from classical\_algorithm.foronoi import Voronoi, Polygon  
import matplotlib.pyplot as plt  
import math  
import copy  
import gc  
  
class DeterministicAlgorithm:  
 def \_\_init\_\_(self, polygon, batch\_size):  
 self.polygon = polygon  
 self.\_original\_polygon = copy.deepcopy(self.polygon)  
 self.batch\_size = batch\_size  
  
 self.edges = []  
 self.points = []  
 self.finished\_points = []  
 self.upcoming\_points = []  
 self.finish\_flag = []  
  
 self.small\_sites = []  
 self.big\_sites = []  
  
 self.small\_sites\_index = []  
 self.big\_sites\_index = []  
  
 def algorithm\_process(self, points, vis\_steps=False, vis\_result=False):  
 self.create\_diagram(points=points, phase\_number=1)  
 self.create\_diagram(points=points, phase\_number=2, vis\_steps=vis\_steps)  
 self.create\_diagram(points=points, phase\_number=3, vis\_result=vis\_result)  
  
 def create\_diagram(self, points, phase\_number, vis\_steps=False, vis\_result=False):  
 if phase\_number == 1:  
 self.points = points  
 self.upcoming\_points = points[self.batch\_size:]  
 current\_set = points[:self.batch\_size]  
 current\_ray = [1] \* self.batch\_size  
 iteration\_times = [0] \* self.batch\_size  
 self.finish\_flag = [None] \* self.batch\_size  
 current\_ray\_determined\_point = [(x + 1, y + 1) for x, y in current\_set]  
 while all(p is not None for p in current\_set):  
 # print(f"\nProcessing point set: {current\_set}")  
 # print(f"\nCurrent\_ray: {current\_ray}")  
 # print(f"Current\_ray\_determined\_point: {current\_ray\_determined\_point}")  
 # print(f"Current\_iteration\_times: {iteration\_times}")  
 current\_set, current\_ray, current\_ray\_determined\_point, iteration\_times = self.process\_small\_sites(current\_set, current\_ray, current\_ray\_determined\_point, points, iteration\_times, phase\_number)  
 self.small\_sites = self.finished\_points  
 self.small\_sites\_index = [self.points.index(p) for p in self.small\_sites]  
 for point in current\_set:  
 if point:  
 self.big\_sites.append(point)  
 self.big\_sites\_index.append(self.points.index(point))  
 # print(f"Phase 1 Success! small\_sites: {self.small\_sites}, big\_sites: {self.big\_sites}")  
 # print(f"small\_sites\_index: {self.small\_sites\_index}, big\_sites\_index: {self.big\_sites\_index}\n")  
 if phase\_number == 2:  
 self.points = points  
 self.upcoming\_points = points[self.batch\_size:]  
 current\_set = points[:self.batch\_size]  
 current\_ray = [1] \* self.batch\_size  
 iteration\_times = [0] \* self.batch\_size  
 self.finish\_flag = [None] \* self.batch\_size  
 current\_ray\_determined\_point = [(x + 1, y + 1) for x, y in current\_set]  
 while all(p is not None for p in current\_set):  
 # print(f"\nProcessing point set: {current\_set}")  
 # print(f"\nCurrent\_ray: {current\_ray}")  
 # print(f"Current\_ray\_determined\_point: {current\_ray\_determined\_point}")  
 # print(f"Current\_iteration\_times: {iteration\_times}")  
 current\_set, current\_ray, current\_ray\_determined\_point, iteration\_times = self.process\_small\_sites(current\_set, current\_ray, current\_ray\_determined\_point, points, iteration\_times, phase\_number)  
 # print(f"Phase 2 Success! - partial result as shown")  
 # print(f"len(self.edges): {len(self.edges)}\nself.edges:{self.edges}\n")  
 if vis\_steps:  
 self.plot\_final\_result()  
 if phase\_number == 3:  
 self.process\_big\_sites(self.big\_sites, points)  
 # print(f"Phase 3 Success! - final result as shown")  
 # print(f"After Phase 3: \nlen(self.edges): {len(self.edges)}\nself.edges:{self.edges}")  
 if vis\_result:  
 self.plot\_final\_result()  
  
 def plot\_partial\_result(self, current\_set):  
 fig, ax = plt.subplots(figsize=(8, 8))  
 x\_coords, y\_coords = zip(\*self.points)  
 ax.scatter(x\_coords, y\_coords, color='blue', label='Points')  
 polygon\_x, polygon\_y = zip(\*[(v.point.x, v.point.y) for v in self.polygon.polygon\_vertices])  
 ax.plot(list(polygon\_x) + [polygon\_x[0]], list(polygon\_y) + [polygon\_y[0]], color='black',  
 label='Bounding Box')  
 for e in self.edges:  
 if hasattr(e, 'origin') and hasattr(e, 'next'):  
 start = e.origin.point  
 end = e.next.origin.point  
 ax.plot([start.x, end.x], [start.y, end.y], color='red')  
 else:  
 start, end = e  
 ax.plot([start[0], end[0]], [start[1], end[1]], color='red')  
 if current\_set:  
 filtered\_set = [p for p in current\_set if p is not None]  
 if filtered\_set:  
 x\_current, y\_current = zip(\*filtered\_set)  
 else:  
 x\_current, y\_current = [], []  
 ax.scatter(x\_current, y\_current, color='green', label='Current Set')  
 ax.set\_aspect('equal', adjustable='box')  
 ax.legend(loc='upper center', bbox\_to\_anchor=(0.5, -0.1), fancybox=True, shadow=True, ncol=3)  
 plt.show()  
  
 def plot\_final\_result(self):  
 fig, ax = plt.subplots(figsize=(8, 8))  
 x\_coords, y\_coords = zip(\*self.points)  
 ax.scatter(x\_coords, y\_coords, color='blue', label='Points')  
 polygon\_x, polygon\_y = zip(\*[(v.x, v.y) for v in self.polygon.polygon\_vertices])  
 ax.plot(list(polygon\_x) + [polygon\_x[0]], list(polygon\_y) + [polygon\_y[0]], color='black',  
 label='Bounding Box')  
 for e in self.edges:  
 if hasattr(e, 'origin') and hasattr(e, 'next'):  
 start = e.origin  
 end = e.next.origin  
 ax.plot([start.x, end.x], [start.y, end.y], color='red')  
 else:  
 start, end = e  
 ax.plot([start[0], end[0]], [start[1], end[1]], color='red')  
 ax.set\_aspect('equal', adjustable='box')  
 ax.legend(loc='upper center', bbox\_to\_anchor=(0.5, -0.1), fancybox=True, shadow=True, ncol=3)  
 plt.show()  
  
 def process\_big\_sites(self, current\_set, points):  
 fresh\_polygon = Polygon([  
 (v.x, v.y) for v in self.\_original\_polygon.polygon\_vertices  
 ])  
 v = Voronoi(fresh\_polygon)  
 v.create\_diagram(points=current\_set)  
 E\_b = v.edges  
  
 # Delete v memory  
 del v  
 gc.collect()  
  
 E\_b\_edge\_incident\_point = [None] \* len(E\_b)  
 for E\_b\_idx, E\_b\_edge in enumerate(E\_b):  
 incident\_point = (E\_b\_edge.incident\_point.x, E\_b\_edge.incident\_point.y)  
 E\_b\_edge\_incident\_point[E\_b\_idx] = incident\_point  
 # print(f"len(E\_b): {len(E\_b)} \nE\_b: {E\_b}")  
  
 total\_points\_number = len(points)  
 total\_iteration\_number = math.ceil(total\_points\_number / self.batch\_size)  
  
 for i in range(total\_iteration\_number):  
 start\_index = i \* self.batch\_size  
 end\_index = min(start\_index + self.batch\_size, total\_points\_number)  
 Q = points[start\_index:end\_index]  
  
 union\_set = list(dict.fromkeys(current\_set + Q))  
 fresh\_polygon = Polygon([  
 (v.x, v.y) for v in self.\_original\_polygon.polygon\_vertices  
 ])  
 v = Voronoi(fresh\_polygon)  
 v.create\_diagram(points=union\_set)  
  
 for E\_b\_idx, E\_b\_edge in enumerate(E\_b):  
 if E\_b\_edge is None:  
 continue  
 if hasattr(E\_b\_edge, 'origin') and hasattr(E\_b\_edge, 'next'):  
 x1 = E\_b\_edge.origin.x  
 y1 = E\_b\_edge.origin.y  
 x2 = E\_b\_edge.next.origin.x  
 y2 = E\_b\_edge.next.origin.y  
 else:  
 (x1, y1), (x2, y2) = E\_b\_edge  
 for idx, point in enumerate(current\_set):  
 if E\_b\_edge\_incident\_point[E\_b\_idx][0] == point[0] and E\_b\_edge\_incident\_point[E\_b\_idx][1] == point[1]:  
 cell\_edges, invalid\_cell\_flag = self.get\_cell\_edges(v, idx)  
 if invalid\_cell\_flag:  
 continue  
 intersection\_1 = None  
 intersection\_2 = None  
 if self.check\_point\_in\_cell((x1, y1), cell\_edges) and self.check\_point\_in\_cell((x2, y2), cell\_edges):  
 continue  
 for edge in cell\_edges:  
 intersection, flag\_continue, flag\_break = self.segment\_segment\_intersection(edge, E\_b\_edge)  
 if flag\_break:  
 break  
 if flag\_continue:  
 continue  
 if intersection is not None:  
 if intersection\_1 is None:  
 intersection\_1 = intersection  
 else:  
 intersection\_2 = intersection  
 break  
 if intersection\_1 is not None and intersection\_2 is not None:  
 candidate\_edge = (intersection\_1, intersection\_2)  
 E\_b[E\_b\_idx] = self.check\_edge\_direction(point, candidate\_edge)  
 elif intersection\_1 is not None and intersection\_2 is None:  
 if self.check\_point\_in\_cell((x1, y1), cell\_edges):  
 E\_b[E\_b\_idx] = self.check\_edge\_direction(point, ((x1, y1), intersection\_1))  
 else:  
 E\_b[E\_b\_idx] = self.check\_edge\_direction(point, (intersection\_1, (x2, y2)))  
 else:  
 E\_b[E\_b\_idx] = None  
  
 # Delete v memory  
 del v  
 gc.collect()  
  
 for e in E\_b:  
 if e is not None:  
 self.edges.append(e)  
  
 def process\_small\_sites(self, current\_set, current\_ray, current\_determined\_point, points, iteration\_times, phase\_number):  
 iteration\_times = [x + 1 for x in iteration\_times]  
 total\_points\_number = len(points)  
 total\_iteration\_number = math.ceil(total\_points\_number / self.batch\_size)  
  
 # Phase 1:  
 # print(f"### Phase 1: ###")  
 current\_line = [None] \* len(current\_set)  
 edge\_site\_pair = [None] \* len(current\_set)  
 nearest\_distance = [float('inf')] \* len(current\_set)  
  
 for i in range(total\_iteration\_number):  
 # print(f"\nProcessing Iteration: {i + 1}")  
 start\_index = i \* self.batch\_size  
 end\_index = min(start\_index + self.batch\_size, total\_points\_number)  
 Q = points[start\_index:end\_index]  
  
 union\_set = list(dict.fromkeys(current\_set + Q))  
 # print(f"union\_set: {union\_set}")  
 # print(f"len(union\_set): {len(union\_set)})")  
 fresh\_polygon = Polygon([  
 (v.x, v.y) for v in self.\_original\_polygon.polygon\_vertices  
 ])  
 v = Voronoi(fresh\_polygon)  
 v.create\_diagram(points=union\_set)  
  
 for idx, point in enumerate(current\_set):  
 cell\_edges, invalid\_cell\_flag = self.get\_cell\_edges(v, idx)  
 if invalid\_cell\_flag:  
 continue  
 # print(f"idx[{idx}]: cell\_edges: {cell\_edges}")  
 for edge in cell\_edges:  
 candidate\_distance = self.distance\_to\_intersection(point, current\_ray[idx], current\_determined\_point[idx], edge)  
 if candidate\_distance is not None and candidate\_distance <= nearest\_distance[idx]:  
 nearest\_distance[idx] = candidate\_distance  
 current\_line[idx] = edge  
  
 index\_i, index\_j = self.get\_site\_pair\_from\_edge(edge, points)  
 edge\_site\_pair[idx] = (index\_i, index\_j)  
 # print(f"current\_line: {current\_line}")  
 # print(f"nearest\_distance: {nearest\_distance}")  
  
 # Delete v memory  
 del v  
 gc.collect()  
  
 # print(f"current\_line: {current\_line}\n")  
  
 # print(f"### Phase 2: ###")  
 # Phase 2:  
 current\_edges = [None] \* len(current\_set)  
 first\_edge\_flag = [False] \* len(current\_set)  
 for i in range(total\_iteration\_number):  
 start\_index = i \* self.batch\_size  
 end\_index = min(start\_index + self.batch\_size, total\_points\_number)  
 Q = points[start\_index:end\_index]  
  
 union\_set = list(dict.fromkeys(current\_set + Q))  
 fresh\_polygon = Polygon([  
 (v.x, v.y) for v in self.\_original\_polygon.polygon\_vertices  
 ])  
 v = Voronoi(fresh\_polygon)  
 # fig, ax = v.create\_diagram(points=union\_set, vis\_result=False)  
 v.create\_diagram(points=union\_set)  
  
 for idx, point in enumerate(current\_set):  
 cell\_edges, invalid\_cell\_flag = self.get\_cell\_edges(v, idx)  
 if invalid\_cell\_flag:  
 continue  
 intersection\_1 = None  
 intersection\_2 = None  
 if first\_edge\_flag[idx] == False:  
 # print(f"(i == 0), cell\_edges: {cell\_edges} for idx: {idx}, point: {point}")  
 for edge in cell\_edges:  
 intersection = self.segment\_line\_intersection(edge, current\_line[idx])  
 # print(f"edge: {edge}, intersection: {intersection}")  
 if intersection is not None:  
 if intersection\_1 is None:  
 intersection\_1 = intersection  
 else:  
 intersection\_2 = intersection  
 break  
 if intersection\_1 is not None and intersection\_2 is not None:  
 candidate\_edge = (intersection\_1, intersection\_2)  
 # print(f"(i == 0), candidate\_edge: {candidate\_edge} for idx: {idx}, point: {point}")  
 current\_edges[idx] = self.check\_edge\_direction(point, candidate\_edge)  
 first\_edge\_flag[idx] = True  
 else:  
 if self.check\_point\_in\_cell(current\_edges[idx][0], cell\_edges) and self.check\_point\_in\_cell(current\_edges[idx][1], cell\_edges):  
 continue  
 for edge in cell\_edges:  
 intersection, flag\_continue, flag\_break = self.segment\_segment\_intersection(edge, current\_edges[idx])  
 if flag\_break:  
 break  
 if flag\_continue:  
 continue  
 if intersection is not None:  
 if intersection\_1 is None:  
 intersection\_1 = intersection  
 else:  
 intersection\_2 = intersection  
 break  
 # print(f"intersection\_1: {intersection\_1}, intersection\_2: {intersection\_2}")  
 if intersection\_1 is not None and intersection\_2 is not None:  
 candidate\_edge = (intersection\_1, intersection\_2)  
 current\_edges[idx] = self.check\_edge\_direction(point, candidate\_edge)  
 elif intersection\_1 is not None and intersection\_2 is None:  
 if self.check\_point\_in\_cell(current\_edges[idx][0], cell\_edges):  
 current\_edges[idx] = self.check\_edge\_direction(point, (current\_edges[idx][0], intersection\_1))  
 else:  
 current\_edges[idx] = self.check\_edge\_direction(point, (intersection\_1, current\_edges[idx][1]))  
  
 # Delete v memory  
 del v  
 gc.collect()  
  
 # print(f"iteration number: {i + 1},current\_edge: {current\_edges}")  
 # plot\_edge\_cut  
 # self.plot\_edge\_cut(current\_edges, fig, ax)  
  
 # Check if iteration time = 1, store the finish current cell flag (finish point)  
 for idx, iteration\_time in enumerate(iteration\_times):  
 if iteration\_time == 1:  
 self.finish\_flag[idx] = current\_edges[idx][0]  
  
 # Check if finished the current cell  
 for idx, current\_edge in enumerate(current\_edges):  
 if math.isclose(current\_edge[1][0], self.finish\_flag[idx][0], rel\_tol=1e-9) and math.isclose(current\_edge[1][1], self.finish\_flag[idx][1], rel\_tol=1e-9):  
 if self.upcoming\_points:  
 self.finished\_points.append(current\_set[idx])  
 next\_point = self.upcoming\_points.pop(0)  
 current\_set[idx] = next\_point  
 self.finish\_flag[idx] = None  
 iteration\_times[idx] = 0  
 current\_ray[idx] = 1  
 x, y = current\_set[idx]  
 current\_determined\_point[idx] = (x + 1, y + 1)  
 else:  
 self.finished\_points.append(current\_set[idx])  
 self.finish\_flag[idx] = None  
 current\_set[idx] = None  
 iteration\_times[idx] = 0  
 current\_ray[idx] = 1  
 else:  
 # Update ray slope & current determined point  
 new\_ray\_slope, new\_determined\_point = self.update\_ray\_slope\_and\_determined\_point(current\_set, idx, current\_edge)  
 current\_ray[idx] = new\_ray\_slope  
 current\_determined\_point[idx] = new\_determined\_point  
  
 if phase\_number == 2:  
 for idx, edge in enumerate(current\_edges):  
 index\_i, index\_j = edge\_site\_pair[idx]  
 if index\_j is None:  
 self.edges.append(edge)  
 i\_small = index\_i in self.small\_sites\_index  
 j\_small = index\_j in self.small\_sites\_index  
 j\_big = index\_j in self.big\_sites\_index  
  
 if i\_small and j\_small:  
 if index\_i < index\_j:  
 self.edges.append(edge)  
  
 if i\_small and j\_big:  
 self.edges.append(edge)  
 # self.plot\_partial\_result(current\_set)  
 # print(f"current\_edges: {current\_edges}")  
 # print(f"current\_set: {current\_set}, current\_ray: {current\_ray}, current\_determined\_point: {current\_determined\_point}, iteration\_times: {iteration\_times}")  
 return current\_set, current\_ray, current\_determined\_point, iteration\_times  
  
 def get\_cell\_edges(self, voronoi\_diagram, idx, tol=1e-9):  
 matching\_point = voronoi\_diagram.sites[idx]  
 edges = []  
  
 start\_edge = matching\_point.first\_edge  
 current\_edge = start\_edge  
  
 if (  
 start\_edge is None or  
 start\_edge.next is None or  
 start\_edge.origin is None or  
 start\_edge.next.origin is None  
 ):  
 return None, True  
  
 visited\_ids = set()  
  
 while current\_edge:  
 edge\_id = id(current\_edge)  
 if edge\_id in visited\_ids:  
 break  
 visited\_ids.add(edge\_id)  
  
 if (  
 current\_edge.origin is None or  
 current\_edge.next is None or  
 current\_edge.next.origin is None  
 ):  
 return None, True  
  
 x1 = current\_edge.origin.x  
 y1 = current\_edge.origin.y  
 x2 = current\_edge.next.origin.x  
 y2 = current\_edge.next.origin.y  
  
 dx = x2 - x1  
 dy = y2 - y1  
 length\_squared = dx \* dx + dy \* dy  
  
 if length\_squared > tol \* tol:  
 edges.append(current\_edge)  
  
 current\_edge = current\_edge.next  
  
 if current\_edge is start\_edge:  
 break  
  
 if len(edges) < 3 or current\_edge is not start\_edge:  
 return edges, True  
  
 return edges, False  
  
 def distance\_to\_intersection(self, point, current\_ray, current\_determined\_point, edge):  
 px, py = point  
 cdp\_x, cdp\_y = current\_determined\_point  
  
 if hasattr(edge, 'origin') and hasattr(edge, 'next'):  
 sp\_x = edge.origin.x  
 sp\_y = edge.origin.y  
 ep\_x = edge.next.origin.x  
 ep\_y = edge.next.origin.y  
 else:  
 (sp\_x, sp\_y), (ep\_x, ep\_y) = edge  
  
 if ep\_x != sp\_x:  
 edge\_slope = (ep\_y - sp\_y) / (ep\_x - sp\_x)  
 if current\_ray == edge\_slope:  
 return None  
 intersection\_x = (sp\_y - py + px \* current\_ray - sp\_x \* edge\_slope) / (current\_ray - edge\_slope)  
 else:  
 intersection\_x = sp\_x  
 intersection\_y = current\_ray \* (intersection\_x - px) + py  
 dx = cdp\_x - px  
 dy = cdp\_y - py  
 if dx != 0:  
 if (intersection\_x - px) \* dx < 0:  
 return None  
 else:  
 if (intersection\_y - py) \* dy < 0:  
 return None  
 distance = math.sqrt((px - intersection\_x) \*\* 2 + (py - intersection\_y) \*\* 2 )  
 return distance  
  
 def segment\_line\_intersection(self, segment, line, tol=1e-9):  
 if hasattr(segment, 'origin') and hasattr(segment, 'next'):  
 x1 = segment.origin.x  
 y1 = segment.origin.y  
 x2 = segment.next.origin.x  
 y2 = segment.next.origin.y  
 else:  
 (x1, y1), (x2, y2) = segment  
  
 if hasattr(line, 'origin') and hasattr(line, 'next'):  
 p1 = line.origin.x  
 q1 = line.origin.y  
 p2 = line.next.origin.x  
 q2 = line.next.origin.y  
 else:  
 (p1, q1), (p2, q2) = line  
  
 dx1, dy1 = x2 - x1, y2 - y1  
 dx2, dy2 = p2 - p1, q2 - q1  
  
 denominator = dx1 \* dy2 - dy1 \* dx2  
 if abs(denominator) < tol:  
 return None  
  
 t = ((p1 - x1) \* dy2 - (q1 - y1) \* dx2) / denominator  
 if t < 0 - tol or t > 1 + tol:  
 return None  
  
 intersection\_x = x1 + t \* dx1  
 intersection\_y = y1 + t \* dy1  
 return (intersection\_x, intersection\_y)  
  
 def segment\_segment\_intersection(self, segment1, segment2, tol = 1e-9):  
 if hasattr(segment1, 'origin') and hasattr(segment1, 'next'):  
 x1 = segment1.origin.x  
 y1 = segment1.origin.y  
 x2 = segment1.next.origin.x  
 y2 = segment1.next.origin.y  
 else:  
 (x1, y1), (x2, y2) = segment1  
  
 if hasattr(segment2, 'origin') and hasattr(segment2, 'next'):  
 p1 = segment2.origin.x  
 q1 = segment2.origin.y  
 p2 = segment2.next.origin.x  
 q2 = segment2.next.origin.y  
 else:  
 (p1, q1), (p2, q2) = segment2  
  
 # Check the Special Situation  
 # cond1 = (abs((x1 - p1) \* (q2 - q1) - (y1 - q1) \* (p2 - p1)) < tol and  
 # min(p1, p2) - tol <= x1 <= max(p1, p2) + tol and  
 # min(q1, q2) - tol <= y1 <= max(q1, q2) + tol)  
 #  
 # cond2 = (abs((x2 - p1) \* (q2 - q1) - (y2 - q1) \* (p2 - p1)) < tol and  
 # min(p1, p2) - tol <= x2 <= max(p1, p2) + tol and  
 # min(q1, q2) - tol <= y2 <= max(q1, q2) + tol)  
 #  
 # flag\_continue = cond1 and cond2  
  
 cond3 = (abs((p1 - x1) \* (y2 - y1) - (q1 - y1) \* (x2 - x1)) < tol and  
 min(x1, x2) - tol <= p1 <= max(x1, x2) + tol and  
 min(y1, y2) - tol <= q1 <= max(y1, y2) + tol)  
  
 cond4 = (abs((p2 - x1) \* (y2 - y1) - (q2 - y1) \* (x2 - x1)) < tol and  
 min(x1, x2) - tol <= p2 <= max(x1, x2) + tol and  
 min(y1, y2) - tol <= q2 <= max(y1, y2) + tol)  
  
 flag\_break = cond3 and cond4  
  
 dx1 = x2 - x1  
 dy1 = y2 - y1  
 dx2 = p2 - p1  
 dy2 = q2 - q1  
  
 cond\_collinear = abs((x1 - p1) \* dy2 - (y1 - q1) \* dx2) < tol and abs(dx1 \* dy2 - dy1 \* dx2) < tol  
 flag\_continue = cond\_collinear  
  
 denominator = dx1 \* dy2 - dy1 \* dx2  
 if denominator == 0:  
 return None, False, False  
  
 t = ((p1 - x1) \* dy2 - (q1 - y1) \* dx2) / denominator  
 u = ((p1 - x1) \* dy1 - (q1 - y1) \* dx1) / denominator  
  
 if not (0 - tol <= t <= 1 + tol and 0 - tol <= u <= 1 + tol):  
 return None, False, False  
  
 intersection\_x = x1 + t \* dx1  
 intersection\_y = y1 + t \* dy1  
 return (intersection\_x, intersection\_y), flag\_continue, flag\_break  
  
 def check\_point\_in\_cell(self, point, edges, tol=1e-9):  
 x, y = point  
 ref = None  
  
 for edge in edges:  
  
 if hasattr(edge, 'origin') and hasattr(edge, 'next'):  
 x1 = edge.origin.x  
 y1 = edge.origin.y  
 x2 = edge.next.origin.x  
 y2 = edge.next.origin.y  
 else:  
 print(edge)  
 (x1, y1), (x2, y2) = edge  
  
 dx = x2 - x1  
 dy = y2 - y1  
 dxp = x - x1  
 dyp = y - y1  
 cross = dx \* dyp - dy \* dxp  
  
 if abs(cross) < tol:  
 continue  
  
 if ref is None:  
 ref = 1 if cross > 0 else -1  
 else:  
 if (cross > 0 and ref < 0) or (cross < 0 and ref > 0):  
 return False  
 return True  
  
 def plot\_edge\_cut(self, current\_edge, fig, ax):  
 for edge in current\_edge:  
 if edge is not None:  
 (start, end) = edge  
 if start is not None and end is not None:  
 ax.plot(  
 [start[0], end[0]],  
 [start[1], end[1]],  
 linestyle='--',  
 color='purple',  
 linewidth=3,  
 label='Edge Cut' if edge == current\_edge[0] else None # 只添加一次图例  
 )  
 handles, labels = ax.get\_legend\_handles\_labels()  
 if 'Edge Cut' in labels:  
 ax.legend(loc='upper right')  
 fig.canvas.draw()  
 plt.show()  
  
 def update\_ray\_slope\_and\_determined\_point(self, current\_set, idx, current\_edge, tol = 1e-9):  
 current\_point = current\_set[idx]  
 start\_point = current\_edge[0]  
 end\_point = current\_edge[1]  
 current\_point\_x, current\_point\_y = current\_point  
 start\_point\_x, start\_point\_y = start\_point  
 end\_point\_x, end\_point\_y = end\_point  
  
 temp = 0.000001  
  
 if abs(start\_point\_x - end\_point\_x) < tol and abs(start\_point\_y - end\_point\_y) < tol:  
 temp = 0.1  
 if current\_point\_x == end\_point\_x:  
 if end\_point\_y < current\_point\_y:  
 determined\_point\_x = end\_point\_x - temp  
 else:  
 determined\_point\_x = end\_point\_x + temp  
 determined\_point\_y = end\_point\_y  
 ray\_slope = (current\_point\_y - determined\_point\_y) / (current\_point\_x - determined\_point\_x)  
 return ray\_slope, (determined\_point\_x, determined\_point\_y)  
 elif (current\_point\_x < end\_point\_x and current\_point\_y < end\_point\_y) or (current\_point\_x < end\_point\_x and current\_point\_y > end\_point\_y):  
 ray\_slope = (current\_point\_y - end\_point\_y) / (current\_point\_x - end\_point\_x)  
 ray\_slope -= temp  
 determined\_point\_x = current\_point\_x + 1  
 determined\_point\_y = ray\_slope \* (determined\_point\_x - current\_point\_x) + current\_point\_y  
 return ray\_slope, (determined\_point\_x, determined\_point\_y)  
 elif (current\_point\_x > end\_point\_x and current\_point\_y > end\_point\_y) or (current\_point\_x > end\_point\_x and current\_point\_y < end\_point\_y):  
 ray\_slope = (current\_point\_y - end\_point\_y) / (current\_point\_x - end\_point\_x)  
 ray\_slope -= temp  
 determined\_point\_x = current\_point\_x - 1  
 determined\_point\_y = ray\_slope \* (determined\_point\_x - current\_point\_x) + current\_point\_y  
 return ray\_slope, (determined\_point\_x, determined\_point\_y)  
 elif current\_point\_y == end\_point\_y and current\_point\_x < end\_point\_x:  
 ray\_slope = -temp  
 determined\_point\_x = current\_point\_x + 1  
 determined\_point\_y = ray\_slope \* (determined\_point\_x - current\_point\_x) + current\_point\_y  
 return ray\_slope, (determined\_point\_x, determined\_point\_y)  
 elif current\_point\_y == end\_point\_y and current\_point\_x > end\_point\_x:  
 ray\_slope = -temp  
 determined\_point\_x = current\_point\_x - 1  
 determined\_point\_y = ray\_slope \* (determined\_point\_x - current\_point\_x) + current\_point\_y  
 return ray\_slope, (determined\_point\_x, determined\_point\_y)  
  
 def check\_edge\_direction(self, point, edge):  
 (ax, ay) = point  
 (bx, by), (cx, cy) = edge  
  
 cross = (bx - ax) \* (cy - ay) - (by - ay) \* (cx - ax)  
  
 if cross < 0:  
 return ((bx, by), (cx, cy))  
 else:  
 return ((cx, cy), (bx, by))  
  
 def get\_site\_pair\_from\_edge(self, edge, points):  
 site\_i\_point = edge.incident\_point  
 site\_j\_point = edge.twin.incident\_point  
 index\_i = None  
 index\_j = None  
 if site\_i\_point is not None:  
 site\_i\_point\_x, site\_i\_point\_y = site\_i\_point.x, site\_i\_point.y  
 for idx, point in enumerate(points):  
 if point[0] == site\_i\_point\_x and point[1] == site\_i\_point\_y:  
 index\_i = idx  
 break  
 if site\_j\_point is not None:  
 site\_j\_point\_x, site\_j\_point\_y = site\_j\_point.x, site\_j\_point.y  
 for idx, point in enumerate(points):  
 if point[0] == site\_j\_point\_x and point[1] == site\_j\_point\_y:  
 index\_j = idx  
 break  
 return index\_i, index\_j