CS345: Algorithms - II
Assignment 1 Solutions

Aditi Goyal - 190057
Yatharth Goswami - 191178

August 21, 2021

I Problem 1 Solution

In this problem, we were asked to devise a more optimized algorithm for computing a set of non dominated points in two dimensions.

1.1 Pseudocode for finding non-dominated points

Algorithm 1: Algorithm to find non-dominated points

```
Input: Set of $n$ distinct points in a plane, $S$
Output: Set of non dominated points present in $S$
Function FindPoints($S$):
    if $|S| \le 1$ then
        return S
    $x_m \leftarrow$ XMedian($S$)                      ▷ x median of points in $S$
    $S_L \leftarrow \{\}$
    $S_R \leftarrow \{\}$
    $y_{max} \leftarrow$ -INF        ▷ Initialising with a large negative value
    $p \leftarrow$ Variable to store the point to be removed
    for point in $S$ do
        if point.x $\ge x_m$ then
            $S_R \leftarrow S_R \cup$ {point}
            if point.y $\ge y_{max}$ then
                $y_{max} \leftarrow$ point.y
                $p \leftarrow$ point
        else
            $S_L \leftarrow S_L \cup$ {point}
    for point in $S_L$ do
        if $p$ dominates point then
            $S_L \leftarrow S_L \setminus$ {point}
    for point in $S_R$ do
        if $p$ dominates point then
            $S_R \leftarrow S_R \setminus$ {point}
    $S_R \leftarrow S_R \setminus \{p\}$
    $R \leftarrow$ FINDPOINTS($S_R$)
    $L \leftarrow$ FINDPOINTS($S_L$)
    return $R \cup L \cup$ {point}
```

1.2 Assumptions

1. For any 2 points $(x_1, y_1), (x_2, y_2) \in S$, $x_1 \neq x_2$ and $y_1 \neq y_2$.
2. On any call of the function FindPoints, there is at most one point in S which lies on the x median of all the points in S.

1.3 Overview of Algorithm

We have used the divide and conquer algorithm described in the class, but with a slight modification. The problem with the naive divide and conquer algorithm was that not all of the recursive sub-procedures were spent in finding the non-dominated points of the original problem. We can see that some of the were returning the set of points which were not actually non-dominated in the original problem.

We tried to resolve this issue in the naive algorithm by trying to find at least one non-dominated point of the original problem in every recursive call. We used the fact that on splitting the problem into two parts the point with the maximum y-coordinate in the right half will always be a non-dominated point of the original problem. Hence, removing this point and the set of points dominated by it and continuing solving the sub-problems ensures that we remove at least one non-dominated point of the original problem in each recursive call.

Now, since there are 2^i recursive calls at the i^{th} level of the recursion tree, this will give us a bound of $\log h$ on the total recursive depth. This argument is still not completely concrete and hence the detailed proof of correctness and time complexity is provided below.

1.4 Proof of correctness

For establishing the correctness of the algorithm, we need to show that we are getting all the non-dominated points of the original problem and no extra point. Let's try to prove this as a claim.

Claim 1.4.1. *Algorithm 1 returns all the non-dominated points from the original set of points and no extra point.*

Proof. Our algorithm first divides the original problem into two halves based on the x-median and then selects the point with the maximum y-coordinate as one of the non-dominated points. We claim that this point is definitely a non-dominated point of the original problem. This is easy to see since every point to the right of this (with greater x-coordinate) has a lower y-coordinate and every left to this point has a lower x-coordinate. Thus, we can safely add this point to our set of non-dominated points and remove all the points which are dominated by this point.

Next, we claim that after removing the points dominated by the maximum y-coordinate point on the right side, the points on the left side and right side essentially become independent of each other in the sense that no other point in the left side can be dominated by a point in the right side any more. This is true, since every point which is remaining on the left side has a y-coordinate higher than any point on the right side and hence can't be dominated. This observation leads us to the fact that solving the sub-problem for the left and right half and then merely taking their union will suffice, since any non-dominated point found for the right half cannot be dominated by some point on the left half and any non-dominated point found on the left side can't be dominated by any point on the right side.

From the last claim, we can say that any non-dominated found in the left or right half will also be a non-dominated point of the original sub-problem. Hence, we can prove the second part of the claim 1.4.1 that we don't return dominated point from the function. Now, all what is left is to show that it returns all of the non-dominated points present in the original problem. This part is also quite easy to prove (though not totally trivial). We will prove this by using the tool of proof by contradiction. Assume on contradiction that there exists a non-dominating point (say P) but is not returned by our algorithm. The only way this is possible is when we don't reach a part containing P. But since the divide and conquer paradigm is proved to explore the complete domain containing the points, therefore the only way it leaves out some point unexplored is if that point was already removed while removing the dominated points of a non-dominated point. Since, we have already showed above that the points removed in every iteration are the ones dominated by a non-dominated point of the original problem, therefore P is dominated by some other point and hence not a non-dominating point. Therefore, we reach a contradiction and hence our algorithm will return all of the non-dominated points.

Talking about the correctness of the base case, which is trivially true. In the case of just one point remaining, we just return that point as that is non-dominated by definition. Hence, we showed that our algorithm returns all the non-dominated points and no more. □

Now we will prove another small claim, which is not important for correctness but important for future analysis.

Claim 1.4.2. *In every recursive call inside Algorithm1 we return at least one non-dominated point from the original problem.*

Proof. This is fairly straightforward to see, since inside each recursive procedure we are inserting at least one point in the set of non-dominated points (point with maximum y-coordinate on the right side) which is also a non-dominated point of the original problem as shown in 1.4.1. The only case when we don't do this is when there is no point to the right side of the median line which is only possible if the size of set of points is 1, which is readily handled by the base case. Hence, proved. □

1.5 Time complexity analysis

- Let $|S| = n$ and let the number of non-dominated points in $S = h$.
- Lines 2-8 take $\mathcal{O}(n)$ time because x median can be found in $\mathcal{O}(n)$ time, rest is initialization.
- Lines 9-16 take $\mathcal{O}(n)$ time because every point in S is encountered exactly once in the loop.
- S_L and S_R are subsets of S and we are iterating over the elements is them. So lines 17-22 take $\mathcal{O}(n)$ time.
- Lines 24 and 25 are recursive calls. Since, S is divided based on x median, we have $|S_L| \le \frac{n}{2}$ and $|S_R| \le \frac{n}{2}$.

- Line 26 takes $\mathcal{O}(h)$ time since it is the union of all non-dominated points.
- Summing up over the above mentioned time complexities, any call to the function FindPoints(S) has $\mathcal{O}(|S|)$ cost associated with it other than the recursive calls to S_L and S_R.
- Consider the recursion tree for the function FindPoints. There will be $\mathcal{O}(h)$ nodes in this tree because each recursive call returns a distinct non-dominated point (called p in the function). The point is distinct because in the further recursive calls, it is removed from S_R.
- A node at level i (from the top) in the binary tree represents a function call for a set of size at most $\frac{n}{2^i}$ (since the recursive calls to S_L and S_R are obey the inequalities $|S_L| \leq \frac{n}{2}$ and $|S_R| \leq \frac{n}{2}$).
- The total time taken by the function is the sum of costs associated with all the nodes of the tree. The total number of nodes in constant ($\mathcal{O}(h)$) and cost associated with a node decreases as we increase its level from the top.

  Claim 1.5.1. *There is a worst case with respect to time complexity where in the recursion tree, every level, except possibly the last, is completely filled.*

  Proof. We prove this by contradiction. Assume that in all worst cases, more than one level is not completely filled. Consider any worst case. By our assumption, it has at least two levels which are not completely filled. Therefore, there is node (let us call it N) at level i which does not have a child (without loss of generality, assume left) and there is another node (let us call it M) at level $i+1$ which is not a leaf node. Without loss of generality, assume that the left child (let us call it M_L) of M is present.

  Consider another tree in which N has left child as the subtree rooted at M_L, M has no left child and other relations between nodes remain same as before. The new tree has a time cost that is greater than or equal to the original tree's cost because each node in subtree rooted at M_L moved up by 1 level (thus doubling their associated cost), while for the rest of the nodes, the associated cost remains same. We can continue this till the tree becomes complete.

  Thus, assuming that the tree was not complete, we created another tree which has higher time cost. $\square$

- Height of this complete binary tree will be $\mathcal{O}(\log_2 h)$ since there are h nodes. Cost at each level of the tree is $\mathcal{O}(2^i.\frac{n}{2^i}) = \mathcal{O}(n)$ since there are 2^i nodes at level i each of which has a cost $\mathcal{O}(\frac{n}{2^i})$.
- Total cost is $\mathcal{O}(n \log_2 h)$ (levels = $\mathcal{O}(\log_2 h)$, cost at each level = $\mathcal{O}(n)$).