Optimization Issues in R-tree Construction

Yannis Theodoridis, Timos Sellis
Knowledge and Database Systems Laboratory
Computer Science Division
Department of Elec. and Comp. Engineering
National Technical University of Athens (NTUA)
Zographou 157 73, Athens, Greece

ABSTRACT

The construction of an efficient tree structure should take into account parameters which can be optimised. Coverage of a node, "dead area" of a node and overlap between nodes are examples of such parameters. Taking these parameters under account is important since minimization of a node coverage leads to more precise searching within the tree, minimization of a node "dead area" reduces the number of unsuccessful hits in the tree, and minimization of the overlap between nodes reduces the number of paths tested in the tree during a search. Existing access methods do not combine all these parameters. In this paper we propose a combination of all of the above criteria into a complex one which would be responsible for the tree construction. We study the algorithms affected and provide preliminary experimental results to indicate the effectiveness of the methods.

1. Introduction

It has been recognized in the past that existing Database Management Systems (DBMSs) do not handle efficiently multi-dimensional data such as boxes, polygons, or even points in a multi-dimensional space. The need to store, retrieve and present multi-dimensional data arises in many applications, such as: Cartography, where maps can be stored and searched electronically, answering efficiently geometric queries, Computer-Aided Design (CAD), for example, VLSI design systems which store many thousands of rectangles, Computer vision and robotics, where geometric objects are among the essential represented entities, and several other areas are continuously surfacing with strong needs for multi-dimensional data handling.

Since database management systems can be used to store one-dimensional data, like integer or real numbers and strings, considerable interest has been developed in using DBMSs to store multi-dimensional data as well. In that sense the DBMS can be the single means for storing and accessing any kind of information required by applications more complex than traditional business applications. However, the underlying structures, data models and query languages are not powerful enough to manipulate more complex data.

*This work has been supported by the ESPRIT II Basic Research Actions Program of the European Community under contract EP 6881 (AMUSING).
In this paper we focus on the problem of designing efficient access methods for multi-dimensional objects. The main operations that have been addressed in the past are:

Point Queries  Given a point in the space, find all objects that contain it
Region Queries  Given a region (query window), find all objects that intersect it.

Of course these operations can be augmented with additional constraints on simple one-dimensional (scalar) data. In addition, operations like insertions, deletions and modifications of objects should be supported in a dynamic environment.

The purpose of this paper is to describe the deficiencies of various access methods that have been suggested in the past and to propose means of overcoming such deficiencies. Section 2 surveys existing indexing methods, classifies them according to certain criteria and presents briefly the main indexing methods that will be examined in the rest of the paper. Section 3 describes the deficiencies of R-tree based access methods giving also an example that presents the effects of some criteria upon the performance of these methods. Then, in sections 4 and 5 we present our suggestions for optimized R-tree construction and report on the results of our study. Finally, we conclude in Section 6 by summarizing our contributions and giving hints for future research in the area of multi-dimensional data indexing structures.

2. Multi-Dimensional Access Methods

In this section we classify and briefly discuss known methods for handling multi-dimensional objects. Our main concern is the storage and retrieval of rectangles in secondary store (disk). Handling more complex objects, such as circles, polygons etc., can be reduced to handling rectangles, by finding the minimum bounding rectangle (MBR) of the given object. In our discussion, we shall first examine methods for handling multi-dimensional points, because these suggest many useful ideas applicable to rectangles as well.

2.1. Methods for multi-dimensional points

The most common case of multi-dimensional data that has been studied in the past is the case of points. The main idea is to divide the whole space into disjoint sub-regions, usually in such a way that each sub-region contains no more than $C$ points. $C$ is usually 1 if the data is stored in core; in case secondary storage is used, $C$ is equal to the capacity of a disk page, that is the number of data records the page can hold.

Insertions of new points may result in further partitioning of a region, known as a split. The split is performed by introducing one (or more) hyperplanes that partition a region further, into disjoint sub-regions. The following attributes of the split help to classify the known methods:
Position
The position of the splitting hyperplane is pre-determined, e.g., it cuts the region in half exactly, as the grid file does [NIEV84]. We shall call these methods fixed. The opposite is to let the data points determine the position of the hyperplane, as, e.g., the k-d trees [BENT75] or the K-D-B-trees [ROBI81] do. We shall call these methods adaptable.

Dimensionality
The split is done with only one hyperplane (1-d cut), as in the k-d trees. The opposite is to split in all k dimensions, with k hyperplanes (k-d cut), as the quad-trees [FINK74] and oct-trees do.

Locality
The splitting hyperplane splits not only the affected region, but all the regions in this direction as well, like the grid file does. We shall call these methods grid methods. The opposite is to restrict the splitting hyperplane to extend solely inside the region to be split. These methods will be referred to as brickwall methods. The brickwall methods usually do a hierarchical decomposition of the space, requiring a tree structure. The grid methods use a multi-dimensional array.

The usefulness of the above classification is twofold: For one, it creates a general framework that puts all the known methods on the map. The second reason is that it allows the design of new methods, by choosing the position, dimensionality and locality of the split, which might be suitable for a given application. Table 1 illustrates some of the most well-known methods and their attributes according to the above classification.

<table>
<thead>
<tr>
<th>Method</th>
<th>Position</th>
<th>Dimensionality</th>
<th>Locality</th>
</tr>
</thead>
<tbody>
<tr>
<td>point quad-tree</td>
<td>adaptable</td>
<td>k-d</td>
<td>brickwall</td>
</tr>
<tr>
<td>k-d tree</td>
<td>adaptable</td>
<td>1-d</td>
<td>brickwall</td>
</tr>
<tr>
<td>grid file</td>
<td>fixed</td>
<td>1-d</td>
<td>grid</td>
</tr>
<tr>
<td>K-D-B-tree</td>
<td>adaptable</td>
<td>1-d</td>
<td>brickwall</td>
</tr>
</tbody>
</table>

Table 1: Illustration of the classification.

Notice that methods based on binary trees or quad-trees cannot be easily extended to work in secondary storage based systems. The reason is that, since a disk page can hold of the order of 50 pointers, trees with nodes of large fanout are more appropriate; trees with two- or four-way nodes usually result in many (expensive) page faults.

2.2 Methods for rectangles
We now present a classification and brief discussion of methods for handling rectangles. These methods fall under the following general categories:

1) Methods that transform the rectangles into points in a space of higher dimensionality [NIEV84]. For example, a 2-d rectangle (with sides parallel to the axes) is characterized by four coordinates, and thus it can be considered as a point in a 4-d space. Therefore, one of the previously mentioned methods for storing points can be chosen. Hinrichs and Nievergelt suggested using the grid file, after a rotation of the axes. The rotation is necessary, in order to avoid non-uniform distribution of points, that would lead the grid file to poor performance.

2) Methods that use space filling curves, to map a k-d space onto a 1-d space. Such a method, suitable for a paged environment, has been suggested, among others, by Orenstein [OREN86]. The idea is to transform k-dimensional objects to line segments, using the so-called z-transform. This transformation tries to preserve the distance, that is, points that are close in the k-d space are likely to be close in the 1-d transformed space. Improved distance-preserving transformations have been proposed [FALO88], which achieve better clustering of nearby points, by using Gray codes. The original z-transform induces an ordering of the k-d points, which is the very same one that a (k-dimensional) quad-tree uses to scan pixels in a k-dimensional space. The transformation of a rectangle is a set of line segments, each corresponding to a quadrant that the rectangle completely covers.

3) Methods that divide the original space into appropriate sub-regions (overlapping or disjoint). If the regions are disjoint, any of the methods for points that we mentioned above, can be used to decompose the space. The only complication to be handled is that a rectangle may intersect a splitting hyperplane. One solution is to cut the offending rectangle in two pieces and tag the pieces, to indicate that they belong to the same rectangle. Recently, Gunther [GUNT91] suggested a relevant scheme for general polygon data, either convex or concave. He suggests that the splitting hyperplanes can be of arbitrary orientation (not necessarily parallel to the axes). The first who proposed the use of overlapping sub-regions was Guttman with his R-trees [GUTT84]. R-trees are an extension of B-trees for multi-dimensional objects that are either points or regions. Like B-trees, they are balanced (all leaf nodes appear on the same level, which is a desirable feature) and guarantee that the space utilization is at least 50%. However, if R-trees are built using the dynamic insertion algorithms, the structure may cause excessive space overlap and dead-space in the nodes, which in turn results to bad performance. A packing technique proposed in [ROUS85] alleviates this problem for relatively static databases of points. However, for update-intensive spatial databases, packing cannot be applied on every single insertion. The R*-tree [SELL87] and the R*-tree [BECK90] methods have been proposed to address the problem of performance degradation caused by the
overlapping regions or excessive dead-space respectively. However, all of the above methods try to address the problem in a rather ad-hoc way. In a later section we propose a general approach to solving the search performance problem of multi-dimensional tree structures by trying to parameterize them and fixing the values of the appropriate parameters in an algorithmic way. Before moving to the main results of the paper we give a brief description of the methods.

2.3. R-tree-based Access Methods

As mentioned above, R-trees are a direct extension of B-trees in k-dimensions. The data structure is a height-balanced tree which consists of intermediate and leaf nodes. The descriptions of data objects are stored in leaf nodes and intermediate nodes are built by grouping rectangles at the lower level. Data objects can be overlapping, covering each other, or completely disjoint; no assumption is made about their properties. As mentioned in section 2, the Minimum Bounding Rectangles (MBRs) of the actual data objects are assumed to be stored in the leaves of the tree. Each intermediate node is associated with some rectangle which completely encloses all rectangles that correspond to lower level nodes. Figure 1 shows an example set of data rectangles and Figure 2 the corresponding R-tree built on these rectangles (assuming a branching factor of 4).

Figure 1: Some rectangles, organized to form an R-tree
Considering the performance of R-tree searching, the concepts of *coverage* and *overlap* are important. 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 overlapping nodes. Obviously, efficient R-tree searching demands that both overlap and coverage be minimized. Minimal coverage reduces the amount of *dead space* (i.e. empty space) covered by the nodes. Minimal overlap is even more critical than minimal coverage. For a search window falling in the area of *k* overlapping nodes at level *h-l*, *h* being the height of the tree, in the worst case, *k* paths to the leaf nodes have to be followed (i.e. one from each of the overlapping nodes), therefore slowing down the search. For example, for the search window *W* shown in Figure 3, both subtrees rooted at nodes *A* and *B* must be searched although only the latter will return a qualifying rectangle. The cost of such an operation would be one page access for the root and two additional page accesses to check the rectangles stored in *A* and *B*.

---

**Figure 2: R-tree for the rectangles of Figure 1**

**Figure 3: An example of a bad search window**
It has been shown, that zero overlap and coverage is only achievable for data points that are known in advance and, that using a packing technique for R-trees, search is dramatically improved [ROUS85]. In the same paper it is shown that zero overlap is in general not attainable for region data objects. However, if we allow partitions to *split* rectangles then zero overlap among intermediate node entries can be achieved. This is the main idea behind the $R^+$-tree structure. Figure 4 indicates a different grouping of the rectangles of Figure 1 and Figure 5 shows the corresponding $R^+$-tree.

Figure 4: The rectangles of Figure 1 grouped to form an $R^+$-tree

Figure 5: The $R^+$-tree built according to Figure 4

Notice that rectangle G has been split into two sub-rectangles, the first contained in node A and the second in P. Avoiding overlap is achieved at the expense of space which increases the height of the tree. However, because the space increase is logarithmically distributed over the tree, the indirect increment of the height is more than offset by the benefit gained by avoiding multiple paths. For example, if we consider again the cost for a search operation based on the
window W of Figure 3 we notice that only the root of the tree and node P need be accessed, thus saving us one out of three page accesses.

Finally, the R*-tree [BECK90] is a variation of the R-tree which unlike the R+-tree does not change the properties of the structure, but rather has a specific way to organize rectangles into nodes so that overall performance is improved (as described in section 3.1.).

After this short review of the structures the following sections address various performance problems and suggest solutions.

3. Optimization of Tree Performance

The correct (performance-wise) insertion of rectangles into a tree is of major importance. When inserting a new rectangle (i.e. MBR) the algorithm first tries to find a subtree to direct the new rectangle into (ChooseSubtree). The original R-tree method considers the coverage of a node as the main criterion to be minimized. On the other hand, overlap is the exclusive criterion for R+-tree while R*-tree minimizes coverage (for intermediate nodes) and overlap (for terminal nodes). Based on the experience with the above methods, a combination of the above criteria (and others) should be taken into account for an efficient Insert algorithm.

The ChooseSubtree algorithm performance is affected by the following factors:

- **coverage**: minimization of a node area leads to more precise searching.
- **dead-area**: minimization of dead area in a node leads to faster searching, and
- **overlap**: minimization of overlap between branches of a node leads to less false drops.

A combination of the above criteria could provide an optimized ChooseSubtree algorithm.

Insertion of a branch into a node (terminal or intermediate) may cause node overflow. In that case, the Split algorithm is responsible for the splitting of the node that overflows in two nodes. R-tree method selects the two branches that are the most distant ones. These two branches are the first entries of the two nodes. In a second step, the remaining entries are assigned to one of the two nodes using as criterion the minimum area required to cover the new entry. On the other hand, the R*-tree and the R*-tree methods sort the members of the node that overflows and, then, the partition line between the two nodes is decided according to some criteria like overlap (R+-tree and R*-tree) and coverage (R*-tree). Other factors that affect the performance of the Split algorithm performance are:
- **number of groups**: The R-tree and its variants keep the number of groups after splitting constant, and equal to 2. If this number is variable, the performance of the Insert routine could be improved.

- **level**: splitting of a node should take into account the level of the node in the tree structure. Importance of the coverage of a node, for example, is higher when the level of the node is high.

- **distribution**: distribution of data (e.g. uniform, gaussian etc.) affects the tree structure and performance.

All the above factors should be combined for an efficient Split algorithm.

In section 4 we will present our suggestions for an efficient Insert routine. Before this presentation we provide an example in order to display the insertion steps for the R-tree method and its variants (R+-tree, R*-tree).

**3.1. An example**

We give an example to illustrate the Insertion technique of each of the three methods. We suppose that node capacity is 6 and the node of Figure 6 overflows.

![Figure 6: Some rectangles and their coordinates](image)

The node that overflows is split into two nodes with 3 or 4 elements each one (we suppose that minimum node capacity \( m=M/2=3 \)). In the following paragraphs we present in detail the split algorithm of each method.

**R-tree**
The two most distinct rectangles (id=1, id=5) are selected to be the first entries of the two groups. The remaining entries are assigned to each node using the criterion of minimum area i.e. each rectangle is assigned to the node that will be least enlarged. The final grouping, shown in Figure 7(a), is the following:

1st group : 1,2,3,6
2nd group : 5,4,7

**R+**-tree
The R+ -tree requires that the two new nodes after the splitting of the original one cover mutually dis-joint areas. The first decision considers the partition line that will decompose the space into two sub-regions. This line is selected after a sweep along each axis and the criterion for the selection is the minimization of a cost value. Since the partition line may cut some of the rectangles, those will have to be split into two smaller rectangles to agree with the disjointness property of the method. In our example this happens for rectangles id=2 and id=7. The final grouping, shown in Figure 7(b), is the following:

1st group : 1,2,6,7
2nd group : 2,3,4,5,7

**R***-tree
The R*-tree uses the following method to find good splits. Along each axis, the entries are first sorted by the lower value, then sorted by the upper value of their rectangles. For each sort several distributions of the entries into two groups are determined. The split axis is the one that minimizes a cost value S (S is equal to the sum of all margin-values of the different distributions). In a second step, along the chosen split axis, the distribution with the minimum overlap-value is selected to be the final one. In our example, the first criterion (margin-values) selects x-axis to be the split axis and, among the different distributions of the x-axis, the following distribution, shown in Figure 7(c), is decided to be the final one according to the second criterion (overlap-value):

1st group: 1,6,7,2
2nd group: 3,4,5

**R**-tree  \[ R^+ \text{-tree} \]  \[ R^* \text{-tree} \]

Figure 7: The rectangles of Figure 6 organized according to the various methods
Some comments are appropriate at this point for each method:

**R-tree**
- Because of the least enlargement criterion, the split algorithm tends to prefer the group with the largest size and the most members. It is obvious that this group will be least enlarged, in most cases. The result is that most of the rectangles are directed to the first group.
- If one group has so few entries that all the rest must be assigned to it, the assignment is done without any control. This fact, usually, causes high overlap between the two nodes.

**R+-tree**
- Because of the disjointness property of the method, downward propagation of the split may be necessary [SELL87].
- The most important problem is that a dead-lock is possible because it is not certain that a partition line will be always found. For example, if each of the rectangles in the node overlaps all others then a partition is not possible.

**R*-tree**
- The algorithm uses two kinds of criteria: the margin-criterion to select a split axis and the overlap-criterion to select a distribution along the split axis. This separation may cause the loss of a "good" distribution if that latter one belongs to the axis that has been rejected. This problem can be avoided by combining the performance criteria together.
- Insertion of abnormal rectangles (e.g. long and narrow ones) may direct the algorithm to a bad split axis selection because such rectangles affect the first criterion (margin-value).

In general terms, the performance of the R*-tree split algorithm is very good and it is due to the presence of two performance criteria (margin-value and overlap-value). In our example, however, the final distribution is not the best one. The following distribution

| 1st group: | 3,7,5,2 |
| 2nd group: | 1,4,6 |

is the one with the best sum of overlap-value and margin-value. Unfortunately, it was rejected by the algorithm because it is based on a split over the rejected y-axis.

As a conclusion, all methods split the node that overflows by first sorting its elements (except the R-tree) and then distributing them into a constant number of groups (equal to 2) according to some criteria. We suggest some improvements to this general idea in the following section.
4. Suggestions for an Efficient Access Method

We suggest that insertion of a new rectangle into the tree structure should take into account all the criteria that affect the tree performance, namely:

- **coverage**: total area of a node.
- **dead-area**: sum of the areas in the node that do not belong to any branch of the node.
- **overlap**: sum of overlap areas among the branches of a node.

In the following we formulate such algorithms.

4.1. ChooseSubtree Algorithm

When a new rectangle is inserted into the tree structure, the ChooseSubtree algorithm decides the node which the rectangle will be directed to. We select the node by using a function $F(\text{coverage}, \text{dead-area}, \text{overlap})$ which computes the above criteria for each node. The node with the minimum $F$ is the selected one.

Algorithm ChooseSubtree. Select a leaf node in which to place a new index entry $E$.

CS1 Set $N$ to be the root.

CS2 If $N$ is a leaf, return $N$.

else

Choose the entry in $N$ whose rectangle $R$ minimizes the function $F(N, R)$.

endif

CS3 Set $N$ to be the childnode pointed to by the childpointer of the chosen entry and repeat from CS2.

An example of such an $F$ function could be

Function $F(\text{node } N, \text{ rectangle } R)$

begin

    parameter1 = coverage($R$);

end

parameter2 = dead_area(R);
parameter3 = overlap(R,N);
w1 = w2 = w3 = 1/3; /* weights of parameters */
F = w1 * parameter1 + w2 * parameter2 + w3 * parameter3;
return (F);
end

4.2. Split Algorithm

If the node that was selected by the ChooseSubtree algorithm is full then it overflows with the insertion of a new rectangle. In that case, the Split algorithm is responsible for the distribution of the entries in two nodes. We propose the following Split algorithm:

Algorithm Split. Divide a set of M+1 index entries into two groups.
S1 [Sort entries and pick first entry for each group]
   Sort entries of the node according to some value.
   Set the S first and the S last sorted rectangles as the first entries of the two groups.
S2 [Check if done]
   If all entries have been assigned, stop. If one group has so few entries that all the rest must be assigned to it in order for it to have the minimum number \( m \), assign them and stop.
S3 [Select entry to assign]
   Invoke algorithm PickNext to choose the next entry to assign. Add it to the group with the minimum \( F \) factor. Resolve ties by adding the entry to the group with fewer entries, then to either. Repeat from S2.

Algorithm PickNext
PN1 [Choose the next entry to assign]
   If the last entry was assigned to the first group then the next entry is the last of the remaining ones.
   If the last entry was assigned to the last group then the next entry is the first of the remaining ones.

The algorithm first sorts the entries of the node that overflows according to some value. Possible values are low or high coordinates along some axis, Hilbert or Peano values of some rectangle point e.g. center of rectangle etc. In the following section we test several sorting types. After the sorting procedure, the S first entries and the S last entries are the leaders of
the two groups, respectively. It is obvious that these entries are too distant to belong to the same group. For the remaining entries the selection of the group that they are assigned to is similar to the selection of the node in the ChooseSubtree algorithm. The F factor is computed for each group and the selected group is the one with the minimum F value. To avoid sending the entries to the first group only we use the PickNext technique.

5. Performance Comparison

A performance comparison of the implementations (in C) of the three original methods and our modifications to them as described above, was performed on DECstations 5000/200 running ULTRIX 4.3. To compare the performance of these structures we selected four data files containing 500, 1000, 2000 and 5000 2-dimensional rectangles. For each of these files we performed two operations:

- Insert: Insertion of rectangles in order to build a new tree.
- Search: Searching the tree for entries that overlap with query rectangle.

In the following we report on the results of our study.

5.1. Tests on R-tree method

- **Test1:** The original R-tree method selects the two most distant rectangles to be the first entries of the groups. For each remaining rectangle it assigns it to a group by using the least enlargement criterion. If one group has so few entries that all the rest must be assigned to it, it assigns them to that group. This assignment is a rather ad-hoc decision. In our modification, we first sort the rectangles according to some value (e.g. mid-x value, mid-y value, Peano-value of center etc.) and, after the sorting procedure, we assign the entries to the groups.

- **Test2:** We modify the implementation of Test1 by selecting the rectangles for assignment into the groups according to the PickNext routine that was presented in section 4.2.

The following tables report our results.

<table>
<thead>
<tr>
<th>INSERT OPERATION</th>
<th>ORIGINAL R-Tree</th>
<th>TEST1 (mid-x)</th>
<th>TEST1 (Peano)</th>
<th>TEST2 (mid-x)</th>
<th>TEST2 (Peano)</th>
</tr>
</thead>
<tbody>
<tr>
<td>pages touched per insert</td>
<td>2.17</td>
<td>2.26</td>
<td>2.22</td>
<td>2.20</td>
<td>2.21</td>
</tr>
<tr>
<td>cpu time per insert</td>
<td>0.004</td>
<td>0.004</td>
<td>0.004</td>
<td>0.004</td>
<td>0.005</td>
</tr>
<tr>
<td>space utilization</td>
<td>70 %</td>
<td>44 %</td>
<td>58 %</td>
<td>60 %</td>
<td>58 %</td>
</tr>
</tbody>
</table>
The conclusion of the tests is that the sorting of the rectangles causes an important improvement (about 40%) to the Search performance. We tested several sorting types e.g. low or high coordinates along some axis, Hilbert or Peano values of some rectangle point like the center of rectangle etc. Peano sorting seems to be the most efficient one because it is, more or less, data independent (i.e. not affected by the shape of the rectangles). The contribution of TEST2 is that the new PickNext routine is necessary for the space utilization to be kept high.

5.2. Tests on R⁺-tree method
- **Test1**: The original R⁺-tree method sorts the rectangles according to the low coordinate and sweeps along some axis to find the best partition line. In our modification, we test an alternative sorting type i.e. Hilbert / Peano value of rectangle centers and avoid sweeping by selecting the projection of the mid-rectangle center on some axis as the partition line. The results are shown in tables 4 and 5.

<table>
<thead>
<tr>
<th>INSERT OPERATION</th>
<th>ORIGINAL R⁺-Tree</th>
<th>TEST1 (Peano)</th>
</tr>
</thead>
<tbody>
<tr>
<td>pages touched per insert</td>
<td>2.53</td>
<td>2.78</td>
</tr>
<tr>
<td>cpu time per insert</td>
<td>0.006</td>
<td>0.006</td>
</tr>
<tr>
<td>space utilization</td>
<td>58 %</td>
<td>39 %</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SEARCH OPERATION</th>
<th>ORIGINAL R⁺-Tree</th>
<th>TEST1 (Peano)</th>
</tr>
</thead>
<tbody>
<tr>
<td>pages touched per search</td>
<td>2.97</td>
<td>4.02</td>
</tr>
<tr>
<td>cpu time per search</td>
<td>0.003</td>
<td>0.004</td>
</tr>
</tbody>
</table>

The results show that the performance of the original method is very high because of the exhaustive plane sweep along each axis. This operation guarantees an efficient tree structure.

5.3. Tests on R*-tree method
• **Test1:** The original R*-tree method separates the performance criteria using the margin criterion to select a split axis and the overlap criterion to select the distribution of rectangles along the split axis. In our modification, we use a combination of the two criteria for both steps.

• **Test2:** The original R*-tree method sorts the rectangles according to the low and high coordinate along each axis. In our modification, we add an alternative sorting type i.e. according to the Hilbert or Peano value of rectangles centers.

The following tables report our results.

<table>
<thead>
<tr>
<th>INSERT OPERATION</th>
<th>ORIGINAL R*-Tree</th>
<th>TEST1 (all criteria)</th>
<th>TEST2 (Peano)</th>
</tr>
</thead>
<tbody>
<tr>
<td>pages touched per insert</td>
<td>2.18</td>
<td>2.18</td>
<td>2.20</td>
</tr>
<tr>
<td>cpu time per insert</td>
<td>0.005</td>
<td>0.005</td>
<td>0.005</td>
</tr>
<tr>
<td>space utilization</td>
<td>69 %</td>
<td>68 %</td>
<td>69 %</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SEARCH OPERATION</th>
<th>ORIGINAL R*-Tree</th>
<th>TEST1 (all criteria)</th>
<th>TEST2 (Peano)</th>
</tr>
</thead>
<tbody>
<tr>
<td>pages touched per search</td>
<td>4.00</td>
<td>3.92</td>
<td>4.05</td>
</tr>
<tr>
<td>cpu time per search</td>
<td>0.004</td>
<td>0.003</td>
<td>0.004</td>
</tr>
</tbody>
</table>

Tables 6 and 7: Tests on R*-tree Split algorithm

The results show that the Search performance of the R*-tree is improved slightly when a combination of criteria is effected while it is not affected by the sorting type.

### 5.4. Performance of the new method

We have implemented the suggestions of section 4 for an efficient access method. Our implementation is affected by several parameters that could be tuned. Examples of these parameters are the following:

- type of **sorting** (Hilbert or Peano value of rectangle center, mid-x value etc.).
- minimum node capacity $m$ ($2 \leq m \leq M/2$).
- number $S$ of distant entries ($1 \leq S \leq m$).
- weights $w_i$ of parameters in the F function used in the ChooseSubtree and Split algorithms.
In order to extract conclusions about the structure performance we assumed the following initial values:\(^1\):

- sorting type = (none)
- \( m = M \times 0.2 \)
- \( S = 1 \)
- \( w_{1CS} = 1, w_{2CS} = 0, w_{3CS} = 0 \) (ChooseSubtree algorithm)
- \( w_{1S} = 1, w_{2S} = 0, w_{3S} = 0 \) (Split algorithm)

and tune each one of the parameters. Figures 8-11 report our main results.

---

\(^1\)The initial values have been selected to emulate the original R-tree structure.
The conclusions of the figures 8-11 are the following:

- Peano sorting is the most efficient one. On the other hand, sorting based on axis projections (e.g. mid-x, mid-y) causes a non-uniform performance.
- Low value of the $m$ parameter results to high performance. This is sensible because the entries of the node that overflows have been already sorted by the Split algorithm.
- Low value of the $S$ parameter results to high performance. On the other hand, the $S$ parameter seems not to affect the performance of the method for large amount of data.
- The presence of the "coverage" criterion in the ChooseSubtree algorithm ($w_{1CS}>0$) and the "overlap" criterion in the Split algorithm ($w_{2S}>0$) are necessary for a good performance. Besides, the "coverage" and "dead-area" criteria in the Split algorithm affect the performance in a similar way\(^2\), which means that one of them could be disabled.

We also compare the performance of our new method (tuned according to the optimal values: sort=Peano, $m=2$, $w_{1CS}$ and $w_{2S}$ enabled) with the performance of the three original methods. The following tables report our results.

<table>
<thead>
<tr>
<th>INSERT OPERATION</th>
<th>ORIGINAL R-Tree</th>
<th>ORIGINAL R*-Tree</th>
<th>ORIGINAL R*-Tree</th>
<th>NEW METHOD</th>
</tr>
</thead>
<tbody>
<tr>
<td>pages touched per insert</td>
<td>2.17</td>
<td>2.53</td>
<td>2.18</td>
<td>2.20</td>
</tr>
<tr>
<td>cpu time per insert</td>
<td>0.004</td>
<td>0.006</td>
<td>0.005</td>
<td>0.004</td>
</tr>
<tr>
<td>space utilization</td>
<td>70 %</td>
<td>58 %</td>
<td>69 %</td>
<td>61 %</td>
</tr>
</tbody>
</table>

\(^2\)This is not always true. The two criteria have equivalent behaviour in our tests because most of the data rectangles are disjoint.
The results show that the Search performance of our preliminary implementation is equivalent to the performance of the R*-tree method while the Insert operation is as fast as the one of the original R-tree method.

Concluding our tests we show a figure that presents the Search performance of all the methods that have been tested together with the new one.

6. Conclusions

The construction of an efficient tree structure should take into account parameters which can be optimised. Coverage of a node, "dead area" of a node and overlap between nodes are examples of such parameters. Taking these parameters under account is important since

- Minimization of a node coverage leads to more precise searching within the tree.
- Minimization of a node "dead area" reduces the number of unsuccessful hits in the tree.
- Minimization of the overlap between nodes reduces the number of paths tested in the tree during a search.
Existing access methods do not combine all these parameters. In this paper we proposed a combination of all of the above criteria into a complex one which would be responsible for the tree construction.

The node splitting algorithm is also one of the most important factors for an efficient tree structure. Every variant of R-trees uses its own technique for node splitting. We studied the correlation of the criteria in order to find an ideal combination of them.

The results of our current study show that a static combination of the parameters improves the performance of the R-tree structure at the level of the most popular R-tree variants (R*-tree and R*-tree).

It is a topic of current work to examine other factors of the split algorithm such as the number of groups, the load-factor, etc. All of these factors should be variable and tuned according to the distribution of data, the level of the tree at which splitting occurs, and other characteristics of the tree structure.

7. References


