CS345 Assignment 2

Siddharth Agrawal(150716)
AviRaj(150168)

August 28, 2017

Question 1

Part 1

Algorithm

Pre-Processing

1. Compute the Convex Hull (convex polygon of smallest area enclosing a set of points) for all the points and store them in a cyclic array C_i.
2. Determine the two points in the array C with maximum and minimum x-coordinates (call them p_{max} and p_{min} respectively).
3. Choose the points from C which lie above the line joining p_{min} and p_{max}. Store all these points in an array $C_{i.up}$ in a sorted manner (according to x-coordinate).
4. Remove the points in C from our original set of points P.
5. Check if the set of points P is null.
   - `if yes:` exit.
   - `if no:` go to step 1.

Query

Algorithm 1 Algorithm to find a point in C_i which lies above L

function $UpperPoint(C_{i.up}, lo, hi, L)$
    $mid \leftarrow (lo + hi)/2$
    if $C_{i.up}[mid]$ lies above L **then**
        return $C_{i.up}[mid]$
    else if $C_{i.up}[mid + 1]$ lies above L **then**
        return $C_{i.up}[mid + 1]$
    else if $C_{i.up}[mid - 1]$ lies above L **then**
        return $C_{i.up}[mid - 1]$
    else
        Compute perpendicular distances of $C_{i.up}[mid]$, $C_{i.up}[mid + 1]$ and $C_{i.up}[mid - 1]$ from line L and call them d_{mid}, d_{mid+1} and d_{mid-1} respectively.
        if d_{mid} is least **then**
            return $null$
        else if d_{mid+1} is least **then**
            return $UpperPoint(C_{i.up}, mid + 1, hi, L)$
        else
            return $UpperPoint(C_{i.up}, lo, mid - 1, L)$
        end if
    end if
end function

Algorithm 2 Algorithm to find all the points in C_i above line L, given one point which lies above it

function $AllUpperPoints(C_i, x, U)$
    $left \leftarrow x - 1$
    $right \leftarrow x + 1$
    while $C_i[left]$ is above L **do**
        $U \leftarrow U \cup \{C_i[left - -]\}$
    end while
    while $C_i[right]$ is above L **do**
        $U \leftarrow U \cup \{C_i[right + +]\}$
    end while
end function

Let the number of convex hulls formed in the pre-processing be k' (i.e. $C_1, C_2, ..., C_{k'}$).
Let U (initially empty) denote the set containing all the points above the line L.

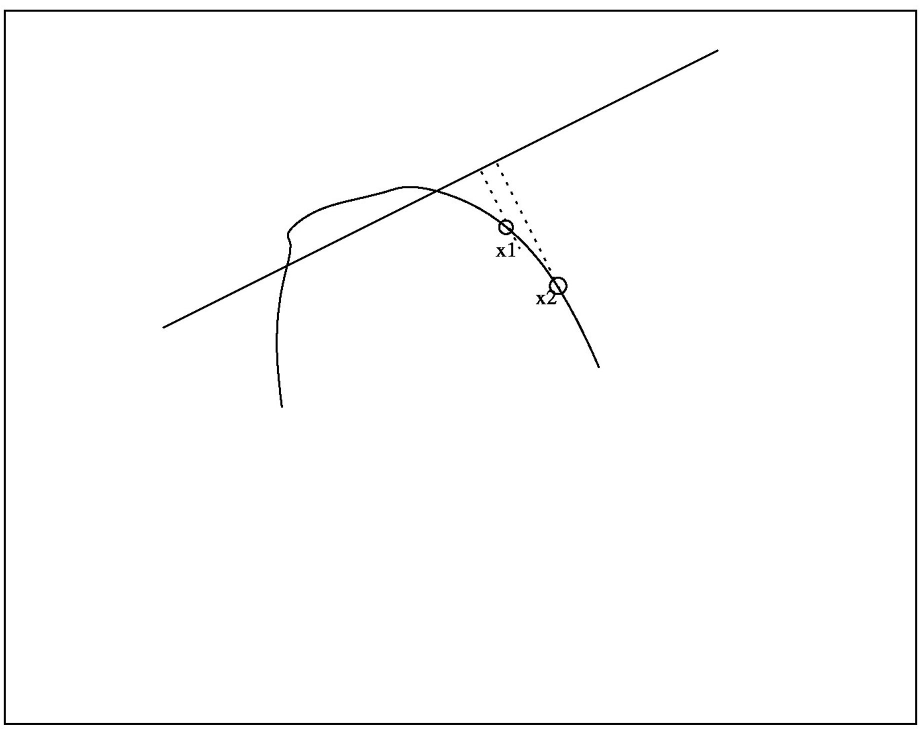
Algorithm 3 Main procedure to find the points in upper plane which uses Algorithm 1 and Algorithm 2

procedure MAIN
  $U \leftarrow \emptyset$
  for $i = 0$ to k' **do**
    $m \leftarrow$ number of points in $C_{i.up}$
    $p \leftarrow UpperPoint(C_{i.up}, 1, m, L)$ ▷ use Algorithm 1
    if $p = null$ **then**
      return U
    else
      $U \leftarrow U \cup \{p\}$
      Let x be such that $C_i[x] = p$
      $AllUpperPoints(C_i, x, U)$ ▷ use Algorithm 2
    end if
  end for
  return U
end procedure

Proof of Correctness

Note that we are dividing all the points in P to k' Convex Hulls. Then we find the upper plane points for each individual Convex Hulls.
In our main procedure Algorithm 3, we are actually just finding the upper plane points individually for each Convex Hull.
So it is suffices to prove the correctness for Algorithm 1 and Algorithm 2.

Algorithm 1
In this algorithm, we just wish to find a single point in C_i which lies above L. Here we use binary search.
`Lemma 1: Given two points on a convex line (wrt a line), the half containing the point closer to the line has a greater chance of intersecting it.`



Clearly, by definition of convexity wrt a line, the distance keeps decreasing or increasing at one side of a point till minimum distance is achieved (note we are only considering case before intersection with line). Hence, the lemma is correct since the case of zero distance (intersection) can only be seen in the side of the smaller distance from line.
Thus, using `Lemma 1`, we can safely say that there atleast exists one point in the half of the lesser distant point, which lies above L (assuming that there is a point in C_i above L). Also, notice that if the mid point is at the least distance and is below the line, then no point in C_i can lie above L.

Now, using Algorithm 1, we have found one point above line L. Using this point, we find all other points above the line using Algorithm 2.

Algorithm 2
Here, we use the cyclic property of our data structure.
So, if some points lie above L, then these are adjacent to each other in the array C_i. Thus if we know only one point in C_i which is above L, then we only need to check its adjacent points. Hence this algorithm successfully updates the upper plane points.

In our main procedure Algorithm 3, we are ensuring that we only check convex hulls which have a point above L (line 6-7).
If all the points in a convex hull are below a line, clearly all points within it are also below the line. Hence we need not check further, hence saving time.

Complexity Analysis

Space

$$\begin{aligned}
\text{Space} &= \sum_{i=1}^{k'} (|C_i| + |C_{i.up}|) \\
&\leq \sum_{i=1}^{k'} (|C_i| + |C_i|) && \because C_{i.up} \subseteq C_i \\
&= 2\sum_{i=1}^{k'} |C_i| \\
&= 2n \\
&= O(n)
\end{aligned}$$

Time

`Algorithm 1`

$$\begin{aligned}
T(n) &= O(1) + T(\frac{n}{2}) && O(1) \text{ time to find perpendicular distances} \\
\therefore T(n) &= O(nlogn)
\end{aligned}$$

`Algorithm 2`

$$T(n) = O(k_i) \qquad k_i \text{ is the number of points in } C_i \text{ above } L$$

`Main: Algorithm 3`
Note: $k' = O(k)$, where k' is the number of convex hulls and k is the total number of points above line L.

$$\begin{aligned}
\therefore T(n) &= \sum_{i=1}^{k'} (logn + k_i) \\
&= k' logn + k \\
&= O(klogn)
\end{aligned}$$

Question 3

Part 1

a) Greedy Step

We first find which two vertices are closest to each other. For this, just find $v_i, v_j \in V$ such that $d(v_i, v_j)$ is minimum.
Let the nodes be i, j respectively. Using Lemma1, they are siblings.
Let their parent node be x. Our main greedy step is to merge these two vertices v_i, v_j into one vertex v'.
Thus,

$$G' = (G\backslash\{v_i, v_j\}) \cup \{v'\}$$

$$\forall v_k \in G\backslash\{v_i, v_j\},\ d(v', v_k) = min(d(v_i, v_k), d(v_j, v_k))$$

Also, we merge nodes i, j, x in $T^*(G)$ into a new node x' which stands for the vertex v'
Then we compute T^* for G'.
After obtaining $T^*(G')$, expand it by expanding x' back into i, j, x and assign $h(x) = d(v_i, v_j)$.

b) Theorem that relates $T^*(G)$ with $T^*(G')$

`Lemma1: There exists atleast one optimal solution for which` v_i, v_j ($d(v_i, v_j)$ `is minimum) are siblings`
Assume the converse, i.e., there does not exist any tree such that v_i, v_j are siblings.
Consider v_i, v_j have LCA x (which is NOT both's parent).
Let y be the sibling of v_i. Swap y with v_j.
Then all nodes in the tree other than subtree rooted at x will remain unchanged.
In the subtree rooted at x, all nodes had label equal to $d(v_i, v_j)$ (since $d(v_i, v_j)$ is minimum and labels have to be less than or equal to it by definition). Hence labels cannot further decrease.
Contradiction!!
Hence there exists an optimal tree with v_i, v_j as siblings.

Consider G and G' as defined above.
Let $T^*(G)$ after greedy step be converted into $T'(G)$ (i.e. my merging nodes i, j, x into a new node x').
Let $T'(G')$ be the tree obtained after expanding the node x' back into i, j, x.

Note: we say that $T \geq T'$ iff $\forall v_i, v_j$ distance between v_i and v_j in T' is less than or equal to their distance in T.

`Theorem 1:` $T^*(G') \geq T'(G)$
Note, that $T'(G)$ is consistent (since $T^*(G)$ was consistent and only change occurs at a single node x').
Hence by definition, $T^*(G') \geq T'(G)$.

`Theorem 2:` $T^*(G) \geq T'(G')$ Consider T'(G') obtained after expanding $T^*(G')$. Let k be a node representing $v_k \in G\backslash v_i, v_j$.
Note, that $T'(G')$ is consistent (since $T^*(G')$ was consistent and only change occurs at a single node x').
Let a be the LCA of i, k. Therefore, a is also the LCA of j, k ($\because i, j$ are siblings, Lemma 1).

$$\because d(v', v_k) \leq min(d(v_i, v_k), d(v_j, v_k))$$
$$\therefore in\ T'(G')$$
$$h(a) \leq d(v_i, v_k)$$
$$\&\ h(a) \leq d(v_j, v_k)$$

Hence, we see that $T^*(G) \geq T'(G')$.

From the above two theorems, we see that our greedy strategy gives us the optimal solution.

Implementation

We store all the edges (v_i, v_j) in a heap H.

function $T_{opt}(H)$
    if $|H| = 2$ **then**
        return tree with two siblings (v_i, v_j) and parent x storing the label $h(x)$ as $d(v_i, v_j)$
    else
        $v_i, v_j \leftarrow Extract(H)$
        Remove all edges from H which contain either v_i or v_j
        $Insert(H, (v', v_k))\ \forall v_k \in G \backslash \{v_i, v_j\}$ $\triangleright$ $d(v', v_k) = min(d(v_i, v_k), d(v_j, v_k))$
        $T \leftarrow T_{opt}(H)$
        Replace node v' in T by the subtree with two siblings (v_i, v_j) and parent x storing the label $h(x)$ as $d(v_i, v_j)$
        return T
    end if
end function

Time Analysis:
For one iteration,
$O(1)$ time for extracting min-edge from heap H,
$O(nlogn)$ time for deleting edges in heap H and
$O(nlogn)$ time for inserting edges in heap H.
Also, there are $n - 1$ iterations.
Thus,

$$T(n) = O(n^2 logn)$$

QUESTION 2.1

Data structure used – Red Black Tree with an additonal field min.

(Min(v) + ancestor) increments equals the value of min element in subtree(v)

INSERT (D, i, x)

{

if (T = NULL)

Create a new node u

val (u) = x

size (u) = 1

incr (u) = 0

min (u) = x

left (u) =NULL

right (u) =NULL

Balance the tree height

return u

else

Size (T) = Size (T) + 1

x = x – incr (T)

if (min (T) > x) min (T) = x

if (left (T) = NULL) s = 0

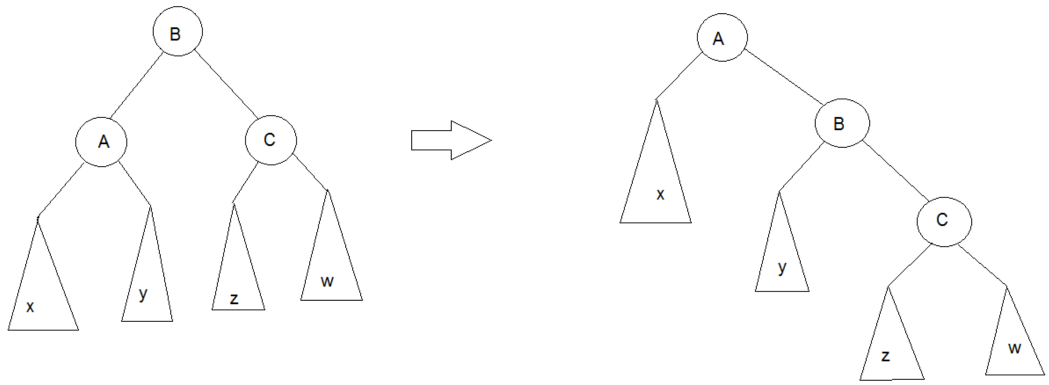
else s = size (left (T))

if (I ≤ s + 1) left (T) = INSERT (left (T) , i, x)

else right (T) = INSERT (right(T) , i –s-1 , x)

return T

}



Taking care of incr ( ) , size ( ) and min ( ) fields in Right-Rotate (B)

Before rotation DO

If (right (A) <> NULL)

incr (right (A)) = incr (right(A)) + incr (A)

incr (A) = incr (A) +incr (B)

incr (B) =incr (B) - incr (A)

After rotation DO

min (B) = minimum (min (left (B) + incr (left(B) , min (right(B)) + incr(right(B)) , val (B)

min (A) = minimum (min (left (A) + incr (left(A) , min (right(A)) + incr(right(A)) , val (A)

Size (B) = size (left (B)) + size (right (B) + 1

Size (A) = size (left (A) + size (right (A)) + 1

Assumptions: min (NULL) = 0

incr (NULL) = 0

size (NULL) = 0

There will be a similar process for left rotate also.

```
DELETE ( T , i)
{
    if (T=NULL)  return
    Let u be the node storing the ith element
    if (left(u) <> NULL & right(u) = NULL)
        if (right(parent(u)) = u )  right(parent(u) = left(u)
        else  left(parent(u)) = left(u)
        incr(left(U)) += incr(u)
    else if (left(u)=NULL & right(u)=NULL)
        if (right(parent(u)) = u )  right(parent(u)) = NULL
        else  left(parent(u) = NULL
    else if (left(u)=NULL & right(u) != NULL)
        if (right(parent(u)) = u )  right(parent(u) ← right(u)
        else left(parent(u)) ← right(u)
        incr(right(U)) += incr(u)
    else
        v = right(u)
        while (left(v) <> NULL )
            size(v) = size (v) -1
            v = left(v)
        w= v
        parent(v) ← NULL
        while (v!=u)
            val(w) = val(w) + incr(v)
            min(v) = minimum of (val(v),min(left(v))+incr(left(v)),
            min(right(u))+incr(right(v)))
            v = parent(v)
        val(u) = val(w);
```

while (u <>T)

size(u) = size(u)-1

min(u) = minimum of (val(v),min(left(v))+incr(left(v)), min(right(u))+incr(right(v)))

u =parent(u)

size(T) = size(T)-1

min(T)= minimum of (val(T),min(left(T))+incr(left(T)), min(right(T))+incr(right(T)))

```
MULTI ADD ( D, i, j, x )
{
    Let u be the node storing ith element
    Let v be the node storing jth element
    w = LCA (u,v)
    val (w) = val (w) +  x
    if ( u <> w )
    {
        val (u) =val ( u ) + x
        If ( right (u) <> NULL )
            incr ( right (u) ) = incr ( right ( u) + x
        while ( parent (u) <> w )
            If ( u = left ( parent (u ) )
                val ( parent (u) ) = val ( parent (u) ) + x
                If ( right ( parent (u) <> NULL )
                    Incr (  right ( parent (u)  ) = incr (  right ( parent (u)  ) + x
            end if
            u = parent ( u)
        end while
    }
    Similar process for v
    Now we should update the min ( ) field of the ancestors of u and ancestors of v both.
```

```
UPDATE (u ,T )

{

        While ( u <> NULL )

                Min (u) = minimum ( min(left (u)) + incr ( left (u)) , min(left (u)) + incr ( left (u)) , val(u) )

                u = parent (u)

        end while

}


UPDATE (v ,T )

{

        While ( v <> NULL )

                Min (v) = minimum ( min(left (v)) + incr ( left (v)) , min(left (v)) + incr ( left (v)) , val(v) )

                v= parent (v)

        end while

}

Assumption : parent (T) = NULL


End MULTI ADD ( D, I, j, x)
```

```
Min ( D, i, j )
{
    Let u be the node storing ith  element
    Let v be the node storing jth element
    w = LCA ( u,v )
    If (u <> w )
        M1= val (u)
        If (right (u) <> NULL )
            M1 =minimum ( M1 , min(right(u)) + incr( right(u))
        M1 = M1 + incr (u)
        While (  parent(u) <> w )
            If (  u = left (parent(u))
                M1 = minimum ( M1 , val ( parent(u) ,
                        min (right(parent(u))) + incr(right(parent(u)))  )
            end if
            M1 = M1+ incr (  parent(u))
            u= parent (u)
        end while
    end if
    Similarly, we can use v to find M2
    M = minimum ( M1 , M2 , , val (w) )
    While (w <> NULL)
        M=M+ incr(w)
        w=parent(w)
    end While
    return M
}
```