IIT KANPUR

CS345A - ALGORITHMSII

Assignment 1

Anjani Kumar
11101

Sumedh Masulkar
11736

January 12, 2014

Contents

1 Non Dominated Points

1.1 $O(nlogn)$ algorithm for non-dominated points in a plane.

Overview of algorithm Given set of points P in a plane.
Divide Step:

1. If there is only one point in a plane, the point itself is non-dominated set of points. Hence, return P.

2. If there are two points, if any point has both x and y co-ordinates both greater than that of other point, then return that point, else return P.

3. Else find the x-median of the points and divide the plane into left half plane and right half plane using the median. And now call the function for both the half planes.

Conquer Step:
Goal: Given the non-dominated points of the two half planes, merge the solution of smaller parts to get the solution of the bigger plane.

Assuming we have two sets of points P_1 and P_2, where P_1 is the set of non-dominated points of the left plane, and P_2 is the set of non-dominated points of the right plane respectively.

The x-coordinate of all the points in right plane are obviously greater than x-coordinate of all the points in the left plane. Thus, we only need to eliminate points from P_1 that are dominated by points in P_2.
Since x-coordinate of points in P_2 is always greater, we only need to look for the y-coordinates.
Let y be the point in P_2 with maximum y-coordinate. Then the dominated points in P_1 are all the points whose y-coordinates are less than y-coordinate of y.
Thus the solution of the plane will be { P_1 - {points in P_1 with y-coordinate $<$ y } } $\cup P_2$.

Pseudo-Code.

```
NonDominatedPts(set of points P){
        //Returns set of non dominated points from P.
if |P|==1 then
    return P;
else if |P|==2 then
    let p1 and p2 be the two points in P;
    if x1 > x2 and y1 > y2 then
        return {p1};        //p1=(x1,y1), and p2=(x2,y2)
    else if x1 < x2 and y1 < y2 then
        return {p2};
    else
        return P;
else
    p* ← x-median(P);                                          //c1 n
    (L,R) ← split(P, p*);                                      //c2 n
    P1 ← NonDominatedPts(L);                                   //T(n/2)
    P2 ← NonDominatedPts(R);                                   //T(n/2)
    P1 ← P1 sorted along y-axis;                               //c3 n
    y ← max y-coordinate in points of P2;                      //c4 n
    P1 ← P1 - {all points in P1 whose y-coordinate ≤ y};       //c5 n
    return (P1 ∪ P2);
}
```

Algorithm 1: $O(nlogn)$ algorithm to find Non Dominated Points

Time Complexity: $O(n)$ + 2T(n/2) = $O(nlogn)$.

Proof of Correctness:

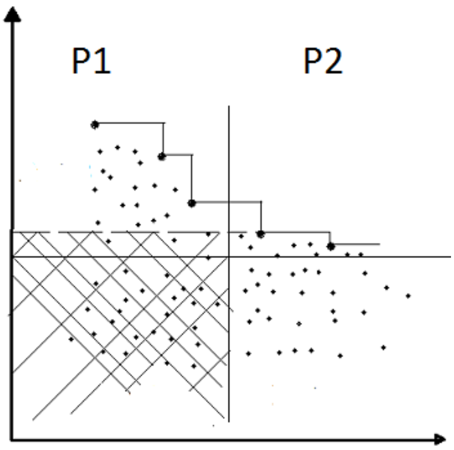

Figure 1: Figure for non-dominated pts

Proof by induction on size of set P.

- If(|P|=1), the point is non-dominated, hence the set P should be returned.
- If(|P|=2), Remove dominanted point if exists, and return P.
- **Induction Hypothesis:** The solution of both half planes, P_1 and P_2 are known, *i.e.* now P_1 and P_2 consist only of non-dominated points among themselves. And we need to merge their solution to get solution of P.
  As shown in figure, the points in P_2 cannot be dominated by any point in P_2(according to induction hypothesis) or P_1(since x-coordinate of any point in P_2 is greater than x-coordinate of any point in P_1).
  See figure(1).
  If p* is the point with maximum y-coordinate in P_2, no point in P_1 which has y-coordinate greater than that of p* can be dominated by any point in P_1(by induction hypothesis) or P_2(since y-coordinate is greater than that of p*).
  Thus, the group of points in P_1 that have y-coordinate less than that of p* are dominated(x-coordinate is also less).
  Hence, we need to remove this group of points from P_1 and remaining $P_1 \cup P_2$ will be the answer.
  Thus, the algorithm is correct.

1.2 $O(nlogh)$ algorithm for non-dominated points in a plane

Idea

The solution to this problem is similar to that of algorithm 1, with a slight modification. Here, we are removing the dominated group of points from P_1 (see fig. 1), before we find non-dominated points of P_1, hence reducing the number of points NonDominatedPts(P_1) returns(h_1), in the following analysis.

Time Complexity Analysis:

.. $T(n,h) = an + T(n/2, h_1) + T(n/2, h_2)$ $//h_1 + h_2 =$ h
Induction Hypothesis: $T(n_o, h_o) \leq$ cn_o log h_o + b. (for some $n_o < n$)

Induction Step: $T(n,h) \leq$ an + c($\frac{n}{2}$) log(h_1) + c($\frac{n}{2}$) log(h_2) + 2b.

$$\leq \text{a}n + \text{c}(\tfrac{n}{2}) \log(h_1 * h_2) + 2\text{b}.$$

Using AM $\geq$ GM, $h_1 * h_2 \leq (\frac{h}{2})^2$.

$$T(n,h) \leq An + cn^{*}log(\tfrac{h}{2}) + 2b$$

$$\leq (cn^{*}logn + b) + (an + b - cn^{*}log2)$$
$$(\leq 0 \text{ for some c} > \text{a+b})$$

$$T(n,h) \leq cn^{*}log(h) + b.$$

Therefore, $T(n,h) = O(nlog(h))$.

Pseudo-Code.

```
NonDominatedPts(set of points P){
        //Returns set of non dominated points from P.
begin
    if |P|==1 then
        return P;
    else if |P|==2 then
        let p_1 and p_2 be the two points in P;
        if x_1 > x_2 and y_1 > y_2 then
            return {p_1};       //p_1=(x_1,y_1), and p_2=(x_2,y_2)

        else if x_1 < x_2 and y_1 < y_2 then
            return {p_2};
        else
            return P;

    else
        p* ← x-median(P);
        (L,R) ← split(P, p*);
        L ← L sorted along y-axis;
        y ← max y-coordinate in points of R;
        L ← L - {all points in L whose y-coordinate ≤ y} ;
        P_1 ← NonDominatedPts(L);
        P_2 ← NonDominatedPts(R);
        return (P_1 ∪ P_2) ;

end
}
```

Algorithm 2: $O(nlogh)$ algorithm to find Non Dominated Points

1.3 $O(ilogi)$ algorithm to maintain non-dominated points in online fashion

Introduction:

Data Structure used: Height Balanced Binary Search Tree (T).
Apart from value(*i.e.* x-coordinate), left, right, we will augment tree with another pointer **predecessor** that stores the predecessor of the node in the tree T(*i.e.* the node which has x-coordinate just before this node).

Functions used:

1. Insert(p_{new}, T): It is used to insert p_{new} in BST tree T such that T remains height balanced.
2. FindSuccessor(): Used to find the successor of the point before it is inserted into BST T.
   We will just need to follow the path where the point will be inserted, initialise successor to null, whenever we move right from a node, this cannot be the successor, so ignore this node. Whenever we move to left of a node, this can be a possible successor, hence update successor. When we reach the position where this new node will be inserted, the value of successor is returned.
   Time Complexity: O(log i).

Idea (What We are trying to do?)

- We are placing the NonDominated(ND) points in a BST(height balanced) according to x-coordinate as key of BST.
- When a new point, p_{new} is inserted in the graph, its y-coordinate is compared with its successor for being a ND point.
  - If its y-coordinate is less than y-coordinate of its successor, then simply ignore the new point.
  - Else if its y-coordinate is greater than its successor, then it is a non-dominated point as well as the successor is non-dominated. Hence, insert p_{new}.
- Subsequently, if the point p_{new} passes the comparison test with its successor s, after inserting p_{new}, all other ND-Points, existing above the successor of p_{new} in the tree(with x-coordinate less than x-coordinate

of p_{new}), need to be compared with p_{new} to check if they still are non-dominanted.

- Those who fail the test are deleted from the tree.
- The test stops if we reach the top of the ND staircase or a ND Point successfully passes the comparison test with point P (note if a point is not deleted, then none of its predecessors need to be deleted anymore).

Time Complexity: $O(ilogi)$.
At each insertion of points into the graph, points will be either inserted or deleted from the tree which takes $O(logi)$ time, Therefore overall time complexity for i steps of insertion is $O(ilogi)$.

Pseudo Code

```
AddNewPoint(p_new, T){
begin
    if T== NULL then
        Insert(p_new,T);
        p_new →predecessor = NULL;
    else
        successor = FindSuccessor(p_new);
        if successor<> NULL and successor→y < p_new →y then
            Insert(p_new, T);
            temp1 = successor→predecessor;
            successor→predecessor = p_new;
            while temp1 <> NULL and temp1→ y < p_new →y do
                temp2 = temp1→predecessor;
                Delete temp1;
                temp1 = temp2;
            end
            P→predecessor = temp1;
        else
            //Ignore p_new.
        end
    end

end
}
```

Algorithm 3: $O(ilogi)$ algorithm to maintain non dominated points

Example

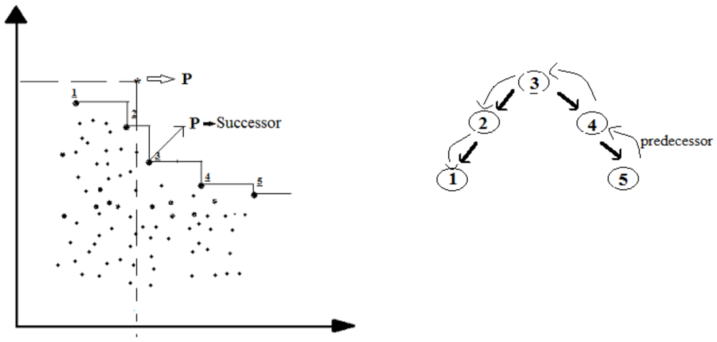

Figure 2: Case 1

Case 1:
In this case, P passes its comparison test with its successor (3) but 1 and 2 fail their comparison test with P because Ps x and y coordinates are both high compared to them. Therefore points 1 and 2 will be deleted from the tree T and point 3s predecessor will be P.

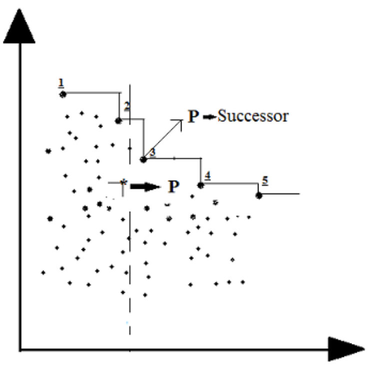

Figure 3: Case 2

Case 2:
In this case where P's y coordinate is less compared to its successor (3), it fails to become a ND Point and therefore, the tree remains as it was before the insertion of P into the plane.

1.4 Non Dominated Points in a 3D plane

Introduction:
In the case where the points are arranged in a 3D space, the solution for calculating all the non-dominant points will remain similar to algorithm 3 with few modifications.

Data Structure used: Height Balanced Binary Search Tree(T).

Functions used:

1. FindSuccessor(p_{new}): Used to find the successor of the point before it is inserted into BST T.(Same as that in 1(c).)
2. Insert(p_{new},T): It is used to insert a node into tree T such that it remains balanced.
3. Both these functions take O(log n) time, where n is the size of tree T.

Note here only the left and right child pointer in the nodes of T need to be stored. Predecessor pointer is not needed.

Idea(what we are trying to do:)

- We look at the 3D plane from a particular axis(z) *i.e.* the points are first sorted according to decreasing order of their z-coordinates.
- The ND Points are stored in tree T according to increasing order of their x-coordinate.
- As we move from the top of the z axis, the point p_{new}s successor(in the tree T) is computed and its y-coordinate is compared with p_{new}.
- This is the only comparison test and the previously computed ND Points are not deleted because their z-coordinate is already higher compared to the new points being checked.

Time complexity: In this algorithm, a point insertion, deletion and successor function takes logarithmic time.
Also sorting aross z axis also takes **O(n log n)** time. Therefore the overall time complexity would be **O(n log n)**.

Pseudo Code:
BuildNonDominatedPtsTree(){

```
begin
    Sort points according to decreasing order of z
    coordinates.          //O(nlogn).
    for p_new ← p_1 to p_n do
                          //O(nlogn).
        if T==NULL then
            Insert(p_new,T);
        else
            successor = FindSuccessor(p_new);
            if successor <> NULL and successor→y < p_new →y then
                Insert(p_new,T);
            else
                //Ignore p_new.
            end
        end
    end
end
```

}
Algorithm 4: O(n log n) Algorithm to find Non Dominated Points in 3D plane.

2 A computational problem of experimental physicist

The total force on i_{th} particle is given as:

$$F_j = \Sigma_{i<j} \frac{Cq_iq_j}{(j-i)^2} - \Sigma_{i>j} \frac{Cq_iq_j}{(j-i)^2}$$

where the first Σ denotes forces from the left, and the second Σ denotes forces from the right.

This calculation of the terms takes $O(n)$ for each j. Thus making the time complexity of the program $O(n^2)$.
If we can calculate these summation faster, we can calculate the force efficiently.
Force on q_2:
$F_2 = \Sigma_{i<2} \frac{Cq_iq_2}{(2-i)^2} - \Sigma_{i>2} \frac{Cq_iq_2}{(2-i)^2} = \frac{Cq_1q_2}{1^2} - \frac{Cq_3q_2}{1^2} - \frac{Cq_4q_2}{(4-2)^2} - \ldots$

Using polynomials,

$$A(x) = q_1 + q_2x + q_3x^2 + \ldots + q_nx^{n-1}$$
$$= \Sigma_{i=1}^{n} q_i x^{i-1}.$$

$$B(x) = q_n + q_{n-1}x + q_{n-2}x^2 + \ldots + q_1x^{n-1}$$
$$= \Sigma_{i=n}^{1} q_i x^{n-i}.$$

$$C(x) = (\frac{1}{1^2}) + (\frac{1}{2^2})x + (\frac{1}{3^2})x^2 + \ldots + (\frac{1}{(n)^2})x^{n-1}$$
$$= \Sigma_{i=1}^{n} (\frac{1}{i^2}) x^{i-1}.$$

$$D(x) = A(x) * C(x)$$

$$= \frac{q_1}{1^2} + [(\frac{q_1}{2^2}) + (\frac{q_2}{1^2})]x + [(\frac{q_1}{3^2}) + (\frac{q_2}{2^2}) + (\frac{q_3}{1^2})]x^2 + \ldots + [(\frac{q_1}{n-1^2}) + (\frac{q_2}{n-2^2}) + \ldots + (\frac{q_{n-1}}{n-1^2})]x^{n-2} + \frac{q_n}{n^2}x^{2n-2}$$

$$E(x) = B(x) * C(x)$$

$$= \frac{q_n}{1^2} + [(\frac{q_n}{2^2}) + (\frac{q_{n-1}}{1^2})]x + [(\frac{q_n}{3^2}) + (\frac{q_{n-1}}{2^2}) + (\frac{q_{n-2}}{1^2})]x^2 + \ldots + [(\frac{q_n}{n-1^2}) + (\frac{q_{n-1}}{n-2^2}) + \ldots + (\frac{q_2}{1^2})]x^{n-2} + \frac{q_1}{n^2}x^{2n-2}$$

Observation: The coefficient of x^0 in $D(x)$ is the force from the left on q_2 divided by C*q_2, coefficient of x^1 is the force from the left on q_3 divided by C*q_3, and hence from left on q_i^{th} particle is coefficient of x^{i-1}(i is from 1 to n), divided by C*q_i.

And the coefficient of x^0 in $E(x)$ is the force on the $(n-1)^{th}$ particle from the right, divided by C*q_{n-1}, and the coefficient of x^1 is the force on the $(n-2)^{th}$ particle from the right divided by C*q_{n-2}, and so on. The $(n-2)^{th}$ term (coefficient of x^{n-2}) is the force on q_1 from the right divided by C*q_1. Hence i^{th} term upto $(n-2)^{th}$ terms is the force from right on q_{n-i}^{th} particle, divided by C*q_{n-i}.

Points to note:

- This also covers the corner cases, *i.e.* force from left on 1^{st} particle is coefficient of C*q_1* coefficient of x^{-1}, *i.e.* 0.
- Similarly, force on q_n from right is coefficient of x^{-1}, *i.e.* 0.
- We only need to consider coefficient of terms upto x^{n-2}. The terms coming later are not relevant to this algorithm. Hence, we can discard them or rather not compute them.
- The Product function used in following pseudo code is the polynomial multiplication taught in class, that takes $O(nlogn)$ time to multiply two polynomials. We will assume this function returns the product polynomial in an array, with index i from 0 to n-1(note we assumed i from 1 to n above).

Proof of correctness:

Looking at the general term of $D(x)$ and $E(x)$, force on particle q_j as given by the algorithm will be:

$$C * q_j * [(\Sigma_{i=1}^{j-1}(\frac{q_i}{(j-i)^2})) - (\Sigma_{i=j+1}^{n}(\frac{q_i}{(j-i)^2}))].$$

which is equal to F_j as given in the formula.
Hence, the algorithm is correct.

Pseudo Code

```
Output F_j(array q_i[ ]){
begin
    A(x) ← q_1 + q_2 x + q_3 x^2 + ... + q_n x^(n-1) ;
    B(x) ← q_n + q_(n-1) x + q_(n-2) x^2 + ... + q_1 x^(n-1) ;
    C(x) ← (1/1^2) + (1/2^2)x^2 + (1/3^2)x^3 + ... + (1/n^2)x^(n-1);
            //The algorithm of product returns array D where element
            //D[i] is the i^th coefficient, i goes from 0 to n-1.
    D(x) ← Product(A(x), C(x));          //O(c_1 n logn)
    E(x) ← Product(B(x), C(x));          //O(c_2 n logn)
    for i ← 1 to n do
        Output C * q_i * (Coefficient(D, i-2) - Coefficient(E, n-i-1));
                //C is the coulomb's constant.
                // The Coefficient(P,i) function returns the coefficient
                //of x^i in polynomial P.
    end
    //(c_3 n) for loop
end
}

Coefficient(polynomial P, exponent i){          //O(1) time.
        //The polynomial is kept as an array.
begin
    if i < 0 then
        return 0;
    else
        return P[i];
    end
end
}
```

Algorithm 5: Pseudo Code for finding forces on particles

Time Complexity:
The text in red shows the time taken by the algorithm is $((c_1 + c_2)$ n logn $+$ c_3n), hence time complexity of the algorithm is $O(nlogn)$.