Optimizing storage utilization in R-tree dynamic index structure for spatial databases

P.W. Huang a,*, P.L. Lin b, H.Y. Lin a

a Department of Applied Mathematics, National Chung-Hsing University, Taichung 40227, Taiwan, ROC
b Department of Computer Science and Information Management, Providence University, Shalu, Taichung, Taiwan, ROC

Received 1 August 1999; received in revised form 1 December 1999; accepted 8 February 2000

Abstract

Spatial databases have been increasingly and widely used in recent years. The R-tree proposed by Guttman is probably the most popular dynamic index structure for efficiently retrieving objects from a spatial database according to their spatial locations. However, experiments show that only about 70% storage utilization can be achieved in Guttman’s R-tree and its variants. In this paper, we propose a compact R-tree structure which can achieve almost 100% storage utilization. Our experiments also show that the search performance of compact R-trees is very competitive as compared to Guttman’s R-trees. In addition, the overhead cost of building a compact R-tree is much lower than that of a Guttman’s R-tree because the frequency of node splitting is reduced significantly. © 2001 Elsevier Science Inc. All rights reserved.

Keywords: Spatial database; Window query; Spatial object; Compact R-tree; Storage utilization; Search performance

1. Introduction

Spatial database systems (Guting, 1994; Shekhar et al., 1999) are becoming more and more popular in recent years. Typical applications of spatial databases include VLSI circuit layout data in Computer Aided Design (CAD) and cartography in a Geographic Information System (GIS). A common operation of these applications is to search for data objects according to their spatial locations. For example, when an area of interest is found while browsing a digital map or a circuit layout, we may draw a “query window” to inscribe that area and ask the system to retrieve all detailed information about the counties in the map or the components in the circuit layout which are inside or overlapped with the query window. Thus an index structure based on objects’ spatial locations is desirable, but classical one-dimensional database indexing structures are not appropriate to multi-dimensional spatial searching. Structures based on exact matching of values, such as hash tables, are not useful because a range search is required (Guttman, 1984).

In recent years, a number of structures have been proposed for handling multi-dimensional data, and a survey of access methods can be found in Gaede and Gunther (1998). In general, the data of multi-dimensional space can be classified into two types: zero-size objects and non-zero size objects. The K–D–B tree proposed in Robinson (1981) and the G-tree proposed in Kumar (1994) are typical structures for indexing zero-size objects. Grid files proposed in Nievergelt et al. (1984) can handle non-zero size objects. The problem with the grid file structure is that directory maintenance can be very expensive, and if the data are non-uniform, the size of the directory can be very large (Kumar, 1994). The R-tree proposed in Guttman (1984) is probably the most popular structure for indexing non-zero size objects. It has been widely used to index the spatial objects in a large pictorial database such as the GIS application (Papadias and Theodoridis, 1997).

Although the R-tree structure has demonstrated its efficiency in indexing a large number of non-zero size objects in a multi-dimensional space, the storage utilization of this structure is not cost-effective. If \(M\) is the maximum number of entries of a node, every node split in an R-tree will generate \(2M - (M + 1) = M - 1\)
empty entries and node splitting may propagate to cause low storage utilization. The experiments in Beckman et al. (1990) showed that the R-tree structure can only achieve up to 70% storage utilization. In this paper, we present a technique of building compact R-trees. The compact R-trees can achieve almost 100% storage utilization which is a significant improvement over Guttman’s R-tree and all its variants. Our experiments also show that the search performance of compact R-tree is very competitive as compared to Guttman’s R-tree.

The rest of this article is organized as follows. In Section 2, we provide an overview of Guttman’s R-tree and its variants. In Section 3, the method of creating a compact R-tree is described. The algorithms for window query search and deleting a data object from a compact R-tree are also presented. The time complexities for the presented algorithms are analyzed. In Section 4, we present the experimental results about storage utilization and search performance of compact R-trees as well as Guttman’s R-tree for comparison. In Section 5, a real system of applying the compact R-tree index structure to a trademark image database is presented. Finally, conclusions are given in Section 6.

2. Overview of R-tree and its variations

The R-tree is a height-balanced tree with index records in its leaf nodes containing pointers to data objects. For a large number of data objects, the R-tree itself is also large and should be disk-resident so that a node in an R-tree corresponds to a disk block. The R-tree index structure is completely dynamic because inserts and deletes can be intermixed with searches and no periodic reorganization is required.

A spatial database consists of a collection of spatial objects and each object has a unique identifier that can be used to retrieve it. Leaf nodes in an R-tree contain index record entries of the form \((I, \text{obj-id})\), where \(I\) is the minimum bounding rectangle (MBR) and \(\text{obj-id}\) is the identifier of the spatial object being indexed. A non-leaf node contains entries of the form \((I, \text{child-pointer})\), where \(\text{child-pointer}\) is the address of a lower node in the R-tree and \(I\) covers all rectangles in the lower node’s entries. Let \(M\) be the maximum number of entries that will fit in one node and \(m\) be a parameter specifying the minimum number of entries in a node such that \(2 \leq m \leq M/2\). Thus an R-tree satisfies the following properties:

- The root has at least two children unless it is also a leaf node.
- Every non-root node has between \(m\) and \(M\) entries.
- All leaves appear on the same level.

Fig. 1 shows an example of a hierarchy of rectangles and its associated R-tree with \(m = 2\) and \(M = 3\). To find all data objects overlapped with a query window, the search process descends the tree from the root in a manner similar to a B-tree. Since a rectangle on a lower level may be contained by more than one rectangle on a higher level, multiple search paths may be required in order to find a desired data object.

![Fig. 1. Spatial objects are arranged in an R-tree hierarchy.](image)
To insert a new data object $E$ into an R-tree, a search operation must be performed first to identify an appropriate leaf node $L$ to accommodate $E$. If $L$ has room for another entry, we simply add $E$ to node $L$; otherwise, a new node $LL$ is allocated and all existing entries in $L$ plus the new entry $E$ are redistributed into $L$ and $LL$ by either a Quadratic Split or a Linear Split algorithm (Guttman, 1984). Then changes are propagated upward by adjusting the areas of rectangles enclosing $E$. Notice that a node split on a lower level may cause a node overflow onto the next higher level. If node split propagation caused the root to split, a new root is created with the two resulting nodes as its children.

To remove an index entry $E$ from an R-tree, we have to locate the leaf node $L$ containing $E$, then remove $E$ from $L$ and adjust all related covering rectangles. If $L$ has too few entries after eliminating $E$, then $L$ is removed from the R-tree, node elimination is propagated upward as necessary, and all covering rectangles are adjusted along the path to the root. Finally, the remaining entries from the deleted nodes are reinserted into the R-tree.

The concepts of coverage and overlap are important when considering the search performance in R-trees. Coverage of a level of an R-tree is defined as the total area of all the rectangles associated with the nodes of that level. Overlap of a level of an R-tree is defined as the total area contained within two or more nodes of a level. Reducing coverage will decrease the amount of dead space and lower the probability of false hits in search. Minimizing overlap will reduce the chances of multiple paths’ search.

There are several variations to R-trees such as $R^+$-trees (Sellis et al., 1987), $R^*$-trees (Beckman et al., 1990), and packed R-trees (Roussopoulos and Leifker, 1985). They all try to improve the search performance by minimizing “coverage”, “overlap”, or circumference of directory rectangles. $R^*$-trees can be viewed as an extension of K-D-B-trees to cover rectangles instead of points. When a node overflows in an $R^*$-tree, a recursive node splitting may occur. In other words, a split in an overflowed node may cause several splits in its descendant nodes to guarantee that no overlap exists among the entries on each tree level. In the worst case, an insert operation may cause the whole $R^*$-tree to be restructured.

The $R^*$-tree proposed by Beckmann et al. applies a “forced reinsert” strategy based on the concept that different sequences of insertions will build up different R-trees and the search performance of the tree suffers from its old entries. Thus if an entry is inserted into a full node of an $R^*$-tree, a node split does not occur right away. Instead, $p$ (about 30% of $M$) entries of the overflowed node are removed and reinserted into the neighboring nodes on the same tree level.

The packed R-tree is not suitable for a dynamic structure because it assumes that all data objects must be known a priori. Although the search performance of $R^+$-tree and $R^*$-tree is better than that of Guttman’s R-tree, the implementation costs of the former two structures are both higher than that of Guttman’s R-tree structure. Among these three dynamic structures, $R^*$-tree has the highest storage utilization which was 73% according to the experimental result reported in (Beckman et al. (1990)). The readers may refer to (Salzberg, 1994; Lanka and Mays, 1991; Gaede and Gunther, 1998) for more information about multi-dimensional indexing structures.

3. Compact R-tree

The simplicity and low implementation cost of Guttman’s R-tree structure make it more popular than $R^+$-tree and $R^*$-tree structures. However, all the above three dynamic structures suffer from low storage utilization problem. Thus the main goal of our compact R-tree structure is to maximize the storage utilization while retaining the same search performance as in Guttman’s R-tree.

The method of building a compact R-tree is different from that of building a Guttman’s R-tree. In a dynamic environment, the mechanism of maintaining a dynamic index structure is invoked by the insert and delete operations. Inserting index records for new data objects in an R-tree structure is similar to insertion in a B-tree in that new index records are added to the leaves, nodes that overflow are split, and splits may propagate up the tree.

In our method, we may remove an entry from an overflowed node to an under-full sibling node and install the new entry in the evacuated spot. There are three different situations when an insertion operation is performed.

1. Trying to insert a new entry into an under-full node: No split occurs.
2. Trying to insert a new entry into a full node and there is at least one under-full node at the same level: An entry is selected from the set which contains all the entries in the target full node and the entry to be inserted. Add the selected entry to the most appropriate under-full sibling node.
3. Trying to insert a new entry into a full node and all sibling nodes are also full: The target full node splits to accommodate the new entry and the split may propagate upward.

3.1. The algorithms

In the following algorithms, the rectangle part of an index entry is denoted by $E.I$ and the $obj-id$ or
child-pointer part is denoted by $E.p$. The window query search and delete algorithms are similar to the ones in Guttman’s R-trees; however, the insert algorithm is different. For completeness, let us describe in our way all the three algorithms in this paper. The following notations are used in all algorithms for convenience of explanation:

- $T$: The root of the compact R-tree.
- $M$: The maximum number of entries that will fit in one node.
- $N, C$: A node of the compact R-tree.
- $parent(N)$: The parent node of node $N$.
- $O_{MBR}$: The MBR of data object $O$.
- $OID$: The identifier of data object $O$.
- $F, B, E$: An entry of a node.
- $F.I$: The rectangle part associated with entry $F$.
- $E.I$: The pointer part associated with entry $E$.
- $E_N$: The entry corresponding to node $N$ in $parent(N)$.
- $E_N.I$: The rectangle part associated with entry $E_N$ in $parent(N)$. This rectangle tightly encloses all entry rectangles in node $N$.
- $E_N.p$: The pointer part associated with entry $E_N$ in $parent(N)$. This pointer points to node $N$.

### Algorithm: Insert

**Input**: An index record ($O_{MBR}$, $OID$) for new object $O$ to be inserted and the root $T$ of a compact R-tree.

**Output**: The root of the new compact R-tree.

1. $N \leftarrow T$; $E \leftarrow (O_{MBR}, OID)$.
2. While $N$ is a non-leaf node
   (a) Let $F$ be an entry in $N$ such that $F.I$ needs least enlargement to include $O_{MBR}$. If more than one such entries, let $F$ be the one with smallest rectangle.
   (b) $N \leftarrow F.p$.
3. If $N$ has room for another entry /* An under-full leaf node $N$ is found */
   (a) Install $E$ in $N$.
   (b) Adjust the rectangle $E_N.I$ and all the rectangles covering $E_N.I$ along the path from $parent(N)$ to the root.
   (c) Goto 9.
4. If all sibling nodes of $N$ are full, then Goto 6.
5. /* There is room in a sibling node */
   (a) Choose a set of $M$ entries from all the old entries of $N$ plus $E$ so that the area of the tightly enclosing rectangle for the selected $M$ entries is minimum. Let $B$ be the entry that was not selected.
   (b) If $B \neq E$, swap($B, E$) and adjust $E_N.I$ in $parent(N)$ so that it tightly encloses all entry rectangles in $N$.
   (c) Let $E_C$ be an entry in $parent(N)$ such that node $C$ has room for another entry and $E_C.I$ needs least enlargement to include $E.I$. If more than one such nodes, let $C$ be the one with smallest $E_C.I$.
   (d) Install $E$ in node $C$.
   (e) Adjust the rectangle $E_C.I$ and all the rectangles covering $E_C.I$ along the path from $parent(C)$ to the root.
   (f) Goto 9.
6. /* No room for the new entry and node splitting is invoked */Allocate a new node $NN$.
   (a) Adjust $E_N.I$ in $parent(N)$ so that it tightly encloses all rectangles in $N$.
   (b) Create a new entry $S$ with $S.p$ pointing to $NN$ and $S.I$ tightly enclosing all rectangles in $NN$.
   (c) $E \leftarrow S$; $N \leftarrow parent(N)$; Goto 3.
7. If $N$ is not the root
   (a) Adjust $E_N.I$ in $parent(N)$.
   (b) Call Search($N$, $Q$).
8. Create a new root whose children are $N$ and $NN$.
9. Return the root.

### Algorithm: Search

**Input**: The root node $T$ of a compact R-tree and a query window $Q$.

**Output**: All data objects whose MBRs overlap $Q$.

1. If $T$ is not a leaf, inspect all entries in $T$ to find every rectangle $E.I$ which overlaps $Q$. For every such entry $E$
   (a) $N \leftarrow E.p$.
   (b) Call Search($N$, $Q$).
2. If $T$ is a leaf, check all entries $E$ to determine whether $E.I$ overlaps $Q$. If so, $E$ is a qualifying data object.

### Algorithm: Delete

**Input**: The root node $T$ of a compact R-tree and a data object $O$ to be deleted.

**Output**: The root of the new compact R-tree.

1. Let $E \leftarrow (O_{MBR}, OID)$.$\text{Find the leaf node } L \text{ containing } E$. If $E$ is not found, return “Data object not found” and exit.
2. $N \leftarrow L$; $Q \leftarrow \emptyset$.
3. Case of $L$ By definition, the root must have at least two children if the tree is not a single node */
   (a) $N$ is the root with three or more entries: Remove $E$ from $N$. Goto 8.
   (b) $N$ is the root with only two entries: Assume $F$ is the other entry except $E$ in $N$. Let the node pointed by $F.p$ be the new root of the compact R-tree. Goto 8.
4. /* $N$ is not the root */Remove $E$ from $N$.
5. If $N$ still has at least $m$ entries, update $E_N.I$ in $parent(N)$ and adjust all rectangles covering $E_N.I$ along the path from $parent(N)$ to the root. Goto 8.
6. /* $N$ has less than $m$ entries */Move the entries in the leaves of all the subtrees under $N$ to $Q$.
7. $E \leftarrow E_N$; $N \leftarrow parent(N)$; $E.p \leftarrow nil$; Goto 3.
8. (a) If $Q \neq \emptyset$, invoke Insert algorithm to add every data object of $Q$ into the compact R-tree.
   (b) Return the root of the current compact R-tree.
3.2. Time cost analysis

Assume that there are \( N \) data objects indexed by the compact R-tree. Since the storage utilization of a compact R-tree is close to 100% and the maximum capacity of a leaf node is \( M \), there are \( N/M \) leaf nodes and the height of the compact R-tree is \( H = \log_M(N/M) = \log_M N - 1 \). Furthermore, we also assume that the compact R-tree is stored in the disk; therefore, the performance cost will be dominated by the disk accesses (or the number of nodes visited).

For window query processing, in the worst case, we may perform an exhaustive search (i.e., the query window is big enough to cover all data objects). Then the number of nodes visited is \( 1 + M + \cdots + M^{H-1} = O(M^H) = O(N/M^2) \). For point query processing (i.e., searching for a particular data object), the number of nodes visited is proportional to the height of the compact R-tree which is \( k \cdot H = O(\log_M N) \).

To insert a new entry into a compact R-tree, searching for a suitable node to accommodate the new entry is performed first. The number of nodes visited for this operation is \( H \). If the first target leaf node is full, additional \( M - 1 \) sibling nodes may be visited in the worst case to find a node which has space to accommodate the new entry. Once the new entry has been inserted into this node, the new areas of all the rectangles covering the new entry must be updated. \( H \) nodes may be visited for this update operation. Thus the total number of nodes visited in this case is \( H + (M - 1) + H = O(M + \log_M N) \). If the first target leaf node is full and all sibling nodes of the target node are also full, then a node splitting will occur. Node splitting may propagate upward and a new root node may be created in the worst case. Two nodes are visited for each node splitting. Therefore, the total number of nodes visited is \( H + 2H + 1 = O(\log_M N) \).

To delete an index record from a compact R-tree, \( H \) nodes must be visited in order to find the node containing the target index record. Then it takes constant time to delete the target record. If the number of entries in this node becomes \( m - 1 \) (\( m \) is the minimum capacity of a node), then the remaining entries will be reinserted into the compact R-tree. Thus the number of nodes visited is \( H + (m - 1) \cdot H = O(m \cdot \log_M N) \).

4. Experimental results

We ran several experiments comparing Guttman’s R-tree and our compact R-tree. The data objects were randomly generated and stored in data files so that both the Guttman’s method and our method can use the same set of data objects to build R-trees.

<table>
<thead>
<tr>
<th>( M )</th>
<th>( m )</th>
<th>Guttman</th>
<th>Compact</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Number of nodes</td>
<td>Utilization (%)</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>227</td>
<td>79.96</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>181</td>
<td>75.14</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>138</td>
<td>76.93</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>123</td>
<td>72.24</td>
</tr>
</tbody>
</table>

Table 1 shows the simulation result from the first experiment where the total number of data objects (\( N \)) is 500, the maximum number of entries in a node (\( M \)) is from 4 to 7, and the minimum number of entries in a node (\( m \)) is either 2 or 3. Since we have small \( N \), \( M \), and \( m \), such R-trees can be stored in the main memory rather than in the disk. It can be seen from Table 1 that the number of nodes generated by Guttman’s method is always higher than the number of nodes generated by the compact method in each case. In other words, the R-trees generated by our compact method are denser than the R-trees generated by Guttman’s method. The memory utilization of our compact R-trees is very high (from 97.18% to 99.84%) while the memory utilization of Guttman’s R-trees in these cases is from 72.24% to 79.96%.

A similar experiment was conducted by inserting 1000 data objects. The four cases considered were \( M = 4 \) and \( m = 2 \), \( M = 5 \) and \( m = 2 \), \( M = 6 \) and \( m = 3 \), \( M = 7 \) and \( m = 3 \). Table 2 shows the simulation results generated from these four cases. Again, our compact R-trees were smaller and denser than Guttman’s R-trees. For example, when \( M = 6 \) and \( m = 3 \), Guttman’s method generated 299 nodes while our method generated only 201 nodes. The memory utilization in Guttman’s R-trees is from 72.35% to 79.29% while the memory utilization in compact R-trees is from 98.23% to 99.78% in the above four cases.

If there are a large number of data objects in the spatial database, the R-tree itself must be stored in a disk and a node in an R-tree is equivalent to a disk block. A reasonable assumption for disk-based R-trees is \( M = 40 \) and \( m = 20 \). We created five Guttman’s R-trees by inserting 1000, 2000, 3000, 4000, and 5000 data objects, respectively. Similarly, five compact R-trees were built using the same sets of data objects. As
shown in Table 3, the number of nodes generated by our compact method is about one-half of the number of nodes generated by Guttman’s method. In Guttman’s R-trees, the disk utilization is lower than 55% for each of the cases except R-trees, the disk utilization is lower than 55% for each nodes generated by Guttman’s method. In Guttman’s compact method is about one-half of the number of nodes generated by our compact method is much less than the number of nodes generated by Guttman’s method. Node splitting causes significant overhead cost when building R-trees. Thus the overhead cost of building a compact R-tree is lower than that of Guttman’s R-tree because the frequency of node splitting is reduced significantly.

### Table 3
Disk utilization when \( M = 40 \) and \( m = 20 \)

<table>
<thead>
<tr>
<th>Number of data objects</th>
<th>Number of nodes</th>
<th>Utilization (%)</th>
<th>Number of nodes</th>
<th>Utilization (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>38</td>
<td>68.22</td>
<td>26</td>
<td>98.56</td>
</tr>
<tr>
<td>2000</td>
<td>96</td>
<td>54.56</td>
<td>53</td>
<td>96.79</td>
</tr>
<tr>
<td>3000</td>
<td>150</td>
<td>52.48</td>
<td>78</td>
<td>96.62</td>
</tr>
<tr>
<td>4000</td>
<td>204</td>
<td>51.51</td>
<td>104</td>
<td>96.63</td>
</tr>
<tr>
<td>5000</td>
<td>245</td>
<td>53.52</td>
<td>130</td>
<td>98.63</td>
</tr>
</tbody>
</table>

### Table 4
Average number of disk accesses for retrieving one data object \( M = 40, m = 20 \)

<table>
<thead>
<tr>
<th>Number of data objects</th>
<th>Window size</th>
<th>Gutmann</th>
<th>Compact</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>1/900</td>
<td>1.00</td>
<td>1.63</td>
</tr>
<tr>
<td></td>
<td>1/400</td>
<td>0.67</td>
<td>1.10</td>
</tr>
<tr>
<td></td>
<td>1/100</td>
<td>0.54</td>
<td>0.69</td>
</tr>
<tr>
<td>2000</td>
<td>1/900</td>
<td>1.74</td>
<td>1.95</td>
</tr>
<tr>
<td></td>
<td>1/400</td>
<td>1.25</td>
<td>1.41</td>
</tr>
<tr>
<td></td>
<td>1/100</td>
<td>0.76</td>
<td>0.81</td>
</tr>
<tr>
<td>3000</td>
<td>1/900</td>
<td>1.53</td>
<td>1.53</td>
</tr>
<tr>
<td></td>
<td>1/400</td>
<td>1.23</td>
<td>1.16</td>
</tr>
<tr>
<td></td>
<td>1/100</td>
<td>0.73</td>
<td>0.76</td>
</tr>
<tr>
<td>4000</td>
<td>1/900</td>
<td>1.74</td>
<td>1.74</td>
</tr>
<tr>
<td></td>
<td>1/400</td>
<td>1.12</td>
<td>1.08</td>
</tr>
<tr>
<td></td>
<td>1/100</td>
<td>0.97</td>
<td>1.02</td>
</tr>
<tr>
<td>5000</td>
<td>1/900</td>
<td>1.42</td>
<td>1.41</td>
</tr>
<tr>
<td></td>
<td>1/400</td>
<td>0.98</td>
<td>0.98</td>
</tr>
<tr>
<td></td>
<td>1/100</td>
<td>0.71</td>
<td>0.77</td>
</tr>
</tbody>
</table>

5. Application to image database system

Digital images are convenient media for describing and storing spatial/pictorial information in a variety of domains. Large image databases are being created and used in many applications including criminal identification, multimedia encyclopedia, geographic information systems, on-line applications for art and art history, medical image archives, and trademark database (Jain and Vailaya, 1998). Such databases typically consist of a large volume of images which make them so difficult for the user to browse through the entire database. Therefore, an efficient and automatic method is required to index and retrieve images from the database. Traditional methods use caption and keywords as textual features to annotate and retrieve images. However, as the size of the image database grows, the use of keywords becomes inadequate and complex to represent the content of the image. Besides, using keywords to describe images may be too subjective and labor-intensive.

Using multi-dimensional indexing techniques for image retrieval was discussed in (Rui and Huang, 1999). In this section, we present a prototype image database system which utilizes the compact R-tree as the structure for indexing and retrieving trademark images from an image database. This prototype system demonstrates the applicability of the compact R-tree proposed in this paper to a real and useful application. The database of this prototype system consists of 1000 trademark images. Two shape features, edge moment and edge orientation, were extracted from each trademark image. Then these two shape features of an image were converted into two intervals along the \( x \) - and \( y \) - directions, respectively, in a two-dimensional feature space. Consequently, an image in the database is associated with a rectangle in the feature space. By applying the insertion algorithm presented in Section 3.1 to each rectangle one-by-one, a compact R-tree was built as the indices for the database images. Since a query image can also be represented as a rectangle in the feature space, searching for a desired image in the image database is equivalent to searching for the overlapped rectangles in the feature space.
The process of shape feature extraction is briefly described as follows. First, the Canny edge operator (Canny, 1986) is applied to each image to obtain a segmented image. The points in the segmented image are called edge points. The shape features (edge moment and edge orientation) are extracted from the segmented images. The edge moment used in our system is similar to the second-order first invariant moment as defined in (Gonzalez and Woods, 1992). Let $CE$ be the centroid of the segmented image. The edge moment of the $i$th edge point $P_i$ is defined as

$$M_{0i} = \sum_{i=1}^{n} M'_i,$$

where $M'_i = \sqrt{(x_i - x_C)^2 + (y_i - y_C)^2}$, $M = \sum_{i=1}^{n} M'_i$, $n$ is the total number of edge points, $(x_C, y_C)$ the coordinates of $CE$, and $(x_i, y_i)$ are the coordinates of $P_i$. Furthermore, let $CF$ be the centroid of the original image. The edge orientation of the $i$th edge point is defined as the acute angle between two lines $CECF$ and $CEP_i$.

For each segmented image, we can easily obtain two sets of feature data $\{M_1, M_2, \ldots, M_n\}$ and $\{\theta_1, \theta_2, \ldots, \theta_n\}$. The mean and the variance for each set of data can be computed subsequently. Let $\text{Mean}(M_i)$ be the mean of edge moment, $\text{Var}(M_i)$ be the variance of edge moment, $\text{Mean}(\theta_i)$ be the mean of edge orientation, and $\text{Var}(\theta_i)$ be the variance of edge orientation for a database image $I_j$. Then $I_j$ can be converted into a rectangle $(x_j, y_j, x'_j, y'_j)$ in a two-dimensional feature space where $(x_j, y_j)$ the coordinates of the bottom-left corner point, $(x'_j, y'_j)$ the coordinates of the top-right corner point of the rectangle with $x_j = \text{Mean}(\theta_i) - \text{Var}(\theta_i)$, $y_j = \text{Mean}(M_i) - \text{Var}(M_i)$, $x'_j = \text{Mean}(\theta_i) + \text{Var}(\theta_i)$, and $y'_j = \text{Mean}(M_i) + \text{Var}(M_i)$.
$y_j = \text{Mean}(M_j) + \text{Var}(M_j)$. Then we inserted the rectangle associated with each image into the compact R-tree one-by-one for all trademark images.

Some database images (Pic 1 to Pic 16) and two query images (Q1, Q2) are shown in Fig. 2. The shape features extracted from the images and the rectangles in the feature space for the above 16 database images and two query images are listed in Table 5. Notice that the mean of the edge moment for each segmented image is always 1 because it is normalized; therefore, Mean($M_j$) is not listed in this table. In order to clearly display the rectangles in the feature space, we scaled all rectangles equally and shifted them to some appropriate positions. All 18 rectangles in the feature space are also shown in Fig. 3. The dotted boxes Q1 and Q2 represent the two query images. It can be seen that rectangle Q1 overlaps

<table>
<thead>
<tr>
<th>Pic. No. ($i$)</th>
<th>Edge orientation (°)</th>
<th>Edge moment (in pixels)</th>
<th>Feature rectangle ($FR_i$)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mean($\theta$)</td>
<td>Var($\theta$)</td>
<td>Var($M_j$)</td>
</tr>
<tr>
<td>1</td>
<td>44.07°</td>
<td>25.81°</td>
<td>255.29</td>
</tr>
<tr>
<td>2</td>
<td>43.81°</td>
<td>27.92°</td>
<td>206.31</td>
</tr>
<tr>
<td>3</td>
<td>44.82°</td>
<td>26.02°</td>
<td>249.56</td>
</tr>
<tr>
<td>4</td>
<td>46.01°</td>
<td>25.89°</td>
<td>226.17</td>
</tr>
<tr>
<td>5</td>
<td>53.83°</td>
<td>27.29°</td>
<td>173.84</td>
</tr>
<tr>
<td>6</td>
<td>45.22°</td>
<td>26.03°</td>
<td>234.23</td>
</tr>
<tr>
<td>7</td>
<td>43.83°</td>
<td>26.04°</td>
<td>220.36</td>
</tr>
<tr>
<td>8</td>
<td>42.78°</td>
<td>26.45°</td>
<td>192.88</td>
</tr>
<tr>
<td>9</td>
<td>47.23°</td>
<td>25.81°</td>
<td>195.96</td>
</tr>
<tr>
<td>10</td>
<td>40.49°</td>
<td>26.36°</td>
<td>278.68</td>
</tr>
<tr>
<td>11</td>
<td>43.49°</td>
<td>26.22°</td>
<td>194.78</td>
</tr>
<tr>
<td>12</td>
<td>46.21°</td>
<td>27.15°</td>
<td>210.83</td>
</tr>
<tr>
<td>13</td>
<td>44.53°</td>
<td>25.86°</td>
<td>176.98</td>
</tr>
<tr>
<td>14</td>
<td>44.52°</td>
<td>25.99°</td>
<td>198.94</td>
</tr>
<tr>
<td>15</td>
<td>52.60°</td>
<td>26.37°</td>
<td>269.20</td>
</tr>
<tr>
<td>Q1</td>
<td>45.21°</td>
<td>25.95°</td>
<td>176.36</td>
</tr>
<tr>
<td>Q2</td>
<td>52.56°</td>
<td>26.38°</td>
<td>279.38</td>
</tr>
</tbody>
</table>

Fig. 3. The feature rectangles in the feature space associated with the images in Fig. 2.
with rectangles 6 and 14 whereas rectangle Q2 overlaps with rectangles 11 and 16. Here, the overlap between two rectangles is a fuzzy measure. Let Area\( (A) \) and Area\( (B) \) be the areas of two rectangles \( A \) and \( B \), respectively. We say that \( A \) and \( B \) are overlapped if 
\[
\frac{\text{Area}(A \cap B)}{\text{Area}(\text{MBR}(A \cup B))} \geq \tau,
\]
where \( \tau \) is a predefined threshold. Therefore, Pic 6 and Pic 14 are retrieved if Q1 is submitted as the query image. On the other hand, if Q2 is the query image, then Pic 11 and Pic 16 are retrieved as the result. In our prototype system, there are 1000 trademark images in the database. We took each database image as the query image and performed sequential search as well as compact R-tree search. We stored the compact R-tree in the main memory because the database is small. The system took 2.09 ticks to complete the sequential search and 0.32 ticks to complete the compact R-tree search per query image on an average. The system was implemented on a Pentium II 200 Personal Computer with 1000 ticks per second. Our prototype system demonstrated that compact R-tree can be effectively and efficiently used as the index structure for image databases.

6. Conclusions

This paper presented a technique for optimizing the storage utilization in R-trees which can be used as a dynamic index structure for spatial databases. The Guttman’s original R-tree and its variants such as R\(^+\)-tree or R\(^*\)-tree suffer from low storage utilization problem. Generally speaking, all these index structures can achieve up to 70% storage utilization for a large number of data objects. However, the compact R-tree presented in this paper can achieve 98% or 99% storage utilization which is a significant improvement over all variations of R-trees.

Although the search performance of R\(^*\)-tree and R\(^*\)-tree is better than Guttman’s R-tree, they are not as popular as Guttman’s R-tree due to their higher implementation costs. On the contrary, our compact R-tree is as simple as Guttman’s R-tree and very easy to implement. In addition, the search performance of our compact R-tree is very competitive as compared to Guttman’s R-tree. The experimental results demonstrated that our compact R-tree can achieve optimal storage utilization (close to 100%). The overhead cost of building a compact R-tree is also lower than that of building a Guttman’s R-tree because occurrences of node splitting are reduced significantly.

Acknowledgements

We would like to thank Prof. A.K. Jain and Mr. Aditya Vailaya for providing us the digitized trademark images used in our prototype system.

References