CPS234 Computational Geometry **September 13, 2005**

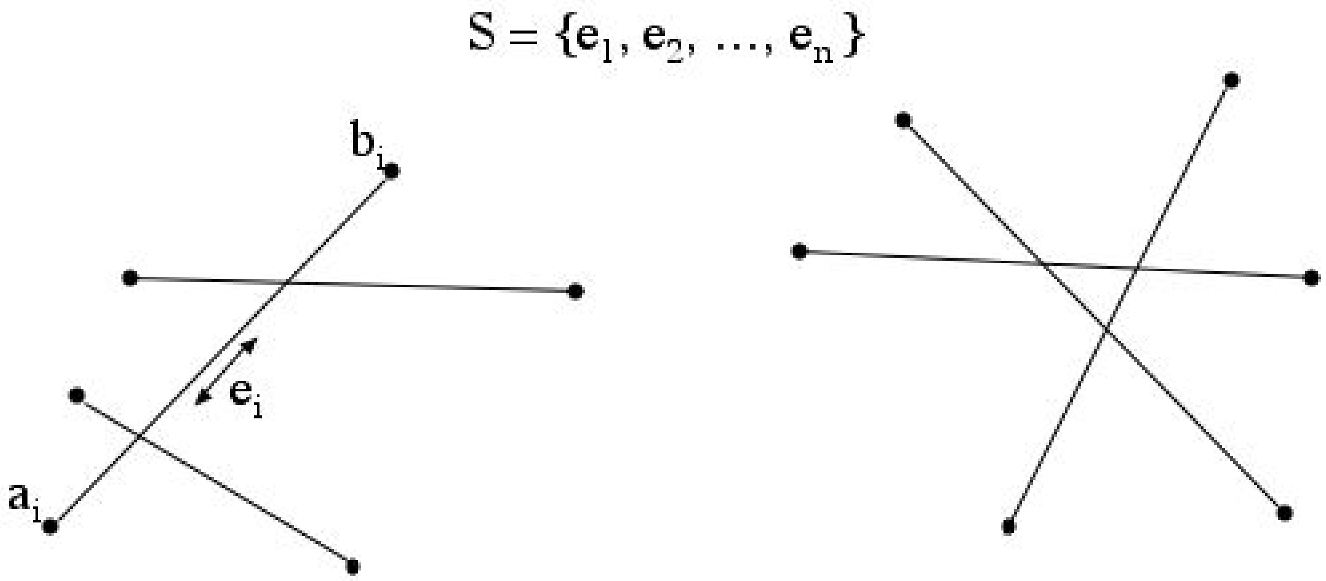
Lecture 5: Line Segment Intersection Detection

Lecturer: Pankaj K. Agarwal *Scribe: Sam Slee*

5.1 Introduction to Segment Intersection

After focusing on convex hulls for the past few lectures, today we move on to detecting intersections between geometric objects. While there are algorithms for detection with any type of object this lecture will just cover the simplest case: detecting an intersection between 2 line segments within a set of line segments.

5.1.1 Types of Intersection Detection Algorithms



The line segment intersection problems that we'll look at today deal only with with line segments defined between points in 2D. That is, for any line segment $e_i = (a_i, b_i)$, the (a_i, b_i) pair give the two points where the line segment starts and ends. For each of those points we have that $a_i \in \Re^2$ and $b_i \in \Re^2$. Since the line segment is defined by these two points, we may also think that $e_i \in \Re^4$. Now, given a set S of these line segments, we can define 3 different types of intersection detection algorithms.

Input: $S = \{e_1, e_2, \ldots, e_n\}$
$e_i = (a_i, b_i)$
$a_i, b_i \in \Re^2$

Output:

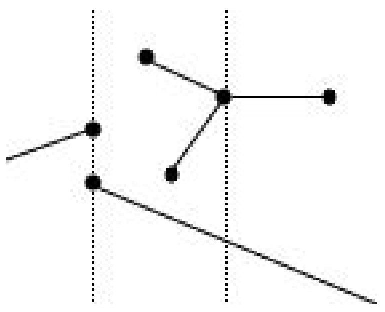
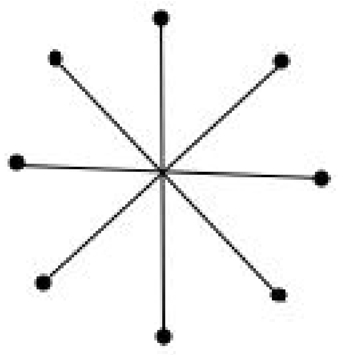
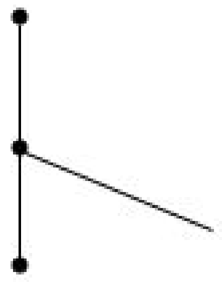
- Detection Algorithm: Does a pair $e_i, e_j \in S$ intersect?
- Reporting Algorithm: Return all intersecting pairs in S (or return all intersection points.)
- Counting Algorithm: Count the number of intersecting pairs.

For now we will focus on the reporting problem of finding all intersecting pairs. The algorithm that we'll look at later will solve this problem and could also be used to solve the counting problem. However, since the counting problem requires a less detailed answer it would seem that it could be solved more quickly.

Assumptions on Input To make this problem a little easier to deal with, we'll make some assumptions on our input set S of line segments. These set of assumptions are used to remove troublesome cases that are referred to as *degenerate cases* and the total set of assumptions is referred to as having our points and segments in *general position*. Be aware, though, that "general position" is a term that is used often with geometric problems and can refer to different sets of assumptions for different problem types. For our segment intersection problem, it refers to the following set of 3 simplifying assumptions.

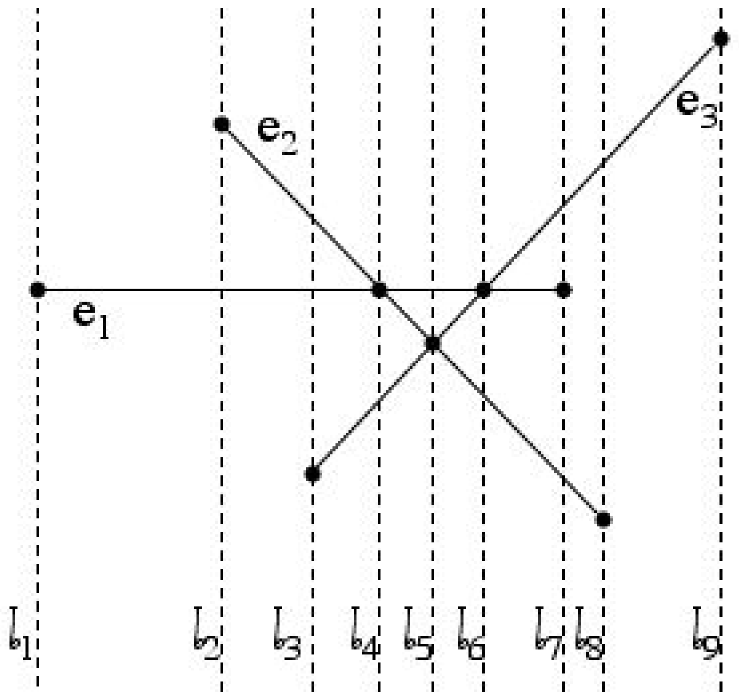
- x-coordinates of all endpoints (of line segments) and all intersection points are distinct.
- Only 2 segments pass through a given intersection point.
- Endpoints do not lie on any other line segment.

The following pictures give examples of cases that are disallowed by those assumptions made above.

5.1.2 A Sweeping Vertical Line

With 3 assumptions made to simplify our problem, we can now use a relatively simple concept to detect intersections between the segments in our input set S. Given our set S, we can now conceptually draw a vertical line through each endpoint and intersection point from that set. Graphically, this results in a picture similar to the following example. The segments below are each labeled e_i for some unspecified ordering of i, and the vertical lines are labeled l_k with the leftmost line being l_0 and with the index label k increasing for lines to the right. Thus, for a given vertical line l_k, l_{k-1} is the next vertical line to its left and l_{k+1} is the next vertical line to its right.



Before moving on, it's worth noting the help that our 3 simplifying assumptions gave here. With those assumptions made, we can conclude that we have at most a single vertical line at any given x-coordinate. The assumption that the endpoints do not lie on any other segment keeps us from having an endpoint and intersection point at the exact same point. The assumption that only 2 segments pass through a given intersection point lets us assume that there's a unique intersection point for each intersection. Finally, the big assumption that the x-coordinates of all endpoints and all intersection points are unique — combined with the 2 other assumptions — assures us that all of our vertical lines in the example above are unique. We don't have to worry about vertical lines being stacked on top of each other at the same x-coordinate.

Remembering this information, we are now ready to make 2 important claims. These claims are about the vertical ordering of the line segments e_i at a given vertical line l_k. This vertical ordering is just made by the y-coordinate values of the segments at the x-coordinate where the vertical line l_k lies.

Claim 1 *For any vertical lines l_k and l_{k-1}, the vertical ordering of the segments is the same between l_{k-1} and l_k.*

Claim 2 *For any vertical lines l_k and l_{k-1}, if $e_i \cap e_j$ lies on vertical line l_k, then e_i and e_j are adjacent in the vertical ordering between l_{k-1} and l_k.*

Justifying the Claims The first claim that the vertical ordering of the segments stays the same between adjacent vertical lines should make sense. Between two adjacent vertical lines l_{k-1} and l_k there are no intersection points — because if there were any we would have another vertical line between l_{k-1} and l_k. Since no intersections occur in this area we can't have any segments swap places in the vertical ordering. Similarly, we know that no left or right endpoint occurs in the space between l_{k-1} and l_k. This means that no segment is added to or removed from our ordering within this space. Thus, the vertical ordering stays exactly the same — the same segments and the same order — in the space between any pair l_{k-1}, l_k of adjacent vertical lines.

Now for the second claim, that segments e_i and e_j are adjacent in the vertical ordering between l_{k-1} and l_k before the segments cross at the x-coordinate of l_k. To see why this is so, remember our claim that only 2 segments pass through any given intersection point. So, only e_i and e_j can meet at their intersection point on the vertical line l_k. All other segments must pass through l_k strictly above or below both e_i and e_j. This means that e_i and e_j are adjacent in the vertical ordering at l_k. Further, since the first claim we made said that the vertical ordering remained unchanged between adjacent vertical lines, we must have that segments e_i and e_j were adjacent in the vertical ordering in the entire space between lines l_{k-1} and l_k.

5.2 Algorithms For Segment Intersection Detection

Using the claims and concepts given by the vertical line techniques just described, we can now start piecing together an algorithm for reporting line segment intersections. To do this, we'll conceptually sweep a single vertical line from left to right, maintaining the vertical ordering as we go. Our claim in the previous section that no changes occur in this vertical ordering between adjacent vertical lines l_{k-1} and l_k is particularly helpful here.

Those vertical lines l_k were just drawn at segment endpoints or intersection points. So, now when we sweep a single vertical line from left to right we really only have to jump it from one endpoint or intersection point to the next one in the left-to-right ordering. With a little bit of work to update the vertical ordering changes at each point, this vertical line sweep can accurately portray the vertical ordering through the entire space $\Re^2$. One possible catch is that when we start running any such algorithm we only know the endpoints. We can't already know the intersection points because that's what we're trying to find. It turns out to be an easy problem to fix as our algorithm will find those intersection points as it sweeps.

Data Structures One last tool we need for our algorithm is a data structure capable of holding and maintaining the data that we need. Specifically we'll use two data structures: (1) a tree structure T for maintaining the current vertical ordering of segments intersected by the line sweep and (2) a queue Q holding the endpoints and intersection points still to be visited by our line sweep in left-to-right order. For the tree structure most any good, dynamic data structure such as a heap or red-black tree will work. For the queue, we'll need a priority queue so that we can insert intersection points into the proper order as we discover them.

Notation For both data structures we'll use the notation INSERT(e, A) for inserting a segment/point e into the data structure A. For the tree structure, a segment is inserted based on its place in the vertical ordering at the current x-coordinate of the line sweep. For the priority queue, a point is inserted into the left-to-right order based on its x-coordinate. The function DELETE(e, T) removes e from T and DELETE-MIN(Q)

removes the leftmost element in the priority queue Q and returns it to the program. SWAP(a, b, T) causes the segments a and b to swap places in the vertical ordering maintained by the tree T. Finally, $a \cap b$ is used to denote the point of an intersection between two segments a and b and $\emptyset$ is used to denote an empty set.

5.2.1 The Line-Sweep Algorithm

Line-Sweep Algorithm

Input: S, a set of n line segments $e_i = (a_i, b_i)$ with $a_i, b_i \in \Re^2$.
Output: A listing of all of the segment pairs $e_i, e_j \in S$ that intersect.

```
$\triangleright\triangleright$ Comment: Initializing $Q$ to hold the endpoints and $T$ to be empty.
$Q \leftarrow$ endpoints of all $e_i \in S$.
$T \leftarrow \emptyset$

while( $Q \neq \emptyset$ )
    $p =$ DELETE-MIN$(Q)$

    $\triangleright\triangleright$ Comment: First case.
    if $p$ is a left endpoint of a segment $e$
        INSERT$(e, T)$
        Let $e_i$ and $e_{i+1}$ be the segments adjacent to $e$ in $T$.
            if $e_i \cap e \neq \emptyset$
                INSERT$(e_i \cap e, Q)$
            if $e_{i+1} \cap e \neq \emptyset$
                INSERT$(e_{i+1} \cap e, Q)$
        $\triangleright\triangleright$ Comment: ** Extra statement to be inserted here is explained in section 5.2.3.

    $\triangleright\triangleright$ Comment: Second case.
    if $p$ is a right endpoint of a segment $e$
        DELETE$(e, T)$
        Let $e_i$ and $e_{i+1}$ be the segments adjacent to $e$ in $T$.
        if $(e_i \cap e_{i+1} \neq \emptyset)$ && $(x(e_i \cap e_{i+1}) > x(p))$
            INSERT$(e_i \cap e_{i+1}, Q)$

    $\triangleright\triangleright$ Comment: Third case.
    if $p$ is a intersection point $p = e_i \cap e_{i+1}$
        $\triangleright\triangleright$ Comment: Reporting an intersecting pair that was found.
        REPORT$(e_i \cap e_{i+1})$
        SWAP$(e_i, e_{i+1}, T)$
        if $(e_{i-1} \cap e_{i+1} \neq \emptyset)$ && $(x(e_{i-1} \cap e_{i+1}) > x(p))$
            INSERT$(e_{i-1} \cap e_{i+1}, Q)$
        if $(e_{i+2} \cap e_i \neq \emptyset)$ && $(x(e_{i+2} \cap e_i) > x(p))$
            INSERT$(e_{i+2} \cap e_i, Q)$
end-while
```

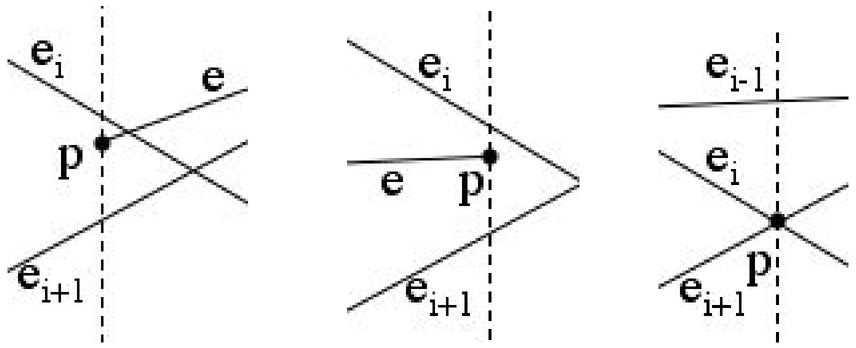


Figure 5.1: The figures given above show the 3 possible cases: (1) p represents a left endpoint, (2) p represents a right endpoint, or (3) p represents an intersection point.

5.2.2 Analysis of the Line-Sweep Algorithm

Correctness The Line-Sweep Algorithm just described was first given by Bentley and Ottmann in 1979. Algorithms of this type — with a sweeping line in $\Re^2$ or a sweeping hyper-plane in higher dimensions — are used in various computational geometry problems. To verify the correctness of this line-sweep algorithm we can note what happens at each of the 3 possible cases for a point p taken from the priority queue Q. If p is a left endpoint — the leftmost case in figure 5.1 — then clearly we must add the segment e that it represents to the vertical ordering. In the same way we must remove e if p is a right endpoint — the middle case in figure 5.1. Similarly, if p is an intersection point — the rightmost case in figure 5.1 — then we know that there is a pair, and only one pair, of intersecting segments at this vertical position. Thus, we have to swap those two segments in the vertical ordering.

Finally, when looking for new intersections in each of 3 possible cases for a point p our job is fairly simple. It is always sufficient to check only the segments that have just been made adjacent to each other in the vertical ordering. This comes from the second claim that we made long ago. When two segments e_i and e_j intersect at a vertical line l_k, they must have been adjacent in the vertical ordering in the space between l_{k-1} and l_k. Thus, we will always run across a vertical line l_{k-1} where two segments are adjacent before the line l_k where those segments would intersect. In this way, if we just always check newly adjacent pairs of segments we are guaranteed to detect a coming intersection point before our left-to-right sweep reaches the y-coordinate of the intersection point.

Time and Space Analysis Now that we have the intuition for why the algorithm is correct, we can look at the time and space required for the algorithm to run. Let n be the number of line segments in the input set S and let k be the number of intersection points between the segments in that set. The **while-loop** that takes up the majority of the program runs once for every endpoint or intersection point, so that equates to $2n + k$ runs of that loop. For the running time of each iteration, note that every operation on our data structures takes $O(\log n)$ time. Thus, each iteration will only take $O(\log n)$ time and the total running time is $O((n + k) \log n)$.

As for the space required, note that the worst possible case for the tree data structure is to store all n line segments at the same time. So, it takes $O(n)$ space. For the priority queue, it's worst case would be to store all endpoints and intersection points at the same time, a total of $O(n + k)$ space. Thus, the total space requirement is $O(n+k)$.

About Earlier Assumptions Before the line-sweep algorithm was described we made 3 simplifying assumptions that helped us design the algorithm. Looking back, we can now see that it's not difficult to overcome those assumptions to allow the algorithm to work in general cases. One of the assumptions that we made was that only 2 segments passed through a given intersection point. This allowed us to perform only a single swap at each intersection point to update the vertical ordering of the segments.

At first, it may seem that removing this assumption would possibly require our algorithm to perform much more work when dealing with each intersection point. Indeed, the algorithm must be modified and more work is now needed for some intersection points. Still, we get that the overall workload remains reasonable. Specifically, consider an intersection point p and degree(p) or the number of segments passing through p. When summing over all possible intersection points we get that:

$$\sum_p \text{degree}(p) = O(n+k)$$

Although not proven here, this result comes from results about planar graphs and the fact that our set of line segments can be used to form a planar graph. In any case, this bound means that we can perform the necessary reordering, even at the busiest intersection points, and still leave the final running time intact. Also, for the case of the x-coordinates of endpoints not being unique, we can modify the algorithm so that right endpoints are handled first, then intersection points and finally left endpoints. This will save some time by first removing segments that no longer need to be maintained and will wait to add new points after any swapping of old points has already been done. As for ties, like between two intersection points with the same x-coordinate, some other pre-defined ordering may be used if necessary.

5.2.3 Complexity Results of Segment Intersection Detection Problems

For the line-sweep algorithm described in this lecture we had $O((n+k)\log n)$-time and $O(n+k)$-space requirements. While not bad, these bounds can be improved. In the 90's Pach and Sharir proved a lower bound of $\Omega(n \lg n)$ for the running time and an upper bound of $O(n \lg^2 n)$. Another lower bound of $\Omega(n \log n + k)$-time has also been shown and achieved for randomized algorithms. We will see such an algorithm in the next lecture. For the space requirement, Brown in 1979 showed how only $O(n)$ space could be needed. This was done by including a simple line in our line-sweep algorithm at the end of the first case for p (when p was a left endpoint for a segment e).

```
** if $x(e_i \cap e_{i+1}) > x(p)$
       DELETE$(e_i \cap e_{i+1}, Q)$
```

At the point where this line is inserted, a new segment e has just been placed between e_i and e_{i+1} in the vertical ordering. Yet, we know that e_i and e_{i+1} must be adjacent in the vertical ordering just before they intersect. So, we know that these two segments will again become adjacent in the vertical ordering, and since our algorithm checks all possible cases where this could happen, we will be able to detect the $e_i \cap e_{i+1}$ intersection point again later in the running of the algorithm. This means we can safely delete that intersection point for now. It also means that we're now able to store only $O(n)$ points in the queue without increasing the time complexity of the line-sweep algorithm beyond $O((n+k)\log n)$.

The bounds given above were for the reporting version of the segment intersection problem. This was the problem that our line-sweep algorithm was designed to solve and the focus of this lecture. However, the other types of problems — detection and counting — are also important and have known complexity bounds. For the *detection* problem, matching upper and lower bounds have been found to give the complexity $\Theta(n \log n)$. A bound on a similar question can be used to give the lower bound for this result.

Input: A set of elements $\{x_1, x_2, \ldots, x_n\}$
Output: Are all of the x_i's unique? (Yes/No)

This problem has a lower bound of $\Omega(n \log n)$ and can also be used to lower bound the detection problem.