CHAPTER 8

STUDY OF THE EFFECTS OF THE BOUNDING SHAPES
OF OBJECTS IN R*-TREE

Multidimensional indexing techniques approximate the data objects in the space using some bounding shapes. The shapes of the bounding structures affect the performance of the indexing method. R*-tree covers the objects of the space using MBRs. The representations of MBRs and computations performed on them are simple and cost effective. But MBRs do not accurately reflect the shape of the objects and hence adversely affect the search process. This is due to empty spaces introduced by the MBRs. Empty spaces are extraneous spaces included to construct the bounding shape around the object. This space is treated as part of the object in every process in which the object is considered. There is scope for using other shapes to cover the objects that would mitigate the empty space problem. This work does a comparative study of the effects of covering the objects using MBRs, MBSs and MBPs.

8.1 Preamble

R*-tree is a multidimensional data partitioning indexing structure, in which the \( d \)-dimensional objects are abstracted as \( d \)-dimensional rectangles in a \( d \)-dimensional space. The partitions that group the objects also follow suit. The sides of these bounding rectangles are \( d-1 \) dimensional objects and iso-oriented which are themselves bounded by next lower dimensional objects with iso-orientation. The objective of this work is to study the effects of covering the objects with non iso-oriented many sided polygons in R*-trees and compare them with MBRs and MBSs.

R*-tree primarily use MBRs to cover the objects and collection of objects. The major drawback of using MBRs is that it introduces empty spaces in the MBRs as shown in Figure 8.1. The empty spaces are treated as part of the object during query processing. Hence, those objects that do not intersect the query, but whose MBRs
intersect the query are also retrieved during the execution of the query. This results in inaccurate answers to the queries. For example, the query $q$ in Figure 8.2 retrieves the objects $\text{Obj}_1$ and $\text{Obj}_2$ which are not the desired objects. This also results in the traversal of unnecessary branches of the indexing structure costing time and space.

![Figure 8.1 Object and the empty space introduced by MBR](image)

Figure 8.1 Object and the empty space introduced by MBR

![Figure 8.2 Query intersecting empty spaces](image)

Figure 8.2 Query intersecting empty spaces

Variations to MBRs are proposed in the form of spheres [105], cells [99] and boxes [106]. The Sphere trees use MBSs instead of MBRs, whereas the Cell trees use MBPs designed to accommodate arbitrary shape objects. Box-tree uses axis-aligned boxes as bounding volumes. They provide worst-case lower bounds on query complexity, showing that box-trees are close to optimal and present algorithms to convert box-trees to R-trees, resulting in R-trees with almost optimal query complexity.
The Cell tree is a clipping-based structure and, thus, a variant of Cell trees has been proposed to overcome the disadvantages of clipping. The variant uses *oversize shelves*, that is, special nodes attached to internal ones, which contain objects that normally should cause considerable splits [100, 101]. Similarly to Cell trees, [102] propose the structure of polyhedral trees or P-trees, which use MBPs instead of MBRs. The proposals in the literature that use MBPs have the following disadvantages:

a. Clipping is used which enters an object in more than one node. This results in the traversal of multiple paths for a single object,
b. The balance of the trees is compromised, that is, the lengths of the paths of the branches from root to leaves vary,
c. The partitioning of the data set is more space oriented than data oriented, the prime property of R*-tree,
d. The issue of empty spaces is not directly addressed,
e. The choice of the bounding shapes is domain specific and not generic and
f. The usage of MBPs as data partitioning method was not directly addressed.

Hence, the bounding shapes that dominated as alternative to MBRs in data partitioning paradigm are MBSs or a combination of MBRs and MBSs. But geometrically, spheres occupied more space and included more empty space. This resulted in the natural deterioration of the performance. But still MBSs were promoted as an alternative for the domains whose objects were circular. Some attempts were made to combine MBRs and MBSs. They enhanced the performance in case of NNQs but the representation became more complex and the storage cost was prohibitive. Figure 8.3 shows objects bounded with spheres and combination of spheres and rectangles.

Covering objects with polygons were a natural choice to overcome the drawbacks introduced by rectangles and spheres. Polygons reduce the empty spaces, thus reducing the retrieval of undesired objects and traversal of unnecessary paths. Figure 8.3 shows the coverage of an object with a polygon. But due to the complexity
involved in evolving the polygons to cover the objects, few attempts were made in this direction. Changing the covering shape affects the insertion and deletion algorithms. This work modifies the algorithms for rectangles and spheres in R*-tree to algorithms for polygons in R*-tree. The modified version of the R*-tree is named as *Polygon-tree*.

![Figure 8.3 Objects bounded with (a) sphere (b) sphere and rectangle (c) polygon](image)

8.2 *Polygon-tree*

Polygon-tree is the newly proposed indexing technique. This tree retains all the characteristics of an R*-tree, but modifies the MBRs to MBPs with required number of sides. An irregular $n$-sided polygon within a MBR, where $n \leq 5$, takes less space to cover an object. The number of sides is a parameter of and is a constant for the tree. This saving of space will lead to less empty space among the data objects and hence improved performance. The MBP of directory nodes are constructed to cover the MBPs
of all its child nodes. The space savings will be more evident with the directory nodes than with the leaf nodes.

To add up to the merits, MBP approximates the shape of the object better than a MBR. As an added advantage, the number of sides of the polygon used for generating MBP can be fine-tuned to suit specific applications. For instance, in the case of GIS applications, where there is much of asymmetry and large number of sides in data polygons, it would be better if one can have MBPs with larger number of sides. On the other hand, in CAD/CAM applications where one deals with only symmetrical objects, it is enough if lesser number of sides is used for MBP generation. The algorithm to generate polygons around objects is given in Figure 8.4.

```
Algorithm MinimumBoundingPolygon(n, Px, Py)
/* Trace out the MBP P of n sides for given polygon of a variable number of sides, whose vertices are given by arrays Px and Py */
1. begin
2. Calculate the center of the polygon using Px and Py;
3. Form n equal sectors around the center;
4. Locate the most distant points in each sector. If there are no edges in that particular sector, the center of the polygon is taken as the most distant point;
5. Connect the distant points to form the bounding polygon P such that the actual object is not intersected;
6. return P;
7. end;
```

Figure 8.4 Algorithm to generate polygons around objects

In the complexity front, the algorithm to construct MBP for an object is more complex than algorithms that use MBRs and MBSs both in time and space. But the computational powers of the computing systems are ever increasing. Also the storage
costs are continually decreasing. Given this scenario, it is more appropriate to find ways to decrease the input-output cost of any operation. Hence, the complexity of computations and storage can be compromised with reduced input-output cost. The algorithms for insertion, splitting, overflow handling and reinsertion are modifications of R*-tree algorithms. The modified algorithms are given in Figure 8.5.

```
Procedure ChooseSplitAxis(Node)
/* Algorithm to choose the split axis of an overflowing node */
1. begin
2. for each axis in Node do
3. begin
4. Sort the entries by the lower then by the upper value of their polygons;
5. Determine all distributions;
6. compute S, the sum of all margin-values of the different distributions;
7. end;
8. return the axis with the minimum S as split axis;
9. end;

Procedure ChooseSplitIndex(Node, SplitAxis)
/* Algorithm to choose the split index of an overflowing node */
1. begin
2. In the chosen split axis, choose the distribution with minimum overlap value;
3. Resolve ties by choosing the distribution with minimum area-value;
4. return the chosen distribution;
5. end;
```
**Procedure Split(Node)**
/* Algorithm to split an overflowing node */
1. begin
2. \( \text{SplitAxis} \leftarrow \text{call} \ \text{ChooseSplitAxis}(\text{Node}); \)
3. \( \text{Dist} \leftarrow \text{call} \ \text{ChooseSplitIndex}(\text{Node}, \ \text{SplitAxis}); \)
4. Assign Dist to two new nodes;
5. return two new nodes;
6. end;

**Procedure OverflowTreatment(Node)**
/* Algorithm to handle a overflowing node */
1. begin
2. if the level is not the root level and this is the first call of OverflowTreatment in the given level during the insertion of one polygon then call ReInsert(Node);
3. else
4. call Split(Node) and accommodate new nodes in proper places and readjust the parents;
5. return;
end

**Procedure ChooseSubtree(Poly)**
/* Algorithm to identify node for inserting a new MBP */
1. begin
2. initialize Node \( \leftarrow \) root;
3. while (1)
4. begin
5. if Node is a leaf then return Node;
6. if the child pointers in Node point to leaves then
choose the entry in Node whose polygon needs least overlap enlargement to include the new data polygon Poly;

Resolve ties by choosing the entry whose polygon needs least area enlargement;

if tie still exists then choose the entry with polygon of smallest area;

if the child pointers in Node do not point to leaves then

begin

Choose the entry in Node whose polygon needs least area enlargement to include the new data polygon Poly;

Resolve ties by choosing the entry with the polygon of smallest area;

end;

set Node ← child node pointed by the child pointer of the chosen entry;

end;

begin

for all M+1 entries of a Node do

compute distance between the centers of their polygons and the center of the bounding polygon of Node;

Sort the entries in the decreasing order of their distances computed;

Procedure ReInsert(Node)
/* Algorithm to reinsert during overflow */
begin

5. Remove the first $p$ entries from Node and adjust the MBP of Node; /*$p$ is normally 30% of the node size*/
6. With every removed polygon Poly, Starting with the maximum distance or minimum distance, call Insert(Poly) to reinsert the entries;
7. end;

Algorithm Insert(Poly)
/* Main algorithm to insert a new MBP into a polygon tree */
1. begin
2. compute Node ← Invoke ChooseSubtree(Poly);
3. if Node has less than M entries then accommodate Poly in Node;
4. if Node has M entries then call OverflowTreatment(Node);
5. if OverflowTreatment was called and a split was performed then propagate OverflowTreatment upwards if necessary;
6. if overflowTreatment caused a split of the root then create a new root;
7. Adjust all MBPs in the insertion path such that they are MBPs enclosing their children MBPs;
8. end.

Figure 8.5 Polygon-tree algorithms

8.3 Experimental Results and Discussion

Experiments were conducted to prove the effectiveness of the ameliorations proposed. The experiment was conducted using a 2-dimensional data set of 200000 data points over the space $[0, 1]^2$. The data points were randomly distributed in the space. Twelve R*-trees were constructed for varying combinations of node sizes and bounding shapes. During the process of constructing the tree, the data points were inserted in the same sequence. The specifications of the R*-trees are:
a) node size 8 with MBRs as bounding shapes,
b) node size 8 with MBS as bounding shapes,
c) node size 8 with 8-sided polygons as bounding shapes,
d) node size 8 with 12-sided polygons as bounding shapes,
e) node size 8 with 16-sided polygons as bounding shapes,
f) node size 8 with 20-sided polygons as bounding shapes,
g) node size 16 with MBRs as bounding shapes,
h) node size 16 with MBS as bounding shapes,
i) node size 16 with 8-sided polygons as bounding shapes,
j) node size 16 with 12-sided polygons as bounding shapes,
k) node size 16 with 16-sided polygons as bounding shapes and
l) node size 16 with 20-sided polygons as bounding shapes.

Three parameters were chosen for the comparison of the performances using various covering shapes. They are:

a) Number of nodes in the tree (NTot)
b) Average number of nodes accessed per insertion in the tree (IAvg)
c) Percentage of empty space in the tree (ESPer)

NTot is computed as the count of the nodes in the entire tree. The formula for the computation for IAvg is

\[ IAvg = \frac{1}{n} \sum_{i=1}^{n} I_i, \]

where \( I_i \) is the number of nodes accessed per insertion; \( n \) is the number of insertions.

The formula for the computation of ESPer is

\[ ESPer = \frac{\sum_{\forall \text{Node a tree}} \text{(area of empty space)}}{\sum_{\forall \text{Node a tree}} \text{(total space of node)}} \times 100 \]
The first comparison performed was on the number of nodes in the R*-trees constructed. The R*-tree with spheres as bounding shapes had more number of nodes and that number continually reduced as the number of sides that were used for the polygons. The results are graphically shown in Figure 8.6.

The second comparison performed was on the average number of nodes visited to insert a data point during the R*-tree construction. The R*-tree with spheres as bounding shapes accessed more number of nodes on an average during insertion and that number continually reduced as the number of sides that were used for the polygons. The results are graphically shown in Figure 8.7.

The last comparison performed was on the empty space in the R*-tree. The R*-tree with spheres as bounding shapes had higher percentage of empty space and that percentage continually reduced as the number of sides that were used for the polygons. The results are graphically shown in Figure 8.8.

The experiments showed that as the number of sides of the polygons increased, the empty spaces reduced. Reduction in the empty space resulted in better performance. Ironically, a sphere may be treated as a polygon with infinite sides, and must give optimal performance. On the contrary, spheres gave the worst performance among the different variations in the experiment. Hence, there should be an optimal number of sides for a polygon, beyond which the performance may deteriorate. Moreover, the increase in the number of sides in a polygon resulted in lengthy computations that were expensive in terms of space and time. Hence, for the proposed method to be effective, it is mandatory to arrive at an optimal number of sides a polygon can have.

Covering the objects with $d$-dimensional polygons may be cost prohibitive for implementation for $d \geq 3$. The simple reason is that a $d$-sided object needs many $(d-1)$-sided objects to cover $d$-dimensional bounding polygons. Representations become more complex and hence the computations. Hence, the proposed method may be applicable only where the dimension of the data space is 2. But nevertheless, the experiments provide an insight to the effects of empty spaces in the bounded regions.
Figure 8.6 Comparison of the number of nodes in the R*-trees constructed for various combinations of node sizes and bounding shapes

Figure 8.7 Comparison of average number of nodes accessed per insertion in the R*-trees for various combinations of node sizes and bounding shapes
8.4 Summary

The chapter studies the effects of various bounding shapes of the objects and data in a multidimensional space. Using MBRs and MBSs introduce empty spaces that result in the deterioration of performance of R*-trees. An alternative to them, namely, non iso-oriented polygons, has been proposed. The improvement in the performance of the R*-tree due to the usage of the polygons was experimentally verified.