Homework 3

Elliott Pryor
Collaborated with: Nathan Stouffer

09 March 2021

Problem 1

Given an polygonal chain P of n vertices, we define an vertex v as a *local max* if v if all edges adjacent to v are to the left of v. Show that can determine if a polygonal chain with k local maxes is simple in $O(n \log k)$ time.

Problem 2

A friend of yours from the civil engineering department wants to analyze whether a dangerous portion of a river will flood. He presents you with the following (admittedly rather unrealistic) model of the river. The portion of the river of interest is modeled as an x-monotone polygon P that is bounded between two vertical lines at $x = x^-$ and $x = x^+$ (see Figure). The river is bounded on its left and right ends by two vertical line segments of lengths w^- and w^+, respectively. Inside the polygon are some number of disjoint x-monotone polygons that represent islands in the river. Let n denote the total number of vertices, including both the outer banks of the river and the islands.

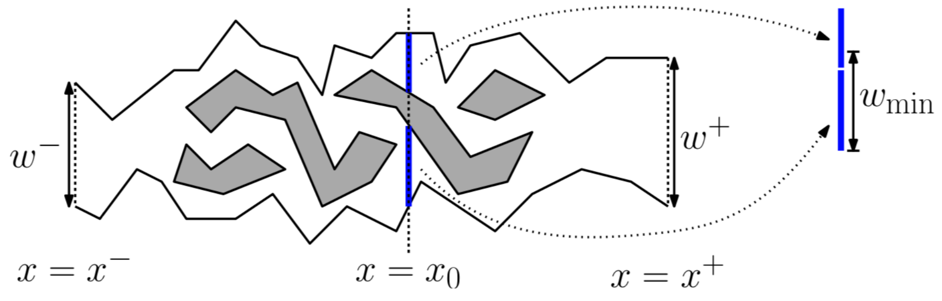


Figure 1: Problem 2: River

Your friend tells you that the in order to avoid a flood, the width of the river (not counting islands) at every vertical cut must be at least some minimum value w_{min}. For example, in the figure, the sum of the two blue vertical segments at $x = x_0$ must be at least w_{min} in order to avoid a flood. Given the polygon P and the value w_{min}, present an $O(n \log n)$ time algorithm that determines whether the river will flood, that is, whether there is a vertical cut whose total width is smaller than w_{min}. If it will flood, your algorithm should output the value x_0 of the bottleneck, that is, the location where the sum of vertical lengths (excluding islands) is the smallest.

1. Hint 1: There is an (uncountably) infinite number of possible vertical cuts to consider. Prove that it suffices to check the width at a discrete set of locations, whose number is $O(n)$.
2. Hint 2: There is a bit of a trick to updating the vertical widths (excluding the islands). For partial credit, explain how to do it under the assumption that the sweep line can only intersect a constant number of islands at any time. (For example, in the figure, the sweep line never hits more than two islands at a time.) For full credit, explain how to do it even if the number of islands hit by the sweep line at any time could be as high as $\Omega(n)$.

So we first create a list of the event points. An event point stores an x location, and a pointer to a vertex. We create one event point per vertex. Each event point will also have the slope of the 'left' edge and the slope of the 'right' edge. One of these are 0 if the vertex is leftmost or rightmost in polygon (e.left = 0 if e is leftmost point). Then we have a sign value (s) which is -1 or 1 if the vertex is on upper hull or lower hull respectively. (One small technicality that the river bank is labeled with opposite sign (ie e.s of a vertex on upper bank of river is $+1$ instead of -1))

We then sort the event points by x coordinate.

On the sweepline we will store 2 variables: a, b that will define our width function.

We iterate over the events. For each event we set $a = f(e)$, and $b = b + e.s(e.right - e.left)$. Our function f is evaluated as $f(e) = a + b(e.x - e_{-1}.x)$ where $(e.x - e_{-1}.x)$ denotes the distance between this point and the previous event point.

We then have a global variable storing the minimum value of a.

Algorithm 1 Flood!!!

```
1: function FLOOD($B, P[]$)
2:     initialize events
3:     sort events by $x$
4:     $min \leftarrow +\infty$
5:     $a \leftarrow w^-$, $b \leftarrow 0$
6:     for $e \in events$ do
7:         $a \leftarrow f(e)$
8:         $min \leftarrow a$ if $a < min$
9:         $b \leftarrow b + e.s(e.right - e.left)$
10:    end for
11:    return min
12: end function
```

Runtime: Initialize events takes $O(n)$ since it is done with a single linear pass through events and constant time operations. Sorting takes $O(n \log n)$. The internal operations of the for loop are $O(1)$. It runs over each event so there are $O(n)$ operations in the for loop.

Thus in total: $O(n \log n)$

Correctness: This hinges on the correctness that our function f correctly evaluates the width at a point.

Proof. By Induction.

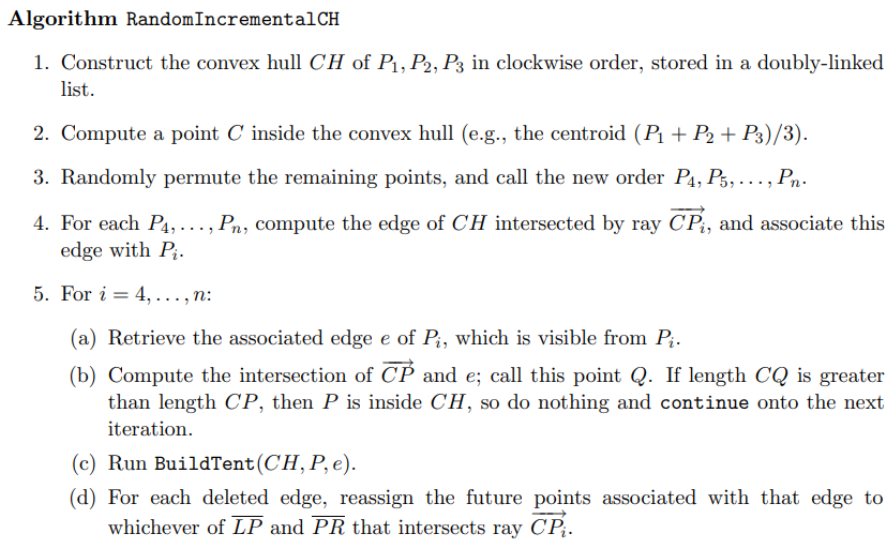
Base case is at the leftmost point. We initialize $a = w^-$ and $b = 0$, so $f = w^-$ at the leftmost point.

Inductive step: assume that the width is correct at the previous event point, show that it is correct at this event point. It just is IDK why is hard to prove. $\square$

Problem 3

1. (7 points) Describe and analyze an algorithm that computes the convex hull of a set of n points in the plane using randomized incremental construction in expected $O(n \log n)$ time. For this problem you are welcome to find an algorithm and its analysis on the web, but please cite where you found it, describe it concisely in your own words, and make the analysis very concise. Where does the log-factor come from?

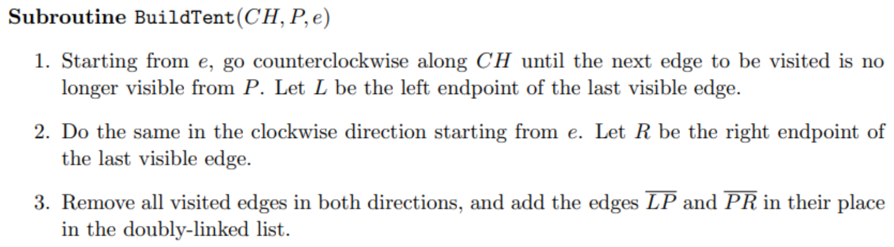
2. (3 points) Give an example of a set of points in the plane, and a particular input order, that causes the convex hull algorithm to run in $O(n^2)$ when the points are added in this particular order. Make sure it is clear how your example generalizes to arbitrary values of n.



Algorithm `RandomIncrementalCH`

1. Construct the convex hull CH of P_1, P_2, P_3 in clockwise order, stored in a doubly-linked list.
2. Compute a point C inside the convex hull (e.g., the centroid $(P_1 + P_2 + P_3)/3$).
3. Randomly permute the remaining points, and call the new order $P_4, P_5, \ldots, P_n$.
4. For each $P_4, \ldots, P_n$, compute the edge of CH intersected by ray $\overrightarrow{CP_i}$, and associate this edge with P_i.
5. For $i = 4, \ldots, n$:
   (a) Retrieve the associated edge e of P_i, which is visible from P_i.
   (b) Compute the intersection of $\overrightarrow{CP}$ and e; call this point Q. If length CQ is greater than length CP, then P is inside CH, so do nothing and `continue` onto the next iteration.
   (c) Run `BuildTent`(CH, P, e).
   (d) For each deleted edge, reassign the future points associated with that edge to whichever of $\overline{LP}$ and $\overline{PR}$ that intersects ray $\overrightarrow{CP_i}$.

(a) Randomized Convex Hull algorithm



Subroutine `BuildTent`(CH, P, e)

1. Starting from e, go counterclockwise along CH until the next edge to be visited is no longer visible from P. Let L be the left endpoint of the last visible edge.
2. Do the same in the clockwise direction starting from e. Let R be the right endpoint of the last visible edge.
3. Remove all visited edges in both directions, and add the edges $\overline{LP}$ and $\overline{PR}$ in their place in the doubly-linked list.

(b) Subroutine 'build tent' that detects what edges to replace

Figure 2: Randomized convex hull algorithm taken from Lecture 15 of 15-750 Graduate Algorithms at Carnegie Mellon University

Above is the description copied from Carnegie Mellon University lecture notes.

Alg We start by initializing the convex hull to be 3 random points from the point set. They then find some point in the interior of the convex hull: the centroid $C = \frac{P_1+P_2+P_3}{3}$. (the middle of the convex hull).

Then considering a random permutation of the points, they compute an edge that is 'visible' to P_i. An edge is associated with P_i if ray C, P_i intersects the edge. Then, we build the convex hull:

For each point, get the edge that is visible to it. Then check if the point is within the convex hull. Let Q be the point where the ray C, P_i intersects the visible edge. The point is inside iff $dist(C, P_i) < dist(C, Q)$. If P_i is inside, then we don't need to do anything: move on.

If P_i is outside, then we run BuildTent, to compute the left and right vertices. These are the vertices that are tangent to P_i. They then delete the edges between L, R and replace them with L, P_i and P_i, R. Then the points that were associated with deleted edges get re-assigned to either L, P_i or P_i, R based on the ray intersection from the initialization.

Runtime: Step 1, takes $O(1)$ since we make a convex hull of fixed size 3. Step 2, takes $O(1)$ since we compute the centroid. Step 3, takes $O(n)$ since we shuffle all the points. Step 4, takes $O(n)$, this is a little bit less trivial. But we know that we can check line intersections in $O(1)$. We then consider an edge in the convex hull, we can iterate through the entire list of points and check if the ray intersects the line segment in constant time. So this takes $O(n)$. We then only have 3 edges to consider, so $O(n)$ in total. So the entire initialization routine goes $O(n)$

Then we use backwards analysis. We remove a random point P_i from the convex hull, and analyze the cost of doing so. When we remove a point from the CH we must pay a cost of 'undoing' the BuildTent operation. So we pay a cost related to the number of points whose ray intersects these two edges. For each point q outside the CH, ray C, q intersects one edge, which is defined by 2 vertices. So the probability that we pay the cost of q by removing P_i is the probability P_i is one of these 2 vertices: or $2/|CH|$. There are at most $n - i$ vertices on the CH (every vertex that we haven't removed) and i points outside the CH. So summing over all i we get: $\sum_{i=0}^{n} i \cdot 2/(n-i)$. We do change of variables $j = n - i$ (intuitively j is number of vertices on the CH)

$$\sum_{i=0}^{n} i \cdot 2/(n-i) = \sum_{j=0}^{n} (n-j) \cdot 2/j \leq \sum_{j=0}^{n} (n) \cdot 2/j = 2n \cdot \sum_{j=0}^{n} 1/j = 2n\Theta(\ln n) = O(n \log n)$$

The log factor comes from the harmonic series. But physically it comes from the probability that we have to update what edge a point is associated with.

Problem 4

Consider the following instance of the trapezoidal map point location data structure. The left side shows the map, and the right side shows the corresponding DAG. Describe the resulting trapezoidal map and DAG after segment xy has been added.

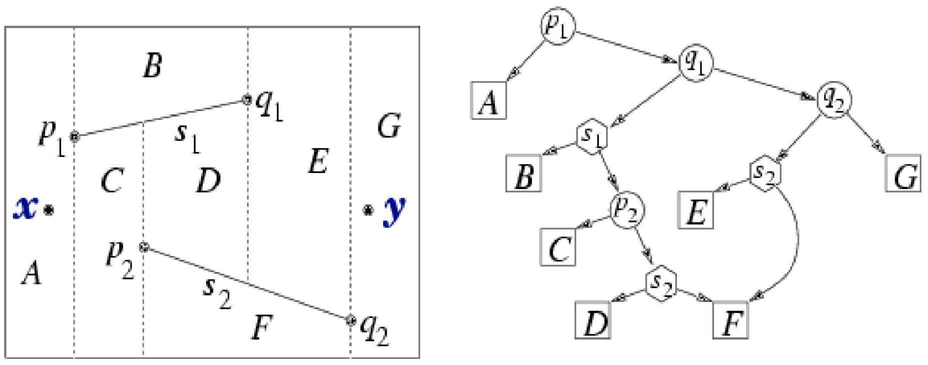
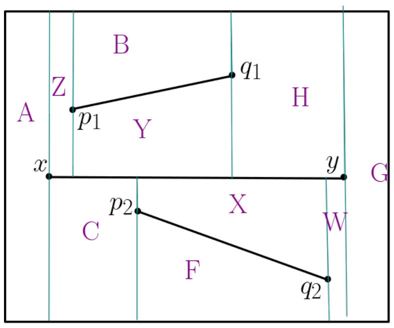
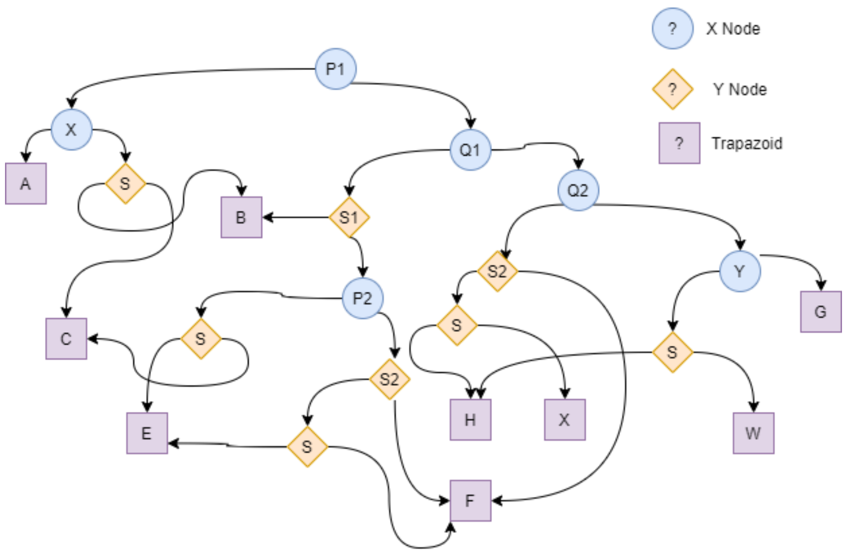


Figure 3: Problem 4: Trapezoid Map



(a) Trap map after adding segment xy



(b) DAG after adding line segment xy

Figure 4: DAG and Tree resulting from adding segment xy

So the DAG became a bit of a tangled mess. I apologize.

Essentially we just add vertical lines from x and y. Then we trim the lines that we intersect with the horizontal line segment. Then in order to build the tree. We take the original map, and apply the 3 cases to it, although we only need cases 1,3. We place these in where the trapazoids A, C, D, E, and G were. Because we intersect them they get modified in some way.

Problem 5

Consider the following algorithm:

```
FindMax(A,n){
    // Finds maximum in set A of n numbers
    if(n==1) return the single number in A
    else {
        x = extract random element from A // in constant time; x is removed from
        A
        y = FindMax(A,n-1)
        if(x<=y) return y;
        else
        Compare x with all remaining elements in A and return the maximum
    }
}
```

1. (4 points) Argue that this algorithm is correct, and give its worst-case runtime. (The runtime is proportional to the number of comparisons made.)
2. (6 points) Compute the expected runtime of this algorithm. (Hint: Introduce an indicator random variable for executing the else branch in the *i*-th step, and use backwards analysis to simplify the analysis.)

1.

Correctness:

Proof. By Induction.

Base case $|A| = 1$, therefore the only element in A is the max and it is trivially true.

Inductive step: assume we have a max for the $k-1$ elements, show we find max for k elements. Let $A_k = A$ when $|A| = k$, and $A_{k-1} = A_k \setminus \{x\}$. Clearly A_{k-1} is what is covered in the next recursive layer down. So by the inductive assumption y is the max of A_{k-1}.

If $y \geq x$ then y is the max of A_k. We partition A_k into A_{k-1} and x. By inductive assumption y is max of A_{k-1}, and we have that $x \leq y$, so y is the max of both partitions, therefore is the max of A_k.

Otherwise, we search through each element of A_k and find the max. This clearly finds the maximum, because if it didn't one element would not have been considered.

Thus the inductive step holds. $\square$

At worst case, we always choose $x = \max A_k$. Then we always enter the else statement. Since we remove elements one at a time until $|A| = 1$, our recursion depth is $n - 1$. If we enter the else statement we check all the elements of k.

So our recurrence relation is $T(k) = T(k-1) + k$. So therefore at worst case it is $O(n)$

2.

Consider iteration i of the algorithm. So $|A| = i$. Let R_i be the random event that the else branch is executed Let P_i be the probability taken over all permutations of A of R_i.

There is only one number $x \in A$ such that $x > y$. Since $y = \max(A \setminus \{x\})$. Thus the probability P_i is the probability that we select this number. So $P_i = 1/i$.

This means that the expected number of times that we enter the else statement over all iterations is: $E = \sum_{i=1}^{n} P_i = \sum_{i=1}^{n} 1/n = \Theta(\ln(n)) = O(\log n)$ since this is the harmonic series. Each time we enter the else statement takes $O(n)$ time since we iterate over the whole list.

Thus the running time is $n + O(n \cdot \log n)$ where the first n comes from the recursion depth. We consider the recursion depth separately since the $\log n$ factor got ‘summed out’. So the overall running time is $O(n \log n)$