Autumn 2020
SUID 006339689
Jake Taylor

Final Project
CS 106B
Due: Sunday, November 15

1 Problem description

Imagine your moving to a new city and your looking for a new home. Let's suppose you LOVE SubCity's tuna sandwiches and you want to find the closest restaurant location relative to a potential home location. Luckily you have access to a database of h house locations and r restaurant locations, which may look like this:

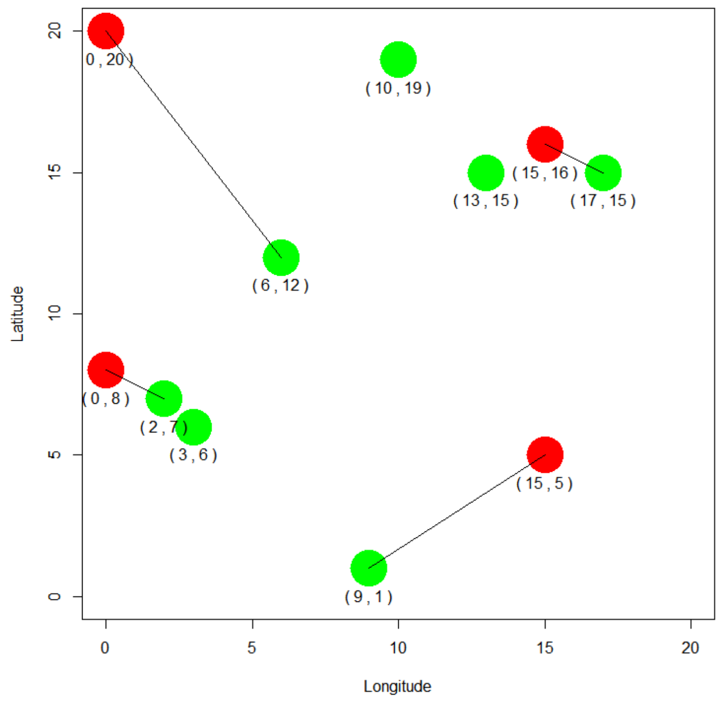

Figure 1: Red home locations are connected by lines to their nearest neighbor restaurant represented in green! *You are not responsible for creating a graphic, just finding a solution!*

Specifically, write two functions:

```
Vector<int> nearestNaive(Vector<Vector<int>>& restaurants, Vector<int> house);
Node* nearestFast(Node* root, Vector<int> house);
```

Where **nearestNaive** accepts as input a **vector of restaurant locations** and your **home location** and returns the **nearest restaurant stored in a vector**. **nearestFast** accepts the **node of a tree data structure** and your **home location** and returns the **nearest restaurant stored in a node**. Note that all "locations" are stored coordinate-wise in vectors (i.e. x = 15, y = 5 is {15,5}). As indicated by the name, the nearestNaive should be a warmup solution that inspires the faster nearestFast and also allows for making rock solid test cases.

You can assume the following details about the problem:

1. The number of restaurants (r) and houses (h) is static; no need to add or remove locations.
2. The $O(hr \log r)$ nearestNaive solution should use **only** *sorting* and *distance computations* while the $O(h \log r)$ nearestFast solution will use a *tree structure* along with a *recursive search algorithm* (we are assuming the tree data structure is built for you beforehand!).
3. The locations of the restaurants are randomly distributed (think Poisson Distribution).
4. For this problem, we will be working in the L_2 distance metric in only two dimensions. That means, that $d(\vec{x}, \vec{y}) = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2}$ where $\vec{x}, \vec{y} \in \mathbb{R}^2$.

Provided to you is starter code that will help you generate the data structure you will search through. Read through **kdtree.h** and understand it's structure. Specifically, our search data structure uses the following definition of a node:

```
// structure that holds a locations and pointers to two other nodes
struct Node {
    Vector<int> location = {0,0};
    Node* left;
    Node* right;
};
```

Additionally, you may find it useful to write helper functions to complete your task. Consider the following suggestions:

```
// calculate squared distance between a query point and a node
double distSquared(Node* candidate, Vector<int> query);
/* TODO: Your code here */

// determine the closer node to a query point
Node* closest(Vector<int> query, Node* node1, Node* node2);
/* TODO: Your code here */

// helper function that computes something recursively
Node* nearestNeighborRec(Node* root, Vector<int> query, int depth, int& numVisited)
/* TODO: Your code here */
```

Finally, you should test your function to ensure correctness! Here is a provided test corresponding to the example from the figure above (feel free to copy this into your own test cases!):

```
PROVIDED_TEST("7 node tree with house location from example"){
    struct Node *root = NULL;
    Vector<Vector<int>> locations =
    {{3, 6}, {17, 15}, {13, 15}, {6, 12}, {9, 1}, {2, 7}, {10, 19}};
    int n = locations.size();
    for (int i=0; i<n; i++)
        root = insert(root, locations[i]);
    Node* NN = NULL;
    Vector<int> house = {15,16};
    NN = nearestFast(root, house);
    cout << "Nearest Neighbor found at x = " << NN ->location[0] <<
            " y = " << NN -> location[1] << endl;
    EXPECT_EQUAL(17, NN ->location[0]);
    EXPECT_EQUAL(15, NN ->location[1]);
}
```

2 Solutions and test cases

2.1 Code for Naive Solution

```
//Using the Priority Queue data structure as a tool to sort and find the NN!
Vector<int> nearestNaive(Vector<Vector<int>> locations, Vector<int> candidate) {
    PQHeap pq;
    Vector<DataPoint> NN(locations.size());
    /* Add all the elements to the priority queue. */
    for (int i = 0; i < locations.size(); i++) {
        double distance = distSquared(locations[i], candidate);
        pq.enqueue({integerToString(i), distance});}
    for (int i = 0; i < NN.size(); i++) {NN[i] = pq.dequeue();}
    DataPoint result = NN[0];
    return locations[stringToInteger(result.name)];
}
```

2.2 Code for Fast Solution

The **nearestFast** uses a recursive helper function called **nearestNeighborRec** that handles the recursion:

```
// Recursive function that finds the nearest neighbor between a
// K-D tree and a query point. Uses backtracking to eliminate options
// from the search area.
Node* nearestNeighborRec(Node* root, Vector<int> query, int depth, int& numVisited){
    if(root == NULL){return NULL;}
    // modulo operation cycling through dimensions
    int currDimension = depth % d;
    Node* nextBranch = NULL;
    Node* otherBranch = NULL;
    // checking to see which half-plane we recurse down
    if(query[currDimension] < (root -> location[currDimension])){
        nextBranch = root -> left;
        otherBranch = root -> right;
    } else {
        nextBranch = root -> right;
        otherBranch = root -> left;
    }
    // search down the half-plane containing our query
    Node* temp = nearestNeighborRec(nextBranch, query, depth + 1, numVisited);
    Node* best = closest(query, temp, root);
    // current radius around best point
    double radiusSquared = distSquared(best -> location, query);
    // 1-d distance calculation to the alternative half-plane while backtracking
    double alternativeDistance = query[currDimension] - root -> location[currDimension];
    // Check the other path while backtracking to the root node
    // note we are comparing squared distances to save computations!
    if(radiusSquared >= pow(alternativeDistance, 2)){
        temp = nearestNeighborRec(otherBranch, query, depth + 1, numVisited);
        best = closest(query, temp, best);}
    return(best);
}
```

2.3 Discussion of Naive Solution

The runtime of naive solution will be $O(h(r \log r + r)) = O(hr \log r)$ since for each h, we have to compute $O(r)$ distance calculations and then sort these distances $O(r \log r)$ and return the nearest neighbor $O(1)$. Although the search time is slow, note there is no pre-computing required; everything is done on the fly with an active query point.

2.4 Discussion of Fast Solution

This entire solution will run in $O(h \log r)$ **given a static, balanced tree**. Each of the h lookups cost on average $O(\log r)$, which is just one tree traversal! This solution differs significantly from the naive implementation by making use of a *spatial index* that is computed beforehand and stored in a *K-D Tree* (in this case, $K = 2$). The benefit of this index is that it allows a query point to be located in a partition of the search space, and then instead of naively checking each point against the query point, we can simply check each partition against a running nearest neighbor candidate that we update as we backtrack back to the root node! One major tradeoff for this faster search speed is that we require a balanced tree for this faster runtime, which our K-D Tree doesn't necessarily produce since each individual element is added in a random order to the tree. In order to fix this, we could presort the data before adding to the K-D tree which would cost $O(kr \log r)$ upfront to build a balanced tree. I used this Wikipedia Reference to validate my claims about building K-D trees.

2.5 Comparison of Approaches

If we know upfront that $q >> r$ where q is the number of queries and r is mostly static, then K-D trees make the most sense. This is why in most spatial applications, some sort of spatial index is used for lightening fast searches. In an application where we would expect to add nodes quite frequently, k is a very large, or the points are not random (clustered or dispersed) the K-D tree approach doesn't do as well. Thus, the naive implementation or an approximate nearest neighbor might suffice for a different application.

2.6 Test Cases

A simple test case was generated by hand and validated against the naive implementation. Then, a more challenging test case was generated randomly and validated against the naive implementation. Using these results, we are finally able to validate against the fast solution and show a graphic output as shown below. Finally, the number of nodes visited is printed in the fast solution which for 1000 locations, is typically under 30. Quite a big improvement from naively sorting all 1000!

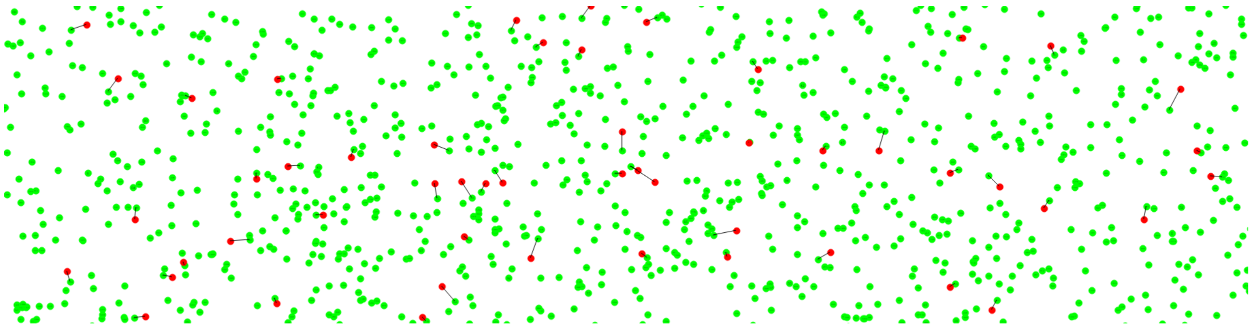

Figure 2: Test homes connected to their 1-NN restaurants

3 Problem motivation

The conceptual motivation for this problem is quickly finding nearest neighbors either in spatial databases (such as a Geographic Information System) or in machine learning algorithms (specifically K-Nearest Neighbor classification and regression). Working through this problem would allow students to see ways to cleverly cut down on computations while backtracking out of a recursive solution. Meanwhile, you will have to manage pointers in a tree struture, which really cemented this concept for me personally. Also, seeing which applications the 'fast' solution is valid for is good to learn that the 'naive' solution isn't always that naive!

My personal motivation was that during my internship last Summer, the devs on my team kept saying "and then we load our C++ function that uses K-D trees" and I had no idea what they meant. So this project was an attempt at understanding the most elementary version of this algorithm in a new language (for me). The nearest neighbor twist also came from using nearest neighbors as part of a locally weighted regression algorithm we had to build in STATS 305 (I coded this naively and wondered how I could do this faster in R!). This problem really allowed me to combine my past statistical, CS 161, and internship knowledge to code something in a new language (C++) that is very useful.

4 Concept mastery, common misconceptions

The course learning goals assessed by this problem are the same as the proficiencies a student would need to complete it. They are: exposure to algorithmic analysis, pointers, trees, recursion, backtracking, sorting, pqheaps, and C++. So perfect for CS 106B! Some difficulties that came up in designing the problem was identifying the most appropriate spatial indexing algorithm to use. Alternatives include R-trees and quad-trees. K-D trees were chosen because of their simplicity and wide use in industry. Some common misconceptions that might pop up is cycling through the dimensions on each level of recursion using the modulo operator (this was tricky for me at first!). But once you realize this is just a binary search that iterates over the different coordinate axes, it is a breeze! Some bugs that may develop might be incorrect backtracking along with the typical pointer issues (accessing null pointers, keeping track of the root of trees, correctly backtracking with pointers, etc.).

4.1 Ideas for extensions

Some ideas presented themselves as natural extensions while creating this project. They include:

1. Generalizing the dimension k from 2 to any arbitrary dimension. Then analysis can be done on the efficacy of 1-NN search in a high dimensional setting (i.e. *Curse of Dimensionality* in Statistics literature).

2. Generalizing the distance metric from euclidean to other distance metrics. For example the L_1 "taxicab" distance may be more appropriate in cities, but how would this affect the search radius when backtracking out of a recursive solution? What about distances parameterized along a curved path?

3. Generalizing the 1-NN search to the famed, K-NN search. This could be implemented efficiently by using a priority queue (combining elements of the naive and fast solution!)

4. Using different tree structures. What if instead of points, we had line features? Then a R-tree would be more appropriate.

4.2 References

I used this video heavily: https://www.youtube.com/watch?v=Glp7THUpGow
Along with the pseudocode at: https://youtu.be/Glp7THUpGow?t=338.
Further, my naive test example was adapted from: https://www.geeksforgeeks.org/k-dimensional-tree/.
Used this reference mostly for inspiration: https://blog.mapbox.com/a-dive-into-spatial-search-algorithms-ebd0c5e39d2a

4.3 Implemented Extension

I had enough time to implement the K-NN search using a bounded priority queue and was able to pass a small test (the example given in the prompt) as well as graphical tests on larger datasets ($r > 10,000$). As nearest neighbors are identified while backtracking, they are added to the priority queue but only retained if they are smaller than the current neighbor at the end of the queue. One small change is needed, distances must be made negative in the PQueue to ensure the largest NN is dequeued if another candidate has a smaller distance. Below is a graphic showing $r = 10,000$, $h = 30$ and K chosen randomly out of the set $\{100, ..., 200\}$. Source code is included in the code submission but is omitted from the writeup for brevity.

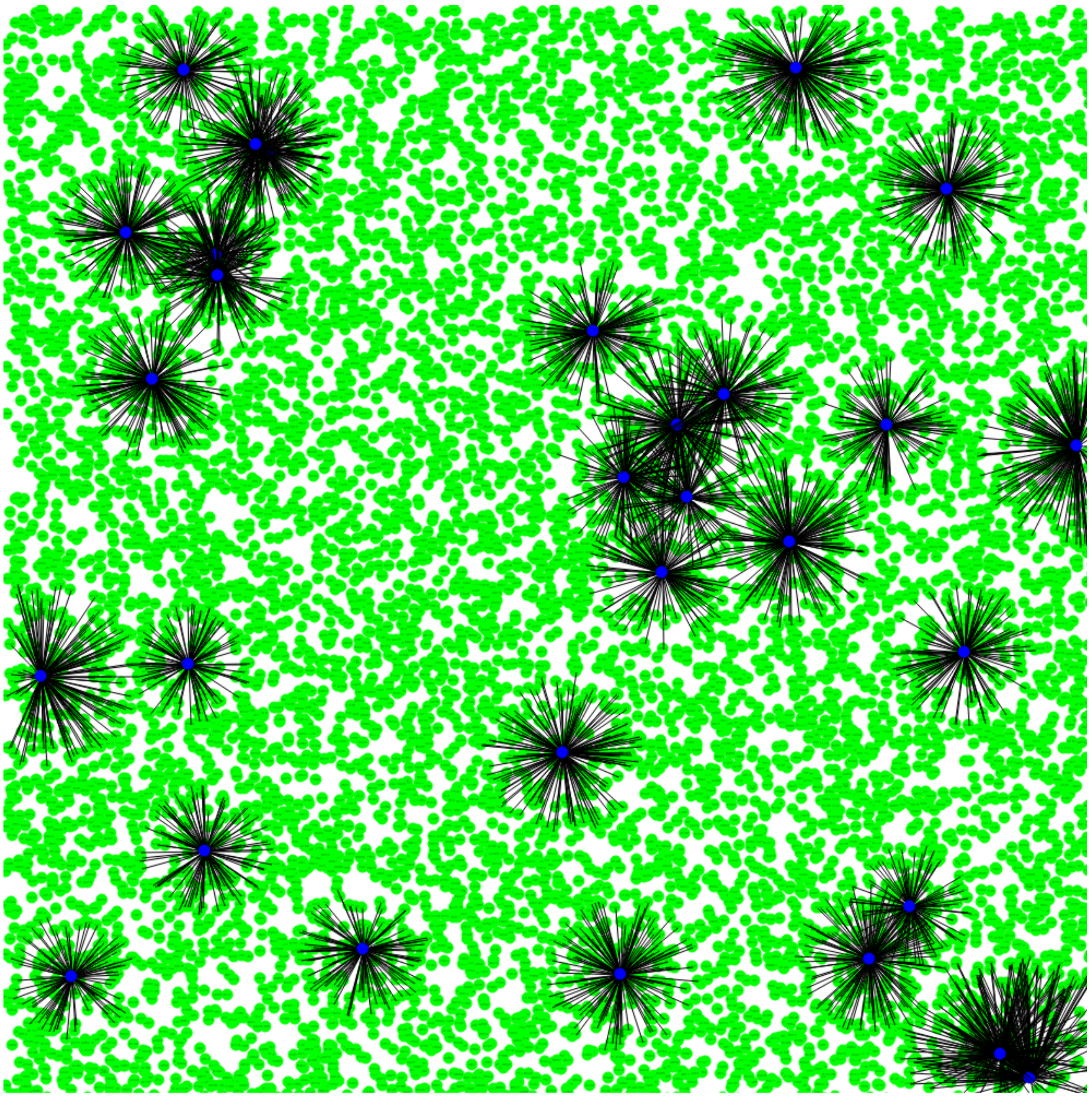

Figure 3: Test homes in blue connected to their K-NN restaurants where K is chosen randomly