Brigham Michaelis
February 13, 2018
A00364835

Computational Geometry Assn.2

1. (20 points) In this exercise, we study a special point location problem. Let S be a set of n disjoint line segments whose upper endpoints lie on the horizontal line y = 1 and whose lower endpoints lie on the horizontal line y = 0 (e.g., see Fig. 1). These segments partition the horizontal strip $[-\infty, +\infty] \times [0 : 1]$ into n + 1 regions. Give an O(n log n) time algorithm to build a binary search tree on the segments of S such that given any query point p in the strip, the region containing the query point p can be found in O(log n) time. Please also describe the query algorithm.

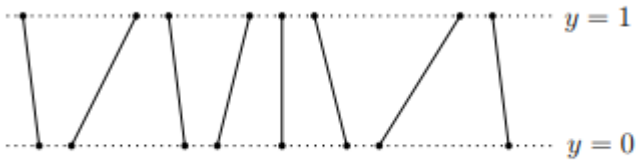


Figure 1: An example for Question 1.

**Algorithm Description**

First we need to build the binary search tree. This can be done by using a balanced BST and inserting "region" objects into this tree. These region objects contain the four points that make up the region. In linear time we can take two segments at a time and insert them into the tree, with every segment except the first and the last being used twice. But the question remains "what value will we use to sort these regions in the tree?" The value that we will use can be generic, we will give the first region inserted into the tree a value of 0, the next one will have a value of 1, then 2, then 3, and so on until all regions have been inserted into the tree. Because we are performing n insertions into a balanced BST this will take in total **O(n log n)** time.

Second we describe our query algorithm. This algorithm is simply performing a "find" on the balanced BST. Because these regions are made up of two line segments, we can determine in constant time whether a point is contained in a region.(Just check that the point is on the left side of the right segment and is also on the right side of the left segment.) Now we just have to perform a find on the balanced BST. Something like the following will work just fine:

```
if( the point is on the left side of the right line segment of the region ){
       if( the point is on the right side of the left line segment of the region ){
              // We are done and the region containing the point was found.
       }
       else {
              //traverse down the left side of the tree
       }
}
else {
       // traverse down the right side of the tree
}
```

Using the above methods we can make the tree in **O(n log n)** time, and we can perform a find in **O(log n)** time.

**Psuedocode**
Not Required
**Correctness**
Not Required
**Time Analysis**
Not Required

2. (20 points) Let S be a set of n triangles in the plane. The triangles are pairwise disjoint, i.e., the boundaries of the triangles do not intersect and no triangle lies completely inside another triangle. Let P be a set of n points in the plane. Design an O(n log n) time algorithm to solve the following problem: For each point q in P, determine whether q is contained in a triangle, and if yes, report that triangle. Hint: Use the plane sweeping technique.

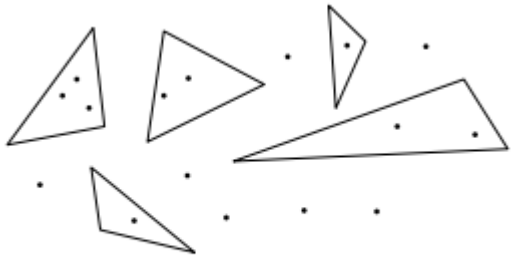

Figure 2: An example for Question 2.

**Algorithm Description**

In class we have been discussing the sweeping line technique, as was discussed in class we maintain a "sweeping status" of line segments. This problem is slightly different, instead of maintaining a sweeping status of line segments we will be maintaining a sweeping status of triangles. Obviously we are going to use a balanced balanced BST for our sweeping status, and much like the previous problem we are going to insert triangle objects into our balanced BST. But now we have to answer the question: "what value will we use to sort our balanced BST?" Before we answer this question let's take a step back and review. The sweeping status for this problem is made up of the triangles in the order that the sweeping line intersects them from left to right. When a new triangle is encountered it is inserted into the balanced BST and the sorting value that is used is the minimum "x-value" of the two intersections that are made by the sweeping line passing through the triangle, this can be done in constant time. Now that we have explained how our sweeping status works we show how to solve the problem.

1. Sort all the points in S and P. This will take **O(n log n)** time by ignoring constants.
2. Now that we have this sorted list of points, we are going to use our plane sweeping technique, starting with point with the max "y-value" and working our way down. At this point there are no triangles in our sweeping status. And while our sweeping status is empty all encountered points in the set P are not in any triangle.
3. Once a point is encountered that is part of a triangle, that triangle is then added to the sweeping status, which again is just a balanced BST. This triangle will remain in the BST until it's other two points have been encountered by the sweeping line, once that happens it will be removed from the balanced BST.
4. While the sweeping status is not empty, for every point we encounter with our sweeping line that is not part of a triangle we will perform a "find" on the sweeping status(where we compare the point's x-value and the minimum x value of the left-most intersection point of the triangle and the line L), and in a binary search fashion we will be able to determine in **O(log n)** time whether or not some point is contained in one of the triangles in the sweeping status. In a sense we are asking if the point is contained within the two intersection points of any given triangle, this operation takes constant time. If it is bound by the two intersection point we are done and report it. Otherwise wise we traverse down the left or right subtree as appropriate, and we also make sure to have a stop condition if we don't find the point and we have reached unity in the tree.

This works because we can determine in constant time whether some point is within the intersection points of a triangle and at worst we will perform **O(log n)** of these operations per point.

To recap:
We have n points in set P and 3n points in set S. We need **4n log(4n)** time to sort all these points. Ignoring the constants we have **O(n log n)** time to sort.

Once we have our sorted list of points we then put all these points into an event queue and pull them out one by one.

If our sweeping status is empty any points encountered are not in any triangle.

If our sweeping status is not empty then we perform a find on our sweeping status which takes **O(log n)** time per point, to determine if any triangle in our sweeping status contains that point and report as necessary. At the very worst this will take "**n**" find operations each one taking **log(n)** time. Thus our entire algorithm takes **O(n log n)** time to run

**Psuedocode**
Not Required
**Correctness**
Not Required
**Time Analysis**
Not Required

3. (20 points) A disk consists of a circle plus its interior and is represented by its center point and radius. Two disks intersect if they have a point in common. Given a set of n disks in the plane, design an O(n log n) time algorithm to determine whether there are two disks that intersect.

Note that if one disk completely contains another, these two disks also intersect. Also note that the radii of these disks may be different.

**Algorithm Description**

For this problem the first thing we are going to do is add to an event queue all the min and max y-value points of the circles. What do we mean? For every circle we are going to compute the point where its max y-value occurs and the point where its min y-value occurs.(This is found by adding and subtracting the radius to and from the y-value of the center point respectively.) Now we will sort all the values in the event queue by their y-coordinates in decreasing order which by definition is going to contain 2n points. This operation will take **O(n log n)** time ignoring constants. Once that step is finished we are going to

Now that we have our sorted event queue we start pulling values off of this queue, starting with the point with the max y-value and ending with the point with the min y-value. For every point we pull off we are going to either remove a circle from the sweeping status or add a circle to the sweeping status.

Here we present an observation, if two circles intersect, then right before the sweeping status line L hits the intersection point p the two circles are neighbors in the sweeping Status S(L).

We show the psuedo-code here for reference.(It is nearly identical to the psuedo code given for the line segment problem.)

```
Build a max-heap Q on all min and max y-value points on the circles with their y-
values as the keys.
S(L) = null;
while (Q is not empty){
      p = extract-max(Q)
      if(p is the upper point of a circle S){
            insert the circle S into S(L) // This insertion is based on what circle
                                          // is intersected first by the sweeping
                                          // status line L. This value can be
                                          // found in constant time, and because
                                          // we are using a balanced BST for our
                                          // sweeping status, all necessary
                                          // operations take O(log n) time to
                                          // perform. (insert, delete, and search)
            find S_L and S_R, the two neighbors of S in S(L);
                  if(S intersects S_L || S intersects S_R){ return true; }
      }
      if(p is the lower point of a circle S){
            find S_L and S_R
            remove S from S(L)
            if(S_L intersects S_R){ return true }
      }
}
```

Using the above algorithm we spend **O(n log n)** time to build the max-heap, then as move from the maximum y-value to the minimum y-value, at the very most every circle will perform an insertion and a deletion, and four finds. Because each of these operations takes **O(log n)** time to perform and at most we are performing six per circle, then our final runtime becomes **O(n log n)** time.

**Psuedocode**
Not Required
**Correctness**
Not Required
**Time Analysis**

4. (20 points) Let S be a set of n disjoint line segments in the plane, and let p be a point not on any of the line segments of S. We wish to determine all line segments of S that p can see, that is, all line segments of S that contain some point q so that the open segment pq doesn't intersect any line segment of S. Refer to Fig. 3 as an example. Design an O(n log n) time algorithm for this problem. (Hint: use a rotating half-line with its endpoint at p to sweep the plane.)

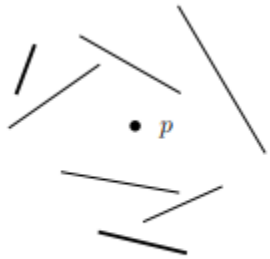


Figure 3: An example for Question 4: $p$ can see all segments except the two thick segments.

**Algorithm Description**

For this problem we first have to do some preprocessing. We need a list of points, each line segment will contribute one point to this list. The trick is that this point needs to be the first point that would be given if we were to draw this line segment in the direction of increasing angle Θ using polar coordinates. Given two points of a line segment this point can be found in constant time. We simply compute the angle theta for each point **O(1)** (constant time) and we return the point that gave the smaller angle Θ, however if the angle measure between the two points is greater than 90 degrees or π radians, then we return the other point instead. Now that we have this list of n points, we need to sort this list in polar coordinates by increasing angle Θ. This operation will take **O(n log n)** time to perform. Once the list is sorted, we can now use a rotating half-line with its endpoint at p to sweep the plane.

We show the psuedo code for our algorithm here:

1. In the order of the above given list pull off a point.
2. If that point is the beginning of a line segment, add that line segment to the sweeping status S(L). Also add the other point of the line segment to the event queue. Note that the sweeping status is storing the lines in the order that they are being intersected by the sweeping line S(L). Also note that when a point is added to the event queue, if the point that is added crosses the $\Theta=0$ angle mark from the original point than $2\pi$ radians must be added to the angle of the second point to correctly account for ordering within the event queue.
3. If the point is the endpoint of a line segment then it is removed from the sweeping status S(L).
4. This procedure continues as stated above until all points have been pulled from the event queue. A critical part of this algorithm is that every line segment that occupies the first slot in the sweeping status S(L) is visible to the point p. So the problem becomes one of keeping track of the first item in the sweeping status and reporting that it is visible. This visibility check is performed at every event. There is however one important point, when we start this algorithm we are only guessing that we can see the first point given in the list, it may be that we cannot see it because some other line segment may be blocking our view. Thus we need to traverse the original event queue by at most every line segment or $\pi$ radians. Whichever comes first, and for this extra half loop we may have to report that we cannot actually see a line segment, because our first $\pi$ radians in the sweep were guesses. Even if this is the case it does not destroy our overall runtime of **O(n log n)**. Thus we can report which line segments are and are not visible in **O(n log n)** time.

**Psuedocode**
Not Required
**Correctness**
Not Required
**Time Analysis**
This algorithm works because first do n **O(1)** operations (find the first encountered point), then we sort those points **O(n log n)**, and for every point we are performing at most one insertion, one deletion, for both the line segment and the points in it. Which makes our overall runtime **O(n log n)** time.