Brigham Michaelis
April 13, 2018
A00364835

Computational Geometry Assn.5

1. Let S1 be a set of n disjoint horizontal line segments and S2 be a set of n disjoint vertical line segments.

(a) Design a plane sweeping algorithm to compute the number of intersections of the segments in S1 ∪ S2 in O(nlog n) time. Since no two horizontal segments intersect and no two vertical segments intersect, an intersection only happens between a horizontal segment and a vertical segment.

Note that the total number of intersections could be as large as $O(n^2)$, but your algorithm should compute this number in O(nlog n) time. This implies that you cannot report these intersections one by one. **(15 points)**

Algorithm Description
Before we provide the description for the algorithm, we need to establish some facts.
First we are going to be using range trees.
Second, this is a 1-D problem.
Third, because this is a 1-D problem:
Range Reporting takes up O(k + log(n)) time.(following the algorithm discussed in class.) and
Range Counting takes up O(log n) time. This can be accomplished by having each node keep track of the number of nodes (or line segments) in its subtree.

So how are we going to do this?
First we sort all of the line segments from top to bottom. Now we are going to do a plane sweep from top to bottom. Our sweeping status is going to be all of the vertical line segments that are active or that have have been inserted into our sweeping status but not removed. The sweeping status will be a 1-D range tree and as we insert and remove these line segments from our sweeping status we will be updating the the number of line segments that are stored in each subtree of our sweeping status. Now that have a sweeping status, and it only takes log(n) time to insert and delete from this range tree (or sweeping status.) and each node knows how many line segments are in its subtree we can now perform range counting on this sweeping status in O(log(n)) time. Our range will be the horizontal line segments, their starting and ending points. Basically we are going to perform the range counting operation on our sweeping status. This will be done by computing the in-order successor to the leftmost point on the horizontal line segment for our sweeping status and computing the predecessor to the rightmost point on the horizontal line segment for our sweeping status. With these two points, and with our sweeping status in the form that it is in, we can compute a running sum of the number of intersections as we move our sweeping line down the plane using the algorithm discussed in class by counting over and over again the number of nodes stored in each Lowest Common Ancestor.
TO RECAP:
First we sort the all the line segments by y-coordinate.
Then we start our sweeping status from the top of the plane and move downward.
When we hit the first given point of a vertical line segment we place it onto our sweeping status, this takes **O(log(n))** time.
When we hit the last given point of a vertical line segment we remove it from our sweeping status, this takes **O(log(n))** time.

When we encounter a horizontal line segment we treat it as a range and report the number of nodes that are stored in the subtree of the Lowest Common Ancestor of the appropriate successor and predecessor. This takes **O(log(n))** time.

Psuedocode

Not Required.

Correctness

This works because we first sorted the line segments and then performed **n** insertions and **n** deletions and **n** range counting operations. We have only used up **O(n*log(n))** time to compute the number of intersections contained in all the line segments.

Time Analysis

Sorting all **2n** line segments: **O(n*log(n))** time
Inserting all **n** vertical line segments into the sweeping status: **O(n*log(n))** time
Removing all **n** vertical line segments into the sweeping status: **O(n*log(n))** time
Range Counting for all **n** horizontal line segments:**O(n*log(n))** time
Our total runtime is **O(n*log(n))** time.

(b) Design a plane sweeping algorithm to report all intersections of the segments in S1 ∪ S2 in O(n*log(n) + k) time, where k is the total number of intersections of all segments. So this is an output-sensitive algorithm because the running time is not only a function of the input size but also a function of the output size. **(15 points)**

Algorithm Description

This problem is basically a continuation of the previous problem, only now we have to use Range Reporting. As mentioned before this operation only takes **O(log(n)+k)** time. But we need to perform this operation **n** times. So what does this mean? It means that for every horizontal line segment. We are going report the subtree of the Lowest Common Ancestor of the appropriate successor and predecessor. Basically we are going find the appropriate Lowest Common Ancestor **n** times and report every line segment in the Lowest Common Ancestor.

Psuedocode

Not Required.

Correctness

This works because we are able to report all the nodes in a 1-D range tree in **O(log(n)+k)** time, and we are going to do this operation n times.

Time Analysis

Our total time is **O(n*log(n)+k)** time. Because we are using a **O(log(n)+k)** time algorithm **n** times.

2. (30 points) Given a set P of n points in the plane, design an efficient data structure to answer the following segment dragging queries: for each query, we are given a horizontal line segment s, and the goal is to report the first point of P that will be hit by s if we drag s vertically downwards (e.g., see Fig. 1). Each query segment s is specified by the coordinates of its two endpoints (x1(s), y(s)) and (x2(s), y(s)), with x1(s) ≤ x2(s).

Please describe your data structure and the preprocessing time and space. Please also describe how your query algorithm works and the query time. You will receive full points if your data structure can achieve the following performance:
The preprocessing time (i.e., the data structure construction time) is **O((n)*log(n))**.
The space of the data structure is **O((n)*log(n)).** and
The query time is **O(log^2 n).**
For simplicity, you may make a general position assumption that no two points of P have the same x- or y-coordinate. **(Hint: use range trees.)**

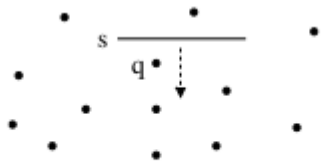

Figure 1: The point q is the first point that will be hit by dragging the segment s vertically downwards.

Algorithm Description
We first start off by talking about how we would solve this problem in 1-D. If we had a set of n points on the x axis, we could insert/sort them into a 1-D range tree. This would consume **O((n)*log(n))** time. Our space would be **O((n)*log(n)).** As for our query time that would be **O(log(n)).** Due to the time that it would take to find the in-order successor to the line s on the x axis. The in-order successor being defined as the smallest point in the tree whose value is large than the x position of the line s.

We are going to do the same thing except in 2-D. This can be accomplished by building a 2-D range tree from all of the **n** points P in the plane. This preprocessing time takes **O((n)*log(n))** time, because each insertion takes up **O(log(n))** time, and we are doing this **n** times. Because we are using a 2-D range tree we also know that our space is **O((n)*log(n)),** As discussed in class. Finally where as in class we talked about reporting all the points in a rectangular range, we are not going to do that. Instead as we perform an in-order successor search, of the y-value of the line s, on each of the **log(n)** horizontal strips. We are going to keep track of the smallest in-order successor we find among all of the **log(n)** strips that falls between the endpoints of the line s. In essence we are performing the 1-D in-order successor search operation on each of the **log(n)** strips that make up the 2-D range tree.
Thus our query time for this problem is: **log(n)*log(n) = O(log^2 n).**

Psuedocode
Not Required.
Correctness
This works because we are making a small modification to the algorithm that was developed in class. Mainly instead of a range box we are asking for the closest in-order successor to the range line.
Time Analysis
Preprocessing time: **O((n)*log(n)).**
Space: **O((n)*log(n)).**
Query time: **O($\log^2$ n).**

3. (10 points) Let S be a set of n points in the plane. Let T be a k-d tree built on S, as described in class. Consider the following queries: Given a point p, determine whether p is in S (so the answer is either “Yes” or “No”). Using the k-d tree T , give an algorithm that can answer the queries in O(log n) time each. Please explain how your algorithm works and argue why the time is O(log n).

Algorithm Description

First as discussed in class we know that the height of a tree is Log(n). Now because each leaf in the k-d tree is an actual point, and the height of the tree is Log(n), then we can determine if a point p is in the tree in Log(n) time. By simply doing a special version of a search, where at every node of the tree we ask:

Does this point reside on one side of the line made by the node v, or the other.

> We then traverse down the appropriate side of the tree and continue to call this recursive search function until we hit a leaf and check if the leaf is the point in question. If it is return yes, otherwise return false.

Psuedocode

Not Required.

Correctness

This works because we are able to reduce our problem size by half, at every node. We are either to the right or to the left or we are above or below the line made by node v. Because we are able to reduce the problem size by half at every step we are able to achieve a time of O(log(n)).

Time Analysis

The time is O(log(n)), because the height of the tree is Log(n), and at every level we are able to reduce the problem size by half as mentioned above.