Analysing the Performance of Nearest Neighbour and Window Queries on R-Trees

Michael Stewart
Honours Student
email: 20947715@student.uwa.edu.au

29 May 2015

Abstract

The R-Tree is the primary data structure for storing and retrieving spatial data. Two of the most popular spatial queries on R-Trees are the nearest neighbour and window query. The nearest neighbour query retrieves the node closest to the given query node. On the other hand, the window query returns all nodes that lie within a given query window. This paper aims to demonstrate that the average-case complexity of the algorithms behind these two queries is $O(logN)$. This is achieved by logging the results of a small Python program that builds R-Trees and performs each query on each tree 1,000 times. It is found that the order of growth of each algorithm is indeed logarithmic and these results demonstrate why the R-Tree is still the primary spatial data structure today.

Keywords: Spatial databases, R-Trees.

1 Introduction

A spatial database holds information relating to the location of objects in space. One of the most important applications of spatial databases is the geographic information system (GIS), which aims to store spatial data that is usable in real-world settings [5, p. 191]. GIS allow for the existence of digital world maps, computer aided design (CAD), transport planning, and other important applications in our everyday lives [4]. The importance of spatial databases is currently growing due to the recent growth in the number

of large datasets available, which include sensor images and moving object trajectories [2].

Spatial databases heavily rely on R-Trees to store and query spatial data. The R-Tree is a tree-based data structure that allows for spatial queries to be performed on spatial databases. It was first introduced by Guttman [3] in 1984 as an alternative structure to the other data structures available at the time for handling spatial data, such as quad trees and k-d trees. The R-Tree proved to be a useful spatial data indexing structure and is now very commonly used in spatial databases.

R-Trees facilitate the ability to conduct *nearest neighbour* and *window* queries with a reasonable performance, even when handling many data points [7]. These two query types are some of the most popular spatial queries. The nearest neighbour query aims to find the closest node to the given node, whereas the window query (also known as the "intersection query") aims to find all nodes that lie within a given query window. This paper aims to empirically show that the average-case complexity of these queries on a database using an R-Tree to store data is $O(logN)$.

2 Previous and related work

2.1 R-Trees

The most ubiquitous paper written on R-Trees is the original paper written by Guttman [3]. Guttman's paper outlines the structure of the R-Tree, as well as the algorithms behind it and its performance results. The performance results documented in this paper pertain to the time taken to insert records into the R-Tree. However, this paper does not document performance results on nearest neighbour or window queries. This paper will present these missing query performance results using a modern system in order to demonstrate that, despite being over thirty years old, the R-Tree is still the premier data structure behind a spatial database system.

2.2 R*-Trees

Beckmann et al. [1] introduced the R*-Tree in 1990, which extends the R-Tree to improve query performance at the cost of increased insertion time. The increased query performance is achieved by optimising a number of features of the R-Tree, such as minimising overlap and decreasing the number of paths that are traversed while performing queries. Beckmann et al. found that the R*-Tree exhibits good performance when handling non-uniform datasets

due to its ability to minimise the area, margin and overlap of the directory rectangles. Because of its increased query performance, the R*-Tree is still commonly used to extend the regular R-Tree in spatial databases today.

2.3 Hilbert R-Trees

Kamel and Faloutsos [4] also developed an extension of the R-Tree in 1993, which is known as the "Hilbert R-Tree" (HRT). The HRT acts like a regular R-Tree during querying, but implements features common to a different tree type, known as the B-Tree, during insertions. This allows for increased performance when conducting insertions. There are two variants of the HRT, which are static and dynamic. These two variants cater to different purposes. The static HRT is most appropriate when the number of modifications to the data is low, while the dynamic HRT is best when the data is frequently modified.

3 Methodology

In order to store data in a tree structure, the R-Tree clusters points together into rectangles based on proximity. It then assembles them into a hierarchal structure in the form of a tree [1]. As shown in Figure 1, nodes are placed into the tree based on their position within the R-Tree's minimum bounding rectangles (MBR). This means that nodes that are close together will be grouped into the same leaf node within the tree. In Figure 1, the R-Tree's capacity is three entries per node, and this is reflected in the tree calculated from the spatial data. The algorithm for inserting nodes into an R-Tree is described in Algorithm 1.

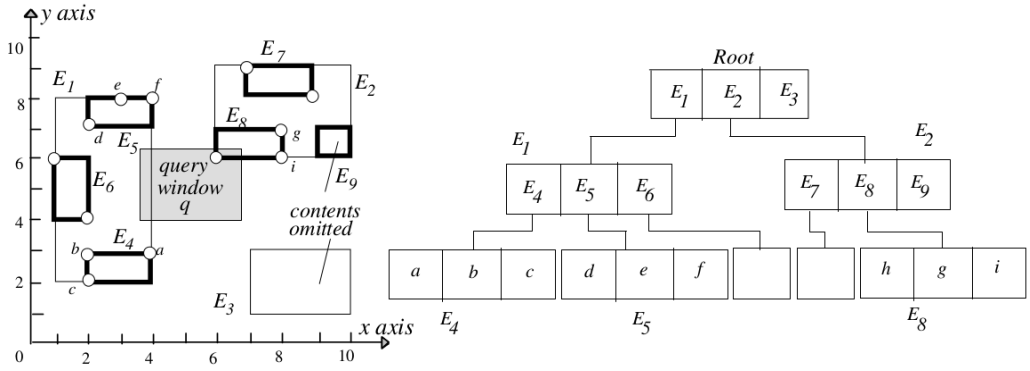

Figure 1: A visual representation of an R-Tree [7].

S1: [**Find position for new record**]
Select a leaf node L in which to place E

S2: [**Add record to leaf node**]
If L has room for another entry, install E
Otherwise, split the node to obtain L and LL containing E and all the old entries of L

S3: [**Propogate changes upward**]
Adjust the tree to include the changes made on L

S4: [**Grow tree taller**]
If node split propogation caused the root to split, create a new root whose children are the two resulting nodes

Algorithm 1: Inserting a new index entry E into an R-Tree [3].

3.1 Spatial Queries

3.1.1 Nearest Neighbour

The nearest neighbour retrieves the data point that lies closest to a query point [7]. One of the most widely used nearest neighbour algorithms was developed by Roussopoulos et al. [6] in 1995. Their algorithm is branch-and-bound and is based on a depth first search that recursively traverses the R-Tree in order to find the node that is closest to the query node. Pruning is used to reduce the number of nodes searched, increasing performance in large search spaces. Figure 2 shows a block diagram that outlines the process of Roussopoulous et al.'s nearest neighbour algorithm.

3.1.2 Window/Intersection

Window queries, also known as intersection queries, aim to determine all nodes that lie within a given query window [7]. These queries are very useful for applications such as cartography and location-based services. The algorithm for processing window queries involves retrieving the root node, and visiting all nodes within it that intersect the query window q. Each visited node is recursively searched down to the leaf level, and any points that lie within q are added to the results. Any entries that do not lie within the q are not searched. This algorithm is outlined in Figure 3.

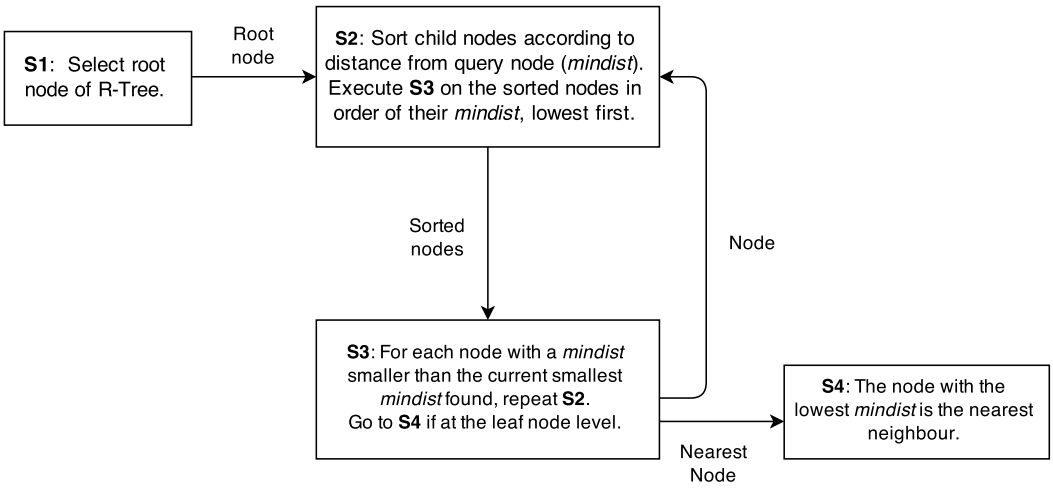

Figure 2: A block diagram that outlines the steps for calculating the nearest neighbour of a given query node [6, 7].

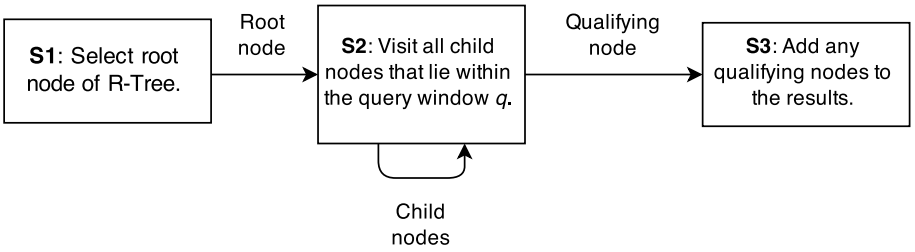

Figure 3: A block diagram that outlines the steps undertaken in a window query [7].

4 Implementation

In order to test the average-case complexity of the two query types, I used `libspatialindex`[1], a C library. This library contains an R-Tree data structure and supports nearest neighbour and window queries. I wrote a small program in Python 2.7 that uses the `Rtree`[2] module to access the `libspatialindex` library and build a R-Trees with a varying number of randomised data points. 1,000 nearest neighbour and 1,000 window queries are conducted on R-Trees ranging in size from 100,000 - 2,000,000 nodes and the time taken to perform these queries is printed. The code for this program is shown in Figure 4.

```
from rtree import index
from random import random
from datetime import datetime
for i in xrange(20):
    timer = datetime.now()
    num_indexes = 100000 * (1 + i)
    print (str(num_indexes) + "\t\t|\t"),
    rands = [random() for i in range(num_indexes)]
    # Add all points to the R-Tree
    tree = index.Index()
    for i, coord in enumerate(rands):
        tree.insert(i, (coord, coord+1, coord, coord+1), None)
    print ("A: " + str(datetime.now() - timer) + "\t"),
    timer = datetime.now()
    # Calculate 1,000 nearest neighbour queries
    for x in range(1000):
        list_nearest = list(tree.nearest((rands[x], rands[x],
        rands[x], rands[x]), 1))
    print ("N: " + str(datetime.now() - timer) + "\t"),
    timer = datetime.now()
    # Calculate 1,000 random window queries
    for x in range(1000):
        list_intersection = list(tree.intersection((random(),
        random(), random() + 1, random() + 1)))
    print ("W: " + str(datetime.now() - timer) + "\t")
```

Figure 4: A small python program that generates 100,000 - 2,000,000 random nodes, adds them to an R-Tree, and conducts 1,000 nearest neighbour and 1,000 window queries.

[1] *Libspatialindex*. http://libspatialindex.github.io/.
[2] *Rtree*. http://toblerity.org/rtree/.

5 Results and Discussion

The purpose of this experiment was to show that the average-case complexity for the nearest neighbour and window queries on an R-Tree is $O(logN)$. The R-Trees generated by my Python program range in size from 100,000 nodes to 2,000,000 nodes in steps of 100,000. All results were obtained on an Asus N53SV laptop, which contains an Intel Core i7-2720QM processor. Table 1 shows the time taken to conduct 1,000 nearest neighbour and 1,000 window queries on randomised R-Trees. These results are graphed in Figure 5.

Number of Nodes	**N.N. Queries**	**W. Queries**
100000	00:01.340	00:07.919
200000	00:01.938	00:15.544
300000	00:02.469	00:25.720
400000	00:03.071	00:32.592
500000	00:03.673	00:39.467
600000	00:04.065	00:46.064
700000	00:04.331	00:54.459
800000	00:04.701	01:07.278
900000	00:05.347	01:16.976
1000000	00:05.253	01:20.213
1100000	00:06.111	01:26.807
1200000	00:06.627	01:39.099
1300000	00:07.147	01:38.672
1400000	00:07.534	01:54.553
1500000	00:07.969	02:08.851
1600000	00:07.930	02:18.135
1700000	00:08.915	02:25.404
1800000	00:08.908	02:37.327
1900000	00:09.341	02:40.230
2000000	00:09.544	02:45.911

Table 1: The time taken to conduct 1,000 nearest neighbour and 1,000 window queries for varying numbers of nodes in an R-Tree (time in minutes).

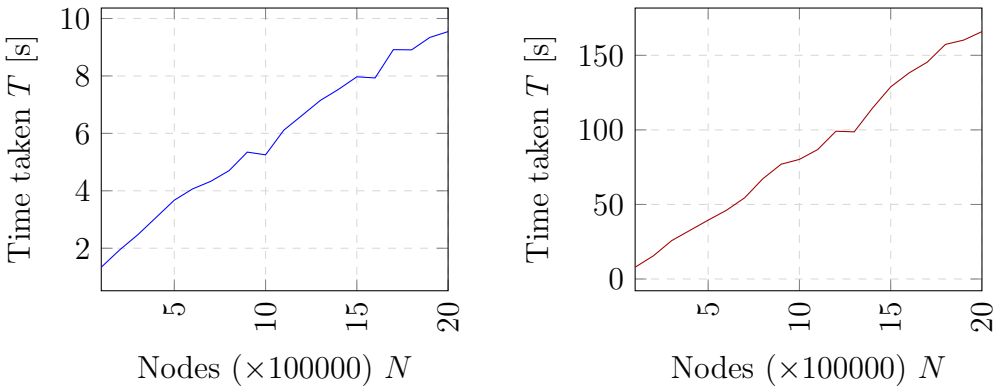

Figure 5: Graph showing the time taken to perform 1,000 nearest neighbour queries (left) and 1,000 window queries (right) on randomised R-Trees of size N.

The results in Table 1 and Figure 5 show that the time taken to conduct nearest neighbour and window queries on a random R-Tree appear to grow logarithmically with respect to the number of nodes in the R-Tree, N. Also, there are certain values of N in the data where the time taken for the nearest neighbour/window queries to be calculated do not follow the same logarithmic path as the other values of N. This is due to the situation whereby portions of the tree that cannot possibly yield nearest neighbours/intersecting nodes are ignored by the algorithms. Ultimately, the average-case complexity of the nearest neighbour and window query algorithms is $O(logN)$, and this is reflected in these results.

5.1 Potential Improvements

In order to improve the accuracy of these results, it would be useful to generate R-Trees with higher numbers of nodes. Graphing the query performance results of even larger R-Trees would demonstrate the logarithmic growth of the running time more accurately. Further improvements to the accuracy of the results could be made by running the tests separately rather than sequentially, to allow the processor to cool down between tests. It is possible that the processor temperature made a difference to the results, and running tests with breaks in between would prevent this.

6 Conclusion

This paper aimed to demonstrate the solid performance of the nearest neighbour and window query algorithms for R-Trees. This was achieved by constructing a simple Python program that built R-Trees of various sizes and conducted 1,000 of each query on each tree. The results of this program were shown in Section 5. These results demonstrate that the order of growth of

the time each query takes to process is logarithmic-based and the average-case complexity is $O(logN)$. The logarithmic average-case complexity of the R-Tree explains why it still plays a critical role in the storage and retrieval of spatial data today.

References

[1] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. *The R*-tree: an efficient and robust access method for points and rectangles*, volume 19. ACM, 1990.

[2] Vania Bogorny and Shashi Shekhar. Spatial and spatio-temporal data mining. In *Data Mining (ICDM), 2010 IEEE 10th International Conference on*, pages 1217–1217. IEEE, 2010.

[3] Antonin Guttman. *R-trees: a dynamic index structure for spatial searching*, volume 14. ACM, 1984.

[4] Ibrahim Kamel and Christos Faloutsos. Hilbert r-tree: An improved r-tree using fractals. 1993.

[5] Karen Kemp. *Encyclopedia of geographic information science*. Sage, 2008.

[6] Nick Roussopoulos, Stephen Kelley, and Frédéric Vincent. Nearest neighbor queries. In *ACM sigmod record*, volume 24, pages 71–79. ACM, 1995.

[7] Jun Zhang, Manli Zhu, Dimitris Papadias, Yufei Tao, and Dik Lun Lee. Location-based spatial queries. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 443–454. ACM, 2003.