Εργασία “K-D AND QUAD TREE” σε python στο μάθημα «Πολυδιάστατες δομές δεδομένων»

Βραχνής Παύλος Α.Μ. : 236010

KD_QUAD-Tree

The program executes only one function, the function run(). In **run** the program asks user to give select between the default file whis is a dataset of 10 elements with 2 dimensions and the 2nd option is to choose himself the path of the file which we are going to use. Then it runs the function file_len and file_dim with input filename where filaname is the file the user chose. The first function finds the elements of the file(rows) and the second one finds the dimension of the file(collumns). Then it makes with the function **read_file** the raw data of the file to a string with a 2d structure (data = (x,y), … ,(x,y)].Afterwards it calls the function type_of_tree which asks th user the type of tree he wants to use between K-D and Quad tree. After that a tree object is created and then we call the **create_root** method which executes on the tree object. Next it finds the time of creating the tree and the size of the tree with the method **size**. If the user chose the k-d tree then it also created a visual represantation of the tree. After the **main_menu** function is beeing called.

Create root

K-D) In this method we sort the data we have according to the axis we are at.So for depth =0 (root) we have axis =0 so we sort them according to the first dimension for depth = 1 according to 2nd for depth =

2 according to the fisrt again and so on. After that we find the median and we separate the left children and the right children of the root which is the median(data[median]). Then a Node object is created that has axis, data, left children, right children and depth. So we created the root of our tree we the data that are needed and adter that we call the method **create_tree.**

Quad) In this case the same things happen but a little differrent. Istead of right and left children we have ne, se ,nw, sw and qdata(split node).The qdata is different than 0 only when the node is a split node and only a split node has ne,se,sw,nw. If the node is a leaf it has none of the above but only its data or if there are two nodes, data2 also. Here we do not find the median but we fins the average of the data we are given and that average is gonna be our split node. We find the average with the method **calc_average**. Then we create a QuadNode object which has ne,nw,sw,sw,depth,qdata) After the method create_tree2 is beeing called.

Create_tree

K-D) In this method we have two variables rcheck and lcheck which are "n" by default and if the node has a right or a left child the rcheck="r" lcheck = "l" accordingly. Then method **child** is beeing called

Quad) In this method we have four variables check1, check2, check3, check4 which are "n" by default and if the node has ne, se, nw or sw the checks1 = "ne" and etc. Futhermore the method **child** is beeing called.

Child

K-D) Here we do what we did in creta the root but the node is not the root but ust a simle node. We sort the data due to the current axis and we find the median which is the node data.After that we separate the left and the right child and we create a Node object if the previous node has a left or a right child. All this are beeing done recursively as after the creation of the Node object we call again the method create_tree until there are no more nodes to add.

Quad) We do the same thing but the main difference is that we have no median but instead we have an average like before and no left and right but instead ne, nw, se, sw.

Main_menu

In this function first we check what option the user chose (K-D or Quad) and we show the aprrppriate menu. There are 8 options: Insert, Delete, Point search, Range search, KNN, Visualitation, Go back (to the start), Exit.

If user presses 1 then the user inserts the element he wants to add and we insert the element with x, y dimensions to the tree with the method **insert**. Then we add the element to the visual tree.

If user presses 2 then the user inserts the elements he wants to delete and we delete the element (if it exists) with the method **delete**. Then we delete the element from the visual tree.

If user presses 3 then the user inserts the elements he wants to find and we check if this element exists in the tree with the method point_search and we return the appropriate result.

If user presses 4 then the user inserts the range he wants to find elements and we search what elements exists inside the range with the method range search and we return the results.

If user presses 5 then the user inserts the elements he wants to find KNN and we search for the KNN with method kNN and we return the results.

If user presses 6 then we show the visual tree.

If user presses 7 then we go back to start.

If user presses 8 then we exit the program.

Insert

K-D) In this method in the beginning we start from the root(we have input 0 so we start from the root). We compare the node axis with the element we want to insert axis. If the elements axis its smaller, then we check if the node we are comparing has a left child. If it doesn't the there is no reason to continue so we insert the element to the left of the node. Else we recursively call itself with input the left child and so on. If its bigger then we check if the node has right child. If no then we insert element as right child. Else we recursively call itself with input right child of the node.

Quad) Here we do as the above but we have ne, se, nw, sw and its beeing done accordingly. Also we have a big difference, the fact tha two nodes can be in a single leaf without creating another split node. So we have to check if not only the first node exists but also the second. If it does the we call the calc_average and we create a split node, else it is inserted in the same leaf. If we create a split node we have to create QuadNode object and the we have to check where its node will go (se, sw, ne, nw) recursively calling itself everytime with input the next node.

Delete

K-D) In delete at first we check if the element is in the tree**, 1)** if it is we check if the node has right child. If it has then the node is replace by the right child and we recursively delete the right child(because it is doubled) as we call itself with input the replace node. If it doesn't have a right child and it has a left child then we replace the node with the maximum nodes data at the current axis of the left subtree with the method **maxvalue**. But the replace node is also doubled so we recursively delete it by calling itself with input the replace node. If it doesn't have either left or right child then it's a leaf and we delete it.Because it cant delete itself we have a variable previous node which is the node the leaf is and we also keep track of where it is(left or right of the previous node) and we delete either the left or the right child of the previous node depending at which direction the leaf is. **2)** If its not then we continue to search left or right of the node and we keep doing it recursively.

Quad) In the Quad tree all the data are stored at leaves, so we just delete the leaves. We have as above sw,se,ne,nw and we also have multiple compares to do with the split nodes until we find the leaves. Also we check if the leafs have two nodes.

Point search

In point_search in both trees we do a lot of compares as well and we do it like before recursively so if for example the elements axis data the user wants to find is bigger than the nodes axis data we are comparing at then we first check if it has a right child and if it has there is no reason to keep searching because it cannot exist. On the other hand though if it has a right child then we call itself with input the right node and we now comparing the element with the right node and so on. And that is the logic for the rest of the compares.

Range search

We do the dame thing as we did at the point search but with ranges for both trees.

kNN

In Knn with the use of numpy and the method **dist** we calculate the distance from the input point of the user to every other node in the tree and we keep only the amount of input nodes of the user.

Visual

This method prints a visual represantation of the kd tree with the **module kdtree** and furthemore it calls the method **visual_tree** which it prints all the nodes the tree has, at both k-d and quad tree.

Size

In this method we run the whole tree and we calculate the size of all the nodes to find the final size of the tree with **getsizeof**.

There are also the functions **file_dim, file_len, read_file, type_of_tree, dist, visual_tree, maxvalue** and the method **calc_average** which are already described previously.

Create file.py file

In this file we created our data in the form of "x -y -". The user is asked to give dimensions, elements number and elements ranges.

Experimental Tests

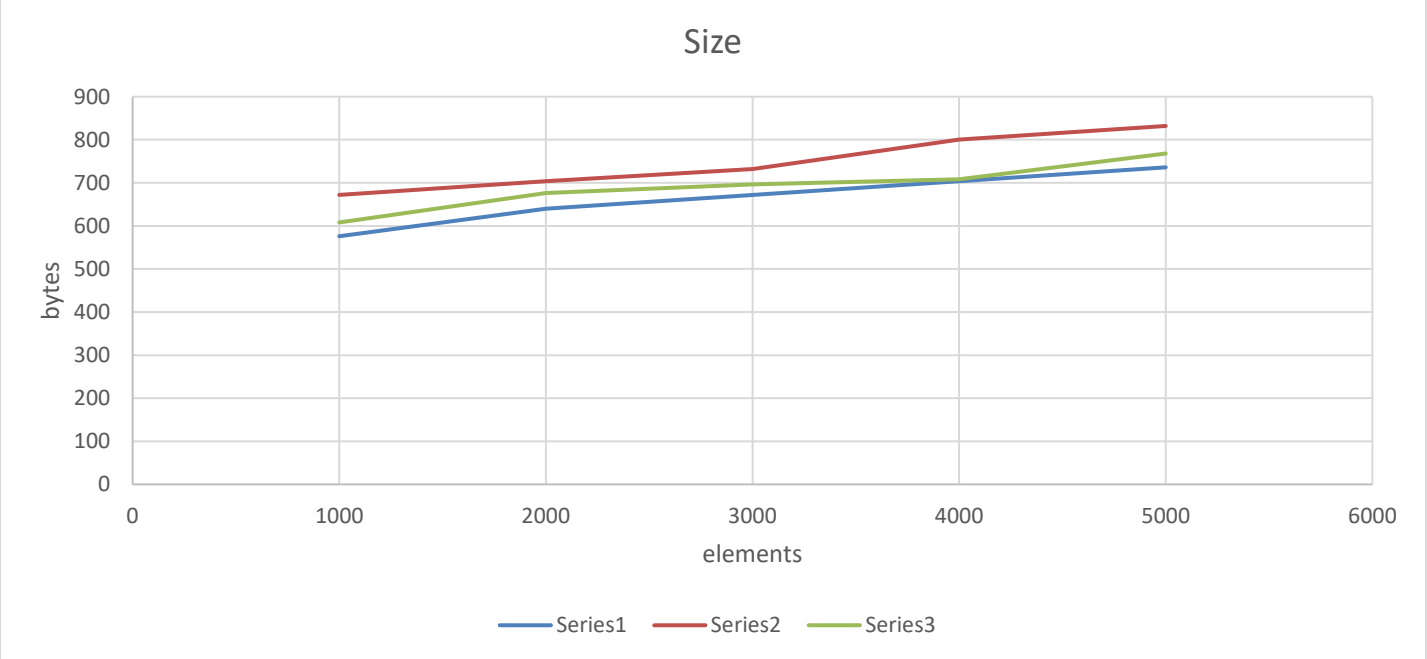
The text files we created have 2 dimensions, with uniform 1 and their range is 1-1000(1.0 -1000.0 -). We used 5 files for testing, from 1000 up to 5000 elements. Point search and Insert was negligent so we didn't included them in the results. The results do not represent exactly the real times of K-D and Quad trees and they are affectted by our implementation. We also did two iimplemantations at the quad tree,

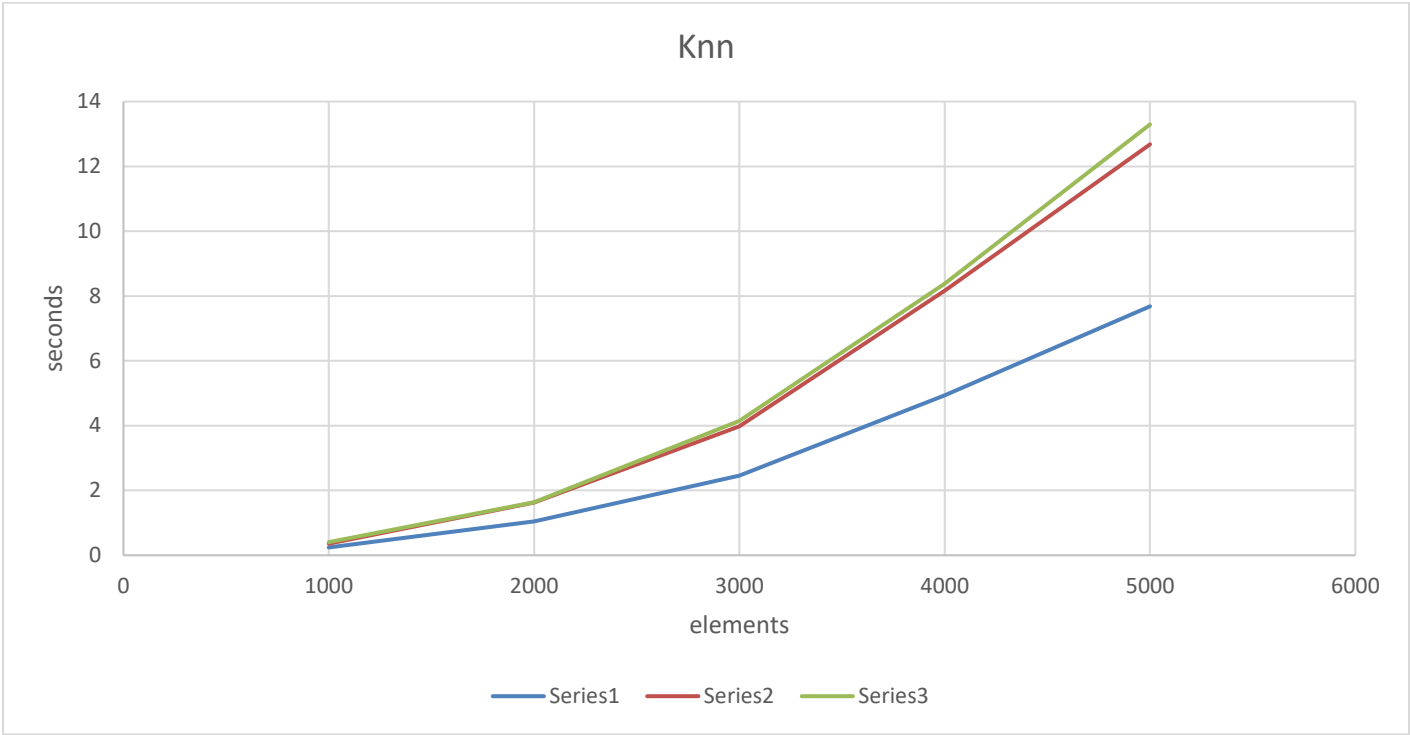
first is where split nodes are the average of all the data for each dimension and second where split nodes are the median of all the data. Below we show our results of the compare between K-D and Quad tree for 2 dimensions :

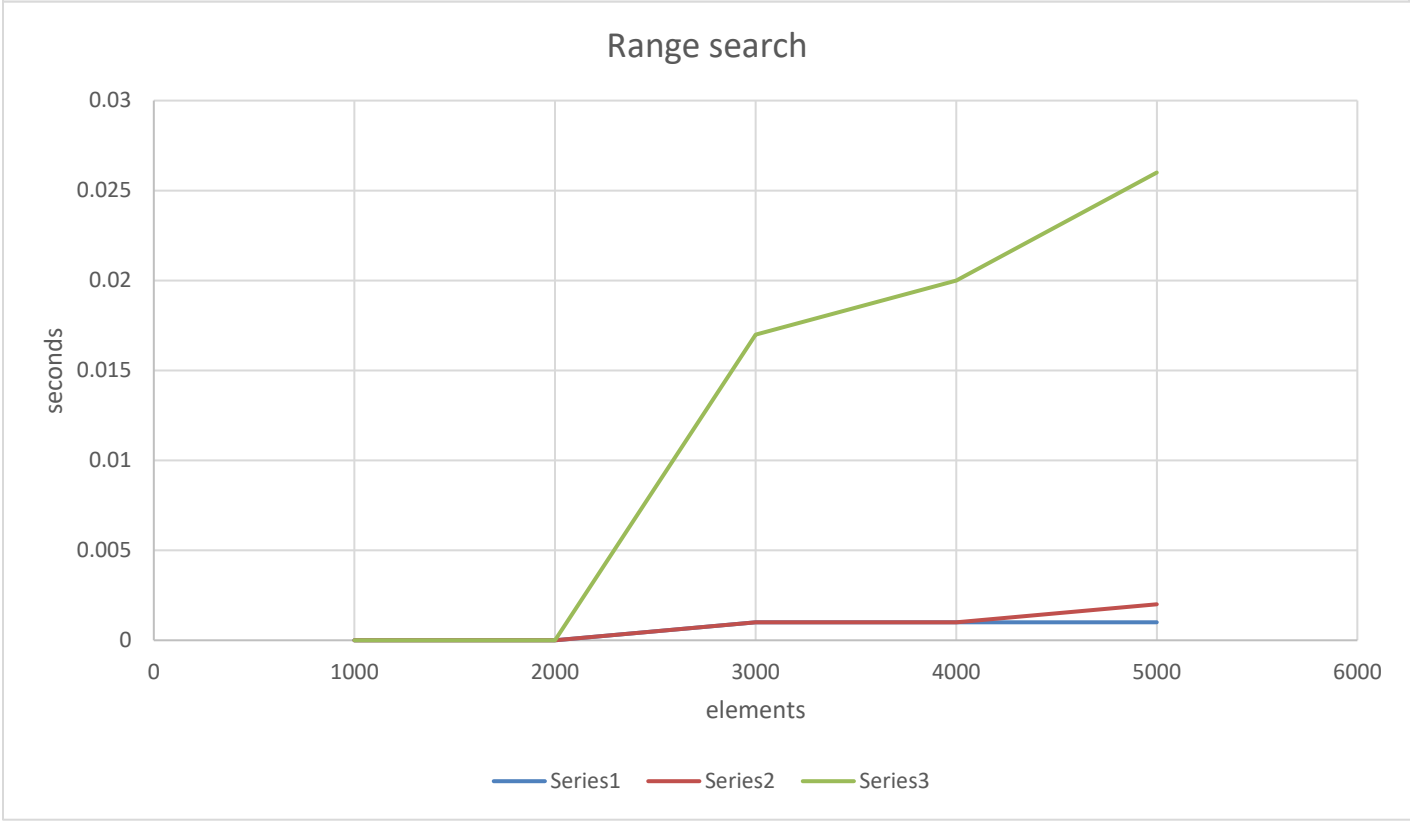
K-D : BLUE

QUAD 1st implementation : RED

QUAD 2st implementation : GREEN




Knn
14
12
10
8
6
4
2
0
seconds
0
1000
2000
3000
4000
5000
6000
elements
Series1
Series2
Series3


Range search
0.03
0.025
0.02
0.015
0.01
0.005
0
seconds
0
1000
2000
3000
4000
5000
6000
elements
Series1
Series2
Series3

Conclusion

The tests we did showed that overall the K-D trees are better in all aspects in 2 dimensions than the Quad.

The first implementation exceeds the second at creation time, knn and range search, but lacks at size where the second one is better. Futhermore the second one can be used to create more than 5k elements where the first one cannot due to recursion depth limit of python.