Computational Geometry: Project Report

Amogh Johri
IMT2017003

***Abstract*—The problem statement was as follows, "Given a set of n non-intersecting line segments of arbitrary orientation, implement the Segment Tree data structure to answer the following queries : Given a vertical line segment R, report all the segments that intersect R. Also implement the algorithm for the one dimensional variant of the problem - Given a set of intervals, report all the intervals that contain a query point." As part of the current project, two separate interactive implementations have been carried out for the 2-Dimensional and 1-Dimensional cases, respectively. The implementation allows user to give the Input Segments and Query Points/Segments in Graphical User Interface (GUI), and visualize the Elementary Intervals, Segment Tree, Individual Queries, Query path in the Segment Tree, and Node information (including the associated Balanced Binary Search Tree) for every segment tree node. Moreover, a time-analysis has also been done against the brute-force method for the 1-Dimensional variant of the problem.**

I. Segment Tree Data-structure

Segment Tree is a balanced binary-tree derived data-structure. For our case, the nodes in the tree correspond to intervals on the x-axis. We divide the input-intervals into *Elementary Intervals*, where the property of the elementary is that is corresponds to the same input-intervals throughout. The intuition is rather simple, for every query point (or segment), to figure out which input-intervals may contain our query, we only need to find the elementary interval corresponding to it. For the segment-tree data-structure, the leaf nodes corresponds to elementary intervals, and the parent nodes correspond to the union of the intervals of their respective child nodes. Since, this is a balanced binary tree derived data-structure, and every query in the 1-Dimensional case, requires moving from the root node to some leaf node, the time-complexity of search is $O(log(n) + k)$, where *n* corresponds to the number of input-intervals and *k* corresponds to the intervals present in the output. Hence, the algorithm has output-dependent time-complexity. Other metrics associated with the data-structure are:

- **Pre-processing Time-Complexity:** $O(nlog(n))$
- **Space Complexity:** $O(nlog(n))$

For the 2-Dimensional case (with non-intersecting line segments), we obtain a set of line-segments corresponding to every elementary interval, which can then be ordered according to their position on the y-axis. For this, we have a balanced binary search tree corresponding to every node of the segment tree. The algorithm now can be broken into two parts, querying through the segment tree to obtain the correct set of elementary intervals, followed by a query in the BBST associated with every segment tree node corresponding to the obtained elementary intervals. Hence, the metrics associated with this case is:

- **Pre-processing Time-Complexity:** $O(nlog(n))$
- **Space Complexity:** $O(nlog(n))$
- **Query Time-Complexity:** $O(log^2 n + k)$

II. Algorithms

Input:

- List of intervals, where each interval contains 2 values (to represent an interval on the real line).
- List of query points, where the point correspond to a single value on the real line.

Output:

- **Pre-processing:** A segment tree corresponding to the list of intervals.
- **Query:** All the intervals which have a query point lying in its range.

We shall briefly discuss the algorithms for all the required steps.

A. Pre-processing

First we discuss generating the elementary intervals.

```
def getElementaryIntervals(intervals):
    arr, elemIntervals = [], []
    for each in intervals:
        arr.append((each[0]))
        arr.append((each[1]))
    arr                 = (sorted(arr))
    elemIntervals.append \
                (Interval(MIN, \
                arr[0]-epsilon))
    for i in range(0, len(arr)-1):
        if arr[i+1] == arr[i]:
            continue
        elemIntervals.append \
                (Interval(arr[i], arr[i]))
        elemIntervals.append \
            (Interval(arr[i]+epsilon, \
                arr[i+1]-epsilon))
    elemIntervals.append \
            (Interval(arr[-1], \
                arr[-1]))
    elemIntervals.append \
            (Interval(arr[-1]+epsilon, \
                MAX))
    return elemIntervals
```

The time complexity for this step is

$$T(n) = O(nlog(n))$$

Due to the sorting operation on the array containing the end-points of the input intervals.
Having obtained the elementary intervals, now we can create the required segment tree. We create the segment tree in a bottom-up fashion.

```
def createSegmentTree(ins):
    elemIntervals = \
    getElementaryIntervals(ins)
    nodes         = \
    [Node(each, height=0) \
    for each in elemIntervals]
    root          =  \
    recursiveCreateSegmentTree(nodes)
    return attachIntervals(root, ins)
```

The **ins** corresponds to the list of intervals. This function is paired up with the following recursicve function.

```
def recursiveCreateSegmentTree(nodes):
    newNodes = []
    i = 0
    while i < len(nodes)-1:
        newNode = Node \
        (Interval.union \
        (nodes[i].getInterval(), \
        nodes[i+1].getInterval()),\
        parent=None, \
        leftChild=deepcopy(nodes[i]), \
        rightChild=deepcopy(nodes[i+1]),\
        height=max(nodes[i].getHeight(),\
        nodes[i+1].getHeight()) + 1)
        newNode.getLeftChild(). \
        setParent((newNode))
        newNode.getRightChild(). \
        setParent((newNode))
        newNodes.append((newNode))
        i += 2
    if len(nodes) % 2 == 1:
            newNodes.append((nodes[-1]))
    if len(newNodes) > 1:
        return \
        recursiveCreateSegmentTree \
        (newNodes)
    return newNodes[0]
```

This concludes our creation of the segment tree. The only remaining step is to attach the intervals to the correct nodes. We cannot attach intervals only at the leaf nodes, as that would result in an $O(n^2)$ space complexity. Hence, we do it in an efficient manner, while also storing these at non-leaf nodes. The algorithm for this is as follows:

```
def attachInterval(curr, interval):
  if curr == None:
    return
  else:
    if interval. \
    contains(curr.getInterval()):
      curr. \
      addInterval(interval.getInterval())
    else:
      if curr.getLeftChild() != None:
        if curr.getLeftChild(). \
        getInterval().overlaps(interval):
          attachInterval \
         (curr.getLeftChild(), interval)
      if curr.getRightChild() != None:
        if curr.getRightChild(). \
        getInterval().overlaps(interval):
          attachInterval \
          (curr.getRightChild(), interval)
    for each in intervals:
        attachInterval \
        (root, Interval(each[0], each[1]))
    return root
```

Finally, this concludes the formation of the data-structure for the purpose of our problem statement.

B. Query

The query algorithm is relatively straight-forward. It is analogous to a search on a binary tree, where at each node we return the list of intervals that are associated with that node.

```
def query(root, value):
    out = []
    curr = root
    while True:
     out.extend(curr.getIntervalArr())
     if curr is not a root:
      if value in curr.getLeftChild()\
      .getInterval():
        curr = curr.getLeftChild()
        continue
      if value in curr.getRightChild()\
      .getInterval():
        curr = curr.getRightChild()
        continue
      return out
```

We only discuss the algorithms for the 1-Dimensional case, as they are very similar for the 2-Dimensional as well. Instead of attaching the intervals to an array for every segment, we do a binary tree insert (for this, we need a self balancing binary search tree). Moreover, in the query algorithm, rather than just appending with all the intervals corresponding to a segment tree node, we do a binary search and add the required intervals only. Apart from these, the algorithms for both the cases are brutally similar.

III. Analysis

A. Query Time Comparison

4000 randomly generated test-runs (in Fig. 1) were carried out (where the number of randomly generated input-intervals were from 1-20,000 in gaps of 50), and each run was made to query for 100 random query points.

In all the plots below, y-axis corresponds to time in seconds. Clearly, we can see (in Fig. 2) that even for modest values of

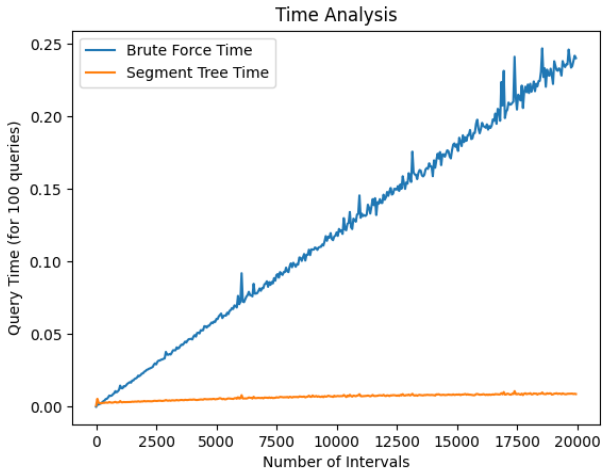

Fig. 1. Query Time Comparison (Brute Force - O(n) vs Segment Tree Query

a few-thousand intervals and 100 queries, the Segment Tree based algorithm gives us immense efficiency boost. However, in this figure it is not possible to understand how the time-taken by the Segment Tree based algorithm increase with the number of intervals, hence, a separate analysis was carried for the that. Here, (ignoring a few jitters which are probably due

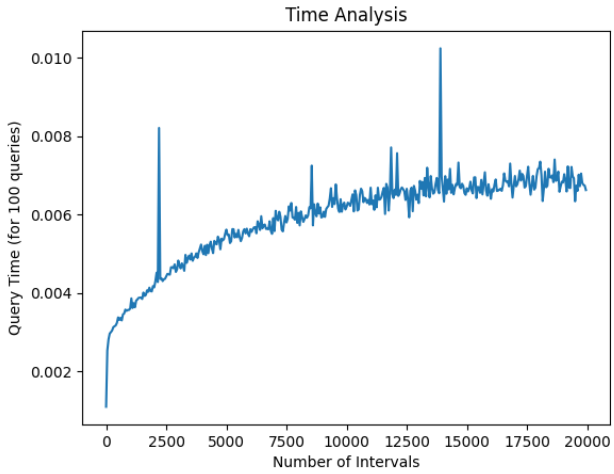

Fig. 2. Query Time Analysis - Segment Tree Query

to my desktop straggling), we can see that the curve follows a logarithmic pattern. This is in-line with what we expect to see, considering the Query time-complexity is $O(log(n)+k)$. Next we analyze how the time-complexity relates with increasing the number of queries (in Fig. 3). For this, we manitain the number of input-intervals at 10,000, and vary the number of queries. We carry a similar analyze, comparing brute force with segment tree based algorithm.

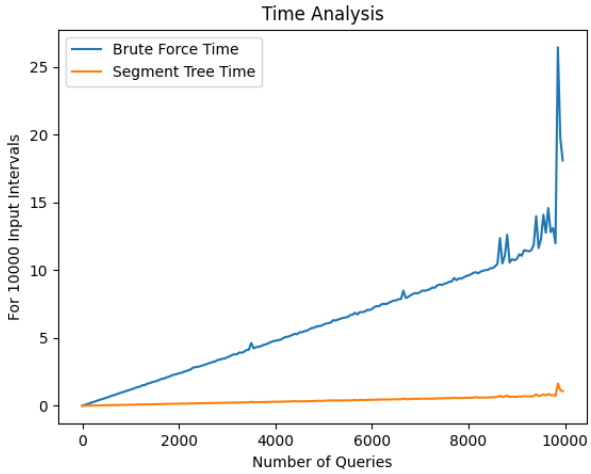

Fig. 3. Query Time Comparison (Brute Force - O(n) vs Segment Tree Query

B. Pre-processing Time Analysis

We also analyze the pre-processing time complexity for segment tree (in Fig. 4). The theoretical complexity for this is $O(nlog(n))$. For this, we vary the number of intervals (from 1-20,000 in gaps of 50), and observe the time required for creation.

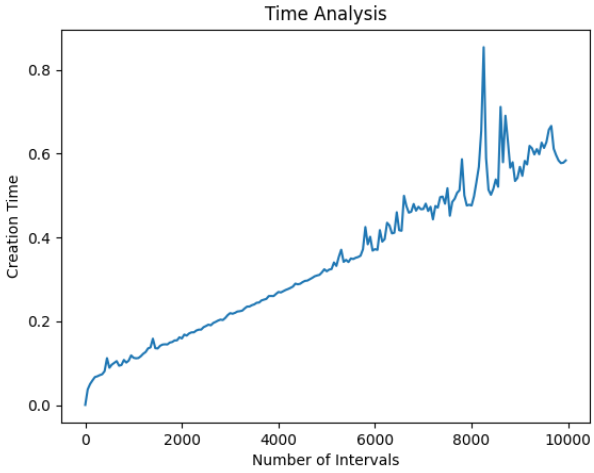

Fig. 4. Pre-processing time complexity analysis for Segment Tree