Comparing the Cost and Efficiency of Nearest Neighbour Search in K-d Trees versus R-Trees

A final project presented for COMP 4202: Computational Aspects of Geographic Information Systems

Karl Damus — 101196754

Carleton University

School of Computer Science

Contents

1 Abstract

Querying spatial data is one of the most important elements of working with a Geographic Information System (GIS). Being able to execute queries on a GIS data structure efficiently and accurately has a tremendous impact on how useful or unuseful your GIS is. Because a GIS has a vast number of applications in the real world including, but not limited to, urban planning, weather forecasting, and agriculture,[7] there are consequences to an inefficient GIS due to your data structures not being able to execute queries in a timely manner. This paper aims to assess the capabilities of R-trees and K-d trees when being used for finding a nearest neighbour. Through analysis and a comparative study, I will identify which data structure is better suited for this specific task.

2 Introduction

2.1 Brief History

Geographic Information Systems being ported as computer applications started in the 1960s. The most widely known system is the Canada Geographic Information System (CGIS) developed by IBM. These were developed on slow mainframes with next-to-zero mass storage compared to today's standards; instructions were sent into the system using punch cards or teletyped in at 300 bits per second. A main reason for transitioning GIS to computerized system is that it would be more efficient to computerize map analysis of agriculture. Well this would be extremely labour intensive to do by hand, computers could handle the data if stored and organized appropriately. This led to the creation of various indexing and tiling schemes such as Z order (previously named Morton order). Z order is still used today for spatial indexing and various other applications in GIS.

It is clear that many advances have been made since the 1960s in terms of storage, computing power, new optimized algorithms, and accessibility. However, it is interesting to note that much of GIS's aspects have remained unchanged. Michael F. Goodchild leaves us with a few questions in his article "Reimagining the history of GIS": How rapidly is GIS evolving and improving as a technology? To what degree are GIS's capabilities constrained by outdated decisions that no longer make sense? And how much has GIS overlooked important opportunities presented by recent technological advancements. He then leaves us with the following statement: "If we were to wipe the slate clean and reinvent GIS today, with today's computing power and visualization capabilities, the result might be very different"[8].

2.2 Purpose of the Study

The purpose of this study is to learn about the implementation of K-d trees and R-trees. I will implement the data structures in Java. The study will also place a strong focus on the performance of executing nearest neighbour queries on the data structures. I will examine the capabilities of K-d trees and R-trees in storing varied amounts of spatial data and assess the performance of each during nearest neighbour query execution. I will compare and contrast the data structures to better understand which is better suited for this use case.

3 Literature Review

3.1 What is GIS?

A Geographic Information System (GIS) is any system that assists in analyzing, creating, managing, and viewing maps using geographical data. GIS is used in almost every industry making it easy to share analysis on information allowing complex problems in science to be worked on and solved collaboratively.

> [GIS] helps people reach a common goal: to gain actionable intelligence from all types of data.[1]

GIS can be placed into 4 main categories. Maps are the foundation for holding visual data. Maps make viewing data and geographical features accessible to the average person. Data is important to any GIS as without data there is nothing to view and analyze. The 3 main categories of data in GIS are *vector data*, *raster data*, and *triangulated data*. Vector data is the most common type of data, it includes simple point, line, and polygon data. Raster data is made up of pixels where each pixel has a value. Raster data can be split into two types: continuous and discrete. Continuous data has cells that gradually change such as satellite imagery. Discrete data has a specific theme where each pixel takes on a specific value instead of a range of values. Triangulated data is a twist on vector data. Triangular irregular networks (TIN) are made up of sets of vertices connected to form a network of triangles. Analysis is the process of evaluating data. Analysis in GIS allows one to interpret the data leading to precise calculations, estimations, or predictions. Applications allow users to work with data and combine all of the 3 other main categories of GIS to be able to work hands-on with GIS.

3.2 Overview of GIS Data Structures

There are a wide variety of GIS data structures available, and which one you use depends on what type of data you are working with. The following are some examples of data structures you can use for the respective categories:

- Raster Data: a simple 2D array will suffice for storing a grid of cells where each cell stores a value corresponding to a specific value (e.g., elevation).
- Vector Data:
  - Points: tuples or structs will work well to store x and y coordinates.
  - Lines: a list of points (tuples or structs) can store lines.
  - Polygons: a list of points (tuples or structs) that closes (identical first and last points) can store polygon data.
- Triangulated Iregular Network (TIN): a triangular struct will work for this where each instantiation of the struct holds the triangle's vertices.
- Network Data Structure: a connected list or matrix will work to represent spatial entities as nodes and their connections (e.g., 2D array).
- Tree Data Structures: a tree structure improves efficiency of various analysis methods. Trees can store all different types of data depending on their implementation. Examples of tree data structures commonly used in GIS include: B-tree, R-tree, quad tree, and K-d Tree.

These data structures allow the storing of the many different types of GIS data and they can be quite efficient at analyzing the data. An example of a method I will implement and discuss is the nearest neighbour search.

3.3 Previous Studies on Cost and Efficiency of GIS Data Structures

Below, I have listed two papers that are relevant to my studies on this topic. I will give a brief overview of what they cover and summarize a brief section.

The Design and Analysis of Spatial Data Structures[5]

Samet, Hanan
University of Maryland

In this paper, Hanan discusses quad trees, K-d trees, range trees, priority search trees, as well as various methods such as the plane-sweep. Hanan discusses searching techniques and says they can be grouped into three basic categories: organization of the actual data, organization of the space from which the data is drawn, and a no-organization-needed category. Hanan uses depth to show the organization level of the data. At a depth of 0 there is no organization. At a depth of 1, there is only one consideration made for organization. At a depth of 2 and beyond, every element is taken into account for organization.

Multidimensional Access Methods[6]

Gaede, Volker and Günther, Oliver
Humboldt University, Berlin

In this paper, Volker and Oliver discuss the efficiency of search operations such as the point query and the region query using various data structures. The talk about the difficulty of choosing the best spatial access method because there are so many to choose from. They list the factors that affect the efficiency as: data nature, query types, and data correlations. Because there are so many to choose from, an access method optimized for one type of query or data may not perform for other types. This means to get complete optimization of spatial queries, you sacrifice space complexity for time complexity. Because of this, many commercial products choose simpler structures like Z-ordering and R-trees instead of going for high performance options. They conclude by saying that until the technological advancements expand more, users will be stuck choosing an access method that provides the best fit for their singular application.

3.4 Factors Influencing Cost and Efficiency

The primary factors influencing cost and efficiency of GIS data structure implementations are user implementation and building the data structures. I will go more into detail about my findings in a later section but as a quick example: building a K-d tree with 10,000,000 points took $\sim$2.5 minutes. Executing 5,000,000 nearest neighbour searches on the same tree took $<$0.01 milliseconds on average. This highlights the difference in creation of the data structure versus executing the highly efficient and specific operations that are made for particular data structures.

4 Methodology

4.1 Data Collection

The foundation of this project lies in having similar data that is relevant to the data structures I implemented. I collected data from a variety of sources to allow myself a few options when playing around with the data structures in my IDE. I also generated my own random point data for simple testing. I mainly gathered data from 'Natural Earth Data'.

I made sure to collect similar data to make the importing process easier and limit having to modify too many parameters to make all the different data sources work on the same system. I also put limits for certain areas of my implementation such as only allowing .shp files to be viewable in the Java Swing portion of my application.

4.2 Selection Criteria

My selection criteria was defined to incorporate varying sizes of data to allow for tests to be varied across the board. I selected some small and some large data sets so that I would have a variety of data to work with. I also made sure the data types I was collecting were meant to work with the data structures I am using. I also chose data that I had a previous understanding of so that I could easily verify what I was analyzing.

4.3 Analytical Tools and Techniques

In order to analyze the performance of GIS data structures I needed to have appropriate tools and techniques. The comparative analysis was done using custom implementations of the data structures in Java and using existing Java libraries to measure the time of executing queries on these data structures with varying sizes of input. Java was used solely for programming the data structures, importing and manipulating the data, displaying the data, and running tests on the data structures.

In order to verify my results I used existing visualization software such as Desmos graphing calculator with my results from small to medium sized inputs. I ensured that the results were correct 5 times for each size data set. I then increased to large sized inputs and using estimation and proven results from smaller data sets, I could confidently gauge whether a given output was correct or not.

For example with my nearest neighbour search I first printed every point added to the data structure in various sizes up to 1000 points, added them to the data structure, ran the nearest neighbour search with a random point to test and outputted the results. I added all the printed points and the testing point to Desmos graphing software and manually checked whether the resulting point was actually the closest to the testing point. I did this 10 times per data structure to ensure my results were valid. Below is my output in Desmos from this test. I further checked larger data sets by both visualization in Java Swing and estimation based on the outputted testing point and the nearest neighbour my algorithm found. I could estimate it was correct based on previous results with smaller data sets and determining whether it was reasonable to assume the outputted point was the closest point in the data set.

5 Implementation

5.1 Software and Tools

The main tools I will be using are:

- Java
- IntelliJ IDEA

The data structures I will be implementing are:

- K-d Tree
- R-tree

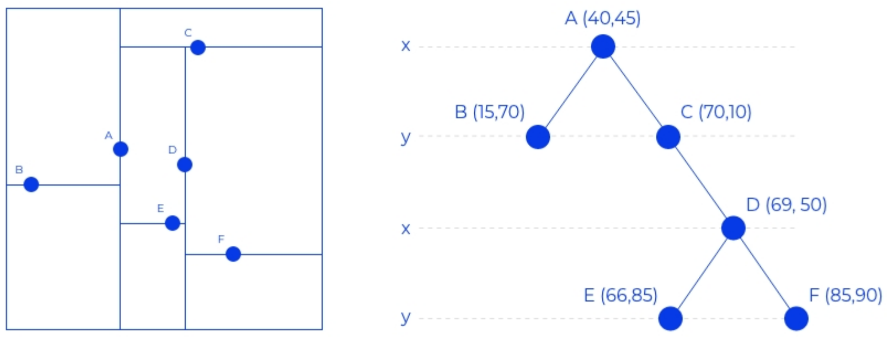


Figure: K-d tree diagram

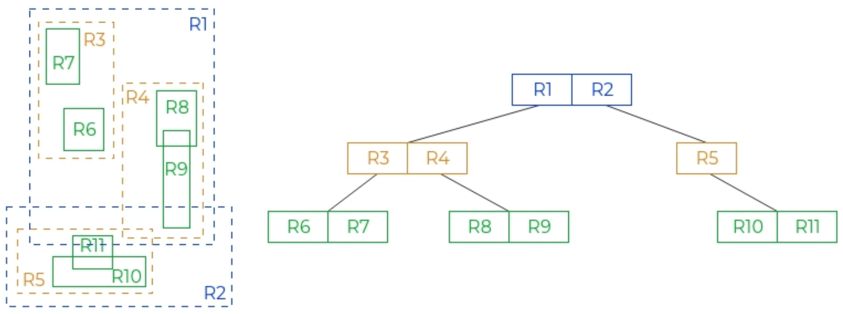
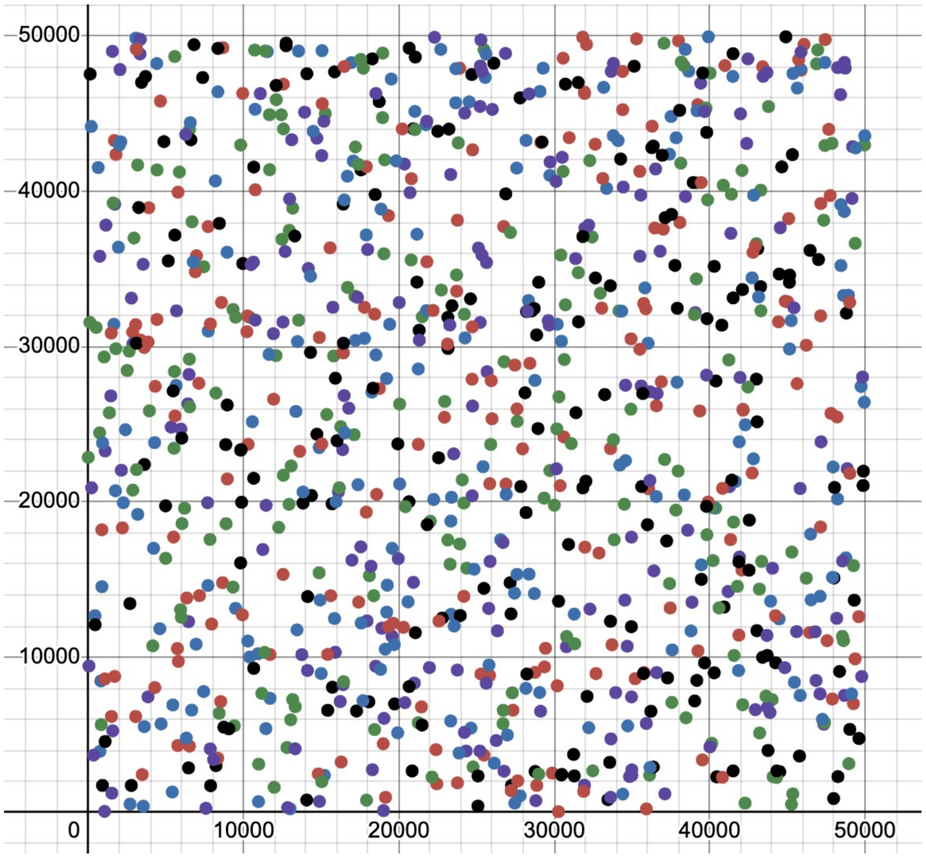


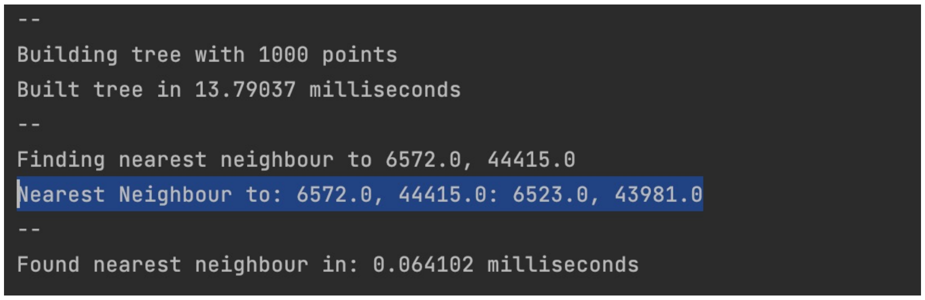
Figure: R-tree diagram

5.2 Testing, Validation, and Results

I will cover a couple examples from my testing and my method for validation. This first test is from nearest neighbour search using Desmos for smaller sized sets. I had my program output every point in the data set and added them to Desmos. I then had my program print out the randomly generated point it was using to find the nearest neighbour to and added it to Desmos. I zoomed in to find the target point and the testing point and visually confirmed it was the nearest neighbour. I did this 5 different times before moving on to larger data sets.

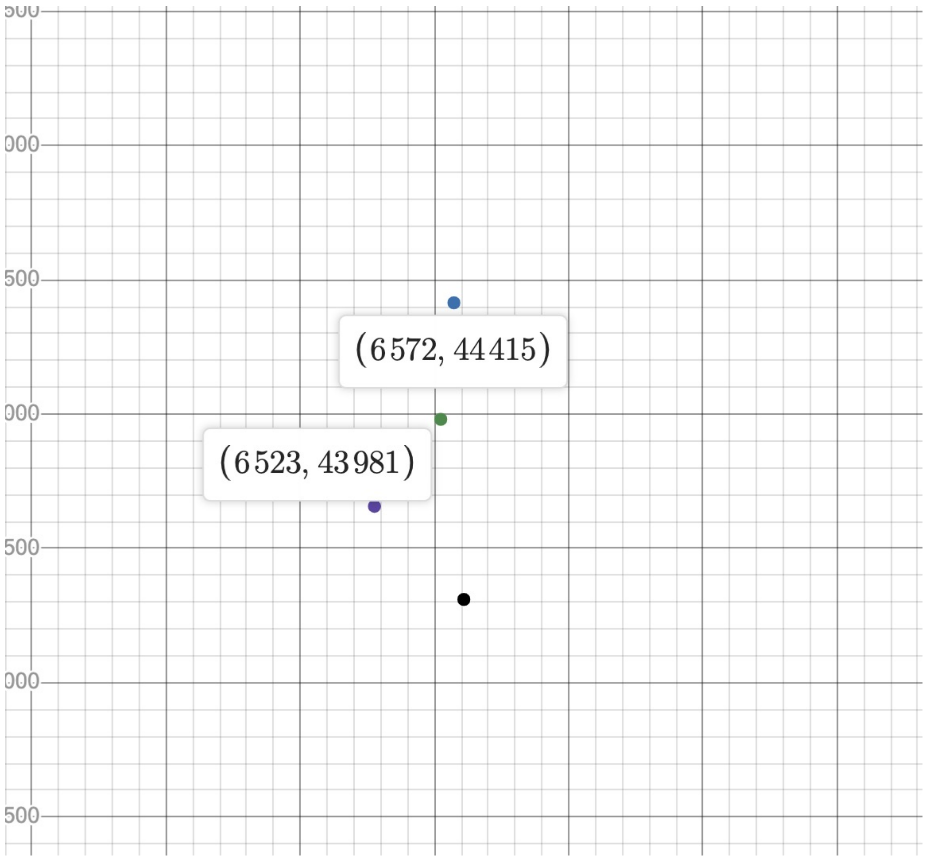


Here is 1000 points visualized in Desmos graphing calculator. Every point is randomly generated for this test. The x and y values have a limit of 50000.



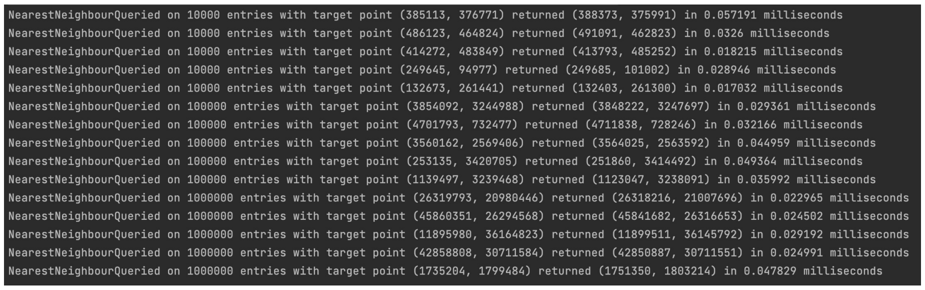
```
--
Building tree with 1000 points
Built tree in 13.79037 milliseconds
--
Finding nearest neighbour to 6572.0, 44415.0
Nearest Neighbour to: 6572.0, 44415.0: 6523.0, 43981.0
--
Found nearest neighbour in: 0.064102 milliseconds
```

Here is the output of the code. It is finding the nearest neighbour to the point $(6572, 44415)$ and produces the point from the data set: $(6523, 43981)$.



Zooming in on the Desmos graph, you can see the blue test point (added in separately from all the points in the data set) and the green nearest neighbour point. The values match the code output and there is clearly no closer point to the blue point than the green one.

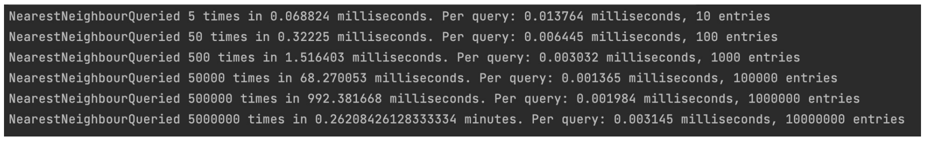
For larger data sets, I had the program output the target point and the nearest neighbour it found. Based on previous confirmed results I was able to determine whether the outputted nearest neighbour was close enough to the target point to confirm it was definitely the nearest neighbour. These results are shown below:



```
NearestNeighbourQueried on 10000 entries with target point (385113, 376771) returned (388373, 375991) in 0.057191 milliseconds
NearestNeighbourQueried on 10000 entries with target point (486123, 464824) returned (491091, 462823) in 0.0326 milliseconds
NearestNeighbourQueried on 10000 entries with target point (414272, 483849) returned (413793, 485252) in 0.018215 milliseconds
NearestNeighbourQueried on 10000 entries with target point (249645, 94977) returned (249685, 101002) in 0.028946 milliseconds
NearestNeighbourQueried on 10000 entries with target point (132673, 261441) returned (132403, 261300) in 0.017032 milliseconds
NearestNeighbourQueried on 100000 entries with target point (3854092, 3244988) returned (3848222, 3247697) in 0.029361 milliseconds
NearestNeighbourQueried on 100000 entries with target point (4701793, 732477) returned (4711838, 728246) in 0.032166 milliseconds
NearestNeighbourQueried on 100000 entries with target point (3560162, 2569406) returned (3564025, 2563592) in 0.044959 milliseconds
NearestNeighbourQueried on 100000 entries with target point (253135, 3420705) returned (251860, 3414492) in 0.049364 milliseconds
NearestNeighbourQueried on 100000 entries with target point (1139497, 3239468) returned (1123047, 3238091) in 0.035992 milliseconds
NearestNeighbourQueried on 1000000 entries with target point (26319793, 20980446) returned (26318216, 21007696) in 0.022965 milliseconds
NearestNeighbourQueried on 1000000 entries with target point (45860351, 26294568) returned (45841682, 26316653) in 0.024502 milliseconds
NearestNeighbourQueried on 1000000 entries with target point (11895980, 36164823) returned (11899511, 36145792) in 0.029192 milliseconds
NearestNeighbourQueried on 1000000 entries with target point (42858808, 30711584) returned (42850887, 30711551) in 0.024991 milliseconds
NearestNeighbourQueried on 1000000 entries with target point (1735204, 1799484) returned (1751350, 1803214) in 0.047829 milliseconds
```

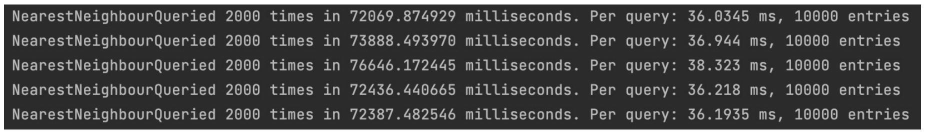
The corresponding nearest neighbour values are close enough to the target points that I could confirm the nearest neighbour search worked for larger data sets as well.

I then ran several tests on varying sizes of inputs to get an average time complexity of executing the nearest neighbour search algorithm on a K-d Tree. I ran $n/2$ queries for each input size of size n. These results are shown below.



```
NearestNeighbourQueried 5 times in 0.068824 milliseconds. Per query: 0.013764 milliseconds, 10 entries
NearestNeighbourQueried 50 times in 0.32225 milliseconds. Per query: 0.006445 milliseconds, 100 entries
NearestNeighbourQueried 500 times in 1.516403 milliseconds. Per query: 0.003032 milliseconds, 1000 entries
NearestNeighbourQueried 50000 times in 68.270053 milliseconds. Per query: 0.001365 milliseconds, 100000 entries
NearestNeighbourQueried 500000 times in 992.381668 milliseconds. Per query: 0.001984 milliseconds, 1000000 entries
NearestNeighbourQueried 5000000 times in 0.26208426128333334 minutes. Per query: 0.003145 milliseconds, 10000000 entries
```

The results for nearest neighbour search using a K-d tree were quick regardless of the size of the data set. Each query took < 0.1ms and most of the time was less than 0.01ms. However, that was not the case with R-tree. For smaller data sets, R-Tree performed very well but as the size of the data set increased to 10000 entries, each query was taking > 35ms. Because of this I chose to limit my input size to 10000 entries and limited the number of queries to 2000. The results are shown below.



```
NearestNeighbourQueried 2000 times in 72069.874929 milliseconds. Per query: 36.0345 ms, 10000 entries
NearestNeighbourQueried 2000 times in 73888.493970 milliseconds. Per query: 36.944 ms, 10000 entries
NearestNeighbourQueried 2000 times in 76646.172445 milliseconds. Per query: 38.323 ms, 10000 entries
NearestNeighbourQueried 2000 times in 72436.440665 milliseconds. Per query: 36.218 ms, 10000 entries
NearestNeighbourQueried 2000 times in 72387.482546 milliseconds. Per query: 36.1935 ms, 10000 entries
```

5.3 Challenges and Solutions

Two major challenges I faced were correctly implementing the nearest neighbour algorithm and time constraints. The resources available online are not well established for implementing complex data structures. It is even hard to find what these data structures are capable of doing, and most of the existing nearest neighbour implementation guides on the first page of Google are just plain wrong. Because of this, I had a hard time making everything work and with such lofty goals from the start I unfortunately had to limit my expectations for this project. I wished I had more time, I would have loved to have incorporated other data structures and spatial query algorithms. However, my professor did mention during my project proposal that my initial goals were quite lofty and I was better off limiting the number of data structures, I just didn't expect I'd have to limit myself this much. This is certainly a project I am going to develop more outside of school as I am very interested and would like to see how much I can implement on my own.

6 Code

Building the K-d Tree

```java
public static KD_Tree build(ArrayList<Point> points) {
    Node root = buildTree(points, 0);
    return new KD_Tree(root);
}

private static Node buildTree(List<Point> points, int depth) {
    // base case
    if (points.isEmpty())
        return null;

    // use the depth to update the coordinate we are comparing
    int axis = depth % points.get(0).coordinates.size();
    // sort the points by their coordinate[axis]
    points.sort((a, b) ->
        a.coordinates.get(axis).compareTo(b.coordinates.get(axis)));

    // find median
    int medianIndex = points.size() / 2;
    Point medianPoint = points.get(medianIndex);

    // split points by median
    List<Point> leftPoints = points.subList(0, medianIndex);
    List<Point> rightPoints = points.subList(medianIndex + 1,
        points.size());

    // build left tree and right tree
    Node leftChild = buildTree(leftPoints, depth + 1);
    Node rightChild = buildTree(rightPoints, depth + 1);

    return new Node(medianPoint, leftChild, rightChild);
}
```

Nearest neighbour in K-d Tree

```java
public Point nearestNeighbour(Point point) {
    Point closest = new Point(null);
    double[] minDist = {Double.MAX_VALUE};

    nearestNeighbour(root, point, closest, minDist, 0);

    return closest;
}

private void nearestNeighbour(Node root, Point point, Point closest,
     double[] minDist, int depth) {
    // base case
    if (root == null)
        return;

    // check the distance and compare to the current min distance
    double dist = distance(root.getPoint(), point);
    if (dist < minDist[0]) {
        closest.setPoint(root.getPoint());
        minDist[0] = dist;
    }

    // use the depth to update the coordinate we are comparing
    // see how K-d trees levels are structured
    int axis = depth % point.coordinates.size();
    double axisDist = point.coordinates.get(axis) -
         root.getPoint().coordinates.get(axis);

    // update closer and further child
    Node closerChild = (axisDist < 0) ? root.l : root.r;
    Node fartherChild = (axisDist < 0) ? root.r : root.l;

    // recursion
    nearestNeighbour(closerChild, point, closest, minDist, depth +
         1);

    // check other side if required
    if (axisDist * axisDist < minDist[0]) {
        nearestNeighbour(fartherChild, point, closest, minDist, depth
             + 1);
    }
}
```

Adding to R-Tree

```
private void add(Rectangle r, int id) {
    add(r.x1, r.y1, r.x2, r.y2, id, 1);
    size++;
}

private void add(float x1, float y1, float x2, float y2, int id, int
     level) {
    // find place to add new node
    Node node = findNode(x1, y1, x2, y2, level);
    Node leaf = null;

    // check if leaf has room for another entry
    // otherwise splite node
    if (node.countEntries() < maxNodeEntries) {
        node.addEntry(x1, y1, x2, y2, id);
    } else {
        leaf = splitNode(node, x1, y1, x2, y2, id);
    }

    Node newNode = adjustTree(node, leaf);

    // if root split, create new root with children being the
         resulting nodes
    if (newNode != null) {
        int oldRootNodeId = rootNodeId;
        Node oldRoot = getNode(oldRootNodeId);

        rootNodeId = getNextNodeId();
        treeHeight++;

        Node root = new Node(rootNodeId, treeHeight, maxNodeEntries);
        root.addEntry(newNode.x1, newNode.y1, newNode.x2, newNode.y2,
            newNode.nodeId);
        root.addEntry(oldRoot.x1, oldRoot.y1, oldRoot.x2, oldRoot.y2,
            oldRoot.nodeId);
    }
}
```

Nearest neighbour in R-Tree

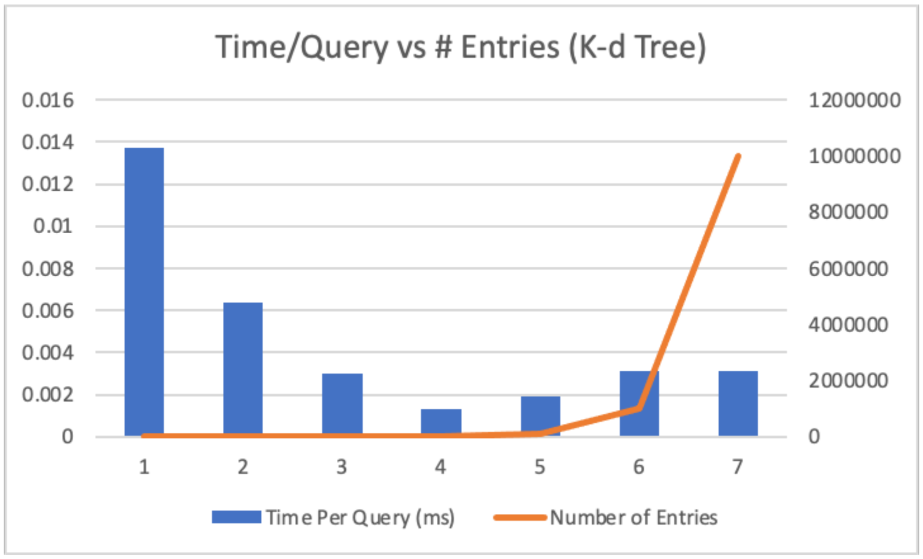
```
private float nearestNeighbour(Point target, Node node, float
    furthestDistanceSquared, ArrayList<Integer> nearestIds) {
    // for each entry in the current node
    for (int i = 0; i < node.countEntries(); i++) {
        // calculate distance from node entry to target
        float tempDistanceSquared =
            Rectangle.distanceSq(node.entriesX1[i],
            node.entriesY1[i], node.entriesX2[i], node.entriesY2[i],
            target.x, target.y);

        if (node.isLeaf()) {
            if (tempDistanceSquared < furthestDistanceSquared) {
                furthestDistanceSquared = tempDistanceSquared;
            }
            if (tempDistanceSquared <= furthestDistanceSquared) {
                nearestIds.add(node.ids[i]);
            }
        } else {
            if (tempDistanceSquared <= furthestDistanceSquared) {
                furthestDistanceSquared = nearest(target,
                    getNode(node.ids[i]), furthestDistanceSquared,
                    nearestIds);
            }
        }

        return furthestDistanceSquared;
    }
}
```

7 Results

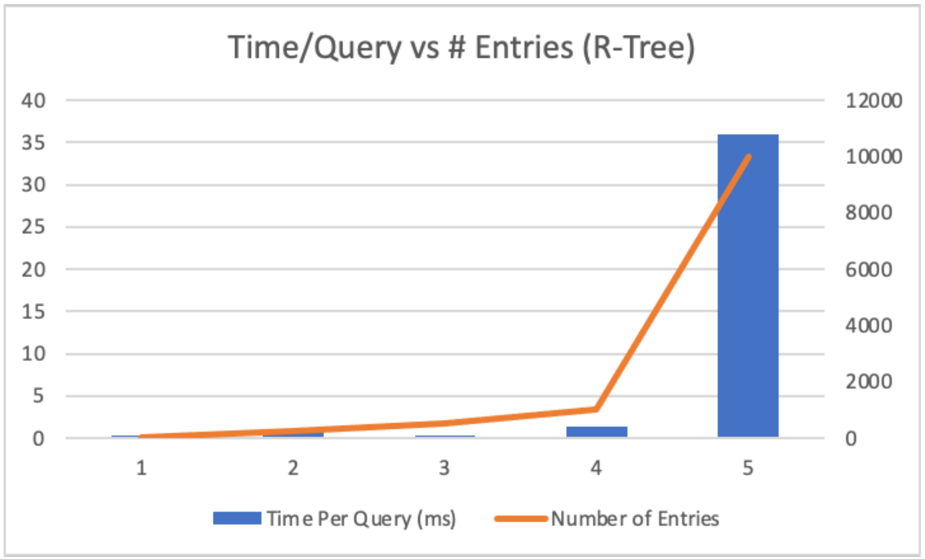
7.1 Comparative Analysis of Results

Comparing the results of both the K-d tree and the R-tree, it is easy to see that the K-d tree fairs much better when the data set is larger. The following graphs allow us to visualize the results quickly and compare the increase in entries to the time per query for both data structures individually.

7.2 Visualization of Results



Above, the effect is actually less optimal when the number of entries is small and becomes more optimal as the number of entries gets larger.



Above, the effect is the opposite where the R-Tree is most effective when the number of entries is the smallest and becomes increasingly most costly as the number of entries gets larger.

7.3 Key Findings

Though they both execute region queries on spatial data and both are trees, R-Trees and K-d Trees are quite different. The main differences between the two are:

R-Trees

Features	**Details**
Balanced	Suitable for inserting, altering, and removing data
Disk-oriented	Organizes data to visually map to the on-disk representation
Coverage	Do not cover the whole data space so empty areas may be uncovered
Data Partitioning	Partition data into rectangles, overlap is minimized
Optimization Strategies	Splitting, bulk loading, insertion and re-insertion
Distance Calculation	D-dimensional minimum distance to bounding rectangles
Best Query	Range queries: find all objects that intersect with a given rectangular region

K-d Trees

Features	**Details**
Balanced	Only when data is loaded in bulk
Memory-oriented	Easier to implement in memory
Coverage	Always cover the whole data space
Data Partitioning	Binary split data space, results in disjointed splits
Optimization Strategies	Splitting, bulk loading, insertion and re-insertion
Distance Calculation	One-dimensional distance calculation; Euclidean distance and Minkowski norms
Best Query	Nearest neighbour query: find nearest neighbour to a target point

8 Discussion

8.1 Interpretation of Results

The comparative analysis between R-Trees and K-d Trees displays a distinct advantage with one over the other when it comes to executing nearest neighbour queries. The K-d tree excels at these queries thanks to their binary splitting. The balanced nature of R-Trees makes them better suited for inserting, altering, and deleting data whereas K-d Trees are better suited for static data that is loaded in bulk. The disk-oriented design of R-Trees makes them better suited for applications in real databases and the memory-oriented design of K-d Trees makes them better suited when memory is a constraint.

8.2 Implications

The findings from this comparative study has several implications for spatial data usage in GIS. Developers and corporations can make educated decisions when choosing what data structure is appropiate for their specific application. Do they have lot's of storage space and dynamic data? R-Trees are better suited in this scenario. Do they want to have quick nearest neighbour searches and even N-nearest neighbour searches on data that won't change? K-d Trees are better suited in this scenario.

8.3 Limitations

There are limitations for both R-Trees and K-d Trees. The performance and efficiencies of these data structures highlighted in this paper may not be universal. Results will vary based on specific implementation, optimization, and the dataset used. This paper also focused mostly on the functional differences between both data structures and did not delve into the differences in complexity of understanding and implementing these data structures. The complexity of implementation can also be a factor on someone using them in a specific project as they may not be worth it to implement.

9 Conclusion

In conclusion, both R-Trees and K-d Trees have specific strengths and weaknesses that make them better suited for different types of applications, despite them both being initially setup similarly (as discussed in section 7.3. In terms of executing nearest neighbour queries, A K-d Tree implementation is the better option. Static data sets are best for K-d Trees and dynamic data sets are best for R-Trees. The choice between the two depends on your specific application and this paper has hopefully allowed you to get a better understanding of when you might use a K-d Tree over an R-Tree and vice versa.

10 Resources

naturalearthdata.com
docs.oracle.com/javase/8/docs/api/
docs.oracle.com/javase/7/docs/api/javax/swing/package-summary.html

11 References

[1] "What Is GIS?" Geographic Information System Mapping Technology, Esri, www.esri.com/en-us/what-is-gis/overview.

[2] "The Different Types of GIS Data." MGISS, 5 Oct. 2023, mgiss.co.uk/the-different-types-of-gis-data/.

[3] "What Is a TIN Surface?" ArcGIS, Esri, desktop.arcgis.com/en/arcmap/latest/manage-data/tin/fundamentals-of-tin-surfaces.htm.

[4] "GIS Dictionary." Tree Data Structure Definition, Esri, support.esri.com/en-us/gis-dictionary/tree-data-structure.

[5] Samet, Hanan. The Design and Analysis of Spatial Data Structures. Addison - Wesley Publishing Company, Inc., Jan. 1994, https://cdn.preterhuman.net/texts/math/Data_Structure_And_Algorithms/The%20Design%20And%20Analysis%20Of%20Spatial%20Data%20Structures%20-%20Hanan%20Samet.pdf.

[6] Gaede , Volker, and Oliver Günther. Multidimensional Access Methods. June 1998, https://www.csd.uoc.gr/~hy460/pdf/volker.pdf.

[7] Lagana, Celeste. "Examples and Uses of GIS." IBM, 18 Dec. 2023, www.ibm.com/blog/geographic-information-system-use-cases/.

[8] Goodchild, Michael. Reimagining the History of GIS. 2018, https://www.tandfonline.com/doi/full/10.1080/19475683.2018.1424737#:~:text=The%20history%20of%20geographic%20information,the%20guidance%20of%20Roger%20Tomlinson.

[9] "KD-Trees." Carnegie Mellon University, www.cs.cmu.edu/~ckingsf/bioinfo-lectures/kdtrees.pdf.

[10] "R-Tree." Simon Fraser University, www2.cs.sfu.ca/CourseCentral/454/jpei/slides/R-Tree.pdf.

[11] "R-Tree." Wikipedia, Wikimedia Foundation, 30 Dec. 2023, en.wikipedia.org/wiki/R-tree.

[12] “K-D Tree.” Wikipedia, Wikimedia Foundation, 16 Oct. 2023, `en.wikipedia.org/wiki/K-d_tree`.