HW 2 - CS 120

Evan Jolley

September 2022

1

1.1

This problem closely mirrors Proposition 4.4 from Lecture 3.

Proof

First we describe the reduction algorithm.

- Convert coordinates to polar form in constant time.

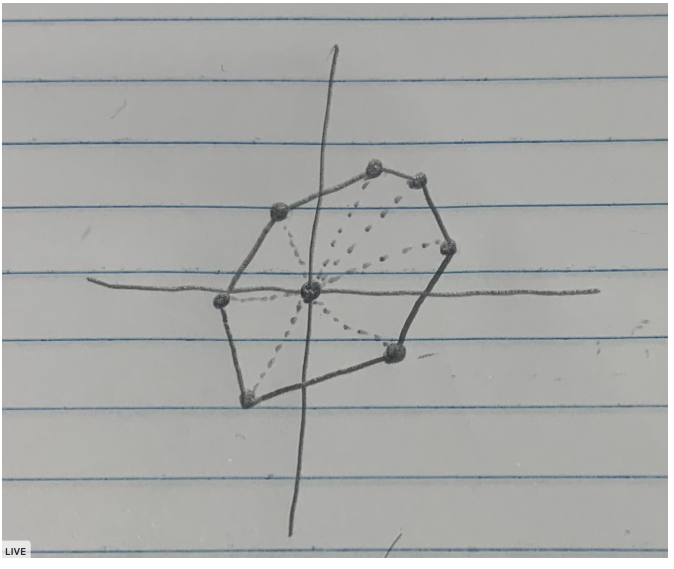
- Form an array of (key, value) pairs where $(K_i, V_i) = (\Theta_i, r_i)$ for all $i \in 1, ..., n$.
- Sort this array by Θ into a sorted array S, by an oracle call to Sorting.
- Given S, for each pair of adjacent elements $(K'_i, V'_i) = (\Theta'_i, r'_i)$ and $(K'_{i+1}, V'_{i+1}) = (\Theta'_{i+1}, r'_{i+1})$, calculate the area of the triangle formed by the origin, (K'_i, V'_i), and (K'_{i+1}, V'_{i+1}) using $A = \sqrt{s(s-a)(s-b)(s-c)}$ where $a = r_i$, $b = r_{i+1}$, $c = \sqrt{a^2 + b^2 - 2 * b * c * cos(\Theta'_{i+1} - \Theta'_i)}$, and s is obtained using Heron's formula. Note that c is calculated using the Law of Cosines. We must not forget to also calculate the area of the triangle formed by the origin, (K'_{n-1}, V'_{n-i}), and (K'_0, V'_0) to complete the interior of the polygon.
- Summing the areas of all calculated triangles will equal the area of the total polygon.

We now want to prove our reduction algorithm

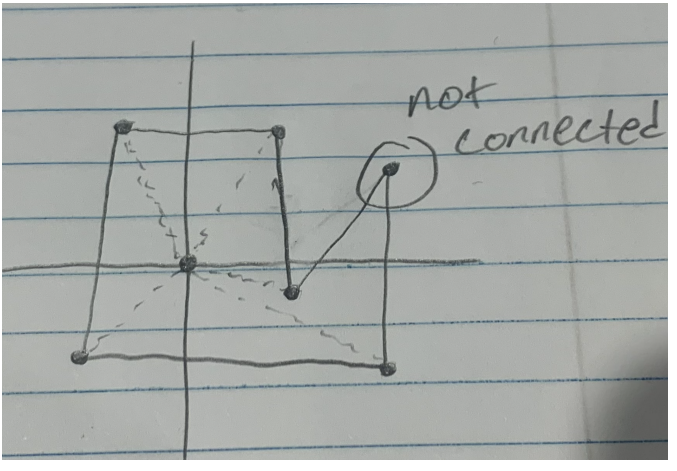
1. has the desired runtime and call size to the Sorting oracle, and
2. is correct.

- Forming an array in the first step takes time $O(n)$. Calculating the areas of the n interior triangles and summing them takes time $O(n)$. So the total time taken by the reduction is $O(n)$. Also, the array passed to Sorting has size n, as claimed.

- Imagine lines extending from the origin to all vertices on the polygon. If we calculate the area of each of the resulting triangles, we will have the area of the entire polygon, as described above.



Here is an example of the interior triangles described in bullet point 2. Note, that in a convex polygon all interior angles are less than 180 degrees. This means that there will always be a line between the origin and the vertices of the polygon that does not intersect the edge of the shape. This method of calculating the area would not always work with non-convex polygons, as pictured below:



1.2

The sorting of the array described in step 3 of the proof will run in time $O(nlogn)$. As we just proved that the reduction runs in time $O(n)$, we can deduce that $AreaOfConvexPolygon$ can be solved in $O(nlogn)$ as the runtime of the sorting dominates that of the reduction: $O(nlogn + n) = O(nlogn)$.

1.3

Γ can be solved in at most $T(n)$.

Instances of size at most $f(n)$ will be sent through Γ, so Γ can be solved in at most $T(f(n))$.

At most $k(n)$ calls to Γ will be made, so the total maximum runtime for the oracle is $k(n) * T(f(n))$

It takes at most $g(n)$ to reduce Π to Γ, so the total runtime including the initial reduction and the total oracle is $O(g(n) + k(n) * T(f(n))$.

2

2.1

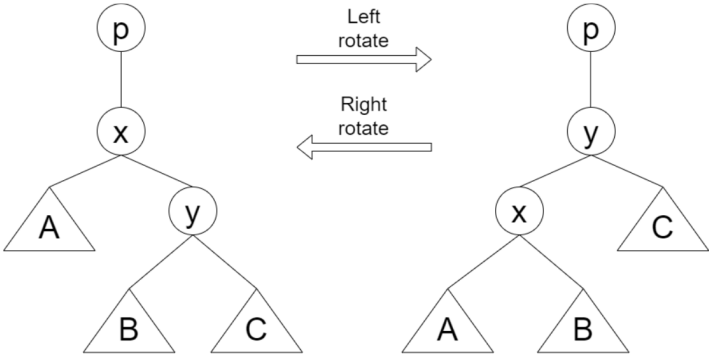
We find the correctness error in the *select* function. When we recursively call *select* on the right side of the tree, we need to update the index (let's call it i). This is because we don't want to return the ith smallest index of the right tree, we want to return the ith smallest index of the entire tree. If the index is greater than the current key, we subtract the size of the left tree and 1 (to account for the current key) and call the function with that number. We don't need to update on the left call, as if i is less than the current key, than the ith smallest index from the current key is equal to the ith smallest index in its left tree.

We find the function that is to slow to be *insert*. At the end of the current implementation of *insert*, the function $self.calculate_sizes$ is run. This function iterates over all elements in the tree to calculate their sizes. However, when inserting, we only need to increment the vertices we touch while inserting

because these are the only sizes that will change (the inserted element will be a descendent of all of them). To fix this you can simply increment the sizes of each vertex you pass over using $self.size+ = 1$.

2.2

I will be referencing this diagram that can be found in lecture 5's notes:



The sizes of p, A, B, and C will remain the same after rotating, so we only need to worry about updating x and y. Thus, no matter the size of this tree, we

will have a set number of operations we must execute, maintaining the runtime O(1).

Let's take a look at left rotate. The new size of y will clearly be the old size of x (A Size + B Size + C Size + 1). Reassigning this value in code would be simple. As for the new size of x, we can calculate this by adding the sizes of its new children (A and B) together. These values can be referenced easily as A is x's previous left child and B is x's previous right child's left child.

We can think about right rotate in the same way, simply reversing everything that was just explained. In this function it would be new x taking old y's size, and new y's size being the sum of B and C. Again, these values are can be easily accessed in code.

No matter what A, B, and C are, these relationships will always stay the same. Thus, if all vertices throughout the tree are correctly sized to begin with, they will be correct after the rotation as well.