

Techniques and Data Structures for Efficient Multimedia Retrieval Based on Similarity

Guojun Lu, *Member, IEEE*

Abstract—As more and more information is captured and stored in the digital form, many techniques and systems have been developed for indexing and retrieval of text documents, audio, images, and video. The retrieval is normally based on similarities between extracted feature vectors of the query and stored items. Feature vectors are usually multidimensional. When the number of stored objects and/or the number of dimensions of the feature vectors are large, it will be too slow to linearly search all stored feature vectors to find those that satisfy the query criteria. Techniques and data structures are thus required to organize feature vectors and manage the search process so that objects relevant to the query can be located quickly. This paper provides a survey of these techniques and data structures.

Index Terms—Efficient search, multidimensional data structures, multimedia databases, multimedia indexing and retrieval.

I. INTRODUCTION

As more and more information is captured and stored in the digital form, many techniques have been developed for indexing and retrieval of text documents, audio, images, and video [24]–[36]. The common indexing and retrieval process can be summarized as follows. Information items (in various media types) in the database are preprocessed to extract features (represented as vectors) and they are indexed based on these features. During information retrieval, a user's query is processed and main features are extracted. The query's main features are then compared with features or index of each information item in the database. Information items the features of which are deemed similar to those of the query are retrieved and presented to the user. Feature vectors are usually multidimensional. For example, in the vector space model for text document retrieval [26], [27], the number of dimensions of feature or document vectors is equal to the number of terms (usually hundreds to thousands) used in the document collection. For audio indexing and retrieval [28], [29], the number of dimensions of feature vectors is equal to the number of features used (such as brightness, variance of zero crossing rate and silence ratio). For color-based image indexing and retrieval [30], [32], the number of dimensions of a color histogram is equal to the number of color bins used (usually at least 64). Similarly, texture and shape are also represented by multidimensional vec-

tors [30], [32], [35]. During retrieval, the query is also represented by a multidimensional vector. The retrieval is based on the similarity or distance between the query vector and the feature vectors of the stored objects. When the number of stored objects and/or the number of dimensions of the feature vectors are large, it will be too slow to linearly search all stored feature vectors to find those that satisfy the query criteria. The situation is made worse as a multimedia information system normally uses a number of different feature vectors. For example, query by image content (QBIC) [30] allows retrieval based on a combination of color, color layout, and texture feature vectors. In the MPEG-7 standard currently being standardized, many feature vectors (called descriptors) are specified [37]. Techniques and data structures are thus required to organize feature vectors and manage the search process so that feature vectors (objects) relevant to the query can be located quickly.

Currently, not all multimedia information retrieval systems use techniques or data structures for quick retrieval. For example, no data structure is used in the QBIC demonstration system [30]. Instead, the distance from each image to each of other images in the collection is precomputed and stored. In this way, images can be retrieved quickly when the query is one of the images in the collection. Some systems make use subject classification to improve retrieval speed (items in a chosen class are compared and searched instead of the entire collection) [33]. Most work in index or data structures is carried out in database and geographic information system (GIS) community. These data structures, including R-tree and k -d tree, are being extended for multimedia information retrieval. For example, in a recent system called ImageScape [38], k -d tree is used for organizing image feature vectors. This paper provides a survey of these and other techniques and data structures.

The main aim of these techniques and data structures is to divide the multidimensional feature space into many subspaces so that only one or a few subspaces need to be searched for each query. Different techniques and data structures differ in how subspaces are formed and how relevant subspaces are chosen for each query.

There are three common query types: 1) point query; 2) range query; and 3) k nearest-neighbor query. In the point query, a user's query is represented as a vector and those objects the feature vectors of which exactly match the query vector are to be retrieved.

In the range query, the user's query is represented by a feature vector and a distance range. All objects which have distances from the query vector that are smaller than or equal to the specified distance range are retrieved. There are many different distance metrics. The ones most commonly used are the L_1 and L_2

Manuscript received September 20, 2000; revised July 13, 2001. The associate editor coordinating the review of this paper and approving it for publication was Dr. Anna Hac.

The author is with the Gippsland School of Computing and Information Technology, Monash University, Churchill, Vic. 3842, Australia (e-mail: guojun.lu@infotech.monash.edu.au).

Digital Object Identifier 10.1109/TMM.2002.802831.

norms (Euclidean distance). An alternative range query is specified by a value range for each of the feature vector's dimensions.

In the k nearest-neighbor query, the user's query is specified by a vector and an integer k . The k objects which have distances from the query vector that are the smallest are retrieved.

A useful technique or data structure should support all three query types. Data structures can also be optimized for a certain type of query if it is known that only one type query will be commonly used for a particular application.

The basic operations carried out on a data structure are search, insertion, and deletion. Search efficiency is considered the most important criterion for selecting data structures because search is normally carried out on-line (and thus needs quick response) and will be carried out many times (the purpose of building the multimedia database is for quick searching). Ease of dynamic insertion and deletion of objects is helpful but not critical as these operations are carried out off-line and by system administrators only. Furthermore, insertion and deletion can be carried out in batches: objects to be inserted or deleted are accumulated in insertion and deletion lists and insertion and deletion are only carried out when a sufficient number of objects need to be inserted or deleted. When sufficient changes (insertions and deletions) have occurred, the data set can be reorganized completely to achieve efficient storage and search.

The rest of the paper is organized as follows. Section II describes a number of filtering techniques to reduce the search space. Some of these techniques are application or feature dependent, others can be applied to most applications.

Section III discusses a number of important multidimensional data structures. In Section III-A, the main concepts of B^+ and B-trees are described. These trees are for organizing objects with single valued keys (i.e., one-dimensional (1-D) feature vectors). However, their concepts are important and are used in multidimensional data structures. Indeed, many multidimensional data structure can be considered as multidimensional extensions of B^+ and B-trees.

Section III-B describes clustering techniques for efficient search. The main idea is to put similar objects or feature vectors in the same groups or clusters and only relevant clusters are searched during retrieval. Clustering can be considered to be a general approach to data organization and multidimensional data structures. Different data structures differ in how clusters are formed and represented.

Section III-C presents the multidimensional B^+ -tree (MB^+ -tree) which is a direct extension of the B^+ -tree. Section III-D discusses the k -dimensional tree (k -d tree) which is a multidimensional extension of the binary tree. Section III-E describes grid files.

Section III-F describes one of the most common and successful multidimensional data structures called the R-tree and a number of its variants.

Section III-G discusses the telescopic-vector-tree (TV-tree) which tries to reduce the number of effective dimensions used for indexing and building the tree.

Section IV summarizes the paper with a brief discussion of the search performance of multidimensional data structures and challenges in research of this area.

II. FILTERING PROCESSES FOR REDUCING SEARCH SPACE

It is usually possible to reduce the search space using filtering processes based on certain criteria. Some of these criteria and filtering processes are application or feature dependent, others are not and can be used in most retrieval processes. The basic idea is as follows. The filtering processes, such as those based on attributes (such as dates and author names), can be carried out very efficiently to select items satisfying certain criteria (e.g., containing the attribute). The search based on complicated features (represented by multidimensional vectors) is then carried out on the selected items only. As the number of selected items is significantly smaller than the total number of items in the database, the overall retrieval process can be completed quickly. In this section, we describe the following filtering methods: filtering with classification and structured attributes, methods based on the triangle inequality, methods specific to color histogram-based retrieval, and latent semantic indexing (LSI) for vector space-based text retrieval.

A. Filtering With Classification, Structured Attribute, and Keywords

Structured attributes, such as the author name and the creation date, are associated with most multimedia objects. If the user is only interested in items satisfying certain attributes, we can use these attributes to do a preliminary selection and then carry out the search based on the more complicated features of the selected items.

When subject classification is available, the user can select the subjects of interest and search the items within these subjects only.

The above two approaches can be applied generally, without restrictions on the features used. For some specific features, some special attributes can be used to reduce the search space. For example, in the region-based shape indexing and retrieval method described in [1] and [2], we can use shape eccentricity as the filtering criterion—only shapes within the specified eccentricity range need to be searched.

B. Methods Based on the Triangle Inequality

Most feature distance measures, such as Euclidean distance, are metrics. Metrics have a property called the *triangle inequality*. We can use this property to reduce the number of direct feature comparison in a database search [3]. The triangle inequality states that the distance between two objects cannot be less than the difference in their distances to any other object (third object). Mathematically, the triangle inequality is written as

$$d(i, q) \geq |d(i, k) - d(q, k)|$$

where d is a distance metric; and i , q , and k represent feature vectors.

The above inequality is true for any k . Therefore, when multiple features are used as the comparison objects, we have the following:

$$d(i, q) \geq \max_{1 \leq j \leq m} |d(i, k_j) - d(q, k_j)|$$

where m is the number of features used as the comparison objects.

We can apply the triangle inequality to multimedia information retrieval as follows.

- 1) We select m feature vectors as a comparison base. Normally, m is much smaller than the total number of items in the database.
- 2) For each item i in the database and each comparison vector k_j , we calculate $d(i, k_j)$ off-line and store it in the database.
- 3) During retrieval, we calculate the distance $d(q, k_j)$ between the query q and each of the comparison vectors k_i .
- 4) We find $l(i) = \max_{1 \leq j \leq m} |d(i, k_j) - d(q, k_j)|$ for each database item i .
- 5) Only items with $l(i)$ less than a preselected threshold T are selected for calculating the distance from q (being $d(q, i)$). The distances between q and other database items need not be calculated as they are guaranteed to be larger than the threshold T which is selected according to the feature used and the user's requirement.

Note that the unselected items based on $l(i)$ definitely have distances from q larger than T , but not all selected items will have distances from q less than T . Assume n is the number of items with $l(i)$ less than a preselected threshold T , the total number of distance calculation is $m + n$. If $m + n < N$, where N is the total number of items in the database, the use of the triangle inequality will improve retrieval efficiency. This should be possible when m and T are properly chosen.

The filtering process based on the triangle inequality can be used in all retrieval techniques the distance measures of which are metrics.

C. Methods Specific to Color Histogram-Based Retrieval

This section describes a number of search space reduction methods that are specific to color histogram-based image retrieval. This retrieval method represents images in the database and query images using histograms with n bins each. The distance between the query and each of the images in the database is calculated as their corresponding histogram distance. Therefore, there are two options to reduce the required amount of computation. The first is to reduce the number of bins n . However, a small n may cause low retrieval accuracy as quite different colors will be classified into the same bin when the number of bins is too small. The second option is to select only a subset of database images for calculating the distances from the query. The number of images in the subset can be much smaller than the total number of images in the database. The question is how to determine this subset.

The solution is to combine the above two options. We first use histograms with very few bins to select potential retrieval candidates, and then use the full histograms to calculate accurate distances between the query and the potential candidates. Striker and Dimai [4] and Ng and Tam [5] describe the details of how to chose the initial bins to achieve optimal results.

A special case of the above idea is to carry out filtering based on the average color of the images [6], [7]. Suppose RGB color space is used for image representation (the scheme is equally

applicable to other color spaces). We find the average color of an image $x = (R_{\text{avg}}, G_{\text{avg}}, B_{\text{avg}})^T$ as follows:

$$\begin{aligned} R_{\text{avg}} &= \frac{\sum_{p=1}^P R(p)}{P} \\ G_{\text{avg}} &= \frac{\sum_{p=1}^P G(p)}{P} \\ B_{\text{avg}} &= \frac{\sum_{p=1}^P B(p)}{P} \end{aligned}$$

where P is the number of pixels in the image; and $R(p)$, $G(p)$, and $B(p)$ are the red, green, and blue components of pixel p , respectively. Given the average color vectors \bar{x} and \bar{y} of two images, we define $d_{\text{avg}}(\cdot)$ as the Euclidean distance between these two vectors as follows:

$$d_{\text{avg}}(\bar{x}, \bar{y}) = \sqrt{\sum_{i=1}^3 (x_i - y_i)^2}.$$

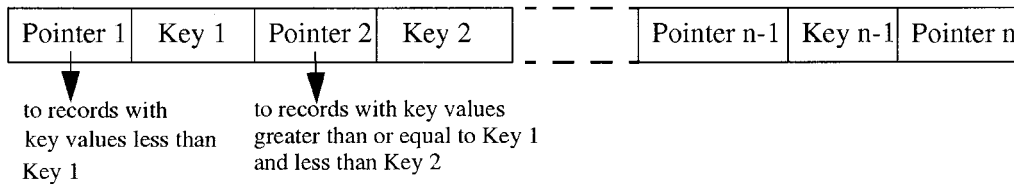
It has been shown that $d_{\text{avg}}(\cdot)$ is a lower bound for the actual distance calculated using the full histogram. That is, the full histogram distance will be no less than $d_{\text{avg}}(\cdot)$. With this result, we can select potential candidates based on $d_{\text{avg}}(\cdot)$ for full histogram distance calculation.

D. Latent Semantic Indexing for Vector Space-Based IR

In the vector space model for text document information retrieval, each document is represented by an N -dimensional term weight vector, each element of the vector being the weight of each of N terms in that document. If a document collection has M documents, then the collection can be represented as a matrix \mathbf{A} of dimension $M \times N$. During retrieval, the query is also represented in an N -dimensional term weight vector. The similarity between the query and each of the stored documents is calculated as either the dot product or the cosine coefficient between the query vector and the document vector.

The above straightforward approach has two main weaknesses. First, a large document collection (such as a library) contains millions of documents with many thousands of terms, i.e., M and N are both very large. Therefore, a very large amount of storage is required. For example, if a library has one million documents with 10 000 terms, we need 10 GB of storage if each element is stored as 1 B. A few years ago, this was a huge amount of storage. Second, at least M multiplications of N -dimensional vectors are required during retrieval if the dot product similarity measurement is used and more are required if the cosine coefficient similarity measure is used. When M and N are large, the required time to complete these calculations is not acceptable for on-line retrieval.

LSI was developed to partially solve the above-mentioned problems [7], [8], [22]. (We say "partially" because other techniques, such clustering and multidimensional data structures to be discussed later, should be combined with LSI for more efficient searching.) The basic idea of LSI is to group similar terms together to form concepts or topics and the documents are then represented by these concepts. As the number of concepts is much smaller than the number of terms, less storage and computation are required. In addition, because of the LSI's ability


 Fig. 1. Node structure of the B^+ -tree.

to automatically group co-occurring and similar terms to create a thesaurus, retrieval effectiveness has also been reported to be improved [8].

LSI is based on the concept of singular value decomposition (SVD). The theorem of SVD is as follows.

Any $M \times N$ real matrix \mathbf{A} can be expressed as

$$\mathbf{A} = \mathbf{U} \times \mathbf{S} \times \mathbf{V}^t$$

where

- \mathbf{U} column-orthonormal $M \times r$ matrix;
- r rank of the matrix \mathbf{A} ;
- \mathbf{S} diagonal $r \times r$ matrix;
- \mathbf{V} column-orthonormal $N \times r$ matrix.

That \mathbf{U} is column-orthonormal means $\mathbf{U}^t \times \mathbf{U} = \mathbf{I}$, where \mathbf{I} is the identity matrix. When \mathbf{S} is nondecreasing, i.e., its elements are sorted in descending order, the above decomposition is unique.

In the context of text document retrieval, the rank r of \mathbf{A} is equal to the number of concepts. \mathbf{U} can be thought of as the document-to-concept similarity matrix, while \mathbf{V} is the term-to-concept similarity matrix. For example, $u_{2,3} = 0.6$ means that concept 3 has weight 0.6 in document 2 and $v_{1,2} = 0.4$ means that the similarity between term 1 and concept 2 is 0.4.

Based on the SVD, we can store matrices \mathbf{U} , \mathbf{S} , and \mathbf{V} instead of \mathbf{A} , reducing the storage requirement significantly. During retrieval, the query document similarity is calculated as follows. The query vector q in term-space is translated into q_c in concept-space by multiplying by \mathbf{V}^t as follows:

$$q_c = \mathbf{V}^t \times q.$$

The similarity between the query and each of the documents is calculated as the dot product or the cosine coefficient between q_c and each of the rows of \mathbf{U} . Therefore, using LSI, we manipulate r -dimensional vectors instead of N -dimensional vectors during the similarity calculation. As r is many times smaller than N , the calculation using LSI is many times faster than using the straightforward method. The search or retrieval efficiency can be further improved by clustering the rows of \mathbf{U} based on their similarity.

III. MULTI-DIMENSIONAL DATA STRUCTURES

It would be slow to carry out the similarity or distance calculation between the query and each of stored items within the search space sequentially one by one, even after search space reduction using the techniques discussed in the previous section. The retrieval efficiency can be significantly improved by organizing the feature vectors into certain data structures. A number of important data structures are reviewed in this section.

A. B^+ - and B-Trees

B^+ - and B-trees are for organizing 1-D feature vectors or single valued keys of stored items. However, many important properties are used in these trees and many multidimensional data structures are developed based on the ideas of B^+ - and B-trees.

A B^+ -tree is a hierarchical structure with a number of nodes [9]. Each node (including the root and leaves) in a B^+ -tree contains n pointers and $n - 1$ keys, as shown in Fig. 1. We call n the degree or order of the tree (being the maximum number of children any node has). Pointer 1 is used to access all records which have key values that are less than key 1. Pointer 2 is used to access all records which have key values that are greater than or equal to key 1 but less than key 2, and so forth. The last pointer, pointer n , is used to access all records which have key values that are larger than or equal to key $n - 1$.

Fig. 2 shows a B^+ -tree with an order of four. Each node has room for four pointers and three key values. Some key value and pointer fields are blank as these nodes have less than three key values (i.e., they are not full). When the rightmost pointer is not used, a zero is written there to indicate the end of the node. The leftmost pointer of the root points to records which have key values that are less than 200. The leftmost pointer of the first level 1 node is used to access all records which have key values that are less than 50, while the second pointer is used to access records which have key values that are greater than or equal to 50 and less than 100. The first three pointers of each of leaf node, if present, point to actual data records. For example, the first pointer of the first leaf node points to the data record identified by key value 10. The fourth pointer of each of the leaf nodes except for the last leaf node points to the next leaf node. These pointers provide sequential access to data records. That is, data records can be sequentially accessed by following the leaf nodes.

When building a B^+ -tree, we must follow the rules as follows.

- 1) The tree must be balanced. That is, every path from the root to a leaf node must have the same length.
- 2) The root node must have at least two children, unless only one record is present in the tree.
- 3) If the tree has order n , each node, except for the root and leaf nodes, must have between $n/2$ and n pointers. If $n/2$ is not an integer, round up to determine the minimum number of pointers. That is, each nonroot and nonleaf node should be at least half full.
- 4) If the tree has order n , the number of key values in a leaf node must be between $(n - 1)/2$ and $n - 1$. If $(n - 1)/2$ is not an integer, round up to determine the minimum number of key values.

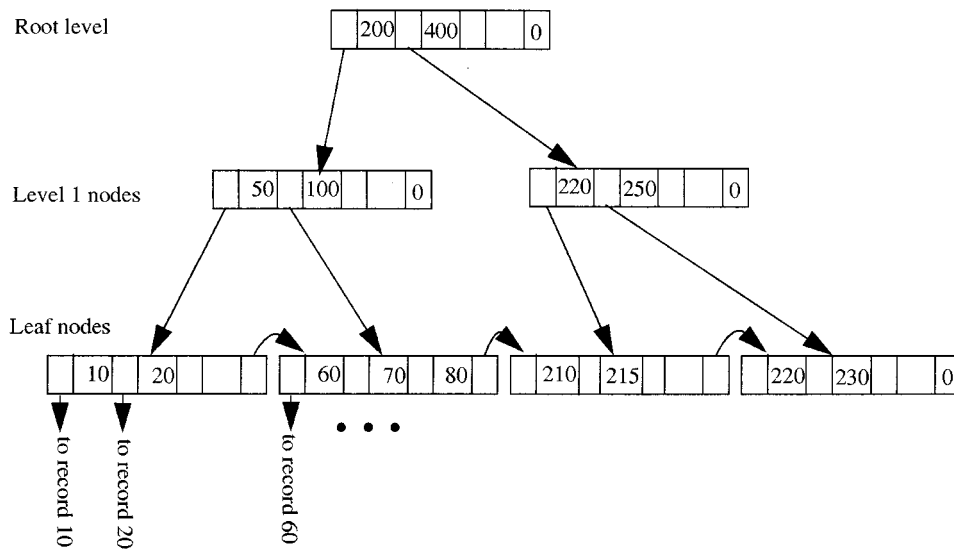
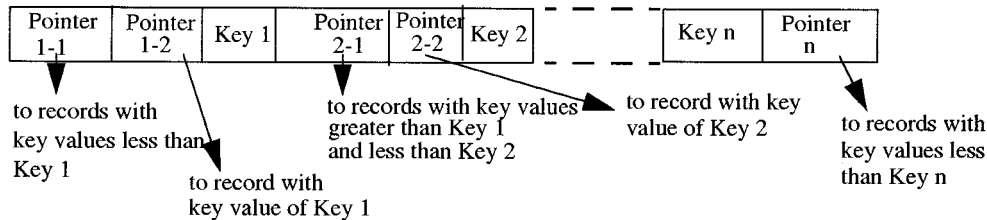
Fig. 2. Example B⁺-tree.

Fig. 3. Nonleaf node structure of a B-tree.

- 5) The number of key values contained in a nonleaf node is one less than the number of pointers.

Direct access in B⁺-trees are very efficient. If the order of the tree is n and number of record is N , the required number of key value comparison and memory (or disk) access is in the order of $O(\log_n N)$. For example, we can find any record in a B⁺-tree with an order of 10 and 100 000 records using five key value comparisons and memory accesses.

The above five rules should not be violated after record insertion and deletion. The insertion and deletion processes can be complicated if the nodes are full before insertion or if the nodes are too empty after deletion.

In B⁺-trees, some key values are repeated in many nodes. For example, in the tree of Fig. 2, the key value 220 is repeated in the second level 1 node and the last leaf node. A B-tree structure is similar to B⁺-tree, but it does not have this type of key value repetition. It achieves this by using two pointers preceding each key value in nonleaf nodes (see Fig. 3 for the general node structure of B-trees). One pointer points to the record with key values greater than the previous key value and less than the key value, and the other pointer points to the record with the key value.

B. Clustering

The cluster-based model was developed for text document retrieval, but the same principle can be applied to any feature with any similarity measurement [10], [23]. It is a generalized approach to multidimensional data structures.

The basic idea of using clustering for efficient retrieval is as follows. Similar information items are grouped together to form clusters based on a certain similarity measurement. Each cluster is represented by the centroid of the feature vectors of that cluster. During retrieval, we calculate the similarity between the query vector and each of the clusters (represented by their centroids). Clusters that have similarities to the query vector which are greater than a certain threshold are selected. Then the similarity between the query vector and each of the feature vectors in these clusters is calculated and the k nearest items are ranked and returned.

As an example, feature vectors in Fig. 4 are grouped into 11 clusters. During retrieval, the query vector is compared with each of the 11 cluster centroids. If we find that the centroid of cluster 2 is most similar to the query vector, we then calculate the distance between the query vector and each of the feature vectors in cluster 2. The number of required distance calculations is much smaller than the total number feature vectors in the database.

In the above clustering-based retrieval method, the similarity is calculated between the query and each of the centroids and each of feature vectors within selected clusters. When the number of clusters is large, multiple levels of clusters can be used to reduce the number of similarity calculations between the query and centroids. Similar clusters are grouped to form superclusters. During retrieval, the query vector is first compared with supercluster centroids, and then with cluster

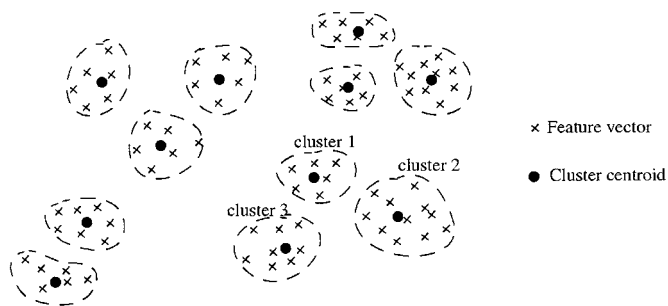


Fig. 4. Clustering example.

centroids within selected superclusters, and finally with feature vectors within selected clusters.

Clustering not only makes retrieval efficient, but also facilitates browsing and navigation. For browsing and navigation, one representative item the feature vector of which is closest to the centroid of its cluster is displayed for each cluster. If the user is interested in a representative item, he or she can view other items in the cluster.

Clustering techniques can be used together with other data structures for higher search efficiency. Similar items are grouped into clusters. Centroids of clusters and/or items within each cluster are organized using a certain data structure for efficient search.

C. Multidimensional B^+ -Tree

The MB^+ -tree is an extension of the standard B^+ -tree from one dimension to multiple dimensions [11]. The structure and the insertion and deletion algorithms of MB^+ -trees are very similar to those of B^+ -trees. However, the search methods are different as MB^+ -trees can support similarity queries (i.e., range and k nearest-neighbor queries).

For ease of visualization and explanation, we describe the MB^+ -tree in a two-dimensional (2-D) space. Extension to higher dimensions is explained later.

1) *Overview of MB^+ -Trees in Two-Dimensional (2-D) Space:* In 2-D space, each feature vector has two dimensions and can be considered as a point in a 2-D (feature) space. The entire feature space can be considered as a large rectangle identified by its lower-left corner (x_{\min}, y_{\min}) and upper-right corner (x_{\max}, y_{\max}) . All feature vectors are within this rectangle. Therefore, if we divide this feature space into rectangular regions so that each region has a similar number of feature vectors and these regions are somehow ordered, feature vectors can be accessed based on their corresponding region numbers. MB^+ -trees are built based on this idea. Therefore, the MB^+ -tree is obtained by extending the B^+ -tree in the following ways:

- 1) replace each key value with a rectangular region;
- 2) the pointers of leaf nodes point to lists of feature vectors within corresponding rectangular regions.

We use an example to illustrate the basic idea. Fig. 5 shows a feature space that is divided into eight regions. Each region is identified by its lower-left corner and upper-right corner. These eight regions do not overlap, but cover the entire feature space. They are ordered in ascending order as follows (we will discuss

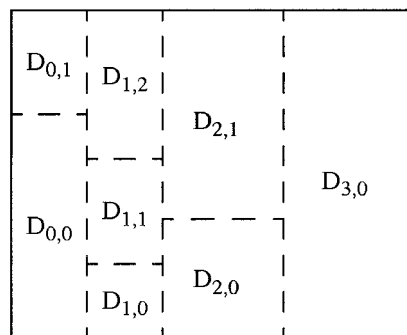


Fig. 5. Rectangle regions of a feature space (based on [11]).

how the feature space is divided into regions and how regions are ordered later):

$D_{0,0}, D_{0,1}, D_{1,0}, D_{1,1}, D_{1,2}, D_{2,0}, D_{2,1}, D_{3,0}$.

Fig. 6 shows a possible 2-D MB^+ -tree for the above regions. Each pointer (except for the last one) in each leaf node points to a list $L_{m,n}$ containing the following information:

- 1) all feature vectors belonging to the corresponding region;
- 2) a pointer associated with each feature vector linking to the actual multimedia object.

In the following, we describe the details of building an MB^+ -tree and search processes for different types of queries.

2) *Building a 2-D MB^+ -Tree:* We first decide the order of the tree and the maximum number of feature vectors in each region (the maximum size of $L_{m,n}$). An MB^+ -tree is built up by inserting one feature vector at a time. Initially, the 2-D MB^+ -tree has only one leaf node that is also the root of the tree. The node has only one region corresponding to the entire feature space, and there is only one list. Each feature vector inserted is simply added to the unique list until it is full. The splitting operation is required for the next insertion and the space will be divided into two vertical strips with a similar number of feature vectors in each strip. The process continues and the space is divided into smaller strips. When the width of a vertical strip reaches a preset value, horizontal dividing is used within the vertical strip. For a region obtained from horizontal dividing, only horizontal dividing is applied.

Regions are ordered in the following way. The vertical strips are ordered from left to right in the horizontal dimension. The regions within the same vertical strip are ordered from bottom to top in the vertical dimension.

When a list $L_{m,n}$ becomes too small because of the deletion of feature vectors, it is merged with another list. Two neighboring regions within the same vertical strip can be merged, and a vertical strip not divided by a horizontal line can be merged only with another such vertical strip. After two lists are merged together, one leaf entry will be deleted from the tree and parent nodes may need to be rearranged.

In summary, the insertion and deletion operations are similar to those of B^+ -trees. The only difference is that the tree is organized based on the ordering of the regions instead of key values and each region corresponds to a list of feature vectors instead of a data record.

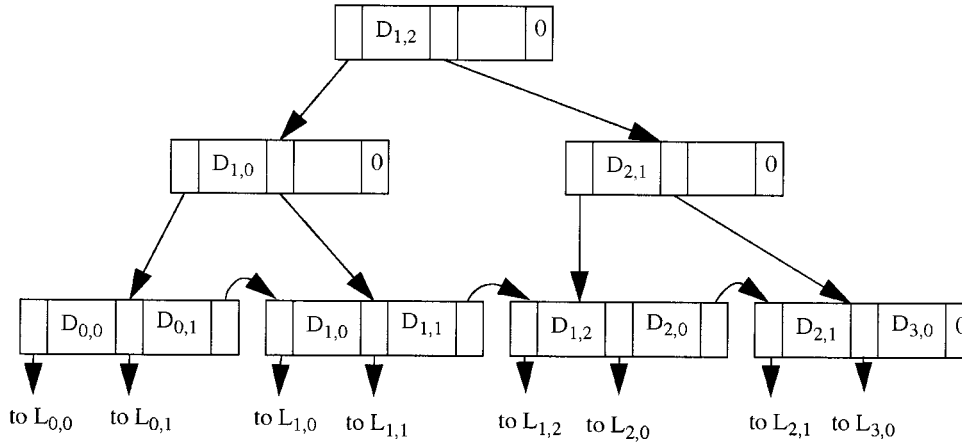


Fig. 6. Possible MB^+ -tree of the regions in Fig. 5.

3) *Search in MB^+ -Trees*: All three types of queries—point, range, and k nearest-neighbor—can be supported by an MB^+ -tree.

a) *Point query*: In this type of query, a query feature vector (x, y) is given. Starting from the root, we find the region that contains the query vector. Then we scan the feature vector list to determine if the region contains the required vector. We can use the point query search to implement a k nearest-neighbor query search.

b) *Range query*: A range query is converted into a rectangle query, specified by its lower-left and upper-right corners. Starting from the root, we find all regions that overlap with the query rectangle. We then scan each of the lists associated with these regions to find all feature vectors that are within the query rectangle.

c) *k Nearest-neighbor query*: Given a point (x, y) and a positive integer k , the k nearest-neighbor query tries to find the k nearest feature vectors with respect to the chosen distance measure. The common distance measure is the weighted Euclidean distance.

We can implement the k nearest-neighbor query in two approaches. First, we can estimate the query rectangle centered at (x, y) from the query and use the range query search method to find candidate feature vectors. Then, we calculate the distance between (x, y) and each of these candidate feature vectors to find the k nearest feature vectors. The estimated query rectangle can be too small or too large, so a few iterations may be required. Second, we can use the point query search to find the region containing (x, y) . We then calculate the distance between (x, y) and each of the feature vectors in the region to find the k nearest feature vectors. However, there are cases where feature vectors in other regions may have shorter distances to (x, y) and should be included in the k nearest feature vectors. We can detect these cases by checking whether the distance to the k th feature vector (after ordering in ascending order of distances) is larger than the distance from (x, y) to any of the boundaries of the region. If so, we have to search for the feature vectors in the neighboring region sharing that boundary. Alternatively, we can draw a small circle centered at (x, y) (suppose Euclidean distance measure is used) and increase the radius until either k feature vectors are contained in the circle or the circle intersects with another re-

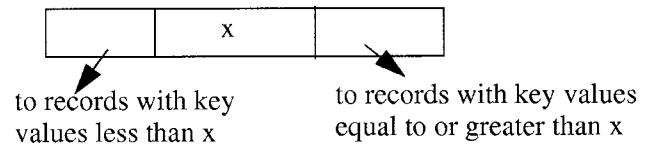


Fig. 7. Node structure of the binary tree.

gion. If the circle containing the k feature vectors is within the region, we are done. Otherwise, we have to search the region intersecting with the circle.

4) *Higher-dimensional MB^+ -trees*: The idea of 2-D MB^+ -trees can be easily generalized into a k -d space for $k > 2$. We order the k dimensions in a desired manner and name them the first dimension, the second dimension, and so on. During insertion, we divide the space along the first dimension first until the edge of a region reaches a certain preset value on that dimension. We then divide each hypercube on the second dimension independently, and so on.

D. The k -d Trees

The k -d tree is an extension of the binary tree [12]. Therefore, let us quickly review the principle of the binary tree. A node in the binary tree has three elements: a key value x , a left pointer pointing to the records having key values less than x , and a right pointer pointing to records with key values equal to or greater than x (see Fig. 7). Each key value in the tree is commonly associated with a data record. When building a binary tree, the first record inserted will be the root. The second record will go left or right depending on whether its key value is less than the root key value. This process applies for each insertion. The binary tree is unbalanced and its structure depends on the order in which records are inserted into the tree.

In a k -d tree, each key is a k -d vector instead of a single-valued number. Therefore, to extend the binary tree to a k -d tree, we have to decide how to branch at each level. If we name the k -dimensions as dimension 1, dimension 2, and so on to dimension k , and name the root of a k -d tree as level 1, its child as level 2, and so on, the k -d tree is constructed as follows. At level 1, the tree is branched based on the value of the first dimension of the k -d vector; at level 2 the tree is branched based on the

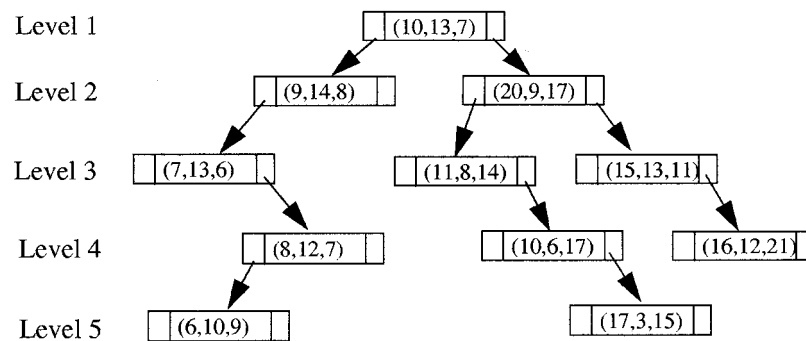


Fig. 8. Example 3-D tree.

value of the second dimension of the k -d vector, and so on. This process continues until all dimensions have a turn and we then start from the first dimension again. Fig. 8 shows a three-dimensional (3-D) tree built from the following 3-D vectors:

(10, 13, 7), (9, 14, 8), (20, 9, 17), (7, 13, 6), (8, 12, 7)
 (6, 10, 9), (11, 8, 14), (15, 13, 11), (10, 6, 17)
 (16, 12, 21), (17, 3, 15).

Note that the tree depends on the order in which the vectors are inserted. The first vector (10, 13, 7) is the root. When inserting the second vector (9, 14, 8), the branch is based on the value of the first dimension. Since 9 is smaller than 10, (9, 14, 8) is inserted to the left of the root. When inserting the third vector (20, 9, 17), since 20 is larger than 10, it is inserted to the right of the root. When inserting the fourth vector (7, 13, 6), we first compare 7 with 10 and decide it should go left as 7 is smaller than 10. We then compare the second elements and find that 13 is smaller than 14, so (7, 13, 6) is inserted to the left of (9, 14, 8). Other vectors are inserted similarly.

The search process in a k -d tree is similar to the insertion process. For example, suppose we want to find the vector (17, 3, 15) in the 3-D tree of Fig. 8. Starting from the root, we go right as 17 is larger than 10. We then go left as 3 is smaller than 9 at level 2. At level 3, we go right as 15 is greater than 14. At level 4, we go right again as 17 is greater than 10 and we find the vector (17, 3, 15). We will discuss search for range queries later.

Deletion in a k -d tree can be complicated. We must find the node to be deleted first using the above search process. If the node is the leaf, we can simply delete it and set the pointer originally pointing to it to nil. We are then done. However, the deletion process is more complicated when the node is not a leaf node. We will not discuss it further here. An alternative approach is to mark the nodes to be deleted and leave the tree unchanged. When a sufficient number of nodes have been marked, we rebuild the tree for the unmarked nodes.

Range queries can be implemented relatively easily in a k -d tree. Basically we need to find all feature vectors with each dimension within a certain range. For example, in a 3-D tree, we may need to find all feature vectors (x, y, z) meeting the following conditions:

$$\begin{aligned} x_1 &\leq x \leq x_2 \\ y_1 &\leq y \leq y_2 \\ z_1 &\leq z \leq z_2. \end{aligned}$$

Only a small part of the tree needs to be looked up to answer the range query. This is because key values decrease on a left branch and increase on a right branch. Therefore, only nodes in a part of the tree will meet the requirement.

The main problem with k -d trees is that the tree is not balanced and depends on the order of object insertion. The search complexity is the same as linear search in the worst case.

E. Grid Files

The grid file is an extension of the fixed-grid (FG) access structure [13]. In the FG structure, an n -dimensional space is divided into equal-sized hypercubes. Each hypercube contains zero or more feature vectors. They are accessed by a pointer associated with the hypercube. Each hypercube is indexed based on its location within the n -dimensional space. We use a 2-D space in Fig. 9 to explain the FG structure. The feature space is divided into 16 fixed equal-sized grids. Feature vectors are scattered in these grids. Feature vectors in each grid are accessed via a pointer associated with it. The pointers of these 16 grids can be arranged in a 2-D array, as shown in Fig. 9(b). The indexes of each pointer are determined by the value ranges of its grid and the value range to index mapping on a linear scale. In Fig. 9(b), the value range 0–49 is mapped to index number 0, 50–99 is mapped to 1, and so on.

Insertion and search in a FG structure are easy. For example, if we want to insert a feature vector (80, 70) into the FG in Fig. 9, we can easily determine that the vector belongs to the grid with indexes (1, 1). We then add the feature vector into the list pointed to by $P_{1,1}$. A search for point query can be done similarly. For example, if we want to find feature vector B (40, 125), we determine that it belongs to the grid with indexes (0, 2). We then scan through the list of feature vectors pointed to by $P_{0,2}$ and find the matching vector. For range queries, we need to retrieve all lists of feature vectors pointed to by grids intersecting with the query range rectangle. We then select those vectors within the specified range. For k nearest-neighbor queries, we need to find the grid to which the query vector belongs, and possibly some neighboring grids depending on the location of the query vector. We then calculate the distances between the query and each of feature vectors in the grid(s) and select the k nearest ones.

The FG structure is simple and very effective when feature vectors are evenly distributed in the feature space. However, when feature vectors are unevenly distributed in the feature space, some grids will be empty or almost empty while others

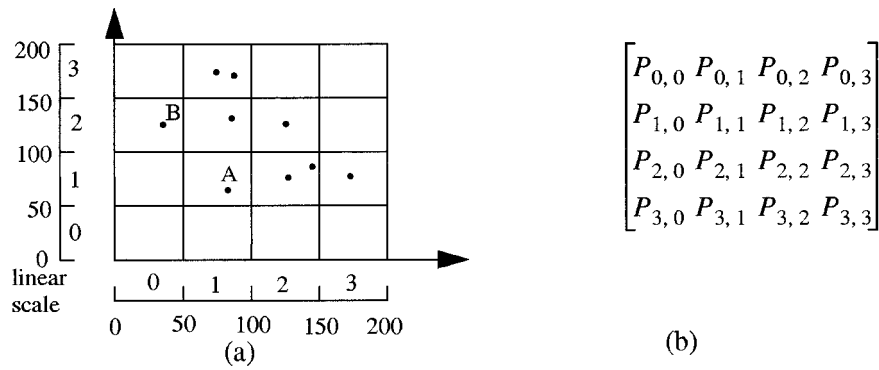


Fig. 9. Example of the fixed-grid structure.

will be too full, leading to lower search efficiency. The grid file is introduced to overcome the problem of the FG structure. The grid file method divides the feature space based on distribution of feature vectors instead of using fixed-size grids. That is, smaller grid sizes will be used for densely populated areas and larger grid sizes for sparsely populated areas. Other concepts are the same as those of the FG structure.

F. R-Tree Family

The R-tree structure and its variants, such as R*- and R⁺-trees, are commonly used for multidimensional data organization [14], [15], [16]. In this section, we first describe the basic R-tree concepts and then briefly describe a number of its variants. In the following discussion, we use the 2-D space as an example. The idea can be easily extended to multidimensional spaces.

1) *An Overview of the R-Tree Structure:* The R-tree is a multidimensional generalization of the B⁺-tree, and hence the tree is height balanced [14]. In a nonleaf node, an entry contains a pointer pointing to a lower level node in the tree and a bounding rectangle covering all the rectangles in the lower nodes in the subtree. The R-tree can be used for either region objects (such as a town boundary in a map) or point data (such as feature vectors in a feature space). When the tree is used for region objects, an entry in a leaf node consists of an object-identifier and a bounding rectangle which bounds the region object. When the tree is used for point data, an entry in a leaf node consists of a pointer pointing to a list of feature vectors and a bounding rectangle which bounds all feature vectors of the list. Each bounding rectangle (of leaf and nonleaf nodes) is represented by the coordinates of its lower-left and upper-right corners. As in the B⁺-tree, each nonleaf node in the R-tree with the exception of the root, must be at least half full. Fig. 10 shows a 2-D feature space with bounding rectangles and the corresponding R-tree.

The R-tree was originally developed for spatial databases such as geographical information systems which deal with region objects. Each bounding rectangle only bounds one region object. Most of the literature on the R-trees does not mention how to handle point data. It may be assumed that point data are a special case of region objects: a region being shrunk into a point. However, this approach to point data handling

is not efficient as each point data or feature vector requires an entry in the leaf nodes. For this reason, we take a different approach to handling point data. Region objects are handled as usual: one bounding rectangle at the leaf nodes bounds only one region object. For point data, one bounding rectangle bounds a list of points or feature vectors. In the following, we describe search, insertion and deletion in an R-tree of region objects and point data separately.

2) *Search, Insertion, and Deletion of Region Objects:* A common query is to find all objects that intersect with the query object. The query object is represented by its minimum bounding rectangle (MBR). Starting from the root, the search algorithm traverses down the subtrees of bounding rectangles that intersect the query rectangle. When a leaf node is reached, the query MBR is tested against each of the object bounding rectangles of the leaf node and those objects that have bounding rectangles which intersect with the query MBR are retrieved.

To insert an object represented by its MBR, the tree is traversed starting from the root. The rectangle that needs the least enlargement to enclose the new object is selected. If more than one rectangle meets the least enlargement criterion, the one with smallest area is selected. The subtree pointed to by the pointer of the selected rectangle is traversed recursively based on the same criteria until a leaf node is reached. A straightforward insertion is made if the leaf node is not full. For each node that is traversed, the bounding rectangle in the parent is readjusted to tightly bound the entries in the node. However, the leaf node needs splitting if it is already full before insertion. A new entry will be added to the parent node of the leaf node and existing entries are adjusted to tightly cover all entries in its leaf nodes if the parent is not full before adding the new entry. Otherwise, the parent node needs to be split and the process may propagate to the root.

The object deletion algorithm uses a similar tree traversal process to that of search: nonleaf nodes that have bounding rectangles which intersect with the MBR of the object to be deleted are traversed. When a leaf node is reached, the MBR to be deleted is compared with each of the entries of the leaf node and the matching entry is deleted. If the deletion of the object causes the leaf node to underflow, the node needs to be deleted and all the remaining entries of that node are reinserted using the insertion algorithm. The deletion of a leaf node may cause further deletion of nodes in the upper levels.

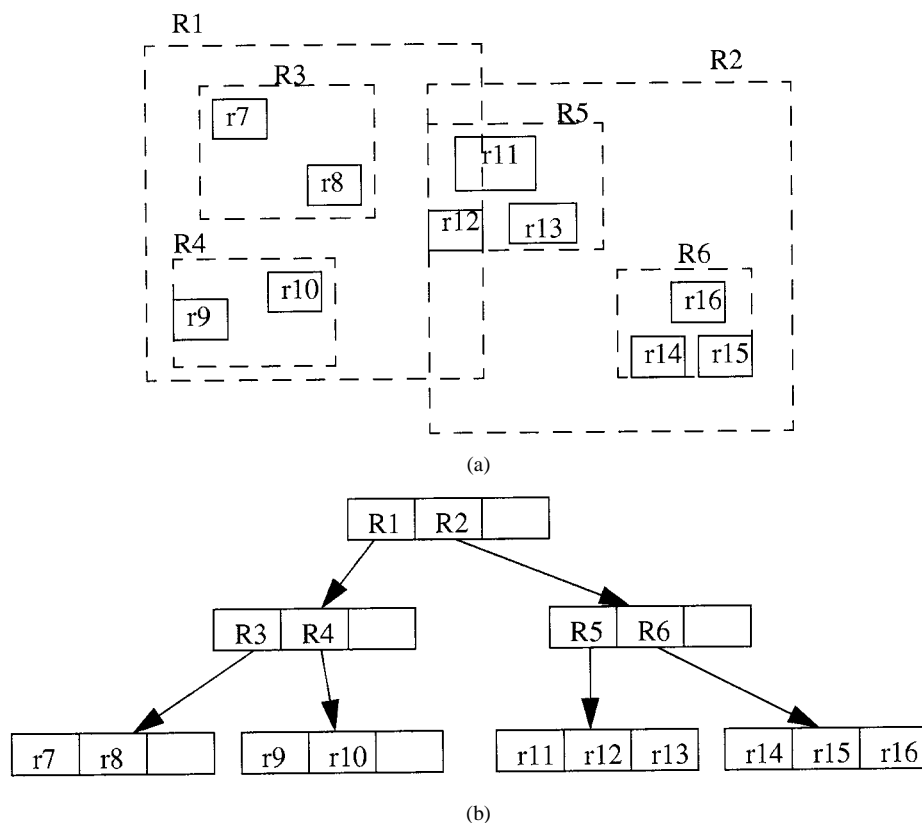


Fig. 10. Structure of an R-tree.

3) *Search, Insertion, and Deletion of Point Data:* We can have point, range, and k nearest-neighbor queries for point data. Point and range queries can be implemented in a similar way to region object search. The only difference is that each entry of the leaf nodes contains a list of points or feature vectors instead of just one object. A k nearest-neighbor query can be implemented as a range query by appropriately estimating ranges on each dimension.

For a point data R-tree, insertion and deletion is done for each point or feature vector instead of each rectangle. Therefore, for point insertion, after reaching an appropriate leaf node, an entry that needs least enlargement and least area is found. The point is then inserted into the list pointed to by the pointer of the entry and the bounding rectangle is adjusted accordingly to bound the new point and existing points if the list is not full. If the list is full, it is split into two lists (and two bounding rectangles) and a new rectangle needs to be inserted. The insertion of a point then becomes the insertion of a region object (an MBR), as discussed in the previous section.

For point deletion, we can delete the point from the selected list directly if the list does not underflow after deletion. Otherwise, the entry has to be deleted and the remaining points of the entry reinserted.

4) *Search Efficiency in the R-Tree:* It has been demonstrated that the search efficiency of an R-tree is largely determined by coverage and overlap [16]. Coverage of a level of an R-tree is the total area of all the rectangles associated with the nodes of that level. Overlap of a level of an R-tree is the total area contained within two or more nodes. Efficient R-tree search demands that both coverage and overlap be minimized. Minimal coverage re-

duces the amount of empty space covered by the nodes. Minimal overlap is more critical than minimal coverage because node overlap leads to the requirement to traverse multiple-paths, slowing down the search. For example, to find r12 in Fig. 10, we have to traverse all nonleaf nodes R1, R2, R3, R4, R5, and R6 because R1 and R2 overlap.

Minimization of both coverage and overlap is crucial to the performance of the R-tree. However, it is impossible to minimize both at the same time. A balanced criterion must be used to achieve the best result.

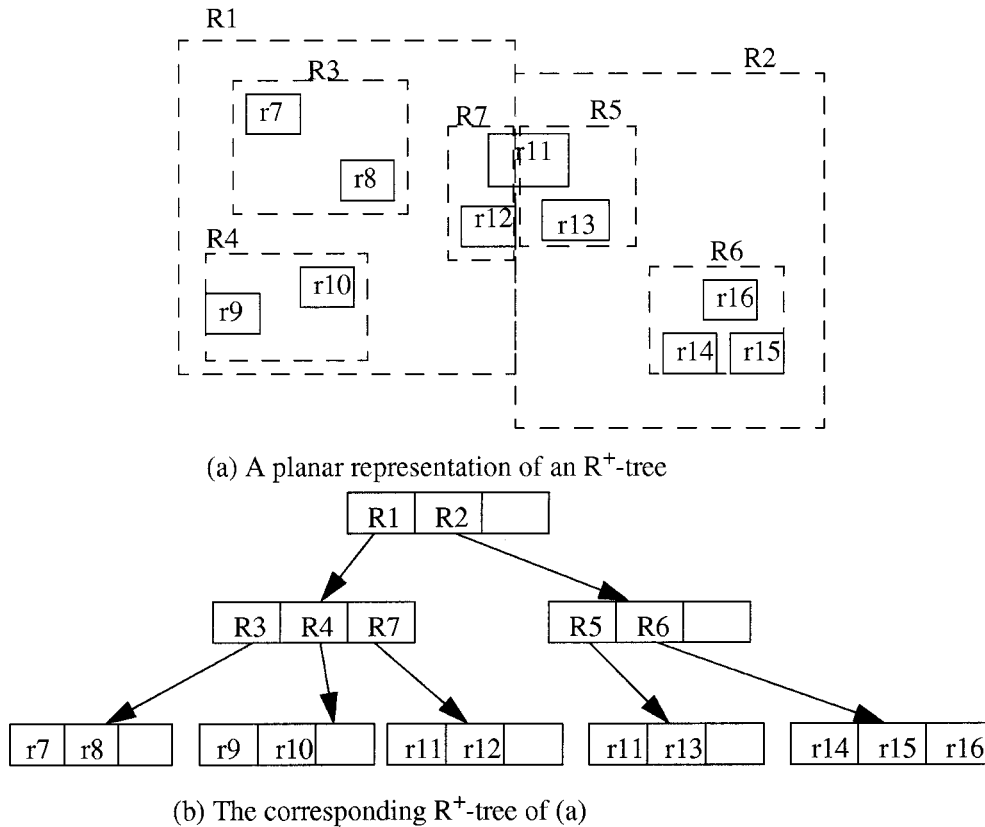
Another factor affecting the R-tree performance is the order of the insertion of data. Different trees may be constructed from the same data set with different insertion orders. Therefore, node reorganization after some insertions and deletions may improve the R-tree performance.

5) *R*-Tree, R+-Tree, and VAMSplit R-Tree:* The R*-tree is an R-tree improved by minimizing coverage [15]. Based on the fact that the clustering of rectangles with low variance in the lengths of the edges tends to reduce the area of the cluster's covering rectangle, the R*-tree ensures to use quadratic covering rectangles in the insertion and splitting algorithms.

The R+-tree was proposed to overcome the overlap problem of internal nodes of the R-tree [16]. The R+-tree differs from the R-tree in the following ways.

- 1) Nodes of an R+-tree are not guaranteed to be at least half full.
- 2) The entries of any internal (nonleaf) node do not overlap.
- 3) Entries of leaf nodes may be repeated.

The repetition of leaf node entries leads to the nonoverlapping of internal nodes. An R+-tree of the feature space in Fig. 10 is

Fig. 11. Structure of an R^+ .

shown in Fig. 11. The main difference from the R-tree in Fig. 10 is that $R1$ and $R2$ now do not overlap, $R7$ is added to bound $r11$ (partially) and $r12$, and $r11$ is repeated in the leaf nodes pointed to by $R5$ and $R7$. The use of disjoint covering rectangles avoids the multiple search paths of the R-tree for point queries and reduces the number of search paths for rectangle search. For example, when searching for $r12$ we do not need to search the subtree pointed to by $R2$.

Another improved R-tree is called the VAMSplit R-tree [17]. The basic idea is that, given a set of multidimensional feature vectors, we split the feature space into rectangles by recursively choosing splits of the data set using the dimension of the maximum Variance and choosing a split that is Approximately the Median.

6) *SS-Tree and SS^+ -Tree*: Many recently proposed multidimensional data structures, including the above-mentioned R^+ -, R^- -, and VAMSplit R-trees, are variants of the R-tree. They differ from the R-tree and from each other mainly in the criteria used for node splitting and branching. Similarity search trees (SS-trees) [18] and SS^+ -trees [19], [20] are two relatively new variants of the R-tree.

The minimum bounding region used in an SS-tree is a sphere. The SS-tree split algorithm finds the dimension with highest variance in feature vector values and chooses the split location to minimize the sum of the variance on each side of the split. To insert a new feature vector, the branch or subtree which has the centroid closest to the new feature vector is chosen.

The SS^+ -tree is similar to the SS-tree. The main difference is that the SS^+ -tree uses a tighter bounding sphere for each node

which is an approximation to the smallest enclosing sphere, and a different split algorithm. For details about SS- and SS^+ -trees, readers are referred to [18]–[20].

G. The Telescopic-Vector-Tree (TV-Tree)

The TV-tree can also be considered a variant of the R-tree [21]. It organizes the data into a hierarchical structure. Objects (feature vectors) are clustered into the leaf nodes of the tree, and the description of their minimum bounding regions is stored in their parent nodes. Parent nodes are recursively grouped as well, until the root is formed. The minimum bounding region can be any shape depending on the application. It may be a hyper-rectangle, cube, or sphere.

The name “TV-tree” came from the concept that the feature vectors can “contract” and “extend” dynamically, resembling a telescope. Thus, it was named the telescopic-vector-tree, or TV-tree. The basic assumption made is that a set of multidimensional feature vectors tends to have same values on some dimensions. Since dimensions with the same values are not useful for discriminating among feature vectors, we need only use dimensions that have different values (called *active dimensions*) to build a tree. The number of active dimensions adapts to the number of objects to be indexed, and to the level of the tree that we are at. This approach effectively reduces the number of dimensions used to organize (index) feature vectors, thus leading to a shallower tree requiring fewer disk accesses.

The main problem with the TV-tree is that it can only be applied to some very specific applications where its assumption

holds. In many general multimedia information retrieval applications, the value of each dimension of feature vectors is a real number and the number of different possible feature vectors is enormous. Thus the assumption that feature vectors will share the same values on many dimensions is not true. For example, in the modest case that each feature vector has ten dimensions and the value on each dimension is an integer ranging from 0 to 255, the total number of possible feature vectors is 256^{10} . The chance that a set of feature vectors will have the same values on many dimensions is extremely low.

IV. DISCUSSION AND SUMMARY

In this paper, we have surveyed a number of application specific filtering techniques and some general multidimensional data structures for efficient multimedia information search and retrieval. They can reduce the required number of distance calculation dramatically. As MB^+ -trees and R-tree family are extensions of B^+ -trees, we need $\log_n N$ levels of nodes, where n is the order of the tree and N is the total number items in the collection or database. At each level, we need n distance calculation to calculate distances between the query and each of the item in that node. Thus, we need a total of $n \log_n N$ distance calculations to answer each query. For example, when $n = 10$, and $N = 100\,000$, a total of 50 distance calculations are required when one of these trees are used instead of 100 000 calculations when linear search is used, assuming all candidate items for a query are located in one leaf node.

k -d trees are not balanced and dependent on the order of insertion of items. Therefore, they are more suitable for applications where items in database are stable. In addition, we need to work out value ranges for each dimension for a given query before we can traverse the tree. For grid files, the number of calculation is equal to N divided by the number of grids.

The above are theoretical computation complexity of different data structures. The search computation complexity of almost all data structures increases exponentially with the number of feature vector dimensions. Thus, the number of dimensions of the feature vectors should be chosen to be as low as possible. In addition, the efficiency and suitability of different data structures vary with distributions of feature vectors. So far, no comprehensive performance comparison has been made among different data structures. Reported performance of different data structures has been obtained on different (usually small) sets of test data. The performance criterion used was usually the number of disk access instead of a combination of a number of criteria. As more and more main memory becomes available on computers, the whole data structure may be brought into the main memory and the number of disk accesses will not be a valid criterion. The number of required operations becomes more important. Nevertheless, we can get some indication of the performance of different data structures from reported work [17]–[19], [21]. White and Jain [17] concluded that the VAMSplit R-tree provides better overall performance than the R^* -tree, SS-tree, and optimized k -d tree variants.

Another challenge in multidimensional data structure research is that k nearest-neighbors may be located in different

leaf nodes, i.e., multiple paths may need to be traversed to find all k nearest-neighbors. Guidelines are required regarding how to determine and minimize the required paths in order to speed up the search process.

In practice, a number of filtering techniques and data structures should be used in combination in different stages of search to speed up the whole search process. For example, we can use classification to divide the entire multimedia database into a number of different classes. Within each class, feature vectors are organized using a certain multidimensional data structure. Within each data structure node, we can use a filtering technique (such as triangle-inequality and average color filtering) to eliminate some unnecessary comparisons/calculations.

So far, most research efforts in multimedia information retrieval systems has been concentrated on retrieval effectiveness, but efficiency of these systems will determine their usefulness. More research efforts in multidimensional data structures are needed.

REFERENCES

- [1] A. Sajjanhar and G. Lu, "Indexing 2-D nonoccluded shape for similarity retrieval," in *Proc. SPIE Conf. Applications of Digital Image Processing XX*, vol. 3164, San Diego, CA, 30 July–1 Aug. 1997, pp. 188–197.
- [2] —, "A grid-based shape indexing and retrieval method," *Aust. Comput. J., Special Issue on Multimedia Storage and Archiving Systems*, vol. 29, pp. 131–140, Nov. 1997.
- [3] A. Berman and L. G. Shapiro, "Efficient retrieval with multiple distance measures," in *Proc. Conf. Storage and Retrieval for Image and Video Databases V*, vol. 3022, San Jose, CA, Feb. 13–14, 1997, pp. 12–21.
- [4] M. Stricker and A. Dimai, "Color indexing with weak spatial constraints," in *Proc. Conf. Storage and Retrieval for Image and Video Databases IV*, vol. 2670, San Jose, CA, Feb. 1–2, 1996, pp. 29–40.
- [5] R. T. Ng and D. Tam, "Analysis of multilevel color histograms," in *Proc. Conf. Storage and Retrieval for Image and Video Databases V*, vol. 3022, San Jose, CA, Feb. 13–14, 1997, pp. 22–34.
- [6] C. Faloutsos, *Searching Multimedia Databases by Content*. Norwell, MA: Kluwer, 1996.
- [7] C. Faloutsos *et al.*, "Efficient and effective querying by image content," *J. Intell. Inf. Syst.*, vol. 3, pp. 231–262, 1994.
- [8] P. W. Foltz and S. T. Dumais, "Personalized information delivery: An analysis of information filtering methods," *Commun. ACM*, vol. 35, pp. 51–60, Dec. 1992.
- [9] R. Elmasri and A. B. Navathe, *Fundamentals of Database Systems*, 2nd ed. Redwood City, CA: Benjamin Cummings, 1994.
- [10] A. Vellaikal and C.-C. J. Kuo, "Hierarchical clustering techniques for image database organization and summarization," in *Proc. SPIE Conf. Multimedia Storage and Archiving System III*, vol. 3527, Boston, MA, Nov. 2–4, 1998, pp. 68–79.
- [11] S. Dao, Q. Yang, and A. Vellaikal, "MB⁺-tree: An index structure for content-based retrieval," in *Chapter 11 of Multimedia Database Systems: Design and Implementation Strategies*, K. C. Nwosu, B. Thuraisingham, and P. B. Berra, Eds. Norwell, MA: Kluwer, 1996.
- [12] J. L. Bentley, "Multi-dimensional binary search trees in database applications," *IEEE Trans. Software Eng.*, vol. 4, pp. 333–340, July 1979. SE-5.
- [13] J. Nievergelt, H. Hinterberger, and K. C. Sevcik, "The grid file: An adaptive, symmetric multikey file structure," *ACM Trans. Database Syst.*, vol. 9, Mar. 1984.
- [14] O. Guttman, "R-tree: A dynamic index structure for spatial searching," in *Proc. ACM SIGMOD Int. Conf. Management of Data*, Boston, MA, 1984, pp. 47–57.
- [15] N. Beckmann *et al.*, "The R^{*}-tree: An efficient and robust access method for points and rectangles," in *Proc. ACM SIGMOD Int. Conf. Management of Data*, Atlantic City, NJ, 1990, pp. 322–331.
- [16] T. Sellis, N. Roussopoulos, and C. Faloutsos, "The R⁺-tree: A dynamic index for multi-dimensional objects," in *Proc. 13th Conf. Very Large Databases*, Brighton, U.K., Sept. 1987, pp. 507–518.
- [17] D. A. White and R. Jain, "Similarity indexing: Algorithms and performance," in *Proc. Conf. Storage and Retrieval for Image and Video Databases IV*, vol. 2670, San Jose, CA, Feb. 1–2, 1996, pp. 62–75.

- [18] ———, "Similarity indexing with the SS-tree," *Proc. 12th IEEE Int. Conf. Data Engineering*, Feb. 1996.
- [19] R. Kurniawati, J. S. Jin, and J. A. Shepard, "SS⁺-tree: An improved index structure for similarity searches in a high-dimensional feature space," in *Proc. Conf. Storage and Retrieval for Image and Video Databases V*, vol. 3022, San Jose, CA, Feb. 13–14, 1997, pp. 110–120.
- [20] J. S. Jin *et al.*, "Using browsing to improve content-based image retrieval," in *Proc. SPIE Conf. Multimedia Storage and Archiving System III*, vol. 3527, Boston, MA, Nov. 2–4, 1998, pp. 101–109.
- [21] K. I. Lin, H. V. Jagadish, and C. Faloutsos, "The TV-tree: An index structure for high-dimensional data," *VLDB J.*, vol. 3, pp. 517–549, Oct. 1994.
- [22] V. S. Subrahmanian, *Principles of Multimedia Database Systems*. San Mateo, CA: Morgan Kaufmann, 1997.
- [23] S. Prabhakar, D. Agrawal, and A. E. Abbadi, "Data clustering for efficient range and similarity searching," in *Proc. SPIE Conf. Multimedia Storage and Archiving System III*, vol. 3527, Boston, MA, Nov. 2–4, 1998, pp. 419–430.
- [24] A. Yoshitaka and T. Ichikawa, "A survey on content-based retrieval for multimedia databases," *IEEE Trans. Knowledge Data Eng.*, vol. 11, Jan./Feb. 1999.
- [25] Y. A. Aslandogan and C. T. Yu, "Techniques and systems for image and video retrieval," *IEEE Trans. Knowledge Data Eng.*, vol. 11, Jan./Feb. 1999.
- [26] W. B. Frakes and R. Baeza-Yates, Eds., *Information Retrieval: Data structures and Algorithms*. Englewood Cliffs, NJ: Prentice-Hall, 1992.
- [27] G. Salton, *Automatic Text Processing—The Transformation, Analysis, and Retrieval of Information by Computers*. Reading, MA: Addison-Wesley, 1989.
- [28] A. Ghias *et al.*, "Query by humming—Musical information retrieval in an audio database," presented at the Proc. ACM Multimedia, San Francisco, CA, Nov. 5–9, 1995.
- [29] E. Wold *et al.*, "Content-based classification, search, and retrieval of audio," *IEEE Multimedia*, pp. 27–36, 1996.
- [30] M. Flickner *et al.*, "Query by image and video content: The QBIC system," *Computer*, pp. 23–32, Sept. 1995.
- [31] W. Niblack *et al.*, "Updates to the QBIC system," in *Proc. Conf. Storage and Retrieval for Image and Video Databases VI, SPIE Proc.*, vol. 3312, San Jose, CA, Jan. 28–30, 1998, pp. 150–161.
- [32] J. R. Bach, "The virage image search engine: An open framework for image management," in *Proc. Conf. Storage and Retrieval for Image and Video Databases IV, SPIE Proc.*, vol. 2670, San Jose, CA, Feb. 1–2, 1996, pp. 76–87.
- [33] J. R. Smith and S.-F. Chang, "Visually searching the web for content," *IEEE Multimedia*, pp. 12–19, July–Sept. 1997.
- [34] H. Zhang, A. Kankanhalli, and S. W. Smoliar, "Automatic partitioning of full-motion video," *Multimedia Syst.*, vol. 1, no. 1, pp. 10–28, 1993.
- [35] G. Lu and A. Sajjanhar, "Region-based shape representation and similarity measure suitable for content-based image retrieval," *Multimedia Syst. J.*, vol. 7, no. 2, pp. 165–174, 1992.
- [36] P. Aigrain, H. Zhang, and D. Petkovic, "Content-based representation and retrieval of visual media: A state-of-the-art review," *J. Multimedia Tools Applicat.*, vol. 3, pp. 179–202, 1996.
- [37] MPEG Home Page. [Online]. Available: <http://www.cselt.it/mpeg/>
- [38] M. S. Lew, "Next-generation web searches for visual content," *IEEE Computer*, vol. 33, pp. 46–53, Nov. 2000.



Guojun Lu (M'96) received the B.Eng. degree from Nanjing Institute of Technology, Nanjing, China, in 1984, and the Ph.D. degree from Loughborough University, Loughborough, U.K., in 1990.

He is currently an Associate Professor at Gippsland School of Computing and Information Technology, Monash University, Churchill, Australia. He has also held positions at Loughborough University, National University of Singapore, and Deakin University. His main research interests are in multimedia communications and multimedia information indexing and retrieval. He has published over 60 technical papers in these areas and has written two books entitled *Communications and Computing for Distributed Multimedia Systems* (Norwood, MA: Artech House, 1996) and *Multimedia Database Management Systems* (Norwood, MA: Artech House, 1999).