Complexity of K-D Trees

Mahee Hossain 1080102

Introduction

A K-D tree (type of binary tree) is a data structure which is used for organising data through the comparison of points in K dimensions. In this simple K-D Tree, points of a map are taken with latitude and longitude used as the x and y values, with data attached to the points. When inserting a new point, the K-D tree will compare the new point with the point at the root, and send it left or right depending on its comparison to the x or y value. The point keeps moving down the table recursively, until it finds a spot on the tree, or is added to an existing spot inside a linked list (if the point is equal).

The inputs used to test the K-D tree are datasets with information about Businesses located in Melbourne. As the K-D tree unfortunately never managed to run (due to an error in the logic of the tree), it was not possible to experiment using the datasets, so we cannot tell how the K-D tree would have run in the two stages. However, we can discuss what theoretically should happen.

To test the complexity of the K-D Tree, various datasets and key files had to be created. Three datasets were already given, one in a random order, one in a sorted (by x coordinate) order, and one sorted by median. Other databases were created using UNIX commands, multiple csv files could be created with the line:

```
for size in 100 1000 10000; do head -n $size data_sortx.csv > data_sortx_$size.csv;
shuf data_sortx_$size.csv > data_random_$size.csv; done
```

Similarly, key files could be created with:

```
cat data_sortx.csv | awk -F ',' '{if (($9~/^[0-9]+/) && ($10~/[0-9,.]+/)) {print $9,$10;}}' |
shuf > ./x head -n 100 ./x | awk '{printf("%f %f\n", $1, $2)}' > data_q1.txt
```

As the tree did not work, only 6 new datasets were created (100, 1000 and 10000 entries for both random and sorted), and only 2 key files were made (100 long for both map1 and map2). If the tree were operational and experimentation was possible, there would be more datasets with median being included, and datapoints being repeated, as well as additional randomly made key files, to account for outliers (i.e. all the keys being near the top of the tree.

Stage 1

Method

1. Create multiple smaller datasets from the median, random, and sorted (in both directions) datasets
2. Make some key files to test the datasets, add some of your own (fake) keys in the files
3. Run map1 with the chosen dataset and key file, record the number of comparisons for each key, and average them (note the difference between real and fake keys)

As mentioned in the introduction, due to an error in the logic of the K-D Tree (most likely two errors, one in traversing the tree when creating it, and one with creating the linked lists found in each of the nodes of the trees), the data can't properly be tested. No matter what data is entered into the map1 program, or what key is used, the number of comparisons is returned as 1 (meaning none of the trees had anything to the left or right nodes). This may imply that all the data is mistakenly being stored in the root node. This shouldn't be happening, but if it did then the complexity going through the tree should be O(1), though the complexity of printing out the linked list (if that also worked) would be O(n).

THEORY PART

However, we can look at the theory to see what the time complexity should have been. When the data is random (i.e. not sorted), the tree should be relatively shorter (shorter than a linked list), as when comparing the points of a node being inserted and the node at the root, the node is equally likely to go either left or right (assuming that the data is perfectly random). Each layer of the tree should have roughly twice as many nodes as the last, meaning that finding any node would have a complexity of O(logn). As the tree is not designed to be perfectly balanced however (i.e. not an AVL tree), some branches of the tree would be longer than others, leading to outliers in the data. These should not be significant however, and thus we would the data to roughly follow a log(n) line.

When the data is sorted however, when inserting a new node, it will always go left (if sorted in descending order) or right (sorted in ascending order), for the sorted dimension. Even though the other dimension's comparisons will be random and thus either go left or right, the tree will still only be able to split every second level. Thus, although the tree would not be a proper stick (no branches, basically a linked list), it would be close to one, and thus the complexity would be O(n) (average comparisons should be roughly 3n/4 if nodes are equally likely to go in either direction for the non-sorted dimension).

Stage 2

Method

1. Reuse datasets from stage 1
2. Make key files as in stage 1, but add the radius in as well

3. Run map2 with the chosen dataset and key file, record the number of comparisons for each key, and average them (note the difference between real and fake keys)

As in stage 1, the map2 program did not run properly, so we must go off theory. Unlike stage 1, where the aim was to find a single node which is closest to the key, we must now find all the nodes in the radius of the key. The function created to find all these nodes simply goes through the tree using in order traversal, thus checking every node to see if it fits the radius. The complexity of this operation should be O(n), with the program hitting every single node. There should be a better approach to this, by checking where the circle of the radius is compared to the current node, and only go in the direction(s) that hold the radius. If this approach is utilised, then the complexity should be closer to O(logn), apart from edge cases (such as when the radius encapsulates all the points, every node will have to go both left and right making the complexity O(n).

Conclusion

Due to errors in the implementation of the K-D Tree, neither the stages could be properly tested, as map1 and map2 could not be run on the datasets. Despite this, due to our understanding of the theory of K-D Trees, we can tell that searching through datasets on a K-D tree should be achieved in O(logn) time, apart from sorted datasets, which would take O(n) time.