

# CSC279 HW5

Hanzhang Yin

Nov/22/2023

## Collaborator

Chenxi Xu, Yekai Pan, Yiling Zou, Boyi Zhang

## PROBLEM 15

### Answer:

1. Assume  $q$  is inside  $P$ . We want to find the closest point to  $q$  in  $\{p_1, \dots, p_n\}$ .

#### Reasoning:

Hard and required  $\Omega(n)$  runtime. Arrange all points on the circumference of a circle like regular convex  $n$ -polygon enclosing point  $q$  as its center. If the algorithm is deterministic and assumes an easy case, some points remain unvisited. For those unvisited point, we one of them closer. Therefore, the algorithm can not find the correct closest point and outputting incorrect results.

2. Assume  $q$  is outside  $P$ . We want to find the closest point to  $q$  in  $\{p_1, \dots, p_n\}$ .

#### Reasoning:

Hard and required  $\Omega(n)$  runtime. Place all points on a quarter-circle like regular convex  $n$ -polygon enclosing point  $q$  while ensuring  $P$  is convex and non-enclosing. Assume easy, then there will be some point that the algorithm (deterministic) will not visit. For an unvisited point, we move it closer. Therefore, the algorithm can not find the correct closest point and outputting incorrect results.

3. Assume  $q$  is inside  $P$ . We want to find the farthest point to  $q$  in  $\{p_1, \dots, p_n\}$ .

#### Reasoning:

Hard and required  $\Omega(n)$  runtime. Similar to Q1, but this time we move an unvisited point further.

4. Assume  $q$  is outside  $P$ . We want to find the farthest point to  $q$  in  $\{p_1, \dots, p_n\}$ .

#### Reasoning:

Hard and required  $\Omega(n)$  runtime. Similar to Q2, but this time we move an unvisited point further.

5. Assume  $q$  is inside  $P$ . We want to find the closest point to  $q$  on  $P$ .

**Reasoning:**

Hard and required  $\Omega(n)$  runtime. Similar to Q1 again, The number of edges equals the number of points, so we still need at least  $O(n)$  runtime.

6. Assume  $q$  is outside  $P$ . We want to find the closest point to  $q$  on  $P$ .

**Reasoning:**

Easy and can be solved within  $O(\log n)$ .

The algorithm finds the closest point to  $q$  on a convex polygon  $P$  in  $O(\log n)$  time using the unimodal nature of the distance function from  $q$  to  $P$ . Using ternary search on the edges of  $P$ , it narrows the search interval until the closest edge is identified, and then computes the closest point on that edge. The convexity of  $P$  ensures the unimodal property, guaranteeing the correctness of the ternary search.

```
def closest_point_on_convex_polygon(q, P):
    n = len(P)
    low = 0
    high = n - 1

    while high - low > 3:
        m1 = low + (high - low) // 3
        m2 = high - (high - low) // 3

        D_m1 = distance_to_edge(q, P[m1], P[(m1 + 1) % n])
        D_m2 = distance_to_edge(q, P[m2], P[(m2 + 1) % n])

        if D_m1 < D_m2:
            high = m2
        else:
            low = m1

    min_dist = float('inf')
    closest_point = None

    for i in range(low, high + 1):
        p1 = P[i]
        p2 = P[(i + 1) % n]
        c = closest_point_on_segment(q, p1, p2)
        D = distance(q, c)
        if D < min_dist:
            min_dist = D
            closest_point = c

    return closest_point
```

7. Assume  $q$  is inside  $P$ . We want to find the farthest point to  $q$  on  $P$ .

**Reasoning:**

Hard and required  $\Omega(n)$  runtime. Similar to Q3, so similar argument can be made.

8. Assume  $q$  is outside  $P$ . We want to find the farthest point to  $q$  on  $P$ .

**Reasoning:**

Hard and required  $\Omega(n)$  runtime. Similar to Q4. The farthest point must lie on the circumcircle as all other points are on an  $n$ -gon, so similar argument can be made.

## PROBLEM 16

*Proof.* **Theorem:**

A convex polygon is fully contained within the largest circumcircle formed by three of its consecutive vertices.

**Lemma:**

Let  $P = \{p_1, p_2, \dots, p_n\}$  represent a convex polygon. Suppose a triangle  $T$  is formed by three vertices of  $P$ , and the circumcircle of  $T$  contains  $P$ . For any edge  $\overline{p_a p_b}$  of  $T$ , there exists a vertex  $p_c$  between  $p_a$  and  $p_b$  such that the circumcircle of  $T_{ab,c}$  contains  $P$ .

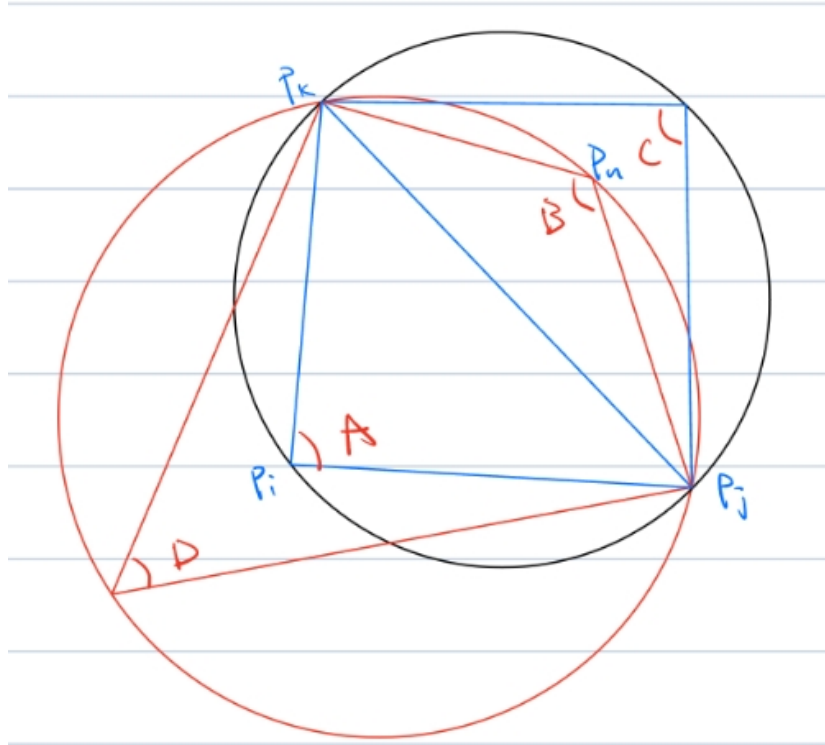
**Proof**

Let  $p_i, p_j, p_k$  be three vertices defining the triangle  $T_{ij,k}$ . Without loss of generality, consider the edge  $\overline{p_j p_k}$  and its corresponding arc on the circumcircle. There exists  $c \in (i, j)$  such that for every  $a \in (i, j)$ , the circumcircle of  $T_{ij,c}$  contains  $p_a$ .

Now, we examine two cases for  $p_c$ :

**Case 1:  $p_c$  lies on the circumcircle of  $T_{ij,k}$**  If  $p_c$  is on the circumcircle of  $T_{ij,k}$ , then the circumcircle of  $T_{j,k,c}$  is the same as that of  $T_{ij,k}$ . Thus, the lemma holds.

**Case 2:  $p_c$  lies inside the circumcircle of  $T_{ij,k}$**  Construct a quadrilateral



with vertices  $p_j, p_c, p_k, D$  that forms a cyclic quadrilateral. By the properties of cyclic quadrilaterals:

$$\angle A + \angle D = \pi \quad \text{and} \quad \angle B + \angle C = \pi.$$

Using these properties:

$$\angle A + \angle B = \pi - \angle C < \pi - \angle D \implies \angle A > \angle D.$$

For any point  $z$  inside or on the circumcircle of  $T_{i,j,k}$ , it cannot satisfy  $\angle p_j z p_k > \angle p_j p_i p_k$ . Thus,  $z$  must lie outside the circumcircle of  $T_{i,j,k}$ , ensuring that the circumcircle of  $T_{i,j,c}$  contains all points between the arc  $p_j p_i p_k$ . Hence, the lemma is proved.

### **Triangulation Construction**

1. Start with three consecutive vertices  $p_i, p_{i+1}, p_{i+2}$  such that the circumcircle of  $T_{p_i p_{i+1} p_{i+2}}$  contains  $P$ .
2. For each new triangle  $T$ , select any edge  $\overline{p_a p_b}$ . If there are no points of  $P$  within the range of vertices  $\overline{p_a p_b}$ , skip this edge. Otherwise, find a point  $p_c$  such that the circumcircle of  $T_{p_a p_b p_c}$  contains  $P$ .
3. Repeat this process iteratively, ensuring that every newly constructed triangle satisfies the condition that its circumcircle contains  $P$ .

This method guarantees that the entire polygon  $P$  is contained within the circumcircle of the final triangulation.  $\square$

## PROBLEM 17

### General Algorithm Thoughts:

1. **Construct the Voronoi diagram** for all sites.
2. **For each Voronoi cell**, examine its corners (vertices).
3. **Check if any corner is at a distance  $\geq l + r$  from its associated site.**
  - **Reasoning:** Corners are the farthest points within a cell from the site.
  - They are **equidistant to the site and neighboring sites**.
  - If a corner is at distance  $\geq l + r$  from the site, it is also that far from neighboring sites.
4. **Conclusion:** If such a corner exists, the site is "good" because all points at that corner are sufficiently distant from all relevant sites.

### Potentially A More Refined and Rigorous Version

The algorithm identifies all "good" points by first constructing the Voronoi diagram of the given points, which efficiently captures proximity relationships in  $O(n \log n)$  time. For each point  $p_i$ , it examines only its neighboring points in the Voronoi diagram, as these are the only ones that could potentially interfere with placing a new circle. By computing the angular intervals where a circle of radius  $\ell$  touching  $C_i$  would intersect any neighboring  $C_j$ , the algorithm determines the directions that are blocked. If there exists at least one direction where such interference does not occur, the point  $p_i$  is therefore "good".

#### # Helper Functions

```
def compute_interfering_angles(p_i, p_j, r, l, d_ij):
    # Calculate the angle between p_i and p_j
    delta_x = p_j.x - p_i.x
    delta_y = p_j.y - p_i.y
    alpha = atan2(delta_y, delta_x)

    # Law of Cosines to find the angular width
    cos_theta = (d_ij**2 + (r + l)**2 - (r + l)**2) / (2 * d_ij * (r + l))
    if abs(cos_theta) <= 1:
        theta = acos(cos_theta)
        # The interfering interval is [alpha - theta, alpha + theta]
        interval = [(alpha - theta) % (2 * pi), (alpha + theta) % (2 * pi)]
        # Handle interval wrapping around 2pi
        if interval[0] > interval[1]:
            return [(interval[0], 2 * pi), (0, interval[1])]
        else:
```

```

        return [interval]
    else:
        # Circles do not intersect; no interfering angles
        return []

# Main Function
def find_good_points(P, r, l):
    # Construct the Voronoi diagram
    # Need  $O(n \log n)$ 
    V = voronoi_diagram(P)

    good_points = []

    # For each point p_i
    # Need  $O(n)$ 
    for p_i in P:
        interfering_angles = [] # List to store interfering angular intervals

        # Get neighboring points in the Voronoi diagram
        neighbors = V.get_neighbors(p_i)

        # For each neighbor p_j
        for p_j in neighbors:
            d_ij = distance(p_i, p_j)

            # Only consider neighbors that may interfere
            if d_ij < 2 * (r + l):
                # Compute the angular intervals of interference
                angles = compute_interfering_angles(p_i, p_j, r, l, d_ij)
                interfering_angles.extend(angles)

        # Compute the union of interfering intervals
        interfering_union = union_of_intervals(interfering_angles)

        # Determine the complement of the union over  $[0, 2\pi)$ 
        non_interfering_angles = complement_of_intervals(interfering_union, 0, 2 * pi)

        # If there is at least one non-interfering angle, p_i is good
        if non_interfering_angles:
            good_points.append(p_i)

    return good_points

```

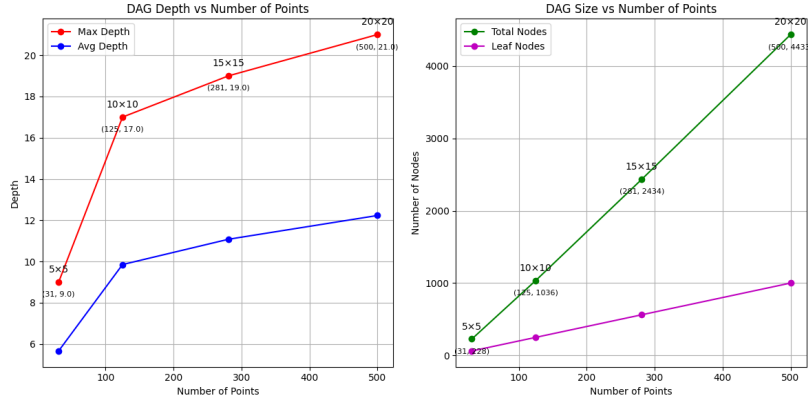
## PROBLEM 18

**Summary:** The randomized incremental Delaunay triangulation algorithm employs a history DAG to efficiently manage point insertion and triangle updates while ensuring the Delaunay property. The algorithm inserts points in random order, using the history DAG for  $O(\log n)$  expected-time point location, followed by triangle splitting and recursive edge flipping to maintain the Delaunay criterion. With an expected time complexity of  $O(n \log n)$  and space complexity of  $O(n \log n)$  (including the history DAG), it offers good average-case performance and practical simplicity.

**Numerical Result:**

Points	Grid	Max Depth	Avg Depth	Total Nodes
25	$5 \times 5$	9	5.66	63
100	$10 \times 10$	17	9.85	251
225	$15 \times 15$	19	11.08	563
400	$20 \times 20$	21	12.23	1001

Table 1: Depth and node statistics for randomized Delaunay triangulation on different grid sizes.



**Short Analysis:**

The result I got is reasonable, with the average depths increasing in  $\log n$  as expected, indicating that the algorithm effectively maintains a balanced DAG structure. Although the maximum depths are somewhat higher than theoretical predictions, they remain within an acceptable range considering the algorithm's randomness and the potential for local depth increases during edge legalization.

**Implementation:**

The following code of randomized Delaunay triangulation algorithm with history



DAG was implemented in Python (I fixed the random seed = 20 for better representation):

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 from typing import List, Set, Tuple, Optional
4 from dataclasses import dataclass
5 import random
6 from collections import defaultdict
7
8 # Basic geometric structures
9 @dataclass
10 class Point2D:
11     x: float
12     y: float
13
14     def __eq__(self, other):
15         if not isinstance(other, Point2D):
16             return False
17         return abs(self.x - other.x) < 1e-10 and abs(self.y - other
18             .y) < 1e-10
19
20     def __str__(self):
21         return f"Point2D({self.x:.2f}, {self.y:.2f})"
22
23 class Triangle2D:
24     def __init__(self, points=None, p1=None, p2=None, p3=None):
25         if points is not None:
26             self.vertices = list(points)
27         else:
28             self.vertices = [p1, p2, p3]
29
30     def get_points(self):
31         return self.vertices
32
33     def get_point(self, index):
34         return self.vertices[index]
35
36     def set_points(self, points=None, p1=None, p2=None, p3=None):
37         if points is not None:
38             self.vertices = list(points)
39         else:
40             self.vertices = [p1, p2, p3]
41
42 class DagNode:
43     def __init__(self, triangle_index: int):
44         self.triangle = triangle_index
45         self.children = []
46
47     def append_child(self, new_node):
48         self.children.append(new_node)
49
50     def get_index(self):
51         return self.triangle
52
53     def get_children(self):
54         return self.children

```

```

54
55 class TriangulationMember(Triangle2D):
56     def __init__(self, points, adj_list, dag_node, is_active=True):
57         super().__init__(points)
58         self.adj_list = list(adj_list)
59         self.dag_node = dag_node
60         self.active = is_active
61
62     def set_active(self):
63         self.active = True
64
65     def set_inactive(self):
66         self.active = False
67
68     def is_active(self):
69         return self.active
70
71     def get_neighbour(self, index):
72         return self.adj_list[index]
73
74     def get_neighbours(self):
75         return self.adj_list
76
77     def get_dag_node(self):
78         return self.dag_node
79
80     def set_neighbour(self, neighbour, new_index):
81         self.adj_list[neighbour] = new_index
82
83 class Triangulation:
84     def __init__(self, init_triangle: Triangle2D, dag_node: DagNode
85 ):
86         adj_list = [0, 0, 0]
87         self.triangles = [TriangulationMember(init_triangle.
88             get_points(), adj_list, dag_node)]
89
90     def get_triangle(self, index):
91         return self.triangles[index]
92
93     def get_triangles(self):
94         return self.triangles
95
96     def size(self):
97         return len(self.triangles)
98
99     def add_triangle(self, triangle):
100         self.triangles.append(triangle)
101
102     def set_triangle_active(self, index):
103         self.triangles[index].set_active()
104
105     def set_triangle_inactive(self, index):
106         self.triangles[index].set_inactive()
107
108     def set_triangle_neighbour(self, triangle, neighbour, new_index
109 ):
110         self.triangles[triangle].set_neighbour(neighbour, new_index

```

```

    )
108
109 # Geometric utilities
110 class GeometryUtils:
111     @staticmethod
112     def point_in_circle(p1: Point2D, p2: Point2D, p3: Point2D, p4:
        Point2D, include_edges: bool) -> bool:
113         matrix = np.array([
114             [p1.x - p4.x, p1.y - p4.y, (p1.x - p4.x)**2 + (p1.y -
                p4.y)**2],
115             [p2.x - p4.x, p2.y - p4.y, (p2.x - p4.x)**2 + (p2.y -
                p4.y)**2],
116             [p3.x - p4.x, p3.y - p4.y, (p3.x - p4.x)**2 + (p3.y -
                p4.y)**2]
117         ])
118         det = np.linalg.det(matrix)
119         return det > 0 if include_edges else det >= 0
120
121     @staticmethod
122     def point_position_to_segment(p1: Point2D, p2: Point2D, p:
        Point2D) -> float:
123         return (p2.x - p1.x) * (p.y - p1.y) - (p2.y - p1.y) * (p.x
            - p1.x)
124
125     @staticmethod
126     def point_in_triangle(p1: Point2D, p2: Point2D, p3: Point2D, p:
        Point2D, include_edges: bool) -> bool:
127         pos1 = GeometryUtils.point_position_to_segment(p1, p2, p)
128         pos2 = GeometryUtils.point_position_to_segment(p2, p3, p)
129         pos3 = GeometryUtils.point_position_to_segment(p3, p1, p)
130
131         if include_edges:
132             return (pos1 >= 0 and pos2 >= 0 and pos3 >= 0) or (pos1
                <= 0 and pos2 <= 0 and pos3 <= 0)
133         else:
134             return (pos1 > 0 and pos2 > 0 and pos3 > 0) or (pos1 <
                0 and pos2 < 0 and pos3 < 0)
135
136 class DelaunayTriangulation:
137     @staticmethod
138     def update_index_in_neighbour(triangulation: Triangulation,
        triangle_index: int,
139                                 neighbour_index: int, new_index:
        int):
140         if neighbour_index != 0:
141             neighbour = triangulation.get_triangle(neighbour_index)
142             for i in range(3):
143                 if neighbour.get_neighbour(i) == triangle_index:
144                     triangulation.set_triangle_neighbour(
                        neighbour_index, i, new_index)
145
146     @staticmethod
147     def find_index_in_neighbour(triangulation: Triangulation,
        triangle_index: int,
148                                 neighbour_index: int) -> int:
149         for i in range(3):
150             if triangulation.get_triangle(neighbour_index).

```

```

151         get_neighbour(i) == triangle_index:
152             return i
153     return 3
154
155     @staticmethod
156     def flip_edge(triangulation: Triangulation, triangle_index: int
157                  , point_index: int):
158         triangle = triangulation.get_triangle(triangle_index)
159
160         if triangle.get_neighbour((point_index + 1) % 3) != 0:
161             adj_triangle = triangulation.get_triangle(triangle.
162                 get_neighbour((point_index + 1) % 3))
163             adj_point_index = (DelaunayTriangulation.
164                 find_index_in_neighbour(
165                     triangulation, triangle_index,
166                     triangle.get_neighbour((point_index + 1) % 3)) + 2)
167                 % 3
168
169             if GeometryUtils.point_in_circle(
170                 triangle.get_point(0), triangle.get_point(1),
171                 triangle.get_point(2), adj_triangle.get_point(
172                     adj_point_index), False):
173
174                 triangulation.set_triangle_inactive(triangle_index)
175                 triangulation.set_triangle_inactive(triangle.
176                     get_neighbour((point_index + 1) % 3))
177
178                 current_index = triangulation.size()
179                 new_triangle_index1 = current_index
180                 new_triangle_index2 = current_index + 1
181
182                 # Create new triangles
183                 new_triangle1 = Triangle2D(
184                     p1=triangle.get_point(point_index),
185                     p2=triangle.get_point((point_index + 1) % 3),
186                     p3=adj_triangle.get_point(adj_point_index)
187                 )
188                 new_triangle2 = Triangle2D(
189                     p1=triangle.get_point(point_index),
190                     p2=adj_triangle.get_point(adj_point_index),
191                     p3=triangle.get_point((point_index + 2) % 3)
192                 )
193
194                 # Set up adjacency lists
195                 adj_list1 = [
196                     triangle.get_neighbour(point_index),
197                     adj_triangle.get_neighbour((adj_point_index +
198                         2) % 3),
199                     new_triangle_index2
200                 ]
201                 adj_list2 = [
202                     new_triangle_index1,
203                     adj_triangle.get_neighbour(adj_point_index),
204                     triangle.get_neighbour((point_index + 2) % 3)
205                 ]
206
207                 # Update neighbors

```

```

200         DelaunayTriangulation.update_index_in_neighbour(
201             triangulation, triangle_index,
202             triangle.get_neighbour(point_index),
203             new_triangle_index1
204         )
205         DelaunayTriangulation.update_index_in_neighbour(
206             triangulation, triangle.get_neighbour((
207                 point_index + 1) % 3),
208             adj_triangle.get_neighbour((adj_point_index +
209                 2) % 3),
210             new_triangle_index1
211         )
212         DelaunayTriangulation.update_index_in_neighbour(
213             triangulation, triangle_index,
214             triangle.get_neighbour((point_index + 2) % 3),
215             new_triangle_index2
216         )
217         DelaunayTriangulation.update_index_in_neighbour(
218             triangulation, triangle.get_neighbour((
219                 point_index + 1) % 3),
220             adj_triangle.get_neighbour(adj_point_index),
221             new_triangle_index2
222         )
223
224         # Create DAG nodes
225         dag1 = DagNode(new_triangle_index1)
226         dag2 = DagNode(new_triangle_index2)
227
228         # Add triangles to triangulation
229         triangulation.add_triangle(TriangulationMember(
230             new_triangle1.get_points(), adj_list1, dag1
231         ))
232         triangulation.add_triangle(TriangulationMember(
233             new_triangle2.get_points(), adj_list2, dag2
234         ))
235
236         # Update DAG
237         triangle.get_dag_node().append_child(dag1)
238         adj_triangle.get_dag_node().append_child(dag1)
239         adj_triangle.get_dag_node().append_child(dag2)
240         triangle.get_dag_node().append_child(dag2)
241
242         # Recursively check new edges
243         DelaunayTriangulation.flip_edge(triangulation,
244             new_triangle_index1, 0)
245         DelaunayTriangulation.flip_edge(triangulation,
246             new_triangle_index2, 0)
247
248     @staticmethod
249     def discard_bounding_vertexes(triangulation: Triangulation):
250         bounding_triangle = triangulation.get_triangle(0)
251         for i in range(triangulation.size()):
252             if triangulation.get_triangle(i).is_active():
253                 triangle = triangulation.get_triangle(i)
254                 for j in range(3):
255                     if (triangle.get_point(j) == bounding_triangle.
256                         get_point(0) or

```

```

250         triangle.get_point(j) == bounding_triangle.
251             get_point(1) or
252         triangle.get_point(j) == bounding_triangle.
253             get_point(2)):
254             triangulation.set_triangle_inactive(i)
255             break
256
257 @staticmethod
258 def get_triangulation(triangulation: Triangulation, dag:
259     DagNode,
260     points: List[Point2D]):
261     shuffled_points = points.copy()
262     random.shuffle(shuffled_points)
263
264     for point in shuffled_points:
265         DelaunayTriangulation.incremental_step(triangulation,
266             dag, point)
267
268     DelaunayTriangulation.discard_bounding_vertexes(
269         triangulation)
270
271 @staticmethod
272 def incremental_step(triangulation: Triangulation, dag: DagNode
273     , point: Point2D):
274     current_node = DelaunayTriangulation.locate_point(
275         triangulation, dag, point)
276     triangulation.set_triangle_inactive(current_node.get_index
277         ())
278     current_triangle = triangulation.get_triangle(current_node.
279         get_index())
280
281     # Check if point already exists
282     if (point == current_triangle.get_point(0) or
283         point == current_triangle.get_point(1) or
284         point == current_triangle.get_point(2)):
285         return
286
287     # Split triangle
288     current_index = triangulation.size()
289     for i in range(3):
290         new_triangle = Triangle2D(
291             p1=point,
292             p2=current_triangle.get_point(i),
293             p3=current_triangle.get_point((i + 1) % 3)
294         )
295         adj_list = [
296             current_index + ((i + 2) % 3),
297             current_triangle.get_neighbour(i),
298             current_index + ((i + 1) % 3)
299         ]
300         dag_node = DagNode(current_index + i)
301         triangulation.add_triangle(TriangulationMember(
302             new_triangle.get_points(), adj_list, dag_node
303         ))
304         current_node.append_child(dag_node)
305
306     DelaunayTriangulation.update_index_in_neighbour(

```

```

298         triangulation, current_node.get_index(),
299         current_triangle.get_neighbour(i),
300         current_index + i
301     )
302
303     # Check and flip edges
304     for i in range(3):
305         DelaunayTriangulation.flip_edge(triangulation,
306                                         current_index + i, 0)
307
308     @staticmethod
309     def locate_point(triangulation: Triangulation, dag: DagNode,
310                     point: Point2D) -> DagNode:
311         for child in dag.get_children():
312             triangle = triangulation.get_triangle(child.get_index())
313             if GeometryUtils.point_in_triangle(
314                 triangle.get_point(0), triangle.get_point(1),
315                 triangle.get_point(2), point, True
316             ):
317                 return DelaunayTriangulation.locate_point(
318                     triangulation, child, point)
319         return dag
320
321 class DelaunayTest:
322     @staticmethod
323     def create_bounding_triangle(points: List[Point2D]) ->
324         Triangle2D:
325         """Create a triangle that contains all points with some
326             margin."""
327         min_x = min(p.x for p in points) - 0.1
328         max_x = max(p.x for p in points) + 0.1
329         min_y = min(p.y for p in points) - 0.1
330         max_y = max(p.y for p in points) + 0.1
331
332         dx = max_x - min_x
333         dy = max_y - min_y
334         center_x = (min_x + max_x) / 2
335         center_y = (min_y + max_y) / 2
336         size = max(dx, dy) * 2
337
338         p1 = Point2D(center_x - size, center_y - size)
339         p2 = Point2D(center_x + size, center_y - size)
340         p3 = Point2D(center_x, center_y + size)
341
342         return Triangle2D(p1=p1, p2=p2, p3=p3)
343
344     @staticmethod
345     def generate_test_points(n: int, include_random: bool = True)
346         -> List[Point2D]:
347         """Generate test points in both grid and random patterns."""
348         points = []
349
350         # Generate grid points
351         for i in np.linspace(0, 1, n):
352             for j in np.linspace(0, 1, n):

```

```

347         points.append(Point2D(i, j))
348
349     # Add random points if requested
350     if include_random:
351         num_random = n * n // 4 # Add 25% more random points
352         random_points = [Point2D(random.random(), random.random
353                                ())
354                           for _ in range(num_random)]
355         points.extend(random_points)
356
357     return points
358
359 @staticmethod
360 def verify_delaunay_property(triangulation: Triangulation) ->
361     bool:
362     """Verify that the triangulation satisfies the Delaunay
363     property."""
364     for i, tri in enumerate(triangulation.get_triangles()):
365         if not tri.is_active():
366             continue
367
368         # Get triangle vertices
369         p1, p2, p3 = tri.get_points()
370
371         # Check against all points
372         for j, other_tri in enumerate(triangulation.
373                                       get_triangles()):
374             if not other_tri.is_active() or i == j:
375                 continue
376
377             # Check if any point from other triangles lies
378             # inside this triangle's circumcircle
379             for point in other_tri.get_points():
380                 if GeometryUtils.point_in_circle(p1, p2, p3,
381                                                  point, False):
382                     return False
383
384     return True
385
386 @staticmethod
387 def analyze_dag_structure(root: DagNode) -> dict:
388     """Analyze the DAG structure and return statistics."""
389     depths = []
390     nodes = []
391     queue = [(root, 0)]
392     visited = set()
393     max_depth = 0
394
395     while queue:
396         node, depth = queue.pop(0)
397         if node in visited:
398             continue
399
400         visited.add(node)
401         nodes.append(node)
402         depths.append(depth)
403         max_depth = max(max_depth, depth)

```



```

398         for child in node.get_children():
399             queue.append((child, depth + 1))
400
401     return {
402         'max_depth': max_depth,
403         'avg_depth': sum(depths) / len(depths) if depths else
404         0,
405         'total_nodes': len(nodes),
406         'leaf_nodes': sum(1 for n in nodes if not n.
407                             get_children()),
408         'branching_factor': len(nodes) / (len(nodes) - 1) if
409                             len(nodes) > 1 else 0
410     }
411
412 @staticmethod
413 def plot_triangulation(triangulation: Triangulation, points:
414     List[Point2D],
415     title: str = "Delaunay Triangulation")
416     -> None:
417     """Visualize the triangulation."""
418     plt.figure(figsize=(12, 12))
419
420     # Plot points
421     xs = [p.x for p in points]
422     ys = [p.y for p in points]
423     plt.scatter(xs, ys, c='red', s=50, zorder=3, label='Input
424         Points')
425
426     # Plot triangles
427     for tri in triangulation.get_triangles():
428         if tri.is_active():
429             vertices = tri.get_points()
430             xs = [v.x for v in vertices + [vertices[0]]]
431             ys = [v.y for v in vertices + [vertices[0]]]
432             plt.plot(xs, ys, 'b-', alpha=0.5, zorder=1)
433
434     plt.title(title)
435     plt.xlabel('X')
436     plt.ylabel('Y')
437     plt.legend()
438     plt.grid(True, alpha=0.3)
439     plt.axis('equal')
440     plt.show()
441
442 def run_comprehensive_test():
443     """Run a comprehensive test of the Delaunay triangulation
444     implementation."""
445     print("Starting Delaunay Triangulation Tests...")
446
447     # Test different grid sizes
448     grid_sizes = [5, 10, 15, 20]
449     results = []
450
451     for n in grid_sizes:
452         print(f"\nTesting {n}x{n} grid...")
453
454         # Generate test points

```

```

448     points = DelaunayTest.generate_test_points(n)
449     print(f"Generated {len(points)} points")
450
451     # Create initial triangulation
452     bounding_tri = DelaunayTest.create_bounding_triangle(points
453 )
454     root_node = DagNode(0)
455     triangulation = Triangulation(bounding_tri, root_node)
456
457     # Run triangulation
458     DelaunayTriangulation.get_triangulation(triangulation,
459 root_node, points)
460
461     # Verify properties
462     is_delaunay = DelaunayTest.verify_delaunay_property(
463 triangulation)
464     dag_stats = DelaunayTest.analyze_dag_structure(root_node)
465
466     results.append({
467         'grid_size': n,
468         'num_points': len(points),
469         'is_delaunay': is_delaunay,
470         'dag_stats': dag_stats
471     })
472
473     # Visualize
474     DelaunayTest.plot_triangulation(triangulation, points,
475                                     f"Delaunay Triangulation ({
476 n}x{n} grid)")
477
478     # Print statistics
479     print(f"Results for {n}x{n} grid:")
480     print(f"- Number of points: {len(points)}")
481     print(f"- Delaunay property satisfied: {is_delaunay}")
482     print(f"- DAG Statistics:")
483     print(f"    - Maximum depth: {dag_stats['max_depth']}")
484     print(f"    - Average depth: {dag_stats['avg_depth']:.2f}")
485     print(f"    - Total nodes: {dag_stats['total_nodes']}")
486     print(f"    - Leaf nodes: {dag_stats['leaf_nodes']}")
487     print(f"    - Average branching factor: {dag_stats['
488 branching_factor']:.2f}")
489
490     # Plot summary statistics
491     plt.figure(figsize=(12, 6))
492
493     # Plot depths
494     plt.subplot(121)
495     plt.plot([r['grid_size'] for r in results],
496             [r['dag_stats']['max_depth'] for r in results],
497             'ro-', label='Max Depth')
498     plt.plot([r['grid_size'] for r in results],
499             [r['dag_stats']['avg_depth'] for r in results],
500             'bo-', label='Avg Depth')
501     plt.xlabel('Grid Size')
502     plt.ylabel('Depth')
503     plt.title('DAG Depth Analysis')
504     plt.legend()

```

```

500 plt.grid(True)
501
502 # Plot nodes
503 plt.subplot(122)
504 plt.plot([r['grid_size'] for r in results],
505          [r['dag_stats']['total_nodes'] for r in results],
506          'go-', label='Total Nodes')
507 plt.plot([r['grid_size'] for r in results],
508          [r['dag_stats']['leaf_nodes'] for r in results],
509          'mo-', label='Leaf Nodes')
510 plt.xlabel('Grid Size')
511 plt.ylabel('Number of Nodes')
512 plt.title('DAG Size Analysis')
513 plt.legend()
514 plt.grid(True)
515
516 plt.tight_layout()
517 plt.show()
518
519 if __name__ == "__main__":
520     # Set random seed for reproducibility
521     random.seed(20)
522     np.random.seed(20)
523
524     # Run the comprehensive test
525     run_comprehensive_test()

```