6.854 Advanced Algorithms

Lecture 17: 11/8/2004 Lecturer: David Karger

Scribes: Matt Rasmussen

Sweep Line

Definition 1 ***Sweep Line Technique:*** *Given some planar problem, sweep a line through the plane dealing with events that occur on the line and leaving behind a solved portion of the problem.*

17.1 Convex Hull

The Convex Hull problem is to find the smallest enclosing convex polygon of a set of given points in the plane.

17.1.1 Algorithm

One method for solving the convex hull problem is to use a sweep line technique to find the upper envelope of the hull. The lower evelope of the convex hull can be found by rerunning the following algorithm with only slight modifications.

Use a vertical sweep line that sweeps from negative infinity to positive infinity on the x-axis. As the line sweeps, we will maintain a partital convex hull for the points left of the sweep line. When the sweep line crosses a new point we will need to update the partial convex hull to include the new point. After the sweep line has gone past all the points, we will have a complete upper envelope of the convex hull.

To determine the order in which the sweep line will cross points we can use a priority queue such a min-heap that will order points by their x-coordinates. Therefore, the only remaining issue is how to insert a new point into an existing convex hull. When inserting a new point, two cases arise: (1) either the existing convex hull can be extended to include the new point or (2) the new point requires the existing convex hull to be modified. Informally, we can distinguish the two cases by noticing that in case (1) the new point causes a "right turn" in the hull's boundary, whereas in case (2) the new point causes a "left turn". This can be determined mathematically by finding the angle between the right most line segment of evelope and the line segment of the right most evelope point and the new point. If the angle is less than 180 degrees then we have a case (1), otherwise we have a case (2).

case 1: Since the new point can be safely added to the existing evelope without violating its convexity, the new point is directly added to the list of evelope points.

case 2: In this case, the new point cannot be safely added to the existing evelope without violating its convexity, therefore, the existing evelope must be modified to accommodate the new point. This can be done by the following procedure. At the new point the evelope. Peform a convexity check on the predecessor to the new point. If the convexity check fails, delete the point from the evelope. Now the new point has a new predecessor. Repeat the convexity check and deletion on the predecessor of the new point until the convexity check is satisfied, at which point we will have a valid upper evelope again.

17.1.2 Analysis

Let there be n points in the problem. Creating the min-heap and performing n extract-mins will have a $O(n \log n)$ runtime. Convexity checks take a constant amount of algebra and therefore take $O(1)$. Case (1) simply extends a linked list in $O(1)$ and may occur a maximum of n times, which implies a $O(n)$ runtime. To find the runtime contribution of case (2), we bound the total number of deletions that may be made from the evelope over the course of the algorithm. This bound is n since each point can be deleted at most once. Therefore, the amortized cost of case (2) is also $O(n)$. This gives a total runtime of $O(n \log n)$.

17.2 Segment Intersections

In the Segment Intersections problem, we are given n line segments and must output the coordinates of all pair-wise intersections.

17.2.1 Algorithm

In this algorithm, we will use a horizontal sweep line that sweeps from positive infinity to negative infinity along the y-axis. The idea, will be to output each intersection as we cross it. Also we will strive to make the algorithm *output sensitive*. That is, although there may be $O(n^2)$ intersections, which would require an $O(n^2)$-time algorithm, we will run in much less time if the number of intersections k is much less than n.

Let a segment be considered "active" if it crosses the current sweep line. As the sweep line sweeps, we will encouter three types of events:

1. new segment becomes active
2. old segment becomes inactive
3. two active segments cross

Events (1) and (2) can be handled with a min-heap containing segment endpoints that are ordered by their y-coordinates. In dealing with the third case, the key idea is that only "neighboring" segments on the sweep line can cross. To provide a quick lookup of neighboring segments we can use a Binary Search Tree (BST) to store the segments by the x-coordinate of their intersection with the sweep line. After inserting a new line into the BST, determine if it will eventually cross with its neighbors. If so, insert a crossing event into the event queue. Later, when a crossing event occurs, we output an intersection, swap the order of the lines segments in the BST, and find their two new neighbors and possible future crossings.

17.2.2 Analysis

Inserting and extracting line segments activations and deactivations from the event queue will take a total of $O(n \log n)$ time. The total time inserting into the BST will take $O(n \log n)$ and the total deletes from the BST will take $O(n)$. The number of crossing events is k, therefore the total time needed for inserting crossing events into the event queue is $O(k \log n)$. Therefore, the total runtime is $O((n + k) \log n)$.