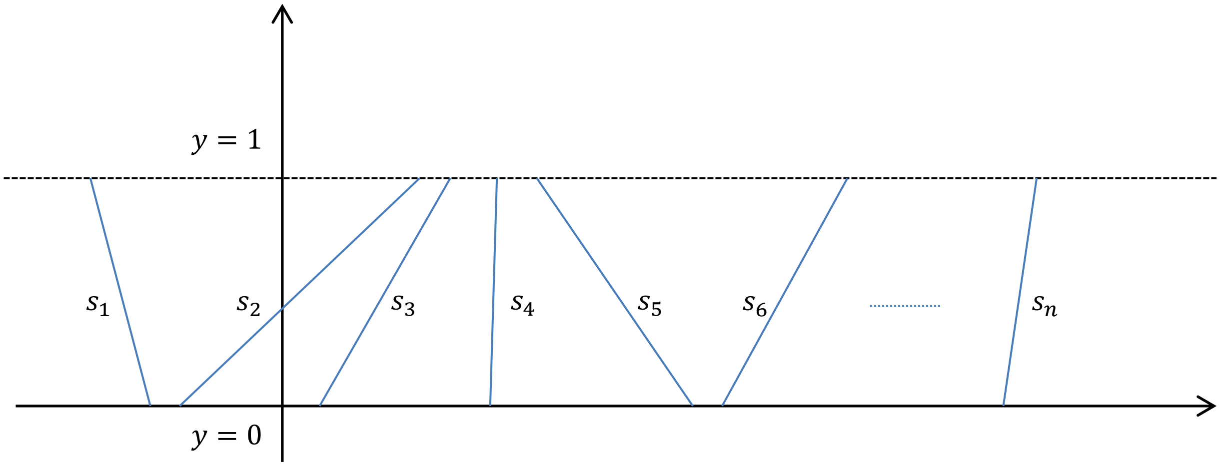
CS623 Winter 2012 \ Assignment #2

Ariel Stolerman

1)

Let S be a set of n disjoint line segments whose upper endpoints lie on the line $y = 1$ and whose lower endpoints line on line $y = 0$, partitioning the horizontal strip $[-\infty: \infty] \times [0: 1]$ into $n + 1$ regions. Following is a $O(n \lg n)$ algorithm top build a binary search tree of the segments in S such that the region containing a query point can be determined in $O(\lg n)$ time:

- Sort all segments in S by their upper endpoint x-coordinate and insert them into a **balanced** binary search tree, such that they are held according to that key. For any segment s_i, also hold at each node:
    - The segment line equation of the form $y(x) = ax + b$.
    - The line equations for the segments s_{i-1}, s_{i+1} – to the right and left of the segment in this node according to the sorting (if s_i is the minimum/maximum according to the sorting, hold *nil* for the left/right neighbor segment equation, respectively).



Following is a description of the query algorithm in detail, given some point $p \in \mathbb{R}^2$:

If p is not in any of the regions, the algorithm returns false. Otherwise, if p is between segment s_i and s_j (such that s_i is to the left of s_j), it returns (i, j); if p is to the right of the rightmost segment s_k, it returns $(k, -1)$, and to the left of the leftmost segment s_l - $(-1, l)$. Denote the endpoints of all segments s_i as $s_i.top, s_i.bottom$.

- If $p.y \notin [0,1]$, return false.
- Let $x \coloneqq p.x$
- Go down the tree constructed earlier, and at each node that contains the segment s_i:
    - Check if p is to the left of s_i by checking that $\angle(s_i.top, s_i.bottom, p) \leq 180°$
        - If it is, check if it is contained in the region between s_{i-1} and s_i (by checking the angle as before with the remaining 3 edges of the region). If yes, **return $(\boldsymbol{i-1}, \boldsymbol{i})$**.
        - If it's not continue to the next node on the **left** subtree.

  - If it is not to the left of s_i:
    - Check if it is contained in the region between s_i, s_{i+1}. If yes, **return $(\boldsymbol{i}, \boldsymbol{i}+\mathbf{1})$.**
    - if not, continue to the next node on the **right** subtree.

  For the checks above, if it is an extreme point (s_1 or s_n), checking if it is to the left of s_1 or to the right of s_n determines also whether those regions contain the point.

Correctness:

Since the segments are distinct, any region is a simple quadrilateral (or one-ended strip for s_1 or s_n), and for any two segments s_i, s_j such that $s_i.top.x < s_j.top.x$, s_i is completely to the left of s_j. Thus sorting the segments by their top x-coordinate sufficiently orders the segments. If a point is not contained in the strip, the algorithm will immediately return false. Otherwise it must be contained in one of the regions, and when strolling down the tree, if it is to the left of some segment s_i then it must be to the left of all segments $s_{i+1}, \dots, s_n$, and the same for the right. Therefore eventually the containing region will be found.

Running-time:

Sorting and constructing the balanced tree is $O(n \lg n)$. In addition, any query will take at most $O(\lg n)$ steps (since it is a balanced tree), and at each node there is a constant number of operations that need to be done to determine if p is in any of the left or right regions, or should the search continue (and in what direction). Therefore any query will take $O(\lg n)$ time.

2)

Following is a description of a plane sweep algorithm to compute all intersection points between circles in a set S of n circles. We address this as a x-axis sweep algorithm, i.e. we simulate running a line $x = a$ where $a \in (-\infty, \infty)$ (starting at $-\infty$). Also, each circle $C \in S$ is represented by its center $o = (a, b)$ and radius r (and we may assume each is unique, otherwise it is easily checked). The algorithm follows the original plain sweep algorithm with the following changes:

- The initial events in the event priority queue are the leftmost and rightmost points on each circle, i.e. $(a - r, b)$, $(a + r, b)$. Each circle is represented by an equation of the form $(x - a)^2 + (y - b)^2 = r^2$.
- The status of the sweep line will be held in a balanced tree, and each circle will be held as 2 halves (arcs), each represented by the circle (center point and radius) and top / bottom (the split into halves is by the diameter parallel to the x-axis).
  - On insertion, both arcs are inserted as neighbors.
  - When checking events and updating the status, it might be that the two arcs will become not-neighbors.
- If a start-circle event is encountered, its 2 arcs are inserted as neighbors into the status, and each of them is checked for intersection with their neighbors in the status (besides the 2 arcs themselves, who are initially neighbors). Calculate the distance $d = \delta(o_1, o_2)$ of the inserted circle centered in o_1 and its neighbor in o_2:
  - If $d > r_1 + r_2$ the circles are distant from one another and there is no intersection point.
  - If $d < |r_1 - r_2|$ one circle contains the other and there is no intersection point.

  - If $d = r_1 + r_2$, there is one intersection point on the line segment $[o_1, o_2]$ (distant by r_1 from o_1). **Print it and continue** (without adding it as an event to the queue).
  - If $d < r_1 + r_2$ there are 2 intersection points p_1, p_2. Let p be the point of intersection of the line segments $o_1 o_2$ and $p_1 p_2$, then we get 4 right triangles $\Delta o_1 p p_1, \Delta o_1 p p_2, \Delta o_2 p p_1, \Delta o_2 p p_2$ and by a set of equations we can calculate p_1, p_2. **We print both of them to the screen and add them both to the priority queue**.
- If an end-circle event is encountered, its 2 arcs are extracted from the status and their previous neighbors are checked for intersections with their new neighbors as described above.
- If an intersection event is encountered, the two intersected arcs should be switched in the status, and intersection with their new neighbors are to be checked as described above.

Correctness:

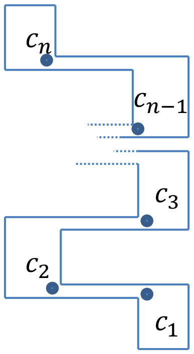
Most of the correctness is derived from the original sweep algorithm. As for the changes: a circle becomes relevant once the sweep line first intersects it, which is on the circle's leftmost point. It becomes irrelevant once the sweep line stops intersecting it. If two circles intersect, that will be found out when they are neighbors by at least 1 arc. When they intersect at only one point, no need to add that point as an event, since right before that point and right after it, they must have the same relation to each other (the left circle stays the other one's left neighbor all along and vice versa; this can be proven more rigorously, showing that if that's not the case we can always take a small ϵ-environment around the intersection point for which it is true). If they intersect at 2 points, their neighbors before and after each of the intersection points may change, so these points must be added as events. It may be the case that both circles are printed as intersecting in those 2 points twice, but that's ok (also running-time wise).

Running time:

The running time doesn't change asymptotically: checking intersection between 2 circles is still constant, and when there are 2 intersection points they are printed twice, increasing the k part by a factor of at most 2. Therefore the total running time stays $O\big((n+k)\lg n\big)$.

3)

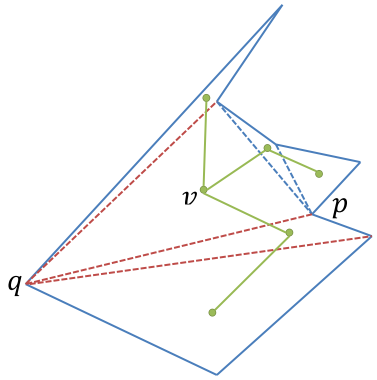
Following is an example of a rectilinear polygon with n vertices that necessitates $\lfloor n/4 \rfloor$ cameras to guard it:



Clearly each rectangular piece of this polygon necessitates its own camera, thus any 4 vertices that make that piece need to be covered by one camera. The total number of cameras is then ¼ of the number of vertices, as required.

4)

The statement is **false**, as a dual graph of the triangulation of a monotone polygon may not be a chain, even when following the triangulation algorithm in the book (3.3, page 55). Consider the following example:



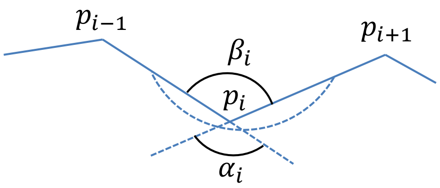
Clearly the polygon is monotone with respect to the y-axis. The triangulation algorithm will draw diagonals for the first time from p (the two dashed blue diagonals), next from q (the 3 dashed red diagonals) and by that will finish the triangulation. However, the dual graph (in green) of that triangulation is not a chain, as the vertex v has degree 3. Note that the polygon above has a triangulation that derives a chain dual graph (crossing all diagonals possible from q), but that is not the triangulation derived from the algorithm in the book.

5)

Following is an algorithm that determines whether a polygon P with n vertices is monotone with respect to any line, i.e. there exists some line l such that P is monotone with respect to l.

The idea of the algorithm is to find a range of angles (a wedge) in which a slope of a legal line l (a line that the polygon can be monotone with respect to) can reside. For that purpose we need to look at any vertex of P with interior angle > 180, as those inner corners impose a restriction on the slope of any l as described above: the slope of l's perpendiculars must be in that range. The algorithm will run as follows:

- Initialize a polar wedge $\alpha = 2\pi$ (i.e. a whole circle).
- Run over each vertex p_i in the chain $p_1, p_2, \dots, p_n$ describing P in order (consider the subscripts module n):
    - If the P-interior angle $\angle p_{i-1}p_ip_{i+1} > \pi$, denote its corresponding wedge α_i as the wedge between the two extensions of $p_{i-1}p_i$ and $p_{i+1}p_i$ towards inside P, and β_i – towards outside P:



  - Update $\alpha \coloneqq \alpha \cap (\alpha_i \cup \beta_i)$, and if α is empty, return **false.**
- If α is still not empty after going over all vertices in P, return **true**.

Correctness:

Note that any interior angle ≤ 180 doesn't impose anything on any candidate l. The slope that each interior angle > 180 imposes on such l is that its perpendiculars can have a slope only within $\alpha_i \cup \beta_i$, otherwise p_i will break the monotonicity of P with respect to it. Therefore the intersection of all such α_i (starting at $\alpha = 2\pi$, signifying there are initially no restrictions) expresses all restrictions together. Therefore if after going over all vertices α is still non-empty, we can pick a line l with perpendicular slope in that range, and we know P will be monotone with respect to it.

Running time:

Going over the entire chain, at each vertex we do a constant amount of work:

- Checking the interior angle is constant using simple orientation tests.
- Calculating α_i and β_i is also constant given the 3 consecutive vertices p_{i-1}, p_i, p_{i+1}.
- Calculating the intersection $\alpha \cap (\alpha_i \cup \beta_i)$ can be done in constant time, if α (and α_i, β_i) is represented by a starting and ending angles.

Therefore the total amount of work is $O(n)$, (efficient) as required.