Computational Geometry: Lecture 4

Don Sheehy

April 6, 2010

1 Linesweep for Line Intersections

Last time we talked about the **line sweep** paradigm for the problem of reporting all the intersections of a collection of line segments in the plane. The key insight was to look at a one-dimensional slice of the input at a time. Let's assume that none of the input segments are exactly horizontal. The input segments intersect a horizontal line at a a collection of points. As we move this line down continuously, the points corresponding to the segments move. We can therefore use the binary tree technology that we understand for maintaining the order of a set of points on a line and just make sure that we update it every time the order of the points changes. In fact, if the order of the points does change for any pair, then that means we have an intersection of the corresponding lines segments and we want to report that anyway.

Last time, we described in pictures how such an algorithm might work. Today, we'll actually write some pseudocode and walk through a sample run of the algorithm.

There are two data structures that we need to maintain. First, there is a binary tree T that contains the line segments that intersect the current sweepline. Second, there is an event queue that contains the events to be processed priority order by decreasing y value. The Q is initialized with the top and bottom vertices of each line segment.

For convenience, let's abstract out an algorithm that checks if two new line segments intersect some time in the future and adds the intersection even to Q.

Algorithm 1 Check

```
  Input: line segments $A$ and $B$ and time $t \in \mathbb{R}$
  if $A$ and $B$ intersect at $(x, y)$ and $y > t$ then
    Add event $AB$ to $Q$
  end if
```

Now, the main algorithm is just a loop that pulls the next element off of Q and processes it. There are only three types of events: additions (top of a line segment), removals (bottom of a line segment), and intersections. The ProcessEvent algorithm does the main work.

Algorithm 2 ProcessEvent(e)

```
Input: an event $e$ consisting of of one or two line segments and a time $t \in \mathbb{R}$.
if $e = (A+)$ is an addition then
    insert $A$ into $T$
    Check($A, pred(A), t$)
    Check($A, succ(A), t$)
end if
if $e = (A-)$ is an removal then
    Check($pred(A), succ(A), t$)
    Delete $A$ from $T$
end if
if $e = (A, B)$ is an intersection then
    Check($A, succ(B), t$)
    Check($B, pred(A), t$)
    Swap the order of $A$ and $B$ in $T$
end if
```

For the input in Figure 1, the event queue will have the following events.

- A+
- B+
- C+
- **BC**
- D+
- **AB**
- A-
- **CD**
- C-
- B-
- D-

The events in bold are the intersections discovered in the course of the algorithm. The state of the tree T as we go through these events is presented in Figure 2. This assumes that swap was implemented by deleting and reinserting and also that no re-balancing of the tree was done.

Clearly, in order to get good runtimes we need to use a balanced binary tree data structure. You should be able to convince yourself that using any good priority queue data structure, that the running time is bounded by $O(n \log n |I| \log |I|)$. Since $|I|$ may be as large as $O(n^2)$, this means that our algorithm has a running

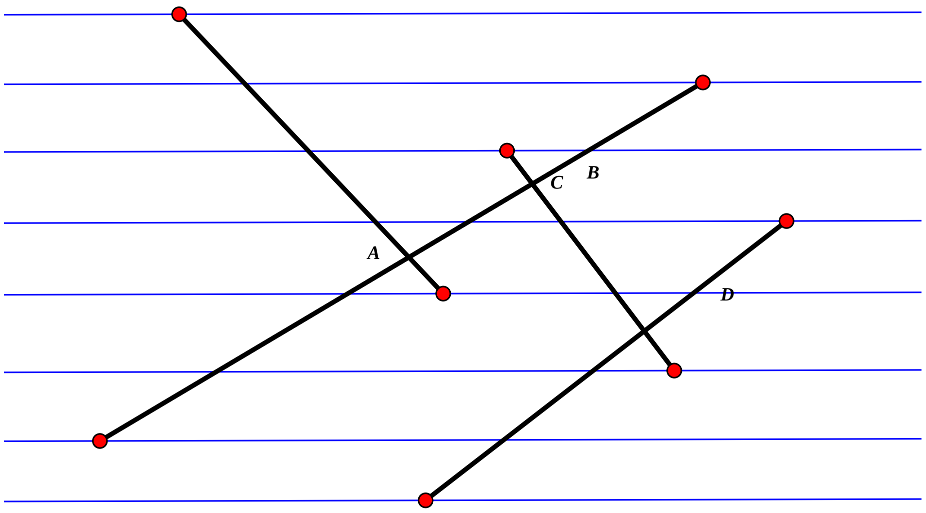

Figure 1: A sample input to for the Line Intersection problem. The horizontal lines mark time and are included only to make the vertical ordering visually obvious.

time of $O((n + |I|) \log n)$. It should be noted that better algorithms exist and in fact, it is possible to achieve $O(n \log n + |I|)$ using a random incremental algorithm similar to the one we saw for 2D convex hull.

2 Linear Predicates

Now, we're going to change gears a little and take an idea from sweeplines to understand those determinant calculations a little better.

With the sweepline, we restricted our view of the input lines by one dimension and treated them as points. Let's turn this idea on it's head and think of our 2D problems on points as being a restriction of a 3D problem on lines. More specifically, given a set of points $P \subset \mathbb{R}^2$, treat them as points on the plane $z = 1$ in $\mathbb{R}^3$. This lifting of the plane into $\mathbb{R}^3$ has a nice effect in that it allows us to replace our statements about affine combinations with statements about linear combinations. For example the affine combination

$$\begin{bmatrix} p_x \\ p_y \end{bmatrix} = \sum_{i=1}^{n} \alpha_i \begin{bmatrix} x_i \\ y_i \end{bmatrix}, \text{ where } \sum_{i=1}^{n} \alpha_i = 1 \tag{1}$$

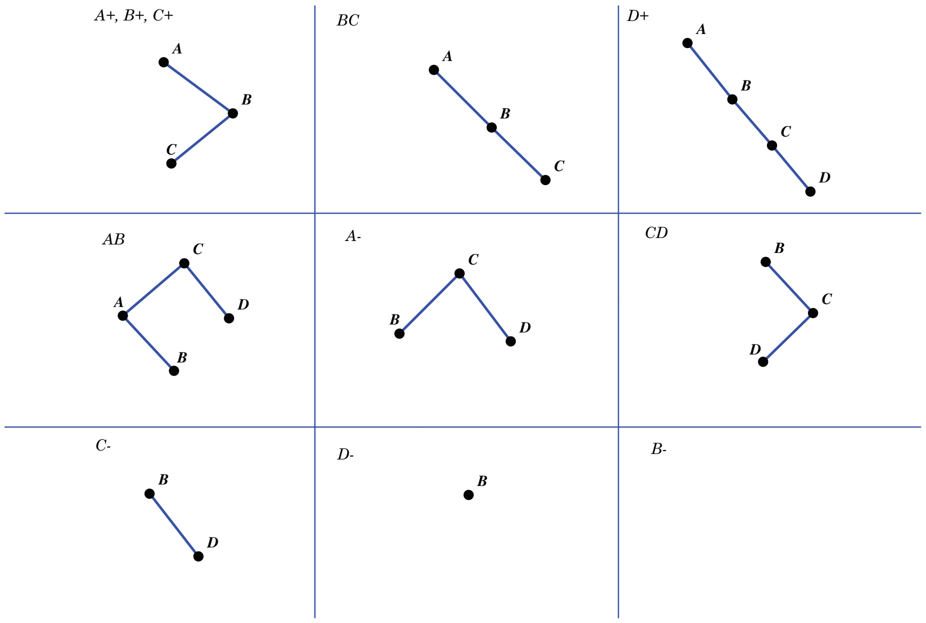

Figure 2: The state of T as we walk through the algorithm.

can be rewritten as

$$\begin{bmatrix} p_x \\ p_y \\ 1 \end{bmatrix} = \sum_{i=1}^{n} \alpha_i \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix}. \qquad (2)$$

Now we're just doing linear algebra. Each point in the plane is associated with the span of a vector, that is, a line through the origin. We saw this explicitly when we expanded the determinant formulation of CCW. Recall we wrote:

$$\det \begin{bmatrix} b_x - a_x & c_x - a_x \\ b_y - a_y & c_y - a_y \end{bmatrix} = \det \begin{bmatrix} a_x & b_x & c_x \\ a_y & b_y & c_y \\ 1 & 1 & 1 \end{bmatrix} \qquad (3)$$

This gave a cleaner formulation of CCW because it implicitly handled the difficulty of not having a fixed origin. Recall that this was the main difference between the point sets we have been dealing with and a vector space.

The lifted coordinates of the points are known as the homogenous coordinates in projective geometry. You can think of the word "projective" in this setting as describing the understanding of the points as projections of a line onto the plane $z = 1$.

3 The InCircle Test

Here is a new problem. Given three points a, b, c (not all on a line) in the plane, determine if a fourth point d is inside, outside, or on the unique circumcircle of a, b, c (denoted $cc(abc)$).

One possible approach to this might be to try to find the center p of $cc(abc)$ and then compare $|p - a|$ to $|p - d|$. This would work. In order to do so, we would set up some linear equations describing the perpendicular bisectors of the edges of $\triangle abc$. The common intersection is p. In fact, I'm willing to bet that such a routine has been implemented many times in buggy code all over the world.

We'd like to solve this problem in a similar manner to before. That is, we want to devise a matrix such that the sign of the determinant tells us the answer. The key to making this idea work is the so-called **Parabolic Map**. For a point $a = \begin{bmatrix} x \\ y \end{bmatrix}$, the parabolic lift of a is $a^+ = \begin{bmatrix} x \\ y \\ x^2 + y^2 \end{bmatrix} = \begin{bmatrix} a \\ |a|^2 \end{bmatrix}$.

Lemma 3.1. *A point d is inside the circumcircle of $\triangle abc$ if and only if d^+ lies below the plane through a^+, b^+, c^+.*

Proof. This will be in your next homework. $\square$

Now, we have seen how to check what side of a plane a point lies on; it is just our standard linear predicate. So, assuming the points a, b, c are oriented counterclockwise, the InCircle test can be written as follows.

$$\textsc{InCircle}(a, b, c, d) = \operatorname{sign}\det \begin{bmatrix} a & b & c & d \\ |a|^2 & |b|^2 & |c|^2 & |d|^2 \\ 1 & 1 & 1 & 1 \end{bmatrix} \tag{4}$$

The parabolic lifting generalizes to higher dimensions as well. That is we can test points in higher dimensional balls by adding a new coordinate for the lifting.

Next time, we'll look at an interesting use of the InCircle test known as the *Delaunay Triangulation*.