CS 478 Computational Geometry
Homework 4

Barkın Saday
Bilkent University
21902967

Question 1

Overview:
We walk along both polygon boundaries in order (I assumed input vertices are ordered, counter clockwise) and use convexity to find and follow intersection points in linear time.

Given:

- Both input polygons are convex and represented as ordered lists of vertices in counter-clockwise (CCW) order. **(a simple assumption on input)**
- The input size N refers to the total number of vertices in both polygons.
- Basic geometric operations (segment intersection, orientation tests, and point-in-polygon test for convex polygons) are considered constant time.

Algorithm:

We can use some sort of *simultaneous edge-walking* method (a known technique) for convex polygon intersection (see [1], [2]). The idea is to traverse the edges of both polygons in order, maintaining the current edges being considered from each polygon. At each step:

- We check whether the current edges intersect. If so, the intersection point is added to the set J.
- Using the orientation of the edge directions, we decide which polygon's edge to advance.
- In parallel, we track which polygon is currently "inside" the other, and collect vertices accordingly to construct the output boundary.

This process completes in linear time since each edge is visited once and intersection checks are constant time.

Pseudo-code:

```
POLYGON_INTERSECTION(P1, P2):
    i, j ← 0, 0
```

```
J ←
result ←
e1 ← edge(P1[i], P1[i+1])
e2 ← edge(P2[j], P2[j+1])

repeat until all edges visited:
    if e1 and e2 intersect:
        J ← J  {intersection point}
        result ← result  {intersection point}

    if e2 is to the left of e1 (by orientation):
        if P2[j+1] inside P1:
            result ← result  {P2[j+1]}
        j ← j + 1
        e2 ← next edge of P2
    else:
        if P1[i+1] inside P2:
            result ← result  {P1[i+1]}
        i ← i + 1
        e1 ← next edge of P1

return result (in CCW order)
```

Correctness:

- Each edge from both polygons is processed exactly once.
- The convexity of input polygons ensures that the intersection is also convex, so the algorithm's boundary walk constructs a valid result.
- Since we only insert intersection points or vertices that lie inside the other polygon, no extraneous points are added.
- The algorithm handles disjoint, overlapping, and fully-contained scenarios uniformly.

Time Complexity:
The algorithm visits each of the N edges exactly once, and each primitive operation (orientation check, intersection test, inclusion check) is constant time. Thus, the total complexity is:

$$\boxed{O(N)}$$

Question 2

Overview:
We try to satisfy a series on inequalities. Each inequality represents the range where a vertical input segment and the solution line can intersect. If we can satisfy all the inequalities together, there exist at least one solution line (can be multiple).

Assumptions:

- Each vertical segment is defined by a fixed x_i and spans from $y_i^{\min}$ to $y_i^{\max}$.
- There are no two segments with the same x_i value (we can perturb them infinitesimally if needed).
- Basic geometric operations (computing slopes, comparisons) are constant time.

Idea:

A line $y = ax+b$ intersects a vertical segment located at x_i with vertical extent $[y_i^{\min}, y_i^{\max}]$ if and only if:

$$y_i^{\min} \leq ax_i + b \leq y_i^{\max}$$

This implies that any candidate line must satisfy the system of inequalities:

$$y_i^{\min} - ax_i \leq b \leq y_i^{\max} - ax_i, \quad \text{for all } i = 1, \ldots, n$$

For a fixed slope a, this defines an interval of valid b-values for each segment. The intersection of all such intervals must be non-empty for a valid line to exist.

Efficient Algorithm (Linear-Time):

We observe that we do not need to test all possible slopes. Instead, we can derive bounds on the slope by comparing all segment pairs.

- For each pair of segments (i, j), define two slope constraints:
  - $s_{ij}^{\min}$: The slope of the line from the top of segment i to the bottom of segment j:
    $$s_{ij}^{\min} = \frac{y_j^{\min} - y_i^{\max}}{x_j - x_i}$$
  - $s_{ij}^{\max}$: The slope of the line from the bottom of segment i to the top of segment j:
    $$s_{ij}^{\max} = \frac{y_j^{\max} - y_i^{\min}}{x_j - x_i}$$
- These slopes define an allowable range $[s_{ij}^{\min}, s_{ij}^{\max}]$ for a line to stab both segments i and j.
- Compute the global intersection of all such slope intervals:
  $$a_{\min} = \max_{i<j} s_{ij}^{\min}, \quad a_{\max} = \min_{i<j} s_{ij}^{\max}$$
- If $a_{\min} \leq a_{\max}$, then a feasible slope a exists for which a line intersects all segments.

Optimized Linear-Time Variant:

We fix a reference segment (e.g., the one with the smallest x_i), and compute slope intervals only with respect to that segment. Since we process each of the remaining $n-1$ segments once, we achieve:

$$\boxed{O(n)} \quad \text{time complexity}$$

Correctness:

This method correctly identifies whether a feasible slope a exists such that a line with that slope can pass through the vertical extent of every segment. The reduction to bounding slope intervals guarantees correctness because it captures the essential geometric condition of intersecting all segments. The existence of a line that intersects a set of vertical segments can be determined in linear time using slope bounds, as described in classical computational geometry literature (see [3]). While the explanation and formulation here are written from first principles, the result aligns with known algorithms and supports the correctness and optimality of the approach.

The referenced material supports the correctness and optimality of the approach, although the algorithm here is explained from first principles.

Question 3

Overview:
We can construct a **segment tree** for the **projections** of the input segments on the **x-axis**.

Given:

- Segments in S and the query segment s^* can be arbitrary (not necessarily axis-aligned).
- **Assumed:** Segment-segment intersection tests can be performed in constant time (f.e using line equations).

Idea:

We preprocess the set S using a geometric data structure that allows fast intersection testing with arbitrary query segments. One of the most effective methods is to use a **segment tree** constructed on the *projections* of the segments onto the x-axis ([3]).

This allows us to efficiently filter potential intersecting candidates by their horizontal extent and then apply exact intersection tests only to those candidates.

Algorithm:

Preprocessing Phase:

- Project each segment in S onto the x-axis, producing an interval $[x_{\min}, x_{\max}]$.
- Build a balanced segment tree on these intervals.

- For each node in the tree, store a list of segments that completely span that node's interval.

Query Phase (Given s^*):

- Project s^* onto the x-axis to obtain its interval.
- Traverse the segment tree to find all nodes whose intervals overlap with that of s^*.
- For each candidate segment stored in those nodes:
  - Perform an exact intersection test with s^*.
  - If any intersection is found, return `true`.
- If no intersection is found, return `false`.

Pseudocode:

```
PREPROCESS(S):
    for each segment s in S:
        project s onto x-axis
    build segment tree T on all intervals
    insert segments into corresponding tree nodes

QUERY(s*):
    project s* onto x-axis
    for each overlapping node in T:
        for each segment t in node:
            if s* intersects t:
                return true
    return false
```

Time and Space Complexity:

- Preprocessing time: $O(N \log N)$
- Space: $O(N \log N)$
- Query time (if there exist an intersection): $O(\log N)$

Correctness:

The segment tree organizes the segments in a way that ensures any query segment will only be tested against segments whose horizontal projections overlap with its own. Since this bounds the number of candidates and segment intersection is exact, the method guarantees correctness.

This solution I described is a standard technique in geometric range searching and spatial indexing (see [3]), particularly in the context of segment trees and intersection queries. While our explanation is given from first principles, the algorithm aligns with known solutions in the computational geometry literature.

Question 4

Overview:
We can sweep a vertical line from left to right. Then, we can sum up based on start and end events.

Idea:

We can sweep a vertical line from left to right across the plane and compute the area covered by the rectangles between each pair of consecutive vertical events. The total area is accumulated by summing contributions from vertical strips.

For each rectangle, and event can be thought as:

- A **start event** at its left edge.
- An **end event** at its right edge.

At each event position, we maintain the current set of active rectangles intersecting the sweep line and compute the total vertical length covered. This length, multiplied by the horizontal distance since the last event, gives the area contributed by the strip.

Event-Point Schedule:

We create an event list of all vertical edges of all rectangles. Each event is of the form:

$$(x, \text{type}, y_{\min}, y_{\max})$$

where `type` is either `START` or `END`, corresponding to the left or right edge of a rectangle. We sort the events by their x-coordinate in increasing order. This gives us the full **event-point schedule**.

Sweep-Line Status:

To track which parts of the y-axis are currently covered by rectangles, we use a data structure that supports:

- Inserting and deleting y-intervals $[y_{\min}, y_{\max}]$,
- Efficiently computing the total length of the union of all active y-intervals.

A suitable structure is a **segment tree** built on the set of unique y-coordinates, discretized and sorted. Each node maintains:

- A `count` of how many intervals cover this node's interval.
- A `length` field storing the total length covered if `count` ¿ 0.

As we process each event, we update the segment tree and query the total y-length covered, which is then used to compute the area contribution.

Algorithm (Pseudo):

1. Collect all events from rectangles and sort them by x-coordinate.

2. Initialize the segment tree on the y-interval endpoints.
3. Initialize `prev_x` and `total_area` to 0.
4. For each event $(x, \text{type}, y_1, y_2)$:
    - Let $\Delta x = x -$ `prev_x`.
    - Let ℓ be the total y-length currently covered (from the segment tree).
    - Update `total_area` $\mathrel{+}= \ell \cdot \Delta x$.
    - If `type` is `START`, insert $[y_1, y_2]$ into the segment tree.
    - If `type` is `END`, remove $[y_1, y_2]$ from the segment tree.
    - Set `prev_x` $= x$.

Complexity Analysis:

- Sorting the events: $O(n \log n)$
- Each event results in one segment tree update and one query: $O(\log n)$
- There are $2n$ events (two per rectangle), so total time is:

$$\boxed{O(n \log n)}$$

- Space complexity: $O(n)$ for the event list and segment tree.

Conclusion:

This algorithm computes the area of the union of axis-aligned rectangles efficiently using a vertical plane-sweep approach. It maintains the active vertical coverage using a balanced segment tree and accumulates contributions across vertical strips.

A detailed explanation of this algorithm and data structure can be found in standard computational geometry literature (for example, see [3]).

References

[1] F. P. Preparata and M. I. Shamos, *Computational Geometry: An Introduction.* Springer-Verlag, 1985.

[2] J. O'Rourke, *Computational Geometry in C.* Cambridge University Press, 1998.

[3] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars, *Computational Geometry: Algorithms and Applications*, 3rd Edition, Springer, 2008.