

Algorithms & Adv. Data Structures

Lab 3: Iterative Algorithm Design

Alhagie Boye

Sukhbir Singh

Vamsi Sudersanam

Introduction

In this lab, we will tackle the computational geometry problem of determining whether a given point lies inside a polygon. This problem has practical applications in various fields such as computer graphics, geographic information systems (GIS), and robotics

1. Problem: Determine if a Point is Located Inside a Polygon

Input:

- A sequence $\langle p_1, p_2, \dots, p_n \rangle$ of $n \geq 3$ 2D points. Each point is a pair of x and y coordinates. The points correspond to the vertices of a simple (non-intersecting) polygon. The polygon is connected by line segments between each adjacent pair of points, including a line segment from the last point to the first point.
- The x and y coordinates for a single point distinct from the vertex points.

Output: A Boolean value indicating whether the point is located inside the polygon.

2. Decision Rule and Illustrations

To determine whether a point (X,Y) lies inside a polygon, we employ the Ray Casting algorithm (also known as the Even-Odd Rule). The decision process follows these criteria (**See figure 1**):

- Cast a horizontal ray from the test point P to infinity (or a sufficiently large x-coordinate)
- Count the number of intersections (N) between this ray and the polygon's edges
- Apply the following decision rule:
 - If **N is odd or point lies exactly on a polygon edge** \rightarrow Point lies **INSIDE** the polygon.
 - If **N is even** \rightarrow Point lies **OUTSIDE** the polygon

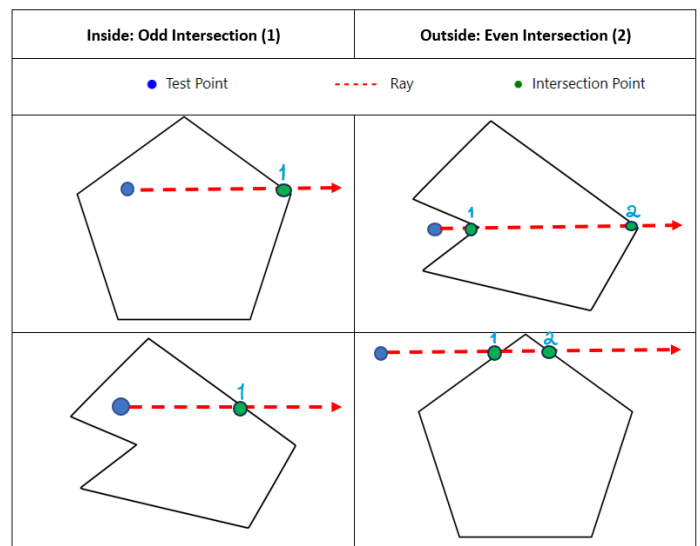


Fig 1: Ray Casting Algorithm Diagram

3. High-Level Pseudocode

```
Initialize crossing_count = 0

For each edge in polygon:
    If ray from point crosses edge:
        Increment crossing_count

    If point lies exactly on edge:
        Return true

If crossing_count is odd:
    Return true    // Point is inside
Else:
    Return false   // Point is outside
```

4. Justification of Correctness for Ray-Casting Algorithm – Sukhbir

1. How it works:

- The algorithm casts an imaginary ray from the point in question and counts how many times it crosses the polygon's edges.
 - Odd number of crossings means the point is inside the polygon.
 - Even number of crossings means the point is outside the polygon.

2. Why it works:

- **Crossing edges:** Each time the ray crosses an edge of the polygon, the point moves from inside to outside or vice versa. If it crosses an odd number of times, the point started outside and is now inside. If even, it either never entered or crossed back out.

3. Convex Vs Concave Polygons:

- **Convex Polygons:** A point inside a convex polygon will always result in an odd number of ray crossings. This is because convex polygons have no inward curves, and the ray crosses the boundary once to enter and once to exit.
- **Concave polygons:** In concave polygons (polygons with inward dents), the algorithm still works because the number of crossings determines whether the point is inside or outside. Even with the concave parts, an odd number of crossings still means inside, and an even number means outside.

4. Edge cases:

- If the point lies exactly on a polygon edge, it can either be treated as "inside" or handled separately depending on the requirements.

Overall: The Ray-Casting algorithm is proven correct by the geometric principles of polygons and lines. The even-odd rule, based on how many times a ray crosses the edges, is a key concept in computational geometry. It applies to any simple polygon and consistently determines if a point is inside or outside, making the algorithm both effective and commonly used in various scenarios.

5. Worst-Case Run Time Analysis - alhagie

The worst for this algorithm is **O(n)**

```
# Initialize crossing count
crossings = 0
test_point_x, test_point_y = point

for i_vertex in range(len(polygon_vertices)):
    segment_start_x, segment_start_y = polygon_vertices[i_vertex]

    # If we're at the last vertex, connect it to the first vertex
    if i_vertex < len(polygon_vertices) - 1:
        segment_end_x, segment_end_y = polygon_vertices[i_vertex + 1]
    else:
        segment_end_x, segment_end_y = polygon_vertices[0]

    # Check that the ray falls within the range of x values for the start and end of the segment
    if ((segment_start_x < test_point_x < segment_end_x) or
        (segment_start_x > test_point_x > segment_end_x)):

        # Calculate the intersection point in the y direction
        t = (test_point_x - segment_start_x) / (segment_end_x - segment_start_x)
        crossing_y = (t * segment_start_y) + ((1 - t) * segment_end_y)

        # Check if the ray crosses the segment above the point
        if crossing_y >= test_point_y:

        # Special case: check if the point lies exactly on a vertical segment
        if (segment_start_x == test_point_x and
            min(segment_start_y, segment_end_y) <= test_point_y <= max(segment_start_y, segment_end_y)):
            return True # Point is on the boundary

# If the number of crossings is odd, the point is inside; otherwise, it's outside
return crossings % 2 != 0
```

Cost	r
C1	1
C2	1
C3	$\sum_{i=1}^n (n-1)$
C4	1
C5	1
C6	1
C7	1
C8	1
C9	1
C10	1

$$\sum_{i=0}^n n - \sum_{i=0}^n 1 \Rightarrow T(n) \in O(n)$$

Time Complexity Analysis

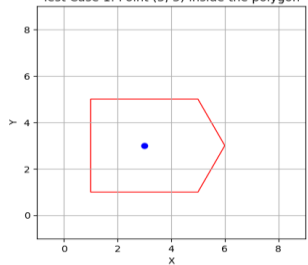
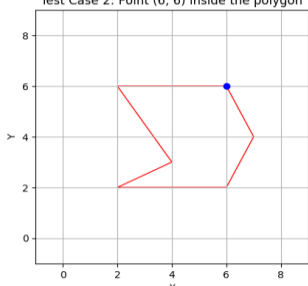
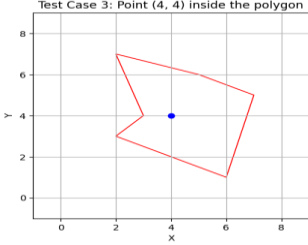
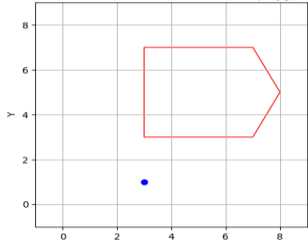
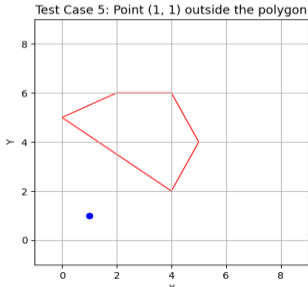
- Initialization:** Initializing the crossings variable and extracting the coordinates of the test point takes constant time, **O(1)**.
- Loop Through Polygon Vertices:** The main part of the algorithm is a loop that iterates through each vertex of the polygon. This loop runs (n) times, where (n) is the number of vertices in the polygon.
- Segment Processing:** For each vertex, the algorithm performs several operations:
 - Extracting the coordinates of the current and next vertex.
 - Checking if the ray falls within the range of x-values for the segment.
 - Calculating the intersection point in the y-direction.
 - Checking if the ray crosses the segment above the point.
 - Checking if the point lies exactly on a vertical segment.

Each of these operations takes constant time, **O(1)**.

- Final Check:** After the loop, the algorithm performs a final check to determine if the number of crossings is odd or even. This takes constant time, **O(1)**.

Worst-Case Time Complexity: Since the algorithm iterates through each vertex of the polygon exactly once and performs a constant amount of work for each vertex, the overall time complexity is **O(n)**, where (n) is the number of vertices in the polygon.

6. Test Cases and Expected vs. Actual Results - Alhagie

Polygon	Point to Predict	Expected Result	Actual Event
<p>Test Case 1: Point (3, 3) inside the polygon</p> 	3,3	True	True
<p>Test Case 2: Point (6, 6) inside the polygon</p> 	6,6	True	True
<p>Test Case 3: Point (4, 4) inside the polygon</p> 	4,4	True	True
<p>Test Case 4: Point (3, 1) outside the polygon</p> 	3,1	False	False
<p>Test Case 5: Point (1, 1) outside the polygon</p> 	1,1	False	False

7. Appendix: Source Code and Test Cases

Github: [boyeaalhagie/Iterative-Algorithm](https://github.com/boyeaalhagie/Iterative-Algorithm)

Test Cases:

```
test_cases = [  
    ([ (1, 1), (5, 1), (6, 3), (5, 5), (1, 5)], (3, 3), True),  
    ([ (2, 2), (6, 2), (7, 4), (6, 6), (2, 6), (4, 3)], (6, 6), True),  
    ([ (2, 3), (6, 1), (7, 5), (5, 6), (2, 7), (3, 4)], (4, 4), True),  
    ([ (3, 3), (7, 3), (8, 5), (7, 7), (3, 7)], (3, 1), False),  
    ([ (0, 5), (4, 2), (5, 4), (4, 6), (2, 6)], (1, 1), False)  
]
```