QUAD TREES

November 5, 2023

Krushna Goel (2022MCB1269) ,
Shweta Maurya (2022MCB1281) ,

Instructor:
Dr. Anil Shukla

Teaching Assistant:
Manpreet Kaur

Summary: In our project, we initially implemented a Quadtree data structure, which includes an integral search function. This function employs the fundamental concept of Quadtree by dividing the spatial domain into four quadrants. It effectively illustrates the process through which the search function navigates from one quadrant to another.

Subsequently, we addressed a specific challenge related to determining the nearest city or hub for various purposes, such as location-based services and delivery logistics. This involved the acquisition of geographical coordinates for a given point, followed by a meticulous processing step. The outcome of this process is the identification of the city that is in closest proximity to the provided coordinates, which is a crucial aspect of optimizing location-based decision-making.

1. Introduction

Transportation and Traveling are an integral part of our daily lives, often operating behind the scenes in ways we might not consciously recognize. Whether it's sending a parcel, delivering crucial medical supplies, or traveling for business or leisure, transportation systems ensure that we and our goods reach our destinations.
Consider the process of sending a parcel. It could be a gift for a loved one, an online shopping delivery, or important business documents. In each case, transportation systems ensure these items reach their intended destinations. Similarly, in the medical field, transportation is vital for the timely delivery of medical supplies such as vaccines, medications, and life-saving equipment to hospitals and clinics.
In all these scenarios, speed is often of the essence. So how can we ensure swift transportation? One way is by determining the shortest distance between the source and destination. For instance, in a medical emergency, we might need to deliver supplies to a specific city as quickly as possible. Or when sending a parcel to a remote area, it might be faster to first send it to the nearest city.

To address this challenge, we have implemented Quad Trees, a tool that helps us quickly identify the nearest city or hub. This reduces the time complexity involved in finding the shortest route, thereby speeding up the transportation process.

2. What are Quadtrees ?

Quadtrees are tree structures designed to efficiently manage and organize data points in a two-dimensional space. In a quadtree, each node can have a maximum of four child nodes. The construction of a quadtree involves the following steps:

1. Partition the current two-dimensional space into four smaller boxes.
2. If a box contains one or more data points, create a child node to represent that box and store information

about the two-dimensional space it covers.
3. If a box doesn't contain any data points, it's not necessary to create a child node for it.
4. Recursively apply these steps to each of the child nodes.

2.1. Insertion in Quad trees

Insertion inquadtreess involves placing a new data point into the tree structure while maintaining the quadtree's hierarchical organization. Here's how insertion typically works in quadtrees:

1. **Starting Point-**
   Begin at the root of the quadtree. The root represents the entire two-dimensional space.

2. **Subdivision:**
   Check if the current node (starting with the root) is a leaf node or an internal node. A leaf node is one that doesn't have children (i.e., it's a box that can contain data points). If the current node is a leaf node and can accept more data points, proceed to the next step. If it's an internal node, you need to determine which of its four children to traverse into.

3. **Insertion:**
   If the current node is a leaf node and has room for more data points, check if the new data point belongs within the boundaries of that node (i.e., whether it falls within the box represented by the leaf node). If it does, insert the data point into that node. If it doesn't fit within the current node, subdivide the current node into four child nodes, effectively partitioning the space into four smaller boxes. Then, reinsert all the data points from the current node into the appropriate child nodes. Finally, insert the new data point into one of the child nodes based on its location.

4. **Recursion:**
   After insertion, if you've subdivided a node, you'll continue to traverse the tree recursively. You repeat the process of subdividing and inserting until you reach a leaf node that can accommodate the data point.

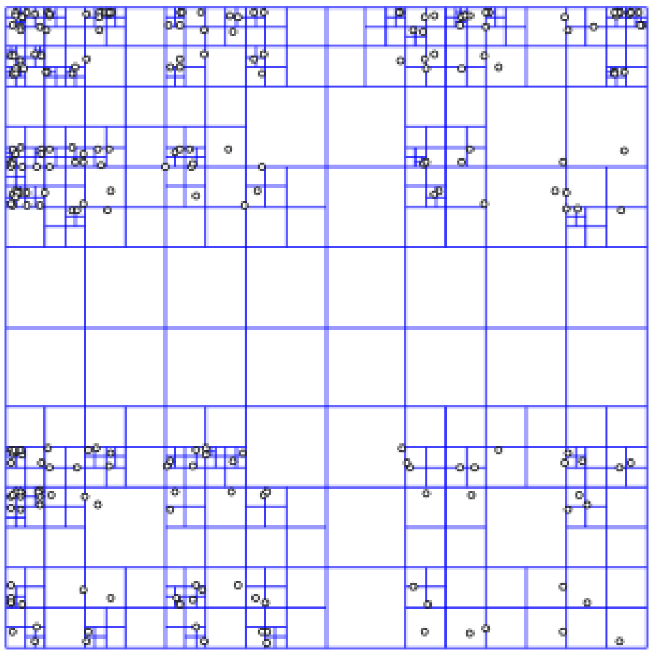


Figure 1: QUAD TREE

2.2. Search in Quad tree

The 'search' function in a Quadtree is responsible for finding a specific point or region within the Quadtree data structure. It works by recursively navigating through the Quadtree's nodes to locate the desired data (e.g., a point or region) that matches the given search criteria. Here's a step-by-step explanation of how the 'search' function typically works:

1. **Start at the Root of the Quadtree:** The search begins at the root node of the Quadtree.

2. **Check if the Current Node is a Leaf Node:** If the current node is a leaf node (i.e., it contains data such as a point or region), the function checks whether the data in that leaf node meets the search criteria.
   - If the data in the current leaf node matches the search criteria, the search is successful, and the function returns the data.
   - If the data doesn't match the search criteria, the search continues.

3. **Determine Which Quadrant the Query Point or Region Falls Into:** The function calculates the midpoint of the current node's bounding box to determine which quadrant (child node) the search point or region belongs to. This is done by comparing the coordinates of the query point or region with the midpoint coordinates.

4. **Recursively Search the Appropriate Child Node:** Based on the determination of the quadrant, the function recursively calls itself on the child node corresponding to that quadrant.

5. **Repeat the Search in the Child Node:** Steps 2 to 4 are repeated in the child node. The function continues to navigate deeper into the Quadtree, considering the quadrants at each level, until it either finds the desired data that matches the search criteria or determines that the data doesn't exist within the Quadtree.

6. **Terminating the Search:** The search terminates when one of the following conditions is met:

   - The search criteria are met, and the desired data is found in a leaf node. In this case, the function returns the data.
   - The entire Quadtree has been traversed without finding a match for the search criteria. In this case, the function terminates without finding a match.

3. Our take on QUAD TREE

In the problem we addressed, we utilized a quadtree to incorporate a node into a quad that included the following components:

1. Two sets of coordinates, defining the position as x, y.

2. The identification of the city located at that specific position.

3. Additionally, within each quad, we recorded the count of cities present.

4. We are also storing the last city inserted into the particular quad

We performed the insertion operation consistent with the standard procedure for quadtree data structures.

4. Algorithms

We used two algorithms for our quad tree:- Insertion and Searching.

4.1. Insertion

The `insert` function is used to insert a node into a quadtree. Here's how it works:

1. **Input Parameters**: The function takes three parameters: a pointer to the root of the quadtree, a node to be inserted, and the name of the city associated with the node.

2. **Midpoint Calculation**: The function calculates the midpoints of the x and y coordinates of the current node's bounding box.

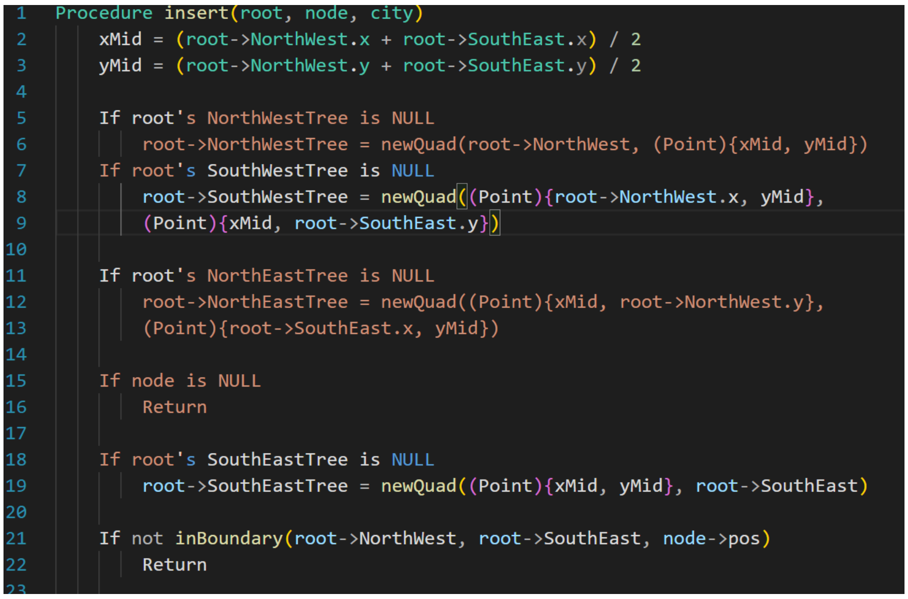
3. **Child Node Creation**: If any of the child nodes of the root (NorthWestTree, SouthWestTree, NorthEastTree, SouthEastTree) are NULL, the function creates new quads for them.

4. **Base Cases**: If the node to be inserted is NULL, the function returns. If the position of the node to be inserted is not within the boundary of the root, the function also returns. If the width and height of the root are both less than or equal to 1, the function checks if the root's node is NULL. If it is, the function assigns the node to be inserted to the root's node and returns.

5. **Node Insertion**: The function increments the number of cities in the root and assigns the city name to the root. It then checks which quadrant the node to be inserted falls into based on the midpoints calculated earlier.

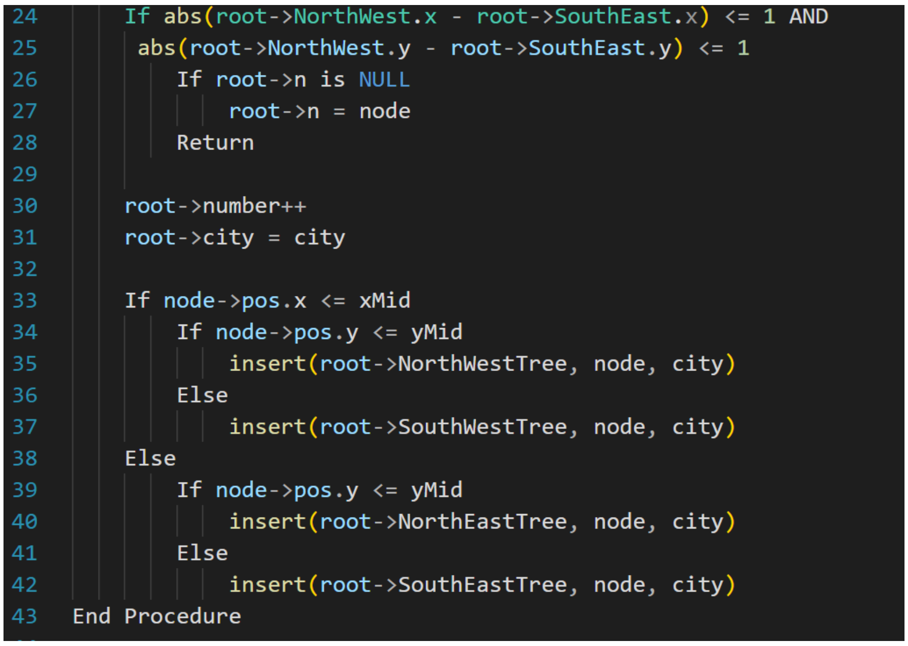
6. **Recursive Insertion**: Depending on the quadrant in which the node falls, the function recursively calls itself with the corresponding child of the root (NorthWestTree, SouthWestTree, NorthEastTree, or SouthEastTree).

Here is the pseudo-code -



```
Procedure insert(root, node, city)
    xMid = (root->NorthWest.x + root->SouthEast.x) / 2
    yMid = (root->NorthWest.y + root->SouthEast.y) / 2

    If root's NorthWestTree is NULL
        root->NorthWestTree = newQuad(root->NorthWest, (Point){xMid, yMid})
    If root's SouthWestTree is NULL
        root->SouthWestTree = newQuad((Point){root->NorthWest.x, yMid},
        (Point){xMid, root->SouthEast.y})

    If root's NorthEastTree is NULL
        root->NorthEastTree = newQuad((Point){xMid, root->NorthWest.y},
        (Point){root->SouthEast.x, yMid})

    If node is NULL
        Return

    If root's SouthEastTree is NULL
        root->SouthEastTree = newQuad((Point){xMid, yMid}, root->SouthEast)

    If not inBoundary(root->NorthWest, root->SouthEast, node->pos)
        Return
```

Figure 2: Insertion algorithm



```
    If abs(root->NorthWest.x - root->SouthEast.x) <= 1 AND
     abs(root->NorthWest.y - root->SouthEast.y) <= 1
        If root->n is NULL
            root->n = node
        Return

    root->number++
    root->city = city

    If node->pos.x <= xMid
        If node->pos.y <= yMid
            insert(root->NorthWestTree, node, city)
        Else
            insert(root->SouthWestTree, node, city)
    Else
        If node->pos.y <= yMid
            insert(root->NorthEastTree, node, city)
        Else
            insert(root->SouthEastTree, node, city)
End Procedure
```

Figure 3: continuation ::

4.2. Search Function

The `search` function is used to search for a point in a quadtree. Here's how it works:

1. **Input Parameters**: The function takes three parameters: a pointer to the root of the quadtree, a point ‘p‘ that we are searching for, and a string ‘path‘ that keeps track of the path taken to reach the current node.

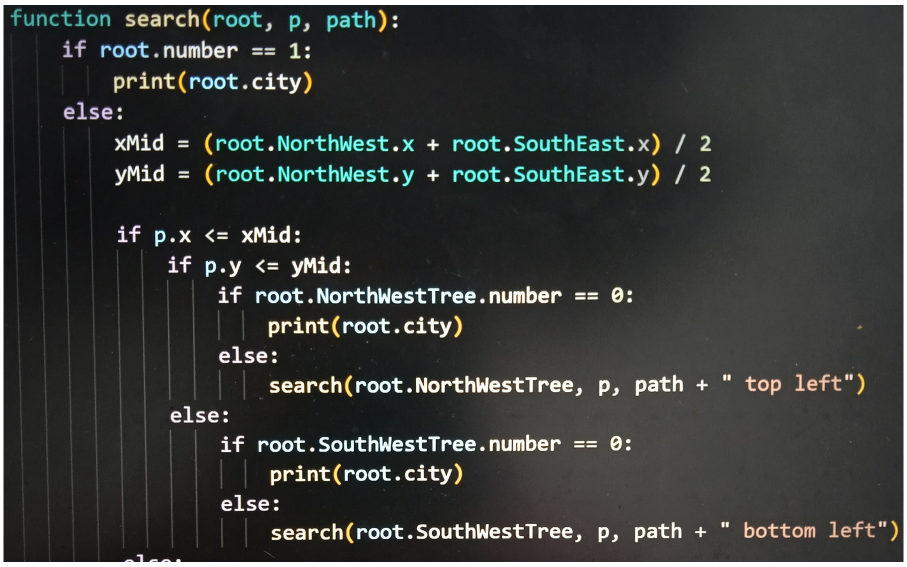
2. **Base Case**: If the current node (‘root‘) is a leaf node (i.e., it contains only one city), the function prints the city's name and returns.

3. **Recursive Case**: If the current node is not a leaf node, the function calculates the midpoints of the x and y coordinates of the current node's bounding box. It then checks which quadrant the point ‘p‘ falls into based on these midpoints.

4. **Quadrant Determination**: Depending on the quadrant in which the point ‘p‘ falls, the function recursively calls itself with the corresponding child of the current node (‘root->NorthWestTree‘, ‘root->SouthWestTree‘, ‘root->NorthEastTree‘, or ‘root->SouthEastTree‘). The ‘path‘ parameter is updated to reflect the quadrant.

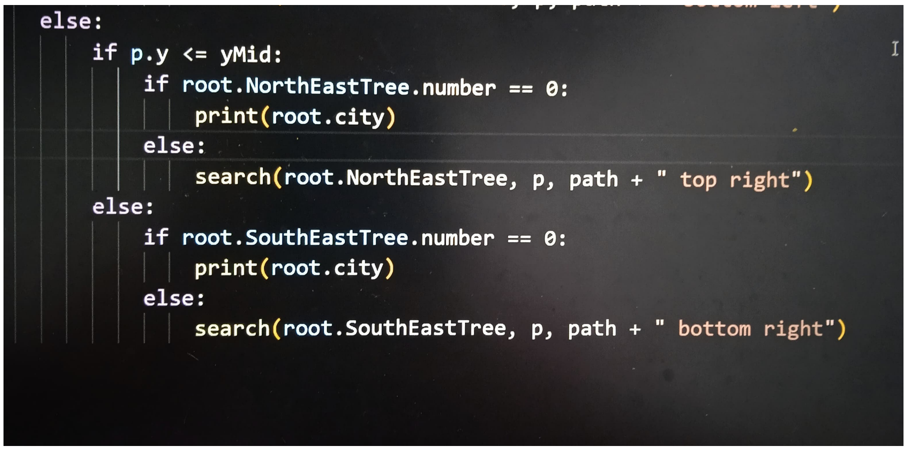
5. **Edge Case Handling**: Before making the recursive call, the function checks if the child node is a leaf node. If it is, the function prints the city's name and returns. This is to handle the case where the point ‘p‘ falls into a quadrant that does not contain any cities.

Here is the pseudo-code -



```
function search(root, p, path):
    if root.number == 1:
        print(root.city)
    else:
        xMid = (root.NorthWest.x + root.SouthEast.x) / 2
        yMid = (root.NorthWest.y + root.SouthEast.y) / 2

        if p.x <= xMid:
            if p.y <= yMid:
                if root.NorthWestTree.number == 0:
                    print(root.city)
                else:
                    search(root.NorthWestTree, p, path + " top left")
            else:
                if root.SouthWestTree.number == 0:
                    print(root.city)
                else:
                    search(root.SouthWestTree, p, path + " bottom left")
```

Figure 4: here is only if part



```
else:
    if p.y <= yMid:
        if root.NorthEastTree.number == 0:
            print(root.city)
        else:
            search(root.NorthEastTree, p, path + " top right")
    else:
        if root.SouthEastTree.number == 0:
            print(root.city)
        else:
            search(root.SouthEastTree, p, path + " bottom right")
```

Figure 5: continuation :: Else part

This function effectively implements a point location query on the quadtree, which can be used to find the city closest to a given point. It demonstrates the efficiency of quadtrees in spatial indexing and search operations

5. Graph

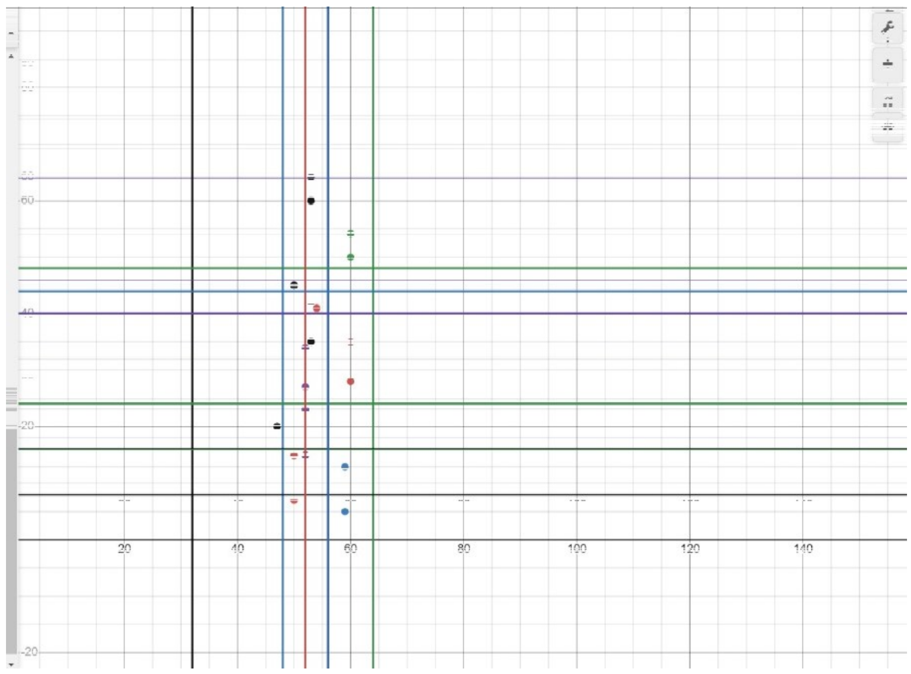


Figure 6: Graphical representation of Quad tree using some of the points we took in our test case

6. Complexties

	Best case	Average case	Worst case
Insertion	O(1)	O(log n)	O(log n)
Search	$O(1)$	$O(logn)$	$O(logn)$

7. Applications of quadtree

Quadtrees are a type of tree data structure used in computer science and GIS (Geographic Information Systems) to represent two-dimensional space. They are particularly useful for tasks such as finding the closest node to a point.

7.1. Fast delivery

This property of quadtrees can be applied to find the nearest airport to a given point. This can reduce delivery time as it will reduce the distance to be covered via road.

7.2. Medical Emergencies

This property of quadtrees can be applied in various fields, including fast shipping or transport of blood/organs in medical emergencies. By representing the locations of hospitals or blood banks as nodes in a quadtree, we can quickly find the closest source of the required blood or organ to the point of demand.

7.3. Conclusion

Thus, quadtrees provide an efficient solution to the problem of finding the closest node to a point in two-dimensional space, with potential applications in critical areas such as medical emergencies.

8. Further Suggestions for Quadtree Implementation

As a team, we have implemented a quadtree which has proven to be effective in finding the nearest node to a given point. This has potential applications in various fields, including fast shipping or transport of blood/organs in medical emergencies. However, there is always room for improvement and expansion. In this section, we propose some further enhancements to our quadtree implementation.

8.1. Dynamic Resizing

If our quadtree implementation is static (i.e., it doesn't change its size once created), we might consider making it dynamic. This means the quadtree could expand or shrink based on the number of nodes it needs to accommodate. This could make our quadtree more memory efficient.

8.2. Balancing

Just like binary trees, quadtrees can become unbalanced, with much more nodes in one quadrant than the others. Implementing a balancing algorithm could help ensure that our quadtree remains efficient even when nodes are not evenly distributed.

8.3. Visualization

Implementing a way to visualize our quadtree could be very helpful for debugging and understanding its behavior. This could be as simple as a text-based printout of the tree structure, or as complex as a graphical user interface.

8.4. Location-Based Services

We could extend our project to include more location-based services. For example, we could implement a range search function that returns all nodes within a certain distance of a given point. This could be useful in applications like local search or geographic clustering.

9. Conclusions

We were able to find the nearest node to a point. Although this is usually done using Djikshtra's algorithm , we were able to achieve similar results using a quadtree. This happened due to out modifications in quad tree like counting the number of nodes in each quad and naming each quad with the city last inserted into it.

10. Bibliography and citations

Acknowledgements

We wish to thank our instructor, Dr Anil Shukla and our Teaching Assistant Manpreet Kaur for their guidance and valuable inputs provided throughout our project.

References

1. https://jimkang.com/quadtreevis/

2. https://www.geeksforgeeks.org/quad-tree/

3. ChatGpt