2D Range Tree with and without Fractional Cascading

CS603 Programming Assignment

Jatin Singhal 22B1277   Avinash Meena 22B1243

Contents

1 Introduction

The task is to design a data structure to answer 2D orthogonal range reporting queries efficiently. Given a static set P of n points in $\mathbb{R}^2$, we wish to preprocess the points such that for a given axis-aligned query rectangle Q, all points inside Q can be reported quickly.

Two implementations are developed:

- (a) A standard 2D Range Tree without optimizations.
- (b) A 2D Range Tree with **Fractional Cascading** to speed up queries.

We compare the time complexities theoretically and empirically by running experiments on random point sets of various sizes.

2 Problem 1: Standard 2D Range Tree

2.1 Pseudocode

Algorithm 1 Build 2D Range Tree

```
procedure BuildXTree(points)
    if points is empty then
        return None
    end if
    Sort points by x-coordinate
    mid ← median point
    node ← new Node with mid point
    node.left ← BuildXTree(points left of mid)
    node.right ← BuildXTree(points right of mid)
    return node
end procedure
```

Algorithm 2 Build Y-Tree

```
procedure BuildYTree(points)
    if points is empty then
        return None
    end if
    Sort points by y-coordinate
    mid ← median point
    node ← new Node with mid point
    node.left ← BuildYTree(points left of mid)
    node.right ← BuildYTree(points right of mid)
    return node
end procedure
```

2.2 Querying Pseudocode

2.3 Approach and Time Complexity

Approach:

- Build a primary BST on x-coordinates.

Algorithm 3 Build Full 2D Range Tree

procedure BUILD2DRANGETREE($XTreeRoot, points$)
  if $XTreeRoot$ is None **then**
    return None
  end if
  $node \leftarrow$ new Node
  $node.point \leftarrow XTreeRoot.point$
  $node.associatedStructure \leftarrow$ BuildYTree($points$)
  Partition $points$ into $leftPoints$ and $rightPoints$ by x-coordinate
  $node.left \leftarrow$ Build2DRangeTree($XTreeRoot.left, leftPoints$)
  $node.right \leftarrow$ Build2DRangeTree($XTreeRoot.right, rightPoints$)
  return $node$
end procedure

Algorithm 4 Query 2D Range Tree

procedure QUERY($xmin, xmax, ymin, ymax$)
  $splitNode \leftarrow$ find node where x splits
  if $splitNode$ is None **then**
    return empty set
  end if
  Initialize $result \leftarrow \emptyset$
  Search left and right subtrees:
    Collect points within $[ymin, ymax]$ from associated Y-trees
    Only visit subtrees where x-ranges intersect $[xmin, xmax]$
  return $result$
end procedure

- At each node, build a secondary BST (Y-tree) on y-coordinates for all points in the subtree.
- For a query, find the split node and explore only relevant subtrees.

Complexities:

- Preprocessing Time: $\mathcal{O}(n \log n)$
- Query Time: $\mathcal{O}(\log^2 n + k)$ where k = number of output points

3 Problem 2: 2D Range Tree with Fractional Cascading

3.1 Pseudocode

Algorithm 5 Build 2D Range Tree with Fractional Cascading

```
procedure BuildYList(points, leftPoints, rightPoints)
    Initialize out ← empty list
    Maintain two pointers leftPtr, rightPtr for left and right children
    for each point p in points sorted by y do
        Record leftPtr and rightPtr positions (or −1 if none)
        Append [p, leftPtr, rightPtr] to out
        if p matches leftPoints[leftPtr] then
            leftPtr ← leftPtr + 1
        end if
        if p matches rightPoints[rightPtr] then
            rightPtr ← rightPtr + 1
        end if
    end for
    return out
end procedure
```

Algorithm 6 Query 2D Range Tree with Fractional Cascading

```
procedure Query(xmin, xmax, ymin, ymax)
    Find split node
    Use binary search to locate starting index in associated Y-list
    Follow precomputed pointers efficiently without repeated searches
    Collect points within the y-range, prune unnecessary branches
    return result points
end procedure
```

3.2 Approach and Time Complexity

Approach:

- Similar to the normal 2D Range Tree.

- Instead of full Y-trees, store sorted Y-lists with cross-references (pointers) between parent and child Y-lists.
- Perform only a single binary search at the split node and use pointers thereafter.

Complexities:

- Preprocessing Time: $\mathcal{O}(n \log n)$
- Query Time: $\mathcal{O}(\log n + k)$ where k = number of output points

4 Comparison Between Standard and Fractional Cascading Range Trees

4.1 Experimental Setup

For experiments:

- Random points in the range $[-100, 100]^2$ were generated for different n.
- Random query rectangles were used.
- Measured both preprocessing time and query time.
- Verified that the outputs of both methods match.

4.2 Description of Results

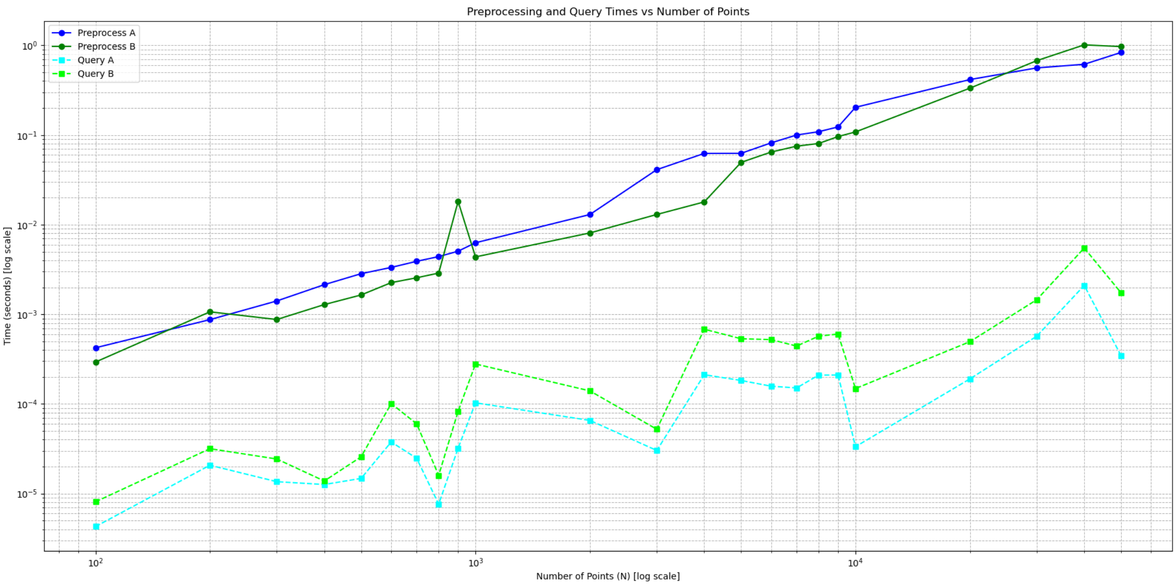

Figure 1: time comparison between standard 2D Range Tree and Fractional Cascading 2D Range Tree.

The plot generated has:

- **X-axis:** Number of points (n), in logarithmic scale.

- **Y-axis:** Time (seconds), also in logarithmic scale.
- **Curves:**
  - Preprocessing times of normal tree (A) and fractional cascading tree (B).
  - Query times of normal tree (A) and fractional cascading tree (B).

4.3 Observations

- Preprocessing times are similar for both methods, as both are $\mathcal{O}(n \log n)$.
- Query time for the fractional cascading approach is significantly better as n increases.
- Standard 2D Range Tree query grows as $\log^2 n$, while fractional cascading is closer to $\log n$.
- Experimental results validate theoretical expectations.

4.4 Conclusion

Applying fractional cascading greatly improves the query efficiency in range trees, especially for large datasets, while keeping preprocessing time and space costs manageable.