CSC282 HW2

Hanzhang Yin

Sep/22/2023

Collaborator

Chenxi Xu, Yekai Pan, Yiling Zou, Boyi Zhang

Question 6

```
# Helper Functions
def CCW(p1, p2, p3):
    return (p2.x - p1.x)*(p3.y - p1.y) - (p2.y - p1.y)*(p3.x - p1.x) > 0

# Main Function (Graham's Algorithm)
def constructSimplePolygon(points):
    # Find the point with the lowest y-coordinate, break ties with x-coordinate
    # Time-complexity O(n)
    p0 = min(points, key = (p.y, p.x))

    points_except_p0 = [p for p in points if p != p0]

    # Sort the points by CCW Order with p0
    # Time-complexity O(n log n)
    sortedPoints = sortPointsByCCW(points_except_p0, p0)

    # Initialize the convex hull with point p0
    hull = [p0]

    # Time-complexity O(n)
    for point in sortedPoints:
        hull.append(point)
        while len(hull) >= 3 and !CCW(hull[len(hull) - 3],
            hull[len(hull) - 2], hull[len(hull) - 1]):
            # Remove the middle point to maintain the CCW property
            hull.pop(len(hull) - 2)

    return hull
```

Question 7

```
def findVisibleSegments(segments, p):
    An "event" defined to be "start" and "end" points of a segments
    # Applying MergeSort Or Similar Approach.
    # Time Complexity: O(nlogn)
    Events are sorted based on the angle from the point p to the start or end point

    We start at angle 0, and we split all segments at angle 0 into two segments
    # Avoid Duplicate Counting
    But, we increase the count only once for a segment if both segment-pieces are counted

    Define "Active" as a balanced binary tree using a "dynamic comparator"
    Define "VisibleSeg" as a set stored visible segement candidate.
    Given an angle a, compare the segments based on the "distance" from p at angle a

    While events are non-empty:
        a = angle of current event
        if current event is a start point:
            # Comparison can be made in O(logn) time by "dynamic comparator"
            if the segement is less than all other elements in Active:
                count += 1
                add segement to VisibleSeg
            add segement to Active
        else:
            Remove segement from Active
            if the smallest element in Active is not already counted yet:
                count += 1
                add segement to VisibleSeg
    return VisibleSeg, count
```

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

# Define Binary Indexed Tree
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/p10

Problem Description:

Polygon triangulation involves partitioning a simple polygon into non-overlapping triangles. The primary challenge lies in achieving efficient algorithms that perform this triangulation in linear time, especially for large and complex polygons.

Approaches and Recent Developments:

Chazelle's [1] groundbreaking 1991 algorithm was the first to achieve deterministic linear-time triangulation of a simple polygon. However, its complexity has driven researchers to seek simpler yet efficient alternatives. Recent (But not that recent) advancements include both deterministic and randomized algorithms that strive to match or approach linear-time performance with greater simplicity:

1. **Deterministic Linear-Time Algorithm:** A new approach leverages the polygon-cutting and planar separator theorems to build a coarse triangulation in a bottom-up phase, followed by a top-down refinement [2]. This method avoids complex data structures like dynamic search trees, relying instead on elementary structures, thus simplifying implementation compared to Chazelle's algorithm.

2. **Randomized Algorithms:** One algorithm computes the trapezoidal decomposition of a simple polygon in expected linear time, enabling linear-time triangulation through known reductions. It simplifies Chazelle's method by performing random sampling [3] on subchains of the polygon rather than its edges.

Possible Thoughts For Approaching (Topological Approach):

To develop a simple linear-time algorithm for polygon triangulation, we can use the ear clipping method. [4] First, decompose the polygon into monotone pieces using an $O(n)$ algorithm (there exists a few currently). Then, apply a simplified ear clipping method to each monotone piece. An "ear" is a triangle formed by three consecutive vertices that lies entirely inside the polygon without containing any other vertices. Since ear clipping operates efficiently on monotone polygons, this approach achieves an overall linear-time ($O(n) + O(n) \sim O(2n)$) algorithm and might be considered running in a simpler manner by avoiding complex data structures. By clipping off ears sequentially, the polygon is completely and efficiently triangulated.

References

[1] Chazelle, Bernard. 1991. "Triangulating a Simple Polygon in Linear Time." *Discrete & Computational Geometry*, 6(3): 485-524. https://doi.org/10.1007/BF02574703.

[2] Amato, Nancy M., Michael T. Goodrich, and Edgar A. Ramos. 2000. "Linear-Time Triangulation of a Simple Polygon Made Easier via Randomization." In *Proceedings of the 16th Annual Symposium on Computational Geometry*, 201–212. https://doi.org/10.1145/336154.336206.

[3] Seidel, Raimund. 1991. "A Simple and Fast Incremental Randomized Algorithm for Computing Trapezoidal Decompositions and for Triangulating Polygons." *Computational Geometry*, 1(1): 51–64. https://doi.org/10.1016/0925-7721(91)90012-4.

[4] Garey, M. R., Johnson, D. S., Preparata, F. P., Tarjan, R. E. (1978). Triangulating a Simple Polygon. Information Processing Letters, 7(4), 175-179