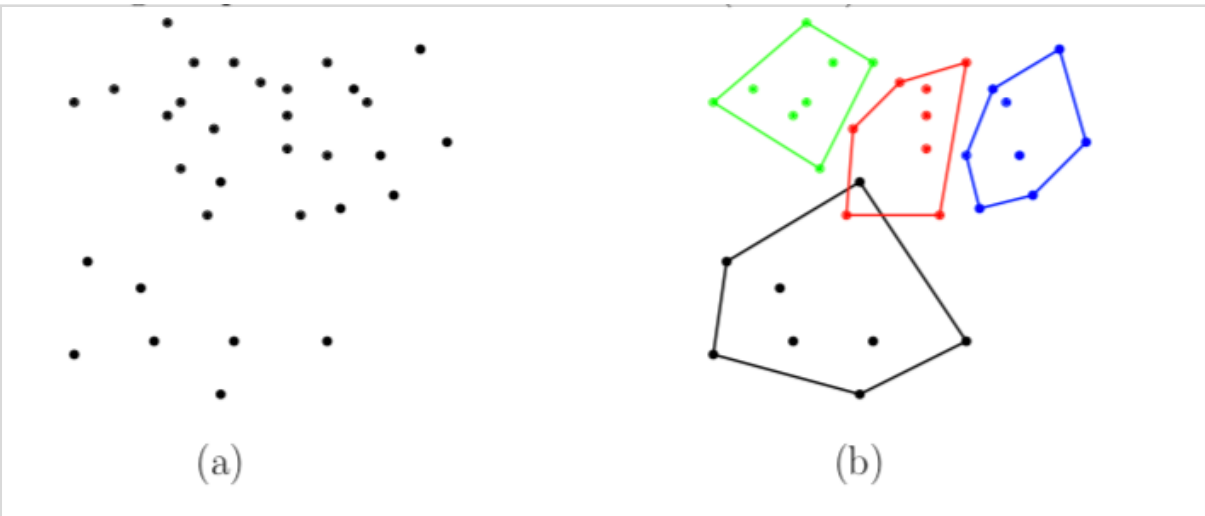
CS 6301.002.20S Lecture 3—January 21, 2020

Main topics are #convex_hulls, #planesweep_algorithms, and #line_segment_intersection.

Chan's Algorithm

- Chan ['96] described an algorithm that's better than both Graham's scan and Jarvis's March. It's actually a combination of both algorithms, but somehow runs in only O(n log h) time.
- Let's build up this algorithm piece-by-piece to see how it works.
- First off, Graham's scan suggests that sorting points is useful. Unfortunately, sorting k points takes Theta(k log k) time.
- However, if we break the point set into n / k subsets of size k, we can sort them all *individually* in O((n / k) k log k) = O(n log k) time. For any constant c and k ≤ h^c, this running time is O(n log h).
- In fact, we even have time to run Graham's scan on these subsets.
- So here's what we'll do: Suppose we know some k where h ≤ k ≤ h^c. We'll discuss how to do this guess later.
- We'll break up the input set P and run Graham's scan on each subset. This will create a collection of n / k mini-hulls that may overlap.



- These mini-hulls are useful for a couple reasons.
- First, any points from a subset that aren't vertices of its mini-hull aren't vertices of the convex hull of P either. We can safely ignore them.
- Second, the mini-hulls have a lot of useful structure since we know the counterclockwise ordering of their vertices.
- Given a vertex p of the convex hull of P, you can figure out the right-most vertex relative to p of any one mini-hull in only O(log k) time using a binary search. I might ask for the details as a homework problem.

- So now we have a faster way to run Jarvis's March! For each vertex p along the hull, you run the binary search on the n / k mini-hulls in O(log k) time each. You'll find the next vertex in O((n / k) log k) time total.
- If $h \leq k \leq h^c$, then you do $h \leq k$ iterations in O((n / k) k log k) = O(n log k) = O(n log h) time total.
- Here's code summarizing what we did. To keep things fast, we'll only compute convex hulls with at most k vertices.
- RestrictedHull(P = {p_1, …, p_n}, k):
  - s ← ceil{n / k}
  - Partition P into disjoint subsets P_1, P_2, …, P_s each of size at most k
  - for j ← 1 to s
    - compute conv(P_j) using Graham's scan
  - ell ← leftmost point of P
  - p ← ell
  - add p to CH
  - i ← 2
  - repeat
    - for j ← 1 to s
      - r_j ← clockwise most point relative to p on conv(P_j)
    - r ← clockwise most point of {r_1, …, r_s} relative to p
    - if r = ell, break
    - else, add r to CH
    - if i > k, return failure (k is too small)
- This whole thing takes O((n / k) k log k) = O(n log k) time total, but to actually output a convex hull in O(n log h) time, we need to guess k so that $h \leq k \leq h^c$.
- To do so, we'll try progressively bigger values of k.
- We can afford to overshoot h by a whole constant power, so we'll try repeatedly squaring k until the algorithm successfully returns a convex hull.
- FastHull(P):
  - k ← 2. CH ← fail
  - while CH = fail
    - k ← min{k^2, n}
    - CH ← RestrictedHull(P, k)
  - return CH
- We're running RestrictedHull several times. Is the algorithm fast enough?
- Consider the ith guess, k = 2^2^i. Running RestrictedHull(P, 2^2^i) takes O(n log (2^2^i)) = O(n 2^i) time.
- The algorithm succeeds as soon as i = ceil{lg lg h} where lg = log_2 h.
- So the total running time (ignoring the ceiling) is proportional to sum_{i=1}^lg lg h n 2^i =

n sum_{i = 1}^lg lg h 2^i.

- If a geometric series has a constant ratio, then its sum is proportional to its largest term. The total running time is O(n 2^lg lg h) = O(n log h). In other words, the time taken to do that last guess is greater than the time taken to do all previous guesses.
- Can we do better? Kirkpatrick and Seidel ('86) say no.

Line Segment Intersection

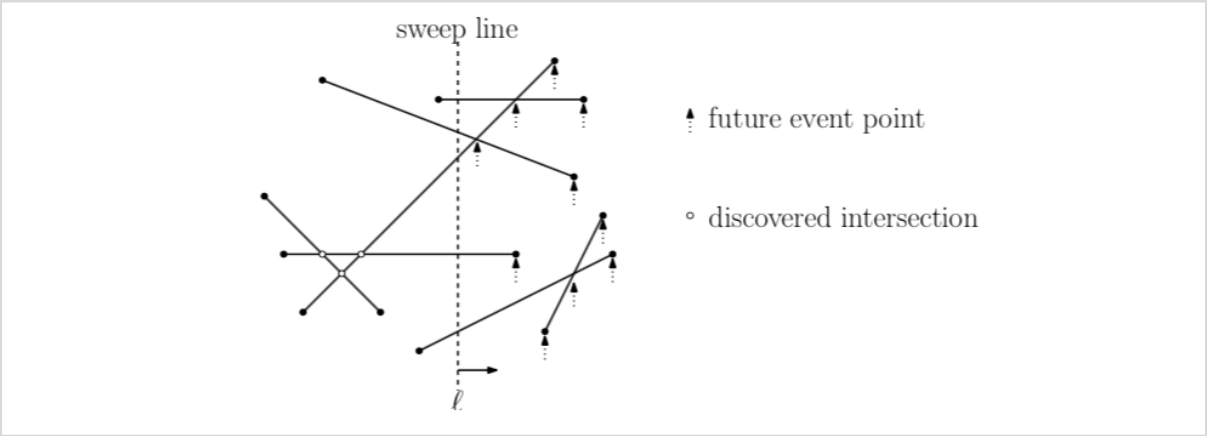
- Time to finally move on from convex hull!
- Suppose we're given a set S of n line segments in the plane. We want to report (output) every one of their intersections.
- Intersection of geometry objects comes up a lot in computational geometry; for example, if you're working with robotics, you need to know whether two objects may intersect.
- This particular problem of line segment intersection is directly applicable to computing map overlays. Say I have a road network represented as some line segments and the boundaries of counties or states represented as another set of line segments. The intersection points tell me where maintenance responsibilities change along road segments. We may also just want to know where you're entering and exiting boundaries on a map.
- Now, you might observe that a collection n line segments might have no intersections, or it might have all (n choose 2) = Theta(n^2) intersections.
- So if all we care about is worst-case performance, then we may as well just test every pair of line segments and output the intersections we find in optimal O(n^2) time.
- But consider that application I just described. Not every road in the state crosses every county boundary. We shouldn't expect anywhere near Omega(n^2) intersections in practice, so we should try to find algorithms that are much faster when there are few intersections.
- We want to find a good output sensitive algorithm like we did for convex hulls.

- Like before, we'll assume some "general position" assumptions:
  - The endpoints of the line segments and the intersection points have distinct x-coordinates.
  - If two line segments intersect, they do so at a single point.
  - No three line segments intersect at a common point.
- The textbook goes into more detail than I will and avoids most of these assumptions.

Plane Sweep Algorithm

- Let n = |S| and I = the number of intersections.

- We'll discuss and algorithm of Bentley and Ottmann ['79] that runs in O((n + I) log n) time.
- The main method behind this algorithm is called a plane sweep.
- What we'll do is sweep a vertical line across the plane from left to right (I'm doing vertical, because that's my and Mount's preference. The book uses a horizontal line going down).
- As the line moves from left to right, we'll be intersecting a subset of the line segments. This subset and possibly some additional information about the line segments is called the *sweep-line status*.
- We'll formally define the sweep-line status shortly. But for now, I'm going to claim any reasonable status changes only when the sweep line hits line segment endpoints or intersections.
- We'll call these points *event points*.



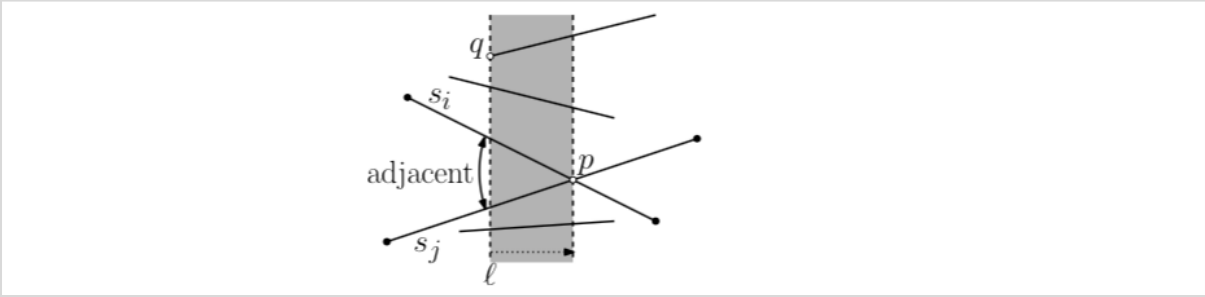
- From the perspective of sweeping the plane, the event points are the only interesting moments. For this reason, the algorithm does *nothing* between event points. You should think of the sweep line moving continuously from left to right to help you understand the algorithm, but between event points, all algorithm data structures remain unchanged.
- But, we do need to know what's going on at event points, and we want to output all the intersections, so the algorithm is going to store a few things:
  1. the partial solution of intersection points already found to the *left* of the sweep line
  2. the current status of the sweep line itself
  3. a *subset* of future event points that need processed, including the next event point the sweep line will cross—more on this later

Detecting Events

- OK, so imagine we're at an event point and we want to know which ones are coming up.
- How do we know when we'll hit the next event point?
- Well, before the plane sweep even begins, we can preprocess all the line segment endpoints. In fact, we can sort all the endpoints in left to right order in O(n log n) time so we know the order in which they will be swept.
- But we can't find the line segment intersections ahead of time. That would defeat the

purpose of running the algorithm in the first place!

- Instead, we'll try to figure out a subset of the event points that is small enough to be manageable but also contains every event point we need before we hit it.
- So say we have several line segments on the sweep line, but there are two that are very close together and have different slopes.
- Lemma: Consider two segments s_i, s_j in S that intersect at some point p = (p_x, p_y). Then, s_i and s_j are adjacent along the sweep line immediately after the previous event.
- Proof:
  - By general position, no three line segments intersect at a common point.
  - Infinitesimally left of p, s_i and s_j are closer than any other pair of segments on the sweep line and therefore adjacent.
  - Let q =(q_x, q_y) be the previous event point. Between q_x and p_x, there are no intersections or new line segments introduced to the sweep line.
  - Therefore, s_i and s_j are adjacent immediately after q_x.



- So, the only line segment intersection point we *must* have computed ahead of time is one from a pair of adjacent segments along the sweep line.
- Not that we necessarily know *which* pair is important ahead of time.
- For this reason, we'll define the sweep line status as the subset of line segments intersecting the sweep line *ordered* from top to bottom along the sweep line. Adjacent pairs are consecutive in this ordering.

- So here's the high level algorithm:
- We'll store the sweep line status in some data structure that lets us remember the order of segments crossing the sweep line and modify it quickly.
- We'll store all segment endpoints and all intersections between adjacent line segments in another data structure called the *event queue* that lets us quickly look up the next event point.
- At each event, we'll update the sweep line status, adjust the collection of possibly relevant event points, and repeat.
- What remains is to give more details on these algorithms and then to describe what updates to do to the sweep line status at each event.

Data Structures

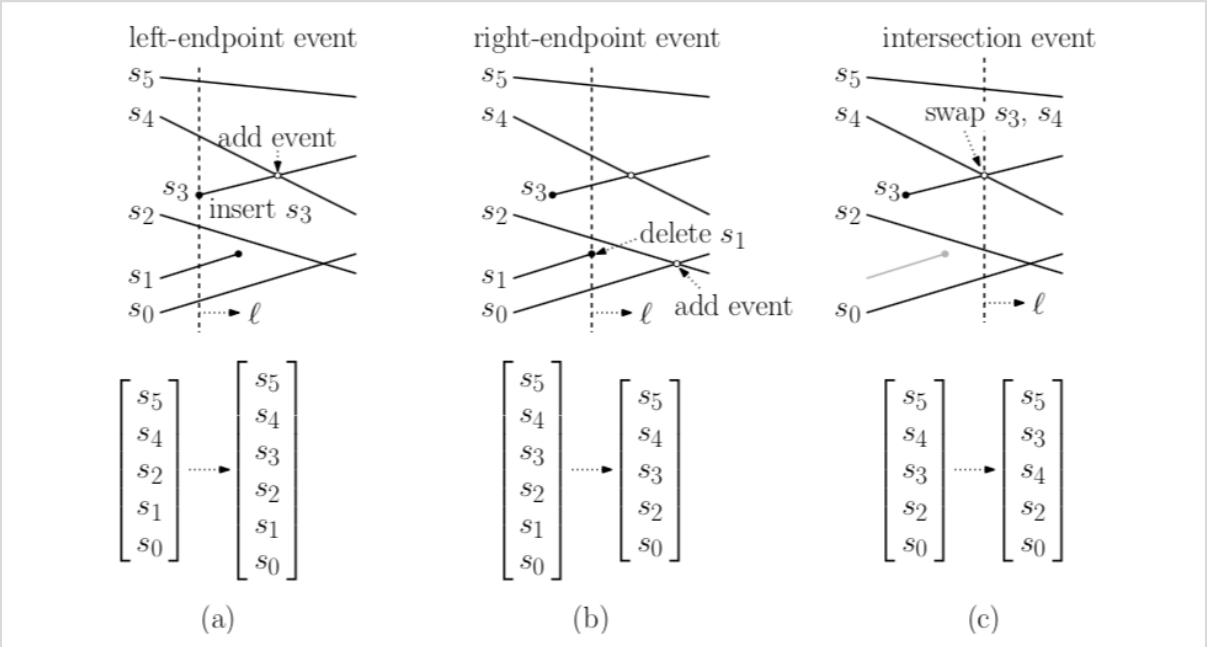
- For sweep line status, we need to know the line segments intersecting the sweep line in top to bottom order, and we need to do quick updates and useful queries
- For this reason, we'll store the status as an *ordered dictionary*.
- An ordered dictionary stores a bunch of ordered elements. There's lots of implementations including balanced binary search trees and skip lists.
- We need one that does the following operations:
  - r ← insert(s): Insert object s and return a reference r to its location in the ordered dictionary.
  - delete(r): Delete entry r.
  - r' ← predecessor(r): Return a reference r' to object immediately before r (or null if r is the first object).
  - r' ← successor(r): Return a reference r' to object immediately after r (or null if r is last object).
  - r' ← swap(r): Swap r and its immediate successor returning a reference r' to its new location.
- With something like a balanced binary search tree, you can store m items in O(m) space and do these operations in O(log m) time each.
- But like when we were doing Jarvis's march, we do need to specify how to do comparisons before we can use an off-the-shelf ordered dictionary.
- Suppose we're sweeping the vertical line through (x_0, 0). To decide which of two line segments i and j is higher, we can express them with the normal line equations y= a_i x + b_i and y = a_j x + b_j. Then line i is higher than j iff a_i x_0 + b_i > a_j x_0 + b_j.

- We'll use a *priority queue* to store the event queue. We need one that supports the following operations:
  - r ← insert(e, x): Insert event e with priority x and return a reference r to its location in the priority queue.
  - delete(r): Delete the entry associated with reference r.
  - (e, x) ← extract-min(): Extract and return the event from the queue with smallest priority x.
- In our setting, the priority will just be the x-coordinate of the event's event point.

Processing Events

- Now we just need to process events. For this, it's probably just easiest for me to write out the algorithm and explain each step as I go.
- SegmentIntersections(S):

- Insert segment endpoints into event queue.
- While event queue is non-empty, extract the next event point p = (p_x, p_y).
- If p is the left endpoint of segment s:
  - Insert s into sweep-line status for line at p_x.
  - Let s′ and s″ be immediately above and below s.
  - If there is an event for intersection of s′ and s″, remove it from event queue.
  - If s and s′ intersect, add intersection to event queue.
  - If s and s″ intersect, add intersection to event queue.
- If p is the right endpoint of segment s.
  - Let s′ and s″ be immediately above and below s.
  - Delete s from sweep-line status.
  - If s′ and s″ intersect, add intersection to event queue.
- If p is an intersection between s′ and s″:
  - Report intersection of s′ and s″.
  - Swap s′ and s″ in sweep-line status.
  - Remove old events involving s′ and s″ from event queue.
  - Add new intersection events between s′ and s″ and their new respective neighbors.



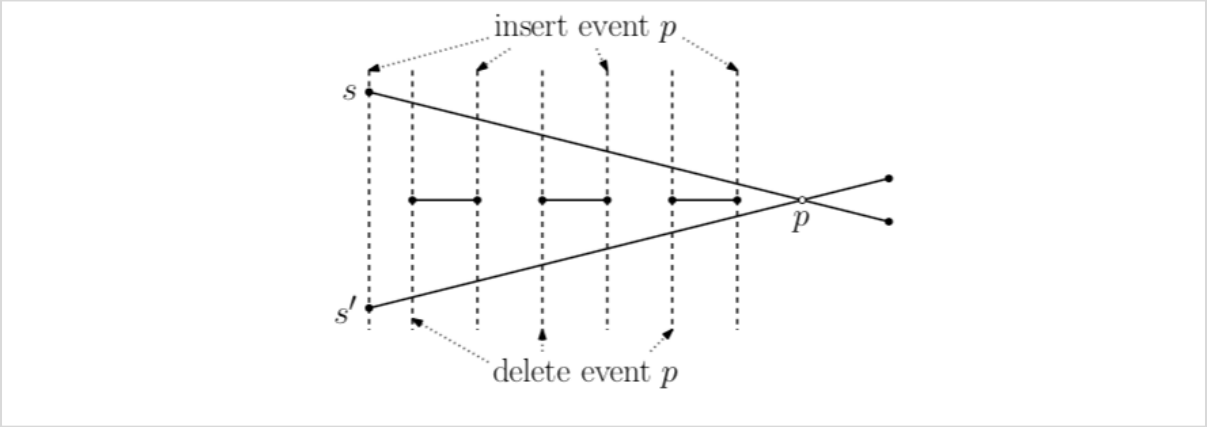
Analysis

- The sweep-line status has ≤ n segments at all times. The event queue has O(n) events at any time.
- Therefore, events take O(log n) time to process.
- I was the number of intersections. There are 2n + I events processed, so the total running time is O((n + I) log n).

- There is also an O(n log n + I) time algorithm I won't be giving in this class, as well as an Omega(n log n + I) lower bound.

Event Deletion

- One final note:
- Why did I find it necessary to delete events when line segments were no longer adjacent? Because you may end up adding the same event Omega(n) times otherwise!



- You could also handle this issue by trying to detect if your event queue already has an event like the book does, but I think that's messier.