CS5633: Analysis of Algorithms
Term Project – Algorithm I

Topic: “Quad Tree, R Tree, and Variant Data Structures for Geographic Information Systems (GIS)”

Submitted By:
Jaidheer Sirigineedi (xak202)
Anshu Tripathi (wbb206)

Problem Statement:

Geographic Information Systems (GIS) are computer-based information systems that provide tools to collect, integrate, manage, analyze, model, and display data that is referenced to an accurate cartographic representation of objects in space. They play a crucial role in applications such as urban planning, navigation, and environmental monitoring. With increasing GIS dataset sizes, efficiently storing and querying spatial data becomes a challenge. Traditional linear search methods are computationally expensive, therefore making specialized spatial data structures essential for optimizing range searches, nearest-neighbor queries, and spatial joins.

Objective:

The aim of this report is to implement **Quad Trees** for spatial indexing in Geographic Information Systems (GIS). We utilize real-world data obtained from Overpass Turbo in GeoJSON format to evaluate the effectiveness of the QuadTree data structure. This study will analyze their efficiency in terms of insertion time, query speed, and memory usage, providing insights into the most effective indexing technique for large-scale GIS applications. The primary operations performed on a QuadTree include insertion, point search, bounding box search, and nearest neighbor search. Through experimental evaluation and performance comparison, this report aims to highlight the strengths and limitations of using QuadTrees for spatial indexing and identify potential areas for optimization and improvement.

Source Code:

The source code files are uploaded separately. They contain the algorithm implementation, and the generated graphs based on the obtained data. Also, the above source code is uploaded to Github which can be found at https://github.com/anshutripathi11/GIS-Project

Algorithm-1 Description:

Quad Tree:

The QuadTree is a spatial data structure that efficiently manages two-dimensional spatial data. It recursively divides the space into four quadrants or regions, forming a hierarchical tree structure. Each internal node of the QuadTree has exactly four children, which represent the four quadrants: North-West (NW), North-East (NE), South-West (SW), and South-East (SE) (using the map analogy).

Representation of Vector Data:

The data structure that we have adopted to store vector data is the quad tree. When stored in the quadtree, an image corresponding to a collection of points is decomposed into four equal quadrants if it contains more than the set capacity. These are in turn split into subquadrants, sub-subquadrants and so on, until each block contains at most four vectors.

Figure 1 demonstrates the hierarchical structure of the QuadTree as applied to a set of spatial points. Each rectangle in the visualization represents a quadrant or subquadrant generated from splitting a region containing more than four points. The distribution of points within the space determines the depth and density of subdivisions, with denser areas resulting in deeper trees and finer granularity. The visualization shows that the QuadTree effectively partitions the space to accommodate varying point densities, demonstrating its ability to manage complex spatial distributions while maintaining efficient data access.

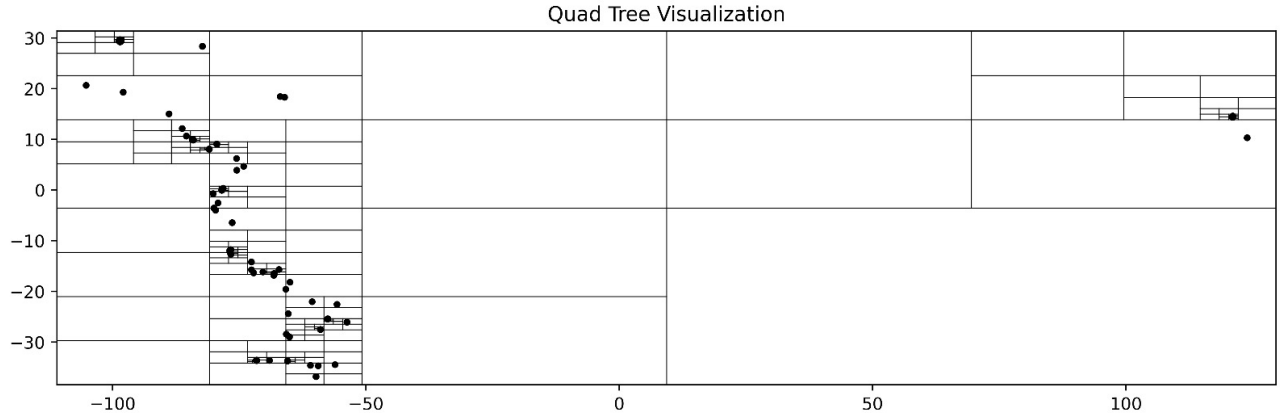

Figure 1: Hierarchical structure of the QuadTree

Insertion Algorithm:

The insertion algorithm starts by checking whether the point to be inserted lies within the boundary of the current node. If the node has not reached its capacity, the point is simply added to the node. However, if the node reaches its capacity and is not a leaf, it subdivides into four child quadrants and redistributes the existing points among them. The insertion continues recursively until the point is added to the appropriate sub-quadrant. This process ensures that the tree remains balanced when points are evenly distributed, but if many points cluster in one region, the node may repeatedly split, increasing the depth.

Step-by-Step Process of Insertion:

Start at the root node.
If the node does not exceed capacity, store the point in the node.
If the node exceeds its capacity:

Subdivide the current node into 4 child quadrants: upper left (ul), upper right (ur), lower left (ll), lower right (lr).

Move the existing points into the correct children.

Recursively insert the new point into the correct child based on its coordinates.

Point Search Algorithm:

The point search algorithm starts by checking if the query point is within the boundary of the current node. If so, the algorithm moves to the appropriate child quadrant containing the point,

recursively searching through the relevant sub-quadrants. If the function finds the point, it returns it; if not, it searches for other overlapping quadrants.

Step-by-Step Process of Point Search:

Start at the root node.
Check if the point lies within the bounding box of the current node.
If not, return None.
If yes:
    If the point exists in the current node, return it.
    Otherwise, recursively search in the appropriate child (ul, ur, ll, lr) based on location.
If all children are searched and point not found, return `None`.

Range Search Algorithm:

The range search using the bounding box search algorithm starts by checking whether the boundary of the current node intersects with the query bounding box. If there is no intersection, the algorithm skips that quadrant entirely. However, if an intersection exists, the algorithm checks each point within the node to see if it lies within the bounding box. It then recursively examines relevant child nodes to collect all the points within the bounding box.

Step-by-Step process of Range Search:

Start at the root node.
If the node's bounding box does not intersect with the input bounding box, return empty list.
If it does:
    Check each point in the current node: if it's inside the input bounding box, add it to the result.
Recursively call this process on all children (ul, ur, ll, lr).
Return the collected points.

Nearest Neighbor Algorithm:

The nearest neighbor search algorithm calculates the distance between the query point and each point in the current node. If the boundary of the current node is closer than the nearest distance found, the search continues recursively into the node's children. The algorithm prunes parts of the tree that cannot contain closer points, which helps to reduce the search space. The result is the point that has the smallest calculated distance from the query point.

Step-by-Step process of Nearest Neighbor:

Start at the root and find the smallest child node that contains the query point.
Keep track of nodes visited in a stack (searched_nodes).
Traverse the searched_nodes from smallest to largest, collecting all points and sorting them by Euclidean distance.
If k points are found, estimate a search radius using the farthest among them.

Define a bounding box around the query point using that radius.
Collect all points inside that bounding box.
Sort them again by distance and return the top k.

Time and Space Complexity Analysis:

The time complexity of QuadTree operations depends on the spatial distribution of the points and the depth of the tree.

Insertion Complexity:

- **Best Case:** The best-case time complexity for insertion is O (log n), as points are uniformly distributed, resulting in balanced subdivisions.
- **Worst Case:** The worst-case time complexity for insertion is O(n), as all points fall into one quadrant, forming a tree with dense clustering that becomes highly unbalanced due to clustered points.
- **Space Complexity:** The space complexity for insertion is O(n), as it stores all points and sub-nodes.

Point Search Complexity:

- **Best Case:** The best-case time complexity for point search is O(log n), as the point is quickly located by traversing a balanced tree.
- **Worst Case:** The worst-case time complexity for point search is O(n), as the point lies in a densely populated quadrant that forms a linear chain.
- **Space Complexity:** The space complexity for point search is O(n), as it stores all points and sub-nodes.

Bounding Box Search Complexity:

- **Best Case:** The best-case time complexity for bounding box search is O(log n + k), where k is the number of points reported, and it is efficient pruning of non-overlapping regions.
- **Worst Case:** The worst-case time complexity for bounding box search is O(n), as the bounding box overlaps most quadrants and forcing a complete traversal.
- **Space Complexity:** The space complexity for bounding box search is O(n), as it stores the complete tree structure.

Nearest Neighbor Search Complexity:

- **Best Case:** The best-case time complexity for nearest neighbor search is O(log n), as the nearest point is located in a small region early.
- **Worst Case:** The worst-case time complexity for nearest neighbor search is O(n)as, all points need to be examined, especially in highly clustered data.
- **Space Complexity:** The space complexity for nearest neighbor search is O(n), as due to storing points and tree nodes.

Performance Analysis:

Insertion Algorithm:

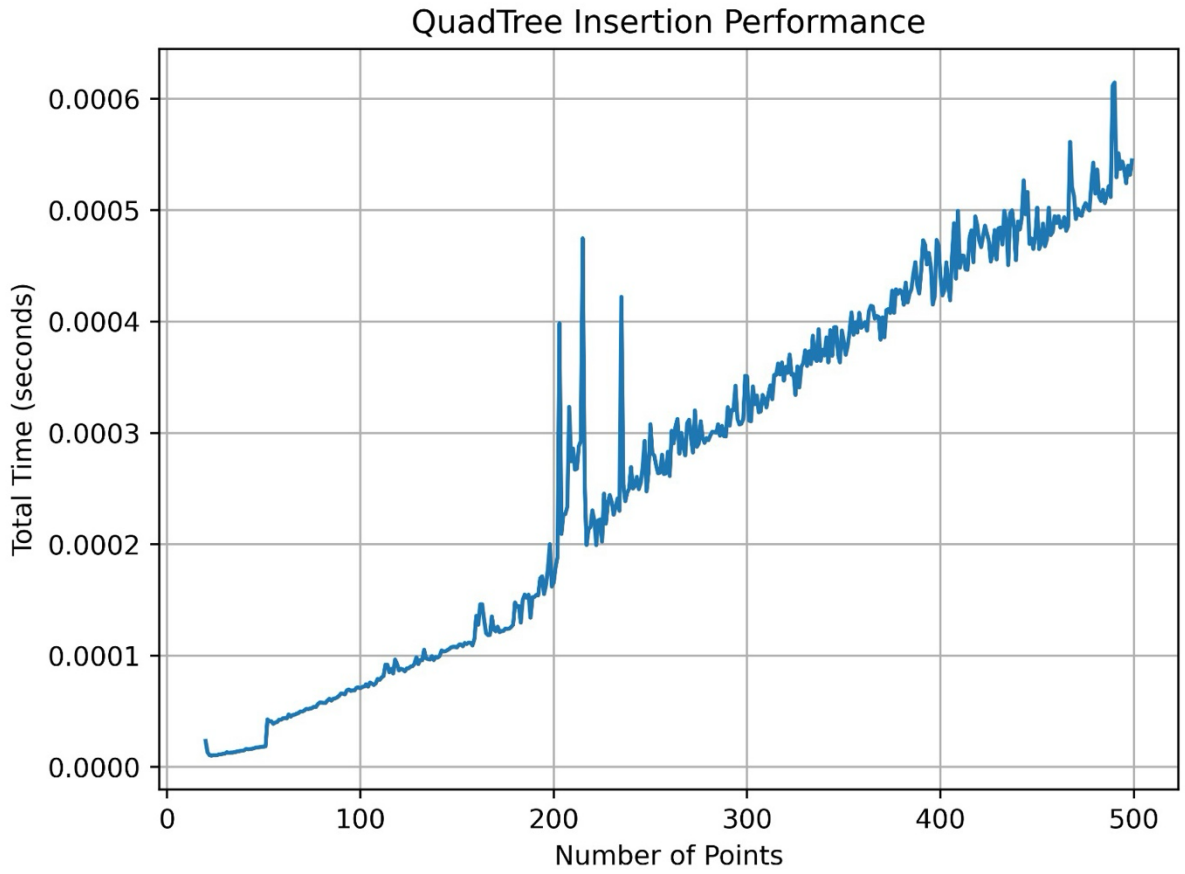

Figure 2: QuadTree Insertion Performance

The graph presented shows the performance of QuadTree insertion as the number of stored points increases from 0 to 500. The x-axis represents the number of points, while the y-axis shows the total insertion time in seconds. As the number of points increases, the graph demonstrates a general upward trend in insertion time, indicating that the QuadTree becomes progressively slower as more points are added. This trend is expected since the insertion time complexity for a balanced QuadTree is approximately O(log n), where n is the number of points.

Quad Tree Insertion vs Varying Capacity:

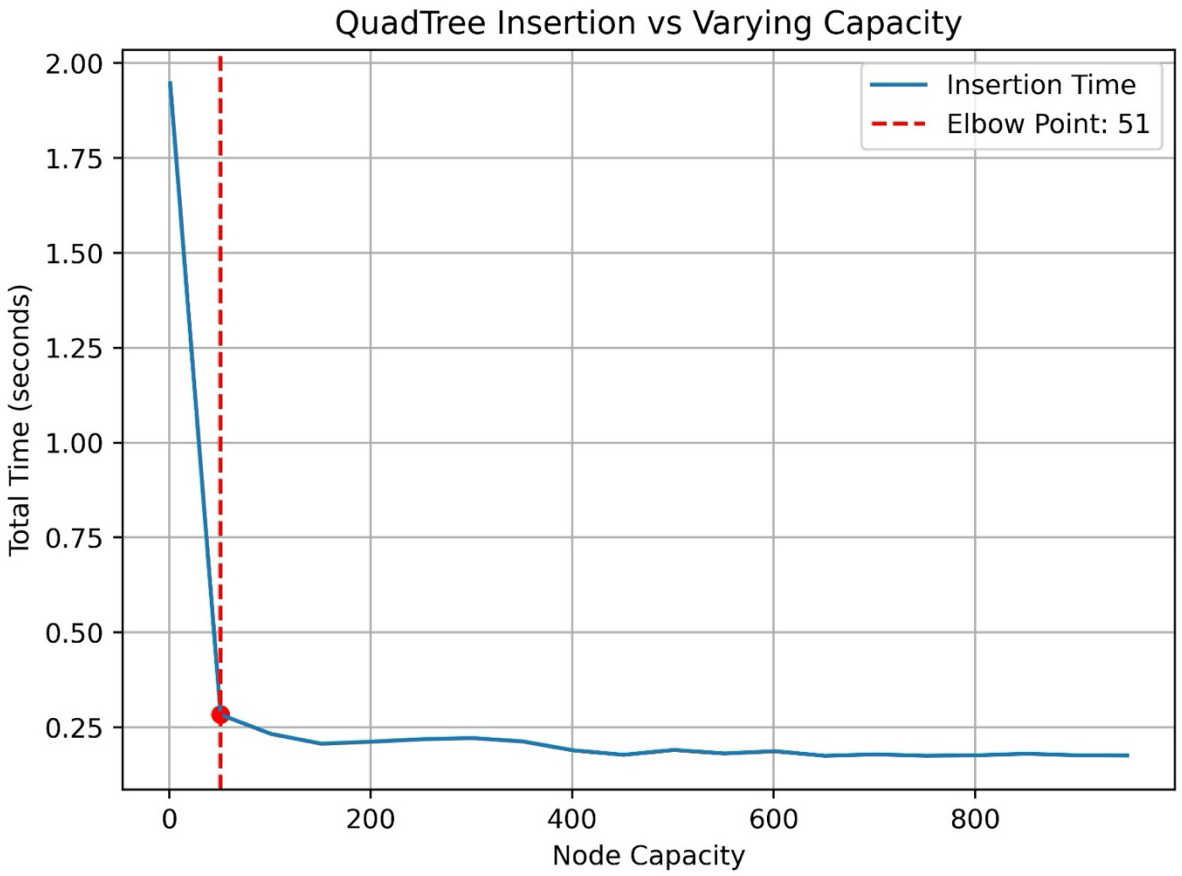

Figure 3: QuadTree Insertion vs Varying Capacity

The graph presented shows the performance of QuadTree insertion as a function of the node capacity. The x-axis represents the node capacity, which varies from 0 to 900, while the y-axis shows the total insertion time in seconds. The solid blue line represents the insertion time, and the dashed red line indicates the elbow point at a node capacity **51**. This elbow point highlights the optimal balance between insertion efficiency and capacity.

The graph shows that insertion time decreases sharply as the node capacity increases from 0 to approximately 50. This rapid decline is due to a reduction in the frequency of node splits. When the node capacity is small, the QuadTree must split frequently as more points are added, resulting in a significant overhead. As the capacity increases, nodes can accommodate more points before splitting, thereby reducing the total insertion time.

After reaching the elbow point (capacity of 51), the insertion time stabilizes and remains nearly constant as the capacity increases. This indicates that further increasing the node capacity does not provide significant performance improvements after reaching the optimal capacity. Instead, it only results in larger leaf nodes that slightly increase the search time without noticeably reducing the insertion time. Thus, the elbow point represents a trade-off between the frequency of node splitting and insertion efficiency.

Point Search Algorithm:

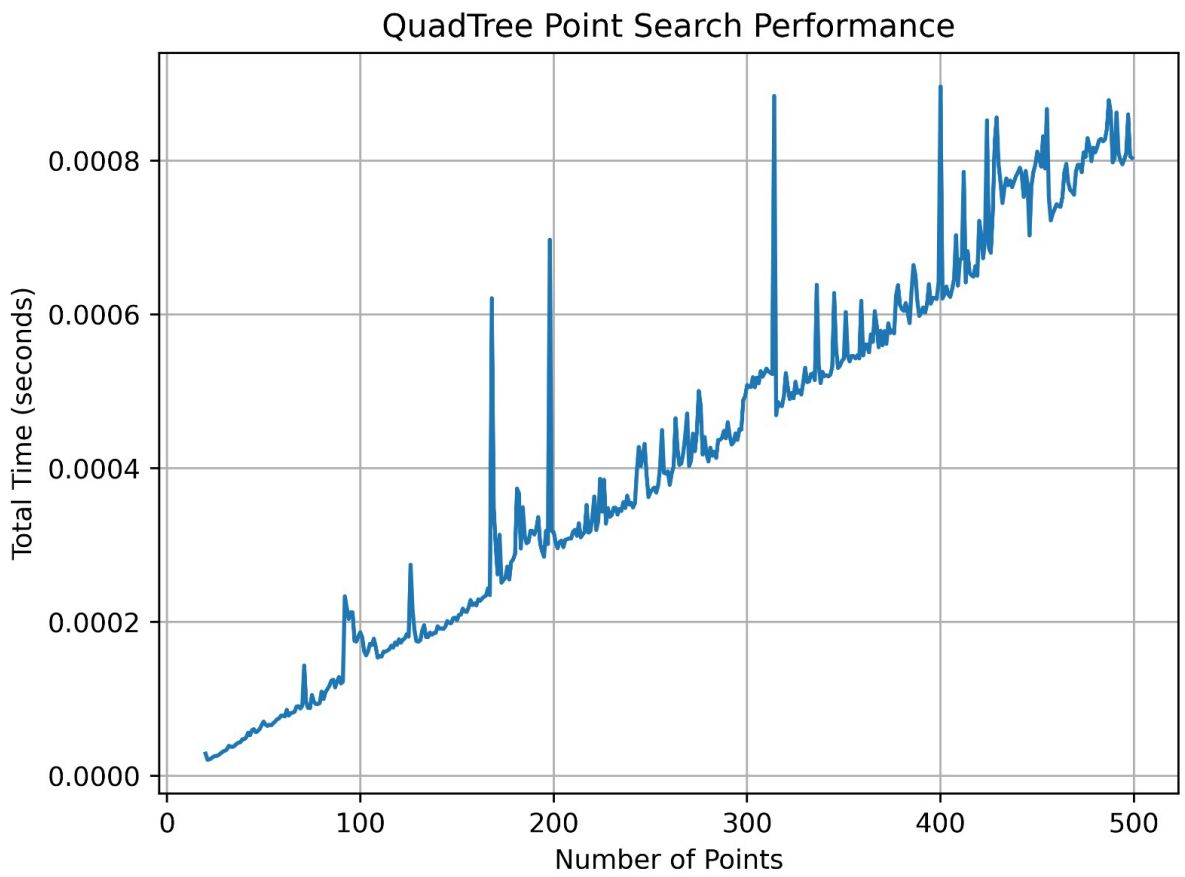

Figure 4: QuadTree Point Search Performance

The graph represents the performance of point search in a QuadTree data structure as the number of stored points increases. On the x-axis, the number of points ranges from 0 to 500, while the y-axis represents the total time taken for point search, measured in seconds. As the number of points increases, the graph demonstrates a general upward trend in search time, indicating that the QuadTree becomes progressively slower as it stores more points. This trend aligns with the expected performance of a balanced QuadTree, where the search time complexity is approximately O(log n).

Bounding Box Algorithm:

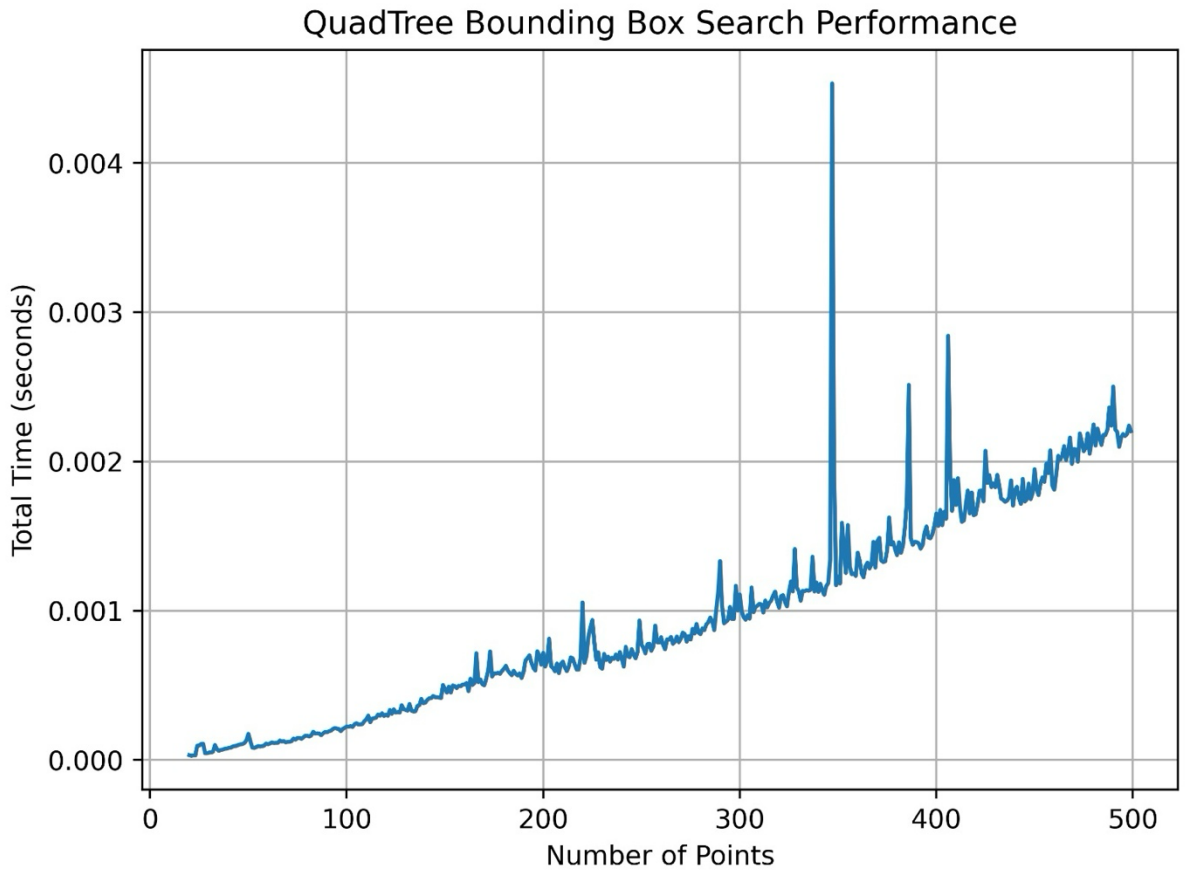

Figure 5: QuadTree Bounding Box Search Performance

The graph presented shows the performance of bounding box search in a QuadTree data structure as the number of points increases from 0 to 500. The x-axis represents the number of points, while the y-axis shows the total search time in seconds. The graph exhibits a general upward trend in search time as the number of points grows, indicating that the bounding box search becomes progressively slower with increasing points. This behavior aligns with the expected performance characteristics of a QuadTree, where the search time complexity is approximately O(log n + k), where n is the number of points and k is the number of points reported within the bounding box.

Comparison of Quad Tree and Brute Force Approach using Bounding Box Algorithm:

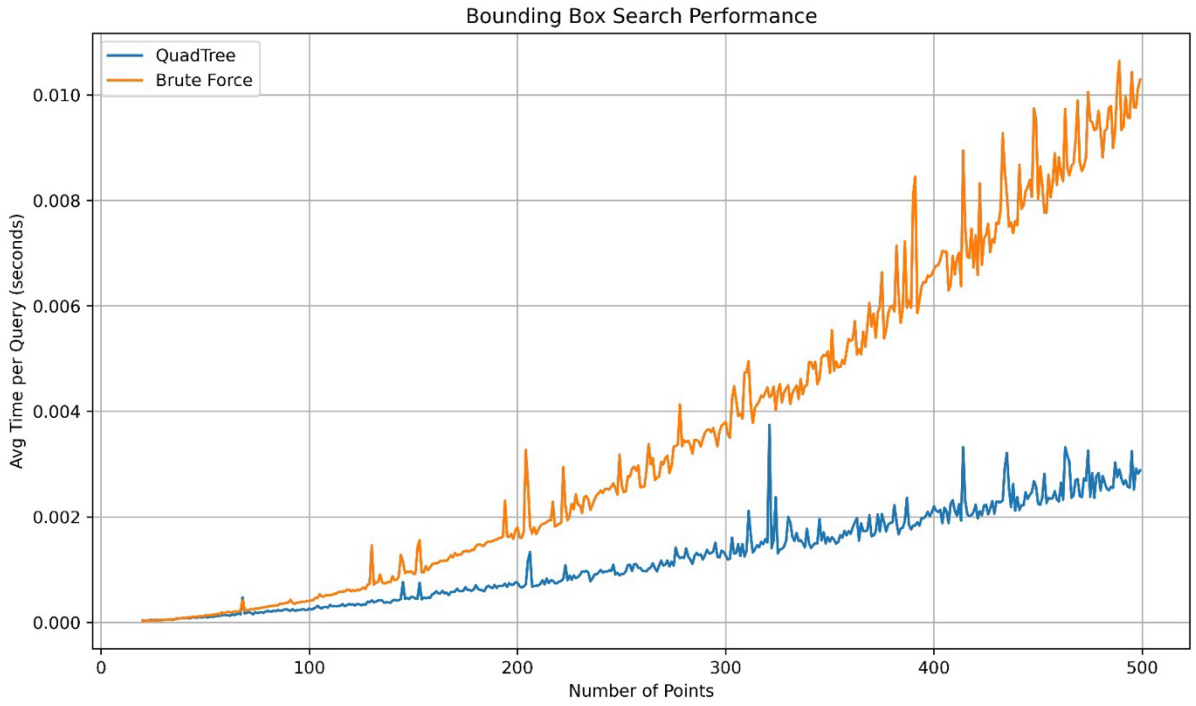

Figure 6: QuadTree vs Brute Force Bounding Box Search Performance

The graph presented compares the performance of the QuadTree and Brute Force methods for bounding box searches as the number of points increases. The x-axis represents the number of points, while the y-axis shows the average time per query in seconds. The blue line represents the performance of the QuadTree method, while the orange line represents the performance of the Brute Force method.

The graph demonstrates that the QuadTree method significantly outperforms the Brute Force approach across all point counts. The search time for the QuadTree method shows a gradual upward trend with some fluctuations and occasional spikes, which are likely caused by clustering or uneven point distribution, resulting in deeper tree traversal. In contrast, the Brute Force method exhibits a steep, linear increase in query time, consistent with its O(n) time complexity for each search.

The difference between the two methods becomes increasingly pronounced as the number of points rises, with the Brute Force method becoming disproportionately slower than the QuadTree method. QuadTree substantially reduces the search time due to its ability to eliminate irrelevant quadrants from the search process, thereby reducing the number of comparisons needed. For this reason, the QuadTree is a preferred choice for spatial queries in Geographic Information Systems (GIS) applications that require efficient bounding box searches.

Nearest Neighbor Algorithm:

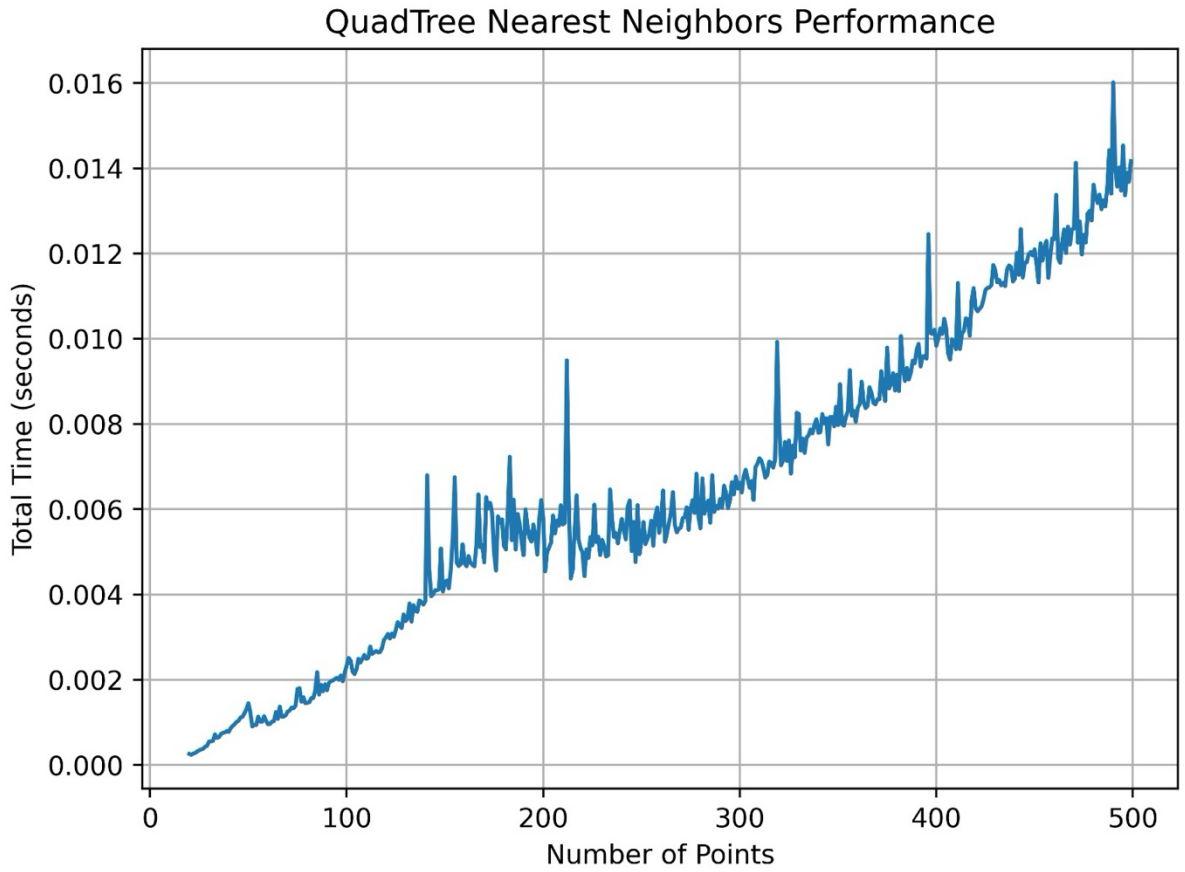

Figure 7: QuadTree Nearest Neighbors Performance

The graph presented shows the performance of the nearest neighbors search operation in a QuadTree data structure as the number of points increases from 0 to 500. The x-axis represents the number of points, while the y-axis shows the total search time in seconds. The graph demonstrates a general upward trend in search time as the number of points grows, indicating that searching the nearest neighbors becomes progressively slower with an increasing number of stored points. This aligns with the expected performance characteristics of a QuadTree, where the search time complexity is approximately O(log n + k), where n is the number of points and k is the number of nearest neighbors being queried.

Comparison of Quad Tree and Brute Force Approach using Nearest Neighbor Algorithm:

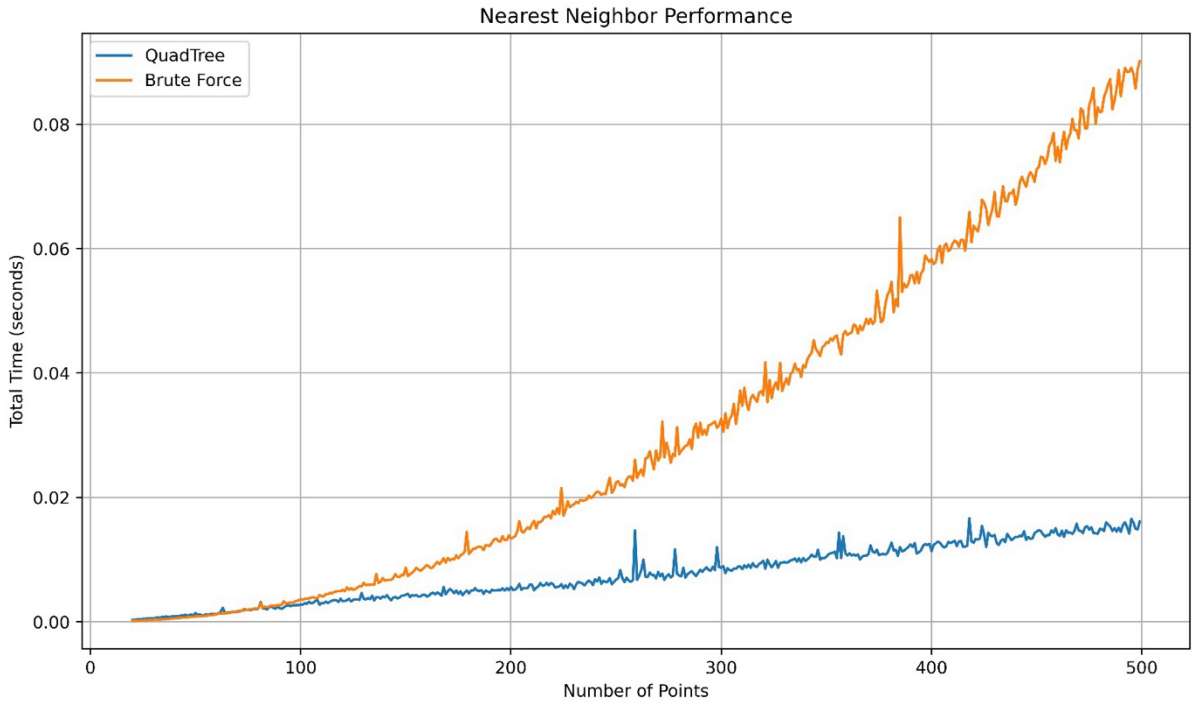

Figure 8: QuadTree vs Brute Force Nearest Neighbors Performance

The graph presented compares the performance of the QuadTree and Brute Force methods for the nearest neighbor search as the number of points increases. The x-axis represents the number of points, while the y-axis shows the total search time in seconds. The blue line indicates the performance of the QuadTree method, while the orange line represents the Brute Force approach.

The graph shows that the QuadTree method consistently outperforms the Brute Force approach across all point counts. The search time for the QuadTree increases gradually, showing a sublinear growth pattern. In contrast, the Brute Force method shows a much steeper growth rate O(n log n + n). This aligns with the time complexity, where Brute Force has O(n) complexity for each query. In contrast, the QuadTree method has an approximate complexity of O(log n + k), where k represents the number of nearest neighbors returned.

The difference between the two methods becomes more evident as the number of points increases, with Brute Force taking significantly longer to complete the nearest neighbor search than the QuadTree approach. The major performance gap highlights the efficiency of the QuadTree structure in reducing search time by utilizing spatial indexing and hierarchical partitioning. On the other hand, the Brute Force method relies on iterating through all points to find the nearest neighbor, resulting in a high computational cost, especially as the dataset grows.

The graph clearly demonstrates the efficiency of the QuadTree approach for nearest neighbor search compared to the Brute Force method. The QuadTree's ability to efficiently index spatial data and minimize search space significantly reduces computation time, making it a more suitable choice for large-scale geographic information systems (GIS).

Real World Analysis:

Parks in San Antonio:

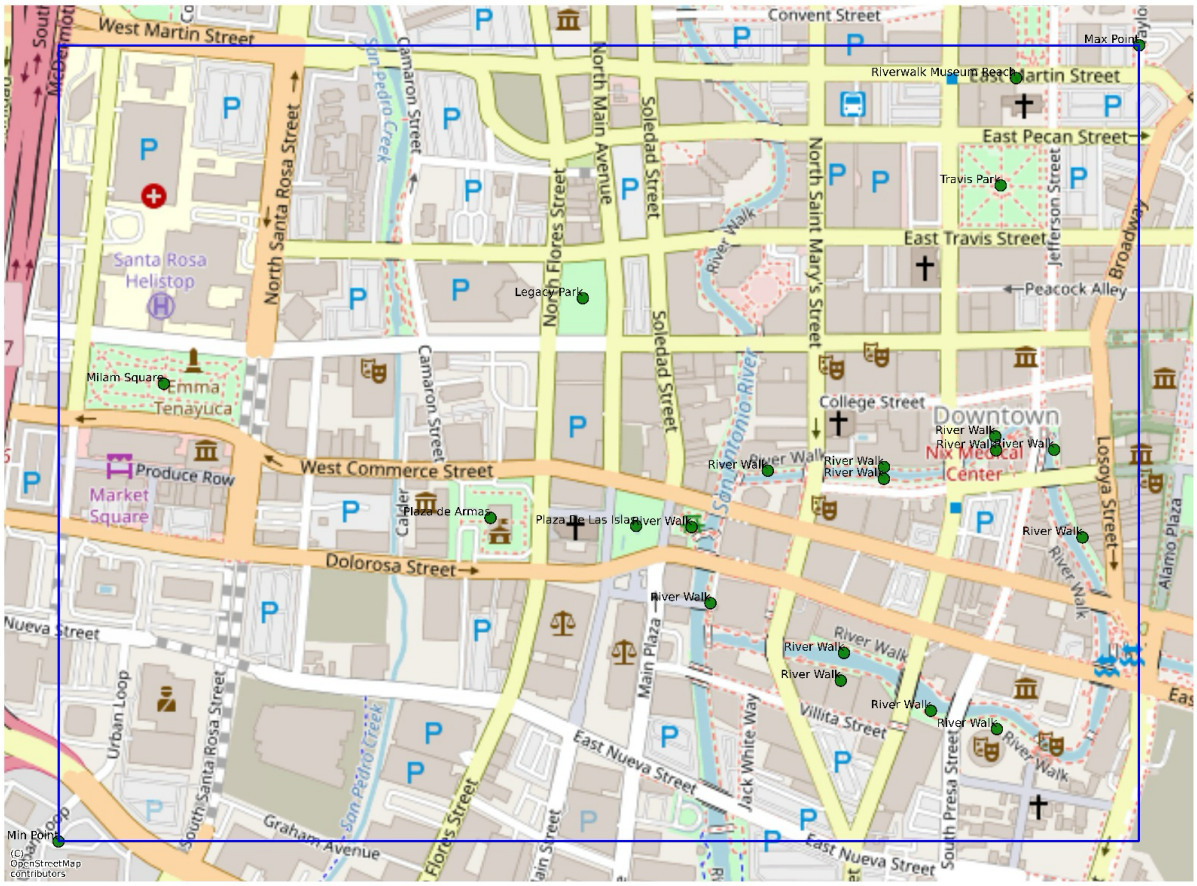

Figure 9: Locate Park using Bounding Box Algorithm

We have identified parks within a specific area in San Antonio using **Range Search**, **which uses the bounding box algorithm** on a city map. The parks are marked with green dots to indicate their locations as detected by our search. By utilizing the QuadTree data structure, the algorithm efficiently locates all parks within the defined spatial region, significantly reducing computational overhead compared to brute-force methods. The bounding box algorithm quickly eliminates quadrants that do not intersect with the query area, allowing for rapid and accurate identification of relevant points. Notable parks within the bounding box include Legacy Park, Milam Square, Travis Park, and several points along the River Walk.

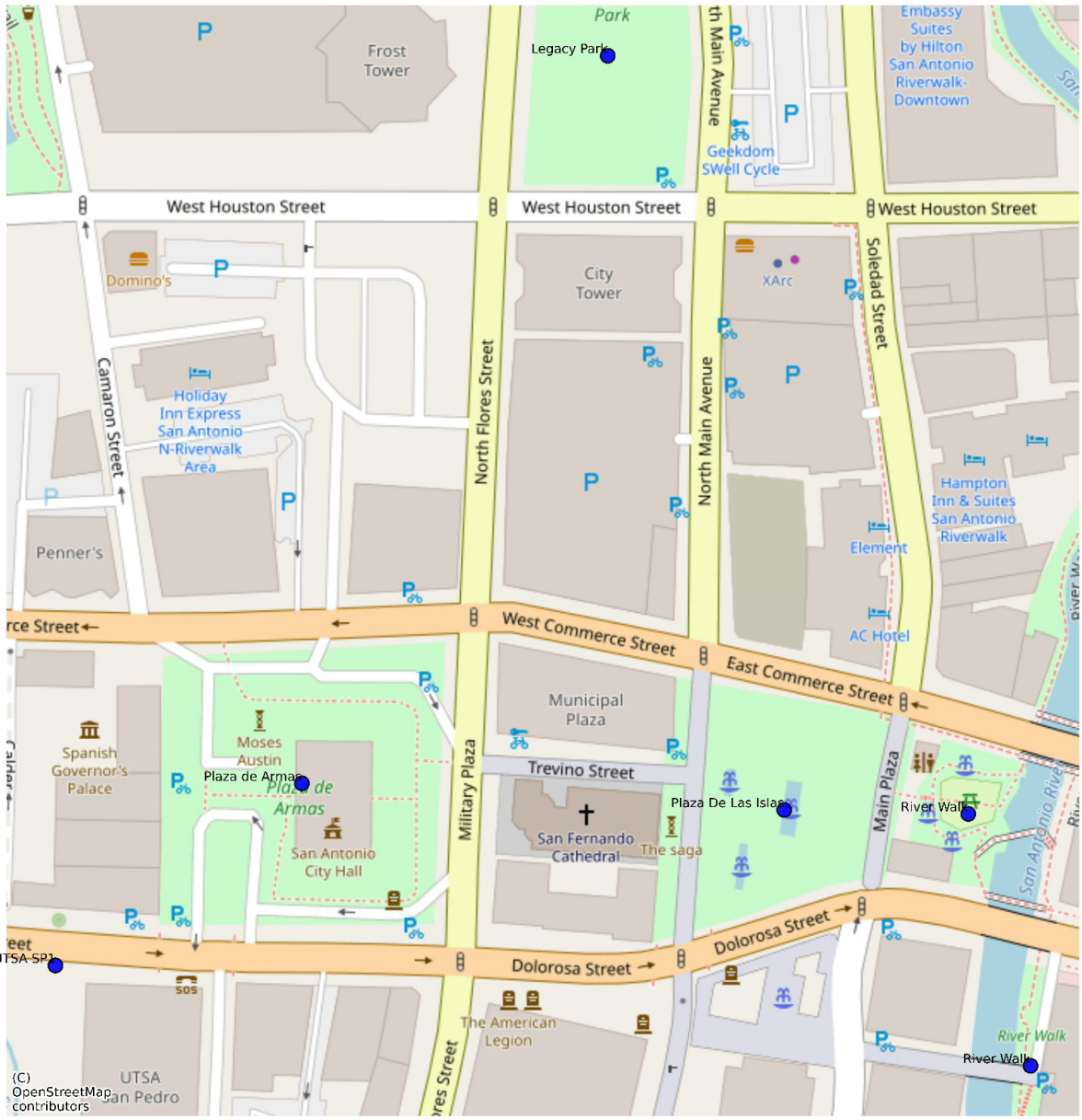

Figure 10: Locate Park using Nearest Neighbor Algorithm

We have identified the closest park near the **UTSA San Pedro building** using the **Nearest-Neighbor Algorithm** on the map of San Antonio. The park is marked with a blue dot, which marks the location determined through this search method. By utilizing the QuadTree data structure, the algorithm efficiently finds the nearest point of interest, significantly reducing the computational cost compared to brute force methods. The nearest-neighbor algorithm navigates relevant quadrants while eliminating areas that cannot contain closer points, enabling rapid and accurate identification of the nearest park. The identified parks in this case are Plaza de Armas, Plaza De Las Islas, River Walk, and Legacy Park.

Fast Food Chain in San Antonio:

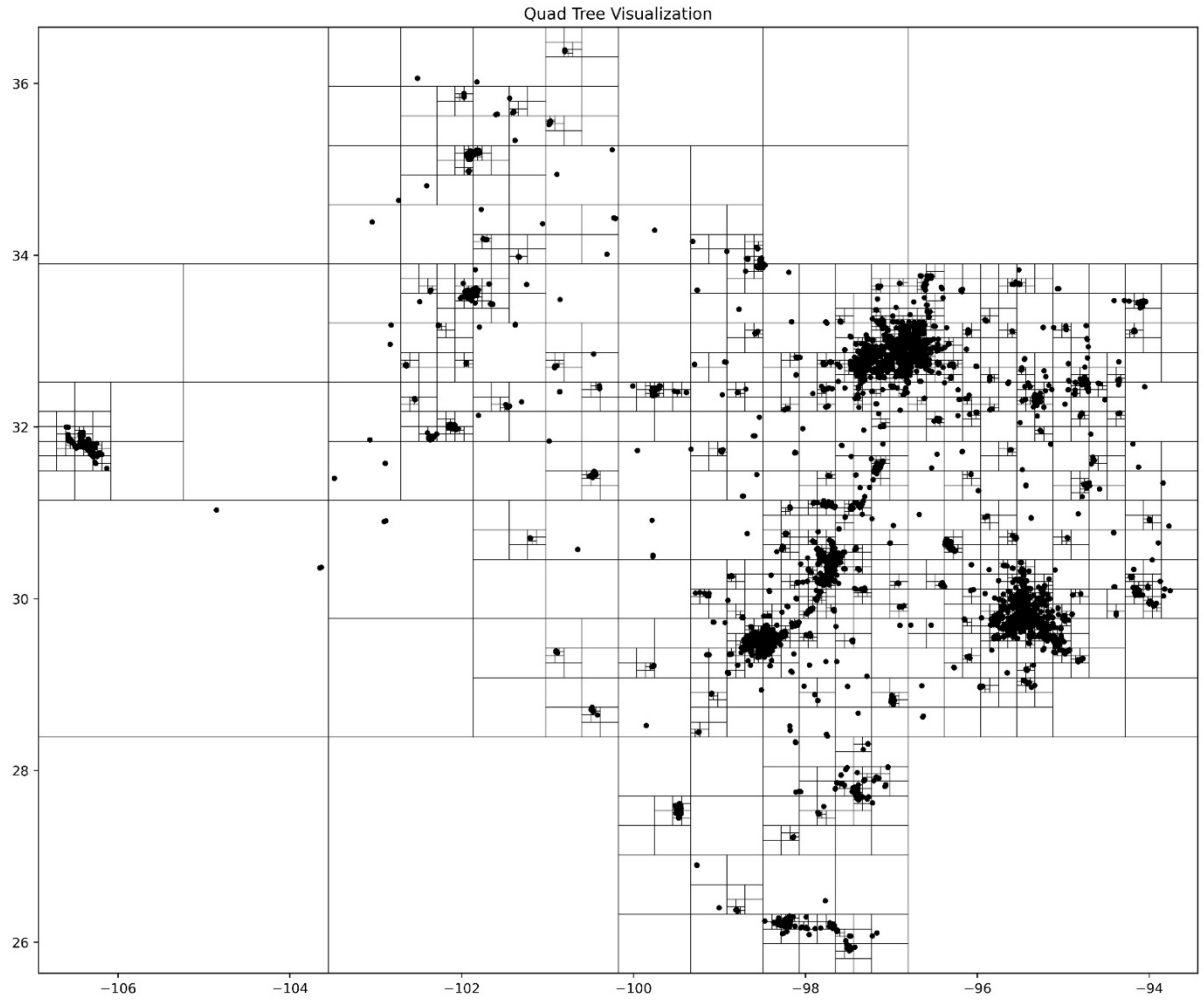

Figure 11: Fast Food Chain Locations across the Texas

This map represents the spatial distribution of fast-food chain locations across the state of Texas, visualized using a QuadTree data structure. It covers major metropolitan areas such as Austin, Dallas, Houston, and San Antonio, where the concentration of points is significantly higher compared to rural and sparsely populated regions. Each black dot represents a fast-food chain location, while the rectangular grid structure illustrates the QuadTree's hierarchical decomposition of space.

The QuadTree effectively partitions the space into quadrants, sub-quadrants, and deeper subdivisions as needed, based on the density of fast-food chain locations. In densely populated urban areas, the QuadTree subdivides into finer grids to accommodate the high concentration of points. In contrast, large empty regions with no fast-food chains remain unpartitioned or minimally subdivided, reflecting the lack of commercial activity in those areas.

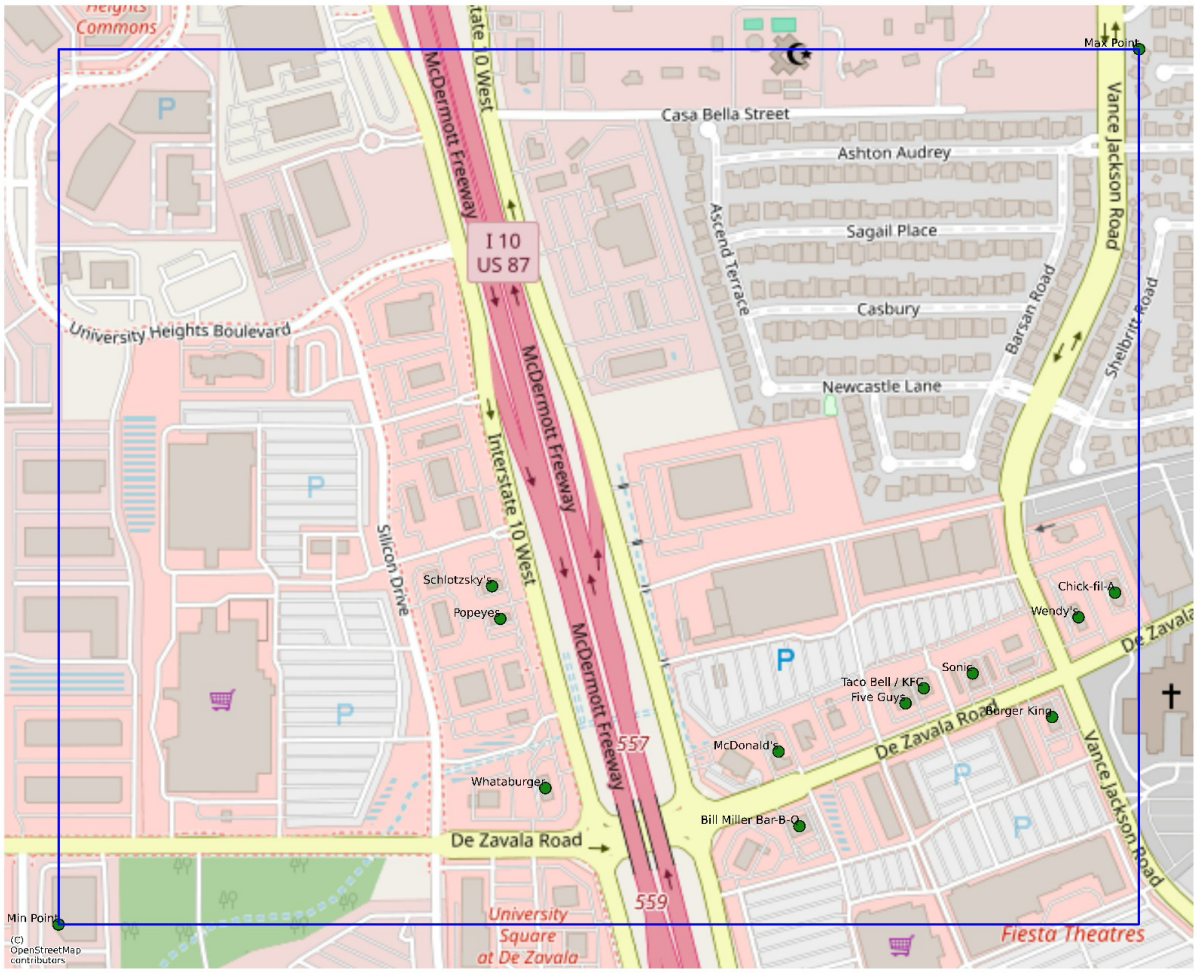

Figure 12: Locate Fast Food Chain using Bounding Box Algorithm

We have identified fast food chain within a specific area in San Antonio using **Range Search**, **which uses the bounding box algorithm** on a city map. The fast-food chain is marked with green dots to indicate their locations as detected by our search. By utilizing the QuadTree data structure, the algorithm efficiently locates all parks within the defined spatial region, significantly reducing computational overhead compared to brute-force methods. The bounding box algorithm quickly eliminates quadrants that do not intersect with the query area, allowing for rapid and accurate identification of relevant points.

Figure 13: Locate Fast Food Chain using Nearest Neighbor Algorithm

We have identified the closest fast-food chain near the **San Antonio International Airport** using the **Nearest-Neighbor Algorithm** on the map of San Antonio. The fast-food chain is marked with a blue dot, which marks the location determined through this search method. By utilizing the QuadTree data structure, the algorithm efficiently finds the nearest point of interest, significantly reducing the computational cost compared to brute force methods. The nearest-neighbor algorithm navigates relevant quadrants while eliminating areas that cannot contain closer points, enabling rapid and accurate identification of the nearest fast-food chain. The identified fast-food chain in this case is Chick-fil-A, Raising Cane's, Whataburger etc.

Reference:

[1] Overpass Turbo. (n.d.). https://overpass-turbo.eu/
[2] Quad Trees A Data Structure for Retrieval on Composite Keys by R. A. Finkel and J. L. Bentley
[3] A geographic information system using quadtrees by Hanan Samet, F Rosenfeld, Clifford a. Shaefer and Robert e. Webber
[4] Encyclopedia of GIS by Shashi Shekhar, Hui Xiong, Xun Zhou