6.854 Advanced Algorithms	
Lecture 1: 11/8/2004	Lecturer: David Karger
Scribes: Aidan Downes	

Sweep Line Algorithms

1.1 Introduction

Sweep line algorithms are used in solving planar problems. The basic outline of a sweep line algorithm is as follows:

1. Sweep a line across problem plane.
2. As the line sweeps across the plane, events of interest occur, keep track of these events.
3. Deal with events that occur at the line leaving a solved problem behind.

1.2 Convex Hull

Given a set of points in a plane, the smallest convex polygon that encloses all of the points in the set is the convex hull of the set. A polygon is convex if any line segment connecting two points in the polygon is entirely closed by the polygon. A convex hull can be represented by it's vertices in cyclic order.

1.2.1 Lower Bounds for Convex Hull Algorithms

The lower bounds for Convex Hull Algorithms is $\Omega(n \log n)$. This can be proved by reducing the sorting problem to an instance of the convex hull problem. One possible reduction is shown below:

1. Let $x_1, x_2, ..., x_n$ be the input
2. Let $p_i = (x_i, x_i^2)$
3. Find the convex hull for all p_i's.
4. Return the x value for each point (remember a convex hull is represented by its vertices in cyclic order).

Let $k(n)$ be the time to compute the convex hull. The total time then is $O(n + k(n))$. Since the lower bound for sorting (using the comparison model) is $\Omega(n \log n)$ then $k(n) = \Omega(n \log n)$.

1.2.2 Sweep Line algorithm for Convex Hulls

This algorithm finds the upper envelop of a convex hull. A similar algorithm can be used to find the lower envelop. The algorithm uses two data structures. The first data structure stores the points that make up the hull of points seen so far. The second data structure stores the points not seen yet in sorted order by x-coordinate. A simple list can be used for first data structure while a heap is well suited for the purpose of the second data structure.

The events of interest for this sweep line algorithm is the arrival of a new point as the sweep line travels from left to right. As each event occurs, the algorithm updates the current convex hull. There are two cases to consider.

Case 1: The y-coordinate of the new point is less than or equal to the y-coordinate of the previously encountered point. In this case we add the new point to the convex hull.

Case 2: The y-coordinate of the new point is greater than the y-coordinate of the previously encountered point. In this case add the new point but we remove the previously added point. We keep backtracking, removing previously added point until we find a point with a y-coordinate less than or equal to the y-coordinate of the newly added point.

The full algorithm is as follows:

1. insert point into heap ordered by x-coordinate
2. Delete-Min from heap
3. Run Convexity check
4. if not convex delete elements from hull backwards till convex check passes
5. add new point to hull
6. go to line 2

Runtime:

Line 1 runs in $O(n \log n)$. Line 2 runs n times with $O(logn)$ each time and $O(n \log n)$ in total. The convexity check in line 3 is $O(1)$ and runs n times. To analyze line 4, note that an element can only be deleted once. For each element deleted, constant work is performed, and therefore $O(n)$ work is peformed in total. Therefore the total running time is $O(n \log n)$.

1.3 Segment Intersections

Segment intersection algorithms returns the coordinates of all pairwise intersections given a set of line segments as input. The naive algorithm tests every segment pair for intersection and runs in $O(n^2)$. The sweep line algorithm solves the problem in $O((n + k)lgn)$ time. If k is not huge (less than n^2) then the sweep line algorithm is a significant improvement over the naive algorithm.

The sweep line algorithm we will use to solve the segment intersection sweeps a line from the top to the bottom of the plane, reporting intersections as they are encountered. As the sweep line moves across the plane it forms intersections with line segments in the plane. When a line segments intersects the sweep line, we say it is active.

Note that in order for two segments to intersect they must be adjacent to each other at some point on the sweep line. This allows us to only check for intersections between adjacent pairs of segments. The algorithm uses two different data structures. One data structure stores all active segments ordered by intersection point with the sweep line. The other data structure is a priority queue which stores events ordered by the distance from the sweep line.

The events of interest in this sweep line algorithm are:

1. the sweep line encounters(intersects) a new segment. The line segment becomes active.
2. an active segment no longer intersects the sweep line. The segment is no longer active.
3. two active segments intersect each other

Case 1: Insert the line segment into the sweep line status data structure. Then check if the line segment intersects any of its new neighbors. If there is an intersection add the intersection to the event queue.

Case 2: Remove the line segment from the sweep line status data structure. The former neighbors of the disposed line segment are now adjacent to each other. Check if the line segments intersects. If they do, add the intersection to the event queue.

Case 3: When there is an intersection event, report the intersection and swap the involved line segments positions in the sweep line status data structure. Each of the swapped line segments have one new neighbor. Check for intersections between the swapped line segments and its new neighbors. Add any intersections to the event queue.

The full algorithm is as follows:

1. Add segment activation and deactivation events to event queue
2. Dequeue an event from the event queue
3. If the event is an activation event then add the line segment to the sweep line status data structure and check for intersections.
4. If the event is a deactivation event, remove the line from the sweep line status data structure and check for intersections.
5. If the event is a intersection event, report the intersection, swap the lines in the sweep line status data structure and check for intersections.
6. Go to line 1.

Runtime:

Line 1 runs in $O(n)$ time. The number of times that the loop that is lines 2-6 runs is equal to the number of events. There are k swaps, n arrivals and n departures. Therefore there are $O(n+k)$ events and the loop runs $O(n+k)$ times. If we use a binary tree to represent the sweep line status and a heap to represent the priority queue, each loop iteration performs $O(logn)$. The loop total runtime is $O((n+k)logn)$ which is the total runtime of the algorithm.

1.4 Voronoi Diagrams

Given a set of points a plane, we want to be able to answer nearest neighbor queries. More specifically, given any point in the plane we want to return the closet point in the given set of points. One idea for solving the problem is to divide the plane regions according to closest point. This reduces the problem to determining the region of the query point.

Voronoi diagrams are the partitioning of a plane with n points into n convex polygons such that each polygon contains exactly one point and every point in a given polygon is closer to its central point than to any other. Voronoi diagrams can be used to solve the nearest neighbor problem.

1.4.1 Size of voronoi diagrams

The size of a voronoi diagrams is $O(n)$.

Proof: Add a point at infinity such that all edges (including infinite edges) have two endpoints. Euler's formula states that $n_v - n_e + n_f = 2$. Not that the number of faces, n_f, in the voronoi diagram is equal to n. Also not that every voronoi vertex has degree 3 but every edges has two end points.

$$2n_e \geq 3(n_v+1) \tag{1.1}$$
$$2(n+n_v-2) \quad geq \quad 3(n_v+1) \tag{1.2}$$
$$n_v \leq 2n-7 \tag{1.3}$$

Since the number of voronoi vertices is linear to the number of input points. The size of the voronoi diagram is $O(n)$.

1.4.2 Sweep line algorithm for construction Voronoi diagrams

In this algorithm we sweep a line from top to the bottom of the plane and maintain a beachfront of parabola, points equidistant for the sweep line and seen sites (input points). Our algorithm maintains the following invariant: The voronoi diagram is complete and unchanging behind the beachfront.

As the sweep lines moves, the parabola forming the beach front also descend.

$$(x-x_f)^2 + (y-y_f)^2 = (y-t)$$

is the equation for a parabola where (x_f, y_f) is coordinate of the focus point and t is the parameter for the sweep line. Fixing x and solving for $\frac{dy}{dt}$ produces

$$2(y - y_f)\frac{dy}{dt} = 2(y - t)(\frac{dy}{dt} - 1) \quad (1.4)$$

$$\frac{dy}{dt} = -\frac{y - t}{t - y_f} \quad (1.5)$$

Thus the higher the focus of the parabola the slower the parabola descends.

The events of interest in our voronoi diagrams are:

1. Site Events. This occurs when the sweep line crosses a new site. When this occurs we create a new parabola for the site. Initially the parabola is a vertical line but it widens to normal parabola as the sweep line continues moving. The new parabola creates two new intersections of parabola in the beach front. We insert these intersections into ordered list of parabolas. (Parabolas are represented by the intersection points).
2. Movement Events. These events occur as the sweep line moves across the plane, changing the beach front parabolas. There are two cases to consider.
   (a) A inner parabola crosses an outer parabola. This does not happen. Consider the moment when the inner parabola becomes adjacent with the outer parabola. The inner parabola has a higher focus than the outer parabola so it descends slower. Therefore the inner parabola does not cross the outer parabola.
   (b) Parabola hits the intersection of two other parabola. All three sites are equidistance from the intersection. The middle parabola cannot absorb the intersection because its focus is too high and thus it descends slower. Therefore the middle parabola disappears from the beach front. This is also known as the circle event.

In recap, site events create parabolas and circle events destroy parabolas. The full algorithm is as follows:

1. Add all site events to the event queue.
2. While queue is not empty:
3. Dequeue next event.
4. If its site event add parabola to the beachfront and compute future circle events with neighbors.
5. If its a circle event then remove parabola from beachfront and check for future circle events.

Runtime: Each event requires $O(logn)$ time to process, mostly updating data structures for the event queue and parabolas. And there are $O(n)$ events. Each site is responsible for one at most one site event and one circle event. The total runtime is therefore $O(n \log n)$.