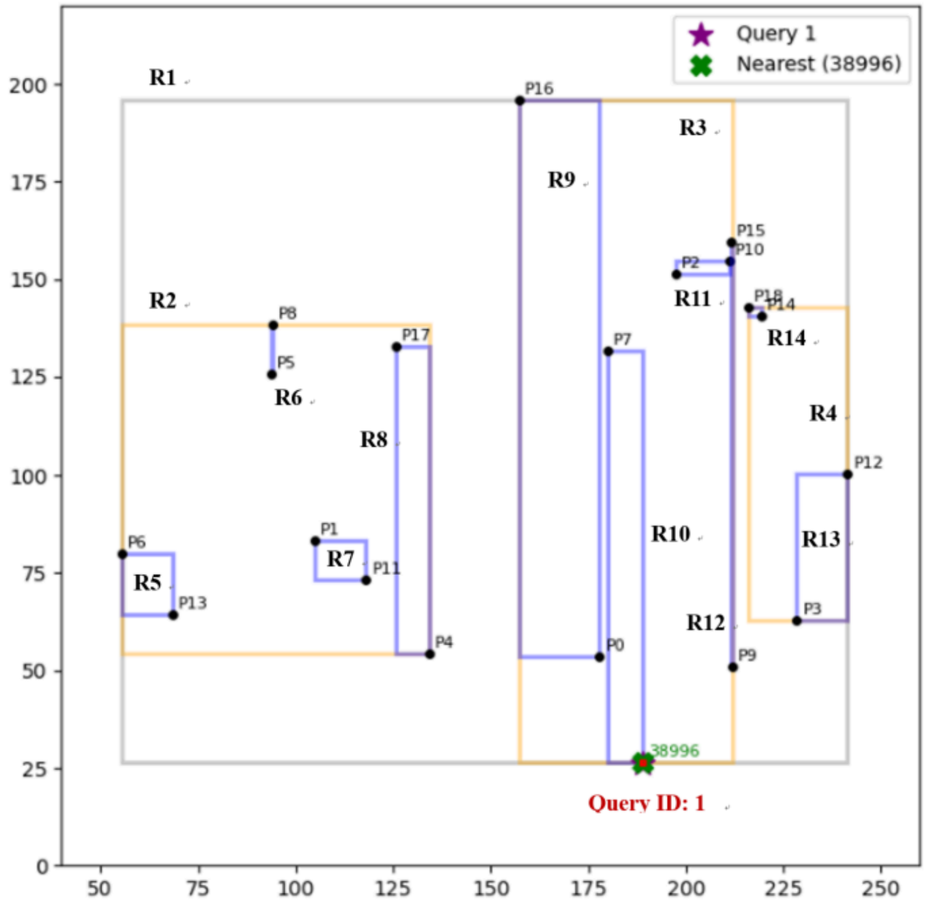
BaB (Nearest Neighbor)

```python
# Create a function implementing the Branch-and-Bound nearest neighbor search on an R-tree
def branch_and_bound(rtree, user_location):
    # Make a list of candidate nodes to explore
    # Start from from the root node
    # Since the root node covers the entire space, the distance is zero
    node_list_explore = [(0, rtree.root)]
    # Initialize variables to store closest point and its distance
    closest_point = None
    shortest_dist = float('inf')
    # Continue exploring candidate nodes while there are still nodes to examine
    while node_list_explore:
        # Reset per-iteration closest candidate
        current_minimum_dist = float('inf')
        current_index = -1
        # Search the candidate node with the smallest distance bound
        for index, (dist, node) in enumerate(node_list_explore):
            if dist < current_minimum_dist:
                current_minimum_dist = dist
                current_index = index
        # Take out the nearest node efficiently (swap with last node, then remove)
        # For example, to remove node B: (A, B, C, D) -> (A, D, C, D) -> (A, D, C)
        node_dist, node = node_list_explore[current_index]
        node_list_explore[current_index] = node_list_explore[-1]
        node_list_explore.pop()

        # After a nearest point is found, skip nodes whose MBR distance is greater than the current shortest distance
        if closest_point is not None and node_dist > shortest_dist:
            continue
```

```python
        # if the node is a leaf, it contains actual data points (not child MBRs)
        if node.is_leaf():
            for point in node.data_points:
                # Compute distance from the query point to each data point
                point_dist = euc_distance(point, user_location)
                # If this point is closer than the current closest, update it
                if point_dist < shortest_dist:
                    shortest_dist = point_dist
                    closest_point = point
        else:
            # If it is not a leaf node, examine its child MBRs
            for child in node.child_nodes:
                # Compute the minimum possible distance from the query point to this child's MBR
                minimum_dist = min_dist_to_mbr(child, user_location)
                # Explore this child only if its MBR could contain a closer point
                if minimum_dist <= shortest_dist:
                    node_list_explore.append((minimum_dist, child))

    # Return the nearest point and its distance
    return closest_point, shortest_dist
```

BaB method starts at the R-tree root and explores the closest MBR first. Then, it prunes any node farther than the current closest distance. If it's a leaf, it checks actual points; otherwise, expand to child MBRs. It repeats the process until finding the nearest neighbor.

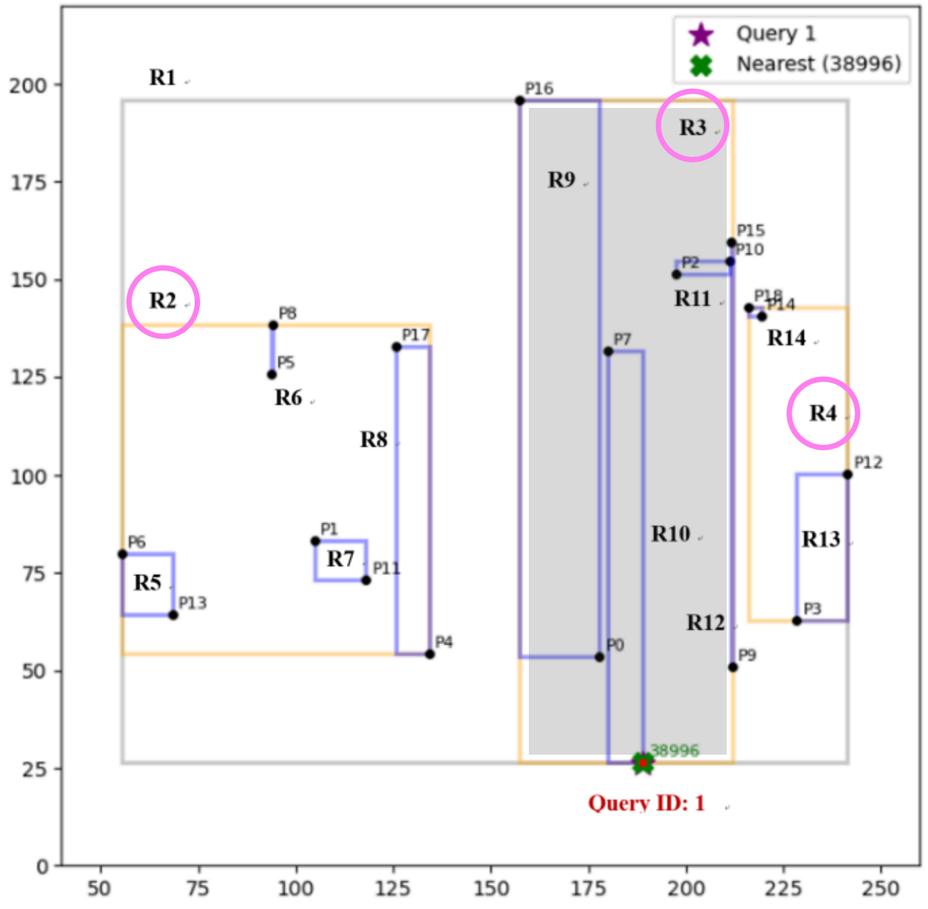
BaB (Nearest Neighbor)



BaB Process

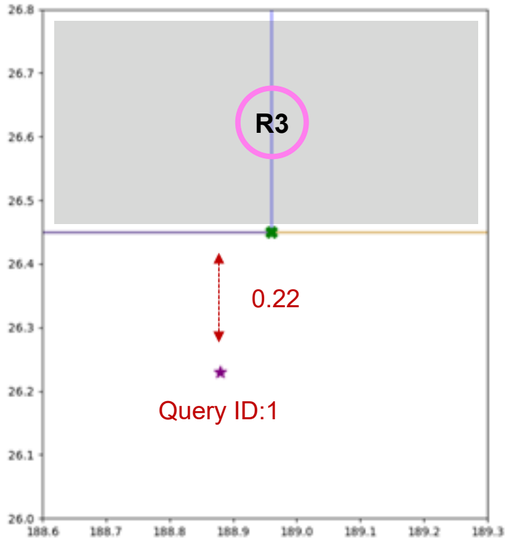
This sample R-tree diagram (*R = 4) is created to explain how to find the closest parking point (point ID: 38996) for the current query point (query ID: 1). The other 19 points were randomly selected from the dataset.

BaB (Nearest Neighbor)



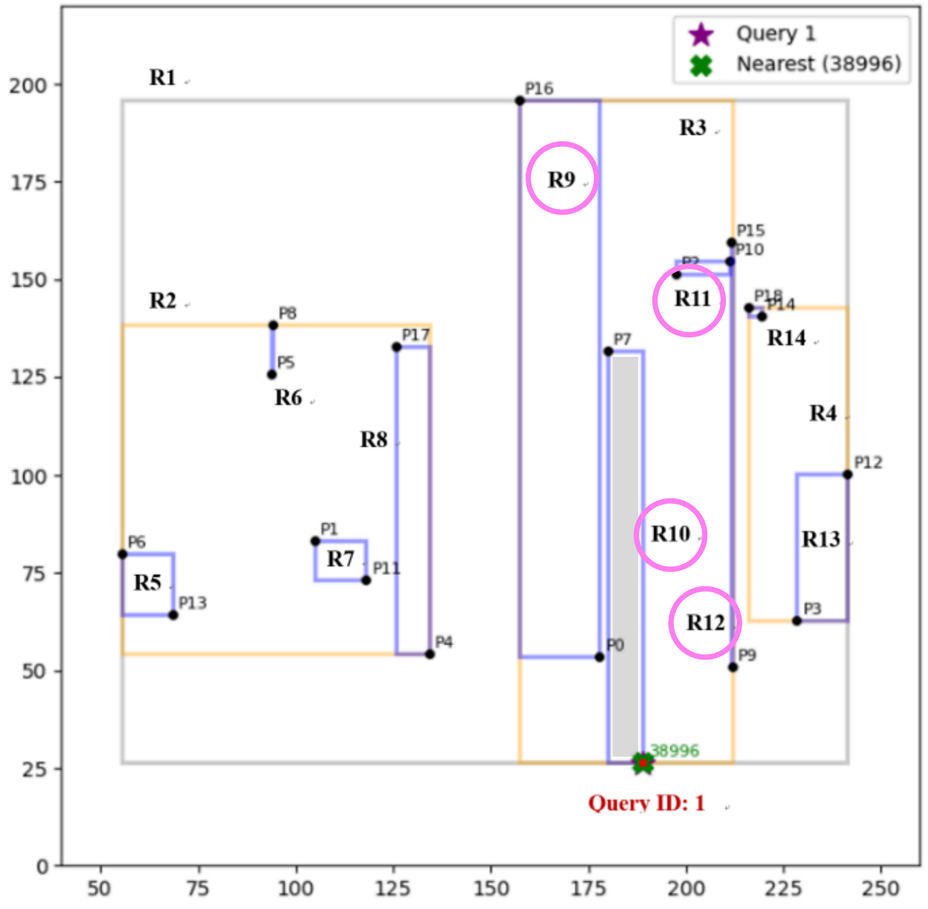
BaB Process

MBR No	minDIst
R2	61.38
R3	0.22
R4	45.65



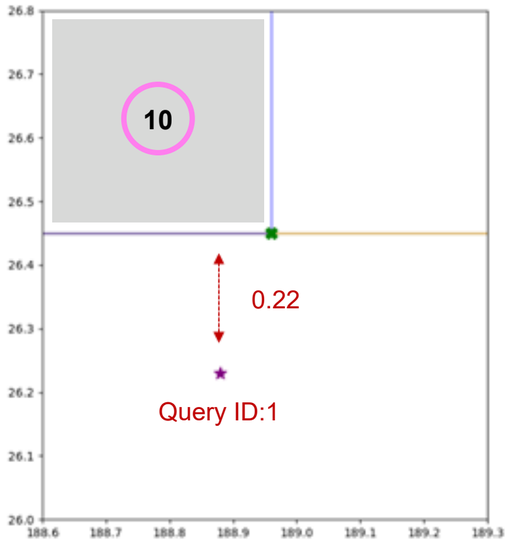
Among the yellow MBRs, **R3** has the smallest minDist to the query point. Therefore, R3 is explored first.

BaB (Nearest Neighbor)



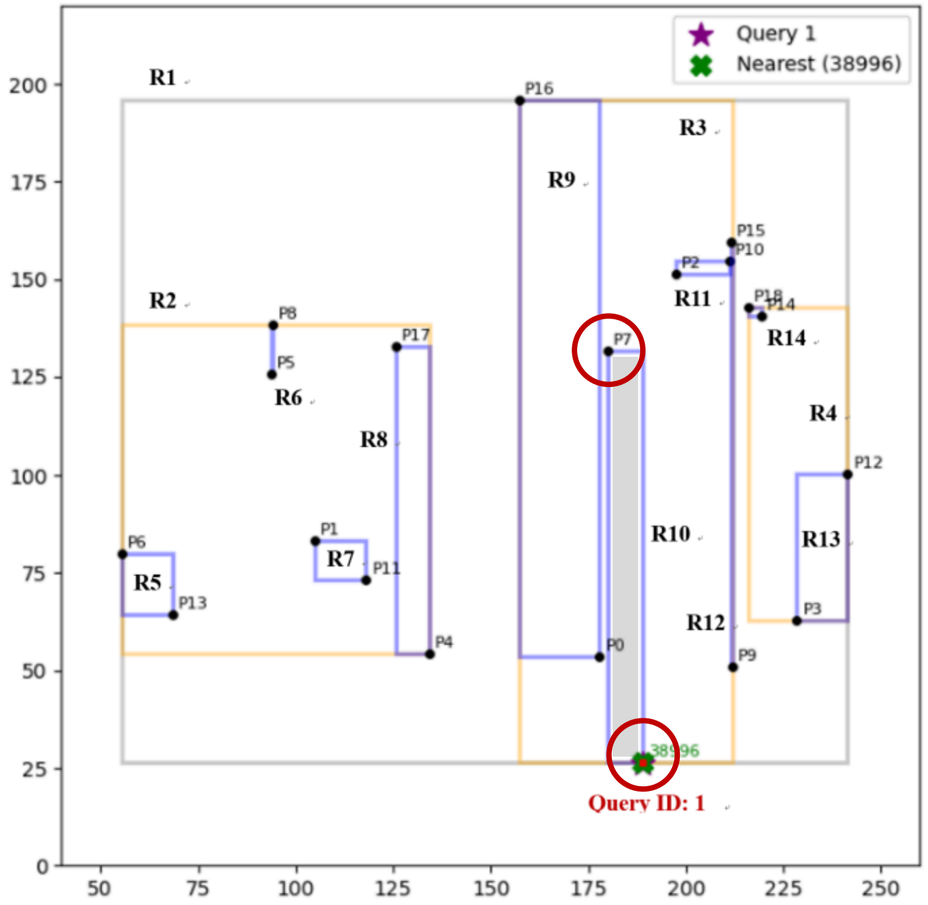
BaB Process

MBR No	minDist
R9	29.64
R10	0.22
R11	125.56
R12	33.48



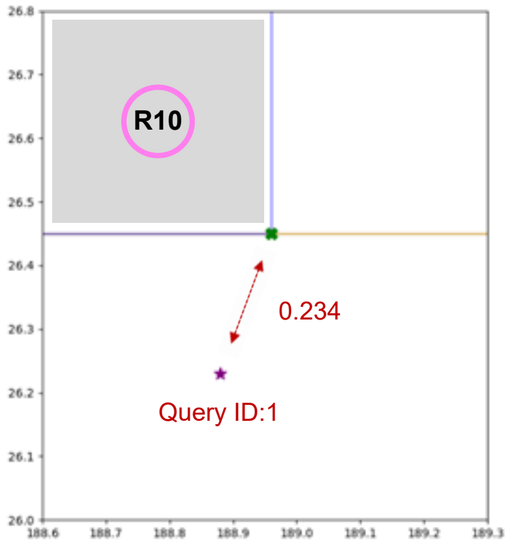
Since the selected node is not a leaf, the algorithm explores its internal MBRs. Among the blue MBRs, **R10** has the smallest minDist to the query point, so it is explored first.

BaB (Nearest Neighbor)



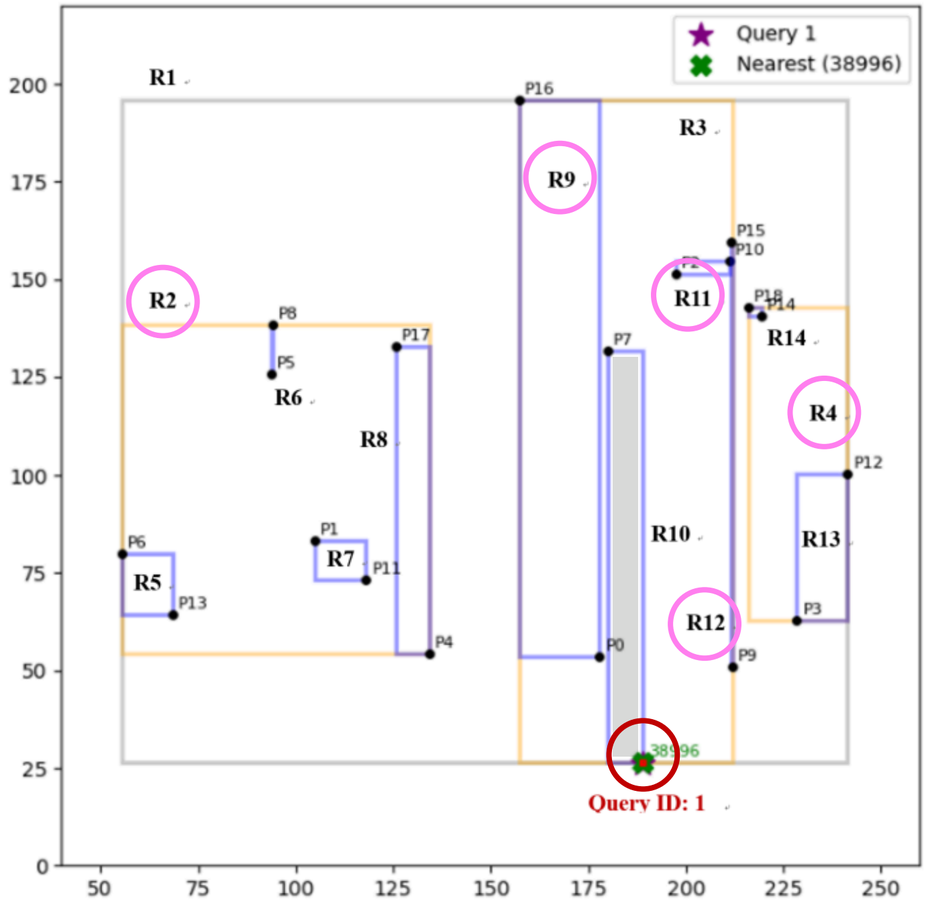
BaB Process

Point No	Distance
P7	105.89
P38996	0.234



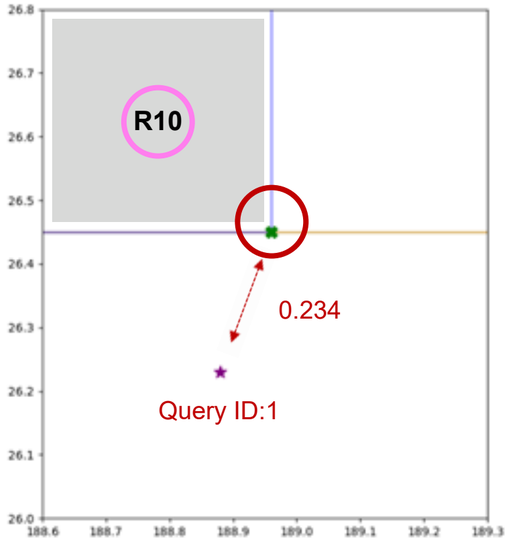
Inside **R10**, there are two parking points, P7 and P38996.
Since **P38996** is closer to the query point than P7, P38996 becomes the current nearest neighbor.

BaB (Nearest Neighbor)



BaB Process

MBR No	minDist
R2	61.38
R4	45.65
R9	29.64
R11	125.56
R12	33.48



No other MBR has a smaller minDist than the current shortest distance (0.234), so the remaining MBRs are pruned. Therefore, the nearest parking point is **P38996** (distance = 0.234).

*Child nodes of R2 and R4 are not considered, since their minDist cannot be smaller than the minDist of R2 and R4, respectively.

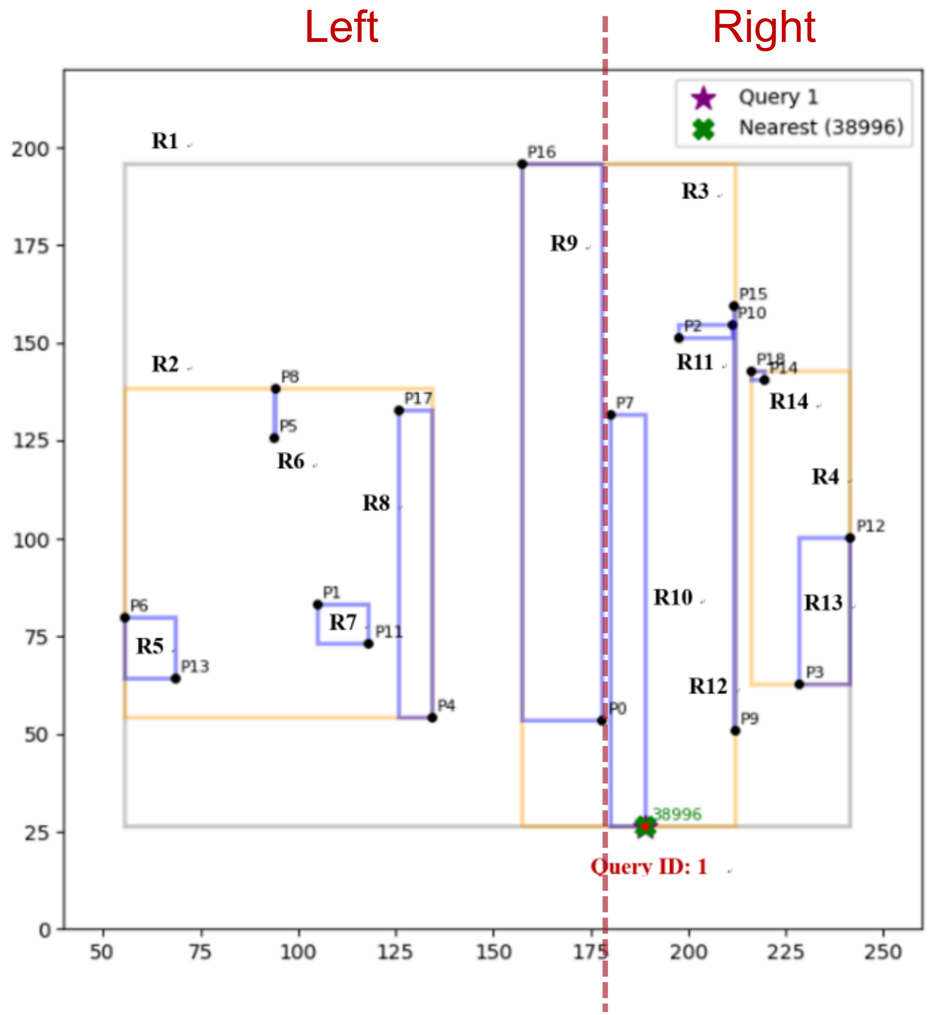
Divide-and-Conquer (Nearest Neighbor)

```python
def run_BaB_DaC():
    parking_points = read_data("parking.txt")
    user_points = read_data("query_points.txt")

    # 1. Sort parking points based on x-coordinate and split them into two halves
    parking_points.sort(key = lambda p: p[0])
    # Compute the median x-value based on (p[0] = x-value)
    mid = len(parking_points) // 2
    # Define points in left half
    left_points = parking_points[:mid]
    # Define points in right half
    right_points = parking_points[mid:]

    # 2. Build an Rtree for each partition
    rtree_left = RTree()
    for x, y, _id in left_points:
        rtree_left.insert(rtree_left.root, (x, y, _id))

    rtree_right = RTree()
    for x, y, _id in right_points:
        rtree_right.insert(rtree_right.root, (x, y, _id))

    # 3. Find the nearest neighbor in both partitions
    # Create a list to store results
    results = []
    # Start recording the processing time (R-tree construction excluded)
    start_time = time.time()
```

```python
for user_x, user_y, user_id in user_points:
    # Search Nearest Neighbor in the left partition
    nn_left, dist_left = nn_identifier(rtree_left, (user_x, user_y))
    # Search Nearest Neighbor in the right partition
    nn_right, dist_right = nn_identifier(rtree_right, (user_x, user_y))

    # Select the closer Nearest Neighbor from left or right R-tree
    if dist_left < dist_right:
        nearest_point, dist = nn_left, dist_left
    else:
        nearest_point, dist = nn_right, dist_right
    # Format the results
    if nearest_point:
        # Extract coordinates and ID (handle case where ID may be missing)
        if len(nearest_point) == 3:
            x_p, y_p, point_id = nearest_point
        else:
            x_p, y_p = nearest_point
            point_id = "Unknown"

        results.append(f"point_id = {point_id}, x = {x_p}, y = {y_p} for user {user_id} (distance = {dist:.4f})")
    else:
        results.append(f"User ID {user_id}: No nearest point found")

# End recording the processing time
end_time = time.time()
# Compute the total processing time
total_time = end_time - start_time

return results, total_time
```

The Divide-and-Conquer method splits the parking points into two partitions based on their x-coordinates. It builds two R-trees and searches them independently. Then, it compares the results from both trees and returns the closer one, effectively reducing the search space and improving efficiency.

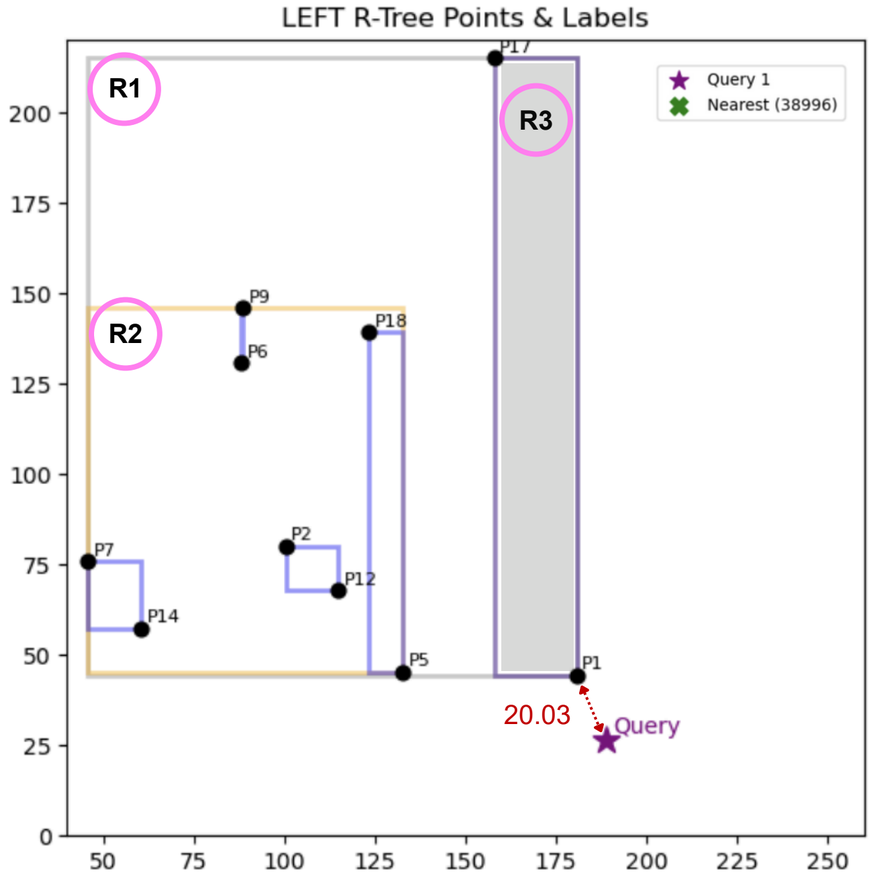
Divide-and-Conquer (Nearest Neighbor)



BaB Divide and Conquer Process

The BaB Divide-and-Conquer method splits the dataset into two partitions based on the median x-coordinate value and builds separate R-trees for each partition.

Divide-and-Conquer (Nearest Neighbor)

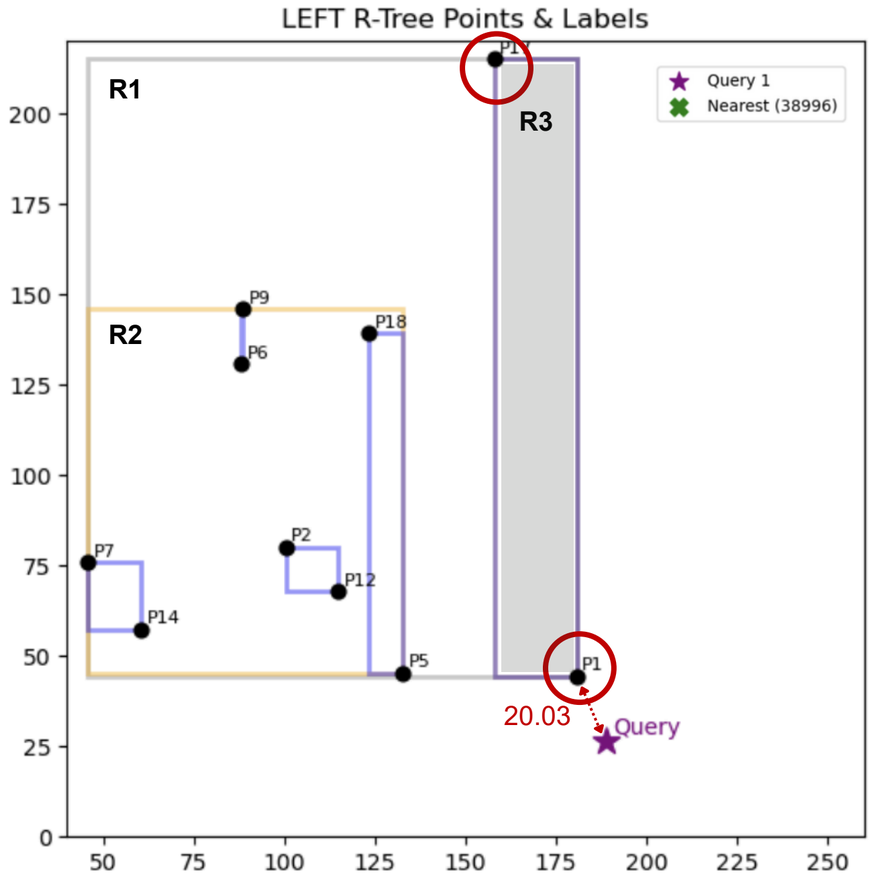


BaB Divide and Conquer Process (Left)

MBR No	minDist
R1 (Root MBR)	20.03
R2	59.23
R3	20.03

Since there is only one root MBR (R1), the search begins from the root. Inside the root MBR, there are two child MBRs (R2 and R3). **R3** has a smaller minDist (20.03) than R2, so R3 is explored first.

Divide-and-Conquer (Nearest Neighbor)

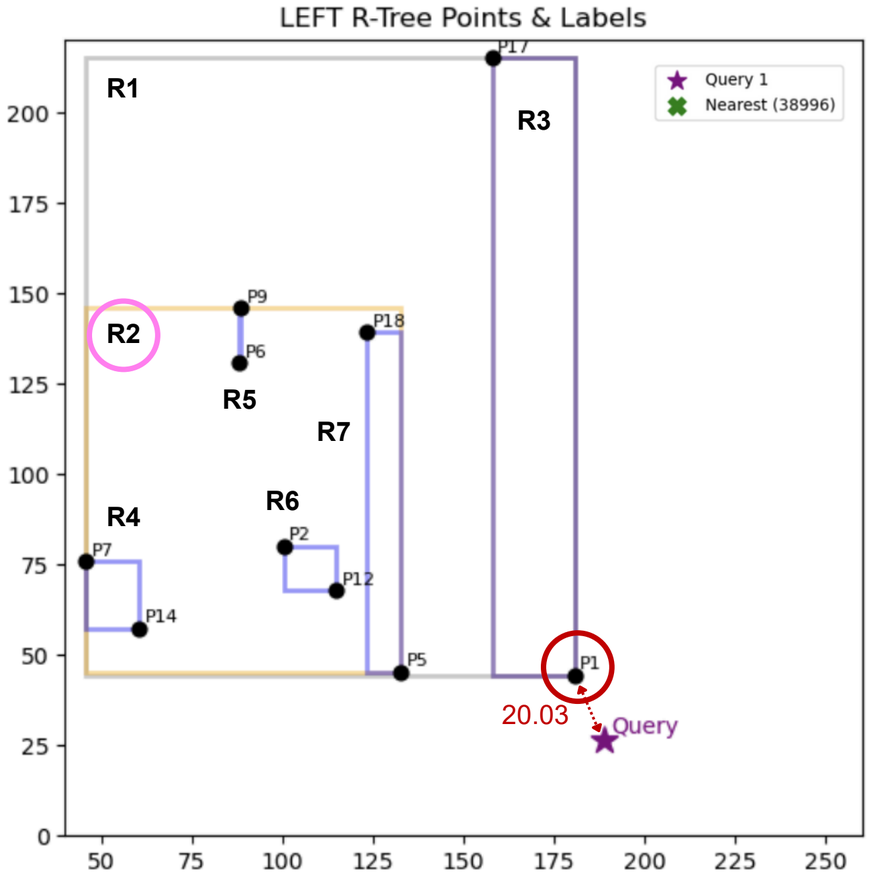


BaB Divide and Conquer Process (Left)

Point No	Distance
P17	191.44
P1	20.03

Since the selected node is a leaf, the algorithm computes the distance from the query to the two points inside it. **P1** is closer to the query (20.03) than P17, so P1 becomes the current nearest point.

Divide-and-Conquer (Nearest Neighbor)



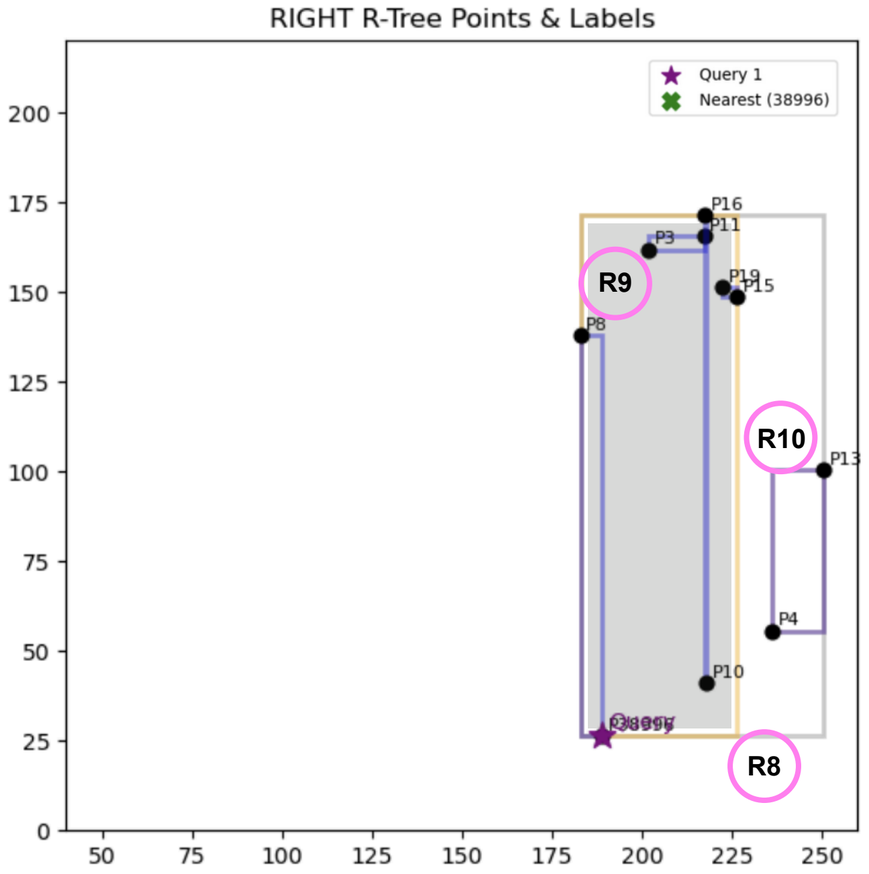
BaB Divide and Conquer Process (Left)

MBR No	minDist
R2	59.23

No other MBR has a smaller minDist than the current shortest distance (20.03), so the remaining MBRs are pruned. Therefore, the nearest parking point in the left partition is P1 (distance = 20.03).

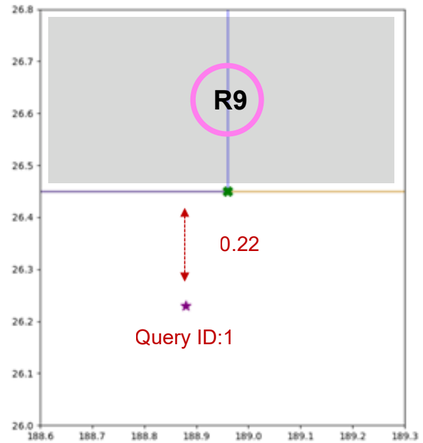
*Child nodes of R2 are not considered, since their minDist cannot be smaller than the minDist of R2.

Divide-and-Conquer (Nearest Neighbor)



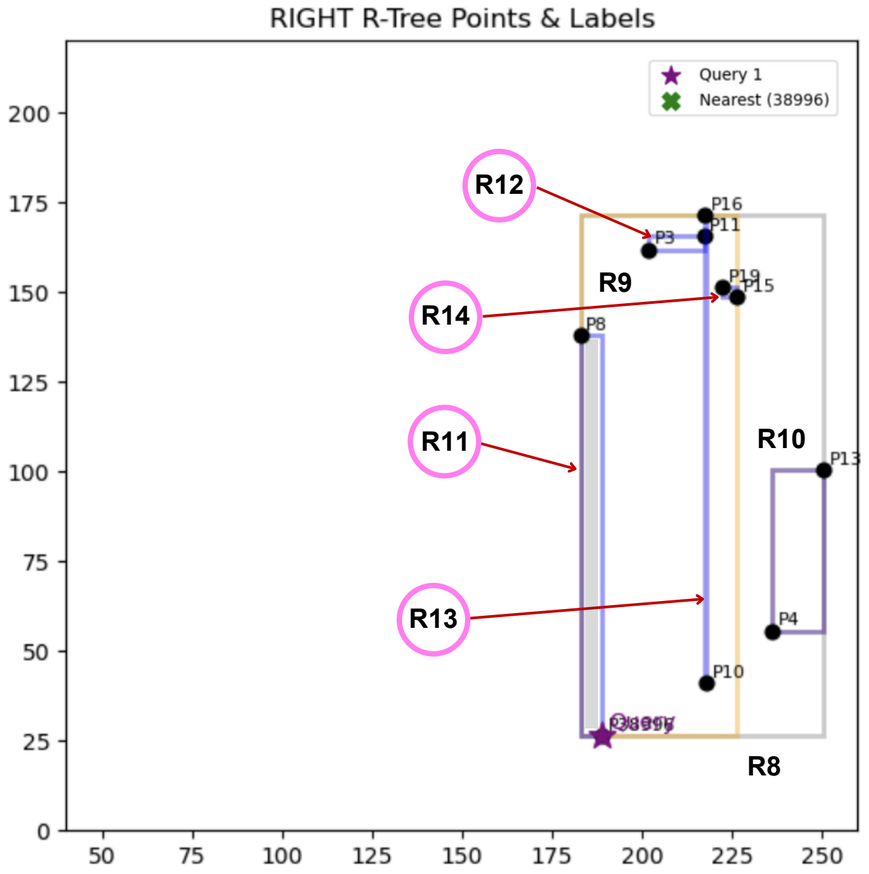
BaB Divide and Conquer Process (Right)

MBR No	minDist
R8 (Root MBR)	0.22
R9	0.22
R10	55.79



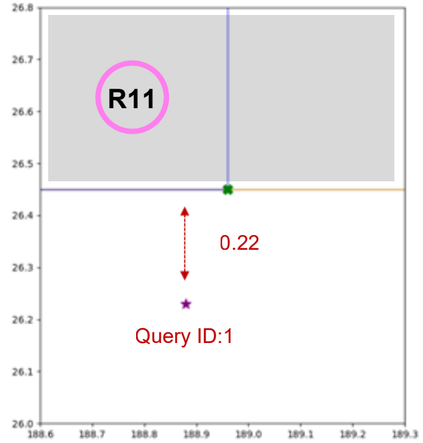
Since there is only one root MBR (R8), the search begins from the root. Inside the root MBR, there are two child MBRs (R9 and R10). **R9** has a smaller minDist (0.22) than R10, so R9 is explored first.

Divide-and-Conquer (Nearest Neighbor)



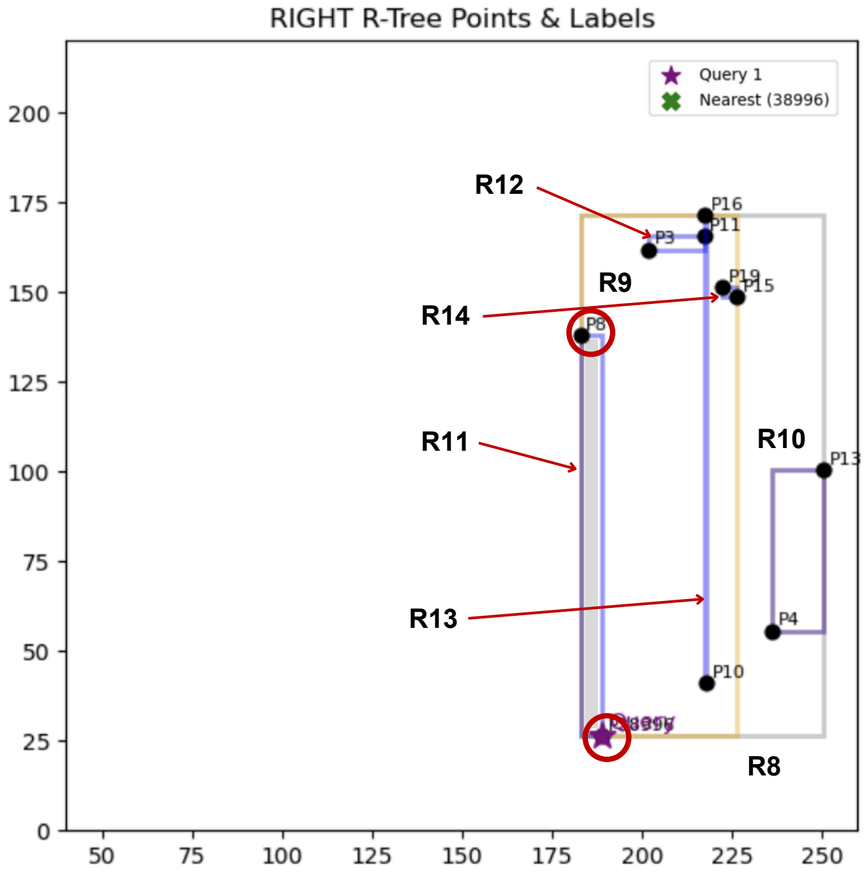
BaB Divide and Conquer Process (Right)

MBR No	minDist
R11	0.22
R12	136.21
R13	32.35
R14	126.96



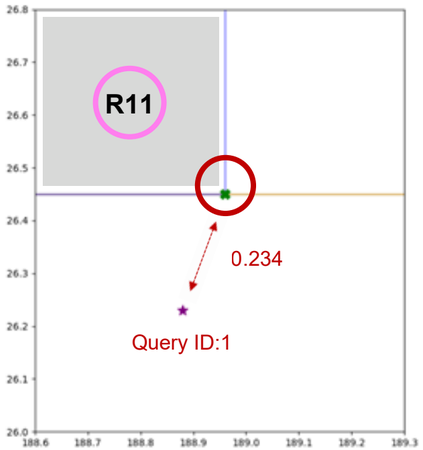
Among the child MBRs of R9, **R11** has the smallest minDist. Therefore, R11 is explored first.

Divide-and-Conquer (Nearest Neighbor)



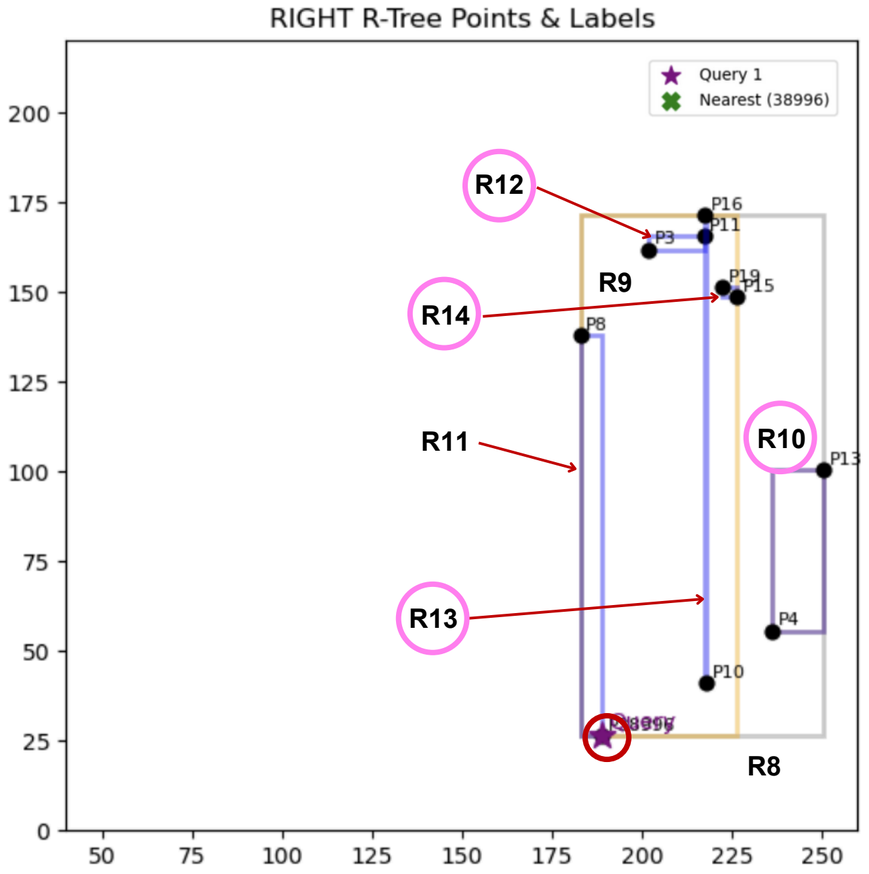
BaB Divide and Conquer Process (Right)

Point No	Distance
P8	112.02
P38996	0.234



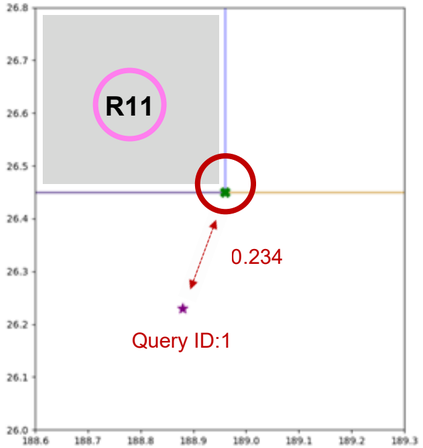
Inside **R11**, there are two parking points, P8 and P38996. Since **P38996** is closer to the query point than P8, P38996 becomes the current nearest neighbor.

Divide-and-Conquer (Nearest Neighbor)



BaB Divide and Conquer Process (Right)

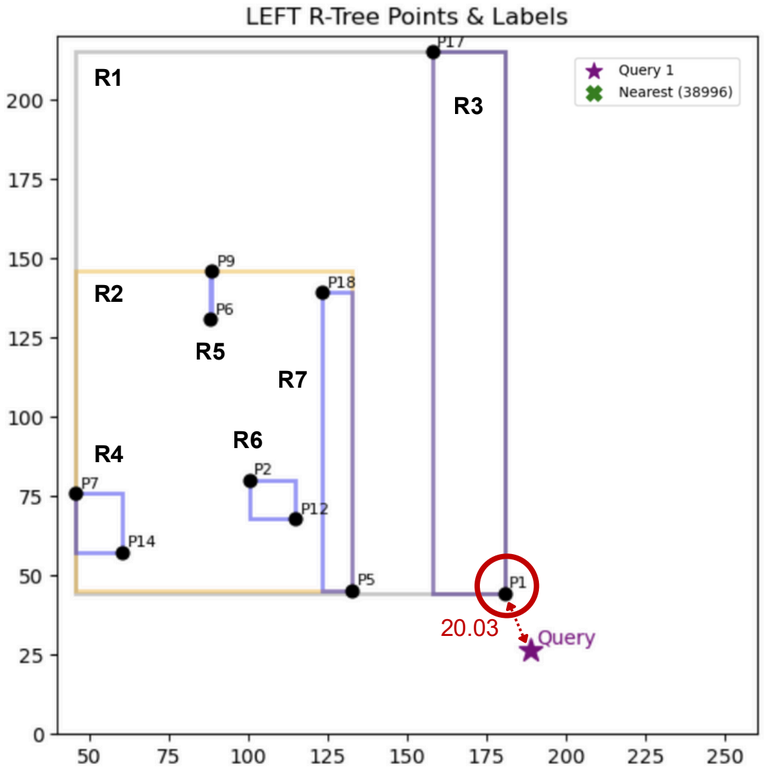
MBR No	minDist
R12	136.21
R13	32.35
R14	126.96
R10	55.79

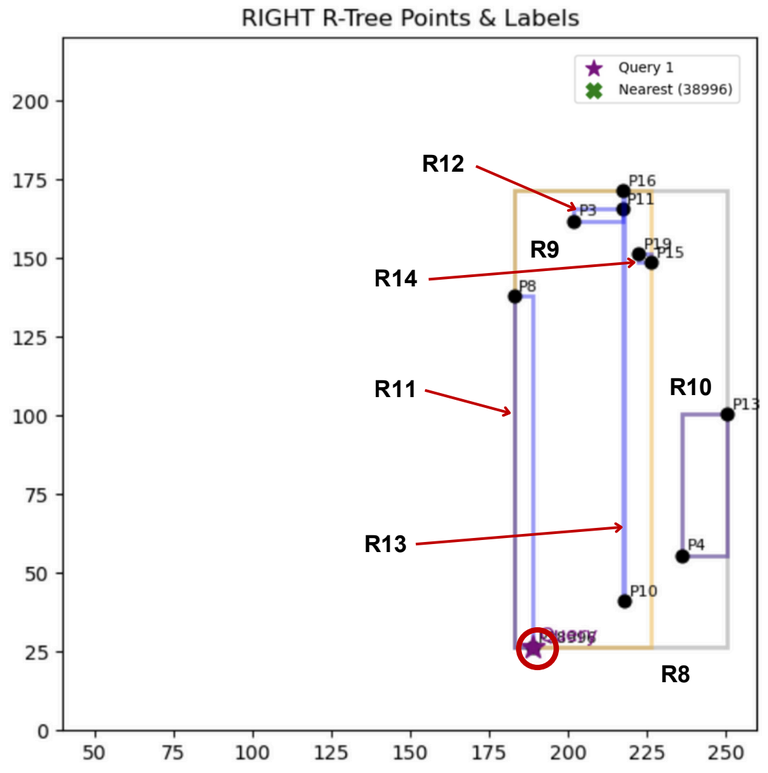


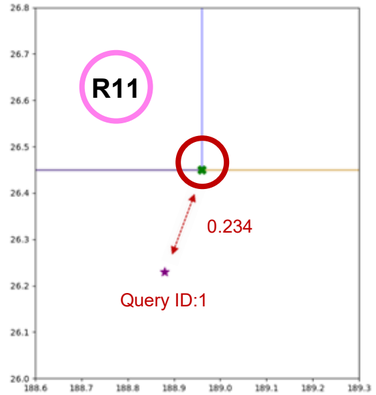
No other MBR has a smaller minDist than the current shortest distance (0.234), so the remaining MBRs are pruned. Therefore, the nearest parking point in the right partition is P38996 (distance = 0.234).

Divide-and-Conquer (Nearest Neighbor)

BaB Divide and Conquer Process (Left and Right)







The left partition finds a nearest distance of 20.03 (P1), whereas the right partition finds a much shorter distance of 0.234 (P38996). Therefore, the global nearest neighbor is **P38996**, which belongs to the right partition.