers. When a leaf node is reached, its entries are put into a priority queue which holds up to k elements as potential answers.

(1) search(r: node; // the root of the indexing tree
k: integer; // number of neighbors to return
d: distance; // maximum distance to the query image

```
        q: vector;)          // features of the query image
(2)   max_d = d;
(3)   initialize an intermediate queue IQ and an answer queue RQ
(4)   IQ ← r
(5)   while IQ is not empty
(6)      next node n ← pop(IQ)
(7)      if n is nonleaf
(8)         for each child c of n
(9)            if dist(c,q) − r(c) ≤ max_d {      //c may contain answer
(10)              insert c into IQ
(11)              if dist(c,q) + r(c) < max_d     // c surely contain answer
(12)                 and RQ is full {             // found k answers already
(13)                 max_d = dist(c,q) + r(c);  //reduce
(14)                 remove all nodes x in IQ if d(x,q) − r(x) < max_d
(15)                 }
(16)              }
(17)     else                                     // leaf node
(18)        for each child c of n
(19)        if dist(c,q) ≤ max_d {                // c maybe an answer
(20)           insert c into RQ
(21)           if RQ overflow {                   // has k + 1 elements
(22)              pop(RQ)
(23)              x = top(RQ)                     //peek top element in queue
(24)              max_d = min(max_d, dist(x,q));
(25)              remove nodes x in IQ if d(x,q) − r(x) < max_d
(26)              }
(27)           }
```

In the algorithm, $dist, r$, and max_d are distance function, radius, and maximal distance from q to be a potential answer, respectively. The priority queues arrange nodes according to their distances to q, with the furtherest one at the top of IQ and the closest one at the top of RQ.

4 Performance Study

The indexing structure and the insertion and search 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 our approach. 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.

4.1 Image Features

The image features used in the experiments fall into three categories: moment invariants, geometry, and Fourier descriptors. Currently in our system, the fea-

tures 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.

Moment Invariants Hu [7] introduced the use of a set of moment invariants based on nonlinear combinations of low-order two-dimensional *Cartesian moments* for pattern recognition. Those features are invariant under image translation, scaling and rotations.

The two-dimensional Cartesian moment, m_{pq}, of order $(p+q)$ of a $N \times N$ gray-scale image $f(x,y)$ over a finite region R is defined as

$$m_{pq} = \int\int_R x^p y^q f(x,y) dxdy \cong \sum_{x=1}^{N}\sum_{y=1}^{N} x^p y^q f(x,y) \tag{2}$$

where the sum is taken over all the $N \times N$ pixels in the image. The centroid $(\overline{x}, \overline{y})$ of $f(x,y)$ is defined by $\overline{x} = m_{10}/m_{00}$, $\overline{y} = m_{01}/m_{00}$. The translation-invariant central moments of order $(p+q)$ are obtained by placing origin at the centroid $(\overline{x}, \overline{y})$:

$$\nu_{pq} = \int\int_R (x-\overline{x})^p (y-\overline{y})^q f(x,y) dxdy \cong \sum_{x=1}^{N}\sum_{y=1}^{N} (x-\overline{x})^p (y-\overline{y})^q f(x,y). \tag{3}$$

In order to obtain the scale-invariants, we define $\mu_{pq} = \nu_{pq}/\nu_{00}^{1+(p+q)/2}$, $p+q \geq 2$ where $\nu_{00} = m_{00} = N$. And also, rotation-invariant features can be constructed from the $\nu_{pq}s$. By using these functions, Hu [7] derived seven moment invariants:

$$\begin{aligned}
M_1 &= \mu_{20} + \mu_{02} \\
M_2 &= (\mu_{20} - \mu_{02})^2 + 4\mu_{11}^2 \\
M_3 &= (\mu_{30} - 3\mu_{12})^2 + (3\mu_{21} - \mu_{03})^2 \\
M_4 &= (\mu_{30} + \mu_{12})^2 + (\mu_{21} + \mu_{03})^2 \\
M_5 &= (\mu_{30} - 3\mu_{12})(\mu_{30} + \mu_{12}) \times [(\mu_{30} + \mu_{12})^2 - 3(\mu_{21} + \mu_{03})^2] + \\
&\quad (\mu_{03} - 3\mu_{21})(\mu_{03} + \mu_{21}) \times [(\mu_{03} + \mu_{21})^2 - 3(\mu_{12} + \mu_{30})^2] \\
M_6 &= (\mu_{20} - \mu_{02}) \times [(\mu_{30} + \mu_{12})^2 - (\mu_{21} + \mu_{03})^2] + 4\mu_{11}(\mu_{30} + \mu_{12})(\mu_{03} + \mu_{21}) \\
M_7 &= (3\mu_{21} - \mu_{03})(\mu_{30} + \mu_{12}) \times [(\mu_{30} + \mu_{12})^2 - 3(\mu_{21} + \mu_{03})^2] + \\
&\quad (\mu_{30} - 3\mu_{12})(\mu_{21} + \mu_{03}) \times [(\mu_{03} + \mu_{21})^2 - 3(\mu_{12} + \mu_{30})^2]
\end{aligned}$$

Geometry Several simple metrics have been used to measure the geometric properties of objects such as area and perimeter. However, these simple measures are scale and size dependent and thus not suitable for similarity search. We first explain some terms below, then give a list of features used in our experiments.

The *area* of an object is expressed as the total number of pixels in the object. The object's *perimeter* is the number of pixels traversed around the boundary of the object starting at an arbitrary initial boundary pixel p_i and returning

to the initial pixel. The centroid μ is the center of gravity for this object, by averaging the coordinates of each pixels in the object. The *bounding rectangle* is the smallest rectangle enclosing the object. A *convex hull* is the minimal convex covering of an object. The radii of gyration, R_{max} and R_{min}, are dynamically equivalent lengths of an object along the directions of axes passing the centroid of the object with zero cross-correlation. The maximum Feret's diameter F_{max} is defined as the longest distance between any two points on the object's perimeter. F_{min} is the width of the object perpendicular to F_{max}. The equivalent circular diameter F_{ecd} is the diameter of an equivalent circular with the same area as the original object.

- *Eccentricity* or *elongation* is defined as:
$$\frac{(\mu_{20} - \mu_{02})^2 + 4\mu_{11}}{area} .$$
- *Extent* or *rectangularity* is defined as the ratio of the area to the bounding rectangle which equals to $F_{max} \times F_{min}$.
- *Form factor* is defined as
$$\frac{\pi \times area}{4 \times perimeter \times perimeter} .$$
- *Roundness* or *compactness* is defined as the ratio of F_{ecd} to F_{max}.
- *Anisometry* is the ratio of R_{max} to R_{min}.
- *Bulkiness* is defined as
$$\frac{12.566 \times R_{max} \times R_{min}}{area} .$$
- *Convexity* is defined as the ratio of perimeter of convex hull to perimeter.
- *Solidity* is defined as the ratio of area to the area of convex hull.
- *Circular variance*, the proportional mean-squared error with respect to solid circle, is defined as
$$\frac{1}{N\mu_r^2} \sum_i (||p_i - \mu|| - \mu_r)^2$$
where μ_r is the mean radius, $\frac{1}{N} \sum_i ||p_i - \mu||$.
- *Elliptic variance*, the proportional mean-squared error with respect to solid ellipse, is defined as
$$\frac{1}{N\mu_{r_e}} \sum_i (\sqrt{(p_i - \mu)^T C^{-1} (p_i - \mu)} - \mu_{r_e})^2$$
where $\mu_{r_e} = \frac{1}{N} \sum_i \sqrt{(p_i - \mu)^T C^{-1} (p_i - \mu)}$, and C is the covariance matrix, $\frac{1}{N} \sum_i (p_i - \mu)(p_i - \mu)^T$.
- *Euler number* is defined as difference between the numbers of connected regions and of holes in an object.

Totally, 11 features are used to measure the object geometry.

Fourier Descriptors By expanding the boundary of a two dimensional object into the frequency domain, Fourier transformation generates a complete set of complex numbers, called *Fourier descriptors* [15], which represent the frequency contents of the boundary of the object. These descriptors are invariant to scaling, rotation, translation, and mirror reflection.

The boundary of a closed shape is a sequence set of successive boundary points T derived from some boundary tracing algorithms. A boundary point $P(x, y)$ is then measured by its curvature function $k(s)$. The parameter s is the path length of the boundary from the initial point P_0 to the current point P. The complex domain of $k(s)$ may be expressed in terms of a pair of cyclic waveforms, the real part, $x(s)$, and the imaginary part, $y(s)$, as

$$k(s) = x(s) + iy(s) \tag{4}$$

which is periodic over the perimeter length T and gives a complete description of the measured shape boundary. By expanding it into Fourier domain, we could obtain the following discrete Fourier series:

$$k\,(s) \cong \frac{1}{T} \sum_{n=0}^{T-1} c(n) \exp\left(i \frac{2\pi s}{T} n\right), \qquad 0 \leq s \leq T-1 \tag{5}$$

where the coefficients $c(n)$, known as *Cartesian Fourier descriptors*, are obtained from

$$c(n) = \sum_{s=0}^{T-1} k(s) \exp\left(-i \frac{2\pi n}{T} s\right), \qquad 0 \leq n \leq T-1\,. \tag{6}$$

Using this method for Fourier descriptors, a truncated series of the curvature function $k(s)$ approximates the original shape. The lower frequency descriptors contain the information of a general key shape. The higher frequency descriptors contain the information about its smaller variant details. By increasing the wave number n, the Fourier descriptors could find more and more fine details of the boundary. In our experiments, a value of 20 is used for n, i.e., the most significant 20 values of $k(s)$ are used as image features.

4.2 Experiments

Several parameters affect the performance of our approach, including m, w_1, w_2, and B. They are listed in Table 1. Different combinations of these parameters are tested to examine their effects on three performance measures: the insertion time, the number of leaf nodes. and the average radius of leaf nodes.

Insertion Time The total insertion time for different m is shown in Fig. 6. We test values from 1 to $B/2$ for m. Note not all values of m apply to all values of B. The weights are set to $w_1 = 1$ and $w_2 = 0$. Under this setting, SS-tree is the special case where $m = 1$.

Table 1. Parameters in algorithms

Parameter	Meaning
B	maximum number of children in a node (branching factor)
m	number of children explored in beam search
w_1	weight for distance
w_2	weight for radius increase

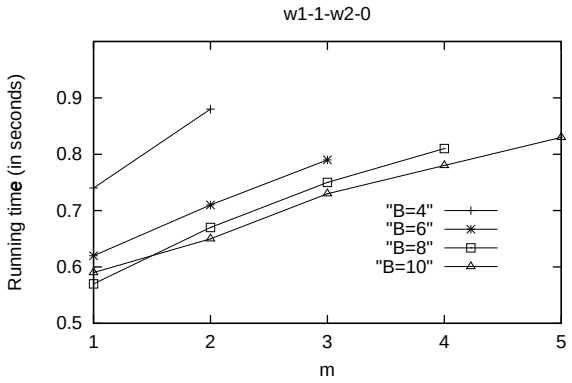


Fig. 6. Insertion time for different values of m

From Fig. 6 it can be observed that the insertion time increases when m grows for a given B. Since the insertion time includes splitting time, it is not always linear to m, but it increases almost linearly with m. For the same m, the time decreases when B increases except when $B = 10$ and $m = 1$. This is probably because the height of tree decreases as B increases, but when B is big, the benefit is offset by the fact that it have to search more children at each level.

Fig. 7 shows the insertion time for different weights of w_1 and w_2 with respect to the branching factor B. The value of m is set to 2 which is the most commonly used value. Similar patterns have been discovered for other values of m. The insertion time increases when B increases because of smaller tree height, but not much when B is large because of more children at each node. The insertion time increases with w_2, especially when B is small. This is because the bigger the w_2, the more emphasis is put on radius increase. This has the tendency to insert the new vector into bigger nodes, which in turn causes more splits and running time. It is interesting to notice that when w_1 is nonzero, the differences are much smaller than when w_1 is zero. It can be concluded that the distance should be taken into consideration when selecting children.

Number of Leaf Nodes The number of leaf nodes in the resulting tree is also computed in our experiments as a measure of tree quality. Our tests show a

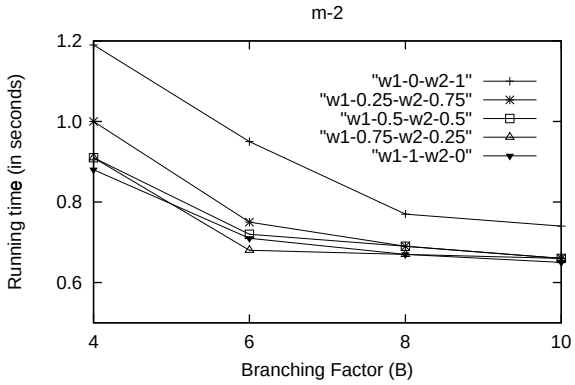


Fig. 7. Insertion time for different weights

tree with smaller number of leaf nodes also have smaller total number of nodes. Besides storage benefit, a smaller tree with fewer node also speeds up search.

The number of leaf nodes for different B is shown in Fig. 8. Again, the weights are set to $w_1 = 1$ and $w_2 = 0$ so that SS-tree is included. When B is small, the beam search reduces the number of leaf nodes. It actually generates more leaf nodes when $B = 10$. This is probably because the beam search algorithm misses the best hosts when B is large. Another interesting finding is that increasing m does not necessarily reduce the number of leaf nodes.

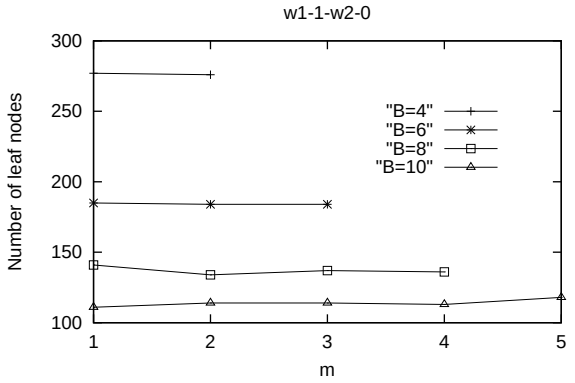


Fig. 8. Number of leaf nodes for different values of m

The effect of weights on the number of leaf nodes is shown in Fig. 9. The value of m is set to 2. The number of leaf nodes increases when w_2 increases.

This echos the earlier observation that the increase of w_2 causes more nodes splitting, especially when $w_1 = 0$.

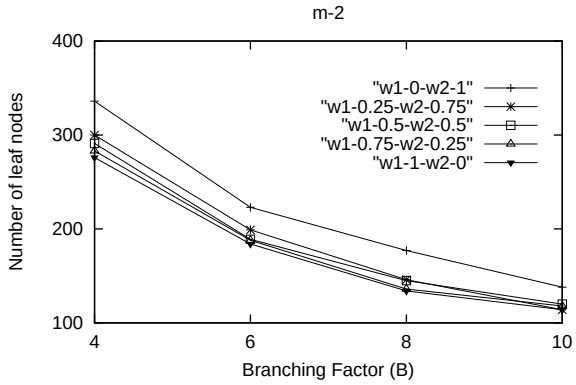


Fig. 9. Number of leaf nodes for different weights

Average Radius of Leaf Nodes Another measure of tree quality is the average radius of leaf nodes which reflects the compactness of leaf nodes.

The average radius of leaf nodes generated for different m is shown in Fig. 10. The average radius of leaf nodes decreases when m increases. This, combined with the fact that less leaf nodes are generated, reveals that the tree is in better shape with beam search.

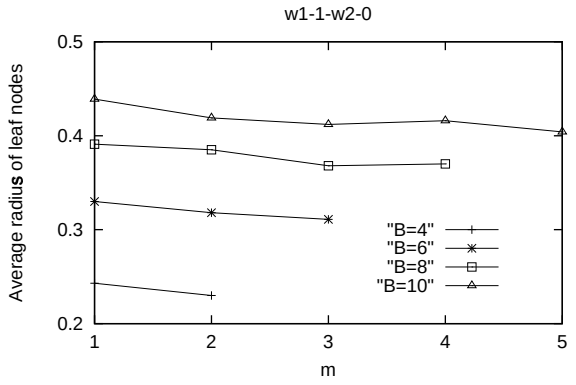


Fig. 10. Average radius of leaf nodes for different values of m

As shown in Fig. 11, the average radius of leaf nodes is the smallest when w_1 and w_2 have non-zero values. This is interesting to notice as it reveals that some mixture of distance and radius increase are more desirable than both extremes when w_1 or w_2 is 0. Generally, equal weights with $w_1 = w_2 = 0.5$ generate good results. Investigation is being conducted to learn more about the weights for different situations.

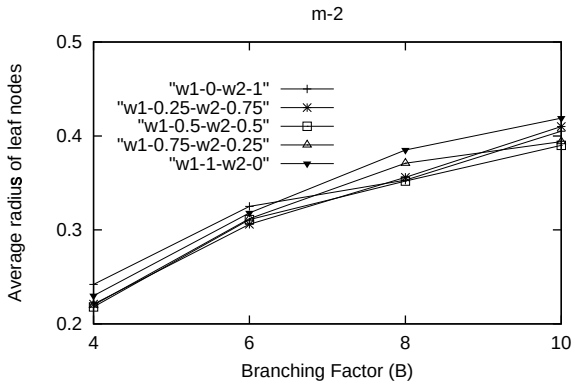


Fig. 11. Average radius of leaf nodes for different weights

To summarize, the quality of the tree is improved with beam search since the number of leaf nodes and average radius of leaf nodes decrease. The increase in total insertion time is insignificant. This demonstrates the advantages of our approach.

5 Discussion

It is assumed that query performance is much more important than update performance and update is much less frequent than query [12]. The same assumptions are taken in this paper. Therefore, it is worth to trade time in insertion/update for better query performance. However, in cases when the update is frequent and costly, the m in the beam search should be reduced, or even degrade to 1 which is equivalent to SS-tree. The introduction of the beam search algorithm, therefore, gives extra flexibility in indexing.

Another issue with indexing structures for similarity search is concurrency control when locking mechanism is used in the image database. The root node has to be exclusively locked during updating. This will not cause serious performance degradation in most cases since most CBIR systems have infrequent updates and they are usually done in batch mode when user querying is not so active. Besides, the strict consistency requirements may be relaxed in some cases. For example, missing one target image by reading a dirty node is not a serious problem, if, say,

20 images are returned. It should be pointed out that the introduction of beam search in our approach does not induce extra burden compared with SS-tree because both have to lock the root node.

Some possible future extensions of our work are discussed below.

- The two key factors in insertion are tree descending and node-splitting strategy. As we have discovered through our experiments, the introduction of beam search in tree descending has some effect on the resulting tree, but not dramatically different from that of SS-tree. This is mainly because we used the same splitting strategy as that of SS-tree. We are currently evaluating other splitting strategies such as those in [3]. It will be interesting to study how the combination of the two can improve the insertion process and thus the tree quality.
- The Euclidean distance is used in our experiments. However, it is not always appropriate for high-dimensional feature space. Other similarity metrics, such as Minkowski's L^p distance, cosine distance, and normalized correlation, may be better candidates for some applications.
  Many other image features have been developed for CBIR, including color, texture, and spatial relationship[5, 11, 9]. It is interesting to apply our approach to the broader feature space.
  Since our approach is independent of similarity metric and image features, it can be applied to other similarity metrics or image features with little modification.
- The effectiveness of beam search in insertion will be more apparent when the tree is deep. This is because the best host will also be hidden deep down the tree and our approach will have advantage to find it than others like SS-tree. The nest step in our research is to expand the image database so that we can evaluate our approach in a larger database.

6 Conclusion

The problem of overlapping in high-dimensional indexing structure has been examined. A novel approach is proposed to solve the problem by introducing a heuristic algorithm, the beam search method, in tree insertion. When a new vector is inserted, the beam search will explore several children in order to find a better host. This will improve the quality of the indexing and thus improve similarity search performance. In addition, a heuristic child selection criterion is developed which takes into account both the distance and the increase of radius when searching for best hosts.

Our experiments on a image database show that the integration of beam search in insertion can improve the quality of indexing and query without incurring much run-time overhead. Besides, the effects of parameters on the indexing results are also studied. Finally, related issues and future directions of the research are discussed.

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 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. 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.
4. 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.
5. V. Gudivada and V. Raghavan. Content-based image retrieval systems. In *IEEE Computer*, pages 18–22, Vol 28, No. 9, 1995.
6. A. Guttman. R-tree: A dynamic index structure for spatial searching. In *Proc. 1984 ACM-SIGMOD Int. Conf. Management of Data*, June 1984.
7. M.-K. Hu. Pattern recognition by moment invariants. *Proc. IRE*, 49, Sept. 1961.
8. 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.
9. P. Kelly, T. Cannon, and D. Hush. Query by image example: the CANDID approach. In *Proc. Storage and Retrieval for Image and Video Databases III*, pages 238–248, SPIE Vol. 2420, 1995.
10. 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.
11. W. Niblack, R. Barber, W. Equitz, M. Flickner, E. Glasman, D. Petkovic, P. Yanker, C. Faloutsos, and C. Taubin. The QBIC project: Querying images by content using color, texture, and shape. In *Proc. Storage and Retrieval for Image and Video Databases*, pages 173–187, SPIE Vol. 1908, 1993.
12. 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.
13. D. White and R. Jain. Algorithms and strategies for similarity retrieval. In *Technical Report, Department of Computer Science and Engineering*, University of California, San Diego, 1997.
14. P. Winston. *Artificial Intelligence (2nd Ed.)*. Addison-Wesley, 1984.
15. C. T. Zahn and R. Z. Roskies. Fourier descriptors for plane closed curves. *IEEE Trans. on Computers*, 21:269–281, March 1972.