# The Nearest State/County Finder

EC504 Final Project Report

## Team Information

| Member | BU email | BU ID | SCC username |
|---|---|---|---|
| Qilong Wang | qilongw@bu.edu | U47157899 | qilongw |
| Dian Jin | jin725@bu.edu | U14412233 | jin725 |
| Yachen Wang | wyachen@bu.edu | U16533002 | wyachen |
| Yu Guo | yyguo@bu.edu | U4307906 | yyguo |
| Tianze Li | tianzeli@bu.edu | U26235868 | tianzeli |

## Abstract

In this project, we have built a program which allow users to enter a latitude and longitude of a location inside the United States and then find several closest counties to that point. The geographical locations of the counties in the US were extracted from the official website of US Board on Geographic Names. Our team proposed a method to solve such a problem which used KD tree data structure. In this report, we will present the theory of our algorithm, the C++ code to achieve this algorithm, instructions to run this code, and also the sample results. This report gives a summary of the procedures followed to accomplish this project.

## Approach: KDTree searching algorithm

### Introduction:

KD Tree is actually a k-dimension tree whose node/point has k dimensions. Every non-leaf node could split the whole zone into two areas, which means in the building and searching aspects, it is extremely similar to the binary search tree, but the only difference is that this tree has 2 *dimensions.* Our project is to search and output k-nearest cities of a randomly selected location within the United States, in this scenario, k is equal to 2, which means the 2-d tree has latitude and longitude serving as its two-dimension coordinates.

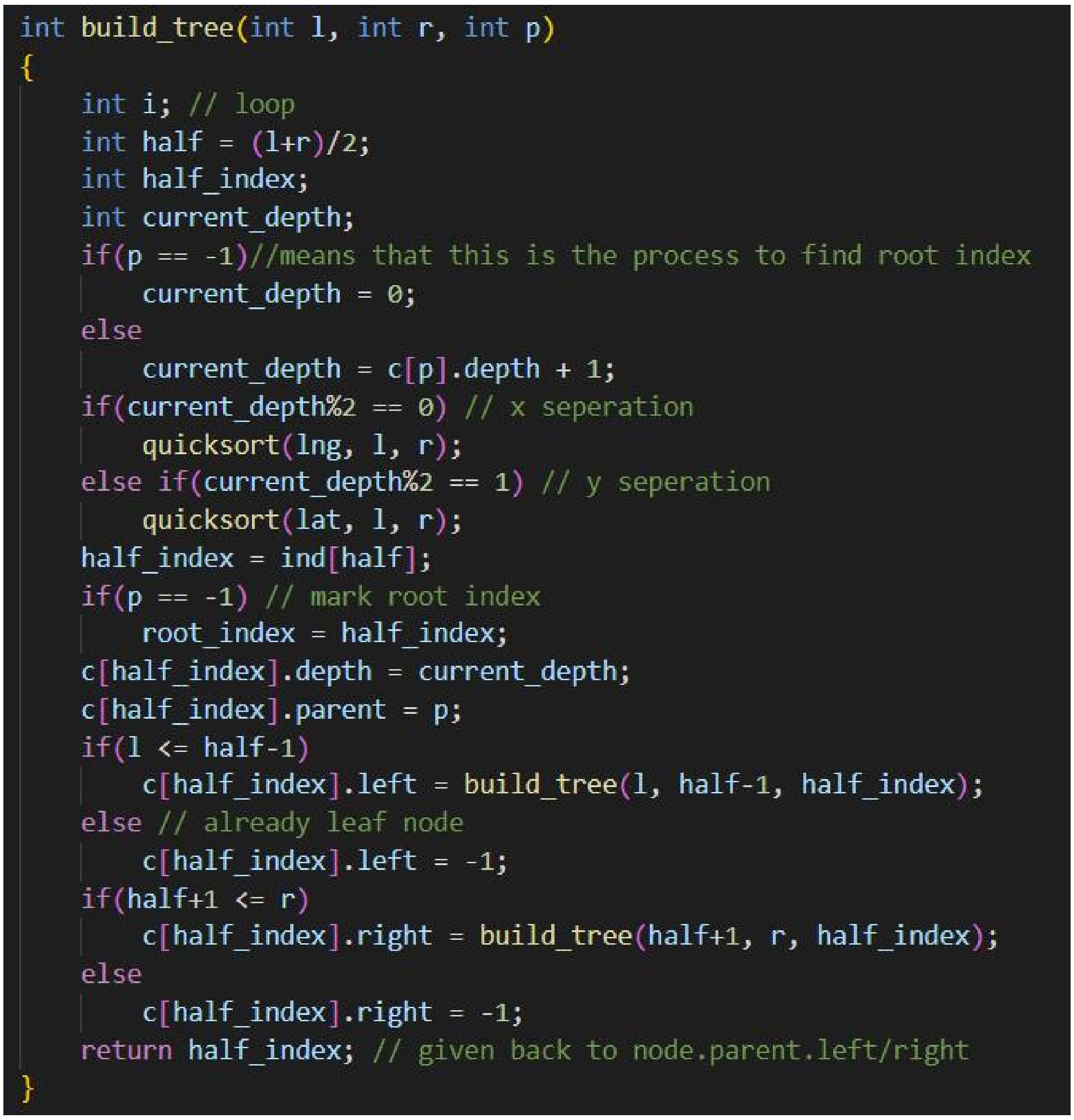
Thus when building a k-d tree and searching for nearest cities, the two coordinates (latitude and longitude) have to be considered and utilized in turn.

## Algorithm:

As described above, it is necessary for us to firstly build the tree and then traverse the tree conditionally(selectively).
To build a kd-tree( in this case, 2d tree), the city dataset needs to be put into an ascending order in both latitude( x dimension) and longitude (y dimension). This is to provide convenience for splitting the area repeatedly.
The reason for selection: it is apparent that when the distance between the input location and the parent node is already larger than the current minimum distance, it is not necessary to traverse the other subtree of the parent node.

## Steps:

1. Building
   Firstly, choose the middle node on the x axis and draw a vertical line, and secondly, choose the middle node on the y axis and draw a horizontal line, finally, repeat the steps above until all the nodes have drawn lines. We keep inserting the nodes to the tree so that we will have our KD Tree built.



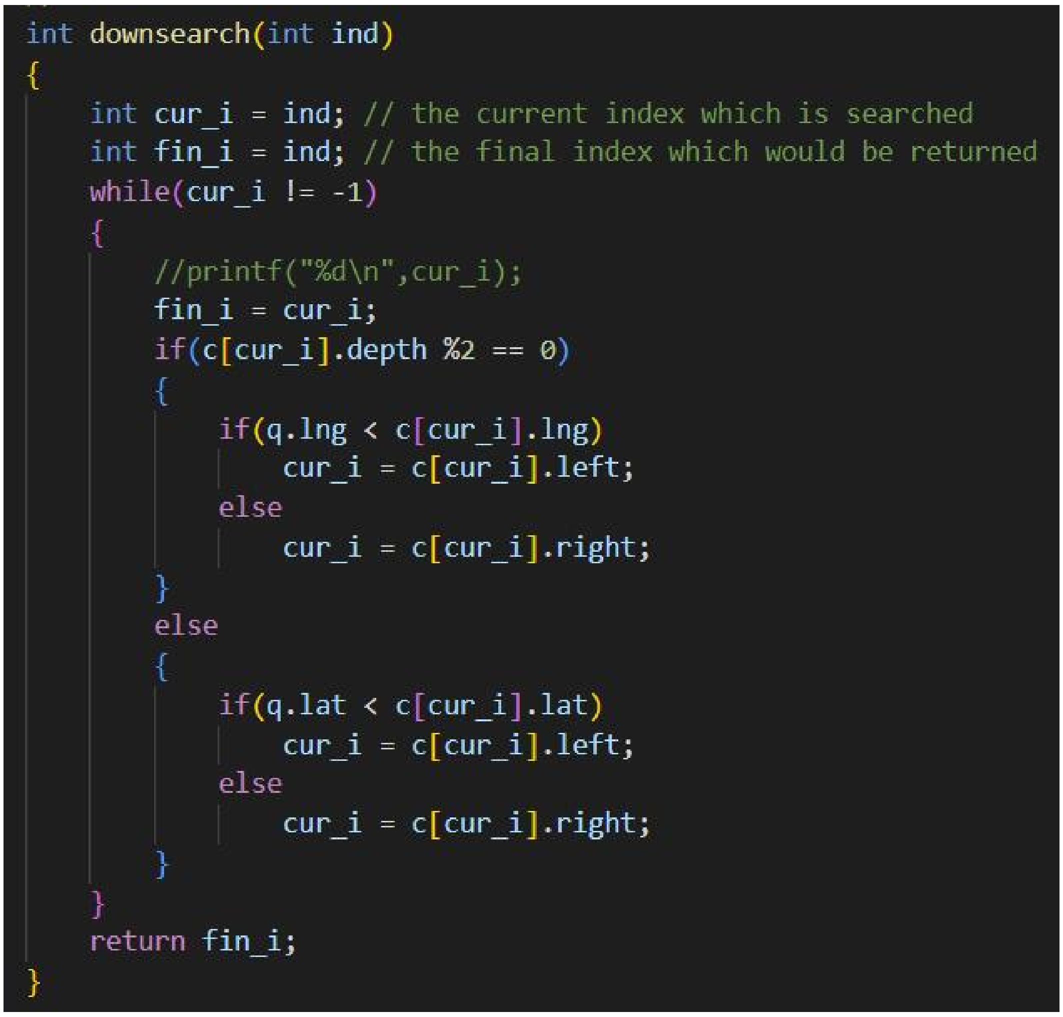
```
int build_tree(int l, int r, int p)
{
    int i; // loop
    int half = (l+r)/2;
    int half_index;
    int current_depth;
    if(p == -1)//means that this is the process to find root index
        current_depth = 0;
    else
        current_depth = c[p].depth + 1;
    if(current_depth%2 == 0) // x seperation
        quicksort(lng, l, r);
    else if(current_depth%2 == 1) // y seperation
        quicksort(lat, l, r);
    half_index = ind[half];
    if(p == -1) // mark root index
        root_index = half_index;
    c[half_index].depth = current_depth;
    c[half_index].parent = p;
    if(l <= half-1)
        c[half_index].left = build_tree(l, half-1, half_index);
    else // already leaf node
        c[half_index].left = -1;
    if(half+1 <= r)
        c[half_index].right = build_tree(half+1, r, half_index);
    else
        c[half_index].right = -1;
    return half_index; // given back to node.parent.left/right
}
```

Figure 1. Code for building the KD Tree

2. Searching

   The K-D (K=2 in out project) Tree just built could be utilized to search k nearest cities of a location within the United States. To begin with, complete a downsearch to the leaf of this tree. Downsearch is similar to binary  search, but in down-search method, first the coordinate with larger variance is selected, in other words, we start the comparison between the current selected location latitude and the root’s latitude. Then after deciding whether  left or right subtree to enter, it is time to alter the coordinate axis and compare the longitude between target location and current node. Steps above repeat until target location reaches out to leaf node.

Then it’s time to do the up-search. First calculate the distance between target location and leaf node. Save it as the current nearest city. Also calculate the distance between parent node and target location. Compare these distances: If the latter is smaller than the former, it is possible the other subtree also has  smaller distances than the current nearest distance. So just enter and search down. If not, just ignore the other subtree and step to the upper level. Repeat until it goes back to the root node. When k is larger or equals to 2, the k nearest cities are saved and finally output in the ascending order.



```
int downsearch(int ind)
{
    int cur_i = ind; // the current index which is searched
    int fin_i = ind; // the final index which would be returned
    while(cur_i != -1)
    {
        //printf("%d\n",cur_i);
        fin_i = cur_i;
        if(c[cur_i].depth %2 == 0)
        {
            if(q.lng < c[cur_i].lng)
                cur_i = c[cur_i].left;
            else
                cur_i = c[cur_i].right;
        }
        else
        {
            if(q.lat < c[cur_i].lat)
                cur_i = c[cur_i].left;
            else
                cur_i = c[cur_i].right;
        }
    }
    return fin_i;
}
```

Figure 2. Code for the downsearch

Here we provide with our team GitHub link:
https://github.com/wyachen/EC504_Nearest-Country-Finder

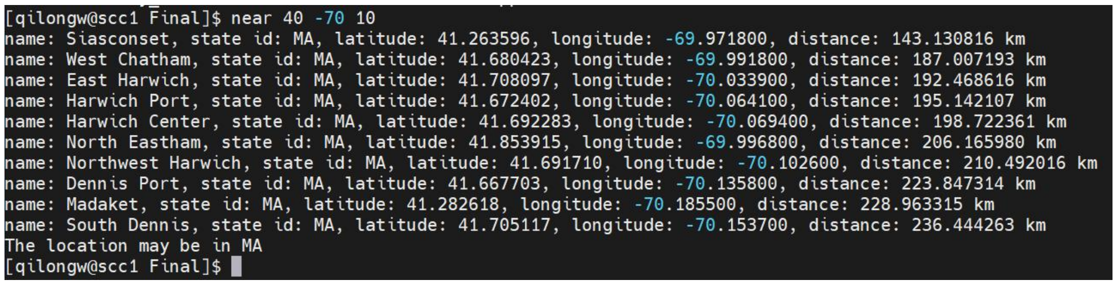
## Instructions for Running the Code

The code can be running on SCC by the Linux terminal. Aftering compile the file by the Makefile, use the command “./near [latitude] [longitude] [number of outputs]”, and the number of outputs should be in the range of 1 to 10. The following command gives a sample:

```
./near 40 -70 10
```

For the latitude, a positive value means the northern latitude, and negative value means southern latitude. For the longitude, a positive scale means the east longitude, and negative scale represents the west longitude. In fact, you can input in any latitude and longtitude, just make sure that location is inside the United States.

## Sample Results



```
[qilongw@scc1 Final]$ near 40 -70 10
name: Siasconset, state id: MA, latitude: 41.263596, longitude: -69.971800, distance: 143.130816 km
name: West Chatham, state id: MA, latitude: 41.680423, longitude: -69.991800, distance: 187.007193 km
name: East Harwich, state id: MA, latitude: 41.708097, longitude: -70.033900, distance: 192.468616 km
name: Harwich Port, state id: MA, latitude: 41.672402, longitude: -70.064100, distance: 195.142107 km
name: Harwich Center, state id: MA, latitude: 41.692283, longitude: -70.069400, distance: 198.722361 km
name: North Eastham, state id: MA, latitude: 41.853915, longitude: -69.996800, distance: 206.165980 km
name: Northwest Harwich, state id: MA, latitude: 41.691710, longitude: -70.102600, distance: 210.492016 km
name: Dennis Port, state id: MA, latitude: 41.667703, longitude: -70.135800, distance: 223.847314 km
name: Madaket, state id: MA, latitude: 41.282618, longitude: -70.185500, distance: 228.963315 km
name: South Dennis, state id: MA, latitude: 41.705117, longitude: -70.153700, distance: 236.444263 km
The location may be in MA
[qilongw@scc1 Final]$
```

Figure 3: Results of running the code

Figure 3 shows the output of running the code. In the command window,users can enter a latitude scale, a longitude scale, and the number of cities they want to search. The input number 10 means the user would like to find 10 nearest counties around the assigned location (40°N 70°W in this example). For each of the output counties, we have the county’s name, the state it located in, the latitude and longitude scale, and their distance to the assigned point. The last line of the output shows which states does the user defined location is most likely to located in. This result is based on the state with the highest frequency of occurrence in the output counties.

$$x = (\lambda 2-\lambda 1) * \mathrm{Cos}((\phi 1+\phi 2)/2);$$
$$y = (\phi 2-\phi 1);$$
$$\mathrm{Distance} = \mathrm{Sqrt}(x*x + y*y) * R;$$

where $\phi$ is latitude, $\lambda$ is longitude, R is earth's radius

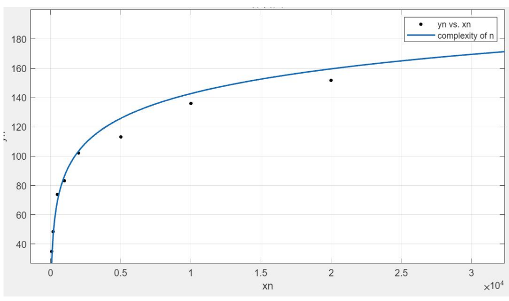
Figure 4: Formula for distance between two points on the Earth

Figure 4 shows the formula used to calculate the distance between each nearest county to the assigned location. Since the Earth is similar to a sphere instead of a flat surface, we need to find the length of the arc on Earth's surface. In the formula, we used the Earth's mean radius, which is 6371km.

## Time Complexity

Since the k-d tree already cuts space during construction, after a single query we approximately know where to look — we can just search the "surroundings" around that point. Therefore, practical implementations of k-d tree support querying for whole neighbors at one time and with complexity , which is much better for larger dimensions, which are very common in machine learning. Its complexity is O(k*log(n)), while k is the number of cities which users want to search, and n is the sample size of data. In our problem, k is between 1-10 and n is 30844.

We also test 451 points to validate its complexity. Figure 4 is the fit line of time complexity vs sample size (using function f(x)=a*log(x)+b), its R square is 0.961 which means the expression is believable. Figure 5 shows the time complexity vs k. Obviously the complexity is proportional to k, so this assumption about the time complexity is reasonable. This fit line has an R square of 0.9927.



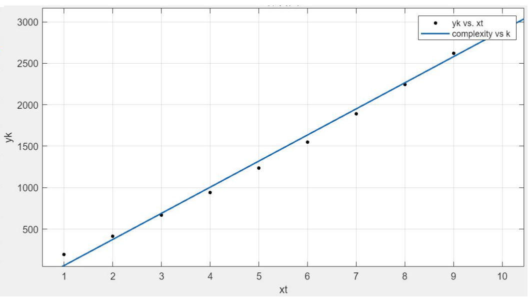


Fig. 4 complexity vs sample size                    Fig. 5 complexity vs k

We also plot the complexity in different k vs sample size. It is shown below.

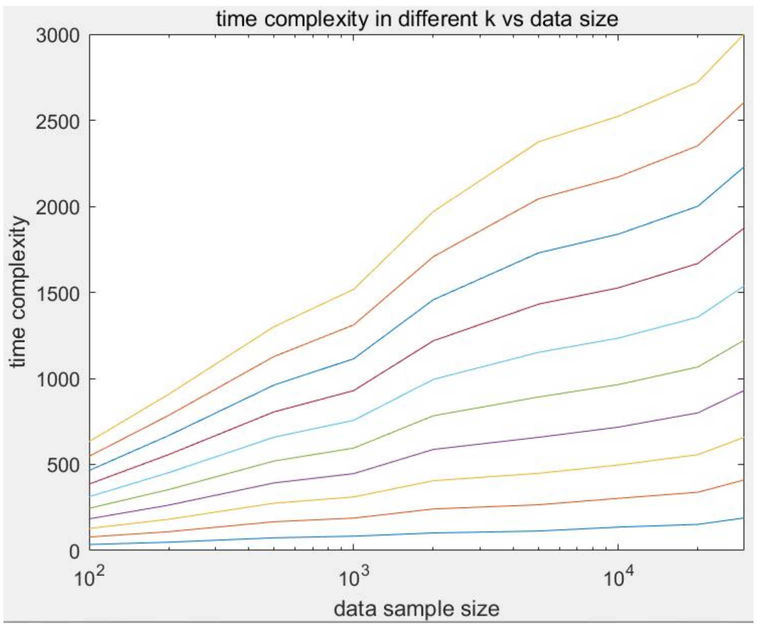


Fig. 6 time complexity in different k vs sample size

## Discussion

In this project, the two most critical variables are longitude and latitude. When sorting one of these variables, the other must follow the positional changes of the first variable. Initially, the project aimed to use structures for handling this data. However, as the structure pointers required constant updating when moved, this led to unsolvable issues.

After evaluating several alternative solutions, we ultimately decided to create separate arrays for city IDs, longitudes, and latitudes. This approach involved using city IDs to reference the database files containing city information, ensuring the integrity of the original database.

Search Tree - Upward Search:

To address the challenges of marking searched arrays and assigning initial distances, we set all nodes' initial distance values to infinity. This approach guarantees that the distance value will always exceed the Earth's maximum point-to-point distance, allowing for overwriting by smaller values.

Simultaneously, we determined whether a node had been searched; if a node's distance matched the current value, we marked it as unsearched and vice versa.

Bugs encountered and solutions:

During the search process, search nodes never entered sibling branches and consistently searched upward. Upon inspection, we identified that only the first node received an initial value during node initialization, which impeded subsequent comparisons. We resolved this issue by assigning initial distance values to nodes upon data import.

The algorithm invariably identified the first searched node as the nearest one. Further examination revealed that the distance function utilized an int type, while the database stored data as doubles, resulting in constant zero distances. Adjusting the function type to double resolved the problem.

Finding K Nearest Neighbors:

Initially, the algorithm could only locate the nearest node, requiring considerably less computational complexity than finding the n nearest nodes. We considered deleting found nodes and searching again before adding them to the results, but maintaining the kd-tree properties after node deletion proved challenging. Moreover, this method significantly increased time complexity and wasted resources processing previously processed information.

We also contemplated recording traversed nodes and selecting the k nodes with the smallest distances. However, this algorithm could not consistently yield the k nearest nodes in proximity to the target node.

During the search process, sibling branch nodes closer than some searched nodes might not be searched due to their longer distance from the nearest node.

Taking these factors into account, we decided first to save the k nearest nodes and their distances. We then updated the nearest node array based on the condition that a node's distance is smaller than the kth nearest node's shortest distance.

Ultimately, we output the k nearest nodes:
As the number of output nodes might not equal the search value k (when the user's required k is less than 5, the search tree still needs to output 5 nearest nodes for subsequent state determination), we attempted to use a traversed node array to output the shortest distance towns. However, some cities were recorded multiple times due to algorithmic issues, resulting in identical outputs. To resolve this, we added a judgment function to the algorithm, called only when searching a node. If a node had already been searched, the function would skip the node.

Debugs:
During the algorithm testing phase, we relied on console input for data entry, but our final version could not utilize this method. We employed argv and argc for data reading and provided feedback when input data failed to meet the required rules. Finally, by using <stdlib.h>, we addressed the issue of improper atoi function calls in some systems.

## References

[1] Z. Pan, "Instruction of KNN Algorithm and KD Tree," CSDN Blog, 2019. [Online]. Available: https://blog.csdn.net/zzpzm/article/details/88565645. [Accessed: 02-May-2023].
[2] L. Wang, "An Explanation of KD Tree," Zhihu, 2019. [Online]. Available: https://zhuanlan.zhihu.com/p/53826008. [Accessed: 02-May-2023].
[3] B. Himite, "Calculating the distance between two points on Earth," Medium, 29-May-2020. [Online]. Available: https://medium.com/swlh/calculating-the-distance-between-two-points-on-earth-bac5cd50c840. [Accessed: 02-May-2023].
[4] C. Veness, "Calculate distance and bearing between two Latitude/Longitude points using haversine formula in JavaScript," Movable Type Scripts, n.d. [Online]. Available: http://www.movable-type.co.uk/scripts/latlong.html. [Accessed: 02-May-2023].