Nearest State/County Finder: Leveraging Geospatial Data for Efficient Query Processing

Team 3
Aowei Zhao aowei8@bu.edu
Haolin Ye haolinye@bu.edu
Jiayu Wang jiayuw@bu.edu
Sen Wang senwang@bu.edu
https://github.com/Leaf-hl/EC-504-Project

1 Abstract

This project implemented in C++ is used to find the K nearest neighbor state/county in the US for one specified coordinate. This system will first load a huge number of reference points in the US. Then it will allow the user to specify one location by inputting its latitude and longitude. K, the quantity of nearest neighbors which should be a number from 1 to 10, is also required for the user to assign. With these parameters, the system will return results with two methods: KD-Tree searching and linear scanning. The corresponding time consumed will also be shown. According to the result of test cases, KD-Tree searching consumes less time compared to linear scanning. However, if we consider the time of KD-Tree building, the result will be the opposite.

2 Implementation

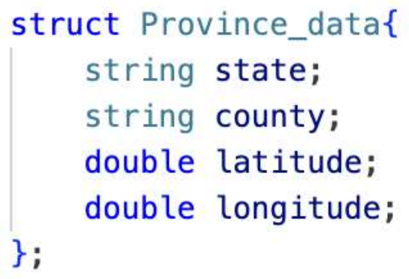
2.1 Program Workflows

This project is mainly divided into two tasks: Data Structure Construction and Efficient Query Processing. With this thought, our program is implemented in the following four steps.

a. The program automatically loads geographical data from the dataset(the default file is “data.txt”) and stores them in one struct Province_data. The name of the dataset can be changed in the source code. The number of nearest neighbors(k) can also be changed.
b. The program asks the user to input two parameters: the latitude and the longitude of the point.
c. The program builds the KD-Tree, finds the k nearest points with KD-Tree and uses linear scanning to find the k nearest points. The distance between two points is computed by the distance calculator module. The consumed time of these three steps and the information of results will then be printed.
d. By majority voting, the state and county of the input point can be inferred from the output points.

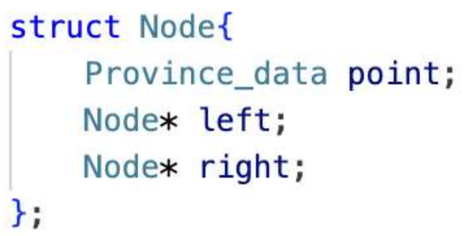
2.2 Data Structure Construction

To store data and build the KD-Tree, we first create two data structures in our program. The first one is Province_data, which consists of two string types and two double types. This struct is used to store all location information.



```
struct Province_data{
    string state;
    string county;
    double latitude;
    double longitude;
};
```

The second one is Node, which is used to build the KD-Tree. Since each location can be viewed as one point on the map, we created one element with Province_data type. The other two elements are pointers. By building left and right child nodes, all nodes will connect with each other and form the tree.



```
struct Node{
    Province_data point;
    Node* left;
    Node* right;
};
```

As for the k nearest points, we use min-heap to store their information. The point with the lowest distance(or highest priority according to the requirement) is located in the front while the point with k-th distance is put at the back. Min-heap can ensure the efficiency of data storage.

2.3 Distance Calculator

According to the proposal, the distance between any two points on earth can be calculated with below formulas

- $x = (\lambda_2 - \lambda_1) \cdot \cos\left(\frac{\phi_1+\phi_2}{2}\right)$
- $y = \phi_2 - \phi_1$
- $\text{Distance} = \sqrt{x^2 + y^2} \cdot R$

, where ϕ is the latitude, λ is longitude, R is earth’s radius (we take the value of mean radius which is 6371 km)

2.4 KD-Tree

Since the dataset is the coordinates of points in North America, there is no need to consider the longitude problem(such as one point has negative longitude while the other one has positive longitude). Therefore, we take longitude as the x axis and latitude as the y axis. Then we build the KD-Tree in 3 steps. Step 1: choose the middle node on the x-axis and draw a vertical line. Step 2: choose the middle node on the y-axis and draw a horizontal line. Step 3: repeat steps above until all the nodes have drawn lines.

After building the KD-Tree, the searching part mainly consists of four steps. Step 1: Down search. Compare from the root to bottom: X-Y-X-Y-X… Step 2: calculate the distance. Update the storage or discard this point. Step 3: up searching. Go to the upper level and calculate the distance. Step 4: decision. Decide whether to search other subtrees. Through this process, the program realizes one efficient k nearest neighbors search.

2.5 Linear Scanning

Unlike the KD-Tree method, this linear approach compares the query point with every point in the dataset to find the k nearest neighbors. The first k points will be directly stored in ascending order of distance. For the rest points, they will be compared with those points one by one to decide whether to replace or discard. Its source code is simpler to write but its performance is less efficient for large datasets.

2.6 Majority Voting

We design a set of weights {1.5, 1.4, 1.3, 1.2, 1.1} to predict the state and county of the input point. We assign the largest weight to the nearest point and the smallest weight to the 5-th nearest point. Its principle is easy to understand. When more than half of points belong to the same state and county, the input point is most likely to be located in this area. If some states and counties own the same proportion of points, the most possible location of the input point should be the area whose points are closer.

3 Instructions for Running the Code

If you want to run the code on your computer, you just need to follow the next steps.

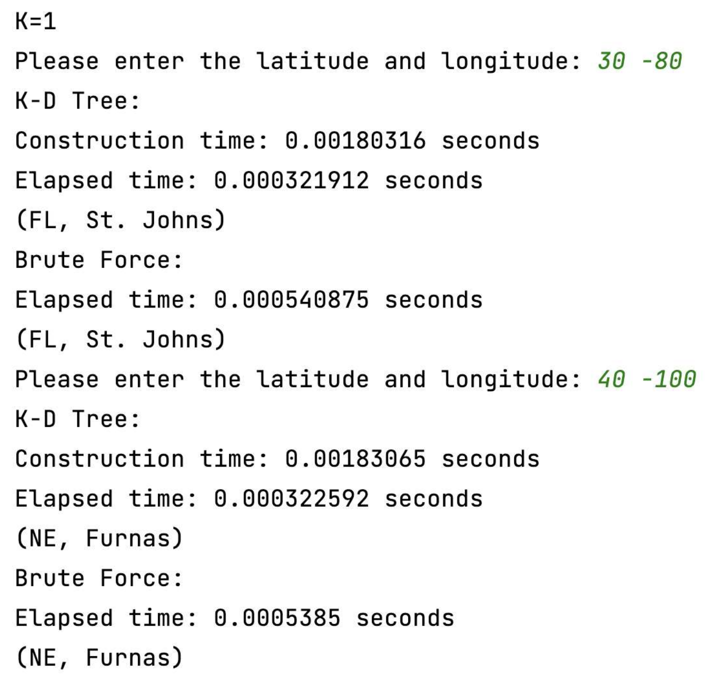
a. Open your terminal
b. Input “git clone https://github.com/Leaf-hl/EC-504-Project.git”. Press Enter.
c. Input “cd EC-504-Project”. Press Enter.
d. Input “make”. Press Enter.
e. Input “./main”. Press Enter.
f. Input two parameters “latitude longitude”. Press Enter.

Then the program will calculate and print the corresponding result. After finishing, you can clean up the compiled file by inputting “make clean”.

4 Result

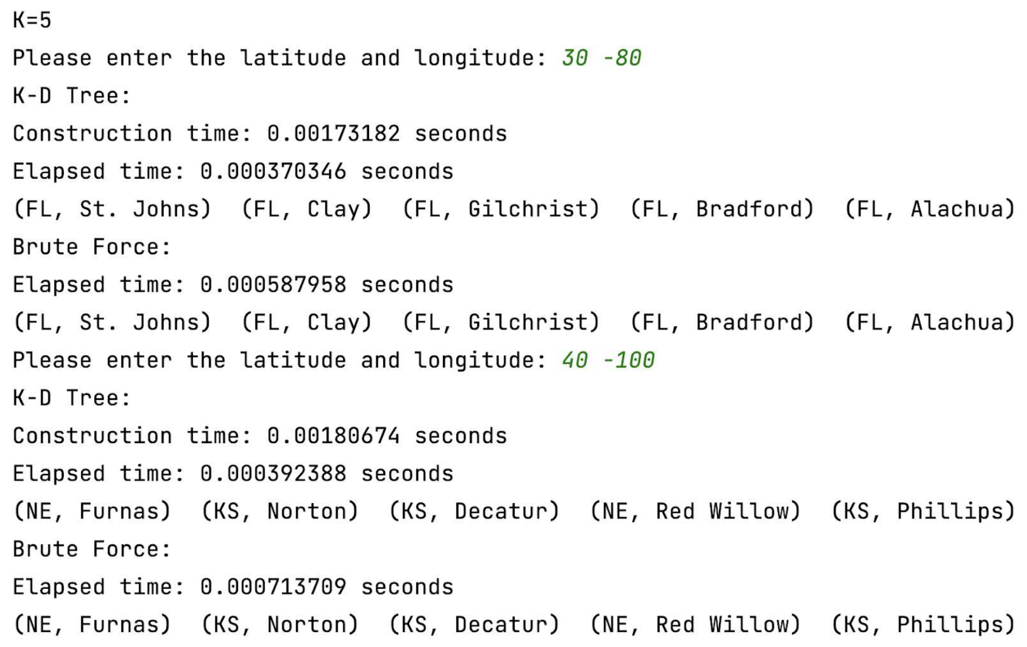
We tested our program with different input using both K-D tree algorithm and brute force method (Minheap) and calculated their running time. In the experiment, we use the same input with different values of K, the sample output with K equals to 1, 5, 10 are shown respectively below:

When K = 1:



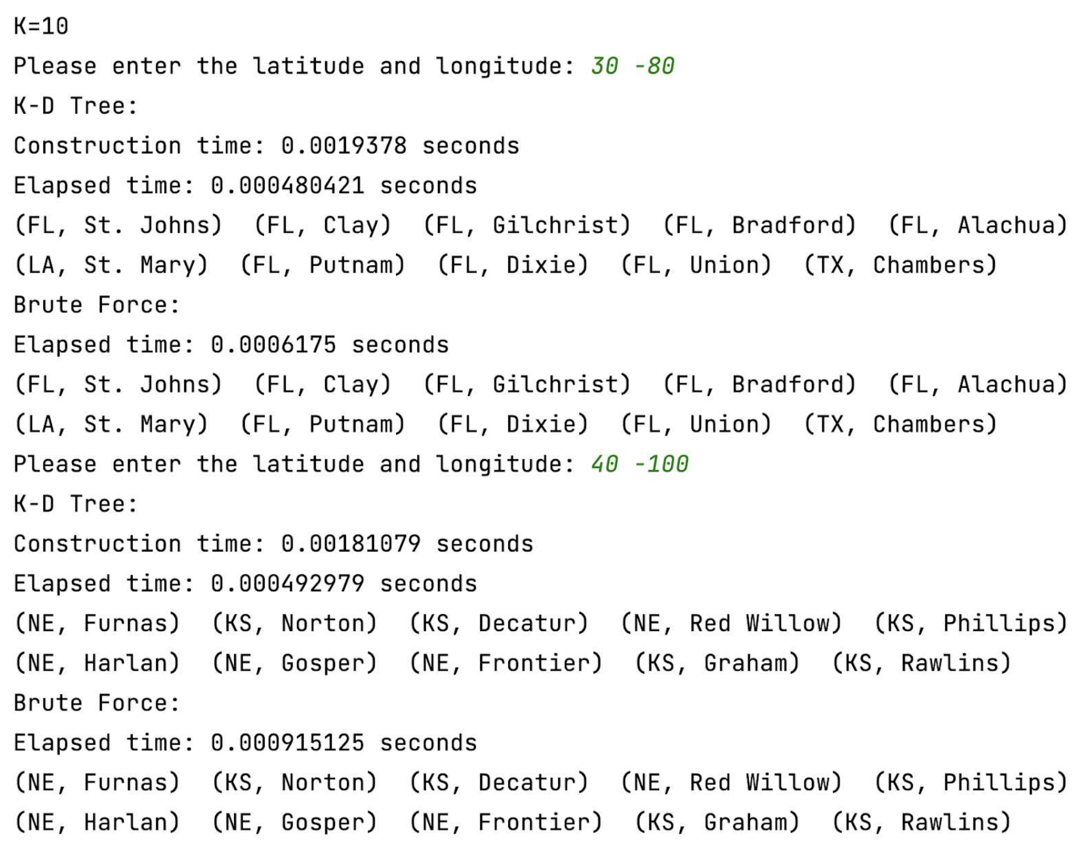
```
K=1
Please enter the latitude and longitude: 30 -80
K-D Tree:
Construction time: 0.00180316 seconds
Elapsed time: 0.000321912 seconds
(FL, St. Johns)
Brute Force:
Elapsed time: 0.000540875 seconds
(FL, St. Johns)
Please enter the latitude and longitude: 40 -100
K-D Tree:
Construction time: 0.00183065 seconds
Elapsed time: 0.000322592 seconds
(NE, Furnas)
Brute Force:
Elapsed time: 0.0005385 seconds
(NE, Furnas)
```

When K = 5:



```
K=5
Please enter the latitude and longitude: 30 -80
K-D Tree:
Construction time: 0.00173182 seconds
Elapsed time: 0.000370346 seconds
(FL, St. Johns)  (FL, Clay)  (FL, Gilchrist)  (FL, Bradford)  (FL, Alachua)
Brute Force:
Elapsed time: 0.000587958 seconds
(FL, St. Johns)  (FL, Clay)  (FL, Gilchrist)  (FL, Bradford)  (FL, Alachua)
Please enter the latitude and longitude: 40 -100
K-D Tree:
Construction time: 0.00180674 seconds
Elapsed time: 0.000392388 seconds
(NE, Furnas)  (KS, Norton)  (KS, Decatur)  (NE, Red Willow)  (KS, Phillips)
Brute Force:
Elapsed time: 0.000713709 seconds
(NE, Furnas)  (KS, Norton)  (KS, Decatur)  (NE, Red Willow)  (KS, Phillips)
```

When K = 10:



```
K=10
Please enter the latitude and longitude: 30 -80
K-D Tree:
Construction time: 0.0019378 seconds
Elapsed time: 0.000480421 seconds
(FL, St. Johns)  (FL, Clay)  (FL, Gilchrist)  (FL, Bradford)  (FL, Alachua)
(LA, St. Mary)  (FL, Putnam)  (FL, Dixie)  (FL, Union)  (TX, Chambers)
Brute Force:
Elapsed time: 0.0006175 seconds
(FL, St. Johns)  (FL, Clay)  (FL, Gilchrist)  (FL, Bradford)  (FL, Alachua)
(LA, St. Mary)  (FL, Putnam)  (FL, Dixie)  (FL, Union)  (TX, Chambers)
Please enter the latitude and longitude: 40 -100
K-D Tree:
Construction time: 0.00181079 seconds
Elapsed time: 0.000492979 seconds
(NE, Furnas)  (KS, Norton)  (KS, Decatur)  (NE, Red Willow)  (KS, Phillips)
(NE, Harlan)  (NE, Gosper)  (NE, Frontier)  (KS, Graham)  (KS, Rawlins)
Brute Force:
Elapsed time: 0.000915125 seconds
(NE, Furnas)  (KS, Norton)  (KS, Decatur)  (NE, Red Willow)  (KS, Phillips)
(NE, Harlan)  (NE, Gosper)  (NE, Frontier)  (KS, Graham)  (KS, Rawlins)
```

From the above result figures, we can find that normally the K-D tree implementation spends less time to find the top K nearest neighbor cities. However, it will take lots of time to construct a K-D tree, but we think such a time-consuming process is meaningful since it’s just like building a dataset. As long as an original K-D tree is constructed, each time we add or delete one element is simple and efficient, leaving us away with doing the repetitive work before each time of search.

5 Conclusions

KD-Tree is generally more efficient for large datasets, while linear scanning is simpler to realize but less efficient. In this project, recursive median finding and space partitioning are used to construct the KD-Tree. The linear search algorithm uses a priority queue to efficiently find the k-nearest neighbors. Significantly, KD-Trees come with the complexity of implementation while they offer greater efficiency for larger datasets. Therefore, linear scanning can have a better performance for datasets with low dimensionality.

6 Reference

[1]Introduction to algorithms- Third edition. Thomas H. Cormen, The MIT Press, 2009

[2]https://github.com/brower/EC504_2023F/blob/main/Project_Instruction_and_Some_Topics/Past_SuggestedProjects/ProjectProposal3-NearestProvinceFinder2022.pdf
[3]https://www.baeldung.com/cs/k-d-trees
[4]https://blog.csdn.net/qq_50332374/article/details/123242226?ops_request_misc=&request_id=&biz_id=102&utm_term=kd%E6%A0%91%E6%97%B6%E9%97%B4%E5%A4%8D%E6%9D%82%E5%BA%A6&utm_medium=distribute.pc_search_result.none-task-blog-2~all~sobaiduweb~default-0-123242226.142