Line Sweep Algorithms

Hariank Muthakana

January 16, 2015

1 Introduction

Line sweep is a useful technique in computational geometry problems. Line sweep algorithms operate by moving or "sweeping" a line over a plane, processing the data as it passes over it. As the line meets key line segments and points, it holds information about what entities it is currently intersecting and keeps them in a certain order.

2 Data Structures

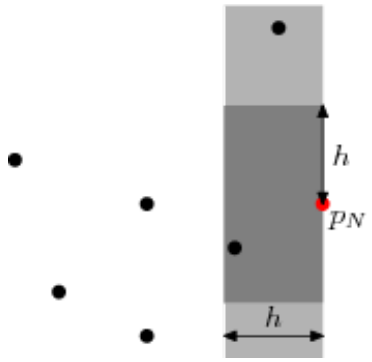
To properly use this technique, we would like to be able to efficiently store and manipulate the current points on our sweep line. We will be using a *balanced binary tree*, which provides insertion and removal operations in $O(\log N)$ time.

The tree represents the line. As the line begins to intersect a segment, we insert it into the tree as a node, and when it leaves we remove it. Note that the balanced binary tree keeps its nodes in a *fixed order*. This ensures that points and coordinates are sorted by their x/y value.

3 Pairwise Distances

Consider the problem of finding the two closest points in a set of points. We can naively do this in $O(N^2)$. However, a better solution exists with a sweep approach.

First, we sort the points by x-value and process from left to right. For each point, we only need to consider distances to points to the left. Suppose the shortest distance we have so far is h.

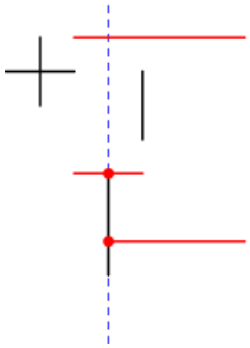


As shown in the figure, we sweep a window of width h, continually aligning its right side with the current point. At each step, we can further narrow our point selection to a set of y values based on h (the shaded box). Any new points that would be closer to the current point would be found in this box. It can be mathematically proven that the number of points in the box is constant in relation to N. Therefore, we can simply find the one with the smallest distance among these, update h, and proceed to the next point.

The initial sort takes $O(N \log N)$ time. There are N insertions and N deletions from the sweep window, each taking $O(\log N)$ time. The inner comparison to the points in the box takes $O(1)$ time for each point, yielding $O(N)$. Therefore, we have a final complexity of $O(N \log N)$

4 Finding Intersections

Another problem we can solve with line sweep is the intersection problem. Given a set of axis-aligned line segments, we would like to count the intersections. Consider the problem with horizontal and vertical line segments. We can use a vertical sweep line as shown:

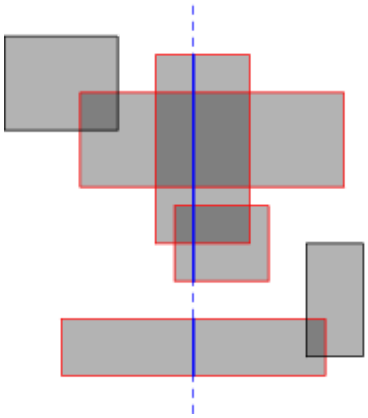


Here, there are three key "events" that the sweep line must handle: the left and right endpoints of a horizontal segment, and a vertical segment. We will process from left to right, and maintain a set of *active* horizontal segments, sorted by y-value. (Note that we only process at one of these three events, since moving the sweep line one unit at a time would take too long).

A horizontal endpoint corresponds to insertion/removal from the tree. However, a vertical segment is trickier. We can do a *range search* on the balanced binary search tree, using the top and bottom of the vertical segment, to identify which horizontal y-values are intersected. To do this range search, we must include extra information in the tree. Each subtree must know its own size. This brings the complexity to $O(N \log N)$.

5 Computing Areas

A final problem we can tackle with line sweep is computing areas. Consider a set of rectangles, parallel to the axes, that may or may not overlap. We aim to find the total area they cover. We begin with a similar approach to the intersection problem. We maintain an active set of rectangles and a vertical sweep line. Our "events" are now the left and right edges of the rectangles:



We may have horizontal lengths, but we need vertical lengths to compute the area. The question is how we can find the total vertical length of the sweep line that is contained in a rectangle (the shaded parts of the line). Once we get this, we can simply multiply it by the horizontal distance between the current events.

The trick is to run a *second sweep* in an inner loop, but with a horizontal line moving downwards. For each rectangle in the active set, we run the same line sweep vertically. As long as we are in a rectangle (which we can ensure by counting how many sides we have crossed), we add to the vertical length. By

running this inner loop for each pair of events, we are able to sum up all of the areas.

Here we can actually use a boolean array to maintain the active set. This allows for a complexity of $O(N^2)$.

6 Problems

1. (USACO FEB12 Silver) Farmer John has purchased a new machine that is capable of planting grass within any rectangular region of his farm that is "axially aligned" (i.e., with vertical and horizontal sides). Unfortunately, the machine malfunctions one day and plants grass in not one, but N ($1 \leq N \leq 1000$) different rectangular regions, some of which may even overlap.

   Given the rectangular regions planted with grass, please help FJ compute the total area in his farm that is now covered with grass.

2. (Topcoder) You have discovered a region in need of power, and you have decided to build a power line there. The region is a Cartesian plane, and consumers are represented as points on the plane. Your power line will be a single straight line that is horizontal, vertical, or parallel to any of two diagonals. Each potential consumer will purchase power from your line if and only if the distance between him and your line is less than or equal to D. This distance is measured as the Euclidean distance. You would like to maximize your profit by maximizing the number of consumers using your power line. You are given `int[]`s x and y containing the coordinates of the consumers. The ith elements of x and y represent the x and y coordinates of the ith consumer. You are also given an `int` D, the value described above. Build your power line to maximize the number of consumers that will use it, and return this maximum number.

7 Other

7.1 Further Reading

- Convex Hull
- Rotating Calipers

7.2 References

- Images from Topcoder