CS345 Theoretical Assignment 1

Abhibhav Garg 150010
Abhisek Panda 150026

August 29, 2017

1 Question 1

To design an $O(n)$ space data structure to answer queries inquiring all the points(not the number of points) above a query line in 2D space in $O(klogn)$ time. Given, the space of points is constant.

1.1 Algorithm

- Divide the space of points into layers of convex hulls. This is achieved by first computing the convex hull for entire set, then remove all the points lying on this hull, then recurse for the remaining points till the point space is exhausted.
- This reduces the given problem to finding the points above a query line in a single convex hull, and repeating the procedure for each hull.
- Consider a single convex hull. We store the convex hull as an array of points on the hull, starting with the leftmost(minimum x coordinate) point on the hull. The points are stored in clockwise manner of their appearance. Also, maintain a pointer to the element corresponding to the rightmost point on the hull(maximum x coordinate). The elements between leftmost and rightmost parts represent the Upper Convex Hull and the other part represents the Lower Convex Hull.
- Note that if we are able to find any 1 point above the query line, we can easily find all other points above the line in $O(k_1)$ time where k_1 is the number of points above the line, by linearly traversing along the array in both directions.
- Any one point above the line(if exists) can be found by using a Binary Search on the array representing upper half of the hull(lets call it U). The binary search is performed based on slope of query line and slopes of the lines joining mid point to its neighbours.
- If the midpoint is above the line, we are done. Else, we join the point to its right and left neighbours. Let slopes of these lines be m_R and m_L respectively. If m is less than m_R, we recurse in right half. If m is greater than m_L, we recurse in left half. Else we return -1. This works since the polygon is convex.
- Since a point above the line, if exists, is sure to be present in the Upper half, we consider only the upper half for binary search.
- Once a point is found above the line, traverse left as well as right of that point along the convex hull, till a below point is reached on both sides.
- We start considering the hulls from the outer side. Note that once binary search for any hull returns -1(i.e., no point above the line is found), we can simply stop there, since it is guaranteed that no inner convex hull will have any point above the line.

Pseudo code for Binary Search:

Algorithm 1 Binary Search to find one point above a given line

```
function SEARCH(U, left, right, m, b)        ▷ m represents the slope of line, b is the intercept
    if left ≥ right then return -1
    end if
    mid = (left + right)/2
    if isAbove(u, mid, m, b) then return mid
    else if m ≤ slope(U, mid, mid + 1) then return search(U, mid, right, m, b)
    else if m ≥ slope(U, mid, mid − 1) then return search(U, left, mid, m, b)
    else return -1
    end if
end function
```

isAbove and slope are utility functions to check if a point is above line and finding slope of a line between 2 points respectively.

1.2 Space Analysis:

We use 1 array to store all the points in a convex hull. This takes $O(n_i)$ space where n_i is the number of points on the hull. This has to be repeated for all convex hull layers formed from the points. Thus, total space required is $\sum O(n_i)$ which is equal to $O(n)$.

1.3 Query Time Analysis:

Given a query line, we have to find number of points above it for each convex hull. For a fixed hull, it takes $O(\log n_i)$ time to locate one point above the line through binary search. It further takes $O(k_i)$ time to report all the points above the line, where k_i is the number of such points. So, total time for query per convex hull is $O(\log n_i) + O(k_i)$.

Thus, total time for the entire point space is $\sum[O(\log n_i) + O(k_i)]$ where i varies over all convex hulls. $\sum O(k_i)$ is equal to $\sum O(k)$ where k is the total number of points required. The worst case time complexity will occur when each convex hull contains exactly 1 point above the line. Since we stop after k iterations of the algorithm(as per last point of the algorithm), i varies from 1 to k only. Approximating n_i as O(n), we obtain a loose bound for the time complexity as $\sum_{i=1}^{i=k} O(\log n)$ which is equal to $O(k \log n)$.

1.4 Pre-Processing Time Analysis

Convex hull can be obtained in $O(n \log n)$ time. It takes extra $O(n_i)$ time to remove the points of ith layer and store it in the corresponding array where n_i is the number of points on convex hull in layer i. Thus, total time complexity is $O(n \log n) + O(n_1) + O((n - n_1) \log(n - n_1)) + O(n_2) + ... = O(n^2 \log n)$

1.5 Proof of Correctness

The main point here is that we start considering the hulls from outermost layer. Since it is convex, atleast one point(if any) is bound to exist in the upper half of the hull such that it lies above the line. This guarantees the binary search to return an above point if it exists at all. This leads us to find all the points on the outer hull that are above the line by linear traversal. Repeating this process for all hulls until one of them returns -1 gives the entire set of required points. The process can be stopped after any layer returns -1 since any inner layer is fully enclosed by the outer layer. Note that no point can be repeated in multiple hulls since we remove all points lying on a hull before processing the next inner hull.

2 Question 2

To design a $O(n)$ data structure to answer the following queries in $O(\log n)$ time.

- Insert(D, i, x): Insert an element x at i^{th} place in the sequence.
- Delete(D, i): Delete i^{th} element from the sequence.
- Multi-add(D, i, j, δ): Add δ to each element of the sequence starting from ith place to jth place.
- Min(D, i, j): Report the smallest element in the sequence starting from ith place to jth place.

2.1 Design

- Design a Balanced BST storing the following fields:
  - size: size of subtree rooted at the node
  - val: value of node
  - pointers to left and right node
  - a parent pointer for fast access to parent nodes
  - color bit
  - inc: amount to be incremented for sub-tree rooted at the node
  - min: minimum value in the subtree rooted at u (ignoring all increments due to parent nodes)
- Initialize the inc field to 0 in all nodes. The min field is recursively initialized as

$$node.min = minimum(node.val, left(node).min, right(node).min)$$

  where the base case is for leaf node, u.min = u.val; Null nodes are suitably handled(by ignoring them)

2.2 Designing the functions:

2.2.1 Multi-add(D, i, j, δ)

This function is very similar to the function discussed in the class for range addition in $O(\log n)$ time. The only difference is that whenever we update the **inc** field of any node or increment the value of a node itself, we also update the min field of its parent node(if exists), since addition in a range may lead to change in minima of its super set. This has to be done recursively for all nodes while moving up in the tree from i(and j) to LCA(i,j). The extra work has to be done for $O(\log n)$ nodes only, thus the time complexity continues to be $O(\log n)$. The updation of parent is carried out as follows:

$$parent(u).min = minimum(parent(u).val,\ sibling(u).min + sibling(u).inc,\ u.min + u.inc)$$

Note that in case sibling(u) does not exist, $parent(u).min = minimum(parent(u).val,\ u.min + u.inc)$

2.2.2 Min(D, i, j)

Let u be the node storing i^{th} element. Let v be the node storing j^{th} element. First we find the LCA of u & v. Let it be w. Then we start again from the root, and calculate a total_inc variable to know the total increment, $\Sigma\ inc_i$ where inc_i is the inc field of each node encountered till reaching target node while starting from root(including the inc field of target node itself).

The minimum value for any subtree rooted at v will be $v.min + \Sigma\ inc_i$.

Similarly, the value of any node v will be $v.val + \Sigma\ inc_i$.

For u, we need to only consider the values of all nodes whose left child is in the path from LCA to u, and the min field of the root of the right subtree, of nodes whose left child is on this path. Similarly for v. Finding the minimum among these values will give us min(i,j)

There are $O(\log n)$ nodes, so it takes $O(\log n)$ time to find the minimum among their values(total_inc is calculated dynamically, so O(1) time is required for updating it).

Similarly, there are $O(\log n)$ subtrees, and we have to check the min field of only their roots. This would again take $O(\log n)$ time.

Thus, the total time complexity becomes $O(\log n)$ for range minima inquiry.

2.2.3 Insert(D, i, x)

Insert operation of value x at index i is similar to the function discussed in class. We traverse to left or right depending on the values of root.size and i. If we move towards right, we modify i as $i - left(root).size - 1$ to account for the left subtree and root. Also, the size field of each node traversed is increased by 1 to accommodate the new node. The new node gets size 1.

However, we also need to update the min for every node in the path(if necessary) and the value of the new node(final value after considering all increments should be x) so that the structure is not disturbed.

The value x', which is stored in the new node, can be calculated easily by subtracting the inc value of each node in the path for reaching x as we traverse the tree. For updating minimum value, consider the following 2 cases:

- We move towards right from root. In this case, the new value to be inserted is more than the value at root. Hence, there is no change in min field of root.
- We move towards left from root. In this case, there is a possibility that min value might get updated. The condition is mentioned below:

$x \leftarrow x - root.inc$
if root.min $\geq$ x **then**
    root.min = x
else
    continue
end if

Note that at the end of insertion procedure, the obtained BST may not be balanced. We thus need to perform rotations in order to restore the balance. However, note that the parameters like min, size and inc are local, i.e, they are only updated for the nodes joined by the edge around which rotation is performed and can be completely determined by their neighbors in the tree. Thus, the balancing operation will take $O(1)$ time, since each rotation can be performed in constant time.

2.2.4 Delete(D, i)

Let u correspond to the i^{th} index. Find the path from root to i^{th} node by similar function to insert(using size field). Decrease the size field of each node along the path by 1. On reaching u, there are 2 cases:

- u is a leaf node. In this case, simply free u and update the min fields of its ancestors recursively.
- u is an inner node. In this case, find the in-order successor of u. During this routine, also decrease the size field of each node in the path to in-order successor by 1. Then, swap the val fields of u and its in-order successor. Finally, update the min field of each ancestor recursively.

Finding i^{th} index and swapping it with its inorder successor will take $O(\log n)$ time. The updation of min field in bottom up manner is also $O(\log n)$.

After deletion, the tree may become unbalanced again. Similar explanation as the insertion procedure leads us to conclude that each rotation can be performed in $O(1)$ time along with updation of corresponding fields. However, since the deletion itself takes $O(\log n)$ time in worst case, balancing would require a further $O(\log n)$ time.

The total time complexity therefore remains to be $O(\log n)$.

The min field is for all ancestors is updated as follows:

Algorithm 2 Iterative procedure for updating min after deletion

.

```
u ← parent(u)
function MinUpdate(D, u)
    while u IS NOT NULL do
        if left(u) AND right(u) then                                  ▷ both children are present
            u.min = minimum(u.val, left(u).min + left(u).inc, right(u).min + right(u).inc)
        else
            u.min = minimum(u.val, child(u).min + child(u).inc)       ▷ Only 1 child is present
        end ifu ← parent(u)
    end while
end function
```

3 Question 3:

3.1 Algorithm Description

The greedy method used for the algorithm is as follows: At every step, we find the nodes that are closest in the graph, ie $argmin_{i,j} d(u_i, u_j)$. These two nodes are assigned the same parent, and the value $h(x)$ for the parent is set as $d(u_i, u_j)$. Now, these two nodes, are combined into a single supernode, say U. Further, all distances in the graph are set as $d(U, u_k) = min(d(u_i, u_k), d(u_j, u_k))$. This super node is added back to the graph as described above, and a smaller instance of the problem is thus obtained. This smaller instance is recursively solved for its solution. For the base case, where there are just two nodes in the graph, tree T^* is constructed by taking a fixed root node (which will serve as the root of our rooted tree), assigning it two children, namely the two nodes, and settings its value h to be the distance between the two children. Then in the solution of the smaller instance, the super node is replaced by a node, whose two children are u_i, u_j, and whose value, as described above, is $h(x) = d(u_i, u_j)$. This completes the algorithm. Note that the algorithm is greedy since at every step we take the two closest nodes in the graph and give them a least common ancestor which is just one level higher in the rooted tree than the nodes themselves.

3.2 Proof of Correctness

The proof of correctness will follow the method described in class. The first part of the method, which involves the reduction of the problem to the smaller step and going from the smaller solution to the bigger solution has already been mentioned above. The proof begins with some lemma, followed by an argument to tie everything together.

Lemma 1. *There exists an optimal solution in which every non-leaf node in the tree has exactly two children.*

Proof. By construction. Take any node N in some optimal solution that has multiple children, say A_i for $i = 1, 2, \ldots$. Create a new node M are follows: Set its children to be $A_2, A_3, \ldots$. Set M and A_1 to be the children of N. Set the value of M to be the same as N. In this new tree, all pairs of nodes have the same tree distance and thus this solution is also optimal. This can be applied recursively till all nodes have just two children. □

Lemma 2. *There exist some optimal solution where the given two nodes u_i, u_j are siblings.*

Proof. Take any optimal solution. Let u_i and u_j have siblings u_i', u_j' respectively. Let u_k be the least common ancestor of u_i, u_j. We can swap u_i' and u_j to make u_i and u_j siblings. The only changes the LCA of nodes in the subtree rooted at u_k. Note that the original value of $h(u_k)$ is bounded by $d(u_i, u_j)$, which is a minimum. Thus, after the swap, all values of h for the nodes in the subtree rooted at u_k can remain constant without disturbing the balance condition, which means our new solution is also optimal, or the values can increase, leading to a contradiction to the original assumption. This completes the proof of the lemma. □

We thus have a way of reducing the problem into a smaller subproblem, and a lemma that relates the problems. We now prove that optimal solutions give optimal solutions in both directions, completing the proof of correctness of the algorithm.

Lemma 3. *Optimal solution of G gives optimal solution of G' under the above transformation.*

Proof. Let $h_G(LCA(u_a, u_b)) = \alpha_{ab}$ for a, b distinct from the i, j that are the minimum. Assume that there exists some tree $T'(G')$ for the smaller problem such that $h_{G'}(LCA(u_a, u_b)) > \alpha_{ab}$ for some fixed a, b. In this new tree, we can now then substitute the combined node back for u_i, u_j without affecting u_a, u_b. We thus have a tree that has higher value for h at for nodes u_a, u_b, a contradiction. Thus $T(G')$ is optimal. □

Lemma 4. *Optimal solution of G' gives optimal solution of G under the above transformation.*

Proof. The value for h for the parent of u_i, u_j is set as their distance, which is the minimum, and no tree can make this higher. For pair of nodes that does not include i, j, if $T(G)$ had a greater h value than $T(g')$ in an optimal solution, we can reconstruct a smaller tree from the optimal condition, and reach a contradiction in the same manner as that in the previous lemma. For every pair of nodes that includes i or

j, since $LCA(u_k, u_i) = LCA(u_j, u_k)$, no higher value of h is possible for the LCA, since it is contrained by the minimum of the distances to u_i and u_j, the same value that was set to the super node by construction. (Note that the fact that an optimal solution with u_i, u_j exists is used in the proof for the last statement, which lets us only worry about trees where u_i, u_j are infact neighbours.). This completes the proof. □

The above four lemma - along with the trivial proof of the base case, complete the proof of the algorithm.

3.3 Efficient Implementation

An efficient implementation using heaps is used. A min-heap of all the edges is maintained with the distance as the key. An array is also maintained with the pointers to the nodes in the heap for constant time node access, and thus logarithmic time key update.

Initially, $\log n^2$ time is taken to get the minimum distance edge. Further, it takes $O(n \log n)$ time to find and update the keys of all edges in the array which have either u_i or u_j as one of their vertices, since there are $2n$ of them. In this update, we can either delete one of two copies of the edges, or set its value to infinity so it doesnt affect the rest of the heap. Finally, it takes constant time to take the solution of the smaller problem and build the solution for the problem at hand. The time taken once the heap is built is given by the recursion $T(n) = T(n-1) + O(n \log n)$ which has solution $T(n) = O(n^2 \log n)$. Thus the total time includes the n^2 time for heapify and $n^2 \log n$ time for solving, giving a final asymptotic time complexity of $O(n^2 \log n)$