EC504

Nearest State and County

Gordon Wallace
Yang Hu
Haotian Cheng

Introduction

This project is to apply different algorithms in location searching practice. The purpose of implementations is to look for the nearest state and county for a given point within the United States. A large number of reference points from the official US Board on Geographic Names dataset are used to construct searchable data structures. A user provides a point with longitude and latitude and the program should search through data structures and provides 10 closest reference points. The closest 5 references points start a majority vote to determine which state and county the given point belongs to.

Three different algorithms, R tree, Quad tree and KD tree, are implemented to serve the same purpose as stated above. The details of each algorithm will be discussed. Also, the results and runtimes will be compared and analyzed further in the report.

Other than different algorithms for fundamental searching purpose, a Graphic User Interface (GUI) is also implemented to let user click on points on a map instead of entering coordinates manually, which is more convenient for users and testers to run the program. The user can choose from the provided algorithms to search for locations. A drop down list from 1 to 10 can be selected for the number of nearest reference point on display. Algorithm implementation and GUI establishment are all accomplished in Java under Eclipse environment.

R Tree

The R Tree is essentially a set of concentric rectangular bounding boxes. Each node of the tree contains two coordinates: one corresponding to the top left corner of a bounding box, and the other corresponding to the bottom right of the bounding box. Each node also contains a pointer to its parent and pointers to its children. Bounding boxes of children of an R Tree node must be fully contained by the bounding box of the parent. At the leaf level, the bounding boxes of nodes correspond to the minimum rectangular box that can be placed around an object corresponding to that node. At all other levels of the tree, the bounding boxes are the minimum rectangular box that can be placed around sub-trees (all children of R Trees are R Trees themselves). R Trees can have any number of children per node, although we used a maximum of three in our implementation. R Trees are naturally conducive to spatial applications because the bounding boxes can be used to map objects.

For our application, we created Java functions that read from the file of places from the US Board on Geographic Names. The process of inserting nodes into the R Tree is somewhat complex because all leaf nodes in an R Tree must be on the same level (both by definition and for the sake of search

efficiency). The tree remains balanced with the use of an additional overflow pointer in each sub-tree and a splitting function. When a new node is inserted, the program checks to see whether it fits into an existing bounding box at the level below root. If it doesn't, the program calls a function to determine which box would have to be enlarged the least to fit the node, and performs the enlargement. The insert function is then called recursively on the sub-tree corresponding to that bounding box. The program continues to recurse down the tree, enlarging boxes if necessary, until it is able to insert at the leaf level. If all three possible positions at the leaf level for a given sub-tree are filled, the node will be inserted into the overflow field of that tree. That tree will then be passed to a split function, which creates a new tree at the next highest level, splits the tree in question in two (assigns two nodes from the overflowing tree to the new tree in our implementation, although this isn't necessarily optimal and could be improved on with more time). The new tree is then assigned to the tree at the next highest level. If that one overflows too, the split function continues to be called recursively, all the way up to the root if necessary. If the root overflows, a new root node is created, and the existing root is split into two sub-trees which each become children of the new root. Whenever nodes are removed from a tree in a split process, a function to resize the tree's bounding box is called to ensure it remains a minimum box.

Though we successfully created an R Tree out of the dataset for this project, our search function didn't work consistently. I (Gordon) was responsible for that, so I should be the one to take any penalties for those problems, although for what it's worth I put in some long nights trying to make it happen. That's why the R Tree is included in the insertion plot at the end of the report, but not the search. The following is the procedure we intend to use for searching the R Tree: beginning at the root node, find the subtree that either contains the coordinate in question or is closest to it. Depending on how many points (N) the user is looking for, multiple bounding boxes may remain in consideration until they can be ruled out. The guiding principle is to eliminate boxes for which the closest possible object (location) contained therein is further away than the largest possible distance to the closest object in another box. Boxes are kept in a list of at least the top N possible closest boxes sorted by this criteria, and as additional boxes are considered, this comparison is performed between the new box and the last box in the list to determine whether the new option being considered would definitely contain a closer object. In cases where it is unclear which box contains the closer object, more than N objects have to be added to the list until this ambiguity can be resolved upon descending to a lower level. In the worst case scenario, bounding boxes overlap to the extent that none can be definitively ruled out until leaf level where specific distances are found.

Quad Tree

To implement a Quadtree, we implement three important classes. Coord Class is used to store the

coordinates with longitude and latitude. Node Class is used to store state, county and Coord correspond to the node point from our graph. Finally, we have QuadTree Class itself which can store its top left and bottom right coordinates which represent its border, QuadTree Children(NW, NE, SE, SW divided by its center) and Node if the QuadTree is a leaf.

With the three classes, the strategy to build a Quadtree data structure is very clear. First, we read the graph information and convert each point on the graph into a Node Class object. During the process, we find the graph border and we store all the Nodes in an ArrayList (java class). Then, we build the root QuadTree with the Graph border. Finally, we insert all stored Nodes in our list one at a time. Whenever a QuadTree contains more than one Node, it will divide into more Quadtrees as its children to store those Nodes.

In order to find Kth nearest points of a given point, we implement a search method in QuadTree class. Step 1, instantiate an Array with length K to store nearest points. Whenever a point is inserted into this array, the array get sorted by the distance to the given point simply using insertion sort. (because K is at max 10) Step 2, start from the root to see if it has child that may contain the target point(given point). Do recursive searching till we find the leaf cell that the target point cannot be any deeper. If the leaf cell contains any point, we store it to the array as one of the closest points for now. Step 3, loop through all the QuadTrees(cells) along the search path to see if the closest point it may contain is closer than the worst point stored in the array. If true, dig into that tree(cell) by repeating step 2. If it does have a closer point, replace the worst point in the array and sort the array.

KD Tree

KD tree is a binary search tree data structure that partitions the space to organize points. For a 2-D space, each coordinate divides the plane either horizontally or vertically. There are different ways of constructing a KD tree. The most common one is to sort all coordinates by their x values and take coordinate that has median x value, inserting as root of the tree. Now, the space is divided into halves. Performing sorting and finding median by the y values on each halve, the left side and right side of the root will be divided into halves again. Alternatively finding median of x and y values and divide the subspaces, the space will be divided into small rectangles. However, this method of inserting a KD tree requires sorting and finding median before inserting every node. Therefore, insertion will be relatively slow especially when there are a large number of coordinates. Another way to insert points is to insert without caring about finding medians. Picking an arbitrary point and divide the subspace where that point is located. The method without sorting might have small advantages on insertion, but it is also likely to construct a very unbalanced tree, which causes large runtime on searching.

In this project, the KD tree is implemented without sorting or finding median. Each node (defined as type N) in tree contains latitude, longitude, calculated x and y coordinates (stretch out the curve into a 2-D plane) and a left node and a right node. The tree contains the root node. To find the closest neighbors of a looked-up point, it goes through the tree and find the rectangle that contains the point, and sets the last node it has accessed as the current closest neighbor. Then starting from the root node again, going through the tree and look for closer neighbor that is even closer than the current closest. At each level of the tree, calculate either difference of x value between the point and the node or difference of y values between the point and the node . If the closest distance is smaller than the difference, the right side of that node isn't worth looking into.

If only one closest neighbor is required, after each time finding a node that is even closer than the current closest, the current closest should get updated. However, in this case, 10 closest neighbors are required. Therefore, it is better to construct a list of nodes and keep updating that list (make the list sorted from smallest distance to largest). Keep the last node in the list as the "current closest". It is very likely that within the circle of the looked-up point and current closest node, there are less than 10 nodes. Therefore, it is necessary to enlarge the range of searching by 5 km (it can be any number, but 5 km seems reasonable in this case) each time until 10 nodes are fully filled.

Comparison

The insertion time of Quad tree is larger than the other two data structures, as shown in Figure 1. To look into details of R tree and KD tree in Figure 2, the runtime of R tree is smaller. A weak logarithmic relationship is presented. In practice, there are many different factors that may affect the runtime of each insertion. It is likely that many of the real tests are not perfectly fit what is supposed to be in theory.

R tree did not participate in searching speed comparison. Searching time of KD tree is extremely large since the tree itself isn't constructed balancely. Although it saved some time on inserting without sorting and finding median, it also hindered the searching time as expected. In practice, insertion time may not be as important, since the coordinates are usually pre-loaded and users can access results more quickly if the tree is balanced. In Figure 3, runtime of Quadtree searching gives an average of 0.0537 milliseconds and KD tree give 808.287 milliseconds. The results are not representable enough to make a conclusion of Quad tree is superior than KD tree, since there were some design decisions were responsible for the KD tree's large searching time.

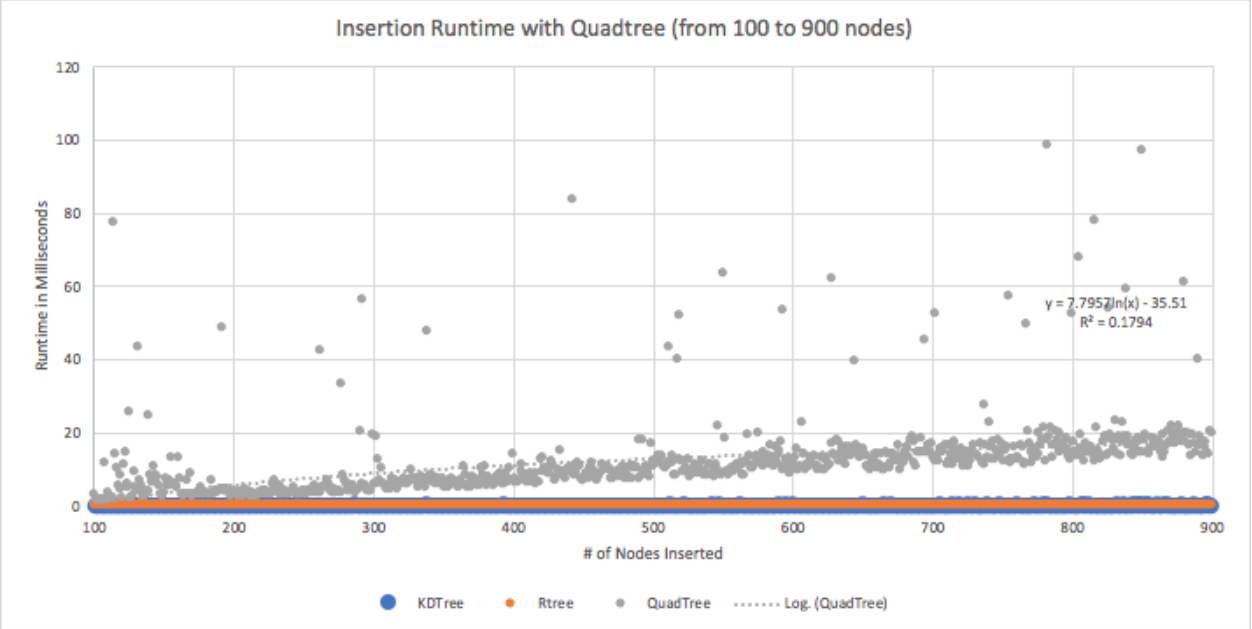

Figure 1: Insertion runtime plot of all three algorithms

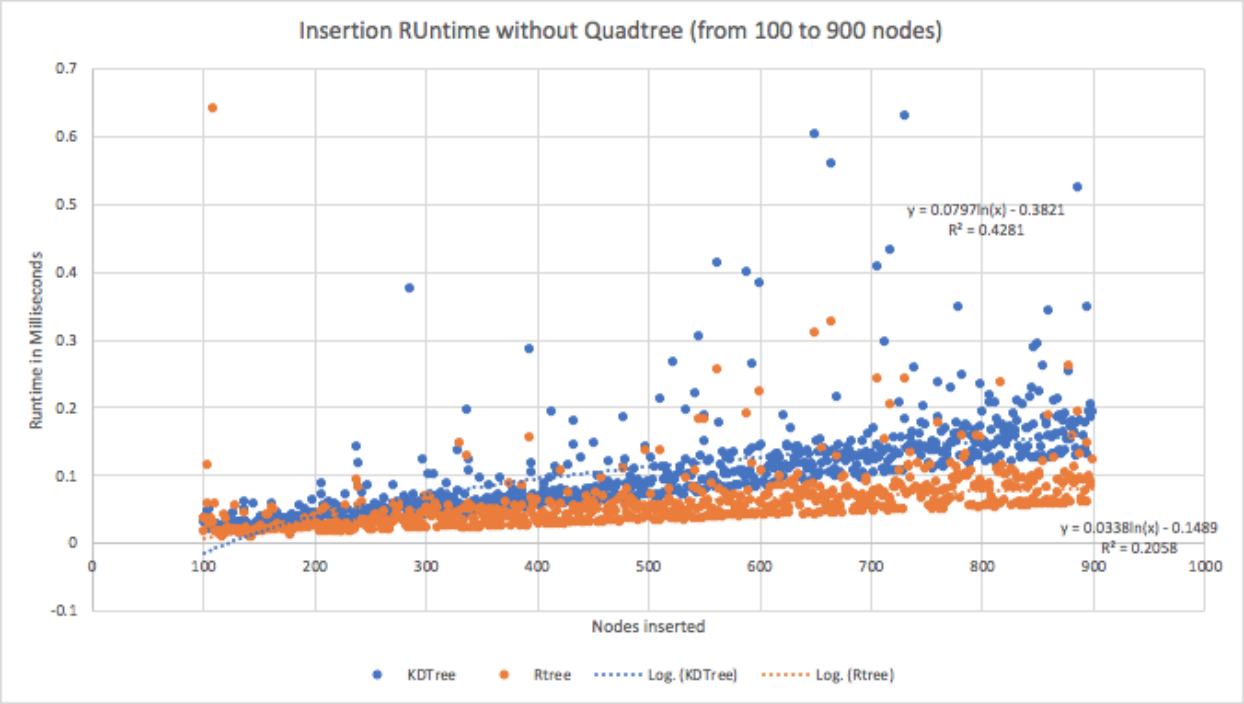

Figure 2: replotted runtime graph of QuadTree and KDTree for detailed comparison

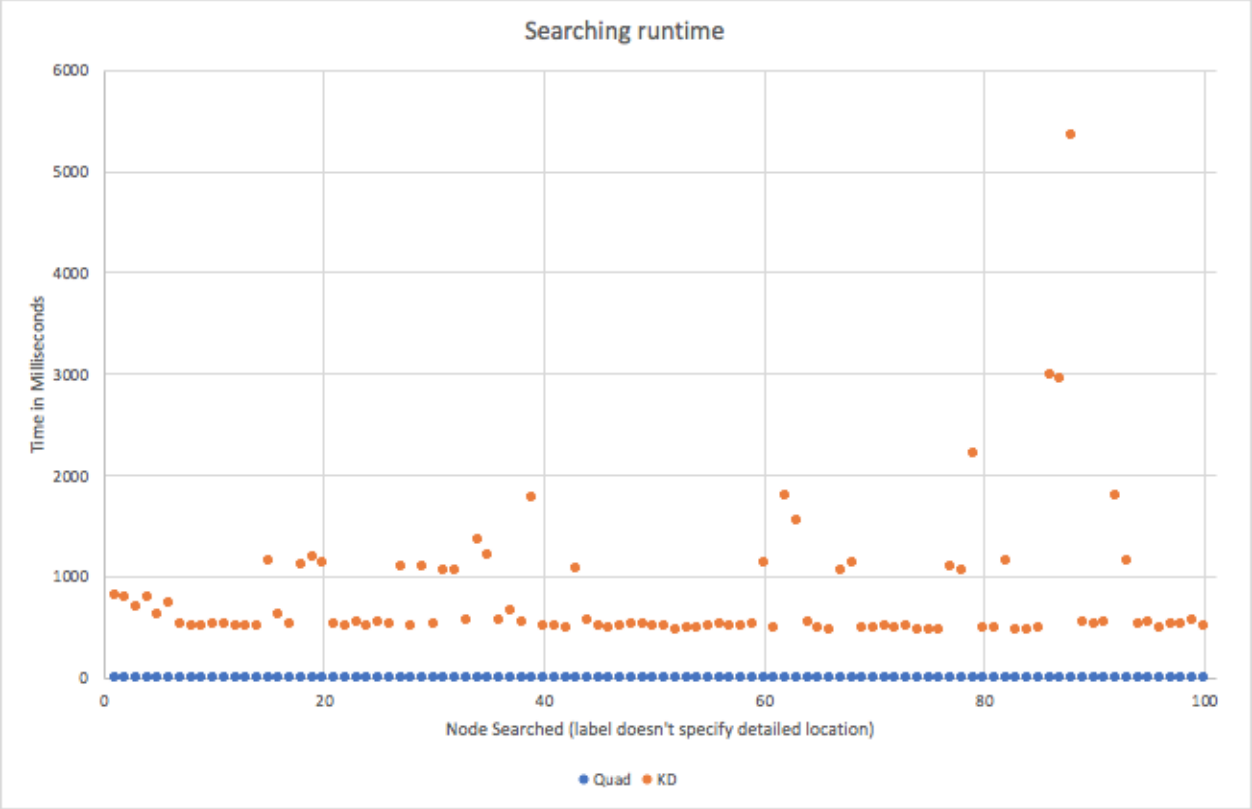

Figure 3: Searching runtime run with 100 random point search

Project Features

1. Storing points from NationalFile_StateProvinceDecimalLatLong us using 3 different data structure (Quadtree, R tree, KD tree)

2. Searching Kth nearest point using 3 different algorithms (Quadtree searching, R tree searching, KD tree searching)

3. GUI for user to directly clicking various location on the US map and display its Kth nearest points and the state and county of the clicked point instead of entering coordinates manually.

References

SAMET, HANAN. “The Quadtree and Related Hierarchical Data Structures .” *UMIACS*, University of Maryland Institute for Advanced Computer Studies, 2 June 1984, http://users.umiacs.umd.edu/~ramani/cmsc878R/p187-samet.pdf.

Huang, Hao, Can Cui, Liang Cheng, Qiang Liu, and Jiechen Wang. "Grid Interpolation Algorithm Based on Nearest Neighbor Fast Search." Earth Science Informatics 5.3-4 (2012): 181-87. Web.

Roussopoulos , Nick, et al. “Nearest Neighbor Queries.” *Department of Computer Science*, University of Maryland, 25 May 1995, http://www.cs.umd.edu/~nick/papers/nnpaper.pdf.