1. Computing intersections among line segments

Line segment intersection

- Compute and report all intersections among a set of n line segments
  - Intersection means “have at least one point in common”
- A trivial solution based on brute-force:
  - For each pair of line segments: Compute the point of intersection between the lines that contain the line segments; if this point lies on both line segments, we have found an intersection
  - There are $O(n^2)$ pairs so this takes $O(n^2)$ time
- So, do we really need to spend $O(n^2)$ time?
  - Yes, there are $\Omega(n^2)$ intersections in the worst case
  - But there could also be just a few...
- Interesting question: Is there a way to relate the running time to the number of intersections actually reported?

Output-sensitive algorithms

- An algorithm for which the time (or space) complexity depends not only on the size of the input ($n$ in our case) but also the size of the output
  - As with Jarvis march for computing convex hulls
- Let $k$ be the number of intersections found
  - “Number of pairs of intersecting line segments.”
- We will now learn about an output-sensitive algorithm that computes all intersections in $O(n \log n + k \log n)$ time
- (There exists faster algorithms that run in $O(n \log n + k)$ time. These are optimal, but more complicated, and will not be covered.)
Lower bound: $\Omega(n \log n + k)$

- The term $k$ is obvious; we have to output the result
- The term $n \log n$ can be shown by a reduction from:
- ELEMENT-UNIQUENESS:
  - Given a set of $n$ numbers, $\{x_1, x_2, ..., x_n\}$, are two of them equal?
  - Requires $\Omega(n \log n)$ comparisons [Ben-Or 1983]
  - (Easy to solve by sorting)
- Recall that in our model all common binary operators like $<, >, =, \text{AND}, \text{OR}, \text{NOT}, +, -, \ast, /$ etc can be performed in unit (constant) time
  - So, ELEMENT-UNIQUENESS requires $\Omega(n \log n)$ time

Observation 1 about our intersections

- Only line segments that overlap if projected onto the $y$-axis might intersect!

Algorithmic technique: Plane sweep

- Imagine a horizontal line sweeping downwards over the plane, starting above all line segments
  - Also called line sweep (or just sweep)
- As this sweep line is moved, we keep track of the set of line segments that it intersects - the status
  - The status contains the intersected line segments ordered in the $x$-direction by how they intersect the sweep line
- In general, the status is invariant. It only changes when the sweep line vertically reaches an event point
- An event point is either
  - an end point of a line segment, or
  - a point of intersection between two line segments

Lower bound reduction

- Let $A$ be any algorithm that computes intersections among line segments
- We can then solve ELEMENT-UNIQUENESS by constructing short line segments for the numbers in the set, run $A$, and check the result
  - Map $x_i$ to, for instance, the line segment $[\langle x_i, 0 \rangle, \langle x_i, 1 \rangle]$
  - Then, $A$ will report intersections if and only if there are $i \neq j, x_i = x_j$
- Since everything else than $A$ takes $O(n)$ time, it must be $A$ that takes $\Omega(n \log n)$ time

\[
\{x_1, x_2, ..., x_n\} \Rightarrow \ x_3 \ x_5 \ x_1 \ x_4 \ ... \ x_n \ \ ... \ x_2
\]
Example

Start.

Example

Example

Example
Example

Observation 2 about our intersections

- If two line segments intersect, they must at some time during the sweep lie next to each other in the status!

Done.
Algorithm

- Use an event queue (a priority queue) to store event points ordered by their y-coordinate
- Insert all 2n end points of line segments in the beginning
- Start with an empty status, and go ("sweep") from one event point to the next while keeping invariant that:
  "Above the sweep line all points of intersection have been computed."
- Important:
  - The status does not contain the coordinates where the sweep line intersects. It holds the order in which line segments intersect the sweep line

Algorithm (cont.)

- This happens at an event point when...
  - An upper end point is reached: Insert the new line segment (into the status) and test it for intersections with its neighbors below the sweep line; insert the intersection points as event points, if any
  - A point of intersection is reached: Swap the line segments that intersect and test them for intersections below the sweep line with their new neighbors. If they intersect, insert the points as new event points
  - A lower end point is reached: Delete the line segment and test whether the two new neighbors intersect below. If they intersect, insert the point as a new event point

Data structures used

- Event queue
  - A balanced binary search tree
    - AVL-trees, red-black trees, etc that support insertion, deletion, neighbor (successor and predecessor), and lookup in O(log m) time when the tree contains m items
    - Define an order between the event points (highest first, leftmost to break ties)
- Status
  - A balanced binary search tree
    - (One can store anything - not just numbers - in such a tree as long as the items are comparable)
  - NB! Comparisons between line segments is done by computing points of intersection with the sweep line when needed only

A degenerate case

Special delicate care must be taken if line segments share end points
**Analysis**

- We inserted $2n$ endpoints into the event queue:
  - $O(n \log n)$ time.
- Each intersection was inserted as an event point:
  - $O(k \log n)$ time.
- Total:
  - $O(n \log n + k \log n) = O((n+k) \log n)$ time.

**Reflection**

- $O((n+k) \log n)$ time...? Better analysis possible - see Lemma 2.3!
- This is worse than the brute-force solution ($O(n^2)$) if $k$ is more than $O(n^2/\log n)$ but better when $k$ is less
  - It is, however, output-sensitive
- Compare with the incremental algorithm that computes convex hulls from Lecture 1
  - Our algorithm is also sort-of incremental...
    - ... makes monotone progress and keeps an invariant
    - but still different
    - ... considers not only input points (end points of line segments) but also intersections that are found during the course of the algorithm
- To determine if there are intersections, we just need to stop the line sweep as soon as the first intersection is found, if it exists
  - So, this (simpler) problem can be solved in $O(n \log n)$ time

**Doubly-Connected Edge Lists (DCEL)**

- A data structure suitable to keep track of subdivisions of the plane into different kinds of regions.

**Some basic definitions (Cont.)**

- A *polygonal chain* is a finite concatenation of line segments where consecutive segments share one end point (“One ends where the next starts”)
- A *simple polygon* is a “closed polygonal chain that doesn’t self-intersect”
  - Simple polygons have vertices and edges that forms their boundaries
  - A convex hull is a simple polygon
**DCEL:s**

- Geometrical data:
  - Coordinates
- Topological data:
  - Pointers to neighboring items

**The overlay of two subdivisions**

**Euler’s formula**

- For an embedding of a planar graph with $v$ vertices, $e$ edges, and $f$ faces (regions bounded by edges, including the outer, infinitely-large, region) without crossing edges,

$$v - e + f = 2$$

(v=7, e=8, f=3)
Applying plane sweep

- Maintain a doubly-connected edge list during the sweep over the subdivisions.

An application: Boolean operations

Application: Computing Closest Pairs using Plane Sweep

- Problem:
  - Given a set of points, find a pair of points with the smallest distance between them
- Requires $\Omega(n \log n)$ time
  - Simple reduction
- Solved in $O(n \log n)$ time by many algorithms, for instance plane sweep

Lower Bound on Closest Pair Computations

- Reduction to Element Uniqueness:
  - Given a set of numbers \(\{x_i\}\) (and faced with having to decide if all are unique), we
    - 1) construct points \(p_i = (x_i, 0)\).
    - 2) run any Closest Point algorithm, and
    - 3) check if the distance between the closest points returned is 0 or not
- If not 0, all are unique and otherwise there are duplicates
- We have solved Element Uniqueness that requires $\Omega(n \log n)$ time, and apart from the Closest Point algorithm, everything we do takes $O(n)$ time
- So, it must be the Closest Point algorithm that takes $O(n \log n)$ time.
Max 6