CS345 Theoretical Assignment 1 Submission

Naman Kumar Jaiswal
220687
namankj22@iitk.ac.in

Fall Semester 2024

Question 1: Pairs of Intersecting Line Segments

We need to find an $\mathcal{O}(n \log^2(n))$ to find number of pairs of intersecting line segments given the starting and ending points of n line segments, all the points lying over a circle.

Construction:

We are given two sets of points $\{p_1, p_2, p_3, \cdots p_n\}$ and $\{q_1, q_2, q_3, \cdots q_n\}$, where the ith line segment starts from p_i and ends at q_i. A point is just a geometric entity, hence for proceeding to solve this problem we need to identify / enumerate the points in some way so that they can be used for further numerical analysis. Here, I will be using the polar coordinate θ to uniquely identify a point (since the radial coordinate is going to be same for all the points).
We will start by taking a random point on the circle and defining it as $\theta = 0$, and labelling all other points as the azimuth angle computed from the line $\theta = 0$ in the anti-clockwise direction from 0 to 2π. In the next step, we will start by defining a line segment as a pair of points defined by (θ_0, θ_1), where θ_0 is the initial point and θ_1 is the final point. Without loss of generality, we will also enforce the relation θ_0 ¡ θ_1. This can be argued because the line segment (θ_0, θ_1) is same as (θ_1, θ_0), enforcing the above relation will ensure that we have an unique representation for a line segment (chord in this context of a circle).
The ith line segment will be (θ_{p_i}, θ_{q_i}) if $\theta_{p_i} < \theta_{q_i}$ or else (θ_{q_i}, θ_{p_i}). Here p & q lose their meaning and now we can represent the line segment as $(\theta_{0i}, \theta_{1i})$. Here, θ_{0i} is the smaller of the two.
We have created a construction where the geometrical definitions have been given a numerical representation and we have transformed the given inputs into a form that can be further used by our algorithm.

Heuristic:

After the above construction, we can observe that for three line segments - i and j, there are two cases that can happen:

1. The $(\theta_{0i}, \theta_{1i})$ line segment nest inside the $(\theta_{0j}, \theta_{1j})$ line segment. That is either j nests inside i $\theta_{0i} < \theta_{0j} < \theta_{1j} < \theta_{1i}$ or vice versa. The line segments, do not intersect in this case. This can be proved geometrically, as there will be no cross between them.

2. The $(\theta_{0i}, \theta_{1i})$ line segment is disjoint from the $(\theta_{0j}, \theta_{1j})$ line segment. That is either i ends before j starts $\theta_{0i} < \theta_{1i} < \theta_{0j} < \theta_{1j}$ or vice versa. The line segments, do not intersect in this case. This can be proved geometrically, as there will be no cross between them as well.

3. The $(\theta_{0i}, \theta_{1i})$ line segment and the $(\theta_{0j}, \theta_{1j})$ line segment are non-nested. That is one crosses the other which is either $\theta_{0i} < \theta_{0j} < \theta_{1i} < \theta_{1j}$ or vice versa (i and j swapped). The line segments, intersect in this case. This can be proved geometrically, as there will be a cross between them and will yield an intersection. We need to count theses cases.

The above heuristic gave us a relation that needs to be checked in $\mathcal{O}(1)$ time to find if a given pair of line segments intersect.

Pre-Processing:

There are six possible cases in which we need to hunt for the last two. In a brute force algorithm, we will need the complexity to go to $\mathcal{O}(n^2)$*$\mathcal{O}(1)$. ${}^nC_2$ pair of points and $\mathcal{O}(1)$ time to find if they intersect. We can make a reduction by reducing the number of cases to 3 (1 favourable) by introducing a form of sorting. In this Sorted array of pairs (each pair representing a line segment) we can apply divide and conquer to reduce the time complexity to something sub quadratic.
The only pre-processing we need to do is to sort an array of pairs by θ_0 in increasing order. If there are two line segments with same θ_0, we will sort by θ_1. Let the sorted array be S.

Logic:

1. **Divide Step**: First we will need to split the given sorted array of line segments into two parts - Left (S_l) and Right (S_r). The split is to be made near the half point (median) of the array so that the sub problems created are roughly equal and we do not encounter a worst case time complexity of $\mathcal{O}(n^2)$ because the sub problem being almost same as the original problem forming a brute force.

2. **Conquer Step**: Here we need to recursively call the function on both the portions - left and right. We will also need to add a base case to prevent the case of never ending recursive calls.

3. **Merge Step**: Here we need to use the following recursive relation and merge the solutions of the two sub problems to generate a solution for the original problem.

$$NumberPairs(S) = NumberPairs(S_l) + NumberPairs(S_r) + CrossPairs(S_l, S_r)$$

   Here, the NumberPairs(P) function counts the number of pairs of intersecting line segments where both line segments are in P, and CrossPairs(P,Q) is the number of pairs of intersecting line segments where first line segment is in P and other is in Q. This is a recursive relation for NumberPairs() function.

4. **CrossPairs Function**: For this term, we need to find for each line segment in S_r (θ_0, θ_1), we need to find the line segments in S_l (θ_{0i}, θ_{1i}) such that $\theta_{0i} < \theta_0 < \theta_{1i}$. Here, we will use an exploit for optimization which is $\theta_{0i} < \theta_0 < \theta_1$ is always guaranteed. From here we can sort the θ_{1i} values of the left half and apply Binary Search of θ_0 and θ_1 over the sorted values to find the indexes which are just less than θ_0 and just more than θ_1. The difference between these indices is the output of our function.

Merging all the way from bottom to up using the recurrence relation will give us the correct answer.

Pseudocode:

Here, we are assuming that the inputs are converted as per the construction given above. The exact conversion steps will depend on how the points are given as input. For the sake of pseudocode we are making this assumption.

1. First we will need to sort the array of line segments. We are allowed to assume sorting to be done in $\mathcal{O}(n \log(n))$

```
S ← sort(Arr);
```

2. Function to count the intersecting pairs

```
function CountIntersectingPairs(S):

    // Base Case
    if (length(S) <= 1)
        return 0;

    // Divide step
    Compute median of S;
    (S_l, S_r) ← Split-by-median(S);

    // Conquer step
    pairs_in_S_l ← CountIntersectingPairs(S_l);
    pairs_in_S_r ← CountIntersectingPairs(S_r);

    // Merge step
    cross_pairs ← CountCrossPairs(S_l, S_r);

    return pairs_in_Sl + pairs_in_Sr + cross_pairs;
```

3. Function to count cross pairs

```
function CountCrossPairs(S_l, S_r):

    theta1_values = [segment[1] for segment in S_l];
    Theta ← sort(theta1_values);

    cross_pairs = 0

    for line_segment (Theta0, Theta1) in S_r:

        // Binary Search Left gives the just smaller index
        // Binary Search Right gives the just larger value
        low = BinarySearchLeft(Theta, Theta0);
        high = BinarySearchRight(Theta, Theta1);

        cross_pairs += high - low;

    return cross_pairs
```

Time Complexity Analysis:

Let us assume that the NumberPairs(P) function takes T(n) time to return the answer, where n is the number of line segments in P. The NumberPairs() function calls itself on half number of points twice and then on the merge step also counts the number of CrossPairs(). For finding the number of cross pairs, we are sorting the left array on the basis of θ_1 values and then we are applying binary search twice for each element in S_r. This amounts to $\mathcal{O}((n/2) * \log(n/2) + 2 * (n/2) * \log(n/2))$. This amounts to $\mathcal{O}((3n/2) * \log(n/2)$ which is bounded by $\mathcal{O}(n \log(n))$.
Now,

$$T(n) = 2 * T(n/2) + cn \log(n) + d$$

Using Split the Recurrence method of solving recurrence, we see that the

$$T(n) = \mathcal{O}(n \log^2(n))$$

We also need to add the pre processing step which involves sorting and takes $\mathcal{O}(n \log(n))$ time. So, Overall Time Complexity is $\mathcal{O}(n \log^2(n)) + \mathcal{O}(n \log(n)) \equiv \mathcal{O}(n \log^2(n))$.

Question 2: Non-Dominated Points

Part a: Online Algorithm for 2D Non-Dominated Points

We need to find an $\mathcal{O}(i\log(i))$ algorithm that returns a set of non-dominated points after being given i points one by one. Apart from this we also need to maintain the set of non-dominated points every time a new element is inserted. We want an average insertion time of $\mathcal{O}(\log(i))$ when we already have i elements in the input set.

Heuristic:

1. The first we need to observe is that when the points of the input array are plotted on a graph, the non-dominated points when connected will form an envelope over the points that have been dominated by some other point. Since, the dominated points will have its coordinates both less than the coordinates of a non-dominated point, the non-dominated points lie on the extremum of the set of point plotted on the graph.
   Secondly, if we have a point (x,y) that is confirmed to be a non-dominated point, the next non-dominated point is going to be somewhere going along the positive x-axis and negative y-axis. The other possible direction is along negative x-axis and positive y-axis. If we go along any other direction, either the new point is non-dominated or the (x,y) is non-dominated.
   Following this pattern we are going to get a staircase pattern going from top-left to bottom-right.
   The above heuristic gave us a relation that if we have have two non-dominated points (x_1, y_1) and (x_2, y_2) such that $x_1 < x_2$ we are confirmed that $y_1 > y_2$. This allows us to store all the points in a single data structure in some sorted manner on the basis of any one coordinate.

2. Let us assume that we have a Data Structure DS storing the non-dominated points in some sorted order. Let us assume that for the i-1 points we have already stored the non-dominated points in the DS. When we insert the i_{th} point, we will need to see that:

   (a) A dominated point in the i-1 points will still have a point that will dominate over it in the new set of i points. This implies that a dominated point will still be a dominated point after the insertion of the i^{th}.

   (b) The i^{th} point may dominate over a point that was previously non-dominated. Hence, we may need to delete this element.

   (c) The i^{th} point may or may not be a non-dominated point. To check for domination, there is no need to check with the dominated points, there will always exist a non-dominated point in the DS which dominates over the dominated point, checking in the DS alone will suffice.

3. Lastly, for the choice of data structure DS, we can see that there are insert and delete operations that need to be done in an average of $\mathcal{O}(\log(i))$ time. We cannot have a linear data structure as they do not gurantee a logarithmic time insert, delete operations. Hence, we need to use some sort of tree based Data Structure. Let us use Red-Black Tree as the choice for the data structure to store the non-dominated points.

Logic:

1. **Construction**: We will build a Red Black tree with the ordering based on x-coordinate (we could have chosen y-coordinate as well, the points inside the tree follow a staircase pattern going from top-left to bottom-right). When we are at the insertion of i_{th} step, the tree will be containing non-dominated points among the i-1 points.

2. **Changes**: The i^{th} element may not remove any element from the tree or may remove some elements from the tree. The elements this point can remove are always going to be continuous. This can be trivially proven from the definition of domination and non-dominated points. Let the tree contain $\{(x_1, y_1), (x_2, y_2), (x_3, y_3) \cdots (x_k, y_k)\}$ after i-1 points, if the new point (x, y) removes some portion of the tree $\{(x_a, y_a), (x_{a+1}, y_{a+1}), (x_{a+2}, y_{a+2}) \cdots (x_b, y_b)\}$, it must satisfy $x_{a-1} > x > x_a > x_{a+1} > \cdots x_b$ and $y_a < y_{a+1} < \cdots y_b < y < y_{b+1}$. This relation not only detects if we need to delete elements from the tree as well as tell what points need to be deleted. Here, we need to delete from a to b (inclusive)

3. **Binary Search Red Black Tree**: For deleting the given segment, we will start by deleting the point a and then go on till b. We will search in the Red Black Tree for the point x_a which is less than x and also just largest among all the points satisfying the condition.
   Check: is y $> y_a$?

   (a) If yes, remove the point a from the Red Black Tree. Repeat this step.

   (b) If no, break from the loop.

4. **Adding the i^{th} point to the tree**: We know that the non-dominated points will always form the staircase pattern from top-left to bottom-right. So, whether or not this point fits in the pattern will decide if we have to add this point to the Red Black Tree. So, if the Red Black Tree is empty, we will insert the point. Else, we binary search to find the point with x_i ¿ x, and is the smallest among the points which satisfy this condition. If the point does not exist, we are at the extreme bottom-right of the staircase, we will add the new point. Else, if the point is found, this point would be the successor of the new point if it gets added. **Check**: is $y > y_i$?

   (a) If yes, add the new point.

   (b) If no, don't add the new point.

Pseudocode:

1. Main function containing the driver code

```
function Main():

    tree = RedBlackTree();          // Initialise to get an empty Red Black
    Tree

    while (input.available) {
        new_point ← input point;
        Update(tree, new_point);
    }

    return tree;
```

2. Function to update the tree with the new incoming input points

```
function Update(tree, point):

    (x, y) ← point;

    // Check for points to delete
    // Find point with x-coordinate just greater than x

    current = tree.FindSuccessor(x);

    while current is not null and current.x <= x:
        if current.y < y:
            // The current point is dominated by the new point (x, y)
            tree.Delete(current);

            current = tree.FindSuccessor(x);  // Continue searching

        else:
            break;                            // Need not search further

    // Add the new point if it is non-dominated
    if tree.IsEmpty() or (current is null or current.y < y):
        tree.Insert(point);
```

Time Complexity Analysis:

This algorithm does not guarantee a bound of $\mathcal{O}(\log(i))$ for the Update operation for the ith point but it does guarantee an average bound of $\mathcal{O}(\log(i))$ and makes sure that after i points are inserted, the bound does not go over $\mathcal{O}(i\log(i))$. This can be proven as follows:

1. There are three types of points among the i points:
   (a) The points that were once a part of the Red Black Tree and but currently are not. These points were once added to the Tree. For addition, we took one search operation and one insertion operation on the tree. In total, they took $\mathcal{O}(\log(i)) + \mathcal{O}(\log(i)) \equiv \mathcal{O}(\log(i))$ time. For deletion, we took one search operation to find the index and one delete operation. They took $\mathcal{O}(\log(i)) + \mathcal{O}(\log(i)) \equiv \mathcal{O}(\log(i))$ time to delete. In total they took they took $\mathcal{O}(\log(i))$ time for getting processed.
   (b) The points who were once a part of the tree and still are. They only took one search operation and one insertion operation. Even they took $\mathcal{O}(\log(i)) + \mathcal{O}(\log(i)) \equiv \mathcal{O}(\log(i))$ time to get processed.
   (c) The points who were never inserted into the tree. They only took one search operation and then were given the dominated tag. They took $\mathcal{O}(\log(i))$ to be processed.
2. There, each point took $\mathcal{O}(\log(i))$ time to be processed and hence we can say that in total they took $\mathcal{O}(i\log(i))$ to get processed.
3. Other than the time taken for point processing, we also have some time taken up by the search operation before the break statement which takes $\mathcal{O}(\log(i))$ more time per query.
4. In total we can say the algorithm took $\mathcal{O}(i\log(i))$ + i*$\mathcal{O}(\log(i)) \equiv \mathcal{O}(i\log(i))$ total.

This bounds the total time taken by the algorithm by $\mathcal{O}(i\log(i))$ and average query time by $\mathcal{O}(\log(i))$.

Part b: Algorithm for 3D Non-Dominated Points

We need to find an $\mathcal{O}(n\log(n))$ algorithm that returns a set of non-dominated points after being given n 3D points.

Heuristic:

1. We will use the algorithm used in the solution of problem 2(a) in this question as well.
2. Reduction to 2D Non-Dominated Points Problem: Once the set of points are sorted by z-coordinate, the problem of determining if a point is non-dominated reduces to checking dominance in the xy-plane. This is because if a point is not dominated in xy-plane by any previously considered points (with higher z), it must be non-dominated in 3D. Hence we are going to sort the points in decreasing z and then query the points one by one in a way similar to the algorithm 2(a) using the Red Black Tree data structure.
3. Lastly, for the choice of data structure DS, we can see that there are insert and delete operations that need to be done in an average of $\mathcal{O}(\log(i))$ time. We cannot have a linear data structure as they do not gurantee a logarithmic time insert, delete operations. Hence, we need to use some sort of tree based Data Structure. Let us use Red-Black Tree as the choice for the data structure to store the non-dominated points.

Logic:

1. **Construction**: First, we sort the set S in decreasing order of their z-coordinate. This ensures that while processing, we only need to consider points that have already been processed for domination. We maintain a Red-Black Tree tree, which stores the non-dominated points encountered so far, ordered by their x-coordinate.
2. **New Point Processed**: For each point p in the sorted set S, We check whether p is dominated by any of the points currently in P by examining its x and y-coordinates. If p is non-dominated, we insert it into P. During this insertion, we might need to remove any points in P that p now dominates, as these points are no longer relevant. For deleting we are going to use the Predecessor(a) function (Find the largest x value in the tree that is less than a) and Successor(a) function (Find the element with the smallest x greater than or equal to a).

Pseudocode:

1. Function that takes the Set of 3D points S as input and outputs the tree of 3D points which are non-dominated as output

```
function FindNonDominatedPoints(S):

    // Sort the points in S by decreasing z-coordinate
    S_sorted ← sort(S, by: z-coordinate, order: decreasing);

    // Initialize an empty Red-Black Tree tree

    tree ← RedBlackTree();

    for each p in S_sorted:

        // Find the element with the smallest x greater than or equal to p.x
        successor = tree.FindSuccessor(p.x);

        // Check if p is dominated by any point in the tree

```

```
        if successor is null or successor.y < p.y:

            // Remove points from the tree that are dominated by p

            // Find the largest x value in the tree that is less than p.x.
            predecessor = tree.FindPredecessor(p.x);

            while predecessor is not null and predecessor.y < p.y:
                tree.Delete(predecessor);

                // Continuue Searching
                predecessor = tree.FindPredecessor(p.x);


            // Insert p into the Red-Black Tree and add it to the Set

            tree.Insert(p);

    return tree;
```

Time Complexity Analysis

The time complexity of our algorithm for finding non-dominated points in a 3D space is as follows:

1. **Sorting the Points:** We start the algorithm by sorting the points array in decreasing order of z. We are allowed to assume that sorting takes $\mathcal{O}(n \log(n)$ time.

2. **Processing Each Point:**

   (a) The points that were once a part of the Red Black Tree and but currently are not. These points were once added to the Tree. For addition, we took one search operation and one insertion operation on the tree. In total, they took $\mathcal{O}(\log(i)) + \mathcal{O}(\log(i)) \equiv \mathcal{O}(\log(i))$ time. For deletion, we took one search operation to find the index and one delete operation. They took $\mathcal{O}(\log(i)) + \mathcal{O}(\log(i)) \equiv \mathcal{O}(\log(i))$ time to delete. In total they took they took $\mathcal{O}(\log(i))$ time for getting processed.

   (b) The points who were once a part of the tree and still are. They only took one search operation and one insertion operation. Even they took $\mathcal{O}(\log(i)) + \mathcal{O}(\log(i)) \equiv \mathcal{O}(\log(i))$ time to get processed.

   (c) The points who were never inserted into the tree. They only took one search operation and then were given the dominated tag. They took $\mathcal{O}(\log(i))$ to be processed.

   There, each point took $\mathcal{O}(\log(i))$ time to be processed and hence we can say that in total they took $\mathcal{O}(i \log(i))$ to get processed.

   In total we can say each query took $\mathcal{O}(i \log(i))$ total after processing i points.

3. **Final Time Complexity:** We have to sum the time of sorting as well as time for processing each query. The required time complexity $\mathcal{O}(n \log n) + \mathcal{O}(n \log n) = \mathcal{O}(n \log n)$

The overall time complexity of our algorithm is $\mathcal{O}(n \log n)$ as required.