CS 478 - HW 2

Barkın Saday

21902967

1 Checking if a DCEL is a Triangulation

DCEL Representation (as we use in the lectures)

The DCEL (Doubly Connected Edge List) data structure represents a PSLG (Planar Straight-Line Graph) with the following fields for each edge:

- **V1**: Origin of the edge.
- **V2**: Terminus (destination) of the edge; defines orientation.
- **F1**: Face to the left of the edge (relative to V1V2 orientation).
- **F2**: Face to the right of the edge (relative to V1V2 orientation).
- **P1**: Index of the first edge encountered after V1V2, when moving counterclockwise around V1.
- **P2**: Index of the first edge encountered after V1V2, when moving counterclockwise around V2.

Algorithm

1. **Iterate over all faces** in the DCEL.
2. **Skip the outer face** (assuming it is known or can be identified) (if not known, it can be found in O(n) as we did something similar in HW 1).
3. **For each bounded face**:
   (a) Start at any edge associated with the face.
   (b) Traverse the face boundary using the **P1** or **P2** pointers.
   (c) Count the number of edges forming the face.
   (d) If any face has **more than 3 edges**, return **False**.
4. If all bounded faces have exactly 3 edges, return **True**.

Time Complexity Analysis

- Each face traversal takes $O(1)$ time (since each face has at most 3 edges).
- We traverse each face **once**, so the total complexity is $O(F)$.
- Using Euler's formula, in a triangulated subdivision, $F = O(V)$, so the final complexity is $O(V)$.

Edge Cases Considered

- **DCEL with no faces** ⇒ Return False.
- **Outer face has more than 3 edges** ⇒ Ignore it. (Since problem states that way)
- **A face with fewer than 3 edges** ⇒ Invalid DCEL structure.

Conclusion

The algorithm efficiently determines whether a given DCEL represents a triangulated planar subdivision in $O(V)$ time. By iterating over faces and verifying their edge count, we ensure correctness while maintaining optimal performance.

2 Maintaining a Dynamic Convex Hull

Adding a Point to the Convex Hull

To insert a new point P into the convex hull CH, we handle two cases:

Case 1: P is Inside or on the Boundary of CH - If P lies inside or on the convex hull, no update is needed. - We can check this in $O(\log H)$ time using a binary search (if the hull is stored in sorted order by angle or coordinates).

Case 2: P is Outside the Convex Hull - Find the **leftmost and rightmost tangent** edges from P to CH. - This identifies the portion of the hull that will be replaced. - Can be done in $O(\log H)$ time using binary search. - Remove all convex hull points between these two tangents. - Insert P and connect it to the tangent points.

Time Complexity for Insertion

- Checking if P is inside CH: $O(\log H)$
- Finding tangents: $O(\log H)$
- Removing K points: $O(K)$
- Inserting P: $O(1)$
- **Worst case**: $O(H)$ (if a large portion of the hull is replaced)

Removing a Point from the Convex Hull

To remove a point P from CH:

1. Check if P is part of CH (if not, return immediately in $O(\log H)$).
2. Remove P from the convex hull.
3. Recompute the hull of the remaining points by updating the upper and lower hull chains.

Time Complexity for Deletion

- Checking if P is in CH: $O(\log H)$
- Removing P and updating hull: $O(H)$
- **Worst case**: $O(H)$

Final Complexity Analysis

Operation	Complexity
Checking if a point is inside CH	$O(\log H)$
Finding tangents for insertion	$O(\log H)$
Removing K points from hull	$O(K)$
Inserting a new point	$O(1)$
Recomputing hull after deletion	$O(H)$

Worst-case complexity: $O(H)$, since in the worst case, we may need to remove and recompute a significant portion of the hull.

Conclusion

The algorithm efficiently updates the convex hull while allowing dynamic insertions and deletions. Insertion involves tangent-finding and point removal, while deletion involves hull recomputation. The worst-case complexity remains $O(H)$, ensuring optimal convex hull maintenance.

3 Efficient Rectangle Containment Queries Using Segment Tree (Doubly)

Data Structure: Segment Tree for 2D Range Queries

We use a **segment tree** to efficiently:

- Count how many rectangles contain a query point $P(x, y)$.
- Insert new rectangles efficiently without checking all existing rectangles.

Structure of the Segment Tree

1. **Primary Segment Tree (on x-coordinates)**:
   - Built on the sorted set of **x-coordinates** of all rectangle edges.
   - Each node represents an **x-interval** and **stores a secondary segment tree.**
2. **Secondary Segment Tree (on y-coordinates)**:
   - Stores active rectangles whose x-ranges intersect this segment.
   - Allows efficient queries for points in the y-dimension.

Querying for a Point $P(x, y)$

To determine how many rectangles contain P:

1. **Find all relevant x-intervals**:
   - Traverse the **primary segment tree** to find all nodes whose x-ranges contain P_x.
2. **Check y-containment in each segment's secondary tree**:
   - Search in the **y-segment tree** for rectangles containing P_y.
3. **Sum up the results**.

Time Complexity for Querying:

- Primary segment tree search: $O(\log N)$
- Secondary segment tree search: $O(\log N)$
- **Total query complexity:** $O(\log^2 N)$

Inserting a New Rectangle

To insert a new rectangle $[x_{\min}, x_{\max}] \times [y_{\min}, y_{\max}]$:

1. **Find the affected x-segments**:
   - Locate the range $[x_{\min}, x_{\max}]$ in the **primary segment tree**.
2. **Insert the rectangle's y-interval**:
   - Update the **secondary segment trees** for affected x-segments.
3. **Maintain balance in the segment tree**.

Time Complexity for Insertion:

- Primary segment tree update: $O(\log N)$
- Secondary segment tree update: $O(\log N)$
- **Total insertion complexity:** $O(\log^2 N)$

Final Complexity Analysis

Operation	Complexity
Point containment query	$O(\log^2 N)$
Insertion of a rectangle	$O(\log^2 N)$

Edge Cases Considered

- Point P is outside all rectangles $\Rightarrow$ Returns 0.
- Point P is on the boundary of a rectangle $\Rightarrow$ Still counted as contained.
- Adding a rectangle covering multiple existing ones $\Rightarrow$ The problem ensures balance is maintained.

Conclusion

The segment tree efficiently maintains and queries non-overlapping rectangles, allowing insertion in $O(\log^2 N)$ and querying in $O(\log^2 N)$. The structured hierarchy enables rapid point-location queries while maintaining balance and efficiency.

4 Modifying the Weight Balancing Algorithm for Monotone Chains

Original Algorithm: Weight Balancing in a PSLG

Below is the original weight balancing algorithm:

```
1  procedure WeightBalancingInRegularPSLG(G)
2  begin
3    for each edge e in G /* Initialization */
4        W(e) = 1
5    endfor
6    for i = 2 to N - 1 /* First pass */
7        WIN(vi) = sum of weights of incoming edges of vi
8        d1 = leftmost outgoing edge of vi
9        if (WIN(vi) > vOUT(vi))
10           W(d1) = WIN(vi) - vOUT(vi) + 1
11       endif
12   endfor
13   for i = N - 1 to 2 /* Second pass */
14       WOUT(vi) = sum of weights of incoming edges of vi
15       d2 = leftmost incoming edge of vi
16       if (WOUT(vi) > vIN(vi))
17           W(d2) = WOUT(vi) - vIN(vi) + W(d2)
18       endif
19   endfor
20 end
```

The original algorithm consists of two passes:

1. **First Pass (Lines 6-12):** Computes incoming edge weights and adjusts the weight of outgoing edges.

2. **Second Pass (Lines 13-19):** Adjusts incoming edge weights to balance the graph.

First Pass Overview:

- Each edge is initialized with a weight of 1 (Lines 3-5).
- Incoming and outgoing weights are computed for each vertex (Lines 6-7).
- The leftmost outgoing edge d_1 is updated if needed (Lines 8-11).

Second Pass Overview (Before Modification):

- Incoming weights are computed for each vertex (Lines 13-14).
- The leftmost incoming edge d_2 is adjusted if needed (Lines 15-18).

The second pass ensures that incoming and outgoing edge weights are balanced, but it does not guarantee that the resulting chains are monotone. Our goal is to modify this step to construct a **monotone complete set of chains**.

Modifications to the Second Pass

To enforce monotonicity, we make the following changes to **Lines 15-18**:

1. **Sort incoming edges** by a monotonic order (x- or y-coordinate).
2. **Select the best edge** to extend an existing chain while preserving monotonicity.
   - If multiple edges are valid, choose the one that **minimizes oscillation**.
   - If no valid edge exists, start a **new chain**.
3. **Update weight assignments** to maintain balance across chains.
4. **Ensure all vertices are included**:
   - If a vertex has no valid incoming edge, start a new chain from it.

Implementation Details

Algorithmic Changes in Lines 15-18:

- Instead of choosing the leftmost incoming edge (Line 15),
  - Sort all incoming edges by x- or y-coordinate.
  - Select an edge that extends an existing chain in a monotonic order.
- If no valid edge exists, start a new chain (Line 16).
- When updating weights (Lines 17-18), prioritize edges that preserve monotonicity.

Time Complexity Analysis

- Sorting incoming edges: $O(\log N)$ per vertex.
- Processing each vertex: $O(1)$.
- **Total complexity:** $O(N)$, since each edge is processed at most once.

Conclusion

By modifying the second pass of the weight balancing algorithm, we construct a **monotone complete set of chains**. This ensures that all vertices are part of a chain while preserving a consistent monotonic order. The modification maintains the original $O(N)$ complexity, ensuring efficiency in large-scale PSLG processing.

5 Limitations of Graham's Scan with Arbitrary Sorting Points

Counterexample

We consider the following set of points:

$$S = \{A(1,1), B(5,0), C(6,3), D(3,6), E(2,4), F(0,3), G(4,5)\}$$

where:

- A, B, C, D, F, G are **true convex hull points**.
- $E(2,4)$ is an **internal point**.
- If we sort the points using **E** as the reference, the sorting order distorts the correct convex hull.

If we incorrectly sort by **angles relative to E**, the convex hull edges FD, DG, GC, CB, BA, AF are not preserved, and an incorrect connection FE or EG may be introduced. This leads to incorrect hull construction by Graham's Scan.

Why This Fails

- Sorting by polar angles assumes that the reference point is on the convex hull. - Using an internal point E **changes the order of points**, leading Graham's Scan to incorrectly remove necessary convex hull edges. - Some valid hull points are skipped, while internal connections (such as $F - E - G$) distort the convex shape.

Conclusion

- **Not every point can be used as a sorting reference in Graham's Scan**. - Using an **internal point** can disrupt the correct order, leading to an incorrect convex hull. - The correct approach is to choose an **extreme point (e.g., leftmost point) to guarantee the proper convex hull sorting order**.

This proves that Graham's Scan **fails** when sorting with an arbitrary internal reference point, and instead an appropriate hull point must be chosen instead.

Visualization of the Failure Case

The following figure illustrates this counterexample:

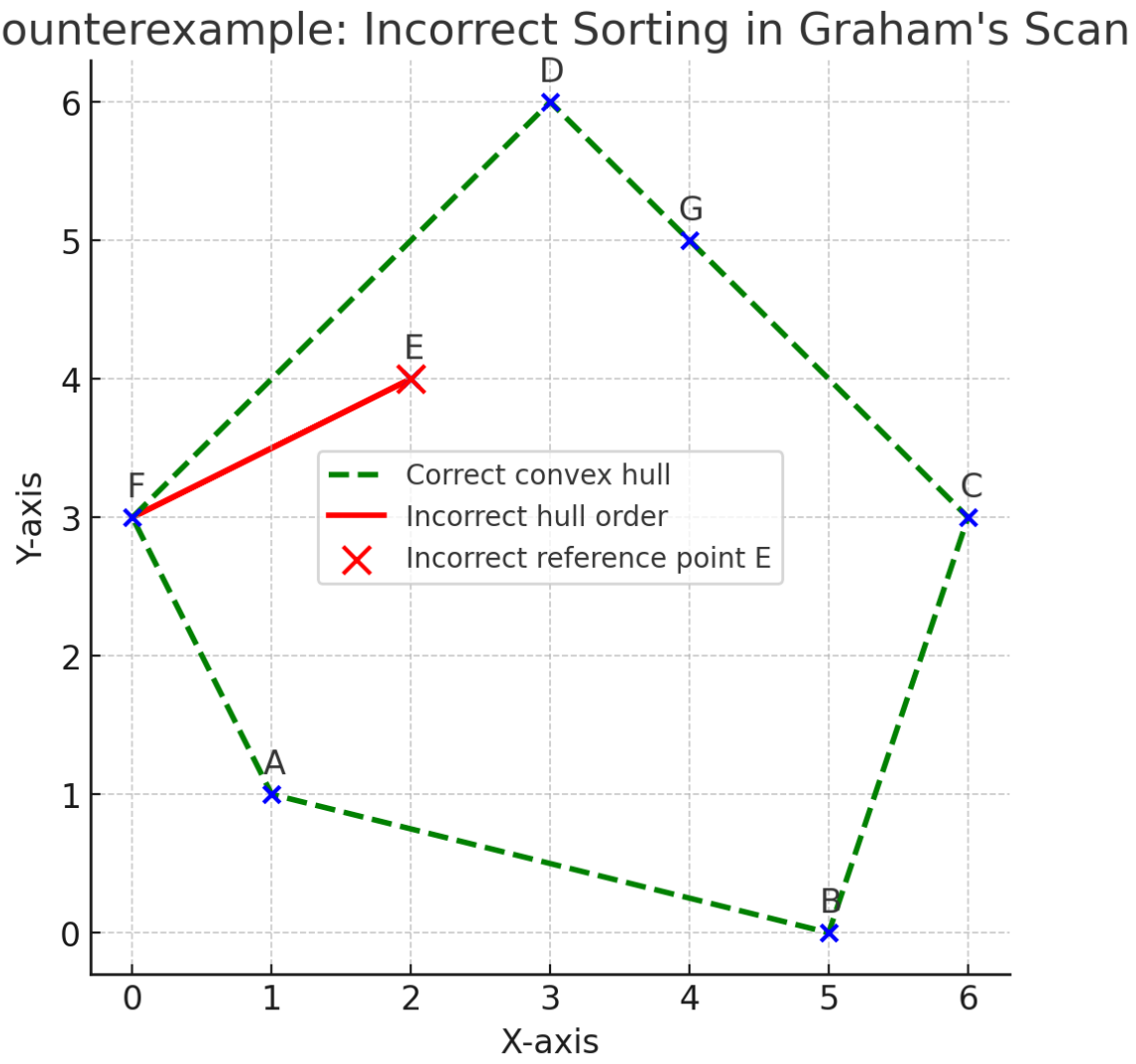

Figure 1: Counter Example