In this lecture, we will study the problem of computing all intersections of a set of line segments in the plane. To address this problem, we will design an algorithm using the plane sweep technique, and introduce the concept of output sensitivity to motivate its runtime compared to a more naive algorithm.

1 Line Segment Intersection

A closed line segment $s = pq$ in the plane is defined by its two endpoints, $p$ and $q$. Formally, for $p, q \in \mathbb{R}^2$, we define $pq := \{(1 - \alpha)p + \alpha q : \alpha \in [0, 1]\}$. The problem we want to address in this lecture is of finding all intersections among a collection of line segments $S = \{s_1, \ldots, s_n\}$ in the plane.

Checking whether a given pair of line segments intersect is easy; it roughly amounts to solving a system of two linear equations. Therefore, there is a simple, brute-force algorithm for the line segment intersection problem: simply check whether every pair of segments has an intersection. This algorithm takes $O(n^2)$-time. Moreover, some sets $S$ of $n$ line segments may have $\Omega(n^2)$ intersections, so any algorithm reporting all intersections necessarily takes $\Omega(n^2)$-time in this worst case. (To see this, consider a set $S$ of $n/2$ horizontal line segments and $n/2$ vertical line segments forming a grid pattern.) So, in some sense this naive $O(n^2)$-time algorithm is optimal.

Although our naive algorithm is optimal in some sense, we will see that there is a much faster algorithm for inputs with few intersecting line segments. In fact, our algorithm’s runtime will explicitly depend on the number of intersections among the input segments. We generalize this way of measuring the runtime of an algorithm in terms of output-sensitivity. An output-sensitive algorithm is one whose runtime depends on the size of its output (here the output size is the number of intersection points). In general, the goal of output-sensitive algorithms is to adapt to the complexity of the input. They should run quickly on simple inputs, and necessarily run slower on more complicated ones.

Note that our naive algorithm is certainly not output-sensitive. It requires $\Omega(n^2)$-time regardless of whether the input segments have no intersections or $\Omega(n^2)$ intersections.

2 A Plane Sweep Algorithm for Line Segment Intersection

Our goal is now to give a more refined, output-sensitive algorithm using the plane sweep technique. To motivate this technique, consider the following modification of our naive algorithm. Project the line segments onto the $y$-axis (i.e., look at the maximal and minimal $y$-coordinate of the points in each segment). Two segments may not intersect even if their projections do, but two segments definitely do not intersect if their projections do not intersect. So, we can refine our original algorithm by only checking for intersections among pairs of segments whose projections onto the $y$-axis overlap.

To find such pairs, we imagine sweeping a horizontal line (called the sweep line) through the plane from top to bottom, starting above the topmost segment. We will keep track of which segments currently intersect the sweep line, and in what order, from left to right. We call this the status of the sweep line.

Note that there are only a few situations in which the status changes. When we encounter the upper endpoint of a segment, we need to add it to our list of intersecting segments, and when we encounter the lower endpoint of a segment, we need to remove it from our list of intersecting segments. Furthermore, the order in which two segments intersect a horizontal line above an intersection is flipped from the order in which they intersect a horizontal line below an intersection.

---

1 This lecture is based on [BCKKO08, Chapter 2.1], which describes an algorithm of Bentley and Ottman [BO79].
Lecture 2: Line Segment Intersection

Figure 1: Diagrams showing how HANDLEEVENTPOINT handles three types of event points (shown in red). When handling an upper endpoint, we check whether the incident segment \( s_j \) intersects its neighbors \( s_i \) and \( s_k \) (Case I, left), when handling a lower endpoint we check whether the neighbors \( s_i \) and \( s_k \) of the incident segment \( s_j \) intersect each other (Case II, center), and when handling an intersection point of \( s_i \) and \( s_j \), we check for intersections between both of the newly adjacent pairs \( s_i, s_k \) and \( s_j, s_\ell \) (Case III, right). Exactly one new pair of intersecting segments is found in each of the three example checks shown here.

they intersect a horizontal line below the intersection. So, as we’re sweeping the plane we make updates to the status when encountering the endpoints of segments, and when encountering segment intersections. We call these points where the status changes event points (see Figure 1).

To simplify the description of our algorithm, we will assume the following three conditions about our input for the remainder of Section 2:

1. There are no horizontal segments.
2. Pairs of segments intersect in at most one point which is in their interiors (i.e., there are no overlapping segments and no intersections at endpoints).
3. No more than two segments intersect at a point.

Call a set of line segments \( S \) satisfying these conditions good. These conditions are not necessary, and modifying the algorithm to handle them is relatively straightforward. We discuss some of these modifications in Section 3.

Before describing our algorithm, we make one more observation: for two segments to intersect, they must be adjacent along the sweep line. We can now describe the high-level idea behind our algorithm. We sweep from top to bottom, and each time we process an event point we check whether any newly adjacent pair of segments has an intersection. If they have an intersection, we add it as an event. Note that at each event point we only need to do at most two intersection checks.

## 2.1 The Algorithm

The key to implementing the algorithm that we have sketched is to use two data structures: a priority queue \( Q \) and a balanced binary search tree \( T \).

The priority queue \( Q \) stores event points according to their priority. Given two event points \( p := (p_x, p_y) \) and \( q := (q_x, q_y) \), we define the ordering \( p \prec q \) to hold if \( p_y > q_y \) or if \( p_y = q_y \) and \( p_x < q_x \). We store event points in \( Q \) according to the ordering \( \prec \), where \( p \prec q \) means that we will consider \( p \) before \( q \). In other words, our plane sweep algorithm will consider event points from top-to-bottom, and from left-to-right along each horizontal line. (This is the same order in which one reads words in a page of English text.) We want the ability to check efficiently whether an event point already exists in \( Q \), so we implement \( Q \) as a balanced binary search tree rather than as a heap.

\[ \text{[2]} \text{In other words, lesser points according to } \prec \text{ have higher priority. This follows the notation in } [\text{GBCKO08}]. \]
The balanced binary search tree $T$ stores the line segments intersecting the sweep line in its leaves in order from left to right. We additionally store information at each internal node to make $T$ searchable. Namely, at each internal node we store the rightmost segment of its left subtree.

Because both $Q$ and $T$ are balanced binary search trees, we can perform insertions, deletions, and lookups in each tree in $O(\log |Q|)$ and $O(\log |T|)$ time, respectively. We now present our algorithm, whose pseudocode is very simple.

**Algorithm 1: FindIntersections($S$)**

- **Input:** A good set $S$ of $n$ line segments.
- **Output:** The set of intersection points of segments in $S$, and a list of which segments intersect at each intersection point.

Initialize a priority queue $Q$ and a balanced binary search tree $T$. Enqueue the endpoints of all segments in $S$ into $Q$.

while $|Q| > 0$ do
  $p \leftarrow Q$.dequeue().
  HandleEventPoint($p, T, Q$).
end

The function $Q$.dequeue() removes and returns the event with highest priority in $Q$. We next specify HandleEventPoint($p, T, Q$) in terms of how it handles three types of event points $p$. Refer also to Figure 1.

1. Case I, $p$ is an upper endpoint of a segment $s_j$. Add $s_j$ to $T$ between its left neighbor $s_i$ and its right neighbor $s_k$. Then check whether $s_j$ intersects either $s_i$ or $s_k$. If so, add the corresponding intersection point(s) to $Q$ if they are not already present.

2. Case II, $p$ is a lower endpoint of a segment $s_j$. Delete $s_j$ from $T$. Then check whether its left neighbor $s_i$ and right neighbor $s_k$ intersect each other. If so, add the corresponding intersection point to $Q$ if it is not already present.

3. Case III, $p$ is an intersection point of two segments $s_j$ and $s_k$. Flip the order of $s_j$ and $s_k$ in $T$. Then check whether $s_k$ intersects its new left neighbor $s_i$, and whether $s_j$ intersects its new right neighbor $s_k$. If so, add the corresponding intersection point(s) to $Q$ if they are not already present.

In each of the above three cases, one or both neighbors may not exist. In that case we omit the corresponding intersection checks.

Finally, we analyze the correctness and runtime of Algorithm 1.

**Theorem 2.1.** Algorithm 1 computes the intersection points of a good set $S$ of $n$ line segments in $O((n + I) \log n)$-time, where $I$ is the number of intersection points among segments in $S$.

**Proof.**

**Correctness:** To prove the correctness of Algorithm 1 we need to show that it reports all intersection points among segments in $S$. (It is clear that no false intersections are reported.) We show this by arguing that if two segments $s_i, s_j \in S$ intersect in point $p$, that they become adjacent beforehand and are tested for intersection.

Let $\ell$ be a horizontal line above $p$ and below all event points above $p$. There are finitely many event points so such a line $\ell$ exists. Furthermore, when the sweep line reaches $\ell$, $s_i$ and $s_j$ are adjacent. On the other hand, $s_i$ and $s_j$ are not adjacent when the algorithm starts since the sweep line begins above all segments. Therefore, there is an event point at which $s_i$ and $s_j$ become adjacent and are tested for intersection.
Runtime: Enqueuing the endpoints of segments into $Q$ takes $O(n \log n)$-time, and initializing $T$ takes constant time. We analyze the time used by \textsc{HandleEventPoint}(p) for an event point $p$. Handling $p$ requires reporting the two segments that intersect at $p$, and deleting $p$ from $Q$. It also requires performing at most two intersection tests, and therefore at most two insertions of new event points into $Q$. Handling $p$ also requires either inserting or deleting a segment, or swapping the order of two segments in $T$. Overall, handling $p$ requires a constant number of updates to each of $Q$ and $T$, and therefore takes $O(\log |Q| + \log |T|) = O(\log(n+I) + \log n) = O(\log n)$-time. Because of our assumptions about the input, there are exactly $2n+I$ event points, so handling all event points takes $O((n+I) \log n)$-time. This dominates the runtime of the algorithm, which therefore takes $O((n+I) \log n)$-time overall, as claimed.

3 Handling Special Cases

We now describe some aspects of how to adapt Algorithm 1 to handle inputs that are not good (i.e., to handle inputs that may contain horizontal segments, more than two segments intersecting at a point, and segments that intersect either at their endpoints or that overlap).

Algorithmically we need to update \textsc{HandleEventPoint} to handle these additional cases, and then we need to update the proof of correctness in Theorem 2.1 accordingly. Understanding the details of how to handle these cases is worthwhile, but doesn’t introduce any substantial new ideas. We therefore omit further discussion of them here; see \textsc{HandleEventPoint} and Lemma 2.2 in [dBCvKO08].

The most interesting aspect of the general case is analyzing the updated algorithm’s runtime. When processing each intersection point $p$, we still only need to perform a constant number of operations on each of $Q$ and $T$ per segment that contains $p$. However, in the general case many segments may contain $p$, so it is not clear that the runtime guarantee in Theorem 2.1 still holds. In the next lemma we show that it does. The idea is to show that on average an event point can only involve a few segments.

Lemma 3.1. Consider a set $S$ of $n$ line segments with $I$ intersection points. Let $m(p)$ denote the number of segments which contain an event point $p$, and let $m := \sum_p m(p)$ be the sum of $m(p)$ over all event points $p$. Then $m = O(n + I)$.

Proof: The key idea is to observe that the line segments in $S$ and their intersection points form a planar graph, and then to apply Euler’s formula.

A planar graph is a graph which can be drawn in the plane in such a way that no two edges cross (except where they meet at vertices). More formally, we define the planar graph $G(S)$ to have a vertex for each endpoint or intersection point (i.e. for each event point), and an edge for each segment piece running between event points.

Let $n_v, n_e, n_f$ denote the number of vertices, edges, and faces of a planar graph, respectively. (Note that the unbounded face counts towards $n_f$.) Euler’s formula says that

$$n_v - n_e + n_f \geq 2,$$

where equality holds if and only if the graph is connected. We next give bounds on $n_v, n_e, n_f$.

By definition,

$$n_v \leq 2n + I. \quad (2)$$

Every event point $p$ is a vertex of $G(S)$, and $m(p)$ is upper bounded by the degree of $p$. Therefore, $m$ is upper bounded by the sum of the degrees of event points $p$. Each edge contributes one to the degree of exactly two nodes, so the sum of the degrees of event points is equal to $2n_e$. Therefore, we get that

$$m \leq 2n_e. \quad (3)$$

Finally, we note that (1) every face is bounded by at least three edges (assuming $n \geq 3$), and (2) every edge is incident to at most two faces. Therefore,

$$n_f \leq 2n_e/3. \quad (4)$$
Plugging our bounds from Equations (2) and (4) into Equation (1), we get that

\[ 2 \leq (2n + I) - n_e + (2n_e/3), \]

so \( n_e \leq 6n + 3I - 6 \). Combining this with Equation (3), we get that \( m \leq 12n + 3I - 12 \), which implies that \( m = O(n + I) \), as desired.

Notice that Lemma 3.1 only bounds \( m \), and says nothing a priori about computation or runtime. However, as we discussed earlier, the runtime of `HANDLE_EVENT_POINT(p)` is bounded by \( O(m(p) \cdot \log n) \). Handling event points again dominates the overall runtime of the general algorithm, so we can bound its overall runtime by \( \sum_p O(m(p) \cdot \log n) = O(m \log n) = O((n + I) \log n) \) using Lemma 3.1.

We are also interested in the space complexity of our algorithms, i.e., how much memory they use. Algorithm 1 and its extension to the general case require storage proportional to the size of \( Q \) and \( T \). We always have that \( |T| = O(n) \) and \( |Q| = O(n + I) \), meaning that the overall space complexity of our algorithm is \( O(n + I) \). However, by modifying \( Q \) only to store intersection points of segments that are currently adjacent we can improve our bound on the size of \( Q \) to \( |Q| = O(n) \). The only algorithmic modification necessary to achieve this is to delete from \( Q \) intersection points of segments which have become non-adjacent. Combining our bounds on the time and space complexity of the generalized version of Algorithm 1, we get the following theorem.

**Theorem 3.2** ([dBCvKO08, Theorem 2.4]). Let \( S \) be a set of \( n \) line segments in the plane which intersect in \( I \) points. There is an algorithm which computes these intersection points and the line segments involved in them, and which runs in \( O((n + I) \log n) \)-time and \( O(n) \)-space.

**References**
