Algorithms and Analysis Assignment 1 Report

Andrew Griffiths - s3458129 and Siyu Zang - s3534987
April 15, 2017

1. Introduction

This report covers the analysis for the nearest neighbour problem using two different approaches. The naive implementation and the KD tree implementation.

Both implementations will have a number of operations performed on them and compared to see which implementation is the best for a given scenario.

2. Setup

2.1 How data generated

For both scenarios, the longitude and latitude were randomly generated. No duplicate coordinates were allowed as in most cases there will only be one hospital, school etc. at a certain longitude and latitude. While it is possible you can get multiple restaurants in one place for example a food court it was decided that seeing as the purpose of this assignment is to compare two different data structures and algorithms there was no point taking this into consideration.

K is a randomly generated number between one and ten. This was done to provide a more real world analysis of the two approaches to the nearest neighbour problem.

2.2 How timing result were measured

The timing for the analysis was done by adding the following code to NearestNeighborFileBased.java.

long startTime;
long endTime;
double estTime;

startTime was started in the try block where the data operations are being parsed. The *endTime* was placed at the end of this code block.

The timing code was placed here as the intention was to only benchmark the amount of time performing operations took for each approach.

For each test in scenario one and two the tests were run on all three core teaching servers started at a different time of day and the results were averaged.

The categories were randomly assigned to each point. The naive implementation uses an *ArrayList* to store data.

3. Design of Evaluation

3.1 Scenario 1 (k-nearest neighbour searches)

For scenario one there are eight different sets of initial points to start with. The following number of points were generated.

- 20000 points
- 50000 points
- 100000 points
- 300000 points

For each of the above data sets 50% of each one was randomly selected to be searched. Two data sets were searched against. One data set was searched for points that should exist and the other data set was searched for points that may or may not exist.

3.2 Scenario 2 (Dynamic points set)

The number of initial points for the data sets in scenario two are the same as scenario one. For each data set there were 50% add operations and 50% delete operations added to the data structure. After that 50% of the initial data set size is then searched. The add and delete operations were for data that may or may not exist.

For example. Twenty thousand randomly generated points are added to the data structure. Ten thousand add operations and ten thousand delete operations are then added randomly to the data structure. After that a random ten thousand search operations are performed on points that may or may not exist.

This was done as it most reflects how these data structures and algorithms would be used in the real world.

4. Analysis

4.1 Scenario 1 (k-nearest neighbour searches)

4.1.1 20,000 points with 10,000 searches for data that should exist.

Approach	**Time (Nanoseconds)**
Naive	50864691226.3

KDTree	57441447656.7

In this scenario, the naive approach is the fastest by 88.5%. While naive is the fastest in this case the amount of time each approach took is pretty much the same in the real world. Because nanoseconds were used to measure the time taken it makes the naive implementation look better than it is. We would expect that as the size of the data set increases the KDTree approach will outperform the naive approach.

4.1.2 20,000 points with 10,000 searches for data that may or may not exist.

Approach	**Time (Nanoseconds)**
Naive	50807829322.0
KDTree	65602566559.0

Again, the naive approach is faster but this time only with a 77.44% performance difference with the KDTree approach. Again, in the real world the time difference isn't significant. The approximately 10% difference between the known data and random could potentially be because at the time of running the previous test one or more of the core teaching servers may have been experiencing a higher workload than normal.

4.1.3 50,000 points with 25,000 searches for data that should exist.

Approach	**Time (Nanoseconds)**
Naive	3.13302158075e+11
KDTree	3.52756469068e+11

Surprisingly the naive approach is the fastest again. Even with more than twice the data to search through the percentage difference in performance remains fairly static at 88.8%. I would have expected the KDTree approach to be faster as in theory tree based data structures are supposed to be faster than arrays. One possible reason for the naive implementation being faster is that while iterating through the array list it checks to see if the category of the current index being checked matches the category of the points being searched for before checking the distance, add that to the fact that arrays are very quick to iterate through, it would make the naive implementation very efficient until the data set grows. Whereas the KDTree implementation performs the distance calculations which would slow it down a bit.

4.1.4 50,000 points with 25,000 searches for data that may or may not exist.

Approach	Time (Nanoseconds)
Naive	3.15734827633e+11
KDTree	3.54648267771e+11

It looks like searching data that may or may not exist has no discernible effect on performance. The difference in performance stayed fairly stable at 89.0%. Not much else to say on this one as the observations and explanation are pretty much the same as the previous test.

4.1.5 100,000 points with 50,000 searches on data that should exist

Approach	Time (Nanoseconds)
Naive	1.24435192252e+12
KDTree	1.08491461735e+12

In this test, the KDTree approach is the fastest. The performance difference between the two implementations is 114.69%. This looks like it will be the point in which the naive implementation will have much slower performance compared to the KDTree. This is because as the data set grows the amount of time it takes to search through the entire array also grows. For tree based data structures the amount of data is constantly being reduced as it traverses down the branches of a tree.

4.1.6 100,000 points with 50,000 searches on data that may or may not exist

Approach	Time (Nanoseconds)
Naive	1.24120144483e+12
KDTree	1.0791917812e+12

The results again are consistent with the searches against data that should exist.
KDTree is the fastest again with a 115% percent performance difference.

4.1.7 300,000 points with 150,000 searches on data that should exist

Approach	Time (Nanoseconds)
Naive	1.13064837016e+13
KDTree	4.3957306971e+12

KDTree is the fastest in this test. The KDTree implementation is 257.21% faster than the naive implementation. At this size, the efficiency of a tree based approach makes a huge difference to performance when it comes to searching for the reasons mentioned in the 100,000-data set analysis.

4.1.8 300,00 points with 150,000 searches on data that may or may not exist

Approach	**Time (Nanoseconds)**
Naive	1.12801412585e+13
KDTree	4.29584166479e+12

Again, KDTree is the fastest implementation. The percentage difference in performance remains fairly stable at 262.58%. Searching for data that may or may not exist seems to have no major impact on performance.

4.2 SCENARIO 2 (Dynamic points set)

4.2.1 20,000 points with 20,000 random add and delete operations with 10,000 searches

Approach	**Time (Nanoseconds)**
Naive	78583870090.7
KDTree	77977385956.3

The KDTree implementation is the fastest by a 100.77% difference in performance.
By adding add and delete operations the efficiency of the tree structure makes a difference and the performance gap between the two implementations will keep growing as the data set size and amount of operations grows. This is because arrays have O(n) performance for additions, deletions and search whereas a KDTree has O(log(n)) performance for addition, deletion and search operations.

4.2.2 50,000 points with 50,000 random add and delete operations with 25,000 searches

Approach	**Time (Nanoseconds)**
Naive	4.85023881502e+11
KDTree	4.34176074841e+11

Again, the KDTree implementation is faster by 111.71 percent. The performance of the KDTree grew as expected.

4.2.3 100,000 points with 100,000 random add and delete operations with 50,000 searches.

Approach	**Time (Nanoseconds)**
Naive	1.95309809824e+12
KDTree	1.24350482493e+12

KDTree is the fastest implementation again. As predicted the performance increase between the naive implementation and the KDTree implementations continues to grow. For this test, the performance increase is 157.06 percent.

4.2.4 300,000 points with 300,000 random add and delete operations with 150,000 searches.

Approach	**Time (Nanoseconds)**
Naive	1.84703074967e+13
KDTree	4.82237858882e+12

For the final test KDTree is the fastest again. There was a significant difference between the performance of the naive and KDTree implementations. The percentage difference between the two is 383.01%

5. Recommendation

5.1 Scenario 1 (k-nearest neighbour searches)

For scenario one based on the results for those tests I would recommend the naive implementation for datasets of 50,000 points or less. The reason is that the naive implementation is much easier to understand and therefore easier to maintain or troubleshoot. For larger datasets however I would recommend the KDTree implementation simply for the better performance with larger data sets.

5.2 Scenario 2 (Dynamic points set)

For scenario two I would recommend the KDTree implementation especially for large datasets as the performance gains over the naive implementation are significant.