CS 603

Programming Project
(210050131 & 210050114)

Q13- a)

Approach:-

1. Fortune's Algorithm for Delaunay Triangulation

Maintained a **beach line** using a `multiset` of arcs.

Handled **site events** (when the sweep line meets a point) and **circle events** (when arcs disappear) using a **priority queue**.

Upon processing each event:

- Inserted new arcs or removed collapsing arcs.
- Dynamically updated edges of the triangulation.

Carefully maintained a list of **valid events** to handle deletions and outdated circle events.

2. Construction of Voronoi Diagram

Extracted **Voronoi edges** from Delaunay triangles by computing **circumcenters** of all triangles.

Used a **map of triangles and adjacency info** to determine shared edges.

For **unbounded regions**, extended the Voronoi edges in the correct direction using:

- The **midpoint** of Delaunay edge.
- **Perpendicular vector** normalized and extended.
- Direction based on orientation with respect to centroid.

Time Complexity Analysis:

The algorithm achieves an overall time complexity of **O(nlogn)** due to the following reasons:

1. **Event Queue Initialization**:
   - The input points are first sorted by their x-coordinate, which takes **O(nlogn)** time.
2. **Handling Events (Site & Circle)**:
   - There are at most **O(n)** site events (one per point) and **O(n)** valid circle events (as each event corresponds to the disappearance of an arc, and there are at most O(n) arcs over the sweep).
   - Each event is inserted into and extracted from the **priority queue**, which supports O(logn) insertion and removal.
   - Therefore, all events together contribute **O(nlogn)** time.
3. **Beach Line Updates**:
   - The beach line is maintained using a `multiset`, and updates (insertions, deletions, neighbor lookups) take **O(logn)** per operation.

    - Since the number of updates is proportional to the number of events (i.e., O(n)), the total cost here is also **O(nlogn).**
4. **Edge Construction**:
    - Each Delaunay edge is added only once and contributes a constant amount of work.
    - The number of Delaunay edges is **O(n)**, so the total work for this step is **O(n)**.
5. **Voronoi Diagram Construction from Dual**:
    - Uses adjacency information from the Delaunay graph.
    - Triangle formation and circumcenter computation are done in **linear time per triangle**.
    - Since the number of triangles in a Delaunay triangulation is **O(n)**, the total work remains **O(n).**

Q13 - b)

Approach and References:-

To solve this, I implemented a **Delaunay Tree** data structure in C++, based on the randomized incremental algorithm described in:

- **“Fully Dynamic Delaunay Triangulation in Logarithmic Expected Time per Operation”** by Devillers, Meiser, and Teillaud.
- **Chapter 3: The Delaunay Tree**, which outlines the hierarchical nature of the structure.

The key idea is to **preserve all intermediate triangulations** created during point insertions and store them in a directed acyclic graph (the Delaunay Tree).

1.Delaunay Tree Structure

- A rooted DAG where each node represents a triangle.
- Triangles are never deleted during insertions but marked as **dead** if they are invalidated.
- Each triangle may have:
  - **Sons**: New triangles created after insertion.
  - **Step-sons**: Triangles adjacent to dead ones via shared edges.
  - **Flags** to mark state (infinite, dead, degenerate).
  - Pointers to neighbors (current and at creation time).

2. Insertion Process

- Locate the **conflict region** R(p): the set of triangles whose circumcircles contain the new point p.
- Mark all triangles in conflict as **dead**.
- Form a **star-shaped polygon** around p by connecting it to boundary edges of R(p).
- Update neighbor relationships.

3. Classes Implemented

- **`point`**: Represents a 2D point.
- **`DT_flag`**: Encodes the status of a triangle (e.g., infinite, dead, last_finite).
- **`DT_list`**: Linked list used to manage sons/step-sons.
- **`DT_node`**: Represents a triangle; holds pointers to:
  - Vertices
  - Neighbors
  - Sons (new triangles)
  - Conflict-checking logic
- **`Delaunay_tree`**: The top-level class managing the triangulation.

4. Key Functions

- `DT_node::conflict(point*)`: Checks if a point lies inside the circumcircle of a triangle using determinant-based computation.
- `DT_node::find_conflict(point*)`: Recursively locates the triangle in conflict with the inserted point.

- `Delaunay_tree::operator+=`: Handles the insertion process, including:
    - Finding and killing all conflicting triangles.
    - Creating new triangles around the inserted point.
    - Maintaining neighbor and DAG links between old and new triangles.

5. Initialization

The triangulation is initialized with an infinite super-triangle composed of:

- One infinite triangle (root).
- Three infinite triangles with same vertices as root (as neighbors of root).

Pseudo Code:-

```
Insertion(p, T):
 if T not visited and p in conflict with T:
  for each stepson S of T:
   Insertion(p, S)
  for each son S of T:
   Insertion(p, S)
  if T is a leaf:
```

```
mark T as killed

for each neighbor N of T:

  if p not in conflict with N:

    create S as son of T and stepson of N

    update adjacency between S and N
```

6. Time Complexity Justification

Insertion: O(logn) Expected Time

- Follows from the randomized incremental insertion:

  - Expected number of conflicting triangles: **constant**.
  - Triangle location and DAG traversal: O(logn).
  - Star-shaped polygon construction and updates: **amortized constant** time.
- Backed by the paper's probabilistic analysis (Lemma 4.3).

Deletion: O(log2n) Expected Time

- Not implemented, but outlined in the paper using:
  - Reconstructing history via a `Reinsert` structure.
  - Updating the `Star` polygon to restore triangulation locally.