Christine Duong and Julian Trinh
EC504
December 1st, 2015
Project Report

Maps Application

Objective:
In this project, we were asked to build an interactive application that displays a map of the US and enables a multi-million number of reference points. We will be using the reference points extracted from the US Board on Geographic Names dataset given to us on BlackBoard.

The app should support the following two types of queries:

1. If a user clicks on a point in the map, the app should display the decimal latitude and longitude of the clicked point as well as the closest 10 points (from the data sheet) to that point.
2. If a user shades a rectangular area, then the application highlights the set of reference points within the area in the map and counts them.

User Interface:
For the user interface of this project, we decided to make an Android Application. We use the Google Maps API fragment as our main display for the points. We also implement a navigation drawer to display the number of queried points and the points themselves. As part of the assignment we enabled the ability for users to do a 10 nearest query and a rectangular bounds query. For both queries, markers indicating returned locations will be displayed onto the map. Hitting the back button (built in on Android phones) will clear the map of any markers. Clicking on each marker will show it's state/county and coordinate info. For the 10 nearest, a simple tap on the map will display the nearest 10 points to the tapped location. Our implementation of the rectangular query takes the current bounds of the visible map instead of a shaded area (approved beforehand) and returns all points within the visible area. To actually do a rectangular query, one would tap and hold the map and the points will be displayed after computation.

Data-Structure:
To hold the reference points, we decided to use a Kd-Tree (in which *k*, or dimension, is 2). The reason why we chose this was because the data is static, the dimension is extremely small relative to the number of reference points, and it performs ranged queries and computes *k*-nearest neighbors quickly and with ease which are our two main operations.

The idea behind a Kd-tree is to split the space in each dimension every level. In our case, it splits the plane in the x and y dimensions, alternating every level of the tree. To search for a point, we move left or right depending on whether it is less than or greater than the node on that axis. For example, in the following figure, (5,4) is to the left of (7,2) because 5 < 7 but (4,7) is on the right of (5,4) because 4 < 7 and 7 > 4.

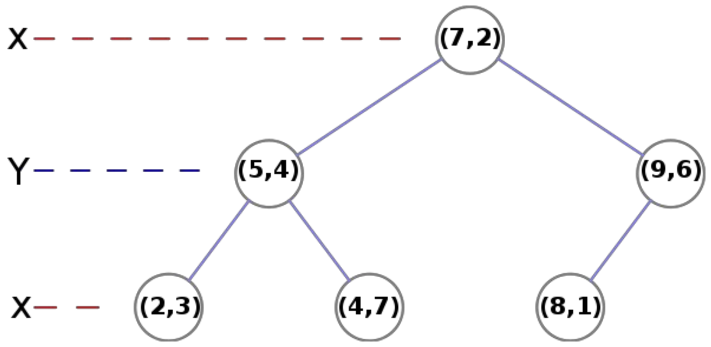
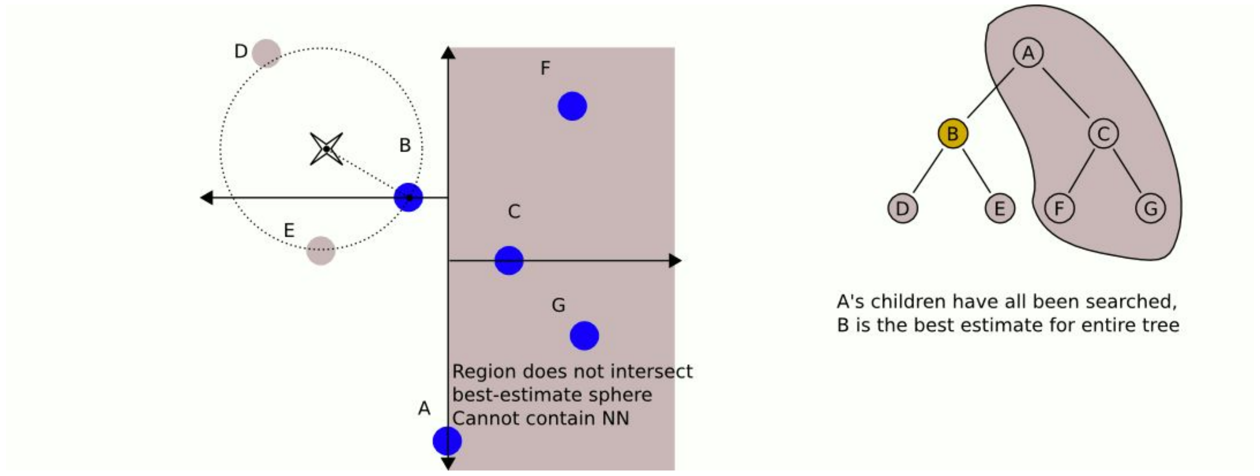


Figure 1: An example of a 2, KD-Tree.

In order to find the nearest neighbor (NN), we search for the target point. Once we reach the node, we call the parent the " current best". We then unwind, recursively towards the root, and does the following steps:

1. Checks if the current node is closer to the current best.
2. Checks to see if there could be any points on the other side of the splitting plane that are closer to the search point than the current best. In other words, it's when the circle around the point (which has radius of the current best) intersects with the plane on the other side. If so, we travel into the other branch (with the same process). Otherwise, we continue going up, eliminating the other branch.

We're done once we reach the root.


D
F
B
C
E
G
A
Region does not intersect
best-estimate sphere
Cannot contain NN
A
B
C
D
E
F
G
A's children have all been searched,
B is the best estimate for entire tree

For 10-NN, we do a similar method except we keep 10 current bests and do not eliminate branches until we have reach that point. The radius of the circle is then the farthest distance (the 10th NN) from the target.

For range queries, we simply traverse the tree by seeing if the regions split by the node intersects with the rectangle bound. For each region that does intersect, we travel down that branch; otherwise we can stop traveling at that node. As we visit a node, we only need to check that it is contained in the rectangle.

Complexity of Operations:
A Kd-tree is not self-balancing; in order to optimize operations, one must always insert the median (of the axis) to evenly split the space. In order to optimize our queries, we pre-processed the original data by continuously finding the median (based on the axis of the level) and rewrote that to a text file so we can directly read from it every time we load.

Operation	***Complexity***
Preprocessing (done once)	$O(n \log^2 n)$
Building the tree (with preprocessed data)	$O(n \log n)$
Building a tree (with original data but taking the median everytime to keep a balanced tree)	$O(n \log^2 n)$
Insert (balanced tree)	$O(\log n)$
Insert (unbalanced)	*Avg: O(log n)* *Worst Case: O(n)*
10-NN	$O(\log n)$
Ranged Query	$O(\sqrt{n} + m)$ where m is the number of points reported

The green boxes represent the complexities of our implementations of the data structure for this project.

Video Demonstration:
A video demonstrating our application can be found here: https://youtu.be/Ju3g2nslMqE

Official List of Duties:

Christine -	Data Structure Research KD Tree implementation (structure, queries) Data File Preprocessing
Julian -	Initial Data Structure Research Android GUI implementation Java socket server implementation

GUI <-> server TCP communication
Live server KD Tree implementation