Nearest State Finder Report

Team Information
Yuzhe Xu yx8756a@bu.edu U11954992
Jingyi Zhang jyz0328@bu.edu U26578499
Cynthia Young mcyoung@bu.edu U87972791

Content

Abstract

This project is to create a system, which can efficiently find the nearest state of the given input location. In this system, users are asked to input a *latitude* and *longitude* of a targeted location, and are required to input an integer value *k* between 1 and 10, for the number of reference points they want to find. Our implementation extracts data from the United States Geological Survey (USGS) website, computes distances using the Haversine formula, employs a KD tree for neighbor search, and confirms states via majority voting. Note that our source data only includes reference points within the United States. We can only ensure the accuracy for input locations inside the US.

Files

In this section we will list all filenames, explain their function and how to run the code.

Data < 2023_Gaz_counties_national.csv> : Downloaded from USGS. It includes the state (USPS), county (NAME), LATITUDE (INTPTLAT), and LONGITUDE (INTPTLONG), which are needed for finding nearest *k* reference points.

USPS	GEOID	NSICOD	NAME	ALAND	AWATER	ALAND_SQMI	AWATER_SQMI	INTPTLAT	INTPTLONG
AL	1001	161526	Autauga County	1539631461	25677536	594.455	9.914	32.532237	-86.64644
AL	1003	161527	Baldwin County	4117725048	1132887203	1589.863	437.41	30.659218	-87.746067
AL	1005	161528	Barbour County	2292160151	50523213	885.008	19.507	31.870253	-85.405104
AL	1007	161529	Bibb County	1612188713	9572302	622.47	3.696	33.015893	-87.127148
AL	1009	161530	Blount County	1670259100	14860281	644.891	5.738	33.977358	-86.56644
AL	1011	161531	Bullock County	1613083468	6030667	622.815	2.328	32.101759	-85.717261
AL	1013	161532	Butler County	2012002548	2701199	776.839	1.043	31.751667	-86.681969
AL	1015	161533	Calhoun County	1569248377	16534047	605.89	6.384	33.770516	-85.827909
AL	1017	161534	Chambers County	1545068570	16988729	596.554	6.559	32.915504	-85.394032

load.cpp under src folder: Source code for finding the nearest state and county.

makefile: File used to compile load.cpp

input.txt and input_output.txt: input.txt is for testing and it is under the input folder. We also created an example.txt file for testing the code (it is found in the main folder directly, not in the input folder). The structure of all input file is as follows:

latitude

longitude

k

Any outputs will display in the console and be written in the corresponding output files, found in the output folder. **input_output.txt** is output file for **input.txt** ,

example_output.txt is output file for **example.txt**.

Additionally, you may also see a github/workflow as well as vscode folders, these are used to configure the environment for VScode and C++, to ensure compatible VScode and C++ versions.

Instructions For Use

Use the command [**make**] on the terminal to compile our source file **load.cpp**.

Type [**make run**] on the terminal to test the i**nput.txt**..

Our makefile only runs **input.txt**, if you want to create other test input files, you should use **./load xxxx.txt** instead of using [**make run**]. For example, we can run **example.txt** by using **./load example.txt**.

Type [**make clean**] on the terminal to remove all the compiled files from the directory.

Methods and Implementation of Code

Loading Reference Points

We imported reference points, comprising latitude, longitude, state, and county data, into the KD-tree from the file named <2023_Gaz_counties_national.csv>.

Distance Computation: Haversine Distance Formula

The distance between two points (reference point and the input point) is calculated using the **Haversine Distance formula**, which involves latitude and longitude.

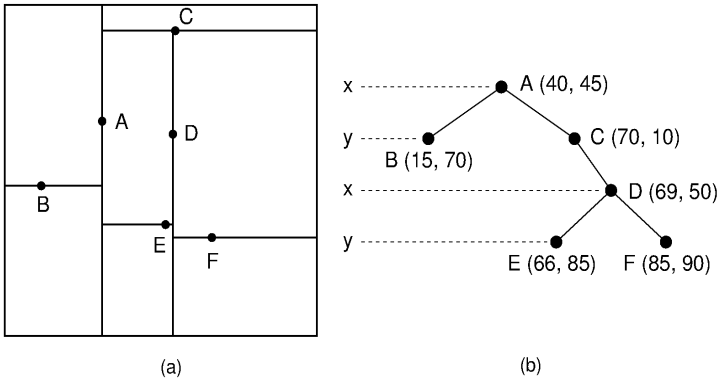
$a = \sin^2(\Delta lat/2) + \cos(lat1) * \cos(lat2) * \sin^2(\Delta lon/2)$
$c = 2 * \text{atan2}(\sqrt{a}, \sqrt{1-a})$
$distance = R * c$
R is the radius of the Earth

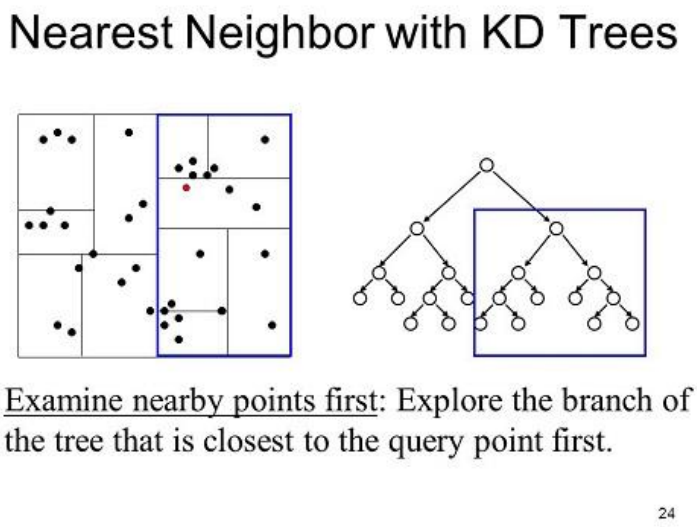
The Haversine formula takes into account the curvature of the Earth's surface, making it more accurate for longer distances.

KD Tree and Nearest Neighbor Search Algorithm

A KD-Tree is a data structure that is particularly suitable for multi-dimensional space

search queries. It has efficient space division; inserting points into a KD Tree effectively divides the space into smaller regions, each containing similar points. It allows for an optimized usage of the nearest neighbor search algorithm by quickly narrowing down the search area. It checks if the current node is closer and then selects a subtree to search deeper. If there are closer points in the subtree then the search path will be explored further (pruning). If a subtree on the other side is likely to contain a closer point, then that subtree is also searched.





Example of how a KD-Tree partitions space via each insertion

Function Implementation for KD-Tree and Nearest Neighbor Search Algorithm

In our source code, we define an internal class named KD-Tree Node to represent the nodes of a KD-Tree. Each node contains geographical coordinates, a left child node and a right child node right.

The process of constructing the KD tree is implemented recursively by inserting new nodes into the tree using the *insertRec(InsertRecursive)* function. At each node, we select the tangent axis based on the current depth and insert the new node into either the left or right subtree depending on the value of the tangent axis.

```
Node* insertRec(Node* root, std::string state, std::string county,
double x, double y, unsigned depth) {
        if (root == nullptr) {
            count++; // Increment the counter when a new node is
created
            return new Node(state, county, x, y);
        }

        // Calculate current dimension (cd)
        unsigned cd = depth % 2;

        if (cd == 0) {
            if (x < root->x) {
                root->left = insertRec(root->left, state, county, x,
y, depth + 1);
            } else {
                root->right = insertRec(root->right, state, county,
x, y, depth + 1);
            }
        } else {
            if (y < root->y) {
                root->left = insertRec(root->left, state, county, x,
y, depth + 1);
            } else {
                root->right = insertRec(root->right, state, county,
x, y, depth + 1);
            }
        }

        return root;
    }
```

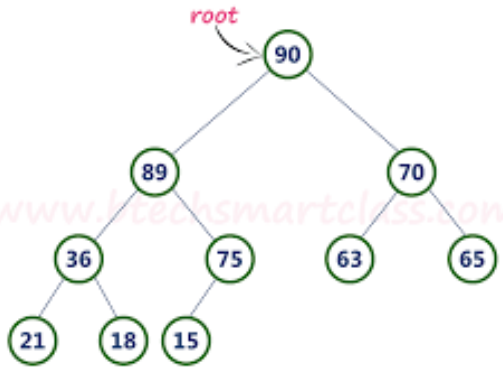
The nearest neighbor search algorithm is also implemented recursively, using the *findKNearestRec(FindNearestRecursive)* function to find the nearest node. At each node, we calculate the distance between the target position and the current node position and update the nearest node and the nearest distance. Then, based on the current depth and the value of the tangent axis, we choose to search further in the left or right subtree, and also determine whether we need to search in another subtree based on the distance from the current tangent axis. Repeat this process until the search path is empty.

```
//search nearest
    void findKNearestRec(Node* root, double x, double y, int k,
std::priority_queue<KdNodeDistance, std::vector<KdNodeDistance>,
Compare>& pq, unsigned depth) {
        if (root == nullptr) return;

        double dist = haversineDistance(x, y, root->x, root->y);

        if (pq.size() < k) {
            pq.push(KdNodeDistance(root, dist));
        } else if (dist < pq.top().distance) {
            pq.pop();
            pq.push(KdNodeDistance(root, dist));
        }


        // Determine which subtree to search
        unsigned cd = depth % 2;
        Node* nextBranch = (cd == 0) ? ((x < root->x) ? root->left :
root->right) : ((y < root->y) ? root->left : root->right);
        Node* otherBranch = (cd == 0) ? ((x < root->x) ? root->right
: root->left) : ((y < root->y) ? root->right : root->left);

        findKNearestRec(nextBranch, x, y, k, pq, depth + 1);

        // Check if we need to explore the other branch
        double distToPlane = (cd == 0) ? std::abs(root->x - x) :
std::abs(root->y - y);
        if (pq.size() < k || distToPlane < pq.top().distance) {
            findKNearestRec(otherBranch, x, y, k, pq, depth + 1);
```

```
        }
    }
```

Max Heap Usage

We used a max heap to store the closest points found and their distance from the query point. It allows us to efficiently keep track of the *k* closest points found so far. First we compare the new node with the node at the top of the queue. If the new node distance is smaller than the distance of the top node, then the top gets popped and the new node is pushed in.



Majority Vote for States and Counties

Once the nearest K reference points are found, the system performs a **Majority Voting** algorithm among the K nearest points, to determine the state of the searched point. The algorithm essentially counts the occurrences or preferences of different options and selects the one that appears most frequently or has the highest count.

Demo/Sample Results

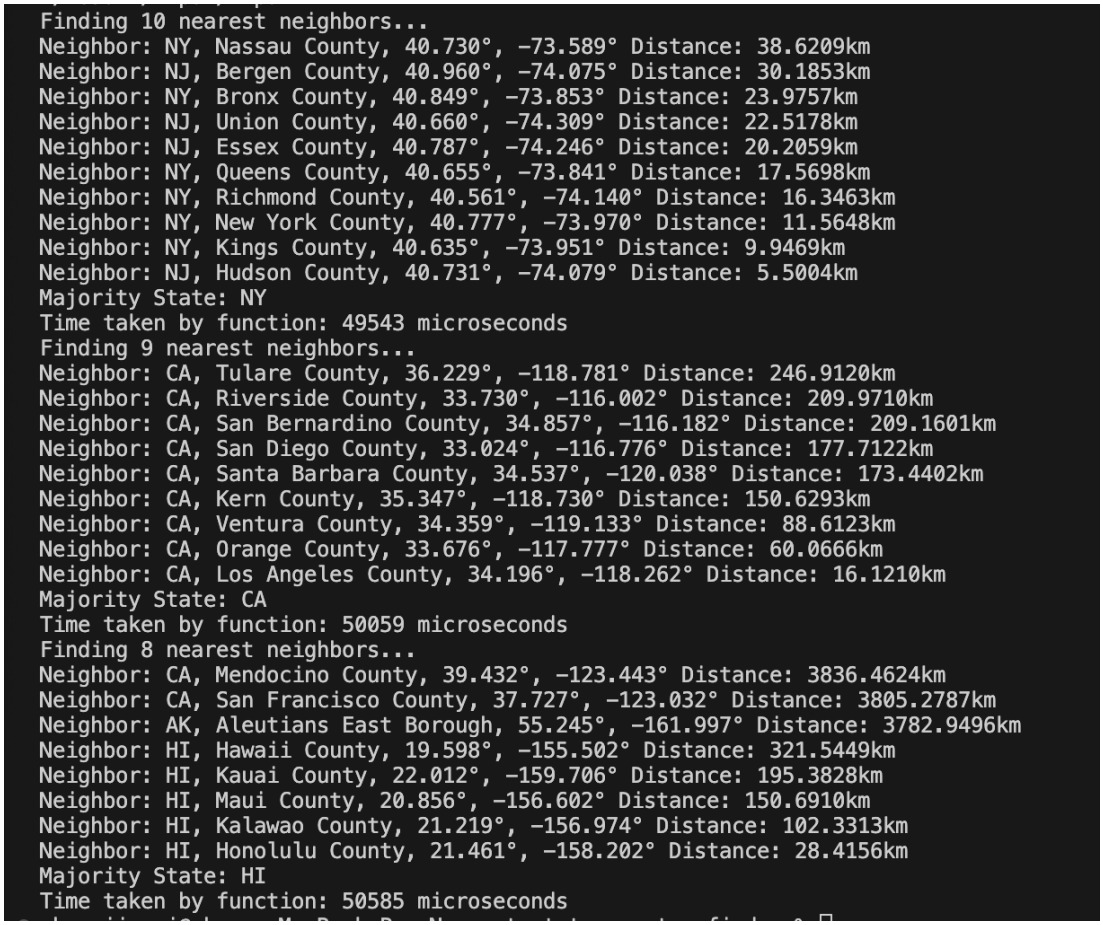
Here, we demonstrate the results utilizing input.txt and example.txt as our test files.

Our input.txt appears as follows:

40.6892
-74.0445
10
34.0522
-118.2437

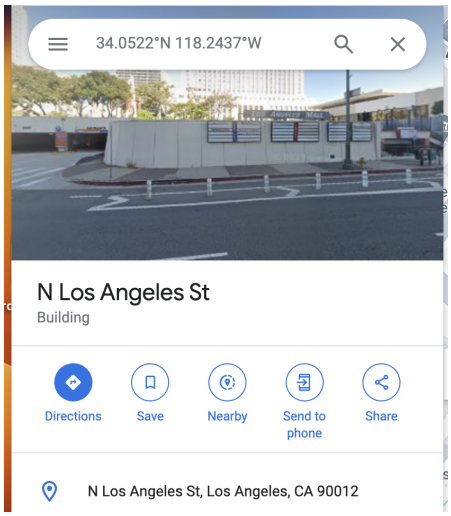
8

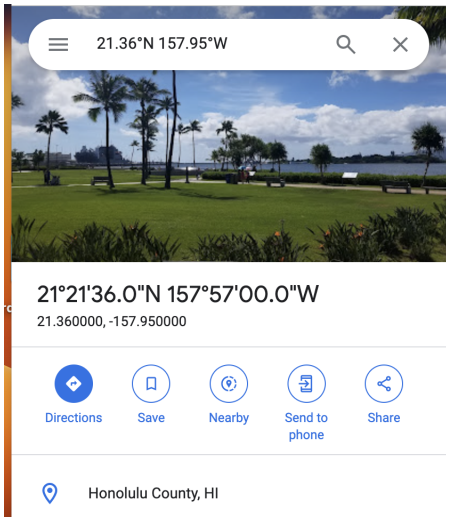
9
21.36
-157.95
8
Which means
Latitude 1
Longitude 1
K 1
…….
The negative longitude means that it is a west longitude and the positive latitude means that it is a north latitude. For the purposes of demonstration, we input points of famous US places. [40.6892° N, 74.0445° W] is the Statue of Liberty, [34.0522° N, 118.2437° W] is Los Angeles, and [21.36° N, -157.95° W] is Pearl Harbor in Hawaii. After running [**make**] to compile load.cpp and after running [**make run**], we get the following output:

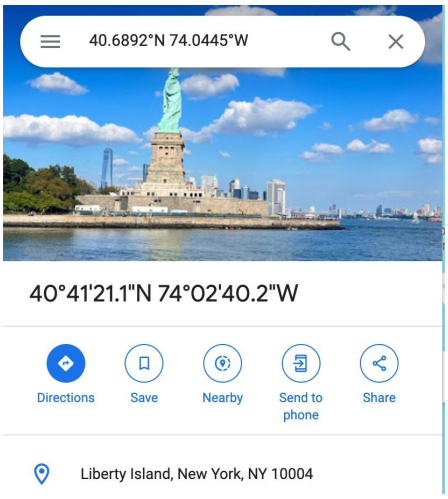


This result is also stored as input_output.txt under the output folder. By using our nearest state finder, **we find that [40.6892° N, 74.0445° W] is in NY (New York) , [34.0522° N, 118.2437° W] is in CA (California), and [21.36° N, -157.95° W] is in HI**

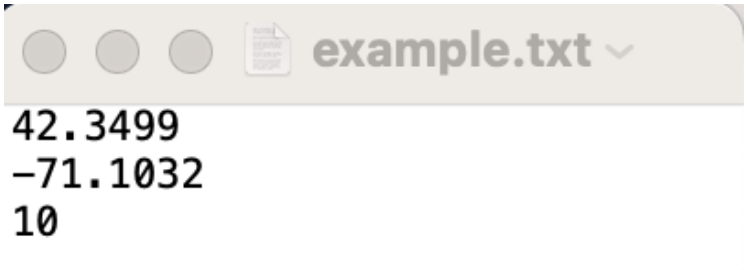
(Hawaii). To confirm that our results are accurate, we cross referenced using Google Maps, and conclude that our nearest state finder shows the right state for each input location.



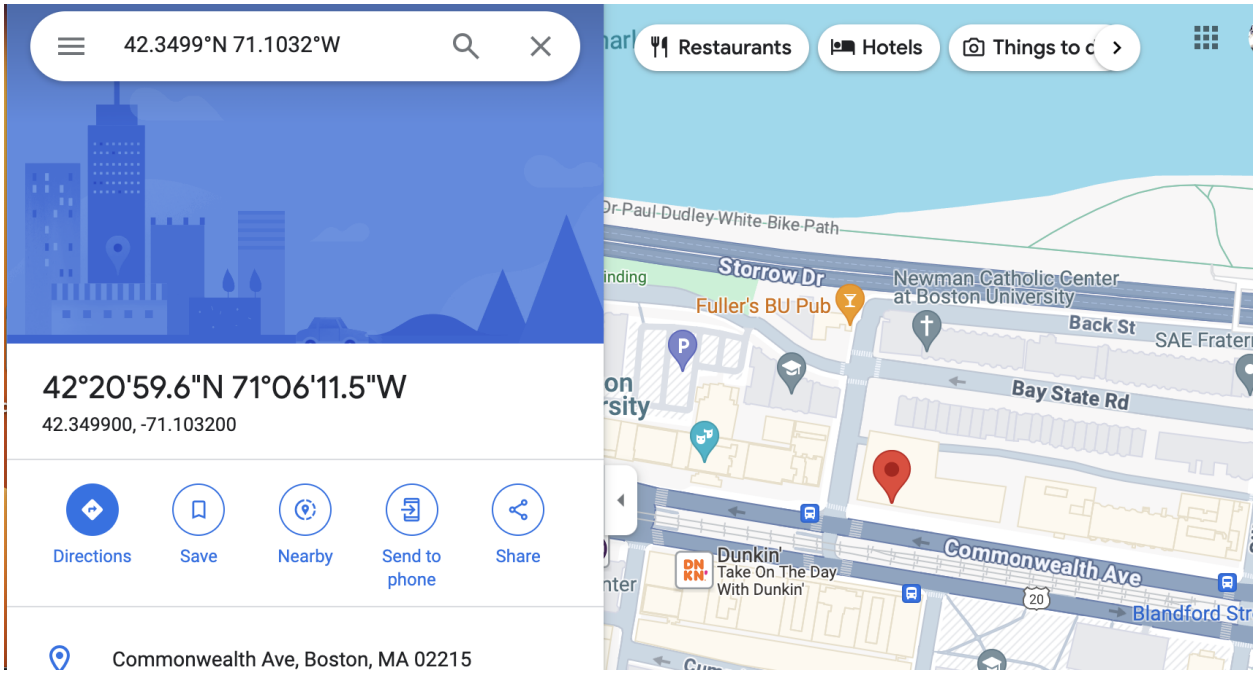




Then we compiled example.txt which showed the same result with Google Maps.




```
zhangjingyi@zhangs-MacBook-Pro Nearest-state-country-finder % ./load example.txt
Finding 10 nearest neighbors...
Neighbor: RI, Bristol County, 41.707°, -71.287° Distance: 73.0934km
Neighbor: NH, Rockingham County, 42.989°, -71.099° Distance: 71.0281km
Neighbor: MA, Worcester County, 42.312°, -71.940° Distance: 68.9414km
Neighbor: MA, Bristol County, 41.749°, -71.089° Distance: 66.8733km
Neighbor: RI, Providence County, 41.870°, -71.579° Distance: 66.2446km
Neighbor: MA, Plymouth County, 41.987°, -70.742° Distance: 50.1298km
Neighbor: MA, Essex County, 42.643°, -70.865° Distance: 37.9704km
Neighbor: MA, Middlesex County, 42.482°, -71.395° Distance: 28.0771km
Neighbor: MA, Norfolk County, 42.172°, -71.181° Distance: 20.8224km
Neighbor: MA, Suffolk County, 42.339°, -71.018° Distance: 7.0945km
Majority State: MA
Time taken by function: 36909 microseconds
zhangjingyi@zhangs-MacBook-Pro Nearest-state-country-finder %
```



Reference

[1]https://www.census.gov/geographies/reference-files/time-series/geo/gazetteer-files.html