TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN – ĐHQG TPHCM

UNIVERSITY OF SCIENCE – VNUHCM

CHALLENGE – DATA STRUCTURES AND ALGORITHMS

TOPIC: KD-TREE

Lớp:	23CLC10
Người hướng dẫn:	Lê Thanh Tùng
Thành viên thực hiện:	23127427 – Vũ Hoàng Minh
	23127451 – Nguyễn Đăng Phôn
	23127517 – Nguyễn Nam Việt

Thành phố Hồ Chí Minh, 2024

Contents

Table of Figures:

I. Member of group:

ID	Name	Tasks	Completion
23127427	Vũ Hoàng Minh	- 1, 4 Research. - Nearest Neighbor - User Interface: Load, Insert. - Reconstruct tree from json file	33.333%
23127451	Nguyễn Đăng Phôn	- 3, 6 Research. - Prepare the Dataset. - User Interface: Nearest. - Report.	33.333%
23127517	Nguyễn Nam Việt	- 2, 5 Research. - Range Search. - Note on Distance Calculation. - User Interface: Range Search. - Converts tree to json file.	33.333%

II. Research:

1. What is the process of constructing a KD-Tree for a given set of points in 2D space? Explain how the choice of axis and split points affects the structure of the tree.

- There are 4 steps to construct a KD-Tree for a given set of points in 2D space.
  - Choose the axis.
  - Sort points and select median.
  - Create child nodes.
  - Recursively apply the process

i. Choose the axis:

At the root level, choose the x-axis to split the points. At the next level, choose the y-axis, and alternate axes as you go deeper into the tree.

ii. Sort points and select median:

- Sort the points based on the chosen axis.
- Select the median point along this axis. The median point becomes the root node (or current node) of the subtree. This ensures that the tree is balanced, as the median splits the points into two roughly equal halves.

iii. Create child nodes:

- The left subtree consists of points to the left of the median (i.e., points with smaller values on the chosen axis).
- The right subtree consists of points to the right of the median (i.e., points with larger values on the chosen axis).

iv. Recursively apply the process:

- Repeats step 1-3 for each subtree, alternating the axis at each level.
- Continue until all points are added to the tree.

2. Time Complexity of Searching in KD-Tree and the Impact of Balancing the Tree

i. Time Complexity of Searching in KD-Tree:

- KD-Tree Structure: A KD-Tree is a binary tree used to partition points in k-dimensional space. Each node represents a point, and the splitting axis alternates at each level of the tree.
- Searching in KD-Tree:
  - At each node, the value of the search point is compared with the value of the node on the corresponding splitting axis.
  - Based on the comparison, the search moves to the left or right child node.
  - This process repeats until the search point is found or a leaf node is reached without finding the point.
- Time Complexity:
  - **Average Case:** If the KD-Tree is well-balanced, the average time complexity for searching is O (logn), where n is the number of points in the tree.
  - **Worst Case:** If the tree is unbalanced (i.e., each node has only one child), the search time complexity can be O(n).

ii. Impact of Tree Balancing:

- Balanced Tree: When the tree is balanced, the height of the tree is O(logn). This results in faster searches because the number of steps to reach a leaf node is minimized.
- Unbalanced Tree: If the tree is unbalanced, the height of the tree can be O(n), leading to significantly slower searches, equivalent to linear search through all points.

3. **Explain how insertions and deletions and handled in a KD-Tree. What strategies might be used to rebalance the tree if necessary.**

   i. Insertion:

- Firstly, checking for the tree is it empty or not. If it's empty, create a new tree with first node.
- Calculate current dimension of comparison. cd = depth % k (k is the number of dimensional point).
- Compare the new point with root on current dimension (cd) and decide the left or right subtree.
  - If point[cd] < root->point[cd]: Insert the new point to the left of the tree.
  - If point[cd] > root->point[cd]: Insert the new point to the right of the tree.

   ii. Deletion:

- Case 1:If the current node contains the point to be deleted and the node to be deleted is a leaf node, simply delete it.
- Case 2:If the current node contains the point to be deleted and the node to be deleted has a right subtree
  - Find minimum of current node's dimension in right subtree.
  - Replace the node with the minimum and recursively delete minimum in right subtree.
- Case 3: If the current node contains the point to be deleted and the node to be deleted has a left subtree.
  - Find minimum of current node's dimension in left subtree.
  - Replace the node with the minimum and recursively delete minimum in left subtree.

- Make new left subtree as right child of current node.

- Case 4: If the current node doesn't contain the point to be deleted
    - If the node to be deleted is smaller than the current node on current dimension, recur for the left subtree.
    - Else recur for the right subtree.

iii. Balance:

There are 2 ways we've found for rebalance a KD-Tree:

- Way 1:
    - Choose the first axis to split: We start with the x-axis.
    - Choose the median point by x-coordinate and split the points into 2 halves.
    - Repeat the process for each half with other axis til we get a balanced KD-Tree.
- Way 2:
    - Use an O(n) algorithm to find the arraysize/2-th largest element along a give axis and store it at the current node.
    - Iterate over all the elements in the vector and for each, compare them to the element was just found and put those smaller in newArray1, and those larger in newArray2.
    - Repeat the process for each half with other axis til we get a balanced KD-Tree.

4. **How does the dimensionality of the data affect the performance of KD-Trees? Dicuss the concept of the 'curse of dimensionality' in this context.**

i. Increased number of nodes:

- In higher dimensions, the number of nodes and possible split points increases exponentially. This results in a much larger tree structure, which can become computationally expensive to construct and traverse.

ii. Decreased efficiency in space partioning:

- In low-dimensional spaces, each split effectively partitions the space, reducing the search space significantly. However, as dimensions increase, each split becomes less effective at partitioning the space, as the volume of the search space grows exponentially.
- The partitions become less useful in narrowing down the search, leading to more nodes being visited during queries.

iii. Increased overlap and boudary complexity:

- Higher dimensions lead to more complex and overlapping regions between the partitions. This overlap increases the number of comparisons needed during search operations, reducing the efficiency of the KD-Tree.

iv. Distance metrics lose meaning:

- In high-dimensional spaces, the concept of distance becomes less meaningful. Points tend to be equidistant from each other, making it difficult to effectively partition the space.
- The ratio of the distance between the nearest and farthest points approaches 1 as the number of dimensions increases.

v. Exponential growth in volume:

- The volume of the space increases exponentially with the number of dimensions. Consequently, the number of data points needed to maintain the same density grows exponentially, leading to sparsity in the data.
- This sparsity makes it challenging to find meaningful nearest neighbors, as most points are far apart.

vi. Increased Search Complexity:

- As the number of dimensions increases, the number of nodes that need to be visited during a search operation increases exponentially.
- In very high-dimensional spaces, the efficiency of KD-Trees degrades, and the search time can approach the time complexity of a linear search, i.e., O(n)O(n)O(n).

⇨ In conclusion, while KD-Trees are effective for low-dimensional data, their performance deteriorates rapidly as the number of dimensions increases due to the curse of dimensionality. This necessitates the use of dimensionality reduction techniques (PCA, t-SNE, …) or alternative data structures for high-dimensional datasets (Ball trees, Cover trees, …).

5. Comparison of Implementing KD-Tree Using Arrays and Linked Lists:

i. KD-Tree Implementation Using Arrays:

- **Advantages:**
  - **Memory Usage:** More efficient as it does not require storing linkage information (pointers) between nodes. The array stores the data points directly without overhead.
  - **Access Time:** Faster access time because the position of each node in the array can be directly calculated using the array index.
- **Disadvantages:**
  - **Insertion and Deletion:** More challenging because adjusting the array to accommodate changes in the tree structure is required. Specifically, deleting a node may require re-adjusting the entire array.
  - **Rebalancing:** Rebalancing the tree is more complex with arrays as it involves shifting data.

ii. KD-Tree Implementation Using Linked Lists:

- **Advantages:**

- **Insertion and Deletion:** Easier because only the links between nodes need to be updated without the need to adjust the entire data structure. These operations are less costly as they involve pointer updates.
- **Rebalancing:** Easier as it involves updating pointers rather than moving data.

- **Disadvantages:**
  - **Memory Usage:** Less efficient due to the need to store linkage information (pointers) between nodes, resulting in overhead.
  - **Access Time:** Slower because accessing a node requires traversing pointers instead of using direct array indices.

6. **Evaluate the use of KD-Trees versus Binary Search Trees (BSTs) for multidimensional data handling. How do the structures compare when used for common operations like search, insert, and delete in a dataset with multiple attributes?**

- A KD-Trees is a special kind of BSTs for high dimensional data (more dimensions than 1.
- A BST excludes regions of number line from a search until the search point is found, a kd-tree works on regions of R^k. It means a BST stores one dimension of data for each node and a KD-Tree stores k dimensions of data for each node.
- Like a BST, which starts at the root and if the point we are searching for is less than the root we proceed down the left branch of the tree. If it is larger, we proceed down the right branch. A KD-Tree starts with a root node with a level of 0. At the i^{th} level, the nodes to the left of a parent have a lower value in the i^{th} dimension. Nodes to the right have a greater value in the i^{th} dimension. At the next level, the tree does the same for the next dimension.

III. Programing:

1. Prepare Process:

- Downloading the file .csv from the link https://gist.github.com/mchoi2000/e5e0486c74abdbb624db43d7f0783255 and read it.
- In this task, we read each line of the file and used stringstream to divide the data for each part of the node. The data is country, latitude, longittude, city and population.

2. Implement a KD-Tree:

i. Insertion:

- Insert a node:

```
Insert(root, data, depth)
{
    If (root is NULL){
            Create a new node containing data
            This new node becomes root of the tree
    }
    Calculate a formula: depth % k which k is set to 2 (latitude and longitude)
    If the answer is 0 root's key and data's key is set to latitude
    Else root's key and data's key is set to longittude.
    If root 's key is less than data's key
            Insert key to the root's RIGHT subtree and update plus 1 for depth
    Else if root's key is greater than data's key
            Insert key to the root's LEFT subtree and update plus 1 for depth
    Else do nothing
}
```

- Time complexity:

- Best case and Average case: O(log N).
- Worst case: O(N).

ii. Range Search:

- Function description:

  The range search functionality in the KD-Tree is a crucial tool for filtering data points that lie within a rectangular region defined by two corners: the bottomleft and top-right corners, specified in terms of latitude and longitude.

- rangeSearch Function:

  Initializes an empty vector to store results and calls the recursive search function.

- rangeSearchRec Function:

  The rangeSearchRec function performs a recursive search within the KD-Tree to identify points that lie within the rectangular region. For each node in the KD-Tree, it checks if the node's point is within a spherical region centered on the rectangle's center (using a check function is the point is in the rectangle). If true, it further checks if the point lies within the rectangular area. If the point is within both the spherical and rectangular regions, it is added to the result list.

- isInRangeRec Function:

  This function checks if a point lies within the rectangular area by comparing the point's latitude and longitude with the coordinates of the rectangle.

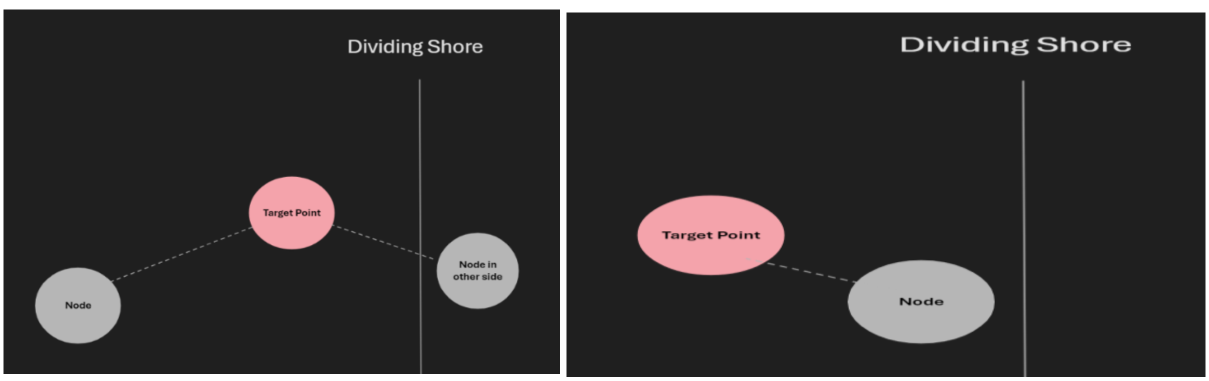
- Time complexity:

  Best case and Average case: O($\sqrt{N}$).

  Worst case: O(N).

iii. Nearest Search:

Nearest Neighbor Search function will traverse from root node to leaf, it is recursive function. When it reaches leaf, it will check the distance from the "target point" (user will give latitude-longitude) to this node by using "Haversine formula", and store this value in variable "best". During each checking, function will check two cases. In case one, the "Node" is the best distance in the left side; but in the other side, it maybe has "Node in other side" having the closer distance, so the function will check in the right side. In case two, the "Node" is the best distance, so the Nearest Neighbor Search function does not need to check in right side. Therefore, it is optimized. Then, it backtracks to the parent node until reaches root.



Case 1 *Case 2*

Figure 3 1: Nearest Neighbor Search in KD-Tree, guided by Haversine calculations

iv. Note on Distance Calculation:

Figure 3.1: Nearest Neighbor Search in KD-Tree, guided by Haversine calculations

The Haversine formula is used to calculate the distance between two points on the surface of the Earth, accounting for the Earth's spherical shape. This formula provides a more accurate distance measurement compared to simpler methods like Euclidean distance.

Formula:

$$d = 2 \cdot R \cdot \arcsin\left(\sqrt{\sin^2\left(\frac{\Delta\text{lat}}{2}\right) + \cos(\text{lat}_1)\cdot\cos(\text{lat}_2)\cdot\sin^2\left(\frac{\Delta\text{lng}}{2}\right)}\right) \quad (1)$$

Where:

- R is the Earth's radius (6371 km).
- **Δlat** and **Δlng** are the differences in latitude and longitude between the two points, converted from degrees to radians.
- **lat_1** and **lat_2** are the latitudes of the two points, also converted from degrees to radians.

3. User Interface:

- Start cmd where the code .cpp is stored and run the program:

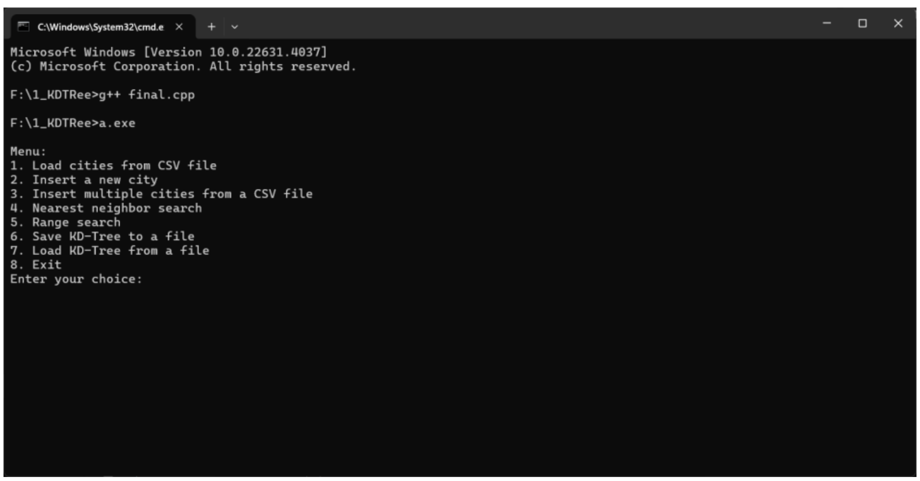


Figure 3 2: Starting cmd and running code in .cpp program

- Choose an option.
- A simple command-line interface that allows users to:
  - Load the list of cities from a CSV file.

    Input a .csv file when you choose option 1.

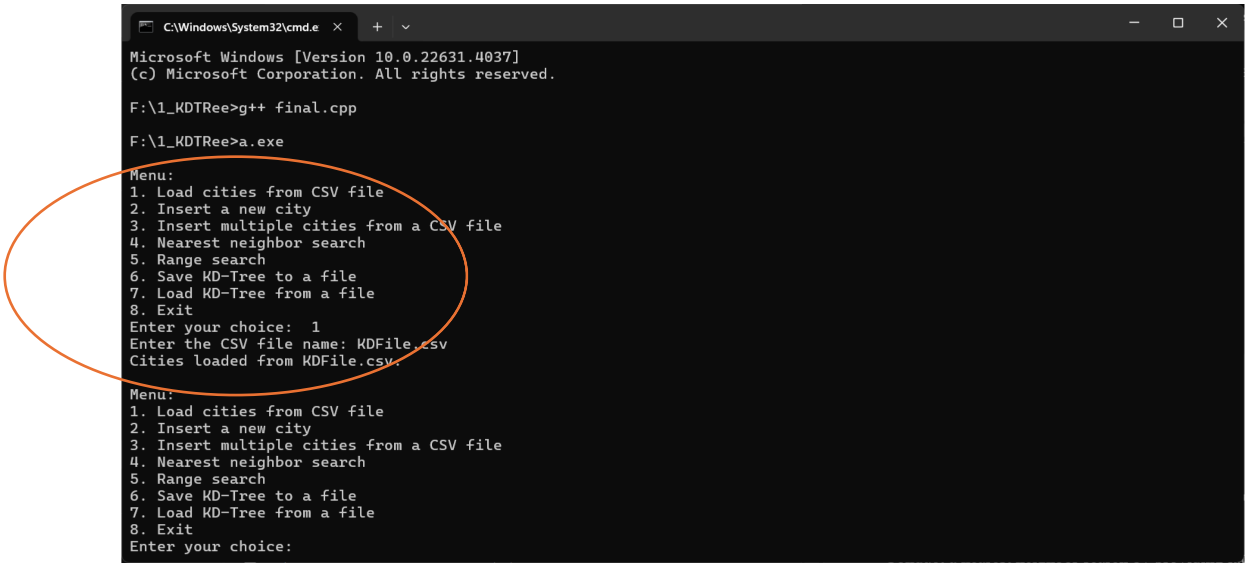


Figure 3.3: Input a .csv file when choosing option 1

- Insert a new city into the KD-Tree directly via the command line.

  Input the data of a city from the keyboard and the program will insert it into the KD-Tree.

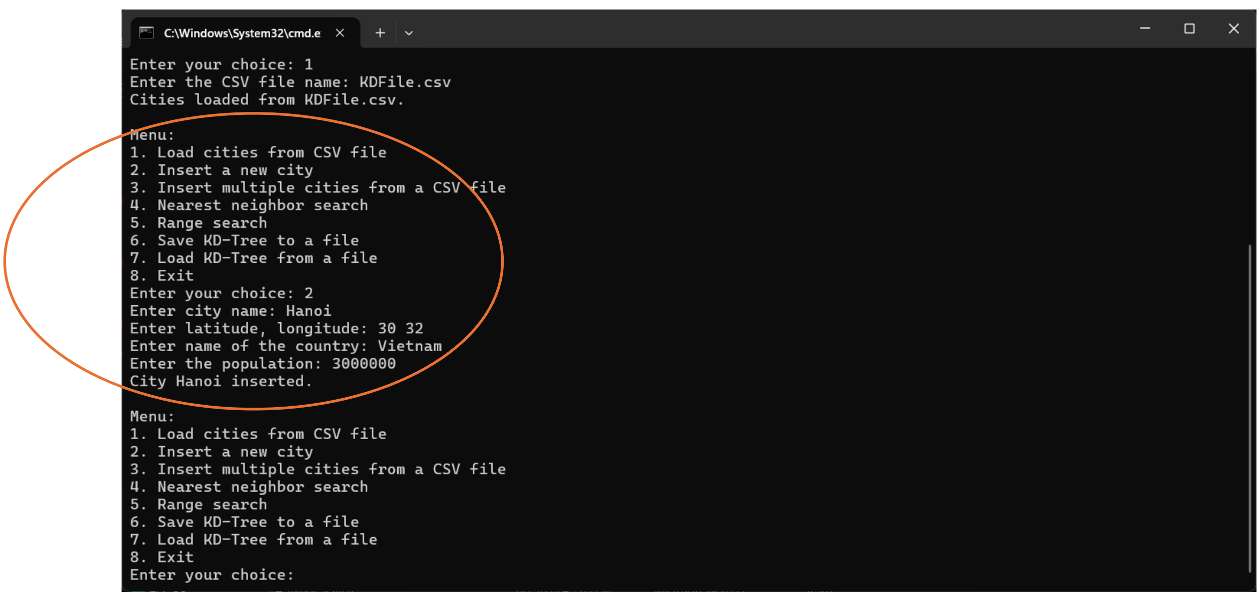


Figure 3.4: Inserting a new city into KD-Tree

- Insert multiple new cities into the KD-Tree from a specified CSV file path.
  - We have a .csv file.

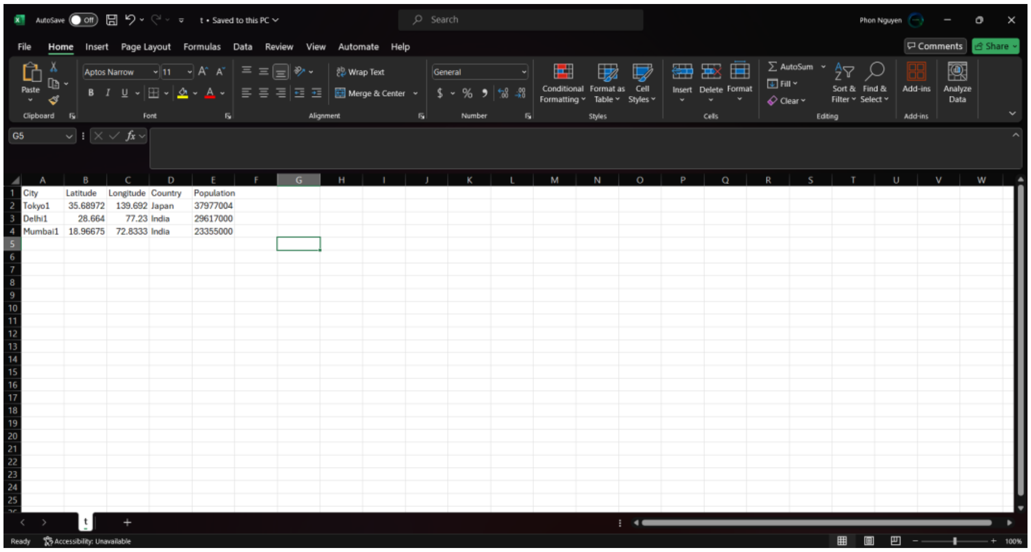


Figure 3.5: The .csv file

- Choose option 3 and input the name of this file.

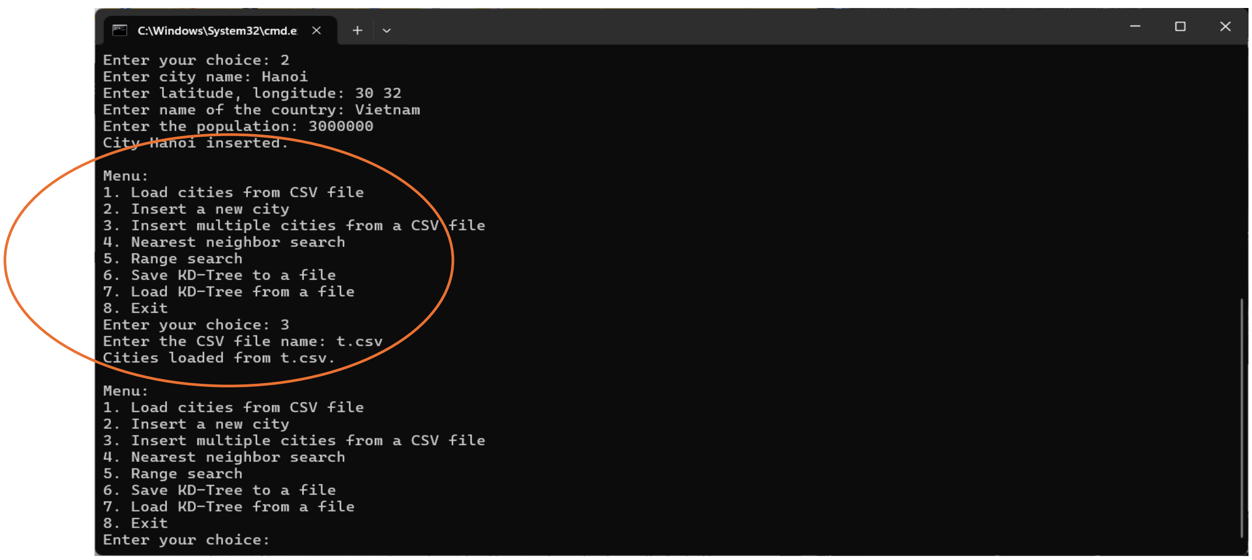


Figure 3.6: Choosing option 3 and inputing the name of csv file

- Conduct a nearest neighbor search by providing latitude and longitude. Choose option 4 and input the latitude and longitude to find nearest neighbor.

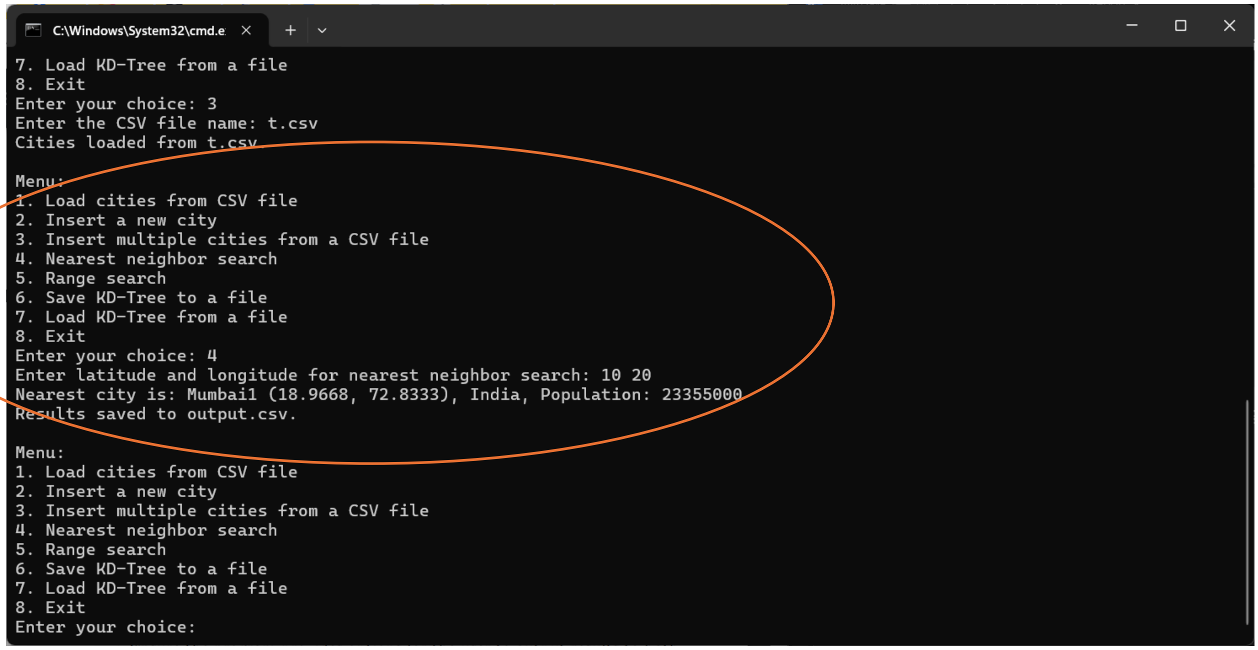


Figure 3.7: Choosing option 4 and inputing the latitude and longtitude to find the nearest neighbor

- Query all cities within a specified rengaular region defined by its geographical boundaries.

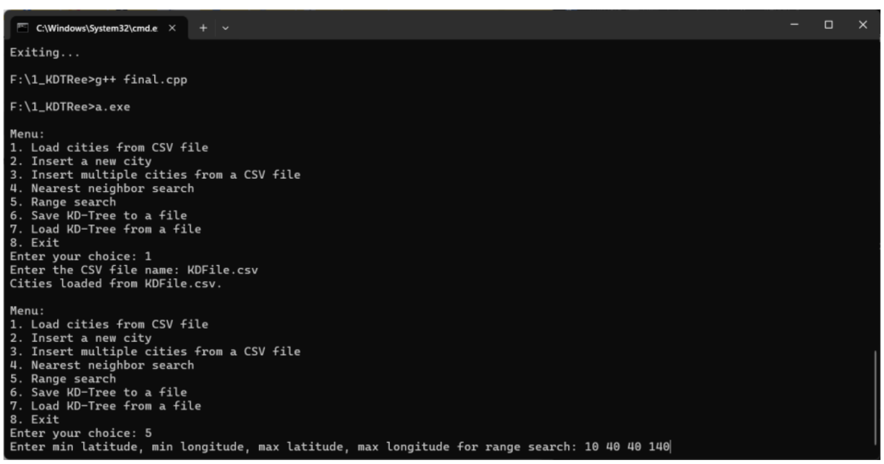


Figure 3.8: Choosing option 5 and inputing min latitude, longtitude and max latitude, longtitude (a coordinate of a rectangle)

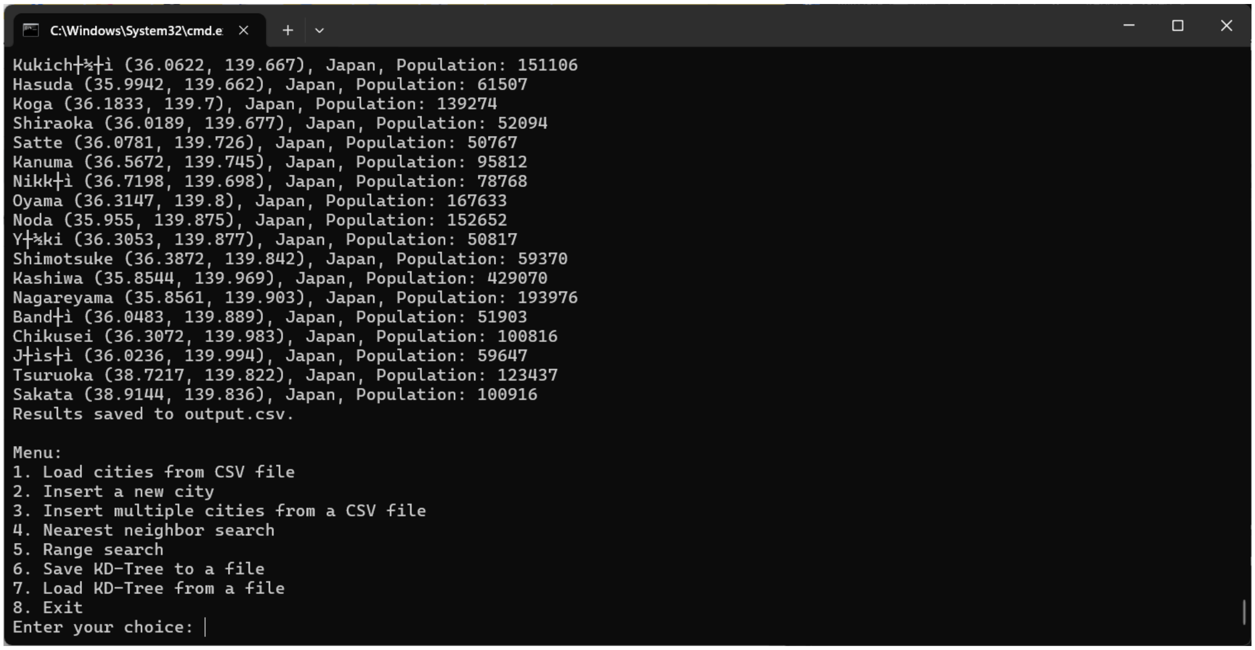


Figure 3.9: The output is all cities inside the rectangle

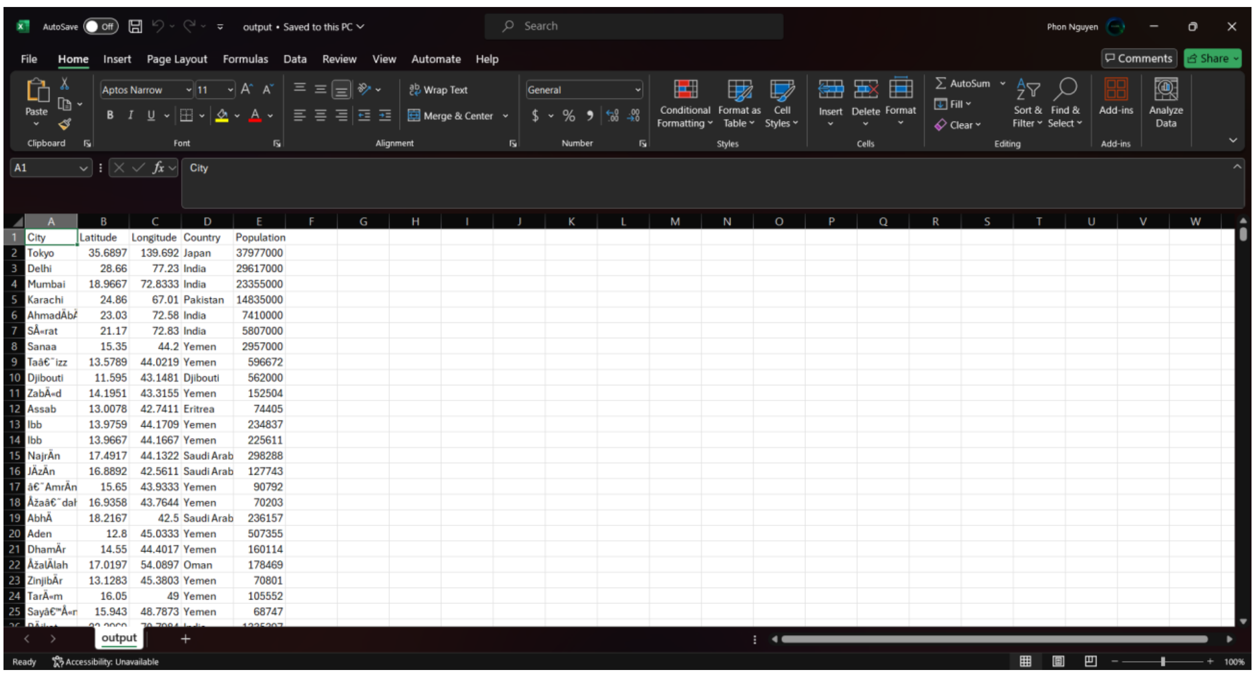


Figure 3.10: All cities inside the rectangle saved into output.csv

4. Extended Functionality:

i. Storing the KD-Tree Structure:

- In **User Interface,** choose option 6 and input .json file to save KD-Tree in json file.

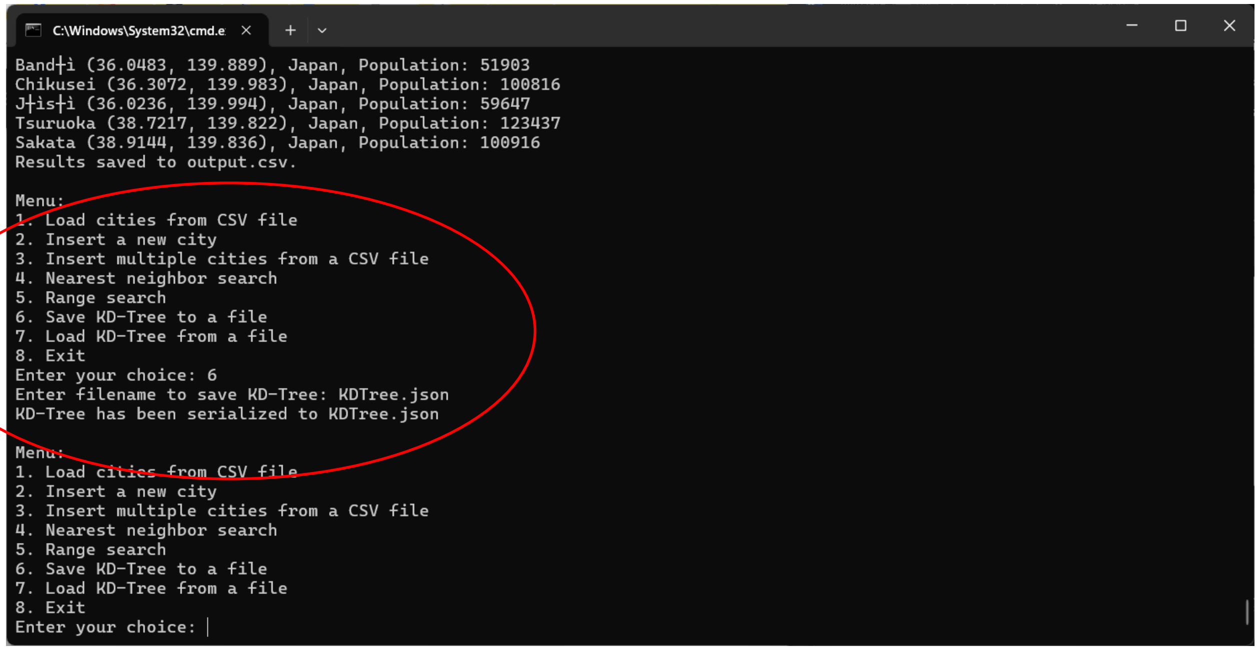


Figure 3.11: Choosing option 6 and inputing .json file to save KD-Tree in .json file

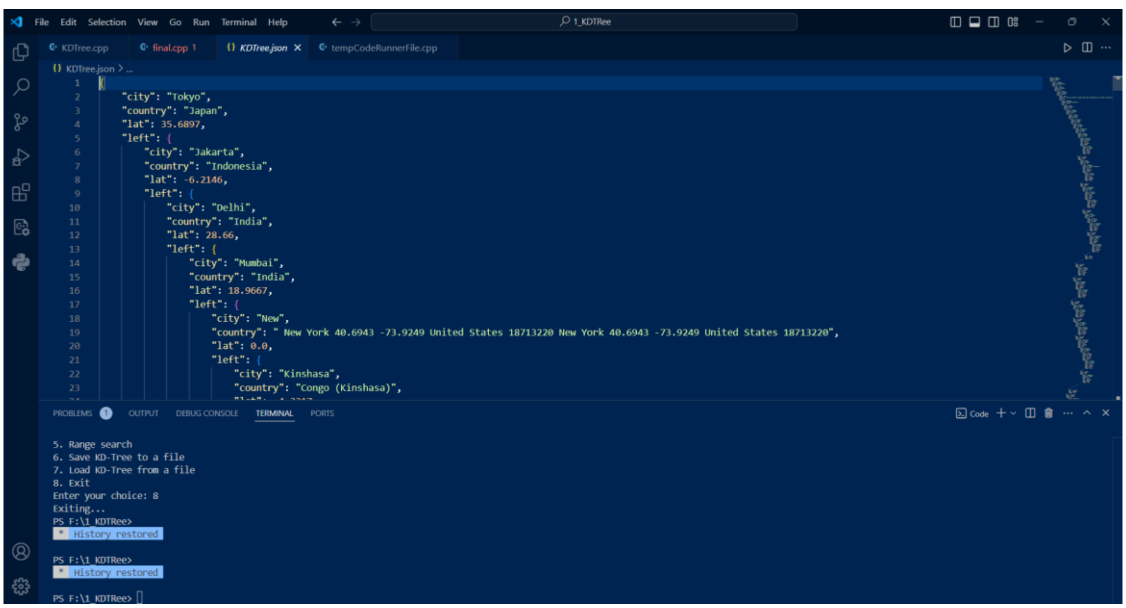


Figure 3.12: KD-Tree in .json file

ii. Reconstruct the KD-Tree from the stored files:

- In **User Interface,** choose option 7 and input .json file to reconstruct KD-Tree from a .json file.

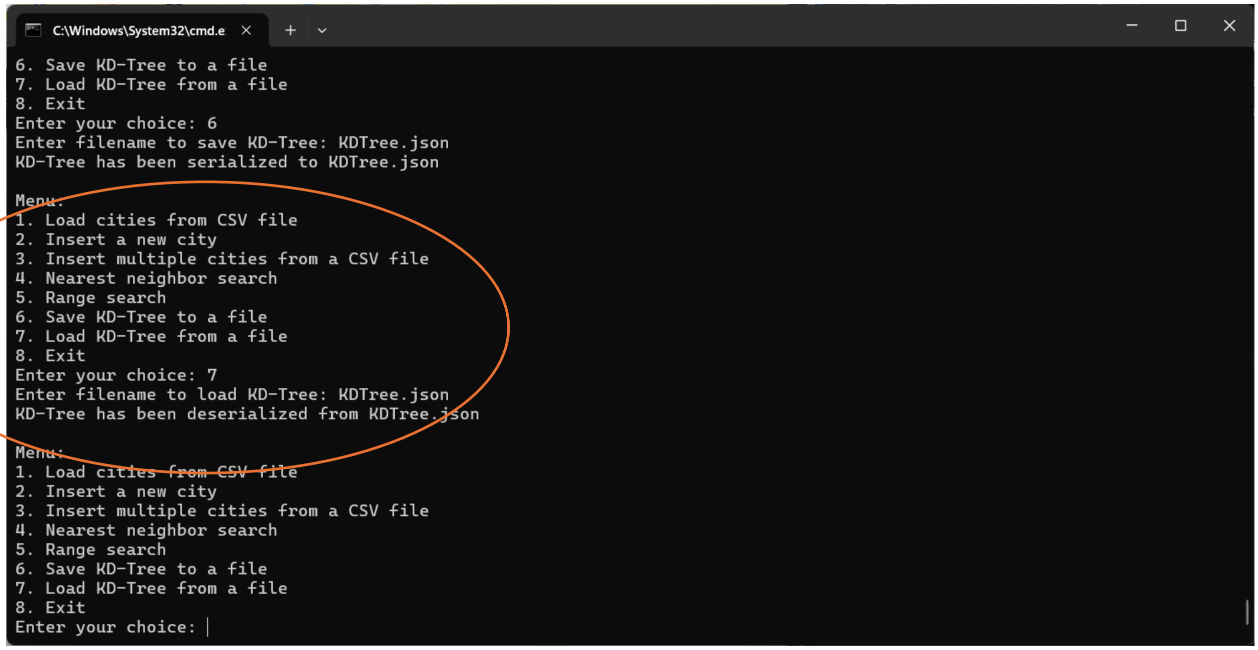


Figure 3.13: Choosing option 7 and inputing .json file to reconstruct KD-Tree from a .json file

IV. References:

[1] [Online]. Available: https://chatgpt.com/.

[2] [Online]. Available: https://www.geeksforgeeks.org/search-and-insertion-in-k-dimensional-tree/?ref=header_outind.

[3] D. E. (. Knuth, "The Art of Computer Programming, Volume 3: Sorting and Searching (2nd ed.)," Addison-Wesley.

[4] [Online]. Available: https://www.youtube.com/watch?v=Glp7THUpGow.

[5] [Online]. Available: https://bitbucket.org/StableSort/play/src/master/src/com/stablesort/kdtree/KDTree.java.