Algorithms and Analysis Assignment 1

Spencer Porteous s3539519
Jarod Wright s3601928

Brief Notes on Implementation

Both of our implementations, Naive and KD-Tree, use quicksort if required to sort our points.
Both of our implementations have 3 separate indexes for points, one for each category.
A class for the KD-Tree nodes was created, it is in the NearestNeigh package in a class named "KDNode".

Dataset Generation

Our dataset generation is located in the generation package as a class named "DataSetGeneration" the supplied function takes the upper left bound as {-37.656614, 144.759037} and the lower right bound as {-38.188780, 145.284321} (These are roughly the bounds of Melbourne). The function takes the amount of data points you want to generate as a function and uses the java.util.Random object to produce random points while also generating random categories for all.

Command Generation

Command generation is located in the generation package as a class named "CommandSetGeneration" it has the 3 scenario functions which use for loops to generate multiple tests. It also has 3 helper functions - Generate a random search between the same bounds as DataSetGeneration with the ability to either choose a k value or randomise a k value; generate a random addition command between the aforementioned bounds; and generate a random deletion command from a supplied dataset.

Measuring Time Taken

As both of us are using Windows computers without a unix/linux terminal Time command, we were unable to run our tests using a console based time command. We instead created a wrapper class in Java in the generation package, called "Testing". It repeatedly calls the NearestNeighFileBased main() with a range of test parameters, uses System.currentTimeMillis() to measure how long each execution takes, and outputs it to a static file named TestSummary.out . We have supplied this class and a TestSummary file in our submission for viewing. There is instructions in the class if you wish to use.

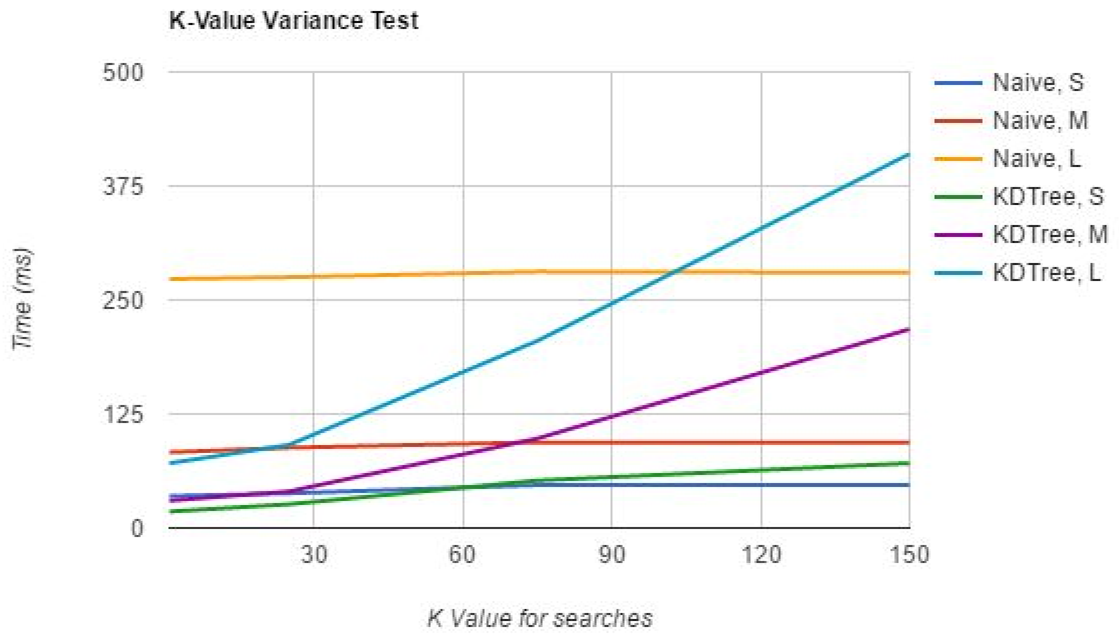
Scenario 1: Variance of K Value

For any value of k the KD-Tree implementation should run significantly faster than Naive. This is because the average complexity of a KD-Tree query is O(k logn) and the average complexity of the Naive approach (with Quicksort) is O(n logn). You can cancel out the logn on each side leaving k < n. k must be lower or equal to n therefore the KD-Tree approach should always be faster.

In this scenario 4 random tests are generated with 3 searches each, with random K values varying between 1 and 150. The 3 searches are respectively of each category (Restaurant, hospital and education). This test is then run on all 3 of the generated dataset sizes we are using using KD-Tree or Naive 10 times, and the time execution for those 10 runs is averaged.

KVariance Test (k value)	Time (ms)					
	Naive, S	Naive, M	Naive, L	KD-Tree, S	KD-Tree, M	KD-Tree, L
1	35	83	273	18	30	71
25	38	88	275	26	40	91
75	47	94	281	52	98	205
150	47	94	280	71	218	410

Note: S, M, and L mean small dataset (1000), medium dataset (2500) and large dataset (7500) respectively. Each test conducted 10 times to produce an average. Each test has 3 searches - one for each category to make up for the randomness of the dataset.



We got expected results in tests in which k = 1, the KD-Tree approach performed faster than the Naive approach, as an average case of O(k logn) is expected for a perfect KD-Tree implementation and O(n logn) for the Naive approach (because Naive uses a quicksort algorithm).

K N	
1 < 1000	KD-Tree should be faster, KD-Tree was faster
1 < 2500	KD-Tree should be faster, KD-Tree was faster
1 < 7500	KD-Tree should be faster, KD-Tree was faster

However on a more extreme end when executing tests where k = 150 the expected results were quite different to what was hypothesised. Cancelling out results in the KD-Tree's approach being faster for all values of k < n, which coincides with the fact that k cannot be larger with n. This however does not reflect the hypothesised results:

K N	
150 < 1000	KD-Tree should be faster, Naive was faster
150 < 2500	KD-Tree should be faster, Naive was faster
150 < 7500	KD-Tree should be faster, Naive was faster

For the implementations tested, the results from the test in which k = 75 and in which k = 150

indicate that over a dataset of 7500 points the KD-Tree approach becomes less efficient at ~100 points. We believe that the reason our hypothesis is incorrect is because of how we return the results. In our KDTree implementation, we have a method insertInOrder which places the result set in order from closest to furthest (this is to pass the Part A test). This function itself has a complexity of n, and it has to be called k times (one time for each result point). This leaves our KDTree with a complexity of O(k logn + nk) which explains why KDTree becomes slower than Naive after a certain point.
As far as lower k values go, our KD-Tree approach is superior to our Naive approach however as the k value grows in relation to the set size one should consider the Naive approach when using our implementation.

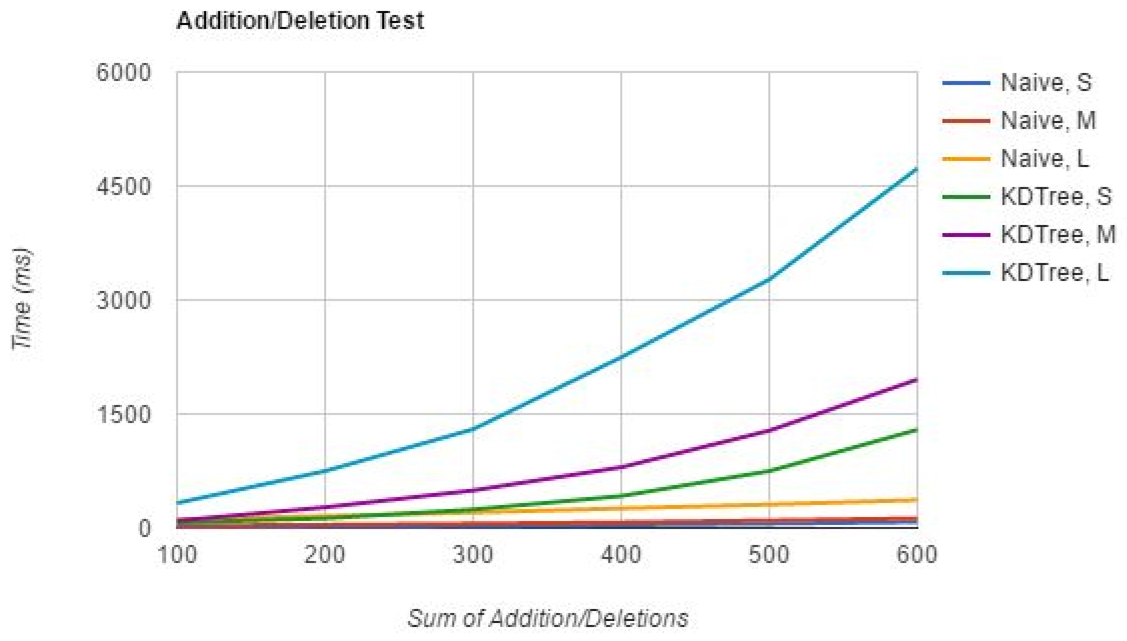
Scenario 1 shows that for a constantly changing dataset with large results, you should be using a Naive approach if using our program.
An example of a program that searches for a large k value of results is a search engine, for example Google or Yahoo. They perform a search and return the k nearest values. Another example is a mapping program like a location finder (kind of like this assignment), where the program has a large dataset, and returns more than 1 value for a search. If our implementation didn't need to return results in the correct order to pass the Part A test then it is clear that a KDTree approach would be the faster option.

Scenario 2: Addition and Deletion of Points

The tests for this scenario were generated by concatenating previous tests. For example. Test 1 has x operations in it. Test 2 has Test 1's x operations, plus its own y operations. Test 3 has both Test 1 and Test 2's operations. This is because the program does not save its changed dataset after runtime, and we wanted to make sure the random factor was not carried through later tests. Each additional test adds 50 additions, 50 deletions, and a random search of the "RESTAURANT" category with a k value of 10.

Sum of Additions/Deletions	Time (ms)					
	Naive, S	Naive, M	Naive, L	KD-Tree, S	KD-Tree, M	KD-Tree, L
100	12	30	112	71	96	328
200	17	42	157	127	272	749
300	25	57	204	243	494	1299
400	39	75	258	420	799	2246
500	55	100	309	749	1281	3270
600	79	127	368	1293	1954	4731

Note: S, M, and L mean small dataset (1000), medium dataset (2500) and large dataset (7500) respectively. Each test conducted 10 times to produce an average. Each search has a K-Value of 10.



The results of this test show that KD-Tree is always slower for this scenario. This is likely due to building and rebuilding of the BST Tree. Both adding a point and deleting a point from the balanced KD-Tree have complexities of O(log n), and in each test 50 of each addition and deletion were called.
The basic operations for Naive are: quicksort the list of points from closest to furthest, then return the first k values of that list. It only quicksorts the list upon search, which drastically limits how long each deletion takes, adding to the unsorted list is simply just appending the new point to the end.
The KD-Tree implementation doesn't have to rebuild for adding a point as just the closest leaf is found and the node is added given the node doesn't already exist.
Despite the KD-Tree implementation having to rebuild the branch around any deletion every time - this should have an average case of O(nlogn) however performance would depend on the location of the deleted node. In the current implementation of the KD-Tree the replacing point needs to be searched for in order to delete it again. This creates a bottleneck in node deletion as far more searches are made than required if the replacing node is stored for deletion. By fixing this, the KD-Tree implementation would potentially run faster than the naive implementation.

The results of this scenario indicate that for a program that requires constant updating of a dataset or database would be better with a naive approach if using our implementation.

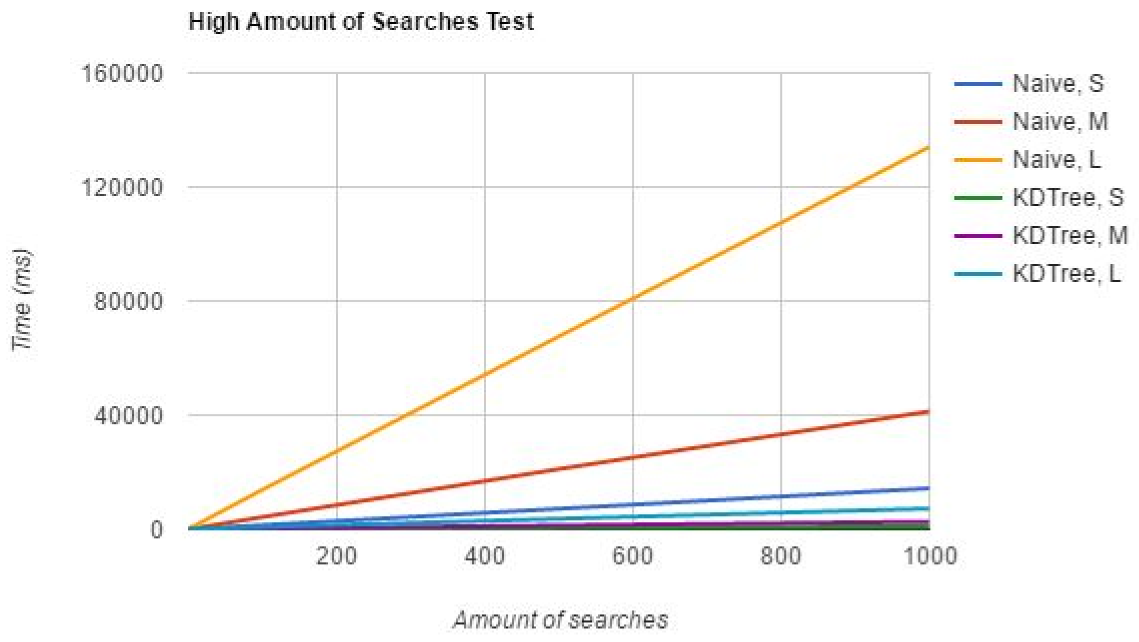
Scenario 3: High Amount of K-Value 1 Searches

After completing the suggested scenarios 1 and 2 we were wondering why you would bother creating a KD-Tree when at high volumes of data Naive seems to be faster. It appears from the results in scenario 2 that Naive searches are always faster if you have a constantly updating dataset. However, in scenario 1 we observed that at low K-Values KD-Tree seems to be a lot faster, so we hypothesised that KD-Tree is a lot faster at doing a high amount of searches where you are only getting 1 nearest neighbour - thus giving us one final scenario to test.

We conducted 4 different tests, with 1 search, 100 searches, 500 searches, and 1000 searches respectively. Each search had a k-value of 1. Each test was conducted 10 times per dataset, and the results were averaged.

	Time (ms)					
Sum of Searches	Naive, S	Naive, M	Naive, L	KD-Tree, S	KD-Tree, M	KD-Tree, L
1	34	59	204	21	19	46
100	1477	4293	13860	122	293	776
500	7205	21104	67527	617	1397	3710
1000	14364	41318	134040	1111	2709	7263

Note: S, M, and L mean small dataset (1000), medium dataset (2500) and large dataset (7500) respectively. Each test conducted 10 times to produce an average. Each search has a K-Value of 1.



The results obtained confirm this hypothesis. When the dataset is not being modified and searches only need to return a single value, the KD-Tree implementation has an average complexity of O(log n) due to problem reduction. However, the Naive implementation has an average complexity of O(n log n). KD-Tree is already over 10 times faster than Naive when the sum of searches is above 100. If you are performing multiple searches that each only require one result, at high speed, KD-Trees will get you those results faster since they're always faster than Naive, according to our data.

Submission notes:
All testing files used are located in the testing folder in the root of the submission.