An Analysis of R-Tree Algorithms

*Guttman’s Quadratic Split vs Greene’s Split*

Alex Church

100825475

COMP 4202, Fall 2019

Contents

[Overview 1](#_Toc26230816)

[Literature Review 2](#_Toc26230817)

[Examining the R-Tree Data Structure 2](#_Toc26230818)

[Basic R-Tree Operations 3](#_Toc26230819)

[Search 3](#_Toc26230820)

[Insert 5](#_Toc26230821)

[Deletion 13](#_Toc26230822)

[Objective & Methodology 15](#_Toc26230823)

[Conclusions 15](#_Toc26230824)

[Works Cited 15](#_Toc26230825)

# Overview

Spatial indexing is crucial problem in the scope of GIS systems, as it allows for geometric objects to be stored in spatial databases where they can be queried optimally. Whereas traditional databases often only allow for data to be stored in rudimentary forms such as numeric and character, spatial databases allow for the storage of more complex multidimensional objects such as points, lines, polygons, TINs, and other geometric representations.

Due to the added complexity of the objects they store, spatial databases require more sophisticated operations that exceed the scope of traditional database operations and basic query languages. Examples of spatial-database-specific operations include computing the distance between objects, searching for objects contained within a range, searching for all objects which overlap or intersect each other, etc. In order to facilitate these operations, spatial indexing was developed as a means of overcoming the shortcomings of traditional indexing methods, which could not efficiently handle the added complexity required by spatial databases.

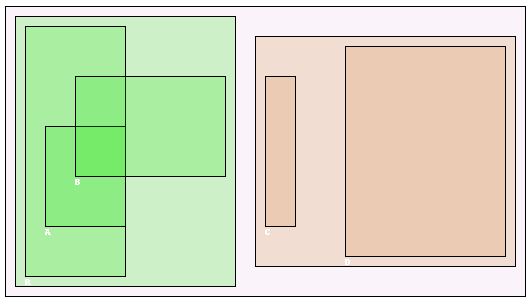
R-trees are a popular spatial indexing data structure, first proposed by Antonin Guttman at the University of California Berkley in 1984.[[1]](#footnote-1) Guttman’s initial proposal of the R-Tree cited it as a dynamic index structure that would allow for efficient searching and updating of multi-dimensional objects stored within a spatial database. He claimed that the R-Tree would accomplish this by efficiently allowing for the representation of objects by intervals in several dimensions. Since Guttman’s initial proposal many different versions of R-Trees have been developed (an example of such being the R\*-Tree), along with similar data structures that build off the R-Tree. In this paper we will focus solely on the original R-Tree implementation proposed by Guttman.

# Literature Review

## Examining the R-Tree Data Structure

The structure of an R-Tree is quite intuitive at its core. It is constructed in a similar fashion to any traditional balanced tree structure, only in the case of an R-Tree each leaf node contains a pointer to an object. For the purposes of this paper we will define an object as a simple geometric object such as a point, line, or polygon, but in practice R-Trees can be used to store a multitude of complex objects (e.g. disk pages). Similar to traditional balanced trees, R-Trees are made up of two types of nodes: leaf nodes, and interior nodes. However an R-Tree stores its nodes as tuples, where interior nodes take the form *(id, child pointer(s))*, and leaf nodes take the form *(id, pointer to spatial object)*, where *id* is the node’s unique identifier defined by a minimum bounding box. The minimum bounding box (*MBB*) is defined as the minimum bounding rectangle that contains all objects the node points to. Due to the structure of the tree, it then reasons that any node’s *MBB* contains all objects that the node points to directly, but also any objects stored along the particular node’s subtree. Nodes generally store pointers to multiple children or objects, restricted by an upper bound *M* and lower bound *m*, where each node (unless the root is a leaf node itself) is guaranteed to store at least *m* pointers, and at most *M* pointers. Due to its balanced structure, all leaf nodes of an R-Tree are guaranteed to appear on the same level.

Below is a simple representation of an R-Tree whose leaf nodes store pointers to rectangular objects (R, A, B, C, D) -



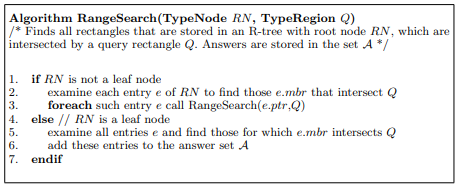
Here the exterior rectangle represents the minimum bounding box for all objects. This would be the root node of the R-Tree, stored as the tuple (MBB id, [green MBB id, red MBB id)]. Its children represent the two leaf nodes of this tree. These are defined by the green and red minimum bounding boxes containing objects (R, A, B) and (C, D) respectively. The leaf node denoted by the tuple (green MBB id, [R, A, B]) would contain three pointers to its objects (R, A, B), while the leaf node denoted by the tuple (red MBB id, [C, D]) would contain two pointers to objects (C, D). The height of an R-Tree is at most , where *m* represents the aforementioned minimum-branching factor, and *N* represents the total number of nodes. However, because nodes will likely contain more than *m* entries, the tree height is generally even more optimal in practice.

## Basic R-Tree Operations

### Search

The underlying structure of the R-Tree makes searching for objects both efficient, and easy to implement. Starting at the root node, you traverse the tree, selecting at each step the subtree whose *MBB* contains the object in question, until you arrive at the leaf node that points to the object. Based on our prior claim about the upper bound on the height of the tree, it surmises that the R-Tree’s basic search operation can thus be executed in run time.

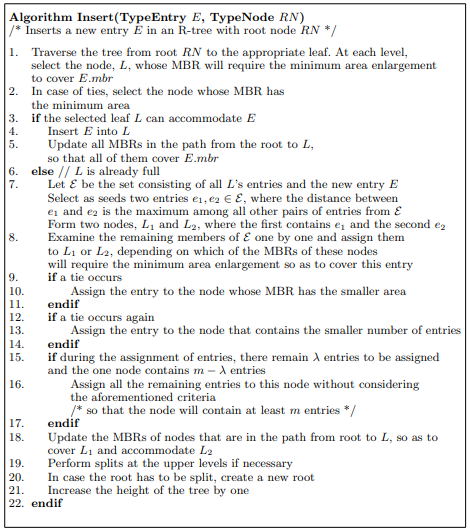
Beyond basic search, other variations on the search algorithm exist, such as the range search algorithm. This allows for a search rectangular range to be passed as a parameter, and returns the set of *MBBs* within the R-Tree that overlap the specified range. The algorithm can be defined as follows -

[[2]](#footnote-2)

Each node is checked at every level to identify whether or not it overlaps region Q. If yes, the subtree of that node is recursively searched as well. It should be noted that this algorithm only acts as a filter when the objects contained within the leaf *MBBs* are not rectangles, as it is possible for two objects to have their *MBBs* overlap without having the objects themselves overlap at any point.

### Insert

Inserting a new object into the R-Tree is a little more complex than simple search. Because only leaf nodes store pointers to objects, inserting a new object will always occur at the leaf node level. Below is the basic description of the insertion algorithm –

[[3]](#footnote-3)

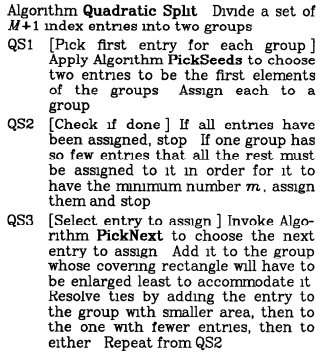
The first step in insertion is to determine which leaf node’s MBB the object should be inserted into. If a leaf node exists such that its *MBB* fully contains the new object, this is a relatively straightforward process as we saw with basic search. However, there exists the possibility that no leaf node’s *MBB* fully contains the object to be inserted. In this case we must determine the *MBB* that requires minimum expansion to accommodate the new object. This expansion may be performed at multiple steps as we traverse the R-Tree from root to leaf level. Once a leaf has been selected, insertion into a node that has fewer than or equal to the predefined upper bound *M* pointers is trivial. If, however, insertion would cause the leaf to violate this bound, the leaf must be split in two, causing two new MBBs to be formed and two new pointers to those MBBs to be added to the leaf’s parent node in place of the single pointer that existed previously. This can cause the parent node to overflow if adding the additional pointer causes it to now violate the upper bound of *M* pointers, which in the worst case can cascade all the way up to the root, where a new root must be created if the root is split. When determining the optimal split for any given node, we must first determine which two subsets of the node would result in the minimal area for both newly created nodes. The worst case time complexity in such a case is costly. There exist, however, heuristic algorithms that can help improve on this worst case complexity when splitting, where there exists a trade-off between the R-Tree’s optimal structure and the time complexity for insertion of a new object.

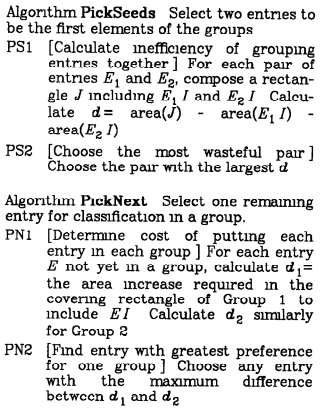
#### Exponential Split

The most costly on the implementations, it checks all possible subsets of a node, and computes the total are for each, before selecting the optimal split that would guarantee minimum total area. This approach guarantees an optimal R-Tree structure, where the wasted space of all *MBBs* is minimized, however, this comes at the cost of efficiency.

#### Quadratic Split

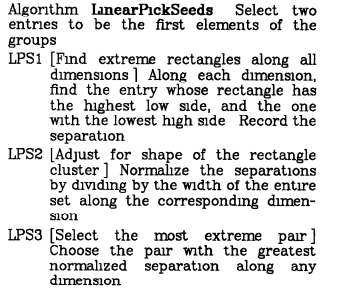
Two children contained within the node are selected as seeds such that if they were to be contained within the same *MBB* it would lead to the most wasted space of all possible two child subsets. These form the basis for the two split nodes. The remaining children are then iterated over, and added to the split node for which adding them would minimize the total wasted space (i.e. the node that requires the least expansion to accommodate the object).



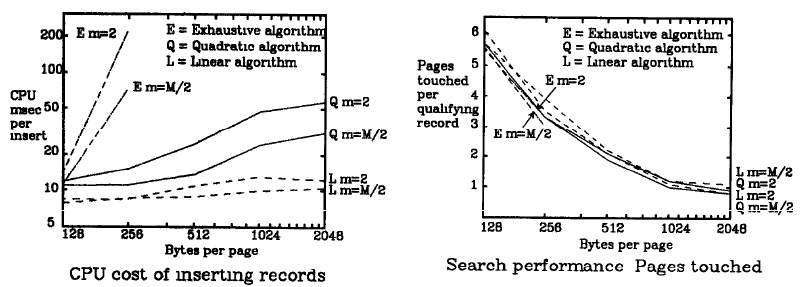


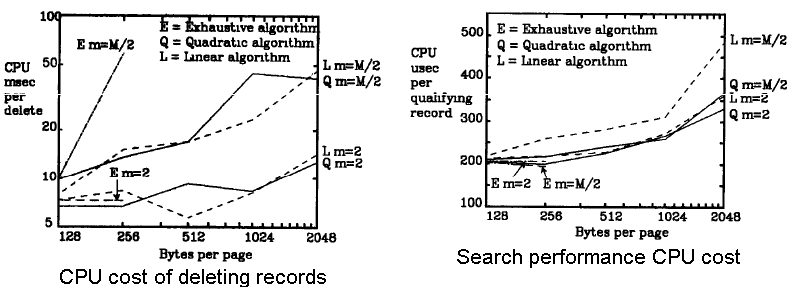
#### Linear Split

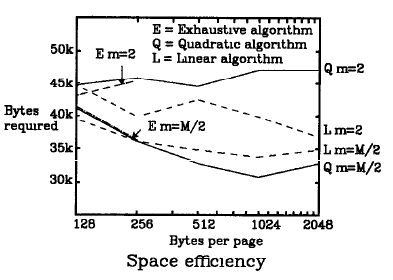
Begin by selecting the two children of greatest distance from each other. The remaining children are then iterated over in random order and assigned to the split node for which adding it would require the least wasted space.



In his paper, Guttman offered the results of a number of performance tests he ran, using the various splitting algorithms, each time using different levels of parameters *m* and *M*. The three tests he ran were 1) Insertion performance on a large number of entries, evaluating performance only for the final 10% of insertions (when the tree was close to its final size), 2) 100 individual search queries, each querying approximately 5% of the total entries, and 3) deletion of 10% of the entries, staggering the deleted nodes across the tree by only deleting every 10th entry. Some snapshots of the results can be seen below –





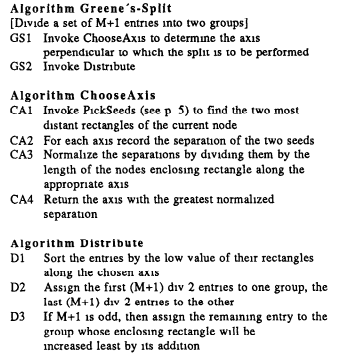


As we would expect, insertion cost increase exponentially as *M* does. Guttman also found that the cost of insertion was minimized with stricter balancing bounds. Similarly, the cost to delete an item is heavily reliant on the minimum bound *m.* Search on the other hand proved to be relatively insensitive to which splitting algorithm was used. This hints at the fact that the performance trade-off the exponential algorithm offers does not lead to an equal increase in data structure efficiency by organizing the tree in the optimal fashion.

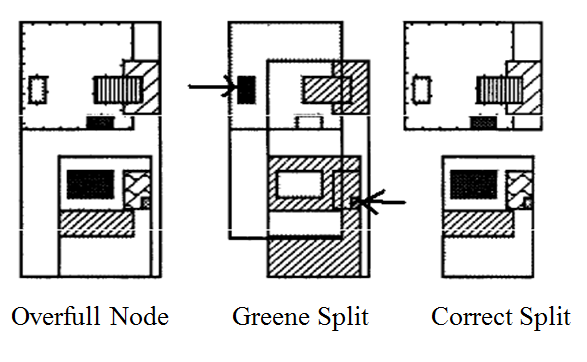
Based on his findings, Guttman himself suggested using this quadratic split algorithm, as it offers the best compromise in the trade-off between effectiveness and efficiency, though best algorithm for any given implementation is entirely dependent on the requirements of the implementation in question. The main critiques against the quadratic split algorithm were that 1) distributing entries during a split would always favour the larger rectangle, as naturally it would require the least expansion, and 2) as soon as one split node is allocated *M*-*m*+1entries, all other entries are allocated to the second split node automatically.

#### Greene’s Split

In 1989 at the International Conference for Data Engineering, Diane Greene proposed a possible alternative to Guttman’s quadratic split, which has since come to be known as Greene’s split.[[4]](#footnote-4) Similar to Guttman’s quadratic split, Greene begins by finding the two most distant entries. Based on those, we must then determine the axis that is perpendicular to the proposed split. The entries are then sorted based on their values along that axis. The first (*M*+1)/2 entries are allocated to one split node, and the remaining (*M*+1)/2 entries are allocated to the second split node. If (*M*+1) is odd then the leftover entry is assigned to the split node whose *MBB* would require the least expansion. The full algorithm is detailed below -

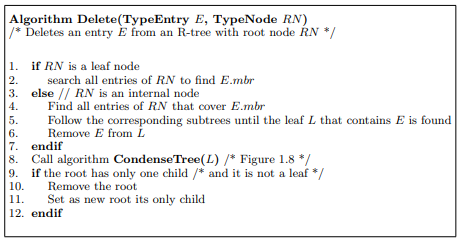
[[5]](#footnote-5)

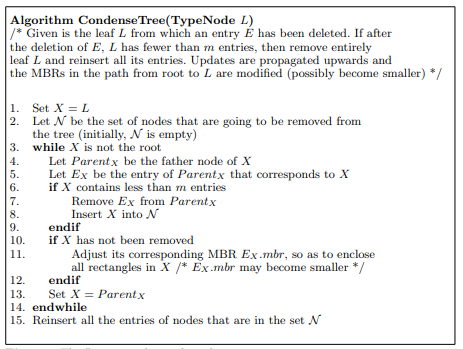
Greene’s test results showed that in a number of scenarios that her proposed splitting algorithm offers a marginal improvement in performance on Guttman’s quadratic splitting algorithm. However, the Greene split is not without its own shortcomings. Firstly, it is no guarantee that the proper axis is selected. There exist cases where splitting on another axis would offer a more optimal split. In these cases Greene’s split can lead to significantly worse performance than the traditional splitting algorithm. Consider the following case for example, where the split occurs along the X axis, when in fact splitting across the Y axis would lead to a much more optimized split –



### Deletion

Similar to insertion, deletion first requires a search to find the leaf that holds the object in question, returning Null if the object in question is not stored in the R-Tree. If the leaf node contains more than the minimum required pointers (*m*), deletion is trivial. However, if deletion would cause the node to fall below the lower bound, we must dissolve the node, re-inserting any remaining pointers back into the tree, which as we saw previously with the insertion algorithm, can have a cascading effect as the R-Tree tries to maintain its balance. The full details of the delete algorithm are as follows -



[[6]](#footnote-6)

# Objective & Methodology

We would like to compare the performance of Guttman’s quadratic split in comparison with Greene’s split. In doing so we will test different upper and lower bounds for parameters *m, M,* on a large number of insertions. Our performance measures will measure efficiency in two ways: 1) speed, measured by average time to insert a significant number of nodes 2) wasted space, defined as the area inside the root *MBB* that is occupied by its ancestors (i.e. more space occupied by a root’s ancestors equates to a greater portion of wasted space).

#### Average Insertion Time

In order to test the efficiency of each split algorithm we will first analyze the time it takes to construct an RTree via an arbitrary number of insertions. In order to do so, we randomly generated 1000 distinct rectangles on the 2-dimensional plane, and generated the tree using these points, first inserting the objects using Guttman’s quadratic split, and then using Greene’s split. This process was then repeated 100 times for each algorithm, and an average time was computed from among all the iterations. This was then repeated again for various values of *M.* The results are as follow –

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Average time for 1000 insertions (seconds)** | | | | | |
| *M value* | *4* | *8* | *16* | *32* | *64* |
| *Rtree* | 10.46249 | 10.75072 | 14.66243 | 18.94153 | 25.74017 |
| *Greene Tree* | 8.666232 | 9.315397 | 14.05755 | 19.13027 | 32.35806 |

# Conclusions

# Works Cited

Beckmann et al., Norbert. *The R\*-Tree: An Efficient and Robust Access Method for Points and Rectangles+.* Praktuche Informatlk, Umversltaet Bremen, 1990.

Greene, Diane. *An Implementation and Performance Analysis of Spatial Data Access Methods.* International Conference of Data Engineering, 1989.

Guttman, Antonin. *R-TREES. A DYNAMIC INDEX STRUCTURE.* University of California Berkley, 1984.

Laura Toma, Department of Computer Science, Bowdoin College. n.d. http://www.bowdoin.edu/~ltoma/teaching/cs340/spring08/Papers/Rtree-chap1.pdf.

1. (Guttman 1984) [↑](#footnote-ref-1)
2. (Laura Toma n.d.) [↑](#footnote-ref-2)
3. (Laura Toma n.d.) [↑](#footnote-ref-3)
4. (Greene 1989) [↑](#footnote-ref-4)
5. (Beckmann et al. 1990) [↑](#footnote-ref-5)
6. (Laura Toma n.d.) [↑](#footnote-ref-6)