Algorithms and Data Structures

Nearest Neighbor Search and Radius Search using a K-Dimensional Tree

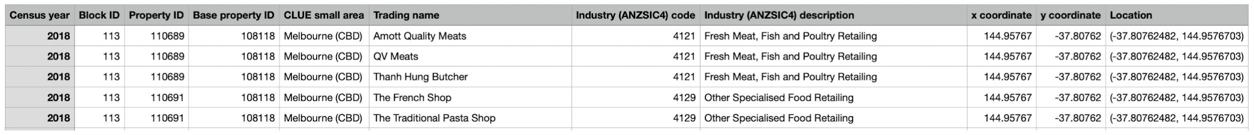
Eisaku Daniel Tanaka

Introduction

Searching through data by location or a range of locations is perhaps one of the most common ways a searching algorithm is used. A tree data structure is a recursive collection of nodes, where each node has references to two children. The children are placed by comparing its key to each node's key. When the key is less than the current node, the child will be inserted to the left reference, and will be inserted to the right reference if it is greater than or equal to the current node's key. Searching for locations can require a few dimensions, for example x-y coordinates if viewing a map from above. K-Dimensional (K-D) trees construct a tree by multi-dimensional keys (e.g. (2,5)), and are able to spatially and hierarchically organize each point it is passed, and such data structure is useful in searching an object by position. They are constructed by alternating the comparing key during insert (e.g. x-coordinate, y-coordinate, x, y, …). A K-D tree is useful because it automatically "decompose(s) space into a small number of cells" (Skiena, 2012), as traversing down the tree will get closer to the searching point. Each node geometrically cuts the plane by its corresponding dimension and would in the ideal case, equally partition the subset of points to two sides of the map, and their children will do so within their partitioned region, and so on.

The K-D tree was applied for an interactive map functionality for the City of Melbourne Census of Land Use and Employment (CLUE) dataset. The user is therefore able to query locations to find nearby businesses or businesses within a specified region. In order to test the usage of K-D trees and its response to differently sorted data, algorithms were implemented for two stages: Nearest neighbor search and radius search. Nearest neighbor search allows for finding the data that contains the coordinates closest to the query point, and radius search makes possible searching for all points contained within a specified radius from the query point.

The CLUE dataset was a CSV file containing rows as below, and the x-y coordinates towards the end were used for the construction of the K-D tree.

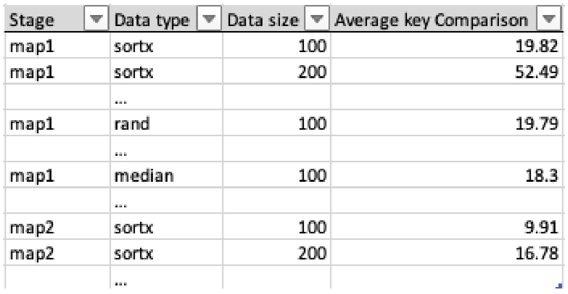


Census year	Block ID	Property ID	Base property ID	CLUE small area	Trading name	Industry (ANZSIC4) code	Industry (ANZSIC4) description	x coordinate	y coordinate	Location
2018	113	110689	108118	Melbourne (CBD)	Amott Quality Meats	4121	Fresh Meat, Fish and Poultry Retailing	144.95767	-37.80762	(-37.80762482, 144.9576703)
2018	113	110689	108118	Melbourne (CBD)	QV Meats	4121	Fresh Meat, Fish and Poultry Retailing	144.95767	-37.80762	(-37.80762482, 144.9576703)
2018	113	110689	108118	Melbourne (CBD)	Thanh Hung Butcher	4121	Fresh Meat, Fish and Poultry Retailing	144.95767	-37.80762	(-37.80762482, 144.9576703)
2018	113	110691	108118	Melbourne (CBD)	The French Shop	4129	Other Specialised Food Retailing	144.95767	-37.80762	(-37.80762482, 144.9576703)
2018	113	110691	108118	Melbourne (CBD)	The Traditional Pasta Shop	4129	Other Specialised Food Retailing	144.95767	-37.80762	(-37.80762482, 144.9576703)

It is important to note that the big O notation will be used to describe the run time of each search. It describes the behavior of algorithms as the number of data (n) scales to infinity. It is calculated by taking the limit of the program approaching n to infinity, and is described as O(n^2) for an algorithm that scales exponentially as the number of data grows. The big O notation takes the dominant order and ignores coefficients. It is also important to note that for a tree data structure the most run-time efficient construction of the tree is if the tree is "balanced". Balanced trees are able to have a logarithmic height and therefore guarantee an order of O(log n) for searching through it. A tree is considered balanced if recursively speaking, the difference between the heights of the left and right subtree is no more than 1.

The construction of the K-D tree can heavily impact the complexity of its searching algorithm, as with any tree, a tree that only inserts to one side of the tree can end up with a stick causing O(n) running time, destroying its advantage of possibly achieving O(log n) running time with a balanced tree. Therefore, in order to test the complexity of the two-stage algorithms based on how input data is sorted during construction, the K-D tree has been constructed exactly in the order of the passed in dataset, and the search algorithms were tested on three differently sorted datafiles: one that is sorted in ascending order by x-coordinates, another that is sorted randomly, and lastly a dataset that has been sorted by the median of its inserting axis. Examining the growth of running time has been done by taking the average key comparisons made out of 100 different search keys on increasing subsets of each of the data types. The number of data in each tested subset were 100, 200, 500, 800, 1000, 2000, 5000, 10000, 12000, 15000, 18000 in order to retrieve enough data to graph the runtime growth trend for each data type. The creation of the subsets, query files, experiment outputs for each subset, and the average key comparisons for each subset data has been recorded through the use of a range of UNIX commands, and will be specified later on.

The results of the experiment were collected in a file called expts.txt and recorded the average key comparisons for each data type to each subset and did this for both stages. The results were then organized and curve fitted using excel. The result expts.txt was in the following format:



Stage	Data type	Data size	Average key Comparison
map1	sortx	100	19.82
map1	sortx	200	52.49
	...		
map1	rand	100	19.79
	...		
map1	median	100	18.3
	...		
map2	sortx	100	9.91
map2	sortx	200	16.78
	...		

“Map1” refers to the Stage 1 experiment with the nearest neighbor search and “map2” is the Stage 2 experimentation with radius search. The stage 1 experimentation of the three data types were done before proceeding to stage 2 for visualization purposes of comparing data types in each algorithm. The data type is specified in the second column (sortx is sorted data), the data size on the third, and the average key comparisons made during the search is on the data size on the fourth column. With these results, this report aims to answer the research question of: **To what extent is the run time of a nearest neighbor search and radius search subject to the sort of the data passed in?**

Stage 1 and Stage 2 Construction and Hypothesis

Nearest neighbor search has been implemented during Stage 1. When doing a nearest neighbor search, the K-D tree must be traversed by keeping track of the node that holds the closest distance. However, traversing a K-D tree by just by determining which side of the partitioned axis the searching point *q* may lie in, is not enough to find the nearest neighbor. One must traverse all of the cells that are contained within the closest distance because even though point *q* may lie on the right side of a certain node, there may be a point on the left side that is closer. This happens if the recorded best distance from *q* contains the left side as well. Keeping the above points in mind, the algorithm was carefully implemented for this experiment. Therefore at each node, it compares the so far nearest neighbor of its subtree, and works to check the other child if the current nearest neighbor’s distance is greater than the difference of the splitting axis of the current node and the query point *q*.

During radius search in Stage 2, the program checks which branch to proceed to, and checks if the distance to the query point is greater than the currently splitting axis’ distance. If not, it will proceed down both branches.

For a standard K-D tree, the average case run time for a balanced tree doing nearest neighbor search is O(log n), and the worst case is O(n) for a balanced tree as well. This worst case can occur if the query point is far away from the entire point set, as the distance is likely to be greater than the difference in the splitting axis. On the other hand for radius search, the best case is O(log n) for a small radius because it would be quite similar to nearest neighbor search, but O(n) for a worst case because if the radius is large enough to cover the entire dataset, the program would have to parse through the entire data. The average case for radius search is an open problem because it is merely dependent on the size of the specified radius.

It can be expected that the general trend of running time (key comparisons) decreases in the order of x-sorted input data, random data, then median data. X-sorted data for both experiments can be expected to grow at a rapid rate because the constructed tree will be right heavy. Each x-axis splitting node will put it's child on the right side. For stage 1 with the x-sorted data, the key comparisons can however be expected to converge to O(log n) because each y-axis splitting point is able to split data in two different references thereby allowing for more efficient search, and the average complexity for nearest neighbor search is O(log n). Stage 2 with x-sorted data, can be expected to be similar but more faster compared to stage 1. This is because the radius is set very small, and therefore it will become easier to prune the unneeded branches while stage 1 will be taking time to "search both sides of the tree".

Random data for stage 1 can be expected to achieve O(log n) because that is the average case, and unlike x-sorted input the tree would not be right heavy. Random data for stage 2 can be expected to be similar, except faster than stage 1 because of the small radius.

Finally median data input can guarantee O(log n) run time because each node placed will be the median of its children. Such balance is made possible because for each node, by taking exactly the median of the data by its inserting axis, and recursively performing this ends with exactly (n-1)/2 elements inserted into each node's children, creating a balanced tree. This can guarantee O(log n) run time.

Method (Both Stage 1 and Stage 2)

1. Prepare the three different types of datasets.
    a. CLUEdata2018_sortx.csv
    b. CLUEdata2018_random.csv
    c. CLUEdata2018_median.csv
2. Create the datafiles with no header line (random was made by shuffling sortx in step 3).
    a. Files are in the format of <data type>_all.csv

```
Tail -n +2 CLUEdata2018_sortx.csv > sortx_all.csv
Tail -n +2 CLUEdata2018_median.csv > median_all.csv
```

3. Create the subset datafiles, in the format of <data type>_<data size>.csv.

```
for size in 100 200 500 800 1000 2000 5000 8000 10000 12000 15000 18000; do head
-n $size sortx_all.csv > sortx_$size.csv; done
```

4. Create the query files
    a. Query files are in the format "q1.txt" and "q2.txt"

Run following commands:

```
cat sortx_all.csv| awk -F ',' '{if(($9~/^[0-9]+/) && ($10~/[0-9,.]+/) ) {print
$9,$10;}}’ | shuf > ./x

head -n 100 ./x | awk '{printf("%lf %lf\n", $1, $2)}' > q1.txt

head -n 100 ./x | awk '{printf("%lf %lf 0.0005\n", $1, $2)}' > q2.txt
```

5. Run each program for all sizes and each data types and output result.
    a. Output made to file in format of “out_<stage (map1 or map2)>_<data type>_<data size>.txt”
    b. Key comparisons for each query should be sent to distinct files in the format of: “<stage (map1 or map2)>_<data type>_<data size>.txt”

```
for map in 1 2; do for size in 100 200 500 800 1000 2000 5000 8000 10000 12000
15000 18000; do for type in sortx rand median; do ./map$map $type_$size.csv
out_$map_$type_$size.txt < q$map.txt > map$map_$type_$size.txt; done ; done; done
```

6. Take average of results in a single file
    a. Output all averages of all experiments to “expts.txt”

```
for map in 1 2; do for type in sortx rand median; do for size in 100 200 500 800
1000 2000 5000 8000 10000 12000 15000 18000; do cat map$map-$type-$size.txt | awk
-v size=$size - v map=$map -v type=$type 'BEGIN{sum=0; n=0}{sum = sum+$4;
n++}END{print ”map”map, type, size, sum/n}' >> expts.txt ; done ; done; done
```

7. When fitting the results to a curve, send results to excel and fit on a scatter plot.

Results and Comparison with Theory

This section will first go through results from Stage 1 and compare the various inputs. Stage 2 will not only compare its various inputs but also compare its results in relation to the results gained in Stage 1.

Stage 1

Sorted Input Search Stage 1

Firstly, the results for the sorted input were as the following:

map1	sortx	100	19.82
map1	sortx	200	52.49
map1	sortx	500	142.55
map1	sortx	800	230.13
map1	sortx	1000	289.89
map1	sortx	2000	463.66
map1	sortx	5000	712.12
map1	sortx	8000	580.21
map1	sortx	10000	537.76
map1	sortx	12000	473.16
map1	sortx	15000	253.79
map1	sortx	18000	89.73

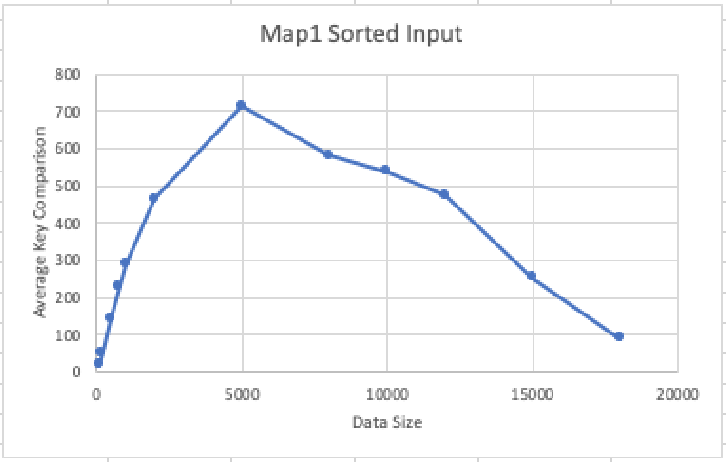


Figure 1

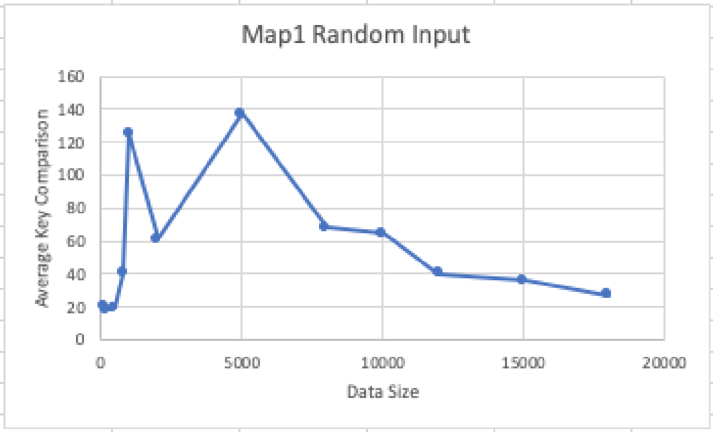
It is evident that until a dataset of 5000 rows, the number of key comparisons continue to rapidly grow. However, from then on, it gradually descends to almost identical to a very low amount of data. This is expected in theory because the data is sorted only by the x-coordinate. Though this is unlikely for a large city dataset like CLUE, if the data were sorted in ascending order for both x and y coordinates, the constructed tree would be a stick, because it would continue to insert each node to the leaf's right reference. Therefore, this essentially creates a one dimensional linked list causing an O(n) run time. However with this sortx dataset, each node that splits by the x-axis would always insert the next node to its right reference, but inserting from a y-axis splitting node, the inserting node can either go to the left or the right reference. Therefore, although this tree would be right heavy causing an initial increase in key comparisons and runtime, as the number of data increases, the splitting point from the y-axis increases and therefore slightly mitigates the effects of the x coordinates being in ascending order. In terms of a map and splitting points, as the number of data increases, the program would have more y-axis splitting points. This makes the search easier because on average it would become easier to "throw away" one side of an entire y-axis partitioned side, and therefore, the x coordinates that are associated to the unneeded side can be thrown away easily, making traversing down to the correct cell easier.

It is also noteworthy that each key comparison is closer to O(log n) because the query points were chosen from the given dataset. The worst case only occurs when the query point is far outside of the passed in dataset.

Random Input Search Stage 1

The results for a random input nearest neighbor search were the following:

map1	rand	100	19.79
map1	rand	200	18.51
map1	rand	500	19.27
map1	rand	800	40.08
map1	rand	1000	125.5
map1	rand	2000	61.33
map1	rand	5000	137.19
map1	rand	8000	68.09
map1	rand	10000	64.59
map1	rand	12000	39.98
map1	rand	15000	36.14
map1	rand	18000	26.9



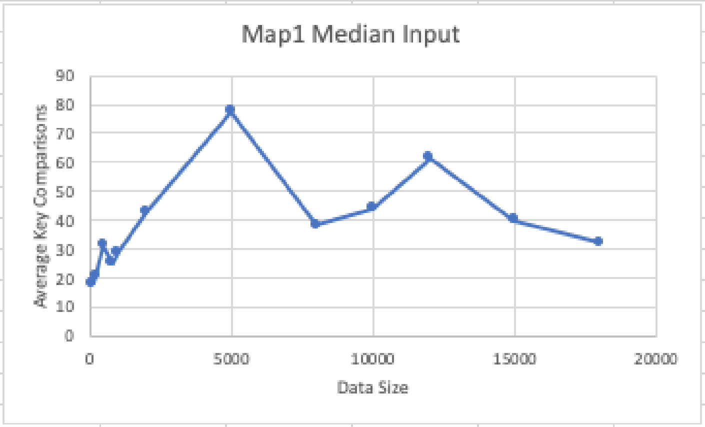
The results with x-sorted input for nearest neighbor search showed a maximum key comparison of 712.12 key comparisons whereas the dataset here has 137.19 as the maximum. This is almost 80% smaller, and therefore more efficient. Therefore, a sharp decrease in the average number of key comparisons can be observed. This can be attributed to the fact that with random input, the data would be more scattered out to the left and right of the tree, getting closer to an O(log n) run time, as opposed to the tree that was right-heavy with the x-sorted input. Therefore, this agrees with the theory because this allows for the tree to be more close to a balanced tree, and therefore be closer to an O(log n) running time.

One peculiar thing is that the average key comparisons made for a data size of 100 is similar to what came out with the sorted input. This could perhaps be because the random inputs coincidentally were initially not scattered and caused a tree structure similar to that with the x-sorted input. However, the average growth with the rest of the data is far less than the x-sorted data, and for 18000 datas, 26.9 key comparisons were made, supporting the argument of an average O(log n) run time.

Median Input Search Stage 1

The results for a median input nearest neighbor search were the following:

map1	median	100	18.3
map1	median	200	20.6
map1	median	500	31.27
map1	median	800	25.16
map1	median	1000	29.09
map1	median	2000	42.87
map1	median	5000	77.45
map1	median	8000	38.51
map1	median	10000	43.96
map1	median	12000	61.59
map1	median	15000	40.19
map1	median	18000	32.19

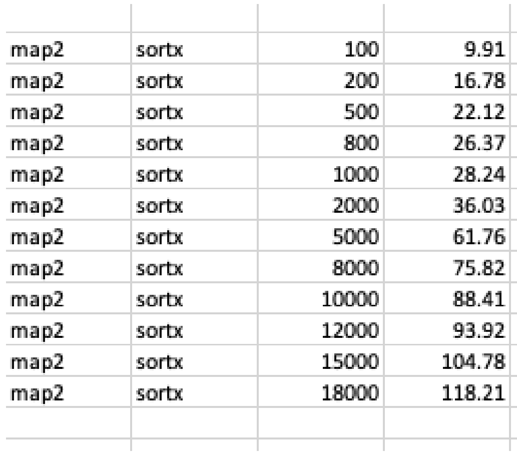


Out of the results from Stage 1, the median input proved to have the smallest average number of average key comparisons with 38.43 comparisons, where the random input had 54.78 average comparisons, and x-sorted input had 320.44 average inputs. The average key comparisons for all of the data sizes came out to be under 100 comparisons. This supports the theory because with each row being the median of the corresponding input axis and data, the tree will be balanced yielding an O(log n) run time for the search.

Stage 2 with comparison to Stage 1

Sorted Input Search Stage 2

Results for x-sorted input radius search (radius = 0.0005):



map2	sortx	100	9.91
map2	sortx	200	16.78
map2	sortx	500	22.12
map2	sortx	800	26.37
map2	sortx	1000	28.24
map2	sortx	2000	36.03
map2	sortx	5000	61.76
map2	sortx	8000	75.82
map2	sortx	10000	88.41
map2	sortx	12000	93.92
map2	sortx	15000	104.78
map2	sortx	18000	118.21

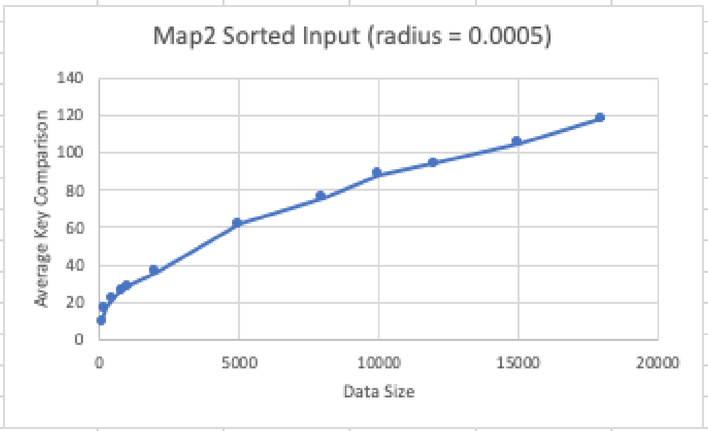
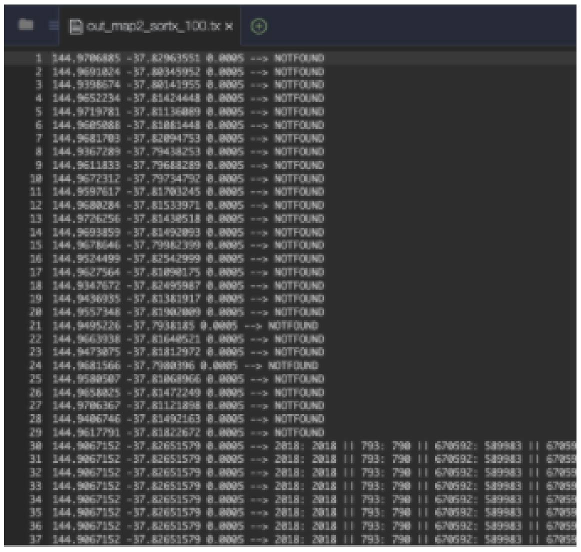


Figure 2

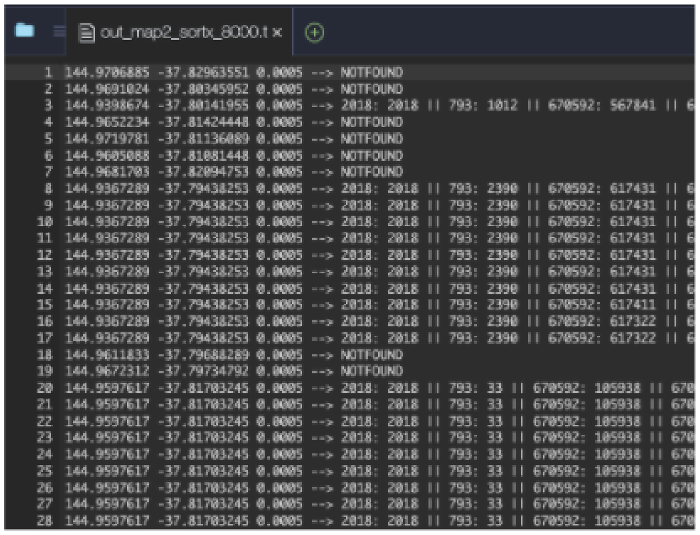
First of all, it is important to note that the overall number of key comparisons heavily diminishes compared to x-sorted input nearest neighbor search (stage 1). The average key comparisons were 56.86 comparisons, which is almost 6 times smaller than Stage 1 sorted input. This can be thought as due to the fact that the radius is set very small at 0.0005. Unlike the nearest neighbor search that has to compare the best results to the difference in splitting axis to the query point, radius search can easily prune the tree with a smaller radius because it would only "check both branches" if the difference in axis is less than the radius, and this is unlikely for a small radius. Therefore, although the program may incur some effects from a right-heavy tree, given the advantage of the data sorted only by the x-axis and the radius being small, it can search through the tree more easily than the nearest neighbor search.

This is supported by the fact that most of the resulting output for map2 sorted input is marked as not found for a small dataset but increases for a large dataset as seen on figure 3 (datasize = 100) and 4 (datasize = 8000):

Figure 3 and 4



```
144.9706885 -37.82963551 0.0005 --> NOTFOUND
144.9691024 -37.80345952 0.0005 --> NOTFOUND
144.9398674 -37.80141955 0.0005 --> NOTFOUND
144.9652234 -37.81424448 0.0005 --> NOTFOUND
144.9719781 -37.81136089 0.0005 --> NOTFOUND
144.9605088 -37.81081448 0.0005 --> NOTFOUND
144.9681703 -37.82094753 0.0005 --> NOTFOUND
144.9367289 -37.79438253 0.0005 --> NOTFOUND
144.9611833 -37.79688289 0.0005 --> NOTFOUND
144.9672312 -37.79734792 0.0005 --> NOTFOUND
144.9597617 -37.81703245 0.0005 --> NOTFOUND
144.9680284 -37.81533971 0.0005 --> NOTFOUND
144.9726256 -37.81430518 0.0005 --> NOTFOUND
144.9693859 -37.81492093 0.0005 --> NOTFOUND
144.9678646 -37.79982399 0.0005 --> NOTFOUND
144.9524499 -37.82542999 0.0005 --> NOTFOUND
144.9627564 -37.81090175 0.0005 --> NOTFOUND
144.9347672 -37.82495987 0.0005 --> NOTFOUND
144.9436935 -37.81381917 0.0005 --> NOTFOUND
144.9557348 -37.81902009 0.0005 --> NOTFOUND
144.9495226 -37.7938185 0.0005 --> NOTFOUND
144.9663938 -37.81640521 0.0005 --> NOTFOUND
144.9473075 -37.81812972 0.0005 --> NOTFOUND
144.9681566 -37.7980396 0.0005 --> NOTFOUND
144.9580507 -37.81068966 0.0005 --> NOTFOUND
144.9658025 -37.81472249 0.0005 --> NOTFOUND
144.9706367 -37.81121898 0.0005 --> NOTFOUND
144.9406746 -37.81492163 0.0005 --> NOTFOUND
144.9617791 -37.81822672 0.0005 --> NOTFOUND
144.9067152 -37.82651579 0.0005 --> 2018: 2018 || 793: 790 || 670592: 589983 || 67059
144.9067152 -37.82651579 0.0005 --> 2018: 2018 || 793: 790 || 670592: 589983 || 67059
144.9067152 -37.82651579 0.0005 --> 2018: 2018 || 793: 790 || 670592: 589983 || 67059
144.9067152 -37.82651579 0.0005 --> 2018: 2018 || 793: 790 || 670592: 589983 || 67059
144.9067152 -37.82651579 0.0005 --> 2018: 2018 || 793: 790 || 670592: 589983 || 67059
144.9067152 -37.82651579 0.0005 --> 2018: 2018 || 793: 790 || 670592: 589983 || 67059
144.9067152 -37.82651579 0.0005 --> 2018: 2018 || 793: 790 || 670592: 589983 || 67059
144.9067152 -37.82651579 0.0005 --> 2018: 2018 || 793: 790 || 670592: 589983 || 67059
```

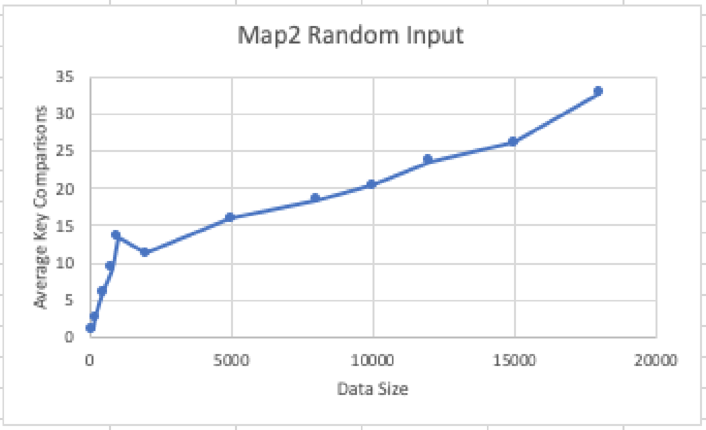


```
144.9706885 -37.82963551 0.0005 --> NOTFOUND
144.9691024 -37.80345952 0.0005 --> NOTFOUND
144.9398674 -37.80141955 0.0005 --> 2018: 2018 || 793: 1012 || 670592: 567841 || 6
144.9652234 -37.81424448 0.0005 --> NOTFOUND
144.9719781 -37.81136089 0.0005 --> NOTFOUND
144.9605088 -37.81081448 0.0005 --> NOTFOUND
144.9681703 -37.82094753 0.0005 --> NOTFOUND
144.9367289 -37.79438253 0.0005 --> 2018: 2018 || 793: 2390 || 670592: 617431 || 6
144.9367289 -37.79438253 0.0005 --> 2018: 2018 || 793: 2390 || 670592: 617431 || 6
144.9367289 -37.79438253 0.0005 --> 2018: 2018 || 793: 2390 || 670592: 617431 || 6
144.9367289 -37.79438253 0.0005 --> 2018: 2018 || 793: 2390 || 670592: 617431 || 6
144.9367289 -37.79438253 0.0005 --> 2018: 2018 || 793: 2390 || 670592: 617431 || 6
144.9367289 -37.79438253 0.0005 --> 2018: 2018 || 793: 2390 || 670592: 617431 || 6
144.9367289 -37.79438253 0.0005 --> 2018: 2018 || 793: 2390 || 670592: 617431 || 6
144.9367289 -37.79438253 0.0005 --> 2018: 2018 || 793: 2390 || 670592: 617411 || 6
144.9367289 -37.79438253 0.0005 --> 2018: 2018 || 793: 2390 || 670592: 617322 || 6
144.9367289 -37.79438253 0.0005 --> 2018: 2018 || 793: 2390 || 670592: 617322 || 6
144.9611833 -37.79688289 0.0005 --> NOTFOUND
144.9672312 -37.79734792 0.0005 --> NOTFOUND
144.9597617 -37.81703245 0.0005 --> 2018: 2018 || 793: 33 || 670592: 105938 || 670
144.9597617 -37.81703245 0.0005 --> 2018: 2018 || 793: 33 || 670592: 105938 || 670
144.9597617 -37.81703245 0.0005 --> 2018: 2018 || 793: 33 || 670592: 105938 || 670
144.9597617 -37.81703245 0.0005 --> 2018: 2018 || 793: 33 || 670592: 105938 || 670
144.9597617 -37.81703245 0.0005 --> 2018: 2018 || 793: 33 || 670592: 105938 || 670
144.9597617 -37.81703245 0.0005 --> 2018: 2018 || 793: 33 || 670592: 105938 || 670
144.9597617 -37.81703245 0.0005 --> 2018: 2018 || 793: 33 || 670592: 105938 || 670
144.9597617 -37.81703245 0.0005 --> 2018: 2018 || 793: 33 || 670592: 105938 || 670
144.9597617 -37.81703245 0.0005 --> 2018: 2018 || 793: 33 || 670592: 105938 || 670
```

Random Input Search Stage 2

The results for a random input radius search:

map2	rand	100	1.09
map2	rand	200	2.71
map2	rand	500	6.08
map2	rand	800	9.32
map2	rand	1000	13.49
map2	rand	2000	11.28
map2	rand	5000	15.95
map2	rand	8000	18.43
map2	rand	10000	20.35
map2	rand	12000	23.6
map2	rand	15000	26.16
map2	rand	18000	32.69

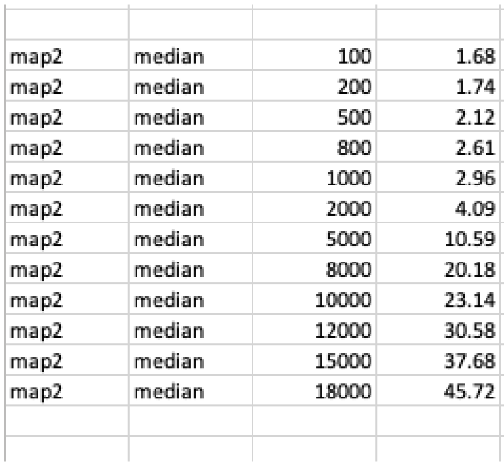


The random input radius search appeared to be much faster than the sorted input, which follows the expected theory. As discussed before, the random input is less likely to cause a right-heavy tree, and therefore will allow for efficient algorithms. The average number across the datasets were 15.09 comparisons.

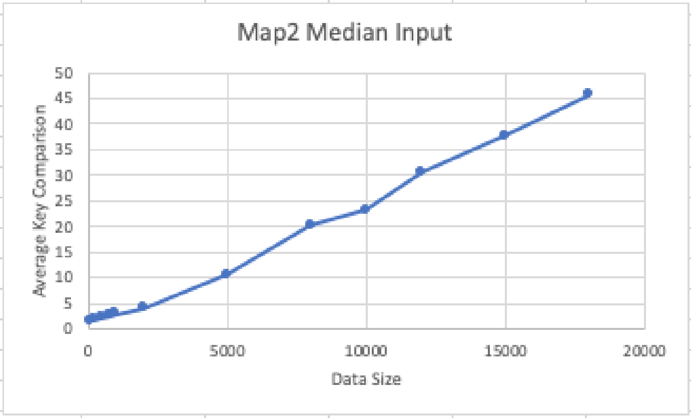
As the data size scales up, the number of key comparisons get closer to $\log_2$ of the datasize. Therefore, this can be confidently said that this program is being able to achieve an O(log n) run time due to the random inputs and the small radius, and therefore supports the theory and expectation.

Median Input Search Stage 2

The results for median input radius search:



map2	median	100	1.68
map2	median	200	1.74
map2	median	500	2.12
map2	median	800	2.61
map2	median	1000	2.96
map2	median	2000	4.09
map2	median	5000	10.59
map2	median	8000	20.18
map2	median	10000	23.14
map2	median	12000	30.58
map2	median	15000	37.68
map2	median	18000	45.72



The average key comparisons made here were 15.25, and once again supports the efficiency of median inputs creating a balanced tree. This therefore supports the best case run time of O(log n). It is much faster than the stage 1 instance as well because of the small radius.

Discussion and Conclusion

In light of the detailed experimentation and analysis above, the research question of: **To what extent is the run time of a nearest neighbor search and radius search subject to the sort of the data passed in?** Has been understood and answered. It was found that the hypotheses at the beginning of the report were accurate, as the longest running time was recorded for x-sorted input data, and scaled down with the random, then the median data input. Another component that was expected was that the nearest neighbor had a greater number of key comparisons compared to radius search. Here is a summary of the average key comparisons for each data type:

	Stage 1 Nearest Neighbor Search	Stage 2 Radius Search
Sortx Data Average Key Comparisons (X-sorted)	320.4425	56.8625
Rand Data Average Key Comparisons (Random)	52.7808333	15.0958333
Median Data Average Key Comparisons	38.4316667	15.2575

All of the results had an O(log n) runtime. For stage 1 this was because with x-sorted data input 1 scaled up rapidly and converged towards O(log n) with an increase in splitting points. Random data input and median data input showed O(log n) as well because of the average case complexity of nearest neighbor search. Stage 2 radius search also ended up having an O(log n) run time with much smaller key comparisons compared stage 1 because of the very small radius. This allowed for easiering searching through the tree.

However, what was interesting with the results was that on average, the median data input showed to have the smallest number of key comparisons. This makes sense because each node would have an equal number of children, creating a balanced tree and allowing for the best case O(log n) search. Although the x-sorted input results and the median results were analyzed to have O(log n) by the end of their experimentation, each number of key comparisons were either unstable or slower than the median data input. The random data input for stage 2 had an average number of key comparisons slightly smaller than the median data, but this was caused by the small radius. With a larger radius, the results should show that the median data input is the most efficient for both stages.

The limitations upon these experimentations should be acknowledged. One flaw upon the experimentation was that only one radius size was experimented. This should have been done with a larger radius to show how the average key comparisons diminish with the passed in data type. A true result of the random data input could have been achieved by shuffling and experimenting the data multiple times. Secondly, the number of queries passed in may have been too small. In order to show a better fit of the graph, the number of queries should have been made much larger to make the average key comparisons more accurate.

Though K-D tree nearest neighbor search algorithms can have cases where the search is O(n), this occurs when the query is far away from the data set and makes each data "check both sides". In order to see how the run time scales with how the data is sorted when constructing a K-D tree without manually balancing it, it is wise to not include this.

An important learning made from this experimentation was that in order to construct a tree that allows for the most efficient operations (search, insert, or delete), is to construct the tree by taking the median of the inserting data set or subset by its splitting axis. This could allow for O(log n) operations.

Resources

Skiena, S. S. (2012). *The algorithm design manual*. London: Springer.