15-451/651: Design & Analysis of Algorithms November 16, 2021
Lecture #20a: Sweep-Line and Segment Intersection last changed: November 15, 2021

1 Preliminaries

The *sweep-line* paradigm is a very powerful algorithmic design technique. It's particularly useful for solving geometric problems, but it has other applications as well. We'll illustrate this by presenting algorithms for two problems involving intersecting collections of line segments in 2-D.

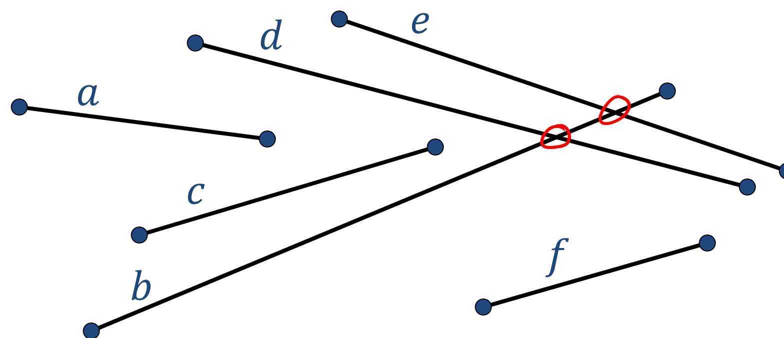
Generally speaking sweepline means that you are processing the data in some order (e.g. left to right order). A data structure is maintained that keeps the information gleened from the part of the data currently to the left of the sweepline. The sweepline moves across absorbing new pieces of the input and incorporating them into its data structure.

Obviously this is very vague. So let's get concrete and solve some problems.

2 Computing all Intersections of a Set of Segments

The input is a set S of line segments in the plane (each defined by a pair of points). The output is a list of all the places where these line segments intersect.

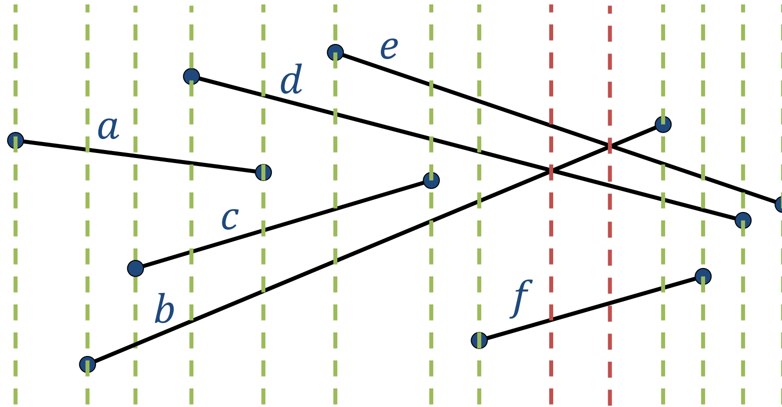
- Input: n line segments in 2D
- Goal: Find the k intersections



As usual, we're going to make our lives easier by making some geometric assumptions. We're going to assume that none of the segments is vertical. We'll also assume that no three segments intersect at the same point. We'll also assume that no segment has an endpoint that is part of another segment. (These assumptions can be avoided by adding some extra cases to the algorithm that do not change the running time bounds.)

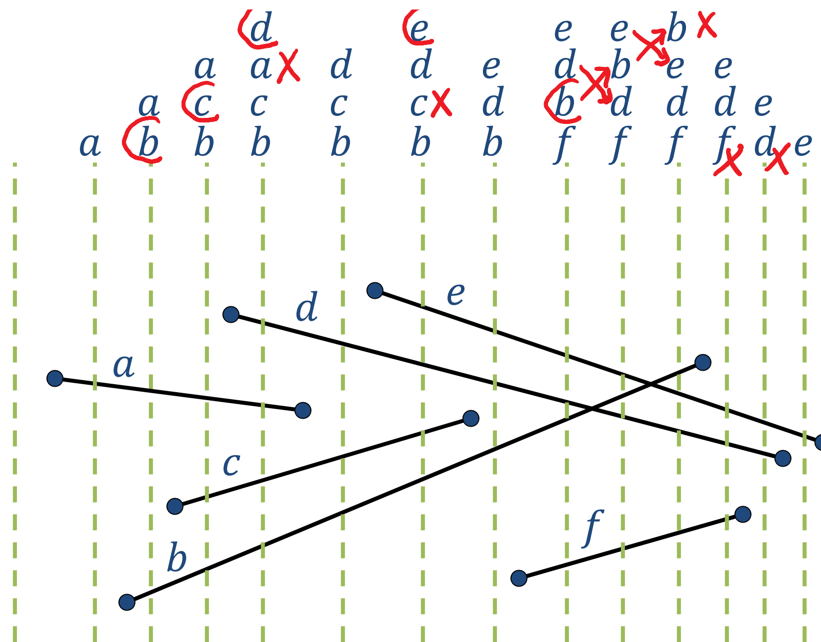
There's a trivial $O(n^2)$ algorithm: Just apply segment intersection to all pairs of segments. The solution we give here will be $O((n+k)\log n)$, where k is the number of segment intersections found by the algorithm.

To get some intuition for what's going to happen, consider the following figure.



Here we have six segments, called a, b, c, d, e, f. There are also two segment intersections. The "interesting" x-coordinates are the ones where something happens: either a segment begins, or a segment ends, or two segments cross.

The following figure shows a vertical dashed line before each interesting x-coordinate. Above the dashed line is a list of the segments in the same order they appear in the diagram, along that dashed line.



Between any two neighboring events, the segments (in that range of x) are in some order by y from bottom to top. This order (and which segments are there) does not change between a pair of neighboring interesting x coordinates.

We're going to maintain a data structure that represents the list shown at the top of the diagram. Let's call this a "segment list". Look at how this list changes as we move across the interesting x values. Only one of three things can happen. (1) A new segment is inserted into the list. (2) A segment is removed from the list. (3) Two neighboring segments in the list cross.

Our goal is to compute the intersections among these segments. The key observation that leads to a good algorithm is that right before two segments cross, they must be neighbors in the segment list. So the key idea of the algorithm is that as our segment list evolves (as we process the interesting x coordinates from left to right), we only need to consider the possible intersections between segments that are neighbors, at some point in time, in the segment list.

So our algorithm is going to maintain two data structures. A segment list (SL) and an event queue (EQ). Each event in the EQ will be labeled with its type ("segment start", "segment end", "segments cross"), as well as the x value where this happens. The EQ is initialized with all the segment starts and segment ends.

The EQ data structure must support `insert`, `findmin`, and `deletemin`. It's just a standard priority queue.

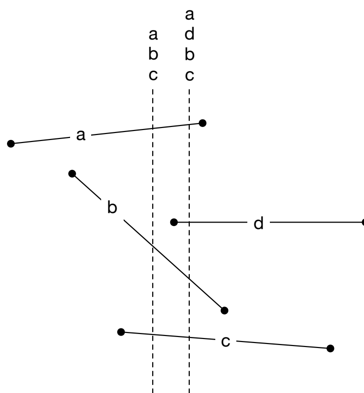
Although a segment is defined by its two endpoints, it's going to be useful to also use a "slope-intercept" representation for the line containing the segment. So segment i will have a left and right endpoint as well as a pair (m_i, b_i) where m_i is the slope of the line and b_i is the y-intercept.

Consider what we need for the SL data structure. The items being stored are a set of segments. The ordering used is the value of $m_i \cdot x + b_i$, where x is the current x value in the ongoing sweep-line algorithm. Requirements for the SL data structure:

- Insert a new segment into the data structure.
- Delete a segment from the data structure.
- Find the successor of a segment in the data structure.
- Find the predecessor of a segment in the data structure.

All of these operations can be done in $O(\log n)$ time using any standard search tree data structure (e.g. splay trees, or AVL trees).

Note in SL, the keys of the nodes store (m_i, b_i) — not the y value at the time of insertion. This is so that when we insert a new segment, we insert it in the right order. An example where this is important is inserting d in the following situation:



In the above example, we want to insert d *above* b even though it is below the y coordinate of b when b was inserted. Since we store (m_i, b_i), we can compute the y-coordinate of all the segments at the current point (here, the start of d). Since the ordering only changes at the interesting points, SL maintains the correct tree ordering.

Now we can write the complete algorithm. Whenever the algorithm says CheckForIntersection (EQ, a, b), this means that the pair of segments a and b are tested for intersection. This function will always

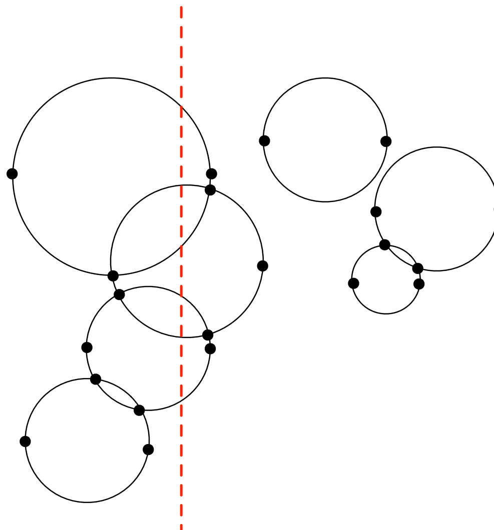
be called when a and b are neighbors in the SL. If they do intersect, CheckForIntersection adds the intersection event to the EQ.

```
def FindSegmentIntersections(S):
        create an empty priority queue EQ
        for each segment S, insert its start and end events into EQ
        create an empty balanced tree SL
        while (EQ is not empty)
                E := deletemin(EQ)
                if E is start of segment s then
                        insert s into SL
                        CheckForIntersect(EQ, s, successor of s)
                        CheckForIntersect(EQ, s, predecessor of s)
                else if E is the end of segment s then
                        CheckForIntersection(EQ, pred of s, succ of s)
                        delete s from SL
                else if E is cross event for segments s1 and s2 then
                        println "s1 and s2 intersect"
                        remove s1 and s2 from SL
                        re-insert s1 and s2 into SL in the opposite order
                        CheckForIntersection(EQ, s1, new neighbor of s1)
                        CheckForIntersection(EQ, s2, new neighbor of s2)
```

It's easy to see that the running time of the algorithm is $O((n+k)\log n)$ using the data structures described. Because there are $O(n+k)$ events, and each one involves a constant number of operations on the SL and EQ data structures.

3 Counting Intersections of Circles (optional)

As an exercise, how would you solve the problem of computing all the intersections of a set of n *circles* in the plane?



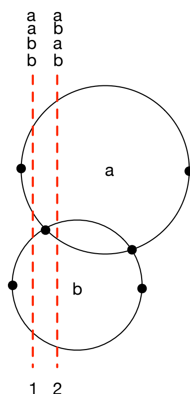
We can again imagine using a plane sweep, walking a vertical line left-to-right across the plane. We have the following events:

Circle Start. The leftmost point on a circle is reached. We need to start tracking the circle.

Circle End. The rightmost point on a circle has been reached. We should stop tracking the circle.

Circle Intersect. Output the intersection, and modify our data structure to account for the intersection.

As with the line segment intersection problem, two intersecting circles will be adjacent on the sweep line just before an intersection.



A complication, however, is that unlike in the line segment case, we don't have a well-defined ordering of the circles along the sweep line. Parts of circle A can be above parts of circle B and at the same time parts of circle A can be below parts of circle B. But notice that at any point in time, the sweep line hits each circle in at most two places: the top semicircle and the bottom semicircle. So, we can represent each circle by 2 semicircles. The sketch of the algorithm is:

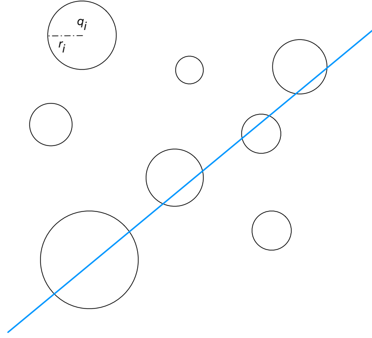
1. A priority queue EQ starts with leftmost and rightmost points of the circles sorted by their x coordinate. As in the point sweep algorithm, intersection points will be added to the queue as we go.

2. When we encounter the start event of circle c, we insert $TopHalf(c)$ and $BottomHalf(c)$ into our balanced tree SL, keyed by the y coordinate of the start event point. When we encounter the end event of a circle c, we remove these two semicircles.

3. Otherwise, the algorithm is the same as for segments: whenever we add, remove, or swap a semicircle, we test its new neighbors for intersection. The intersection test is a bit more complex because we have to test the intersection of two semicircles (but you can work this out on your own).

4 Balloon Pop

Problem: Suppose you are given n non-intersecting circles (balloons) in the plane, where circle i has radius r_i and center point q_i. Find the line that will intersect as many circles as possible.

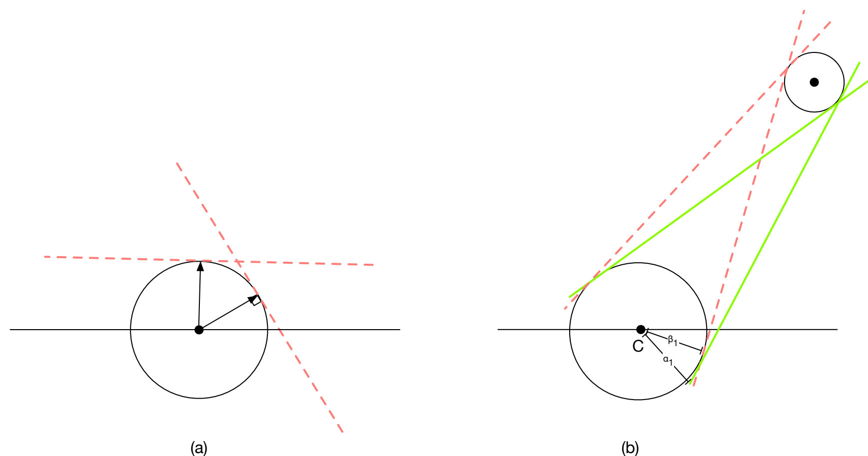


A first try. Note that we can always find a solution that is tangent to one of the circles: take a solution line and translate it until it becomes tangent. In fact, we can always find a solution that is tangent to *two* circles: take a solution line that is tangent to one circle, and rotate it until it becomes tangent to another circle.

That leads to an $O(n^3)$ algorithm: for every pair of circles, compute the four tangent lines that go between them, and for each of those lines, compute (by iterating through the circles) the number of circles they intersect. Keep and return the line with the most balloon hits. (This is an example of a general strategy we have seen before: reduce a problem with a seemingly infinite possible solutions to one with a finite number.)

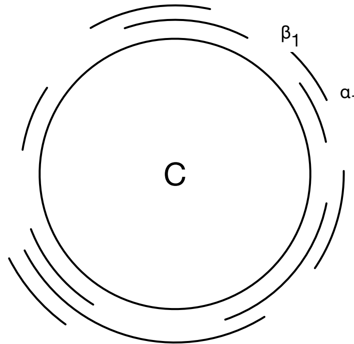
A better algorithm. We can do better, achieving a running time of $O(n^2 \log n)$. This algorithm uses several plane sweeps of a rotating line.

We first use the fact that some optimal line must be tangent to some circle. We guess that it will be circle C (we'll try all circles for this role). Every tangent line to C is a candidate, and each tangent line is specified by some angle α. So the idea is to do a sweep line, but instead of moving the line along the x axis, we rotate it around as tangents to the circle C. See figure (a):



As we are sweeping this line, what are the interesting events? They are when the sweep line first intersects and then leaves (exits) another circle. For a circle i, these occur at angles $(\alpha_{i,1}, \beta_{i,1})$ and $(\alpha_{i,2}, \beta_{i,2})$, where the first element in each pair is the "entering" angle and the second element is the corresponding "leaving" angle. We can compute all of these angles in $O(n)$ time by enumerating through the circles.

These angles define intervals on the border of C (corresponding to the range of sweep lines that will hit each circle):



The angle of greatest depth of coverage by intervals is the tangent line that hit the largest number of circles. To find this most highly covered angle, we can sort the angles, and visit them in increasing order. This corresponds to walking around C. We keep a counter depth, which we increment whenever we encounter an "entering" event, and we decrement whenever we encounter a "leaving" event. We also track the max value of the counter (and corresponding tangent).

One last technicality: when we start walking around the circle C, it may be that some intervals cross the 0 angle. To handle this, we count those explicitly by making a first pass through the intervals (in $O(n)$ time). We start our depth counter at the number of intervals that cross the 0 angle.

To summarize:

```
def MostBalloonPoppingLine(S):
   for C in S:
      for every Q in S \ C: compute α_{Q,1}, β_{Q,1}, α_{Q,2}, β_{Q,2}
      depth = max_depth = number of (α, β) intervals crossing angle 0
      L = sorted list of the α, β angles
      for each angle γ in L:
         if γ is entering angle: depth++
         if γ is leaving angle: depth--
         if depth > max_depth
            max_depth = depth
             best_line_for_C = γ
      end
      if max_depth > global_max_depth:
         best_line = (C, best_line_for_C )
end
```

The running time for each C is $O(n)$ to compute the angles + $O(n \log n)$ to sort the angles + $O(n)$ to walk through the angles to find the max. We have to do this all $O(n)$ times, leading to a final run time of $O(n^2 \log n)$.

4.1 Computing α and β angles

There are a few different cases, but they can all be solved with elementary trigonometry using the fact that a tangent line and the line connecting the centers of the circles form a part of similar right triangles. Consider the case at right: the triangle $\triangle CRA$ is similar to the triangle $\triangle QTA$. We know the lengths of $\overline{QT}$ and $\overline{CR}$ segments (they are the circle radii, r_Q and r_C) and we know the distance d between the circle centers. We can therefore solve for d_Q:

$$\frac{d_Q}{d + d_Q} = \frac{r_Q}{r_C} \implies d_Q = \frac{r_Q d}{r_C - r_Q}$$

when $r_C \neq r_Q$ (when the radii are equal, the angle of the tangent line is just the angle between their centers). The angle QCR can be computed as:

$$\cos^{-1} \frac{r_c}{d + d_Q}$$

which can be subtracted from the angle $\angle 0CQ$ to obtain α_1.

The case where the tangent crosses the $\overline{CQ}$ line is similar.