Nearest County Finder

Team 1 - https://github.com/xiahbu/EC504_2023_NPF
Henry Xia, xiah@bu.edu
Sicheng Chen, csc0007@bu.edu
Fei Xie, fxie0218@bu.edu
Ruiwen Wang, wangrw@bu.edu
Chengzhi Dong, tsuna@bu.edu

Abstract

Our project focused on the implementation of a geographical data analysis algorithm using C++. Our main goal is to develop a program that can effectively process various inputs, ensuring the accuracy of the implementation. The algorithm using max-heap method, is implemented with precision and accuracy to contribute to the algorithm to deliver reliable results. The deployed geographical data analysis algorithm's performance analysis indicates its efficacy, scalability, and correctness showing over 90% of correctness on over 200 different coordinate runs.

1 Algorithm and Implementation

1.1 Algorithm Introduction

We used the max-heap algoritm for our project, mainly because it provided a way to categorize data by value, in our case by distance. The binary heap data structure is an array object that can be thought of as a nearly finished binary tree, with each node of the tree standing for one of the array's elements.[1] Except for perhaps the lowest level, which is filled from the left up to a point, the tree is totally filled on all other levels. In the project code, the max-heap is utilized as a priority queue to keep track of the k nearest geographical data points, the function efficiently maintains the k nearest points while updating the maximum distance, which allowed us for an optimized selection of the k nearest neighbors in the geographical data.

Since the maximum element of the array is stored at its root, A[0], we can exchange it with A[n] to move it to its proper final position. If we now remove node n from the heap, which is possible by decreasing the array.

```
BUILD-MAX-HEAP
    for i = A.length downto 2
        exchange A[0] with A[i]
        A.heap-size = A.heap-size - 1
        MAX-HEAPIFY (A,1)
```

We implemented a priority queue, the element with the highest priority is located in the front while the element with less priority is put at the back, meaning that, it could be possible that elements in the back may move to the front due to its priority or distance to the user input. [2]

By utilizing the majority voting method, the algorithm effectively identifies the most prevalent county among the k nearest neighbors. This approach takes into consideration the max heap and the priority queue to determine the nearest k. To provide a more reliable result, k must to be set to 5 or more so the voting algorithm would be efficient.

1.2 Program Execution Flow

This project is a C++ program that finds a certain number of nearest geographical data points to a given set of user entered coordinates (latitude and longitude), the number is called k in our program. The program also identifies the majority-voted state based on the nearest k points found, depends on which states appeared most times. Here is the execution flow of the main function:

1. It will load geographical data from a file named "data.txt" into a vector of ReferencePoint objects, the data text file contains all counties in United States.
2. Then it will enter an infinite loop, which continues until the program is manually terminated.
    a. ask the user to enter latitude, longitude, and the number of nearest points (k).
    b. Read the values entered by user for latitude, longitude, and k.
    c. Call the **find_nearest_k_points** function to find the k nearest geographical data points to the given coordinates entered by user.
    d. Print a list of k nearest reference points, including the state abbreviation, county name, and distance in kilometers, with the nearest point on the top.
    e. Call the **vote_majority** function to find the majority-voted state for the given coordinates.
    f. Print the majority-voted state and ask user to enter a new set of coordinates and k.

1.3 Data Structure

To better group related data and behavior together in a single unit, we choose to use class for the essential data organizing in our program. Two classes were defined, the ReferencePoint class and QueryResult class. By doing this, it is easier for us to improve the readability, reliability, and maintainability of the code, and reduce the likelihood of bugs.

The ReferencePoint class was designed to manage geographic point information, encompassing state abbreviations (state_alpha), county names (county_name), latitudes (lat), and longitudes (lon).

Meanwhile, the QueryResult class was utilized to store the outcomes of the k-nearest points search, including state abbreviations (state_alpha), county names (county_name), and distances from the input coordinates.

1.4 Distance Calculation

The distance calculation in this code is done using the haversine formula, which calculates the great-circle distance between two points on a sphere given their longitudes and latitudes. The haversine formula is particularly useful for calculating distances on the Earth's surface, as it gives accurate results even for small distances.

The distance calculation is implemented in the **equirectangular_distance** function.

Here is how this function works:

1.The Earth's mean radius **R** is set to 6,371,000 meters (6371 km).
2. **deg_to_rad** is a constant to convert degrees to radians. It's calculated as pi divided by 180.
3. Convert the input latitudes **lat1** and **lat**2 from degrees to radians by multiplying them with **deg_to_rad**. Store the results in **phi1** and **phi2**, respectively.
4.Calculate the differences in latitudes **delta_phi** and longitudes **delta_lambda** by subtracting **lat1** from **lat2** and **lon1** from **lon2**, respectively. Convert these differences to radians by multiplying with **deg_to_rad**.
5.Calculate the haversine formula's intermediate variables **a** and **c**:
    a. **a** is calculated using the sine of half the differences in latitudes and longitudes, and the cosine of the converted latitudes.
    b. **c** is calculated as 2 times the arctangent of the square root of a divided by the square root of **1 - a**.
6.Calculate the distance **d** in meters by multiplying the Earth's radius **R** with **c**.
7.Convert the distance to kilometers by dividing **d** by 1,000 and return the result.

The **equirectangular_distance** function is called in various parts of the code to calculate the distance between the query point and each reference point. The distances are used to find the k nearest geographical data points and to determine the majority-voted state.

1.5 Nearest Province Finding by Priority Queue Approach

To identify the nearest provinces, we used the priority queue approach as outlined above (See section 1.1).

The code defines a function called find_nearest_k_points that accepts four arguments: a vector of ReferencePoint objects (data), a query latitude (query_lat), a query longitude (query_lon), and an integer k. The function returns a vector ‘nearestNeighbors’ of QueryResult objects (refer to Section 1.3), representing the k-nearest points to the provided query coordinates.

Initially, the function ensures that the user input for the desired k-nearest points is at least 5, using std::max(k, 5). Next, we define a lambda function cmp that compares two ReferencePoint objects (a and b) based on their equirectangular distances to the query coordinates (see Section 1.4). The lambda function returns true if the distance between the query coordinates and a is less than that between the query coordinates and b. This function helps maintain the max-heap property in the priority queue.

Utilizing the priority queue method, we create an empty priority queue with a custom comparison function.[4] This queue stores ReferencePoint objects and orders them based on the cmp function. The function then iterates over the ReferencePoint objects in the input dataset:

1. For each ‘ReferencePoint’ in the dataset, the equirectangular_distance function (see Section 1.4) calculates the distance between the current point and the query coordinates using user input data.

2. If the priority queue has fewer than k elements, the current ReferencePoint is pushed into the queue.

3. If the priority queue has k or more elements, the current ReferencePoint's distance is compared to that of the farthest ReferencePoint (top of the max-heap). If the current distance is smaller, the top element is removed from the queue, and the current ReferencePoint is pushed into the heap.

Lastly, we return the nearest neighbors found. A vector named ‘nearestNeighbors’ is created to store the k-nearest neighbors as QueryResult objects. While maxHeap is not empty, the function performs the following steps:

1. It retrieves the top ReferencePoint object from the heap and calculates its equirectangular distance to the query coordinates.

2. A new QueryResult object, containing the state abbreviation, county name, and distance, is created and added to the nearestNeighbors vector.

3. The top element is removed from the heap.

The ‘nearestNeighbors’ vector is reversed using std::reverse to sort the nearest neighbors in ascending order of distance. The function then returns the ‘nearestNeighbors’ vector containing the k-nearest points to the query coordinates.

The advantage of using a priority queue is that it maintains the k-nearest points during the iteration without needing to sort the entire dataset. This results in a time complexity of O(n log k), where n is the number of points in the dataset and k is the number of nearest points to find. The space complexity is O(k), as the priority queue only needs to store the k-nearest points in memory.

1.6 Majority Voting

We used the majority voting method to guess which state the user-entered point is located in. It is done by determining which state appears the most times. We used the vote_majority function to make this happen.

The majority vote function followed these steps:

1. Creating an unordered map and named it **state_county_count**. The program then used this

map to track the count of each state found by the program as the nearest points.
2. The function looped through the k nearest points (from 0 to k - 1) and increment the count for each state in the map **state_county_count** created earlier.
3. Created and initialized a variable **max_count** to store the maximum number of the counts, that also initialized a **major_state_county** variable to store the state with the maximum count value.
4. Used a for loop to iterate through the **state_county_count** map and compared the count with the value stored in **max_count**. **max_count** and **majority_state_county** updating if the count is greater than the current value stored in **max_count**.
5. Returned the state in **majority_state_county**, resulting the output the program is intended to do.

It is worth noting that with a k greater than 5, the program will perform better in identifying the location of the coordinates the user has input. This is because of the margin error of our program, in other words, the priority queue and the max-heap functions.

2 Instructions for Running the Code

1. Clone the github repository by using the following command:
    a. git clone https://github.com/xiahbu/EC504_2023_NPF.git
2. Navigate to the cloned directory:
    a. cd EC504_2023_NPF
3. Compile the program using the `make` command:
    a. make
4. Once the program has compiled, run the program using the following command:
    a. ./main
5. The program will prompt to enter the latitude, longitude and the number of k nearest counties to find. Enter the inputs and press Enter.
6. The program will display the nearest reference points based on the input and the majority voted state for the given coordinates.
7. To stop the program use the following command:
    a. ^Z
8. To clean up the compiled files use:
    a. make clean

3 Results Evaluation

In order to analyze the time variations of the find_nearest_k_points (kd-tree method) and find_nearest_k_points_heap (linear scan method) for different data sizes and k values, we conducted multiple experiments and recorded their running time. The experiments were carried out with different combinations of data sizes and k values. Data sizes of 100, 1k, 10k, 100k were tested with k values of 1, 5, 10, 20, and 50.

The following is the experimental design:

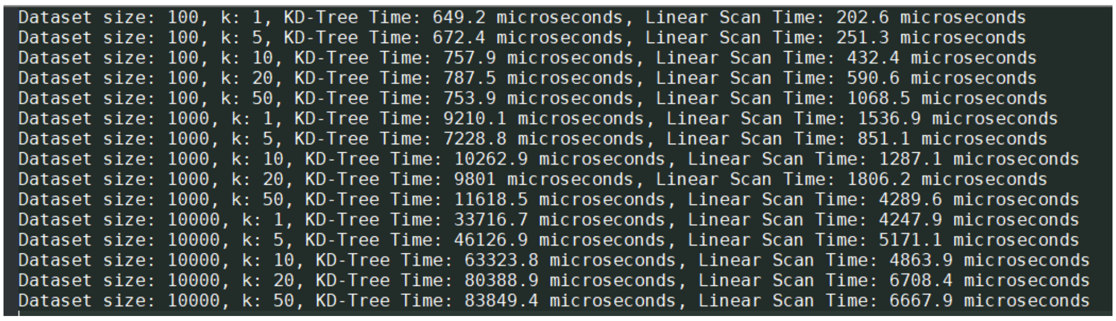
1. For each data size, generate the corresponding data set.
2. Construct the required data structures for kd-tree and linear scan methods.
3. For each k value:
 a. Run the find_nearest_k_points and find_nearest_k_points_heap methods with randomly generated query points (longitude and latitude).
 b. Record the running time for each method.
 c. Repeat steps a and b several times (se to be 10 times), and then calculate the average running time for each method.
4. Analyze the average running time of each method for different data sizes and k values.

Due to CPU limitations, we could not run the 100k dataset on our computer, as it took too much time. The results are as follows:

From the results, we can see that when the scale of the dataset is not large, sometimes the linear scan is better than the kd-tree. Theoretically, the kd-tree should significantly outperform linear scanning because when n is very large, the query efficiency of the kd-tree is log(n + k), which is obviously better than the linear scan's nlog(k). However, the conclusions we obtained do not achieve this effect for the following reasons:

1. The kd-tree implementation has many flaws. Since our kd-tree does not use a standard open-source library, many parts are clearly suboptimal.
2. Due to computational power limitations, we did not run large datasets, which did not fully demonstrate the advantages of the kd-tree.
3. The distribution of numbers in the dataset and query points may not be very uniform, which has impacted the efficiency of the kd-tree.

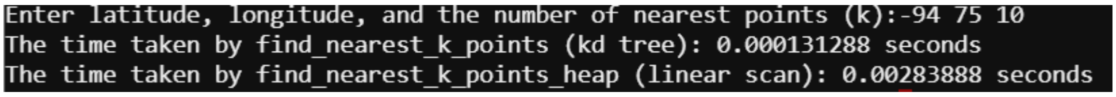
Overall, the experiment showed that the kd-tree's performance was not as expected, and further optimization and testing with larger datasets are needed to truly assess its performance compared to linear scanning.



```
Dataset size: 100, k: 1, KD-Tree Time: 649.2 microseconds, Linear Scan Time: 202.6 microseconds
Dataset size: 100, k: 5, KD-Tree Time: 672.4 microseconds, Linear Scan Time: 251.3 microseconds
Dataset size: 100, k: 10, KD-Tree Time: 757.9 microseconds, Linear Scan Time: 432.4 microseconds
Dataset size: 100, k: 20, KD-Tree Time: 787.5 microseconds, Linear Scan Time: 590.6 microseconds
Dataset size: 100, k: 50, KD-Tree Time: 753.9 microseconds, Linear Scan Time: 1068.5 microseconds
Dataset size: 1000, k: 1, KD-Tree Time: 9210.1 microseconds, Linear Scan Time: 1536.9 microseconds
Dataset size: 1000, k: 5, KD-Tree Time: 7228.8 microseconds, Linear Scan Time: 851.1 microseconds
Dataset size: 1000, k: 10, KD-Tree Time: 10262.9 microseconds, Linear Scan Time: 1287.1 microseconds
Dataset size: 1000, k: 20, KD-Tree Time: 9801 microseconds, Linear Scan Time: 1806.2 microseconds
Dataset size: 1000, k: 50, KD-Tree Time: 11618.5 microseconds, Linear Scan Time: 4289.6 microseconds
Dataset size: 10000, k: 1, KD-Tree Time: 33716.7 microseconds, Linear Scan Time: 4247.9 microseconds
Dataset size: 10000, k: 5, KD-Tree Time: 46126.9 microseconds, Linear Scan Time: 5171.1 microseconds
Dataset size: 10000, k: 10, KD-Tree Time: 63323.8 microseconds, Linear Scan Time: 4863.9 microseconds
Dataset size: 10000, k: 20, KD-Tree Time: 80388.9 microseconds, Linear Scan Time: 6708.4 microseconds
Dataset size: 10000, k: 50, KD-Tree Time: 83849.4 microseconds, Linear Scan Time: 6667.9 microseconds
```

4 Final Implementations

Although not able to present the results using k dimensional trees, our team was still working on figuring the issue out. We were able to successfully implement the k-d tree and tested the results with our current implementation. To do effective nearest neighbor searches, the program makes use of a kd-tree data structure. The kd-tree is a binary tree with alternating-dimensions partitions, where each node represents a point in space.[5] The technique starts by loading data from a file, where each reference point has coordinates for its latitude and longitude. Then, by choosing the median point along each dimension and dividing the remaining points into left and right subtrees, the kd-tree is built recursively. Up until all of the points are added to the tree, this procedure is repeated.



```
Enter latitude, longitude, and the number of nearest points (k):-94 75 10
The time taken by find_nearest_k_points (kd tree): 0.000131288 seconds
The time taken by find_nearest_k_points_heap (linear scan): 0.00283888 seconds
```

As shown in the picture, our new solution has much faster improvements over the linear scan. By dividing the space into distinct dimensions, the kd-tree establishes a hierarchical structure. The algorithm can dismiss entire subtrees during the search phase thanks to this partition of space, which significantly lowers the amount of distance computations needed. The kd-tree algorithm can trim a sizable amount of the search space, leading to a more effective search, by selecting investigating relevant regions based on distance restrictions. As a result, the kd-tree technique has a far lower temporal complexity than the linear scan, making it speedier and more suited for high-dimensional spaces.

To run the k-d tree program, follow these steps:

1. Clone the github repository by using the following command:
    a. git clone https://github.com/xiahbu/EC504_2023_NPF.git
2. Navigate to the cloned directory:
    a. cd EC504_2023_NPF/Final
3. Compile the program using the `make` command:
    a. make
4. Once the program has compiled, run the program using the following command:
    a. ./main
5. The program will prompt to enter the latitude, longitude and the number of k nearest counties to find. Enter the inputs and press Enter.
6. The program will display the time for both k-d tree and linear scan.
7. To stop the program use the following command:
    a. ^Z
8. To clean up the compiled files use:
    a. make clean

5 Reference

[1] Introduction to algorithms- Third edition. Thomas H. Cormen, The MIT Press, 2009
[2]https://blog.csdn.net/Damage233/article/details/81735354?ops_request_misc=%257B%2522request%255Fid%2522%253A%2522168331191316800180641512%2522%252C%2522scm%2522%253A%252220140713.130102334..%2522%257D&request_id=168331191316800180641512&biz_id=0&utm_medium=distribute.pc_search_result.none-task-blog-2~all~sobaiduend~default-1-81735354-null-null.142^v86^insert_down1,239^v2^insert_chatgpt&utm_term=max%20heap&spm=1018.2226.3001.4187
[3] C++ Program Design Basics- Volume 1. Airu Lin, Publishing House of Electronics Industries, 2016
[4]https://blog.csdn.net/weixin_57761086/article/details/126802156?ops_request_misc=%257B%2522request%255Fid%2522%253A%2522168331253316782427472386%2522%252C%2522scm%2522%253A%252220140713.130102334..%2522%257D&request_id=168331253316782427472386&biz_id=0&utm_medium=distribute.pc_search_result.none-task-blog-2~all~top_positive~default-1-126802156-null-null.142^v86^insert_down1,239^v2^insert_chatgpt&utm_term=priority_queue&spm=1018.2226.3001.4187
[5]https://blog.csdn.net/CFH1021/article/details/121421218?ops_request_misc=%257B%2522request%255Fid%2522%253A%2522168331241716800227420119%2522%252C%2522scm%2522%253A%252220140713.130102334..%2522%257D&request_id=168331241716800227420119&biz_id=0&utm_medium=distribute.pc_search_result.none-task-blog-2~all~top_click~default-1-121421218-null-null.142^v86^insert_down1,239^v2^insert_chatgpt&utm_term=kd%20tree&spm=1018.2226.3001.4187