CS5633: Analysis of Algorithms
Term Project – Algorithm II

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

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

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 **R 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 R Tree 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 R Tree 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 R Trees 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 source code is uploaded to GitHub which can be found at https://github.com/anshutripathi11/GIS-Project

Algorithm-2 Description:

R Tree:

An R-tree is a hierarchical data structure used for indexing spatial data such as points, rectangles, and polygons. It organizes objects using Minimum Bounding Rectangles (MBRs) to enable efficient searching for nearest neighbors, conducting range queries, and exploring spatial relationships.

Insertion Algorithm:

The insertion operation starts by identifying the appropriate leaf node for placing the new point or object. This choice is made by selecting the subtree whose Minimum Bounding Rectangle (MBR) requires the least enlargement to accommodate the latest entry. If the selected node exceeds its maximum capacity after the insertion, a node split occurs, utilizing a heuristic strategy (such as linear or quadratic split). Subsequently, the MBRs are adjusted recursively up the tree to maintain the overall structure.

Step-by-Step Process of Insertion:

Here is the step-by-step insertion process used in your R-Tree implementation, written in the requested format:

1. Start at the root node.
2. If the root is a leaf node:
    a. Add the point to the node's list of children.
    b. If the number of children exceeds the node's capacity (B value), perform a split:
        i. Sort the children by coordinate (x or y).
        ii. For all legal split positions, divide the children into two groups.
        iii. Compute the perimeter of each resulting node's MBR.
        iv. Select the split with the minimum total perimeter.
        v. Create two new leaf nodes and insert the groups.
        vi. If the original node was the root, create a new root to hold the two new nodes.
        vii. Otherwise, insert the two new nodes into the parent.
3. If the root is a branch node:
    a. Expand the current node’s bounding box to include the new point.
    b. Choose a child node to recurse into:
        i. Prefer the child whose center is closest to the new point.
        ii. If two children are equally close, prefer the one with fewer children.
    c. Recursively apply the insertion process to the selected child.
    d. If the selected child node exceeds its capacity after insertion:
        i. Perform a split using the same perimeter-based heuristic as in the leaf case.
        ii. Insert the resulting nodes into the current node.
        iii. If the current node exceeds its capacity, repeat the split process up the tree.
4. The insertion ends when no further splits are required, or the root is adjusted to accommodate new child nodes.

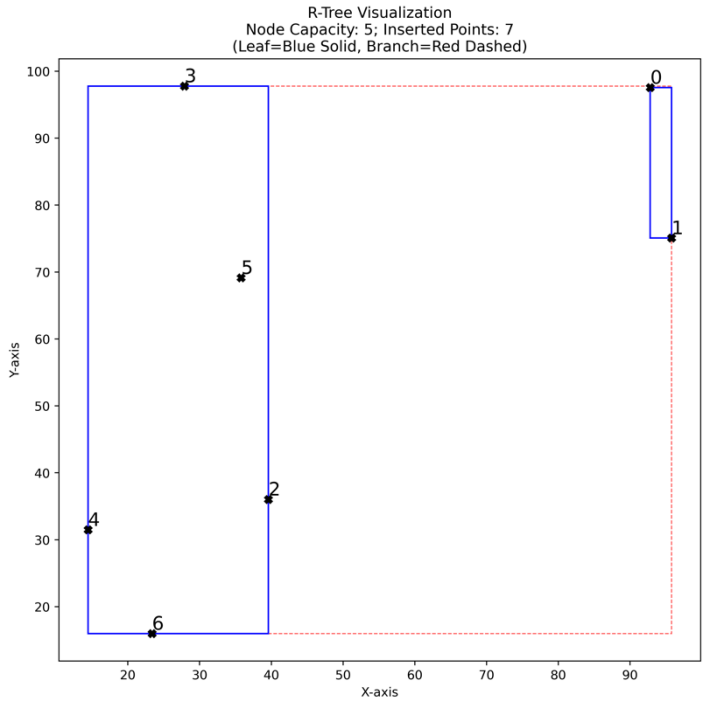

Figure 1: R-Tree Snapshot after inserting 7 points. Node Capacity: 5

In the above figure 1, we can node splitting in action. The left node (solid blue) has reached its max capacity and when 2 new nodes are added the node is split and the points are grouped to two nodes. The red dashed rectangle represents the branch node which contain leaf nodes

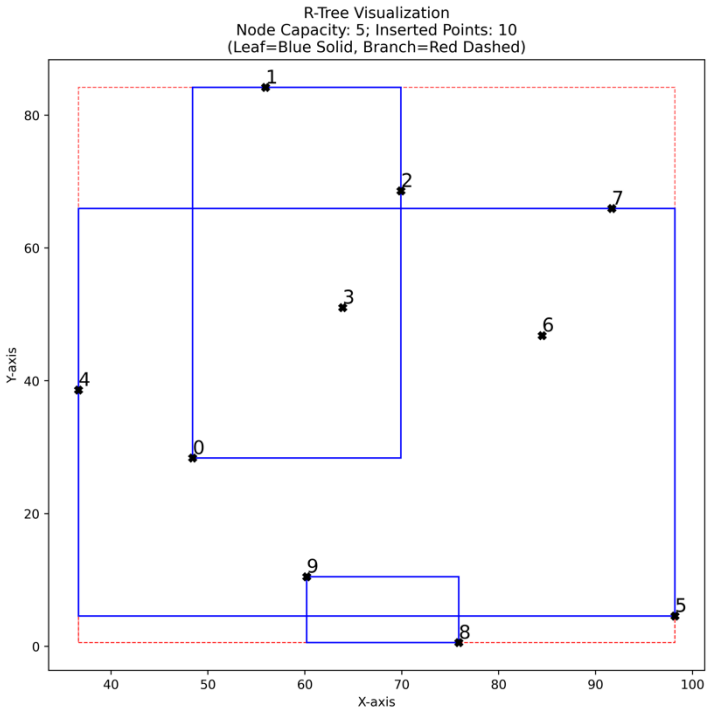

Figure 2: R-Tree Snapshot after inserting 10 points. Node Capacity: 5

In the above figure 2, we can node splitting and node overlap in action. When 10 points are inserted to the R-Tree we can see the leaf-node overlap. The splits are made on the basis of Perimeter Minimization Heuristic. This results in tight MBRs

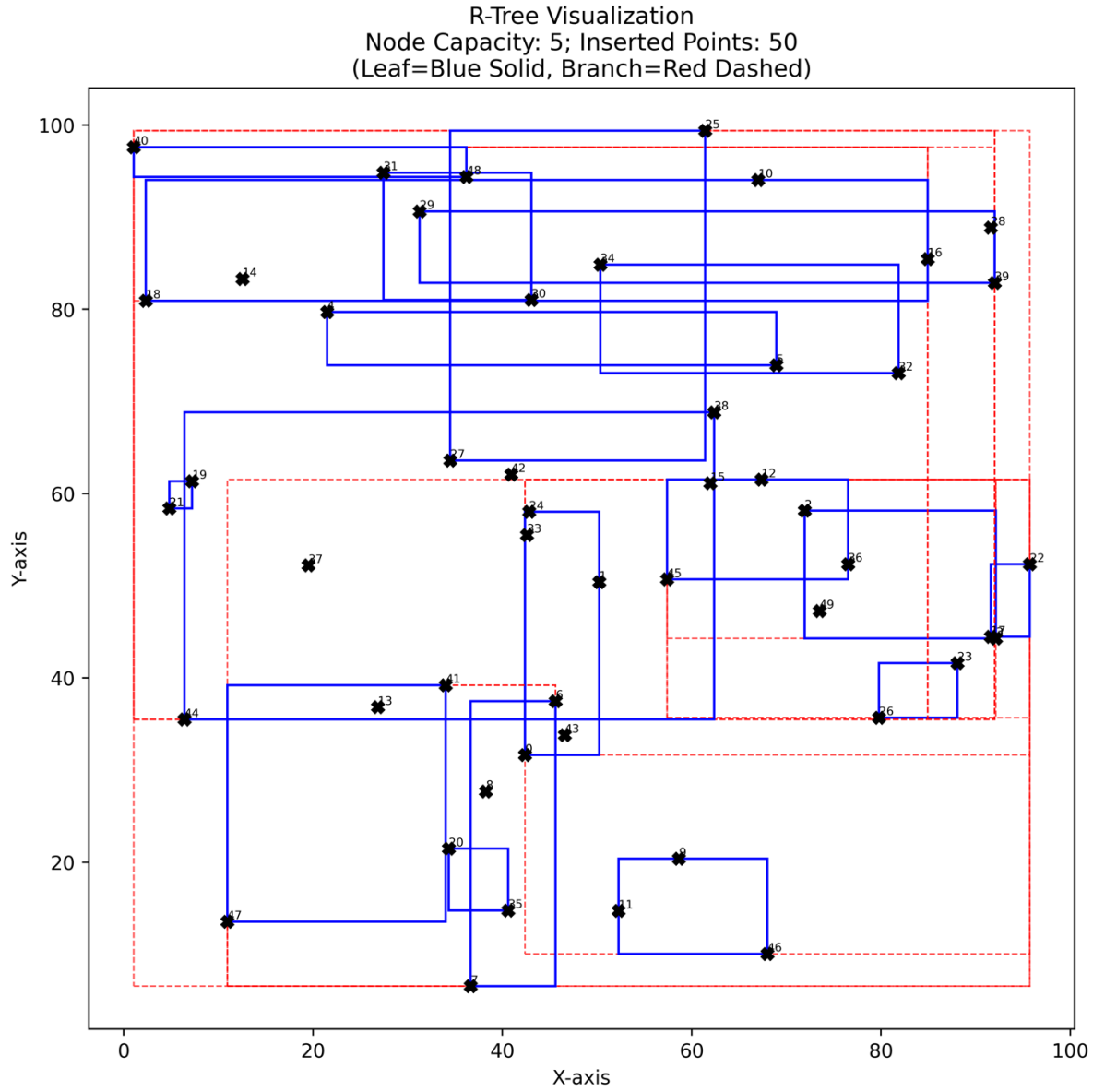

Figure 3: R-Tree Snapshot after inserting 50 points. Node Capacity: 5

In the above figure 3, we inserted 50 points into R-Tree with max node capacity of 5. We can see the multiple leaf nodes (solid blue line) and internal branch nodes (dashed red line). The internal branch nodes can contain a max of 5 leaf nodes and if the overflow occurs, they split into two branches and share the leaf nodes. The overflow check is propagated to root node to ensure R-Tree structure. The root node is the largest dashed rectangle which contains all the other nodes.

Point Search Algorithm:

The point search operation traverses the R-Tree from the root to the leaves, checking whether the target point lies within any child MBRs at each level. If a match is found at a leaf node, the point is returned. By default, R-Trees are intended for rectangular data. Therefore, points are represented with a degenerate rectangle with length and width set to very small constant like 10^{-6}

Step-by-Step Process of Point Search:

1. Start at the root node.
2. If the node is a leaf:
    a. Iterate through each point in the node.
    b. For each point, check if its coordinates exactly match the query point's coordinates.
    c. If a match is found, return the point.
    d. If no match is found in this leaf, return `None`.
3. If the node is a branch:
    a. For each child node:
        i. Check if the child's bounding rectangle contains the query point.
        ii. If it does, recursively apply the point search on that child.
        iii. If a match is found in any child, return the point.
4. If no matching point is found in any relevant subtree, return `None`.

Range Search Algorithm:

The range search with the bounding box search algorithm follows a similar traversal pattern but returns all entries that intersect with a specified rectangular area. The R-Tree enhances efficiency by pruning irrelevant branches and skipping subtrees whose Minimum Bounding Rectangles (MBRs) do not overlap with the query box. This approach is significantly more efficient than linear scan methods.

Step-by-Step process of Range Search:

1. Start at the root node.
2. If the node is a leaf:
    a. Initialize an empty list to hold results.
    b. For each point in the node:
        i. Check if the point lies inside the given bounding box.
        ii. If it does, add the point to the result list.
    c. Return the result list.
3. If the node is a branch:
    a. Initialize an empty list to hold results.
    b. For each child node:
        i. Check if the child's bounding rectangle intersects with the bounding box.

ii. If it does, recursively apply the range search on that child.
iii. Append the returned results to the current result list.
c. Return the result list.
4. If no children intersect the bounding box, return an empty list.

Nearest Neighbor Algorithm:

The nearest neighbor search algorithm explores the tree using a best-first traversal strategy. It maintains a priority queue ordered by the minimum distance from the query point to each MBR, prioritizing the expansion of the most promising branches. The search continues until the closest actual object is found and all other branches with greater distances are pruned.

Step-by-Step process of Nearest Neighbor:

1. Start at the root node.
2. If the current node is a leaf node:
   a. Calculate the Euclidean distance from the query point to each point in the node.
   b. Return the point with the smallest distance (or top-k closest if performing k-NN).
3. If the current node is an internal (branch) node:
   a. For each child node:
      i. Compute the distance from the query point to the **center of the child's MBR (Minimum Bounding Rectangle)**.
   b. Choose the child with the smallest center distance.
   c. Recursively perform nearest neighbor search on the selected child.
4. Stop when a leaf node is reached and nearest point(s) found.

Time and Space Complexity Analysis:

The time complexity of R-Tree 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 tree is well-balanced and MBR overlaps are minimal, and insertion proceeds down a single path with efficient node splits.
- **Worst Case:** The worst-case time complexity for insertion is O(n), as overlapping MBRs may cause traversal through multiple branches and excessive node splits, especially in highly clustered or unbalanced datasets.
- **Space Complexity:** The space complexity for insertion is O(n), as each spatial object is stored once, along with its associated MBRs at internal nodes.

Point Search Complexity:

- **Best Case:** The best-case time complexity for point search is O(log n), as it is well-structured trees with minimal MBR overlap and the algorithm quickly descends to the correct leaf node containing the point.
- **Worst Case:** The worst-case time complexity for point search is O(n), as multiple subtrees might need to be explored due to its overlapping MBRs.
- **Space Complexity:** The space complexity for point search is O(1), as it uses constant extra space for traversal pointers or a small stack in a recursive implementation.

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 when MBRs are well-separated and few entries fall within the query range, irrelevant subtrees are pruned efficiently.
- **Worst Case:** The worst-case time complexity for bounding box search is O(n), as the query box overlaps many MBRs or if the tree is poorly clustered and most or all nodes may need to be visited.
- **Space Complexity:** The space complexity for bounding box search is O(k), where k is the number of results returned, and additional space is needed to store results temporarily.

Nearest Neighbor Search Complexity:

- **Best Case:** The best-case time complexity for nearest neighbor search is O(log n), as in a well-organized R-Tree and evenly distributed points, the nearest neighbor is found 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(h), where h is the height of the tree.

Performance Analysis:

Insertion Algorithm:

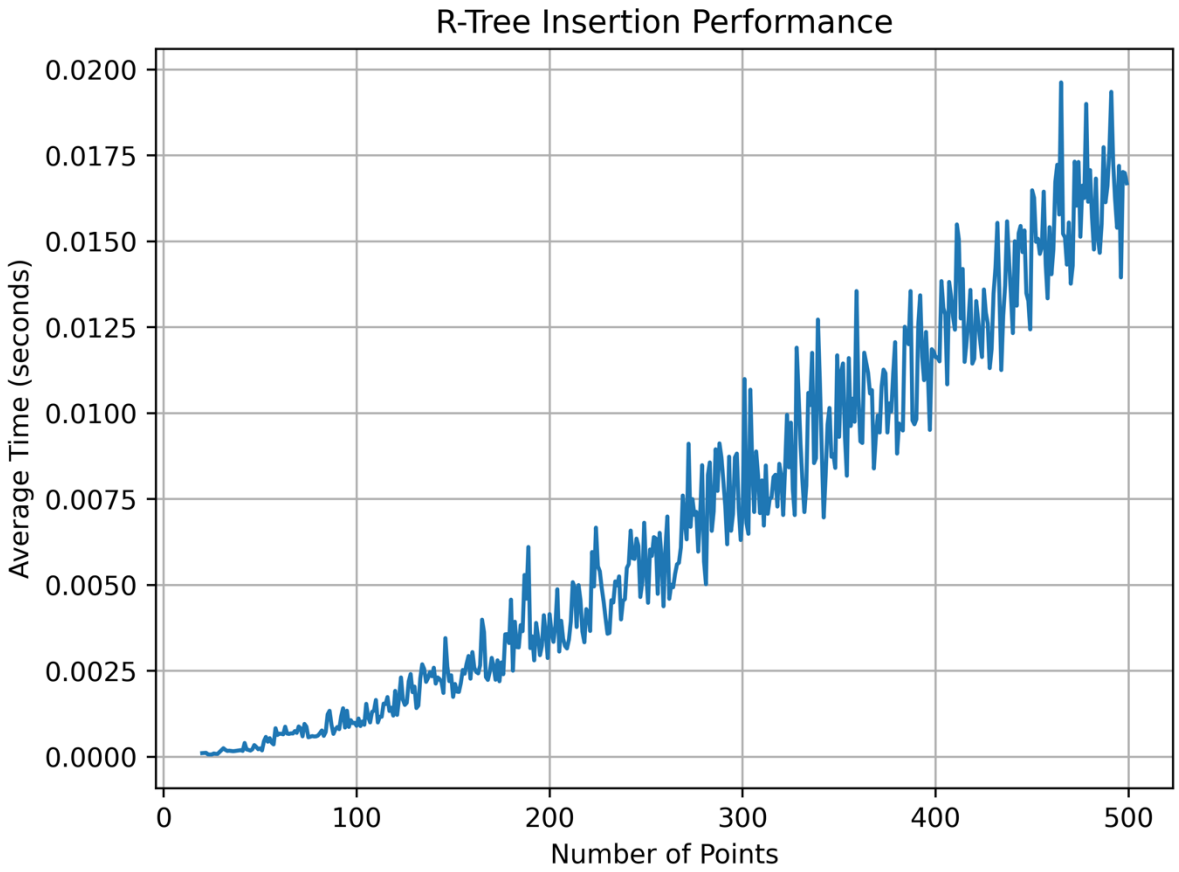

Figure 4: R-Tree Insertion Performance

The graph presented shows the performance of R-Tree 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 average insertion time in seconds. As the number of points increases, the graph demonstrates a general upward trend in insertion time, indicating that the R-Tree becomes progressively slower as more points are added. This trend is expected since the insertion time complexity for a balanced R-Tree is approximately O(log n), where n is the number of points. Fluctuations and occasional spikes in insertion time, particularly in areas with steep climbs, can be attributed to node overflows and the resulting splits that occur when a node surpasses its maximum capacity. These incidents introduce additional overhead in maintaining the tree structure and updating the minimum bounding rectangles (MBRs) across the affected levels. Overall, the graph reflects the typical performance behavior of R-Trees under dynamic insertion workloads and highlights their efficiency in managing spatial data, especially when balanced effectively.

R-Tree Insertion vs Varying Capacity:

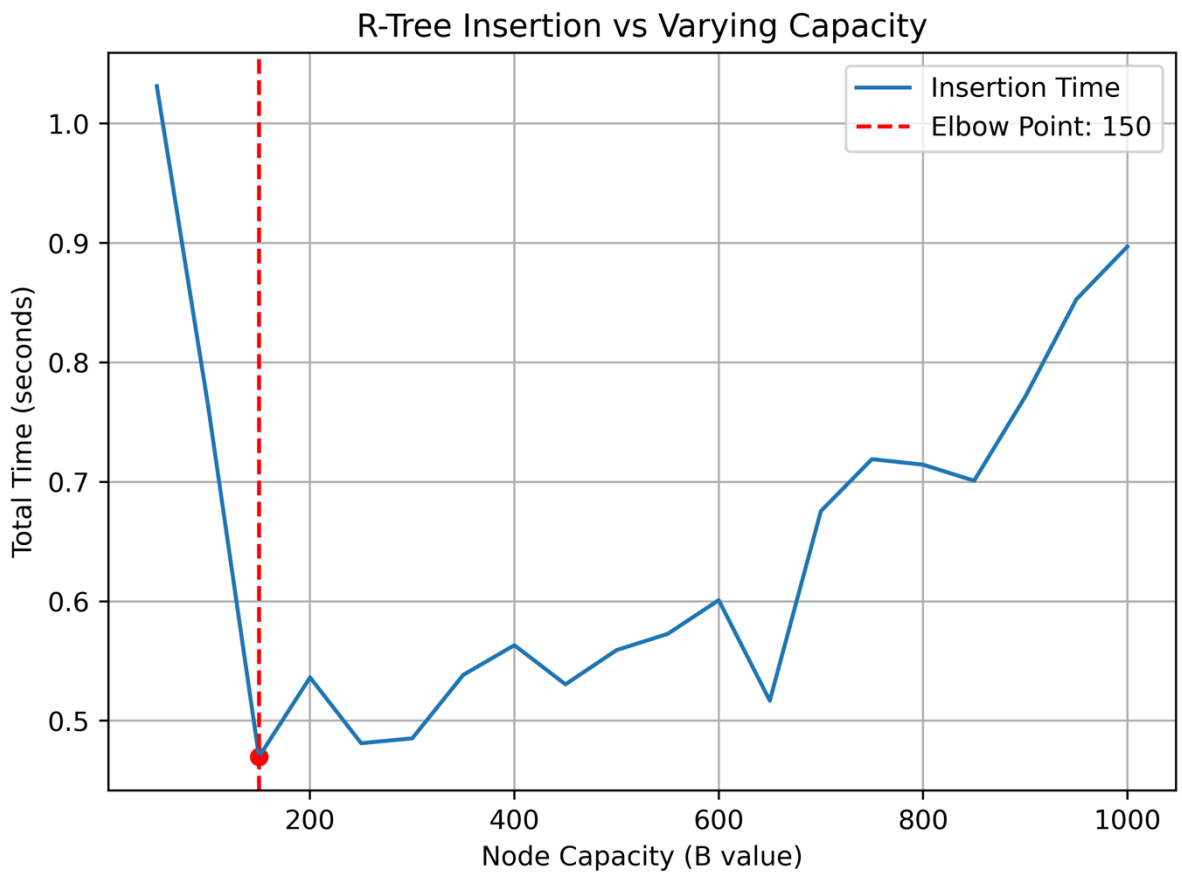

Figure 5: R-Tree Insertion vs Varying Capacity

The graph presented shows the performance of R-Tree insertion as a function of the varying node capacity (B value). The x-axis represents the node capacity, which varies from 0 to 1000, 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 **150**. 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 150. This rapid decline is due to a reduction in the frequency of node splits. When the node capacity is small, the R-tree 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 150), the insertion time increases gradually. This indicates that further increasing the node capacity does not provide significant performance improvements after reaching the optimal capacity. This can be due to increased heuristic computation and internal sorting of leaf nodes. This 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:

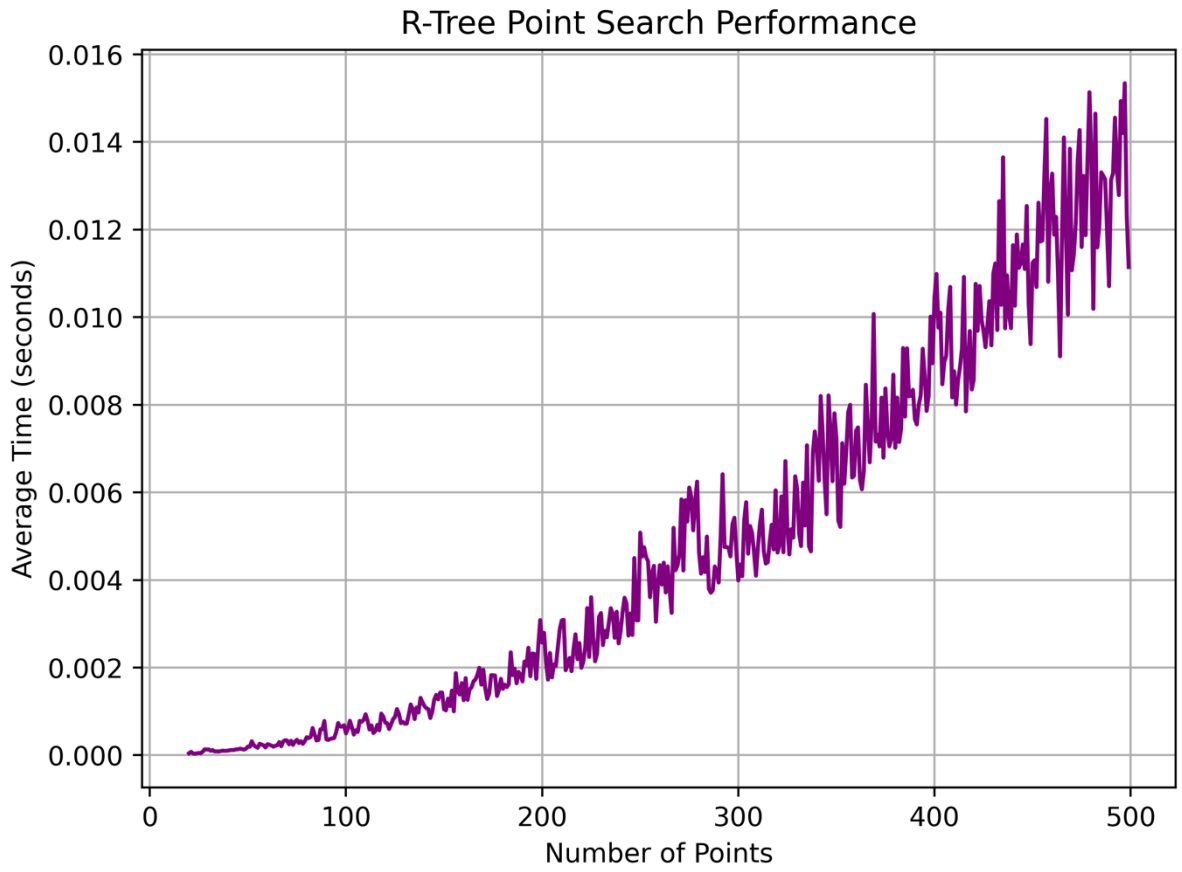

Figure 6: R-Tree Point Search Performance

The graph represents the performance of point search in a R-Tree 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 average 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 R-Tree takes longer to locate a specific point as the dataset grows. This trend aligns with the expected performance of a well-balanced R-Tree, where the search time complexity is approximately O(log n).

The nature of R-trees utilizes Minimum Bounding Rectangles (MBRs), which can lead to overlapping regions and multiple branches being explored during searches. This overlap contributes to the fluctuations and small spikes in the graph, particularly when the number of points exceeds 300. Despite these variations, the overall performance remains scalable and efficient. The R-Tree maintains acceptable query speeds even as the volume of data increases.

Bounding Box Algorithm:

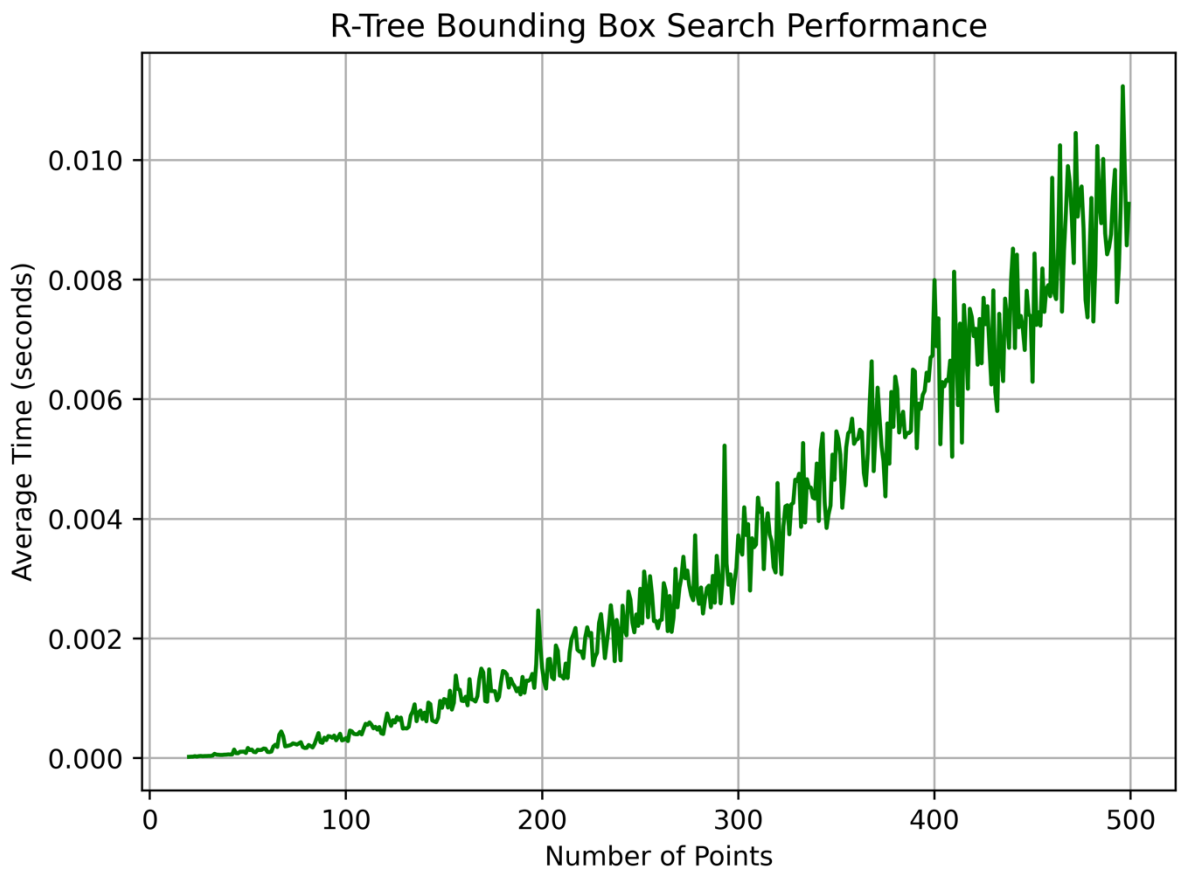

Figure 7: R-Tree Bounding Box Search Performance

The graph presented shows the performance of bounding box search in an R-Tree 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 average time per query in seconds. The graph exhibits a steady upward trend in query time as the number of points grows, indicating that the bounding box search becomes gradually more expensive as the dataset grows. This behavior aligns with the expected performance characteristics of an R-Tree, 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. As the tree expands and more points are added, the number of nodes and overlapping Minimum Bounding Rectangles (MBRs) increases.

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

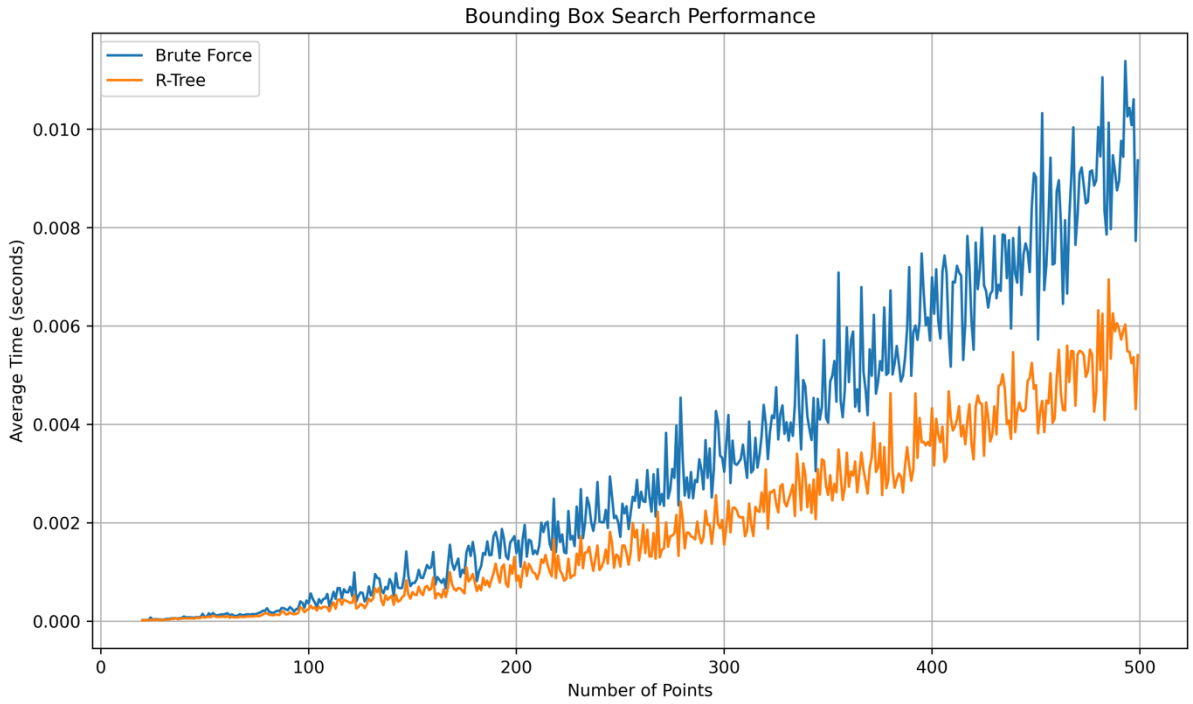

Figure 8: R-Tree vs Brute Force Bounding Box Search Performance

The graph presented compares the performance of the R-Tree and Brute Force methods for bounding box searches as the number of points increases from 0 to 500. The x-axis represents the number of points, while the y-axis shows the average time per query in seconds. The orange line represents the performance of the R-Tree method, while the blue line represents the performance of the Brute Force method.

The graph demonstrates that the R-Tree method significantly outperforms the Brute Force approach across all point counts. The search time for the R-Tree method shows a gradual upward trend with moderate fluctuations, likely due to overlapping bounding rectangles or variations in local data density. In contrast, the Brute Force method exhibits a steep, linear increase in query time, which aligns with its O(n) time complexity for each search operation.

The performance gap between the two methods becomes increasingly pronounced as the number of points grows, with the Brute Force method becoming disproportionately slower than the R-tree. This difference arises because the R-Tree effectively prunes the search space by eliminating branches whose Minimum Bounding Rectangles (MBRs) do not intersect with the query region, thus reducing the number of required comparisons.

As the dataset size increases, the advantages of using the R-tree become even clearer. Its ability to maintain hierarchical spatial indexing allows for faster and more scalable performance, making the R-tree a preferred choice for spatial queries in Geographic Information Systems (GIS) applications that frequently involve bounding box searches or handling large-scale datasets.

Nearest Neighbor Algorithm:

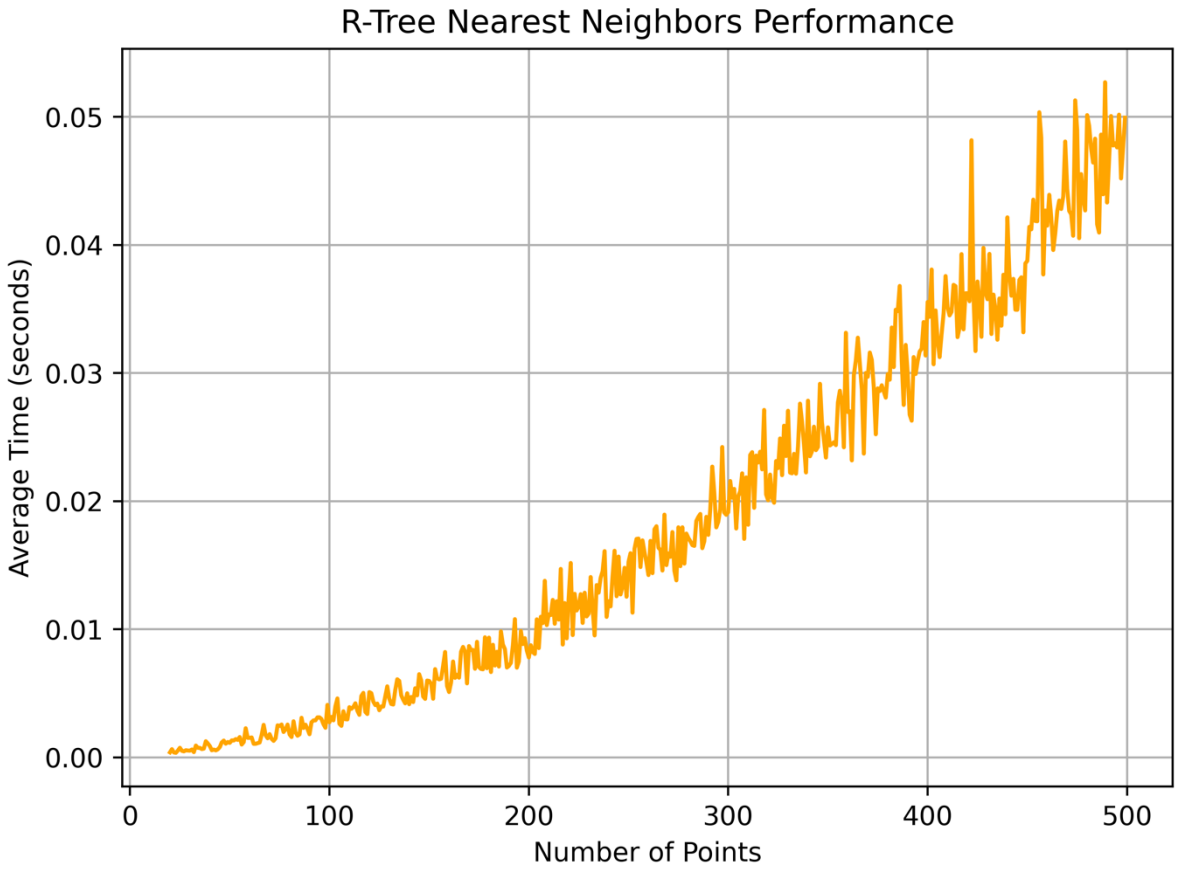

Figure 9: R-Tree Nearest Neighbors Performance

The graph presented shows the performance of the nearest neighbors search operation in a R-Tree 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 average search time in seconds. As the number of points increases, the graph demonstrates a gradual upward trend in query time, indicating that the R-Tree becomes slightly slower as the dataset grows. This behavior is expected since nearest neighbor search in R-Trees generally follows an average-case complexity of O(log n). Fluctuations in the graph may result from overlapping bounding rectangles or clustered regions, which necessitate a deeper traversal. Overall, the R-Tree maintains efficient performance and is well-suited for nearest-neighbor queries in GIS applications.

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

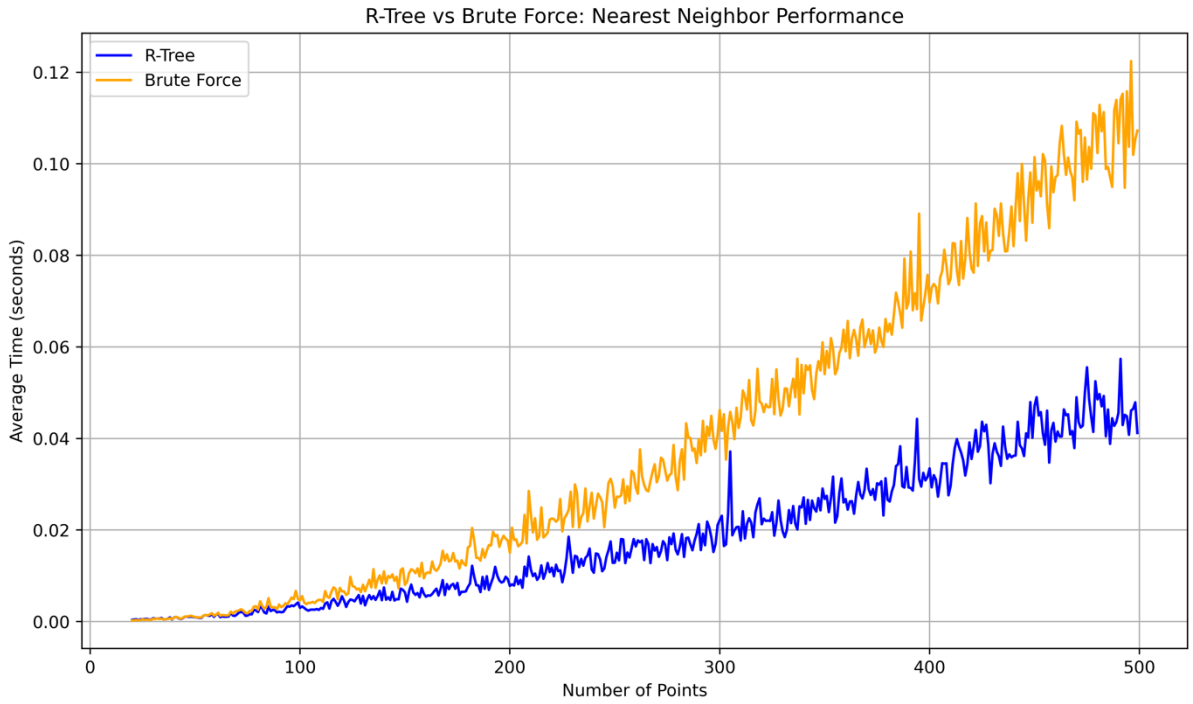

Figure 10: R-Tree vs Brute Force Nearest Neighbors Performance

The graph presented compares the performance of the R-Tree 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 average search time in seconds. The blue line indicates the performance of the R-Tree method, while the orange line represents the Brute Force approach.

The graph shows that the R-Tree method consistently outperforms the Brute Force approach across all point counts. The search time for the R-Tree increases gradually, showing a sublinear growth pattern due to its spatial indexing and efficient pruning strategy. 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 R-Tree 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 R-Tree approach. The significant performance gap highlights the efficiency of the R-Tree structure in minimizing query time by organizing spatial data into hierarchical Minimum Bounding Rectangles (MBRs) that reduce the number of branches visited during search. 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 demonstrates the efficiency of the R-Tree approach for nearest neighbor search compared to the Brute Force method. The R-Tree's ability to efficiently index and traverse spatial data makes it a more suitable and suitable choice for large-scale Geographic Information Systems (GIS) applications, where fast proximity queries are essential.

Real World Analysis:

Parks in San Antonio:

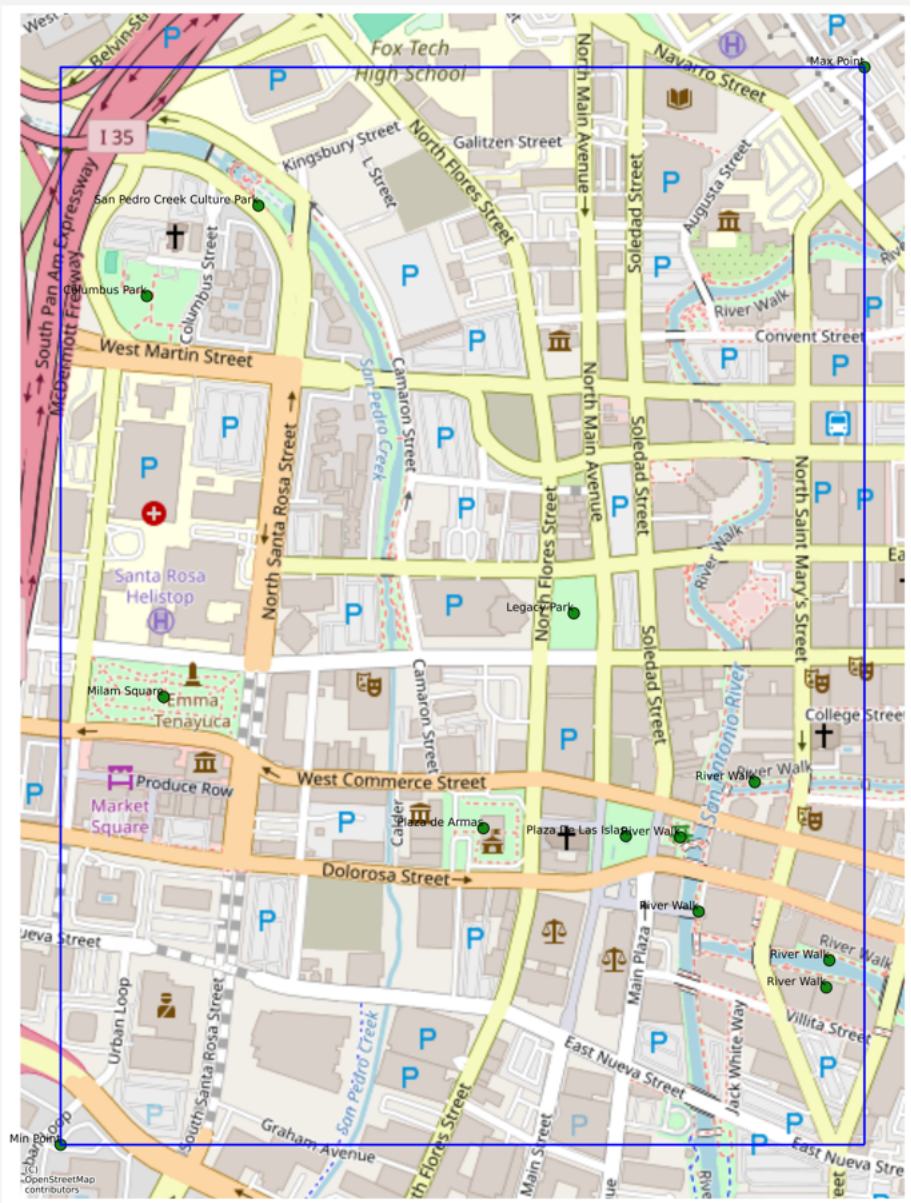

Figure 11: Locate Park using Bounding Box Algorithm (R-Tree)

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 R-Tree data structure, the algorithm efficiently locates all parks within the defined spatial region, significantly reducing computational overhead compared to brute-force methods. This visualization demonstrates the effectiveness of the bounding box search in detecting nearby parks using spatial indexing techniques in Geographic Information Systems (GIS). Notable parks within the bounding box include Legacy Park and several points along the River Walk.

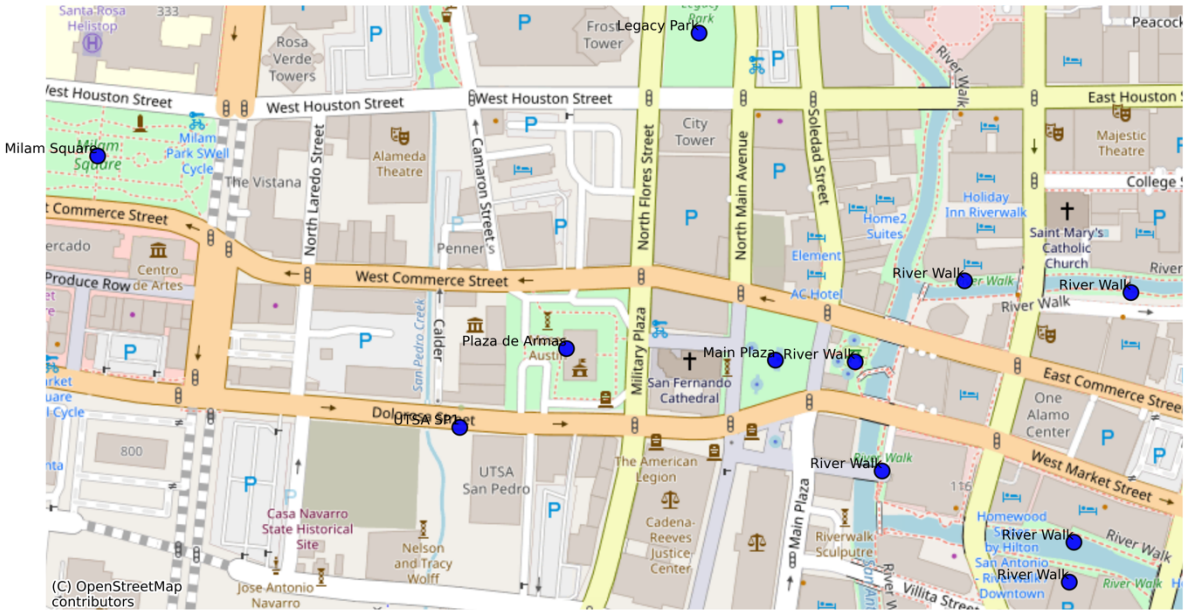

Figure 12: Locate Park using Nearest Neighbor Algorithm (R-Tree)

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 R-Tree 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:

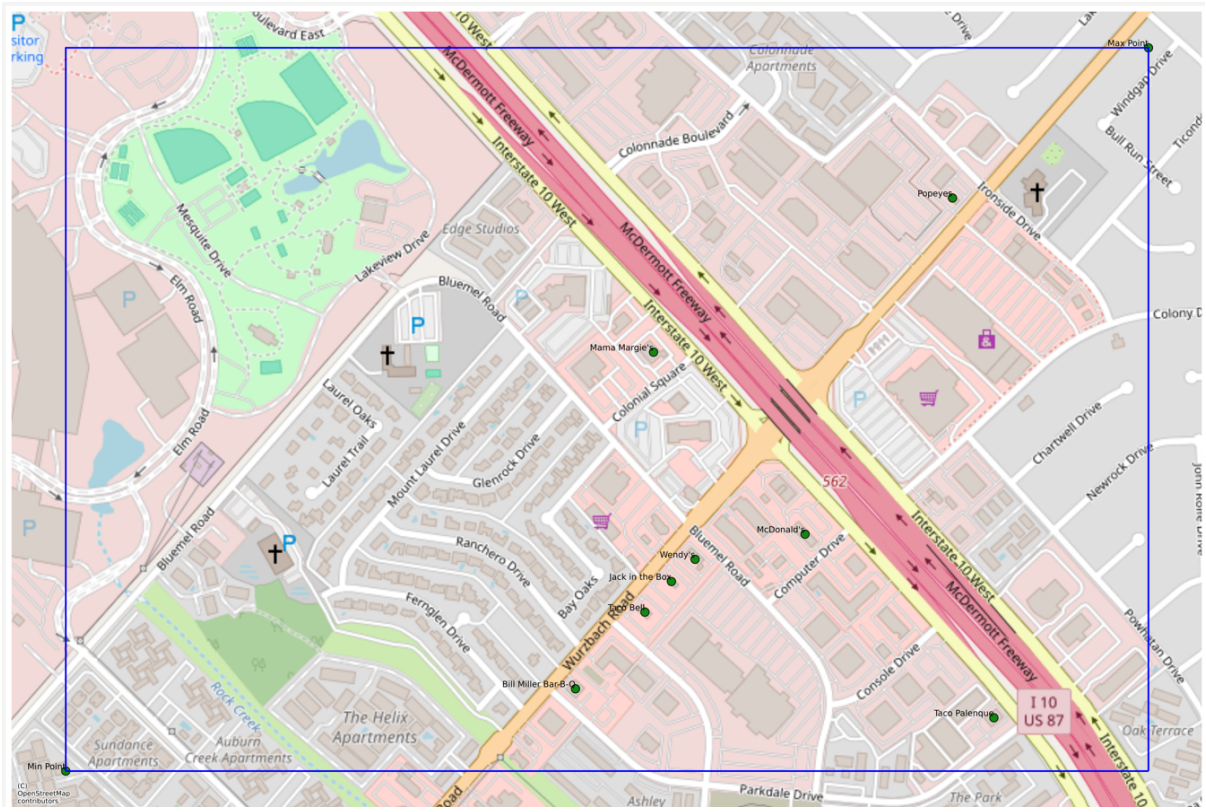

Figure 13: Locate Fast Food Chain using Bounding Box Algorithm (R-Tree)

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 R-Tree data structure, the algorithm efficiently locates all fast-food chain within the defined spatial region, significantly reducing computational overhead compared to brute-force methods.

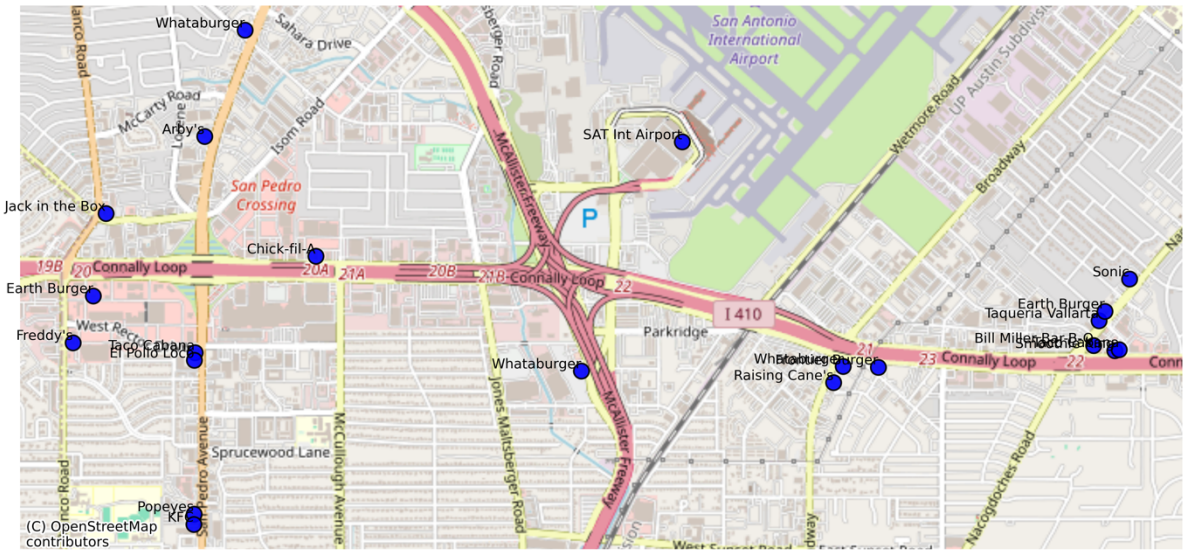

Figure 14: Locate Fast Food Chain using Nearest Neighbor Algorithm (R-Tree)

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 R-Tree 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] Guttman, A. (1984). R-Trees. ACM SIGMOD Record, 14(2), 47–57. https://doi.org/10.1145/971697.602266
[3] Samet, H., Rosenfeld, A., Shaffer, C. A., & Webber, R. E. (1984). A geographic information system using quadtrees. Pattern Recognition, 17(6), 647–656. https://doi.org/10.1016/0031-3203(84)90018-9
[4] Shekhar, S., Xiong, H., & Zhou, X. (2017). Encyclopedia of GIS. In Springer eBooks. https://doi.org/10.1007/978-3-319-17885-1