CS345 Theoretical Assignment 1

Ayush Agarwal, 13180

M.Arunothia, 13378

Contents

1 Non-Dominated Points

1.1 Overview

Given a set of coordinates P, we create list of each layer in the following manner. First sort the coordinates based on Y-coordinate in descending order. Then maintain an array A of size n. Start with the first coordinate from the sorted array P(of all coordinates). This point will be a non-dominated point and will be a part of layer 1. Update the first index of A with the x-coordinate of this point. Now take the second point from P. If its x coordinate is greater than the x-coordinate of earlier point, it means that it will be part of layer 1. If so then add it to layer 1 and update the layer 1's index in A. Otherwise it will be in second layer, so add it in layer 2 and update the layer 2's index in A with it's x coordinate. Repeat the above procedure for all points.

1.2 Pseudo-Code

```
Non-Dominated-points(P)
{
        $P \longrightarrow reverse\_sort(P)$ //sort in descending order of Y
        $Layer[n]; A[n]$
        $A[0] = P[0].x$
        $Layer[1].push(P[0])$
        $i = 1; right = 1$
        $while(i < P.length())$
                $point = P[i]$
                $index = binary\_search\_predecessor(A, 0, right, point.x)$
                                // returns the predecessor's index
                $Layer[index].push(point)$
                $A[index] = point.x$
                $If(index > right)right++$
        $returnLayer$

}

binary_search_predecessor(A, left, right, x) {
        If no entry in A is less than x, return $right + 1$
else return the index of maximum x coordinate less than x.
}
```

1.3 Time Complexity

Sorting step takes $O(nlogn)$ time, followed by binary_search for each point which takes maximum $logn$ time per point. $while$ iterates for all the points and in each iteration binary_search is invoked, thus the loop takes $n * logn$ time. Overall

algorithm takes time
$O(nlogn) + O(nlogn) = O(nlogn)$

1.4 Proof of Correctness

1.4.1 What is to be proved?

As we go along the iterations, say we have covered k points then, we have partially constructed layers. If the new $k+1$th point lies between the ith and $i+1$th (that is the x predecessor of $k+1$ is the last point encountered in layer i) of these layers then it has to belong to layer i.

1.4.2 Reasoning by Contradiction

- It cannot belong to any layer $> i$ as the layers are increasingly drawn in x.
- It cannot belong to any layer $< i$ as this point is clearly not dominated by the points in layer i that is seen so far.
- Hence, The point should belong to layer i.

2 Open Rectangle Query

2.1 Data Structure Design :

Given an array 'a' of 'n' coordinate points, we construct a Binary Search Tree (BST) call it 'data' in the following manner.

- Sort the array 'a' w.r.t the x-coordinates of the points. Call this sorted array 'b'.
- Divide 'b' into $\frac{n}{Log[n]}$ parts, starting from the beginning. Index each of the part incrementally from 1 to $\frac{n}{Log[n]}$.
- Construct BST 'data' with $\frac{n}{Log[n]} = N$ nodes from 'b' using the above indexing for the comparisons.
- Now, we have a BST 'data' with 'N' nodes augmented with an array of $Log[n]$ size at every node and, also x_min and x_max of its *logn* chunk. Sort this array at every node on basis of y-coordinates of the points.
- Every node is augmented with another inversely sorted array of y_max(maximum y coordinate of each logn size array) of each of the nodes belonging to the subtree, accompanied by a pointer to that node. In short a node at level i from the bottom will be augmented with two arrays, one of size *logn* say short_array which contains the sorted array of according to y, other will be an array of size 2^i say long_array.
- This completes the description of augmented BST 'data'.

2.2 Preprocessing :

STEP 1 : Start

STEP 2 : Make a bst as specified above with each node having *logn* size sorted array of its chunk. For the other array start traversing from the leaf node. The leaf node will have will have only one entry in its long_array, i.e, the maximum y coordinate in its logn chunk, and the pointer to itself.

STEP 3 : Now consider the parent node of this leaf node. It will have 3 entries in its long_array in the sorted order, two from its children and the third from itself(which has a pointer to itself).

STEP 4 : At any non-leaf node, merge the long_array of its two children and its y_max (as in merge sort).

2.3 Pseudo Code :

```
subtree_points(node n, x1, x2, y_bottom, flag)
{
    for i in n.long_array:
        if $i.y < y\_bottom$ :
            print_points($*$ i.pointer, x1, x2, y_bottom)
}

print_points(node n, x1, x2, y_bottom, flag)
{
    if $flag == 1$: //intermediate node
        for i in n.short_array :
            if(i < y_bottom) return;
            else print i
    else //boundary node of the required range
        for i in nth chunk in original array :
            if(x1 <= i.x <= x2 && i.y >= y_bottom) print i
}
```

root is the head of preprocessed binary tree

```
query($x1, x2$ , y_bottom, root)
    if x2 < root.x_min
        query($x1, x2$ , y_bottom, root.left)
    else if x1 > root.x_max
        query($x1, x2$ , y_bottom, root.right)
    else if (x1, x2 belongs root)
        print_points(root, x1, x2, y_bottom, 0)
    else if (x1 belongs root)
        print_points(root, x1, x2, y_bottom, 1)
        binary_search(x2, right, y_bottom, root.right)
    else if (x2 belongs root)
        print_points(root, x1, x2, y_bottom, 1)
        binary_search(x1, left, y_bottom, root.left)
    else
        binary_search(x1,left, y_bottom, root.left)
        binary_search(x2,right, y_bottom, root.right)
```

```
binary_search ($x$, type, y_bottom, root)
    if(type == "left")
        if(x1 < root.x_min)
            subtree_points(root.right, x1, infinite, y_bottom, 0)
            binary($x$, left, y_bottom, root.left)
        else if (x1 belongs root)
            print_points(root, x1, infinite, y_bottom, 0)
        else binary_search ($x$, right, y_bottom, root.right)
    else
        repeat similarly for $type == "right"$
```

2.4 Space Complexity :

The data structure we invented, is a BST of size $N * (augmentation\_size)$. Therefore, space used is - (Refer Sub section Data Structure Design)

- Due to x_{min},x_{max} and flag augmentation - $N * O(1)$

- Due to short_array augmentation - $N * Log(n)$

- Due to long_array augmentation - at every level in the tree this augmentation takes N space, implying a total of - $N * Log(n)$

- Thus, overall space complexity is =
  $O(N * Log(n)) = O(n)$ as $N = n/log(n)$.

2.5 Time Complexity :

2.5.1 Query Time :

- Locating the x_1 and x_2 in our BST takes O(Log(n)) time each.
- As we have *long_array* to skip all those nodes which don't have any point in range, we precisely traverse only to those nodes which can yield us atleast one point that is to be reported. Moreover, in each such node, we traverse the *short_array* from max to till $>= y_{bottom}$ is satisfied and this keeps our search within the bounds of O(k).
- Hence, overall time for a query is O(k+Log(n)).

2.5.2 Pre-processing Time :

- The first sort based on x coordinates requires O(n*Log(n)).

- The construction of BST with all its augmentation requires -
$T(n) = 2 * T(n/2) + an$ // O(n) to take care of the merging of the augmentations as described above
= O(n*Log(n))
- Hence, overall pre-processing requires O(n*log(n)).

2.6 Proof of Correctness

2.6.1 What is to be proved?

- Every point in the given range of (x_1, y_1) and (x_2, y_2) is getting reported.
- Every reported point is in the given range.

2.6.2 Explanation

- The array is getting sorted initially on the basis of x-coordinates and the chunks of $log(n)$ size are formed from this sorted array. The BST is constructed on these chunks based on their increasing x-range and hence, from the correctness of BST functioning, we are assured to get the correct x range when we traverse the BST on its basis.
- From the augmentation of the BST, we see that every $short\_array$ is sorted within on the basis of y-coordinates. Hence, on traversing it from max to min order till $y >= y_{bottom}$ ensures that we report points only within range.
- No point in the given range will get missed out as we do a search on the BST for x range (similar to one discussed in class) and as we traverse a sorted $short\_array$ till we satisfy $y >= y_{bottom}$.
- Hence, this proves the correctness of the algorithm.

3 Constraint of each commando

3.1 Overview

This problem is approached using the divide and conquer strategy. As the square dimension is in powers of 2, we can divide the square we get in every iteration into 4 squares of equal size. Let 'P' = p(x,y) be the coordinates of Prime Minster's cabin. Define a recursive function Report(Square,n,P) which works by divide and conquer. Where 'Square' gives the square boundaries and 'n' gives its side length.

3.2 Algorithm

- Start.
- **if(**$n < 2$**)** return with reporting "No points"
- **else if(**$n == 2$**)** return position of commando as the one diametrically opposite to P in the square. Orientation will be facing P so that he can cover the L-shape leaving out Prime minister's office.
- **else** divide the square into four equal squares of side length n/2. P_1, P_2, P_3 and P_4 are estimated as follows. The quadrant that has P currently will retain it as its P_1 and this square is called $Square_1$. The other three quadrants will have their P_i to be the point that is diametrically opposite to the only corner that they have of the bigger square. where $i \in \{2, 3, 4\}$.
- Report(n,P,Square) = Report(n/2,P_1,$Square_1$) + Report(n/2,P_2,$Square_2$) + Report(n/2,P_3,$Square_3$) + Report(n/2,P_4,$Square_4$) + the commando position and orientation who can cover P_2, P_3 and P_4.
- Stop.

3.3 Pseudo Code

```
Report(n,P)
{
        if($n < 2$)
                print "No Commandos required". return

        else if($n == 2$)
                Return Position = diametrically opposite point to P.
                and Orientation = direction facing P. return

        else
                $P_1$ = P and $Square_1$ is the corresponding Square.
                Find $Square_i$ and $P_i$ as defined above.where $i \in \{2, 3, 4\}$.
Report(n,P,Square) = Report(n/2,$P_1$,$Square_1$) +
Report(n/2,$P_2$,$Square_2$) + Report(n/2,$P_3$,$Square_3$) +
Report(n/2,$P_4$,$Square_4$) + the commando position and orientation who
can cover $P_2$, $P_3$ and $P_4$.

    return
}
```

3.4 Time Complexity

$$\begin{gathered} T(n) = 4 * T(n/2) + a \\ = 4^{log(n)} + constant \\ = O(n^2) \end{gathered}$$

3.5 Proof of Correctness

3.5.1 What is to be proved? or Claim

That given any square of side length 'n' (a power of 2) and Prime Minister's office P we can exhaust the rest of the square with non overlapping pieces of L-shaped tiles (L-shape containes 3 unit sqaures that can be guarded by a commando and hence is equivalent to the given problem).

3.5.2 Proof by Induction

Induction is carried out on the length of the square.

3.5.3 Base Cases

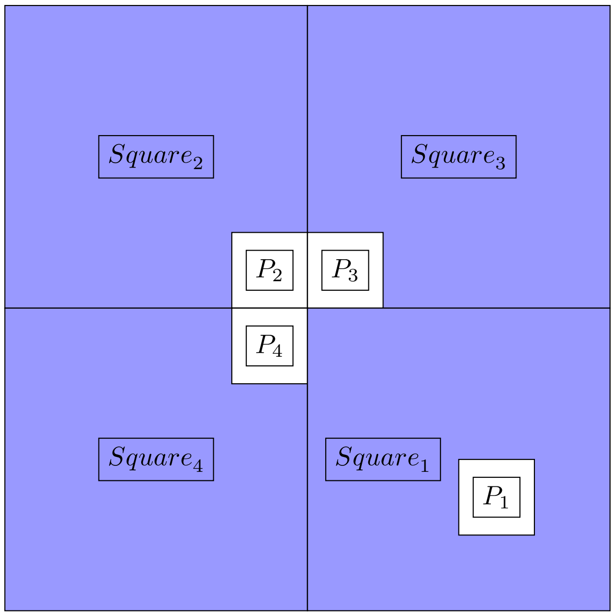
- $(n < 2)$ - No commandos needed.
- $(n == 2)$ - Only one commando needed and he should be placed so as to cover 'L' shape that leaves out Prime Minister's cabin.

3.5.4 Hypothesis

Let $n = 2^k$.
Let us assume that the Claim is true for all $i <= k$ where $k > 1$ and both are integers.

3.5.5 Inductive Step

Let us prove that claim for $k+1$ is true.
The square of size 2^{k+1} can be broken into four squares of size 2^k. As the claim holds for sizes $<= k$, these 4 squares can be exhausted in the required way, (i.e) PM cabin anywhere we chose it to be and the rest exhausted with non overlapping 'L' shapes.



Then chose, the PM cabin's as shown above. This way just by adding one more commando, on p_2 facing towards $Square_1$, we can exhaust our required full square!
Thus, proved by induction.
Note, this has to be an optimal solution as we are ensuring that every box is guarded by just one commando (this is because of the combine step which does not disturb any guarded box and the base case for which the claim holds).