16 Line Segment Intersection (March 25)

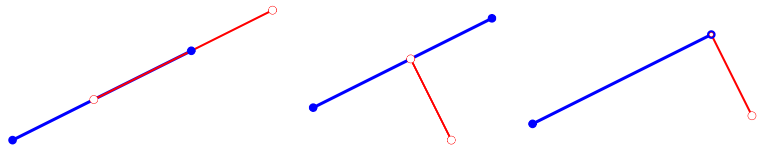
16.1 Introduction

In this lecture, I'll talk about detecting line segment intersections. A line segment is the convex hull of two points, called the *endpoints* (or *vertices*) of the segment. We are given a set of n line segments, each specified by the x- and y-coordinates of its endpoints, for a total of $4n$ real numbers, and we want to know whether any two segments intersect.

To keep things simple, just like in the previous lecture, I'll assume the segments are in *general position*:

- No three endpoints lie on a common line.
- No two endpoints have the same x-coordinate. In particular, no segment is vertical, no segments is just a point, and no two segments share an endpoint.

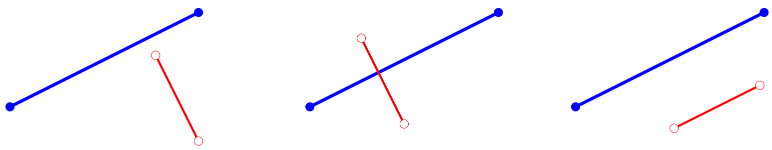
This general position assumption lets us avoid several annoying degenerate cases. Of course, in any real implementation of the algorithm I'm about to describe, you'd have to handle these cases. Real-world data is *full* of degeneracies!



Degenerate cases of intersecting segments that we'll pretend never happen:
Overlapping colinear segments, endpoints inside segments, and shared endpoints

16.2 Two segments

The first case we have to consider is $n = 2$. ($n \leq 1$ is obviously completely trivial!) How do we tell whether two line segments intersect? One possibility, suggested by a student in class, is to construct the convex hull of the segments. Two segments intersect if and only if the convex hull is a quadrilateral whose vertices alternate between the two segments. In the figure below, the first pair of segments has a triangular convex hull. The last pair's convex hull is a quadrilateral, but its vertices don't alternate.



Some pairs of segments.

We don't want to use a full-fledged convex hull algorithm just to test two segments. Fortunately, there's a simpler condition: *Two segments* $\overline{ab}$ *and* $\overline{cd}$ *intersect if and only if the endpoints* a *and* b *are on opposite sides of the line* $\overleftrightarrow{cd}$, *and the endpoints* c *and* d *are on opposite sides of the line* $\overleftrightarrow{ab}$.

To test whether two points are on opposite sides of a line through two other points, we use the same counterclockwise test that we used for bulding convex hulls. Specifically, a and b are on opposite sides of $\overleftrightarrow{cd}$ if and only if exactly one of the two triples a, c, d and b, c, d is in counterclockwise order. So we have the following simple algorithm.

```
TwoSegmentsIntersect(a, b, c, d):
    if CCW(a, c, d) = CCW(b, c, d)
        return false
    else if CCW(a, b, c) = CCW(a, b, d)
        return false
    else
        return true
```

16.3 A Sweep Line Algorithm

To detect whether there's an intersection in a set of more than just two segments, we use something called a *sweep line* algorithm. First let's give each segment a unique label — I'll use letters, but in a real implementation, you'd probably use pointers/references to records storing the endpoint coordinates.

Imagine sweeping a vertical line across the segments from left to right. At each position of the sweep line, look at the sequence of (labels of) segments that the line hits, sorted from top to bottom. The only times this sorted sequence can change is when the sweepline passes an endpoint or when the sweepline passes an intersection point. In the second case, the order changes because two adjacent labels swap places.[1] So "all" we have to do is check every pair of segments whose labels are ever adjacent in the label sequence.

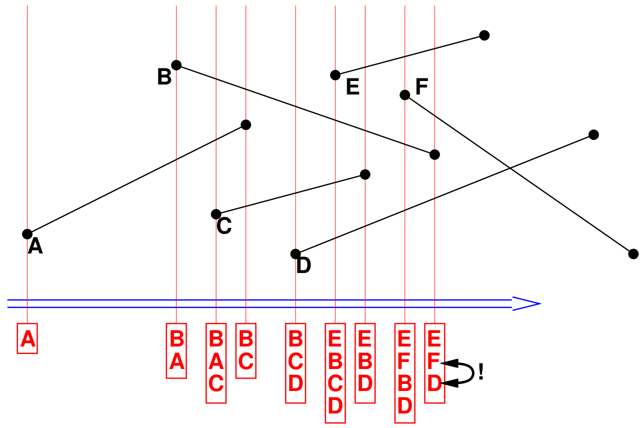
The algorithm begins by sorting the $2n$ segment endpoints from left to right by comparing their x-coordinates, in $O(n \log n)$ time. The algorithm then moves the sweepline from left to right, stopping at each endpoint.

We store the vertical label sequence in some sort of balanced binary tree that supports the following operations in $O(\log n)$ time. (Amortized time is good enough to get a worst-case time bound for the whole algorithm, so we can use a splay tree or a scapegoat tree.) The search tree does *not* store any explicit keys, only segment labels.

- **Insert** a segment label.
- **Delete** a segment label.
- Find the **neighbors** of a segment label in the sorted sequence.

Whenever the sweepline hits a left endpoint, we insert the corresponding label into the tree in $O(\log n)$ time. In order to do this, we have to answer questions of the form "Does the new label X go above the old label Y?" To answer this question, we test whether the new left endpoint of X is above segment Y, or equivalently, if the triple of endpoints left(Y), right(Y), left(X) is in counterclockwise order.

[1] Actually, if more than two segments intersect at the same point, you could have a larger reversal, but this won't have any effect on our algorithm.



An example of the sweepline algorithm.
The boxes show the sequence stored in the binary tree.

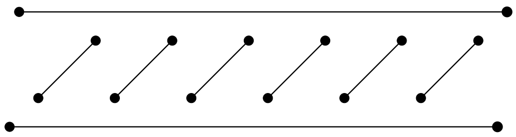
Once the new label is inserted, we perform two pairwise intersection tests, between the new segment and its two neighbors in the label sequence. For example, in the figure above, when the sweepline hits the left endpoint of F, we test whether F intersects either B or E. These tests require $O(1)$ time.

Whenever the sweepline hits a right endpoint, we delete the corresponding label from the tree in $O(\log n)$ time, and then check whether its two neighbors intersect in $O(1)$ time. For example, in the figure, when the sweepline hits the right endpoint of C, we test whether B and D intersect.

If at any time we discover a pair of segments that intersects, we stop the algorithm and report the intersection. For example, in the figure, when the sweepline reaches the right endpoint of B, we discover that F and D intersect, and we halt. Note that we may not discover the intersection until long after the two segments are inserted, and the intersection we discover may not be the first one that the sweepline would hit first. If the algorithm reaches the nth right endpoint without detecting an intersection, we report (correctly) that no two segments intersect.

For each endpoint that the algorithm processes, we spend $O(\log n)$ time updating the binary tree, plus $O(1)$ time performing pairwise intersection tests — at most two at each left endpoint and at most one at each right endpoint. The entire sweep requires $O(n \log n)$ time in the worst case, so the overall running time is $O(n \log n)$.

Note that our sweepline algorithm doesn't try to avoid redundant pairwise tests. In the figure below, the top and bottom segments would be checked $n-1$ times, once at the top left endpoint, and once at the right endpoint of every short segment. But since we've already spent $O(n \log n)$ time just sorting the inputs, $O(n)$ redundant segment intersection tests make no difference in the overall running time.



The same pair of segments might be tested $n-1$ times.