Complexity and Algorithms: Homework-4 solutions

March 9, 2017

Answer 1:
We can solve this problem by modifying the visibility problem discussed in class. Instead of finding topmost line, we keep track of ranking of line segments by adding tokens for nodes in the balanced binary search tree. In the token, we save number of nodes in the subtree rooted at this node. When encountering events, we need to update the tree structure and token values accordingly. When a node is inserted, for every node on the traverse path to the inserted location, their token increase by 1. Similar for deletion, but we decrease tokens by 1 on the traverse path to the node to be deleted.

With the augmented tree data structure, we can report $Kth$ topmost line(or node in the binary tree) as follows. Assume we have $i$ nodes in the right subtree $T_r$. If $k = i + 1$, root is the kth node. If $k < i$, continue search right subtree. If $k > i + 1$, continue search for the left subtree $(k - i - 1)th$ largest.

*Please report if you find any error in the solutions*
Pseudocode:

KTH-LARGEST(v,k):
    if v = NULL then
        return NULL;
    end
    else if k > COUNT(v)+1 then
        return KTH-LARGEST(LEFT-CHILD(v),k-COUNT(v)-1)
    end
    else if k = COUNT(v)+1 then
        return VALUE(v);
    end
    else
        return KTH-LARGEST(RIGHT-CHILD(v),k)
    end

GENERALIZED-VISIBILITY(S,k):
    E = the end points of the line segments in S sorted from left to right, stored with
    references to their line segments, and whether they are right or left endpoints.;
    T = an empty balanced binary search tree of line segments which supports
    looking up the kth largest element, where insertion points are determined by
    evaluating each line segment at a given x-coordinate;
    R = an empty list of line segments;
    L = a NULL line segment;
    if p is a left endpoint then
        Insert into T at x the line segment whose right endpoint is p;
    end
    else
        Remove from T the line segment whose right endpoint is p;
    end
    L = KTH-LARGEST(root(T),k);
    if L ≠ NULL then
        y' = the y-coordinate of L at x;
        Set the left endpoint of L to (x,y');
    end
    return R;
Complexity analysis:
The initial sorting of endpoints takes $O(n \log n)$. The kth largest element in T is looked up in $O(\log n)$. Hence, total complexity is $O(n \log n)$.

Correctness argument: At each event, the algorithm records the kth topmost line segment intersected by the sweep line using the KTH-LARGEST procedure; by definition of the problem, this generates the correct result. Every time an end point is encountered, a line segment is inserted into or removed from the BST, which may change the current kth topmost line segment. Before this is done, the previous kth topmost segment is capped at the current x coordinate and added to the result. Then, the BST is updated.

Answer 2:
Solving this problem in linear time requires a small modification to the SELECT algorithm. When deciding which subset of S to continue to search for the median in, we use the sums of the weights of their elements, rather than the numbers of elements they contain, to determine where to recurse. Computing these sums takes $O(n)$ time, so the overall complexity of the modified algorithm is identical to that of SELECT.

Complexity analysis: As explained in the procedure, the time complexity will be $O(n)$.

Correctness: The WEIGHTED-MEDIAN procedure invokes a more general WEIGHTED-SELECT algorithm with a value of $c = 1/2$.

In the base case, the procedure iterates through S in sorted order, accumulating the sum of the weights seen so far. As soon as it sees a weight $w$ greater than or equal to $c$, it returns the current value $x_i$. Since the loop did not exit before, and all elements $x_j$ such that $x_j < x_i$ have been processed, this means $x_i$ is the weighted median $x_k$ such that $\sum_{x_j < x_i} w_j < c$, meeting the definition of the weighted median when $c = 1/2$.

In the recursive case, the procedure partitions S into 3 subsets as described.

Pseudocode:
Let $c$ be a real number number such that $0 \leq c \leq 1$

WIGHTED-MEDIAN(S)
    return WEIGHTED-SELECT(S,1/2)
Answer 3:

We use the concept of weighted median and binary search to solve this problem. Suppose we have narrowed down our search for the kth smallest element between two indices in each array, resulting in h pairs of indices. We would like to find a value which could be used to partition the elements in each array in such a way that the remaining search space is pruned by a constant factor. When we have h sorted arrays of equal size, we can easily find such a value. However, when we have arrays of unequal sizes, we want to weight the medians differently depending on how many elements they prune from the search space. We can do just this using the concept of weighted median. Once we have found this median, we find the partition index of each sorted array using binary search. We recurse or return the median based on how the number of remaining elements compares to k. Below is a sketch to further explain the idea.
Medians weighted by the size of the array

sorted order (closest to impact)

discarded when

# elements ≈ median
invariant in set $S_1 < K$

discarded when

# elements $> K$
in set $S_1 > K$

$S_1$: set of all elements that are less than $M$.

Since medians of each array are weighted by their respective length of array length, we discard at least $\frac{1}{4}$th of the total number of elements at each iteration.

Where:

\[ S_1 : \text{set of all elements that are less than } M. \]
Pseudocode:
SELECT-FROM-SORTED-ARRAY(A₁, A₂, ..., Aₜ,k):

N = ∑ᵢ |Aᵢ|;
if N ≤ h log (n) then
    A = the concatenation of A₁, A₂, ..., Aₜ;
    return SELECT(A,k);
end
else
    M = get medians of each array, weighted by the size of the array;
    for each array Aᵢ do
        [S₁, S₂, S₃] = BINARY-SEARCH(Aᵢ,M)
    end
    /* After this loop S₁ will have all elements less than M, S₂ will have all elements equal to M, and S₃ will have all elements greater than M */
    if k ≤ S₁ then
        return SELECT-FROM-SORTED-ARRAYS(pruned arrays, k);
        /* All the elements in the arrays(having median element larger than the weighted median M), that are larger than their respective medians are pruned */
    end
    else if k > S₁ then
        return SELECT-FROM-SORTED-ARRAYS(pruned arrays, k-number of pruned elements);
        /* All the elements in the arrays(having median element smaller than the weighted median M), that are smaller than their respective medians are pruned */
    end
end

Complexity analysis: Finding the weighted medians in h arrays will take \( \mathcal{O}(h) \). Binary search on all the elements of all the arrays will take \( \mathcal{O}(h \log n) \). Since we make sure that atleast 1/4 of the total number of elements are pruned and each recursion costs \( \mathcal{O}(h \log n) \), we have the following recursive relation.
\[ T(hn) = T\left(\frac{3hn}{4}\right) + ah \log n \]
\[ T(h \log n) = bh \log n \]

This can be solved to obtain the final complexity of the algorithm. That is, \( O(h(\log n)^2) \)

**Correctness:** In the base case, the algorithm simply runs the SELECT on the combined elements to produce the correct answer. In the recursive case, we only discard the elements that are either less than the medians or greater than the medians of the arrays.

**Answer 4:**

**Approach:** A naive approach to solve the problem would be to use LP approach on \( hn \) half planes (i.e., each side of the convex polygons). Unlike general LP problem, we can take advantage of the fact that the vertices have already been computed and are given in sorted order. Hence, we can relate this problem to the problem 3. Here also, we have \( 2h \) arrays that are given in sorted order. We do the following procedure to determine if \( n \) polygons have a common intersection. First, apply SELECT-FROM-SORTED-ARRAYS procedure from the previous problem to obtain the weighted median (of the line segment endpoints represented by the numbers in the \( 2h \) arrays) and use that point for the probe operation. Then, binary search on each array using the weighted median, in order to detect, in each array, endpoints that are less than \( (S_1) \), or greater than \( (S_2) \) median. Perform probe line operation to determine which side of the median we want to go next. Perform this operation recursively till you have \( O(h \log n) \) line segments. When you have \( O(h \log n) \) line segments left then perform LP algorithm to solve the problem.

When our probe line intersects \( 2h \) line segments (or half planes) and all of the lower line segments (or upper half planes) come before the upper line segment (or lower half planes), we know that the intersection is non-empty. If we have pruned the number of line segments to fewer than \( 2h \), then we know that an h-way intersection is impossible.

**Pseudocode:**

Let \( P_1, P_2, \ldots, P_h \) be \( h \) polygons with \( n \) sides each, where each polygon is represented as an array of edges starting at leftmost vertex.
COMMON-INTERSECTION-IS-EMPTY()

for each $P_i$ do

$m_i = \text{the index of rightmost point in } P_i$;

$U_i = P_i[\ldots m_i]$;

$L_i = P_i[m_i\ldots]$ /* in reverse order. we can do this since points of polygons are stored in clockwise order */

$U = U_1, U_2, \ldots, U_h$;

$L = L_1, L_2, \ldots, L_h$;

return HULL-INTERSECTION-IS-EMPTY(U,L);

end
**Complexity:** Each time a recursive call is made, the algorithm is invoked with roughly half the set of vertices it started with. Also, each recursive run is dominated by the runtime of selecting weighted median.

\[ T(hn) = T(hn/2) + O(h(\log n)^2) \]

\[ T(h \log n) = O(h \log n) \]

This will give:

\[ O(h(\log n)^3) \]