CSC282 HW2

Hanzhang Yin

Sep/16/2023

Question 6

```
# Helper Function
function CCW(p1, p2, p3):
    #True if counter-clockwise turn
    return (p2.x - p1.x) * (p3.y - p1.y) - (p2.y - p1.y) * (p3.x - p1.x) > 0

function sortPointsLexicographically(points):
    # Sort the points by x-coordinate, breaking ties with y-coordinate
    # Time-complexity O(n log n)
    return sorted(points, key: (point.x, point.y))

# Main Function
function constructSimplePolygon(points):
    # Sort points lexicographically
    sortedPoints = sortPointsLexicographically(points)

    # Build the upper chain
    upperChain = []
    for point in sortedPoints:
        upperChain.append(point)
        while len(upperChain) >= 3 and not CCW_left(upperChain[len(upperChain) - 3],
                                                     upperChain[len(upperChain) - 2],
                                                     upperChain[len(upperChain) - 1]):
            # Remove the middle point to maintain the CCW property
            upperChain.pop(len(upperChain) - 2)

    # Build the lower chain
    lowerChain = []
    for point in reverse(sortedPoints):
        lowerChain.append(point)
        while len(lowerChain) >= 3 and not CCW_left(lowerChain[len(lowerChain) - 3],
                                                     lowerChain[len(lowerChain) - 2],
                                                     lowerChain[len(lowerChain) - 1]):
```

```
        # Remove the middle point to maintain the CCW property
        lowerChain.pop(len(lowerChain) - 2)

# Combine the chains to form the simple polygon
upperChain.pop(len(upperChain) - 1)
lowerChain.pop(len(lowerChain) - 1)
polygon = upperChain + lowerChain

# Return the polygon containing all points
return polygon
```

Question 7

```
function findVisibleSegments(segments, p):

    # Collect all events (segment endpoints) and compute their angles from point p
    # Time Complexity: O(n)
    events = []
    for segment in segments:
        angleStart = computeAngle(p, segment.start)
        angleEnd = computeAngle(p, segment.end)

        # Normalize angles to [0, 2pi)
        angleStart = normalizeAngle(angleStart)
        angleEnd = normalizeAngle(angleEnd)

        if angleStart <= angleEnd:
            events.append({'angle': angleStart, 'segment': segment,
                type': 'start'})
            events.append({'angle': angleEnd, 'segment': segment,
                    'type': 'end'})
        else:
            # Segment crosses the 0 angle
            events.append({'angle': angleStart, 'segment': segment,
                'type': 'start'})
            events.append({'angle': angleEnd + 2 * PI, 'segment': segment,
                'type': 'end'})

    # Sort all events by angle
    # Time Complexity: O(n log n)
    sortedEvents = sortByAngle(events)

    # The BST is ordered by distance from p along the current angle
    activeSegments = emptyBST()

    visibleSegments = emptySet()

    # Sweep through all sorted events to determine visibility
    # Time Complexity: O(n log n)
    for event in sortedEvents:
        angle = event['angle'] % (2 * PI)  # Normalize angle within [0, 2pi)
        segment = event['segment']

        if event['type'] == 'start':
            # Compute the distance from p to the segment along the current angle
            distance = computeDistanceToSegment(p, angle, segment)
```

```
        # Insert the segment into the activeSegments BST with the computed distance
        insertSegment(activeSegments, segment, distance)

        # Check if this segment is the closest one currently
        closestSegment = activeSegments.findMin()
        if closestSegment == segment:
            # If it's the closest, add to visibleSegments
            visibleSegments.add(segment)
    elif event['type'] == 'end':
        # Remove the segment from the activeSegments BST
        removeSegment(activeSegments, segment)

        # After removal, check the new closest segment
        closestSegment = activeSegments.findMin()
        if closestSegment is not None:
            visibleSegments.add(closestSegment)

return visibleSegments
```

Question 8

The following code addressed the problem: counting intersections of axis-aligned segments in $O(nlogn)$ time. Note that the codec is implemented in *Python*.
The algorithm uses a sweep-line technique combined with a Binary Indexed Tree (BIT) to achieve this time complexity. Segments are represented with endpoints and classified as horizontal or vertical; y-coordinates are compressed into integer indices for efficient BIT operations. Events are created: horizontal segments generate add and remove events at their x-endpoints, while vertical segments generate query events at their x-coordinate. Events are sorted by x-coordinate, processing adds before queries and queries before removes when x-values are equal. As the sweep line advances, the BIT maintains the active set of horizontal segments' y-indices. During query events, the BIT efficiently counts active horizontal segments overlapping the vertical segment's y-range. Noticing that BIT costs $O(\log n)$ time updates and queries. Hence, the sweep line framework achieves the $O(n \log n)$ time complexity.

```
# Axis-Aligned Segment Intersection Counting Algorithm
# Time Complexity: O(n log n)

class Segment:
    def __init__(self, x1, y1, x2, y2):
        # Ensure (x1, y1) is the lower-left point and (x2, y2) is the upper-right point
        if x1 > x2 or y1 > y2:
            x1, x2 = min(x1, x2), max(x1, x2)
            y1, y2 = min(y1, y2), max(y1, y2)
        self.x1, self.y1 = x1, y1
        self.x2, self.y2 = x2, y2
        # Determine if the segment is horizontal or vertical
        self.is_horizontal = y1 == y2
        self.is_vertical = x1 == x2

def coordinate_compress(coordinates):
    unique_coords = sorted(set(coordinates))
    coord_dict = {coord: idx for idx, coord in enumerate(unique_coords)}
    return coord_dict

class BIT:
    def __init__(self, size):
        self.size = size + 2  # +2 to avoid indexing error
        self.tree = [0] * self.size

    def update(self, idx, val):
        idx += 1  # BIT uses 1-based indexing
        while idx < self.size:
            self.tree[idx] += val
```

```
            idx += idx & -idx

    def query(self, idx):
        idx += 1
        result = 0
        while idx > 0:
            result += self.tree[idx]
            idx -= idx & -idx
        return result

    def range_query(self, l, r):
        return self.query(r) - self.query(l - 1)

def count_intersections(segments):
    horizontal_segments = []
    vertical_segments = []
    y_coords = []

    for seg in segments:
        if seg.is_horizontal:
            horizontal_segments.append(seg)
            y_coords.append(seg.y1)
        elif seg.is_vertical:
            vertical_segments.append(seg)
            y_coords.append(seg.y1)
            y_coords.append(seg.y2)

    # Coordinate compression for y-coordinates
    y_coord_map = coordinate_compress(y_coords)
    max_y_idx = len(y_coord_map)

    events = []

    # Create add and remove events for horizontal segments
    for seg in horizontal_segments:
        y_idx = y_coord_map[seg.y1]
        events.append((seg.x1, 0, y_idx))  # Add event
        events.append((seg.x2, 2, y_idx))  # Remove event

    # Create query events for vertical segments
    for seg in vertical_segments:
        y1_idx = y_coord_map[seg.y1]
        y2_idx = y_coord_map[seg.y2]
        events.append((seg.x1, 1, min(y1_idx, y2_idx), max(y1_idx, y2_idx)))

    # Sort events by x-coordinate and event type
```

```
events.sort(key=lambda x: (x[0], x[1]))

bit = BIT(max_y_idx)
intersection_count = 0

for event in events:
    if event[1] == 0:
        # Add event: add horizontal segment's y-coordinate to BIT
        y_idx = event[2]
        bit.update(y_idx, 1)
    elif event[1] == 2:
        # Remove event: remove horizontal segment's y-coordinate from BIT
        y_idx = event[2]
        bit.update(y_idx, -1)
    else:
        # Query event: count overlapping horizontal segments
        y1_idx, y2_idx = event[2], event[3]
        count = bit.range_query(y1_idx, y2_idx)
        intersection_count += count

return intersection_count
```

Question 9

Picked Probelm: **Simple Linear-Time Polygon Triangulation**
(https://topp.openproblem.net/p10AGR00)