CS91T: Computational Geometry **Fall 2023**

Homework 2

David Yang and Nick Fettig

1. **Suppose you are given n points $p_1, \ldots, p_n \in \mathbb{R}^2$, and you want to know whether connecting these points in order will give you the boundary of a simple polygon. Give an $O(n \log n)$-time algorithm for checking this.**

   Assume P is formatted as an array list of all points $p_i, \ldots, p_n$. Also assume we have a line segment class to hold points.

Algorithm IsSimplePolygon(P)

```
1: Set EP := empty list, will hold stored event points
2: Set LS := empty list, will hold stored line segments
3: for each i ∈ {1, ..., n} do
4:     Append line segment ℓ_i = p_i p_{(i+1)%n} to LS
5:     Append two endpoints of ℓ_i to EP
6: end for
7: Sort event points in reverse order of y-coordinate
8: Run modified[1] PlaneSweep algorithm on set of line segments LS.
9: if at any point, there are any other intersections then
10:    return false
11: end if
12: if PlaneSweep concludes without returning then
13:    return true
14: end if
```

Runtime Analysis: In our algorithm, lines 1 to 6 represent our preprocessing steps. In lines 1 and 2, we create empty lists used to hold our event points and line segments, respectively. In lines 3 to 5, we create n line segments corresponding to our n potential edges of the simple polgon, and add $2n$ endpoints to our event points list; this totals $O(n)$ time. In line 6, we call a modified version of PlaneSweep, which runs in $O((n+k)\log n)$ time for n points and k incidences between segments and intersection points. By construction, our modified PlaneSweep algorithm will halt if there are intersections that are not at the vertices of the boundary; since there are $O(n)$ such vertices, there are at most $m = O(n)$ incidences considered by PlaneSweep and so our algorithm will run in $O(n + O(m)) \log n = O(n \log n)$ time, as desired.

Proof of Correctness: Connecting the n points $p_1, p_2, \ldots, p_n$ will only give the boundary of a simple polygon if these connections are piecewise linear, simple, and closed. The piecewise

[1] We allow line segments ℓ_i, ℓ_{i+1} to intersect at $p_{(i+1)\%n}$ and do not count these towards any intersections.

linear and closed conditions are given by construction, and so it suffices to check that the connections are simple, meaning the edges do not intersect.

By definition, our PlaneSweep algorithm will do just that – clearly, consecutive edges ℓ_i, ℓ_{i+1} are allowed to intersect at $p_{(i+1)\%n}$, but any other intersection corresponds to a violation of the simple condition for a simple polygon.

On the other hand, if our PlaneSweep algorithm does not find any other intersections, we have the boundary of a simple polygon, as desired.

2. **Give an $O(n \log n)$-time algorithm to determine whether there are any overlapping disks in a set of n disks.**

Assume we are given an array list of disks, D. Given the disks, we can easily find each disk's center point and radius in D, and this information will also be passed into the algorithm.

Algorithm OverlappingDisks(C)

1: Set $A \coloneqq$ empty list, will hold stored arcs and the disk they originate from
2: Set $EP \coloneqq$ empty list, will hold event points
3: **for each** $i \in \{1, \ldots, n\}$ **do**
4: Append highest and lowest vertical points in disk D_i to EP (along with value i)
5: Split D_i at those points; append the resulting two arcs of D_i to A (along with value i)
6: **end for**
7: Sort event points in reverse order of y-coordinate
8: Run PlaneSweep algorithm on event points with the extra conditions:

- Maintain a left-to-right ordered list of arcs at y-coordinates of event points using a BST
- Ignore intersections between arcs belonging to the same disk
- For each new neighbor relation introduced:
  - Let D_i and D_j be the circles that the arcs with the given endpoints belong to
  - If the distance between the center of D_i and the center of D_j is less than the sum of the radii of D_i and D_j, **return** true
  - Otherwise, insert/remove arc in BST

9: **if** at any point, there are any other intersections **then**
10: **return** true
11: **end if**
12: **if** PlaneSweep concludes without returning **then**
13: **return** false
14: **end if**

Runtime Analysis: Lines 1 to 6 in our algorithm constitute the preprocessing steps. In lines 1 and 2 we create (initially) empty lists A and EP, which will hold our stored arcs and event points. In line 4, we split each of the n disks vertically, appending the highest and lowest points (in terms of y-coordinate) to the list of event points and in line 5, we append the two resulting arcs to the list of arcs; since there are $O(2n)$ event points and $O(2n)$ arcs, we can do this appending in $O(n)$ time.

In line 7, we sort our $O(n)$ event points in reverse order of y-coordinate so that the highest vertical points are listed first; this sorting takes $O(n \log n)$ time. Finally, in lines 8 to 14, we run a modified version of the PlaneSweep algorithm that ignores intersections between arcs belonging to the same disk and that performs another check between neighboring circles

for their corresponding arcs. PlaneSweep runs in $O((n+m)\log n)$ time for n points and m intersection points; here, our modification ignores the $O(n)$ intersections between arcs of the same disk and if any other intersection occurs, the algorithm terminates and we return true. Our extra check that compares the centers of the circles and the sum of their radii is a constant time check for each new neighbor relation introduced. Since each of our $O(n)$ event points introduces a constant number of new neighbor relations, the total time from these checks is $O(n) \cdot O(1) \cdot O(1) = O(n)$. Consequently, our modified PlaneSweep algorithm runs in $O((n + O(n)) \log n) + O(n) = O(n \log n)$ time.

Our preprocessing takes $O(n)$ time, our sorting takes $O(n \log n)$ time, and the PlaneSweep algorithm takes $O(n \log n)$ time; thus, our overall algorithm runs in the desired $O(n \log n)$ time.

Proof of Correctness: We will prove the correctness of this algorithm using induction on the number of event points already handled; more specifically, we will show that the order of the arcs in our BST is properly maintained at each iteration. For the base case, observe that our algorithm correctly handles the first event point; PlaneSweep simply adds the arc of the event point to a BST, and the order is correct given there is just one arc. For the inductive step, we will show that our algorithm maintains the proper segment ordering (or concludes) at a new event point given that the previous event points are correctly handled. When a new event point is reached, if it is the endpoint of a previous arc, it will be removed from the BST. On the other hand, if it constitutes a new arc, we will perform an extra check (explained below) and insert it into the proper position in the BST. Since the order of the arcs in the BST can only change at event points, we know that the order of the BST is properly maintained after handling this new event point, as desired. Thus, by induction, our modified PlaneSweep properly handles the order of the arcs at each event point.

PlaneSweep with no modifications finds intersections between the arcs of each disk; equivalently, it detects intersections between the boundaries of the disks. We ignore "trivial" intersections between arcs belonging to the same disk since such an intersection does not correspond to a pair of overlapping disks. It remains to show that our additional modification to the PlaneSweep algorithm detects a disk that is fully contained in another since such an intersection will not be found by a typical PlaneSweep algorithm.

To detect such a pair of disks, we claim that it suffices to check the distance between the center of the disks and compare it to the sum of the radii of the disks. More formally, we claim that disk D_i overlaps D_j if and only if the sum of the radii of D_i and D_j is greater than the distance between their centers. For the forward direction, note that if D_i and D_j overlap, then the distance between their centers is the sum of their radii minus some nontrivial length corresponding to their overlapping region, so the sum of their radii must be greater than the distance between their centers. For the reverse direction, we appeal to the contrapositive; if D_i and D_j do not overlap, the distance between their centers is greater than the sum of their radii, and so we are done.

Thus, our modified PlaneSweep algorithm, which checks an extra condition between neighboring disks at event points, will find any overlapping disks in a set of n disks, as desired.

3. **Let S be a set of n triangles in the plane with disjoint boundaries. Let P be a set of n points in the plane. Give an $O(n \log n)$-time algorithm that lists all points in P that lie outside of all triangles.**

Algorithm PointsOutsideTriangles(S, P)

1: Set $EP :=$, an empty list that will store event points
2: Set $E :=$, an empty list that will store edges and the triangles they belong to
3: Set $O :=$ an empty list which will store points outside of all triangles.
4: **for** each triangle $T_1, \ldots T_n$ in S **do**
5: Find the three vertices of the triangle and append each vertex to EP, along with the number i signifying it is a vertex of triangle T_i
6: Append each edge of triangle T_i to E, along with the number i signifying it is an edge of T_i
7: **end for**
8: **for** $i \in \{1, \ldots, n\}$ **do**
9: Append point P_i to EP
10: **end for**
11: Sort points in EP by reverse order of y-coordinate
12: Run vertical PlaneSweep (sweeping down) algorithm on event points in EP by:

- Maintaining a left-to-right ordered list of segments at y-coordinates of event points using a BST
- At each event point v' corresponding to a vertex of a triangle in S
  - Consider the two (if they exist) neighboring segments of v' in the BST and the triangles they belong to
  - If v' lies in one of those triangles, disregard it (and make a note to disregard all other vertices of the triangle v' in EP)
  - Otherwise, add it to the BST
- At each event point p' corresponding to a point in P
  - Consider the two (if they exist) neighboring segments of p' and the triangles they belong to
  - Check if p' lies outside each of those triangles; if so, add to O

13: **return** O

Runtime Analysis: Lines 1 to 10 constitute the preprocessing phase of our algorithm; we create empty lists and then go through each of the n triangles in lines 4 to 6, finding each of the three vertices to a given triangle and appending the resulting details to their appropriate lists. This takes $O(n)$ time. Similarly, in lines 8 to 10, we append each of the n points in P to the list EP, which takes $O(n)$ time.

In line 11, we sort the event points by reverse order of their y-coordinate (so the point with the greatest y-coordinate is first in the list); since there are $3n$ triangle vertices in EP and n points from P in EP, we have a total of $4n$ event points, and this sorting runs in

$O((4n)\log(4n)) = O(n \log n)$ time.

Next, in line 12, we run the vertical PlaneSweep algorithm. We maintain a left-to-right ordered list of segments at the y-coordinates of our event points using a self-balanced BST, which requires $O(4n \log n) = O(n \log n)$ time to maintain. We add the first check (checking event points v' corresponding to vertices in S) as it helps us remove triangles that are enclosed in another. To check whether a new vertex v' is part of one of these triangles, we simply check its two neighboring segments and if those segments belong to the same triangle, check whether v' is enclosed in that triangle; this is a constant time check, to do so, we rely on the fact that v' is enclosed in a triangle if and only if it is to the left of each edge of a triangle (in CCW direction). There are $O(n)$ checks for the $3n$ vertices of our triangles, so this first check totals $O(n)$ time.

Similarly, when checking if a point $p' \in P$ lies outside of all triangles, we simply check the two neighboring segments of p', and check that p does not lie in the corresponding triangles these segments belong to. Once again, each of these checks can be done in $O(1)$ time. If it is outside both of these triangles, we simply add it to our list O, which requires constant time. Since we do a check at each point in P, we do $O(n)$ checks, each running in constant time. Thus, the PlaneSweep algorithm concludes in $O(n \log n)$ time.

Proof of Correctness: The correctness of our algorithm relies on the correctness of the PlaneSweep algorithm. We will prove the correctness of this algorithm using induction on the number of event points already handled; more specifically, we will show that the order of the triangles in our BST is properly maintained at each iteration. For the base case, observe that our algorithm correctly handles the first event point; PlaneSweep simply adds the segment of the event point to a BST, and the order is correct given there is just one point.

For the inductive step, we will show that our algorithm maintains the proper segment ordering (or concludes) at a new event point, given that the previous event points are correctly handled. When a new event point is reached, if it is the second endpoint of a segment, it will be removed from the BST. On the other hand, if it constitutes a new segment, we perform extra checks (with the neighboring segments in the BST, which we can find using binary search) and insert it into the proper position in the BST. Since the order of the segments of the triangles in the BST can only change at event points, we know that the order of the BST is properly maintained after handling this new event point, as desired. Thus, by induction, our modified PlaneSweep properly handles the order of the arcs at each event point.

Note that our event points include the $3n$ vertices of triangles in S as well as the n points in P. We run PlaneSweep with two modifications. As we typically do, we maintain a self-balanced BST of segments at a given y-coordinate as we sweep down from the highest event point.

However, when a new event point is reached, we perform two separate actions depending on whether that point is a vertex in S or a point in P. For vertices in S, we check the two neighboring segments of this event point (which we can find in $O(1)$ time due to our balanced BST) and check whether this vertex is enclosed in the corresponding triangles – if it is, we

disregard the triangle the new event point lies in. Since the n triangles in S have disjoint boundaries, if a vertex v' from a triangle t_1 in S lies in another triangle t_2, this means that triangle t_1 is enclosed in triangle t_2. By adding this constant time check as an extra condition when event points are considered, we maintain an invariant that no triangles in S are enclosed in other triangles.

On the other hand, when an event point $p' \in P$ is reached, we can once again check the neighboring segments of P from the BST and determine the triangles these neighboring segments correspond to. Since our previous condition maintains the invariant that no triangles are enclosed in others, we know that p' is outside of all triangles if and only if p' is outside of the triangles the neighboring segments correspond to. Thus, a constant time check with the two neighboring triangles of the point p' suffices to confirm/deny whether p' lies outside of all triangles in S.

Thus, our modified PlaneSweep algorithm, with two extra conditions (one to maintain the invariant that no triangles are enclosed in another and the other to check whether a point is outside of all triangles in S), will correctly determine all points in P that lie outside of all triangles, as desired.

4. **Suppose you are given a point $z \in \mathbb{R}^2$ and a set S of n disjoint line segments. A segment $\overline{p_i q_i}$ is *visible from z* if there is some segment that connects z to any point on $\overline{p_i q_i}$ and does not intersect any other segments in S. Design an $O(n \log n)$-time algorithm that uses an angular plane sweep — with a rotating ray sweeping around z — to identify all segments that are visible from z.**

   Assume we are given an array list, S, that contains a pair of the points p_i and q_i that make up each line segment, and the point z to check visibility from.

Algorithm FindVisibleSegments(S, z)

```
1:  Set active := an empty AVL tree
2:  Set notVisible := an empty AVL tree
3:  Set ret := an empty Array List
4:  Set sweep := an Array List of each point (in polar) in S
5:  Sort sweep by angle above the horizontal using merge-sort
6:  for each i ∈ {1, ..., 2n} do
7:      if the extension of line z pt_1 intersects with the line belonging to pt_i then
8:          Add the line belonging to pt_i to active on basis of its position between existing
              regions of active[2]
9:      end if
10: end for
11: for each pt in sweep do
12:     if line of pt is in active then
13:         if line of pt is not in notVisible then
14:             Add line of pt to ret
15:         else
16:             Remove line of pt from notVisible
17:         end if
18:         Remove line of pt from active
19:         if the smallest valued line in active is in notVisible then
20:             Remove smallest valued line in active from notVisible
21:         end if
22:     else
23:         Add line belonging to pt to active on basis of its position between existing
              regions of active (using the same BS method as above)
24:         if line of pt is not the smallest valued line in active then
25:             Add line of pt to notVisible (where notVisible is also ordered by relative
              regional position)
26:         end if
27:     end if
28: end for
29: return ret
```

Explanation: We are only checking the $2n$ endpoints of each line. The *active* AVL tree

[2] we can add by binary searching the existing regions.

contains segments where we have not yet found the second endpoint of the segment. The *notVisible* AVL tree contains segments we do not currently see (since some other segment is blocking their previously seen endpoint). Both of these AVL trees are ordered by the relative position of the segments relative to the most recent event point.

Runtime Analysis: Lines 1 to 10 constitute the preprocessing part of the algorithm. We initialize a collection of AVL trees, and array lists, which can all be done in constant time (no values are added just yet). We also perform an initial sweep of the points and sort them by their angle with the horizontal, which can be done in $O(n \log n)$ time. This gives us a list of the points in the order they appear in the counterclockwise direction, stored in *sweep*. In lines 6 to 10, we find the "active" lines at the start of our algorithm; put simply, these are the lines that intersect the extended segment $\overline{z\, pt_1}$; this runs in $O(n)$ time since there are at most n lines that intersect this initial segment. Keeping track of the relative order of these lines takes $O(n \log n)$ time – we use binary search, which runs in $O(\log n)$ time, to find the right region for each of the $O(n)$ lines intersecting the initial line.

Lines 11 to 30 represent the bulk of the algorithm: for each of the points in *sweep* (corresponding to the $2n$ endpoints of the line segments in S), we do two distinct actions, depending on whether we have already seen the other endpoint or not in our angular sweep. The check to see whether the other endpoint has been seen or not can be done in $O(\log n)$ time by searching for the desired line segment in *active*.

Lines 12 to 21 correspond to the case where the line an event point corresponds to is already in *active*, meaning the event point is the second seen endpoint of its line. In this case, we perform a constant number of checks/removals from $notVisible$ and *active*; each of these run in $O(\log n)$ time since we are checking/removing from a BST; thus, this case concludes in $O(\log n)$ time.

Lines 22 to 27 correspond to the other case, where the second endpoint of the current event point has not yet been seen in the angular sweep. In this case, we insert the line corresponding to the event point into the proper position in *active* by binary searching the disjoint regions created by the lines in *active* at the angle of the event point[3]; this runs in $O(\log n)$ time and the insertion into *active* will also take $O(\log n)$ time. Finally, we check an extra condition and perform an extra insertion, which runs in $O(\log n)$ time.

There will be $2n$ for-loop iterations because there are two endpoints for each of the n lines in S. Therefore, the following represents the total run-time of this algorithm:

$$\underbrace{O(n \log n)}_{\text{merge-sort}} + \underbrace{O(2n * (c \cdot O(\log n))}_{\text{angular sweep algorithm}} = O(n \log n).$$

Proof of Correctness: We will prove the correctness of this algorithm using induction on the number of event points already handled; more specifically, we will show that the order of the line segments in *active* and $notVisible$ is properly maintained at each iteration. For the

[3]this works since the lines are disjoint and thus there are no intersections; the binary search is very similar to Lab 3 Question 1

base case, observe that our algorithm correctly handles the first event point; we simply add the segment of the event point to a BST, and the order is correct, given there is just one point.

For the inductive step, we will show that our algorithm maintains the proper segment ordering and gives the correct output at a new event point, given that the previous event points are correctly handled. We once again split our algorithm into two cases.

In the first case, the other endpoint of the segment has already been properly handled. If the segment is not in $notVisible$, then it is visible from z, and thus we add it to ret. If it was, we remove it from $notVisible$. Afterward, we remove the line segment from $active$ since both endpoints have been considered. Finally, we perform an additional check to remove the smallest valued line in $active$ from $notVisible$ if it was in $notVisible$ since it may have been previously blocked from the line segment we've just removed. These steps all work as intended. These checks and removals consequently preserve the ordering of the segments from z to the current angular sweep.

In the second case, where we have not yet seen the segment corresponding to the event point, we insert it into the corresponding position in $active$ by binary searching the disjoint regions created by the lines in $active$ at the angle of the event point. By binary searching for the correct region and inserting the line in the corresponding spot in $active$, we once again maintain the ordering of the segments based on their position from the current angular sweep from z to the event point.

In both cases, our algorithm maintains the proper segment ordering and gives the correct output at a new event point, given that the previous event points are correctly handled. Thus, by induction, our angular plane sweep algorithm correctly identifies all segments visible from z, as desired.