CS91T: Computational Geometry **Fall 2023**

Homework 1

David Yang and Nick Fettig

1. **Let $S, T \subseteq \mathbb{R}^2$. Carefully argue that CH(S) and CH(T) are disjoint if and only if there is some line that separates S and T, meaning that S is entirely on one side of the line and T is entirely on the other side of the line.**

   We will prove both directions of the if and only if.

   Let us begin by proving the reverse direction. Assume that there exists a line ℓ that separates S and T. Note that ℓ separates the plane into two half-planes. Let us use S' to denote the half-plane containing S and no points of T, and T' to denote the half-plane containing T and no points of S. By definition, CH(S) is the intersection of all half-spaces containing S. Since one such half-space is S', we know that CH(S) lies completely on the side of ℓ containing points in S. Similarly, CH(T) is the intersection of all half-spaces containing T. Since one such half-space is T', we know that CH(T) lies completely on the side of ℓ containing points in T. Thus, since CH(S) and CH(T) lie on opposite sides of the line ℓ, they must be disjoint, as desired.

   It remains to show the forward direction. Let CH(S) and CH(T) be disjoint. Then there is some minimum distance between CH(S) and CH(T).[1] Let $\overline{AB}$ be this minimal line segment, with $A \in$ CH(S) and $B \in$ CH(T). We claim that the perpendicular bisector of $\overline{AB}$ is a line ℓ that separates S and T. For the sake of contradiction, assume that ℓ does not separate S and T. Then without loss of generality, there must be some point $S' \in S$ on the opposite side of ℓ from A. Since $S' \in S$, by definition, the convex hull of S must include the segment $\overline{AS'}$, Drop the altitude from B to $\overline{AS'}$ and denote the point on $\overline{AS'}$ as C. By the Pythagorean Theorem, we know that

   $$\overline{BC}^2 + \overline{AC}^2 = \overline{AB}^2$$

   so we must have that $\overline{BC} < \overline{AB}$. However, C is in the convex hull of S (since it is on $\overline{AS'}$) and B is in the convex hull of T, and so we have found $\overline{BC}$, a segment connecting points in the convex hulls of S and T that has smaller distance than our assumed minimum distance segment $\overline{AB}$. We've arrived at a contradiction, and so we know that ℓ separates S and T, as desired.

   Thus, we've shown both sides of the if and only if, and so we can conclude that CH(S) and CH(T) are disjoint if and only if there is some line that separates S and T.

[1] In the case that one set is open and the other is closed, we can approximate the open set with a closed one.

2. **Design an $O(n \log n)$-time algorithm to find the convex hull of n unit circles. Note that you will need, as part of your solution, to devise a reasonable way to represent this hull in data since it will not be a polygon.**

Given n unit circles, we can easily compute the centers of each of these circles. In the following algorithm, we will assume that we have an array $C = [C_1, C_2, \ldots, C_n]$ where $C_1, C_2, \ldots, C_n$ represent the centers of the unit circles. This algorithm will return two values: B, the "vertices" belonging to the convex hull of the unit circles, and A, a list of objects holding arc length data. Each entry of A will hold two points in B and the arc length between these two points.

Algorithm ConvexHullUnitCircles(C)

1: Sort the centers of C from left to right (by their x-coordinate)
2: Set $P \coloneqq$ an array of points outputted by a function, GrahamScan, that produces a convex hull of its input, C
3: Set $B \coloneqq$ a $|P|$-length empty array, $S \coloneqq$ a P-length empty array
4: **for each** i in $\{1, 2, \ldots, |P|\}$ **do** *// To find CH vertices*
5: Set $L \coloneqq$ the line segment $\overline{P_i P_{(i+1)\,\%\,|P|}}$
6: Set $L_{inv} \coloneqq$ the line segment connecting P_i and the point found extending the endpoint of a 90° CCW rotation of L by 1 unit
7: Set $b_i \coloneqq$ the point of intersection between C_i and L_{inv}
8: Set $b_{(i+1)\%|P|} \coloneqq$ be the point of intersection between a line passing through b_i of length L parallel to $\overline{P_i P_{(i+1)\,\%\,|P|}}$ and C_{i+1}
9: Set $B[i] = b_i$ and $B[(i+1)\,\%\,|P|\,] = b_{(i+1)\,\%\,|P|}$
10: **end for**
11: **for** $i = \{1, 2, \ldots, |P|\}$ **do** *// To find CH arc lengths*
12: **if** $i = 1$ **then**
13: Set $s \coloneqq$ 180 - (the angle between $\overline{P_1 P_2}$ and $\overline{P_1 P_{|P|}}$)
14: Add arc-length object with points $b_1, b_{|P|}$ and arc length s to A
15: **else**
16: Set $s \coloneqq$ 180 - (the angle between $\overline{P_i P_{(i+1)\,\%\,|P|}}$ and $\overline{P_i P_{i-1}}$)
17: Add arc-length object with points b_i, b_{i+1} and arc length s to A
18: **end if**
19: **end for**
20: Return A, B

Runtime Analysis: Lines 1-3 are initialization steps. Sorting the centers of the unit circles takes $O(n \log n)$ time, which is performed once. Graham Scan takes $O(n)$ time after sorting.

To find the CH vertices (lines 4-10), each unit circle in the convex hull (at most, n) must be evaluated. It takes constant time, $O(1)$, to find the line segment connecting the centers of circle P_i and circle $P_{(i+1)\%|P|}$ (line 5) and the line segment with inverse slope in the CCW direction and the circle C_i (line 6). Finding points b_i and $b_{(i+1)\%|P|}$ (lines 7-8) also takes constant time as evaluating intersections between lines and circles is $O(1)$. Adding points b_i and $b_{(i+1)\%|P|}$ to their respective positions in B (line 9) also takes constant time. Therefore,

each of the operations in the (at most) n-length for loop is constant time, and the run-time of finding the vertices of the CH is $O(n)$.

To find the arc lengths of the CH (lines 11 - 19), the unit circles (at most, n) that make up the convex hull are evaluated again. Finding the angles between the line segments and creating/storing the arc-length object both take $O(1)$ time. Therefore, each of the operations in the (at most) n-length for loop is constant time, and the run-time of finding the arc-lengths of the CH is $O(n)$.

The sorting step takes $O(n \log n)$ time. Both finding the vertices and arc lengths take $O(n)$ time. Thus, the overall runtime of this algorithm is

$$O(n \log n) + O(n) + O(n) = O(n \log n)$$

as desired.

Algorithm Explanation: Our algorithm for finding the convex hull of n unit circles essentially amounts to finding the convex hull of the centers of the n unit circles, extending a 1 unit boundary around the convex hull of these centers, and "rubber-banding" the corners (corresponding to arcs of circles). To distinguish between vertices and the arc lengths of our convex hull, we split our work into two steps: the first occurs in lines 4 to 10 and the second occurs in lines 11 to 18.

In lines 4 to 10, we use L and use a ninety-degree counterclockwise rotation (we extend by an extra unit in case L is less than 1 unit which can occur if circles are overlapping) to find L_{inv}. The intersection of each L_{inv} and C_i gives us the vertices in our convex hull boundary. In lines 11 to 18, we compute the angles for each arc that connects pairs of consecutive convex hull vertices.

Proof of Correctness: Let S denote our original set of circles and let S' be the set of points that are the centers of the circles in S. We will show that a circle in S appears on the boundary of our final convex hull if and only if the center of the circle lies on the convex hull of S'.

The reverse direction of this statement follows nicely from the algorithm's construction – if the center of a circle lies on $\text{CH}(S')$, then it must lie on the boundary of our final convex hull. This holds due to lines 4 to 10 and lines 11 to 18 of our algorithm, which extend the boundary of $\text{CH}(S')$ and "wrap" the corners of this extended boundary to fit the arcs of the circles, giving us the convex hull of the circles themselves. The forward direction follows similarly – any arc of a circle on the boundary of our final convex hull must have originated from a 1 unit extension from the boundary of $\text{CH}(S')$, meaning that the center of that circle must lie on the boundary of $\text{CH}(S')$.

It remains to show that the shape of our convex hull is optimal – namely, that the boundary of the convex hull of these unit circles consists of straight line segments and arcs of circles in S. We argue that straight lines must be included since the convex hull must, by definition,

include line segments that are tangent to consecutive circles on the boundary. Finally, the boundary of the convex hull will include arcs to maintain the convexity of the hull (if the boundary edges were wrapped by lines, then there is some segment of points in a unit circle that is not contained in the convex hull).

3. **Given a set S of n points in the plane, let's say the boundary of CH(S) is the first *layer* of S, and for each $i > 1$, the i^{th} layer is the boundary of the convex hull of S with the first $i-1$ layers removed.**

   The *peeling depth* of a point $p \in S$ is the number of layers that lie outside of it (i.e. p's peeling depth is $i-1$ if it is in the i^{th} layer. Design an $O(n^2)$-time algorithm for finding the peeling depth of any given point $p \in S$.

   In this algorithm, we assume that we have an array pts $= [p_1, p_2, \ldots, p_n]$ where $p_1, \ldots, p_n$ represent the n points of S in their given order. We want to find the peeling depth of any point $p \in S$.

Algorithm FindPeelingDepth(pts)

```
1: Sort the points in array pts from left to right (by their x-coordinate.)
2: Set visited := an n + 1-length array storing the peeling depth of each point in the
       sorted array. All entries are initialized to 0.
3: Set counter := 0, ℓ := 1
4: while counter < n do
5:     Create US, LS := empty stacks which will contain the upper and lower hulls of
           layer ℓ
6:     Call the subroutine of Graham's Scan to find the upper hull of layer ℓ. At each
           step, only consider points pts[i] where visited[i] == 0, and store selected
           points in US by their sorted index.
7:     Call the subroutine of Graham's Scan to find the lower hull of layer ℓ. At each
           step, only consider points pts[i] where visited[i] == 0, and store selected
           points in LS by their sorted index.
8:     while US is non-empty do
9:         Set visited[US.pop()] == ℓ
10:        Increment counter
11:    end while
12:    while LS is non-empty do
13:        Set visited[LS.pop()] == ℓ
14:        Increment counter
15:    end while
16: end while
```

Runtime Analysis: In lines 1 to 3, we perform some initialization steps. The sorting of the elements in the points array takes $O(n \log n)$ time, and we perform this operation just once.

The main steps occur in lines 6 and 7; each subroutine mentioned takes $O(n)$ time, since each point is pushed and popped at most twice when finding the upper and lower hulls. Since we are only considering points not already found in previous layers, this entire checking and finding process takes at most $O(n)$ time. In lines 8 to 11 and 12 to 15, we simply loop through the stack, popping points and storing the current layer, corresponding to these points' peeling depth. Since each stack contains at most n points, these steps take $O(n)$ time. Consequently, each iteration of the outer while loop in line 4 runs in $O(n)$ time.

Finally, note that there are at most $O(n)$ iterations of the while loop in line 4. Each subroutine of Graham's Scan finds the upper and lower hulls corresponding to layer i; these hulls contain at least one point each. Equivalently, each iteration finds a layer of S, which contains at least one point. Thus, we require $O(\frac{n}{1}) = O(n)$ iterations to find the layer of all n points.

Our sorting step takes $O(n \log n)$ time. Then, we perform $O(n)$ iterations of the loop in line 4, each of which run in $O(n)$ time. Thus, our overall runtime is

$$O(n \log n) + O(n) \cdot O(n) = O(n^2)$$

as desired.

Proof of Correctness: The correctness of our algorithm is essentially given to us from the correctness of Graham's Scan. We know that Graham's Scan finds the upper and lower hull of a set of points. We add the extra condition to only check points that have not already been assigned to a layer; this enforces the fact that once we assign a point to a layer, we will never consider it again and thus, it cannot be erroneously assigned to another layer. After the first subroutine on line 6 completes, we have the upper hull of layer ℓ, and after the first subroutine on line 7, we have the lower hull of layer ℓ. All points in the union of the upper and lower hull are correspondingly assigned to layer ℓ, and this assignment is reflected in line 13.

To summarize, for each iteration of the while loop on line 4, we find the boundary of the remaining non-assigned points in set S using Graham's Scan and then assign the points on the boundary to their corresponding layer. We continue until we have assigned each point in S their peeling depth. Consequently, we can find the peeling depth associated to any point $p \in S$. Since Graham's Scan is correct, so is FindPeelingDepth.

4. **In real-world applications, the set you need to find the convex hull of is often *dynamic*, changing over time. In these situations, you want to be able to handle small changes in the input without having to start over from scratch.**

   To be concrete, suppose you receive n points $p_1, \ldots, p_n \in \mathbb{R}^2$ one at a time, and after each point p_i arrives, you need to return CH$(\{p_1, \ldots, p_i\})$ (as a clockwise sequence of vertices, as usual). Design an algorithm that outputs all n of these convex hulls in $O(n^2)$ time total.

   We assume we are given an array, P, which includes $[p_1, p_2, \ldots p_n]$, the points to generate n convex hulls for.

Algorithm DynamicConvexHull(P)

```
1: Set T := an empty AVL Tree
2: Set Ret := an empty array of size n
3: for each p in P do
4:     Add p to T (where T is arranged by the order of the x-coordinates)
5:     Call arr the n-length array output of an in-order traversal of T
6:     Set CH := an array of points outputted by a function, GrahamScan, that produces
           a convex hull of its input, arr
7:     Add CH to Ret
8: end for
9: Return Ret
```

Runtime Analysis: Lines 1 and 2 are initialization steps for the data structures used for this algorithm.

Lines 3-8 are the bulk of the algorithm. We will need to evaluate each point, p_i, of n total points. Therefore, to keep the algorithm $O(n^2)$, each step must be at most $O(n)$. Adding each point to T, an AVL tree, takes $O(\log n)$. An in-order traversal of AVL tree's takes $O(c)$ time where c represents the tree's size. Because the AVL tree will only hold up to n points, the worst time for generating a sorted array of T will be $O(n)$. Graham Scan takes $O(n)$ time on sorted array arr. Adding a value to an array of known length takes $O(1)$, which is the last step of the algorithm.

Therefore, the total runtime of this algorithm is

$$O(n) \cdot (O(\log(n)) + O(n) + O(n) + O(1)) = O(n^2)$$

Proof of Correctness: DynamicConvexHull works because Graham Scan works. For each point added to T, Graham Scan is used to produce the new convex hull which is added to Ret. As long as the input being fed into GrahamScan is a correctly sorted array of the points up to p_i, the function will return CH$(\{p_1, \ldots, p_i\})$. Therefore, because the in-order traversal of T always yields the correct sorted order, DynamicConvexHull works.