Algorithms and Analysis Assignment 1 Comparison between Naïve and KDTree approach

By S3557766

Experimental Setup

Data Generation and Parameter Settings

In the attempt to generate the data for the experiment, we have decided to use the built-in java randomizer which can be imported via *java.util.Random*. On top of that, to accurately reflect the practical implementations of the program, we decided to stay within the upper and lower bounds for both longitude and latitude given to us in sampleData.txt. This meant that the data generated are float numbers with nine to ten decimal places. Each generated data point was also given a unique ID and category and is written to a text file in a format identical to the one that sampleData.txt adheres to. During the randomization of values, I had a built-in check which made sure that there were no duplicates in value. This was done for the generation of both latitude and longitude which will hopefully make sure that all the points are as unique as possible.

Scenario Generation for Scenario 1 (k-nearest neighbor searches)

To tackle this scenario, we decided to generate 3 data sets with 1000, 5000 and 25000 points. With the 3 data sets, we decided to incrementally increase the amount 'k' (number of nearest neighbors). It is important to note that the following searches were done straight after the construction for the ArrayList (Naïve approach) and KDTree (KDTree approach). This meant that the integrity of both indices should be at its most optimal point. The following table tabulates the different 'k' values we decided to test on and the reason we went with 20 different points for each data set is to get as close to an accurate representation of overall trend.

	1000 Data Points	5000 Data Points	25000 Data Points
Test 1	1	1	1
Test 2	10	50	250
Test 3	20	100	500
Test 4	30	150	750
Test 5	40	200	1000
Test 6	50	250	1250
Test 7	60	300	1500
Test 8	70	350	1750
Test 9	80	400	2000
Test 10	90	450	2250
Test 11	100	500	2500
Test 12	200	1000	5000
Test 13	300	1500	7500
Test 14	400	2000	10000
Test 15	500	2500	12500
Test 16	600	3000	15000
Test 17	700	3500	17500
Test 18	800	4000	20000
Test 19	900	4500	22500
Test 20	1000	5000	25000

The reason we decided to test the whole spectrum was due to the way the program was designed which started with the pre-population of potential closest points in an array. This means that at some point, as the value of ‘k’ approaches the number of points in the data set, the search time should theoretically be shorter for both the Naïve and KDTree approaches. From the test ran above, we can also derive search times with varying number of initial data points but with a constant ‘k’ (i.e. Test 1).

Scenario Generation for Scenario 2 (Dynamic Points Set)

For Scenario 2, we decided to generate 5 different data sets with 250, 500, 1000, 5000 and 25000 data points. A constant addition and deletion is made over all the data sets. For this experiment, we decided to have 1000 additions and a 1000 deletions. However, before any changes were done to the data, we ran identical tests to scenario 1 which served as a baseline comparison to the results we got after the integrity of the KDTree has been compromised by the additions and deletions. This meant that we are effectively doing a before and after comparison of the search time. The tabulated data below shows the sampling points we decided to go with for the value of ‘k’.

	250 Data Points	500 Data Points	1000 Data Points	5000 Data Points	25000 Data Points
Test 1	1	1	1	1	1
Test 2	25	5	10	50	250
Test 3	38	10	20	100	500
Test 4	50	15	30	150	750
Test 5	63	20	40	200	1000
Test 6	75	25	50	250	1250
Test 7	88	30	60	300	1500
Test 8	100	35	70	350	1750
Test 9	113	40	80	400	2000
Test 10	125	45	90	450	2250
Test 11	138	50	100	500	2500
Test 12	150	100	200	1000	5000
Test 13	163	150	300	1500	7500
Test 14	175	200	400	2000	10000
Test 15	188	250	500	2500	12500
Test 16	200	300	600	3000	15000
Test 17	213	350	700	3500	17500
Test 18	225	400	800	4000	20000
Test 19	238	450	900	4500	22500
Test 20	250	500	1000	5000	25000

Timing

In the Naïve and KDTree approach for both Scenario 1 and 2, to test the time it took for the search algorithm to return an answer, we went with the *System.nanoTime()* function. In our test cases, the time starts from the initial population of the array list of points and ends just before the refined array list is returned by the function. This is done in hopes that the time captured accurately reflects the time it took just to search and refine the query results and not the time taken to construct the data structure or compile the program.

Testing Frequency

For Scenario 1, each test was repeated 20 times for each chosen value of ‘k’ with 20 different ‘k’ values multiplied by the number of data sets. Cumulatively, this would have given us results for 3200 individual test as we did the same thing for the Naïve and KdTree approach.

For Scenario 2, we were only interested in evaluating the performance of the KDTree but at the same time, we have increased our data sets to 5 while everything else remains the same as Scenario 1. Cumulatively. This would have given us results for 2000 individual test.

Evaluation

Scenario 1

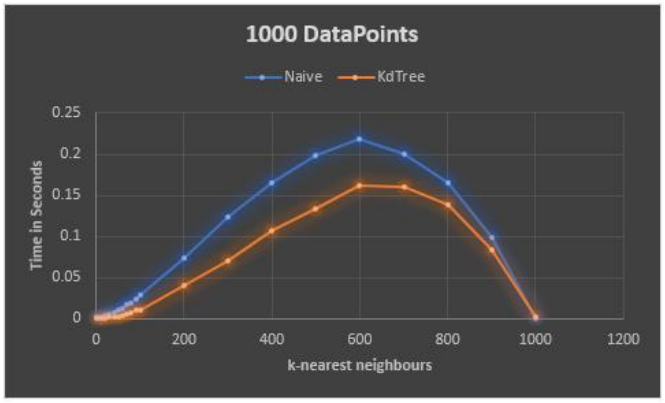


Figure 1

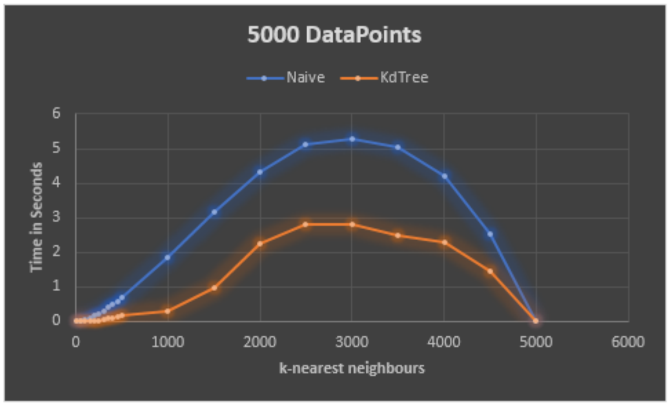


Figure 2

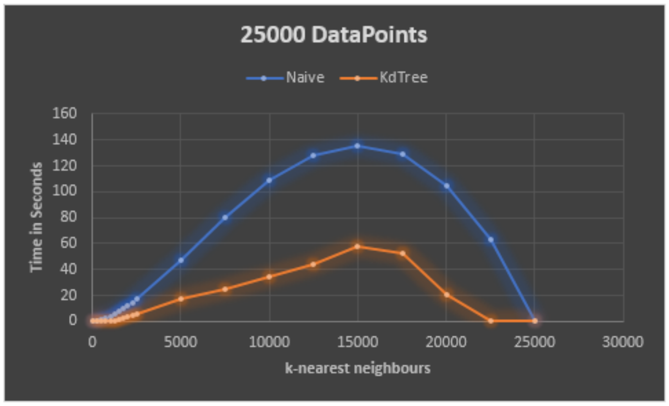


Figure 3

From the data generated by the tests, we found that as the number of data points increases, the Naïve approach suffers heavily in terms of scalability. As you can see from Figures 1, 2 and 3, the spread between the Naïve and KDTree approach grew as we increased the amount of data points. Furthermore, we can see that the trend for the Naïve approach has more of a smooth bell shape compared to the KDTree approach even though the KDTree approach is much more superior when it comes to average search time over the same data set. This variation may be due to the anchoring point for the search query.

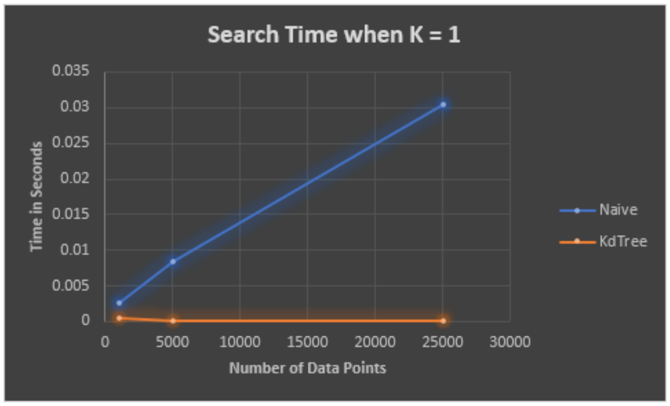


Figure 4

To further enforce my findings, we decided to have a look at the search times when ‘k’ is held constant in all 3 data sets. In this case, we decided to hold the value of ‘k’ at 1 and look at the search time when the query ran against the data sets of sizes 1000, 5000 and 25000. From the graph, we can see that the growth rate is somewhat linear for the Naïve approach while the KDTree approach saw close to constant search time. From our understanding of the theory, the KDTree approach does not reflect a constant search time algorithm. Therefore, it is only logical to deduct that the spread in search time for the Naïve and KDTree approach is so large that it distorted the graphical represented growth time. That aside, it is clear to see that the growth rate in search time length for the KDTree approach is far superior (in terms of slower growth) compared to the Naïve approach which makes it a very attractive algorithm to implement as the size of the data set increases.

Scenario 2

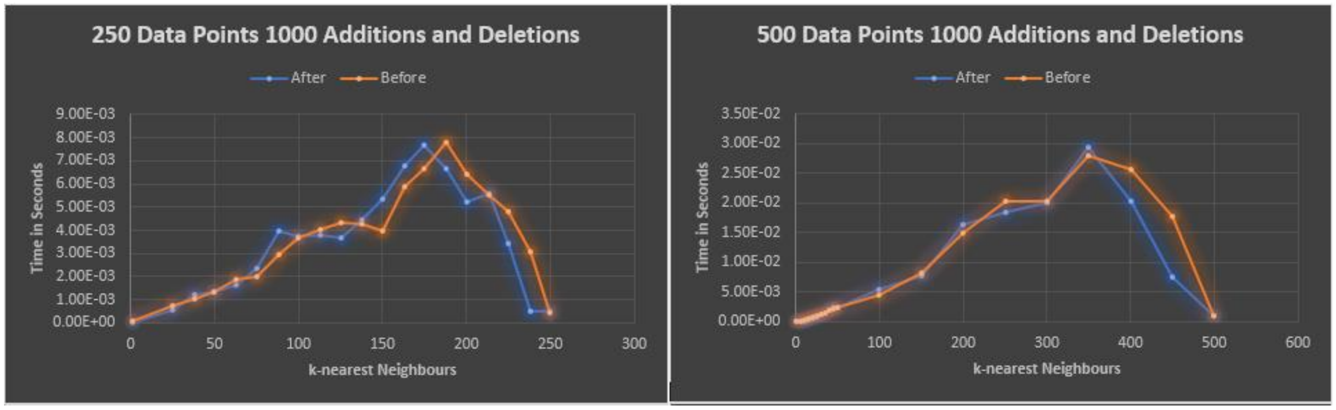
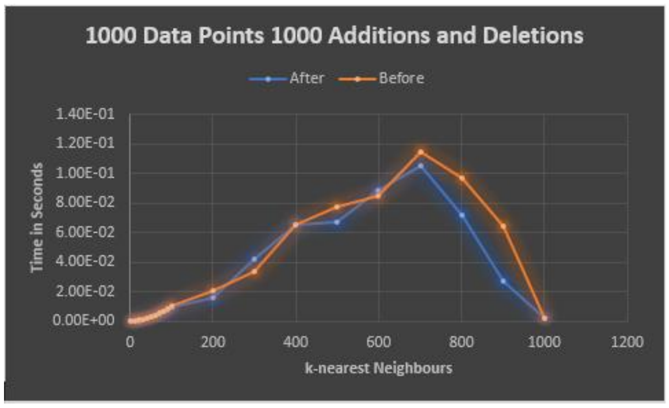


Figure 5 and Figure 6



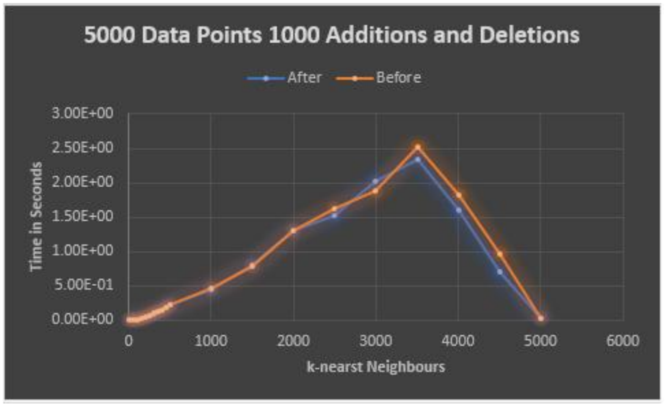


Figure 7 and Figure 8

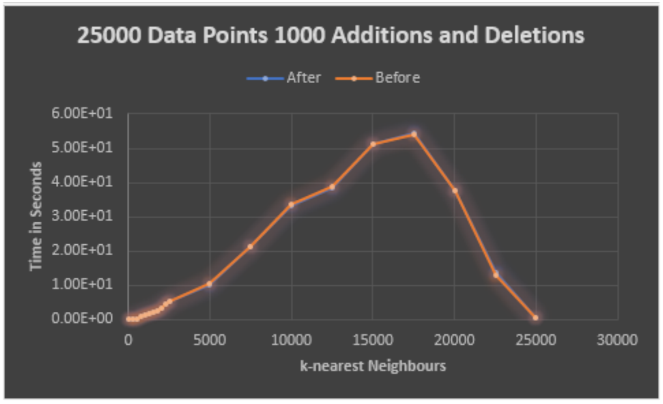


Figure 9

Just looking at the figures above, we can see that as we move from Figure 5 to Figure 9, the destabilization and deterioration when it comes to search time become less and less pronounced especially if we focus on the first half of the graph. Furthermore, it is important to note that due to the lack of tree balancing after every addition and deletion, it should be expected that the search time should never be better than its respective baseline reference. This assumes that the tree is at its most balanced state during the initial indexing phase.

However, the statement above contradicts the results we see on the second half of most of the graphs which tend to show an improvement in search times. One reason that may explain this phenomenon is that the tree may have a very long branch which negates the need to run further checks and computation cycles on whether both the child node has been visited already during the pre-population of the initial array list of results as well as during the reverse traversal of the tree as we return to the root of the tree. The reduction in computation required may very well result in a better search time as 'k' approaches the size of the initial data set.

Moving on to the decrease in deterioration as the initial data points increase; this was an expected result due to the way the scenario was designed. The alteration to the KDTree in terms of percentage decreases as we increase the size of the data points as shown in the dot points below.

- 250 Data Points with 1000 additions and deletions represents an alteration of 400%
- 500 Data Points with 1000 additions and deletions represents an alteration of 200%
- 1000 Data Points with 1000 additions and deletions represents an alteration of 100%
- 5000 Data Points with 1000 additions and deletions represents an alteration of 20%
- 25000 Data Points with 1000 additions and deletions represents an alteration of 4%

This is because we were holding the number of deletions and additions constant at 1000 each across the board. With that in mind, and with the results generated from this experiment, we concluded that the performance of the KDTree approach becomes more and more distorted as a higher portion (percentage) of the datasets are being added and removed in relation to the initial size of the data set.

This is most probably due to the binary tree being unbalanced after each addition and deletion since we were not expected to keep the tree balance when changes were made during the design stage. It is also important to note that the search point used in the experiments were held constant and the performance before and after the changes were made depends heavily on whether the subtree housing those points were altered during the addition and deletion phase of our experiment.

Recommendation

Usage of the Naïve Approach

Considering the effort, it took to implement both types of algorithms, and the results we obtained from the experiments conducted above, we felt that the Naïve approach should only be used when we are interested in managing very small amounts of data points (i.e. less than 1000 points). Even with the lack of re-balancing for the KDTree, the average performance of the KDTree should be superior to the Naïve approach unless the KDTree became so unbalanced that it is effectively a very long branch. If that were to happen, the performance will be effectively be the same as the Naïve approach. However, if there is a very high frequency of addition and deletion done towards the KDTree which transforms the tree into a single long branch, we would highly recommend the Naïve approach just because of the ease of implementation compared to the KDTree approach. Furthermore, if searching is done infrequently, it may very well be sufficient to just implement the Naïve approach.

Usage of the KDTree Approach

In most scenarios, the KDTree approach is much more superior to the Naïve approach. This is apparent from the experiments conducted for Scenario 1. Due to its higher degree of scalability as the initial data set size increases, it will be wise to implement the KDTree approach when the data set is very large. Referring to the graphs in Scenario 1, the spread becomes wider when the number of points exceed 1000 points. As mentioned previously, even with the lack of re-balancing, the performance although it may become inconsistent, should not deteriorate to a level worse that the Naïve approach. Furthermore, if multiple searches are going to be done over the large data set, it will be wise to implement this approach as the time saved will be compounded over each search. The only drawback of this approach will be the effort it takes to implement it compared to the Naïve approach. However, to increase the robustness of this approach, I would recommend the implementation of some sort of re-balancing algorithm which checks if the tree is balances after every addition and deletion.