BÀI 4: CÁC THUẬT TOÁN TÌM KIẾM (tiếp theo)

I. MỤC TIÊU:

Sau khi thực hành xong bài này, sinh viên:

- Áp dụng các thuật toán tìm kiếm vào các bài toán thực tế.

II. TÓM TẮT LÝ THUYẾT:

1. Thuật toán BFS:

2. Thuật toán DFS:

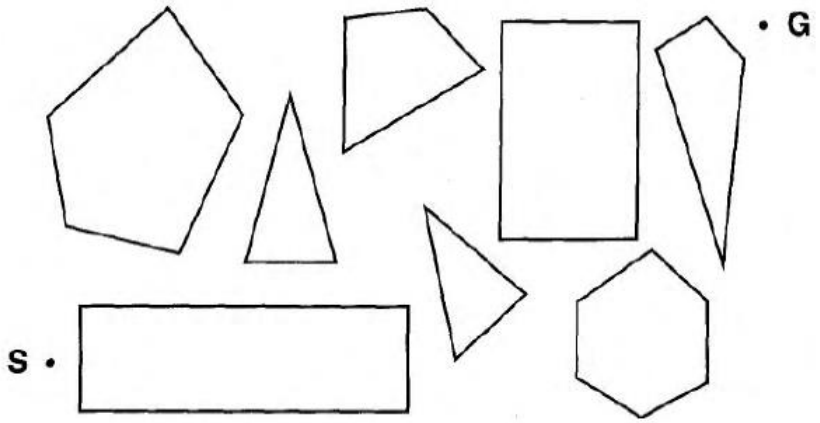
3. Thuật toán UCS:

4. Thuật toán Greedy – Best – First Search:

5. Thuật toán A^*:

III. NỘI DUNG THỰC HÀNH:

1. Xét bài toán tìm đường đi ngắn nhất từ điểm S tới điểm G trong một mặt phẳng có các chướng vật là những đa giác lồi như hình.



a) Giả sử không gian trạng thái chứa tất cả các vị trí (x, y) nằm trong mặt phẳng. Có bao nhiêu trạng thái ở đây? Có bao nhiêu đường đi từ đỉnh xuất phát tới đỉnh đích?

b) Giải thích ngắn gọn vì sao đường đi ngắn nhất từ một đỉnh của đa giác tới một đỉnh khác trong mặt phẳng nhất định phải bao gồm các đoạn thẳng nối một số đỉnh của các đa giác?Hãy định nghĩa lại không gian trạng thái. Không gian trạng thái này sẽ lớn bao nhiêu?

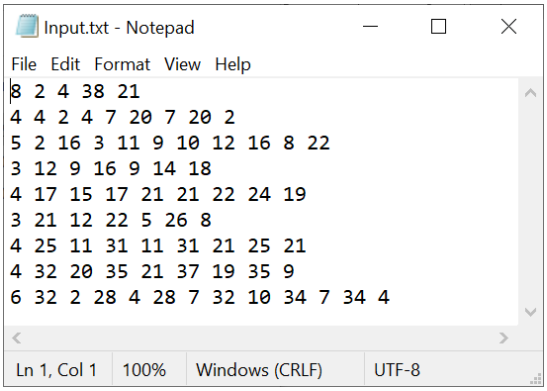
c) Định nghĩa các hàm cần thiết để thực thi bài toán tìm kiếm, bao gồm hàm successor nhận một đỉnh làm đầu vào và trả về tập đỉnh có thể đi đến được từ đỉnh đó trong vòng 1 bước.

⇒ Hàm heristic được sử dụng là khoảng cách Euclide.

d) Áp dụng thuật toán tìm kiếm để giải bài toán.

2. **Dữ liệu đầu vào**

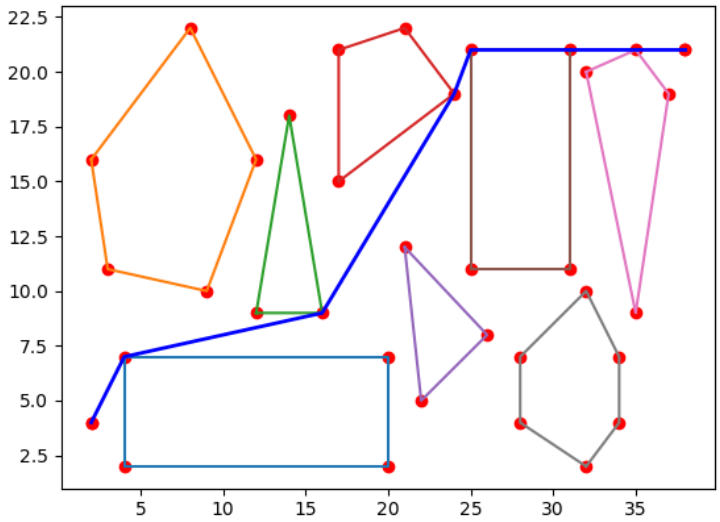
- Dòng 1: N S_x S_y G_x G_y
  - N là số đa giác nằm trong mặt phẳng ($0 \leq N \leq 100$)
  - (S_x, S_y) là tọa độ đỉnh xuất phát, (G_x, G_y): tọa độ đỉnh đích
- N dòng tiếp theo: M X_1 Y_1 … X_M Y_M
  - M là số đỉnh của đa giác ($3 \leq M \leq 10$)
  - (X_i, Y_i): tọa độ thực của đỉnh thứ i trong đa giác.



3. **Dữ liệu đầu ra**: đường đi ngắn nhất từ đỉnh xuất phát S đến đỉnh đích G

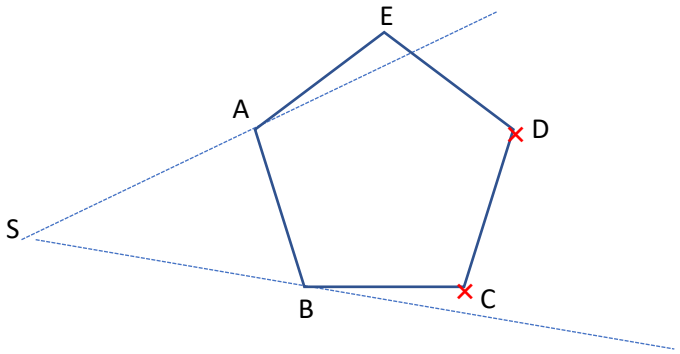
$$(x_0, y_0, S) \rightarrow (x_1, y_1, p_1) \rightarrow \cdots \rightarrow (x_{n-1}, y_{n-1}, p_{n-1}) \rightarrow (x_n, y_n, G)$$

với (x_i, y_i, p_i) là thông tin của một đỉnh trên đường đi, bao gồm tọa độ (x_i, y_i) của các đỉnh và nó thuộc về đa giác p_i.



4. **Xác định những đỉnh có thể đi qua được**, tức là từ S có thể nhìn thấy những đỉnh nào trên bản đồ.

Ý tưởng: Từ S (hay là từ một đỉnh đang xét bất kì) và các cạnh AB của các đa giác, những đỉnh nằm trong cung ASB sẽ bị loại ⇒ những điểm không bị loại là đỉnh nhìn thấy được.



⇒ E là đỉnh nhìn thấy, C và D là 2 đỉnh không nhìn thấy.

Nhắc lại:

- Cho 2 điểm A(x_1, y_1), B(x_2, y_2). Phương trình đường thẳng tạo bởi đoạn thẳng AB có dạng

$$d: \frac{x - x_1}{x_2 - x_1} = \frac{y - y_1}{y_2 - y_1}$$

- Xét vị trí tương đối của C(x_3, y_3) và D(x_4, y_4) đối với đường thẳng d:
  - $d(C) * d(D) \geq 0 \Rightarrow$ C, D nằm cùng phía.
  - $d(C) * d(D) < 0 \Rightarrow$ C, D nằm khác phía.

Các bước thực hiện

- Gọi P là một đỉnh đang xét, V là tập cạnh của tất cả các đa giác
- Với mỗi cạnh đa giác, gọi là AB, trong tập V:
  - Tạo d_1 từ P và A, d_2 từ P và B, d_3 từ A và B
  - Xét tất cả các đỉnh Q còn lại với d_1, d_2, d_3
    - Nếu $d_1(Q) * d_1(B) \geq 0$ và $d_2(Q) * d_2(A) \geq 0$ và $d_3(Q) * d_3(P) < 0$ thì Q là đỉnh không nhìn thấy được từ P.
    - Ngược lại, nhìn thấy được từ P.

6. Cài đặt:

```
from collections import defaultdict
from queue import PriorityQueue
import math
from matplotlib import pyplot as plt


class Point(object):
    def __init__(self, x, y, polygon_id=-1):
        self.x = x
        self.y = y
        self.polygon_id = polygon_id
        self.g = 0
        self.pre = None

    def rel(self, other, line):
        return line.d(self) * line.d(other) >= 0

    def can_see(self, other, line):
        l1 = self.line_to(line.p1)
        l2 = self.line_to(line.p2)
        d3 = line.d(self) * line.d(other) < 0
        d1 = other.rel(line.p2, l1)
        d2 = other.rel(line.p1, l2)
        return not (d1 and d2 and d3)

    def line_to(self, other):
        return Edge(self, other)

    def heuristic(self, other):
        return euclid_distance(self, other)

    def __eq__(self, point):
        return point and self.x == point.x and self.y == point.y

    def __ne__(self, point):
        return not self.__eq__(point)

    def __lt__(self, point):

        return hash(self) < hash(point)
    def __str__(self):
        return "(%d, %d)" % (self.x, self.y)

    def __hash__(self):
        return self.x.__hash__() ^ self.y.__hash__()

    def __repr__(self):
        return "(%d, %d)" % (self.x, self.y)


class Edge(object):
    def __init__(self, point1, point2):
        self.p1 = point1
        self.p2 = point2

    def get_adjacent(self, point):
        if point == self.p1:
            return self.p2
        if point == self.p2:
            return self.p1

    def d(self, point):
        vect_a = Point(self.p2.x - self.p1.x, self.p2.y - self.p1.y)
        vect_n = Point(-vect_a.y, vect_a.x)
        return vect_n.x * (point.x - self.p1.x) + vect_n.y * (point.y - self.p1.y)

    def __str__(self):
        return "({},{})".format(self.p1, self.p2)

    def __contains__(self, point):
        return self.p1 == point or self.p2 == point

    def __hash__(self):
        return self.p1.__hash__() ^ self.p2.__hash__()

    def __repr__(self):
        return "Edge({!r}, {!r})".format(self.p1, self.p2)
```

```
class Graph:
    def __init__(self, polygons):
        self.graph = defaultdict(set)
        self.edges = set()
        self.polygons = defaultdict(set)
        pid = 0
        for polygon in polygons:
            if len(polygon) == 2:
                polygon.pop()
            if polygon[0] == polygon[-1]:
                self.add_point(polygon[0])
            else:
                for i, point in enumerate(polygon):
                    neighbor_point = polygon[(i + 1) % len(polygon)]
                    edge = Edge(point, neighbor_point)
                    if len(polygon) > 2:
                        point.polygon_id = pid
                        neighbor_point.polygon_id = pid
                        self.polygons[pid].add(edge)
                    self.add_edge(edge)
                if len(polygon) > 2:
                    pid += 1

    def get_adjacent_points(self, point):
        return list(filter(None.__ne__, [edge.get_adjacent(point) for edge in self.edges]))

    def can_see(self, start):
        see_list = list()
        cant_see_list = list()

        for polygon in self.polygons:
            for edge in self.polygons[polygon]:
                for point in self.get_points():
                    if start == point:
                        cant_see_list.append(point)
                    if start in self.get_polygon_points(polygon):
                        for poly_point in self.get_polygon_points(polygon):
                            if poly_point not in self.get_adjacent_points(start):
                                cant_see_list.append(poly_point)
                    if point not in cant_see_list:
                        if start.can_see(point, edge):
                            if point not in see_list:
                                see_list.append(point)
                        elif point in see_list:
                            see_list.remove(point)
                            cant_see_list.append(point)
                        else:
                            cant_see_list.append(point)
        return see_list

    def get_polygon_points(self, index):
        point_set = set()
        for edge in self.polygons[index]:
            point_set.add(edge.p1)
            point_set.add(edge.p2)
        return point_set

    def get_points(self):
        return list(self.graph)

    def get_edges(self):
        return self.edges

    def add_point(self, point):
        self.graph[point].add(point)

    def add_edge(self, edge):
        self.graph[edge.p1].add(edge)
```

```
            self.graph[edge.p1].add(edge)
            self.graph[edge.p2].add(edge)
            self.edges.add(edge)

        def __contains__(self, item):
            if isinstance(item, Point):
                return item in self.graph
            if isinstance(item, Edge):
                return item in self.edges
            return False

        def __getitem__(self, point):
            if point in self.graph:
                return self.graph[point]
            return set()

        def __str__(self):
            res = ""
            for point in self.graph:
                res += "\n" + str(point) + ": "
                for edge in self.graph[point]:
                    res += str(edge)
            return res

        def __repr__(self):
            return self.__str__()

        def h(self, point):
            heuristic = getattr(self, 'heuristic', None)
            if heuristic:
                return heuristic[point]
            else:
                return -1


    def euclid_distance(point1, point2):
        return round(float(math.sqrt((point2.x - point1.x)**2 + (point2.y - point1.y)**2)), 3)

    def search(graph, start, goal, func):
        closed = set()
        queue = PriorityQueue()
        queue.put((0 + func(graph, start), start))
        if start not in closed:
            closed.add(start)
        while not queue.empty():
            cost, node = queue.get()
            if node == goal:
                return node
            for i in graph.can_see(node):
                new_cost = node.g + euclid_distance(node, i)
                if i not in closed or new_cost < i.g:
                    closed.add(i)
                    i.g = new_cost
                    i.pre = node
                    new_cost = func(graph, i)
                    queue.put((new_cost, i))
        return node

    a_star = lambda graph, i: i.g + graph.h(i)
    greedy = lambda graph, i: graph.h(i)
```

```
def main():
    n_polygon = 0
    poly_list = list(list())
    x = list()
    y = list()
    with open('Input.txt', 'r') as f:
        line = f.readline()
        line = line.strip()
        line = line.split()
        line = list(map(int, line))
        n_polygon = line[0]
        start = Point(line[1], line[2])
        goal = Point(line[3], line[4])
        poly_list.append([start])
        for line in f:
            point_list = list()
            line = line.split()
            n_vertex = int(line[0])
            for j in range(0, 2*n_vertex, 2):
                point_list.append(Point(int(line[j + 1]), int(line[j + 2])))
            poly_list.append(point_list[:])
        poly_list.append([goal])
        graph = Graph(poly_list)
        graph.heuristic = {point: point.heuristic(goal) for point in graph.get_points()}

        a=search(graph, start, goal, a_star)

        result = list()

        while a:
            result.append(a)
            a = a.pre
        result.reverse()
        print_res = [[point, point.polygon_id] for point in result]
        print(*print_res, sep=' -> ')
        plt.figure()
        plt.plot([start.x], [start.y], 'ro')
        plt.plot([goal.x], [goal.y], 'ro')

        for point in graph.get_points():
            x.append(point.x)
            y.append(point.y)
        plt.plot(x,y,'ro')
        for i in range(1,len(poly_list) - 1):
            coord = list()
            for point in poly_list[i]:
                coord.append([point.x, point.y])
            coord.append(coord[0])
            xs, ys = zip(*coord)  # create lists of x and y values
            plt.plot(xs, ys)
        x = list()
        y = list()
        for point in result:
            x.append(point.x)
            y.append(point.y)
        plt.plot(x, y, 'b', linewidth=2.0)
        plt.show()


if __name__ == "__main__":
    main()
```

7. **Yêu cầu**:
   - Cài đặt và thực thi chương trình. Nếu chương trình bị báo lỗi thì lỗi ở dòng nào và sửa lại như thế nào?
   - Áp dụng bài toán với các thuật toán BFS, DFS và UCS.
   - Viết báo cáo trình bày lại tất cả những gì em hiểu liên quan tới bài thực hành. Nhận xét?