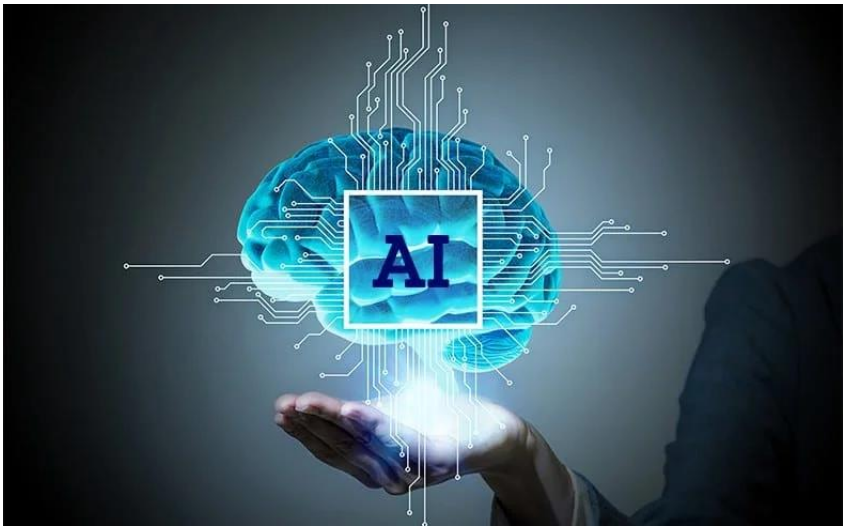
BỘ GIÁO DỤC VÀ ĐÀO TẠO

TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH

KHOA TOÁN - TIN HỌC

BÀI TẬP THỰC HÀNH TUẦN 4

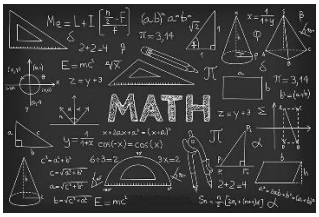


MÔN HỌC: **Nhập môn AI**

Sinh Viên: **Trần Công Hiếu - 21110294**

Lớp: **21TTH_KDL**

TP.HCM, ngày 27 tháng 11 năm 2023



TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN TP.HCM
KHOA TOÁN – TIN HỌC

BÀI BÁO CÁO BÀI TẬP THỰC HÀNH TUẦN 4

HK1 - NĂM HỌC: 2023-2024

MÔN: NHẬP MÔN TRÍ TUỆ NHÂN TẠO

SINH VIÊN: TRẦN CÔNG HIẾU

MSSV: 21110294

LỚP: 21TTH_KDL

NHẬN XÉT

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

Tp.HCM, Ngày……Tháng……Năm 2021

Giảng viên Bộ môn

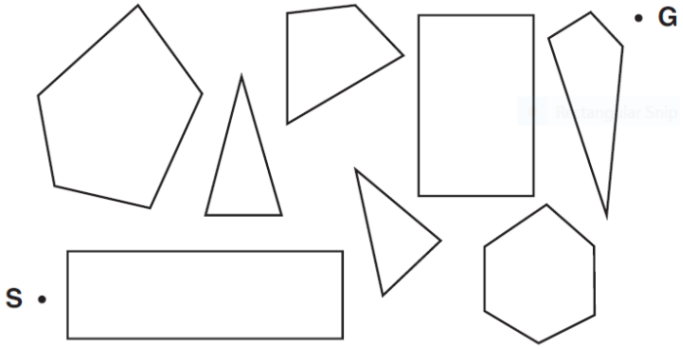
MỤC LỤC

I. CÀI ĐẶT VÀ PHÁT HIỆN LỖI.

- Không có lỗi nào được phát hiện trong lúc cài đặt 👌☺.

II. TRÌNH BÀY CHI TIẾT.

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.




```
1  from collections import defaultdict
2  from queue import PriorityQueue
3  import math
4  from matplotlib import pyplot as plt
```

"from collections import defaultdict": Thư viện này cho phép tạo ra các từ điển với các giá trị mặc định cho các khóa không có trong từ điển. Điều này hữu ích khi xử lý cấu trúc đồ thị, trong đó mỗi nút có thể có nhiều cạnh.

"from queue import PriorityQueue": Thư viện này cung cấp một cài đặt của hàng đợi ưu tiên, thường được sử dụng trong các thuật toán như Dijkstra hoặc A*.

"import math": Thư viện này cung cấp các hàm toán học. Trong ngữ cảnh này, có thể sử dụng để tính toán toán học liên quan đến các thuật toán đồ thị, như tính khoảng cách hoặc các phép toán toán học khác.

"from matplotlib import pyplot as plt": Thư viện này được sử dụng để vẽ đồ thị và các biểu đồ. Nó cung cấp một khung việc vẽ tương tự như MATLAB và thường được sử dụng để trực quan hóa dữ liệu trong Python.

1. Trình bày Class Point.

```
6   class Point(object):
7       def __init__(self, x, y, polygon_id=-1):
8           self.x = x
9           self.y = y
10          self.polygon_id = polygon_id
11          self.g = 0
12          self.pre = None
13
```

Định nghĩa một lớp Point. Lớp này dùng để tạo ra các đối tượng điểm có các thuộc tính như x, y, polygon_id, g, và pre.

x là hoành độ của điểm.

y: Là tung độ của điểm.

polygon_id: Là một định danh (ID) của đa giác mà điểm thuộc về. Giá trị mặc định là -1 nếu không được cung cấp.

g: Là một giá trị số, được sử dụng để lưu trữ chi phí (cost). Giá trị ban đầu được đặt là 0.

pre: Là một tham chiếu đến một điểm trước đó, được sử dụng để theo dõi điểm trước đó trên đường đi tốt nhất.

Phương thức "def __init__(self, x, y, polygon_id=-1):" Phương thức khởi tạo của lớp Point, nhận vào các tham số x, y là hoành độ và tung độ tương ứng của điểm và polygon_id để xác định đa giác mà điểm thuộc về. Nếu không được cung cấp, polygon_id sẽ có giá trị mặc định là -1. Phương thức này cũng khởi tạo các thuộc tính g và pre của điểm.

Lớp Point này có thể được sử dụng để biểu diễn các điểm trong không gian hai chiều (2D) và lưu trữ thông tin về vị trí, thuộc tính đa giác và các giá trị liên quan đến việc tìm đường đi hoặc xử lý đa giác.

```
    4 usages (4 dynamic)
14  def rel(self, other, line):
15      return line.d(self) * line.d(other) >= 0
16
```

Phương thức rel() trong lớp Point nhận vào hai tham số other (điểm khác) và line (đường thẳng) để kiểm tra mối quan hệ vị trí giữa điểm hiện tại và một điểm khác other đối với một đường thẳng line.

line.d(self) và line.d(other): Đây là hai phương thức hoặc thuộc tính của đối tượng line, trả về một giá trị số đại diện cho vị trí tương đối của hai điểm đang xét so với đường thẳng. Ở đây là trường hợp hai điểm đang xét nằm cùng nằm về một phía so với đường thẳng.

Tóm lại, phương thức này trả về giá trị boolean (True hoặc False) để xác định liệu hai điểm (đối tượng self và other) có cùng nằm một phía với một đường thẳng line hay không.

```
4 usages (4 dynamic)
17  def can_see(self, other, line):
18      l1 = self.line_to(line.p1)
19      l2 = self.line_to(line.p2)
20      d3 = line.d(self) * line.d(other) < 0
21      d1 = other.rel(line.p2, l1)
22      d2 = other.rel(line.p1, l2)
23      return not (d1 and d2 and d3)
24
```

Phương thức can_see() trong lớp Point nhận ba tham số:

other: Điểm cần kiểm tra xem có thể nhìn thấy từ điểm hiện tại không.

line: Đường thẳng đại diện cho tầm nhìn hoặc đường thẳng mà điểm hiện tại phải xem xét để xác định có thể nhìn thấy other hay không.

l1 = self.line_to(line.p1): Tạo một đường thẳng từ điểm hiện tại đến line.p1.

l2 = self.line_to(line.p2): Tạo một đường thẳng từ điểm hiện tại đến line.p2.

d3 = line.d(self) * line.d(other) < 0: Kiểm tra xem điểm hiện tại và other có cùng nằm ở hai phía của đường thẳng line hay không.

d1 = other.rel(line.p2, l1): Kiểm tra mối quan hệ vị trí giữa object other và line.p2 dựa trên đường thẳng l1 thông qua phương thức rel(). Kiểm tra thử điểm other và điểm p2 có nằm về cùng một phía so với đường thẳng l1 hay không.

d2 = other.rel(line.p1, l2): Kiểm tra mối quan hệ vị trí giữa object other và line.p1 dựa trên đường thẳng l2 thông qua phương thức rel(). Kiểm

tra thử điểm other và điểm p1 có nằm về cùng một phía so với đường thẳng l2 hay không.

“return not (d1 and d2 and d3)”: Trả về giá trị True nếu không có điều kiện nào cả ba điều kiện d1, d2, và d3 đều đúng. Ngược lại, trả về False.

Tổng thể, hàm này xác định xem từ điểm hiện tại có thể nhìn thấy điểm other trên đường thẳng line không. Nó kiểm tra ba điều kiện: điểm hiện tại và other có cùng nằm hai phía của đường thẳng line hay không, và other có thể nhìn thấy line.p1 và line.p2 không. Nếu cả ba điều kiện đều đúng, nghĩa là điểm hiện tại không thể nhìn thấy other, và hàm trả về False.

```
    2 usages
25      def line_to(self, other):
26          return Edge(self, other)
        2 usages (2 dynamic)
27      def heuristic(self, other):
28          return euclid_distance(self, other)
29      def __eq__(self, point):
30          return point and self.x == point.x and self.y == point.y
31      def __ne__(self, point):
32          return not self.__eq__(point)
33      def __lt__(self, point):
34          return hash(self) < hash(point)
35      def __str__(self):
36          return "(%d, %d)" % (self.x, self.y)
37      def __hash__(self):
38          return self.x.__hash__() ^ self.y.__hash__()
39      def __repr__(self):
40          return "(%d, %d)" % (self.x, self.y)
41
```

“line_to(self, other)”: Trả về một đối tượng Edge đại diện cho cạnh giữa điểm hiện tại và điểm other.

“heuristic(self, other)”: Trả về khoảng cách Euclid giữa điểm hiện tại và điểm other. Hàm được định nghĩa ở đoạn mã phía dưới, tính bằng khoảng cách Euclid giữa 2 điểm self và other.

“def __eq__(self, point):”: Xác định toán tử so sánh bằng (==). Trả về True nếu self có cùng tọa độ với point, ngược lại trả về False.

“def __ne__(self, point):”: Xác định toán tử không bằng (!=). Trả về True nếu self không có cùng tọa độ với point, ngược lại trả về False.

“def __lt__(self, point):”: Xác định toán tử "nhỏ hơn" (<). Sử dụng trong so sánh không gian băm (hashing) của các đối tượng Point.

“def __str__(self):”: Trả về một chuỗi đại diện cho đối tượng Point, chứa thông tin về hoành độ và tung độ.

“def __hash__(self):”: Xác định hàm băm cho đối tượng Point, sử dụng hoành độ và tung độ để tạo một giá trị duy nhất.

“def __repr__(self):”: Trả về một chuỗi biểu diễn cho đối tượng Point, chứa thông tin về hoành độ và tung độ.

2. Trình bày Class Edge.

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

    2 usages (2 dynamic)
    def get_adjacent(self, point):
        if point == self.p1:
            return self.p2
        if point == self.p2:
            return self.p1
```

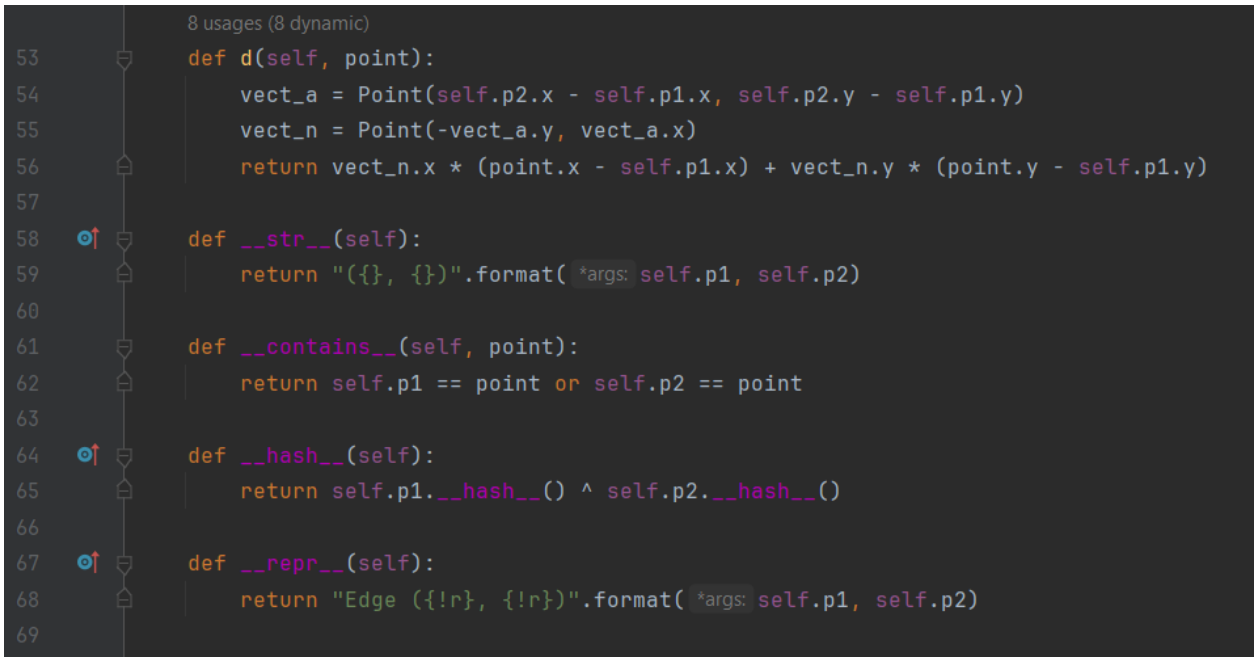
Lớp Edge trong đoạn mã trên định nghĩa một cạnh (đoạn thẳng) dựa trên hai điểm.

Phương thức khởi tạo __init__: Nhận hai đối tượng Point làm đối số và gán chúng cho hai thuộc tính p1 và p2 của đối tượng Edge.

Phương thức get_adjacent: Nhận một đối tượng Point làm đối số và trả về điểm kế cận (adjacent) của nó trên cạnh. Nếu điểm đầu vào (point) là một trong hai đầu mút của cạnh (self.p1 hoặc self.p2), phương thức sẽ trả về đầu mút còn lại của cạnh.

Phương thức get_adjacent có chức năng để truy vấn điểm kế cận của một điểm đối với cạnh. Nó kiểm tra xem điểm đầu vào có phải là một trong

hai điểm đầu mút của cạnh hay không, và nếu đúng, trả về điểm còn lại trên cạnh đó.



```
8 usages (8 dynamic)
53      def d(self, point):
54          vect_a = Point(self.p2.x - self.p1.x, self.p2.y - self.p1.y)
55          vect_n = Point(-vect_a.y, vect_a.x)
56          return vect_n.x * (point.x - self.p1.x) + vect_n.y * (point.y - self.p1.y)
57
58      def __str__(self):
59          return "({}, {})".format( *args: self.p1, self.p2)
60
61      def __contains__(self, point):
62          return self.p1 == point or self.p2 == point
63
64      def __hash__(self):
65          return self.p1.__hash__() ^ self.p2.__hash__()
66
67      def __repr__(self):
68          return "Edge ({!r}, {!r})".format( *args: self.p1, self.p2)
69
```

“def d(self, point)”: Phương thức này tính toán và trả về khoảng cách từ một điểm point đến cạnh, theo cách tính độ dài từ điểm đến đường thẳng mà cạnh biểu diễn. Nó sử dụng phép toán vector để tính toán khoảng cách này.

__str__(self): Phương thức này trả về một chuỗi biểu diễn cho đối tượng Edge bằng cách sử dụng định dạng chuỗi thông tin về hai điểm kết nối của cạnh.

__contains__(self, point): Phương thức này kiểm tra xem một điểm có nằm trên cạnh không. Nó trả về True nếu điểm truyền vào là một trong hai đầu mút của cạnh.

__hash__(self): Xác định hàm băm cho đối tượng Edge bằng cách kết hợp hàm băm của hai điểm đầu mút của cạnh.

__repr__(self): Phương thức này trả về một chuỗi biểu diễn khác cho đối tượng Edge, chứa thông tin về hai điểm kết nối của cạnh.

Các phương thức này cung cấp các chức năng khác nhau cho việc xử lý, hiển thị và so sánh cạnh trong không gian hai chiều (2D). Cụ thể, chúng cung cấp cách tính toán khoảng cách từ một điểm đến cạnh, kiểm tra một điểm có nằm trên cạnh hay không, cũng như cách biểu diễn chuỗi và băm đối tượng Edge.

2. Trình bày Class Graph.

```
1 usage
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
```

__init__(self, polygons): Phương thức khởi tạo của lớp Graph, nhận vào một danh sách các đa giác polygons. Quá trình xây dựng đồ thị được thực hiện ngay trong hàm này.

Duyệt qua từng đa giác trong polygons bằng vòng lặp for.

“if len(polygon) == 2:”: Nếu đa giác chỉ chứa 2 điểm thì loại bỏ điểm thứ 2 (điểm cuối cùng).

“if polygon[0] == polygon[-1]:”: Nếu đa giác là vòng tròn (điểm đầu và cuối giống nhau), thì thêm điểm đó vào đồ thị (self.add_point(polygon[0])).

Nếu không phải là đa giác đơn giản, duyệt qua từng điểm trong đa giác:

“edge = Edge(point, neighbor_point)”: Tạo cạnh (Edge) giữa điểm hiện tại và điểm kề liền sau nó trong đa giác.

Gán các polygon_id cho các điểm và thêm cạnh vào tập hợp các cạnh của đa giác (self.polygons[pid].add(edge)).

Thêm cạnh vào đồ thị và điểm đích của cạnh vào polygons nếu đa giác có hơn 2 điểm.

Tăng pid lên nếu đa giác có hơn 2 điểm, để đánh dấu các đa giác khác nhau trong self.polygons.

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

Đối với mỗi cạnh, sử dụng phương thức get_adjacent để lấy điểm kề với point mà không phải là None. Phương thức get_adjacent được định nghĩa trong lớp Edge để trả về điểm kế cận với một điểm được chỉ định trên cạnh.

Dùng filter để loại bỏ các giá trị None từ danh sách các điểm kề thu được từ các cạnh.

Cuối cùng, trả về một danh sách chứa các điểm kề đã lọc, tức là các điểm mà point có thể kết nối trực tiếp đến trong đồ thị.

```
94      def can_see(self, start):
95          see_list = list()
96          cant_see_list = list()
97
98          for polygon in self.polygons:
99              for edge in self.polygons[polygon]:
100                 for point in self.get_points():
101                     if start == point:
102                         cant_see_list.append(point)
103                     if start in self.get_polygon_points(polygon):
104                         for poly_point in self.get_polygon_points(polygon):
105                             if poly_point not in self.get_adjacent_points(start):
106                                 cant_see_list.append(poly_point)
107                     if point not in cant_see_list:
108                         if start.can_see(point, edge):
109                             if point not in see_list:
110                                 see_list.append(point)
111                         elif point in see_list:
112                             see_list.remove(point)
113                             cant_see_list.append(point)
114                         else:
115                             cant_see_list.append(point)
116         return see_list
117
```

Khởi tạo hai danh sách rỗng: see_list để lưu trữ các điểm có thể nhìn thấy được và cant_see_list để lưu trữ các điểm không thể nhìn thấy được.

Duyệt qua từng polygon trong self.polygons. Đối với mỗi cạnh (edge) trong từng polygon. Duyệt qua từng điểm (point) trong tất cả các điểm của đồ thị.

“if start == point:”: Nếu điểm hiện tại (start) trùng với điểm đang xét, thì thêm điểm đó vào cant_see_list (nghĩa là nó không thể nhìn thấy chính nó).

“if start in self.get_polygon_points(polygon):”: Cụm mã này kiểm tra xem start có trong danh sách các điểm của một polygon hay không. Nếu start nằm trong danh sách các điểm của polygon, nó duyệt qua từng poly_point trong get_polygon_points(polygon). Với mỗi poly_point, nó kiểm tra xem poly_point không thuộc danh sách các điểm kề với start (sử dụng get_adjacent_points(start)). Nếu một poly_point nào đó không nằm trong danh sách các điểm kề với start, nghĩa là không có cạnh nào kết nối start và poly_point, thì poly_point đó được thêm vào cant_see_list. Điều này cho biết từ start, không thể nhìn thấy được poly_point trong polygon đó.

Kế đó, đoạn mã sau có chức năng cập nhật hai danh sách see_list và cant_see_list dựa trên khả năng nhìn thấy của start đối với point.

“if point not in cant_see_list:”: Kiểm tra xem point có trong cant_see_list không. Nếu không, tiếp tục xử lý.

“if start.can_see(point, edge):”: Sử dụng phương thức can_see của start để kiểm tra khả năng nhìn thấy từ start đến point thông qua cạnh edge. Nếu có thể nhìn thấy, thực hiện kiểm tra tiếp.

"if point not in see_list:” Nếu point không có trong see_list, nghĩa là nó chưa được xác nhận là có thể nhìn thấy từ start, thêm point vào see_list.

“elif point in see_list:”: Nếu point đã có trong see_list, loại bỏ point khỏi see_list và thêm vào cant_see_list. Điều này chỉ ra rằng sau khi được xác nhận là có thể nhìn thấy, point không thể nhìn thấy từ start nữa.

“else:”: Nếu point không có trong cant_see_list và không thể nhìn thấy từ start thông qua edge, thì point được thêm vào cant_see_list.

“return see_list”: Trả về danh sách các điểm có thể nhìn thấy được (see_list).

```
118     def get_polygon_points(self, index):
119         point_set = set()
120         for edge in self.polygons[index]:
121             point_set.add(edge.p1)
122             point_set.add(edge.p2)
123         return point_set
124
```

Hàm get_polygon_points trong đoạn mã này nhận vào một chỉ số index và trả về một tập hợp point_set chứa các điểm thuộc về một đa giác có chỉ số index trong đồ thị.

“point_set = set()”: Khởi tạo một tập hợp trống point_set để lưu trữ các điểm thuộc về đa giác có chỉ số index.

“for edge in self.polygons[index]:”: Duyệt qua mỗi cạnh (edge) trong danh sách các cạnh của đa giác có chỉ số index. Với mỗi cạnh, thêm cả hai đầu mút của cạnh (edge.p1 và edge.p2) vào point_set.

“”’return point_set”: Trả về point_set, chứa các điểm thuộc về đa giác có chỉ số index.

```
125     def get_points(self):
126         return list(self.graph)
127
128     def get_edges(self):
129         return self.edges
130
        1 usage
131     def add_point(self, point):
132         self.graph[point].add(point)
133
```

Định nghĩa hàm get_point. Trong hàm get_points, nó trả về danh sách các điểm trong đồ thị bằng cách chuyển đổi tất cả các khóa (keys) trong self.graph thành một danh sách. Cụ thể, nó sử dụng phương thức list() để chuyển đổi tập hợp các khóa thành một danh sách.

Định nghĩa hàm get_edges(). Trong hàm get_edges, nó trả về tất cả các cạnh có trong đồ thị thông qua việc trả về self.edges, một tập hợp (set) các cạnh được lưu trữ trong đối tượng đồ thị.

Định nghĩa hàm add_point. Hàm add_point thêm một điểm mới vào đồ thị. Nó cập nhật self.graph bằng cách thêm một khóa mới tương ứng với điểm mới và đồng thời gán một tập hợp chứa chính nó (điểm đó) vào giá trị của khóa đó. Điều này có thể không thực sự hữu ích hoặc phù hợp với việc thêm điểm mới vào đồ thị, do nó chỉ gán một tập hợp chứa điểm đó vào chính điểm đó, nhưng không thiết lập các kết nối hoặc cạnh liên quan đến các điểm khác trong đồ thị.

```
134     def add_edge(self, edge):
135         self.graph[edge.p1].add(edge)
136         self.graph[edge.p2].add(edge)
137         self.edges.add(edge)
```

Trong hàm add_edge, nó thêm một cạnh mới (edge) vào đồ thị. Cụ thể, nó cập nhật self.graph bằng cách thêm cạnh edge vào tập hợp các cạnh của các điểm edge.p1 và edge.p2. Nó cũng thêm cạnh edge vào tập hợp self.edges, chứa tất cả các cạnh trong đồ thị.

“self.graph[edge.p1].add(edge)”: Thêm edge vào tập hợp các cạnh của điểm edge.p1.

“self.graph[edge.p2].add(edge)”: Thêm edge vào tập hợp các cạnh của điểm edge.p2.

“self.edges.add(edge)”: Thêm edge vào tập hợp self.edges để lưu trữ tất cả các cạnh trong đồ thị.

Điều này cập nhật cả tập hợp cạnh của mỗi điểm và tập hợp chung của tất cả các cạnh trong đồ thị mỗi khi một cạnh mới được thêm vào.

```
139     def __contains__(self, item):
140         if isinstance(item, Point):
141             return item in self.graph
142         if isinstance(item, Edge):
143             return item in self.edges
144         return False
```

Phương thức __contains__ trong đoạn mã kiểm tra xem một phần tử item có tồn tại trong đồ thị hay không dựa trên kiểu dữ liệu của nó (Point hoặc Edge). Nó trả về True nếu item thuộc về đồ thị và False nếu không.

“if isinstance(item, Point):”: Nếu item là một đối tượng kiểu Point, nó kiểm tra xem item có trong tập hợp các khóa của self.graph hay không.

Điều này có nghĩa là nó kiểm tra xem item có là một trong các đỉnh của đồ thị không.

"if isinstance(item, Edge):": Nếu item là một đối tượng kiểu Edge, nó kiểm tra xem item có trong tập hợp self.edges hay không. Điều này có nghĩa là nó kiểm tra xem item có là một trong các cạnh của đồ thị không.

Nếu item không thuộc kiểu Point hoặc Edge, phương thức trả về False.

Tóm lại, phương thức này cho phép ta kiểm tra xem một Point hoặc một Edge có tồn tại trong đồ thị hay không bằng cách sử dụng toán tử in. Ví dụ, nếu point là một đối tượng Point, ta có thể kiểm tra point in graph để xác định xem điểm point có trong đồ thị hay không. Tương tự, với một Edge, bạn có thể kiểm tra edge in graph để kiểm tra xem cạnh edge có trong đồ thị hay không.

```
146     def __getitem__(self, point):
147         if point in self.graph:
148             return self.graph[point]
149         return set()
150
```

Phương thức __getitem__ (hoặc toán tử indexing) được định nghĩa trong đoạn mã để trả về giá trị tương ứng với một point từ đồ thị. Nó hoạt động như sau:

"if point in self.graph:": Nếu point có trong self.graph (tức là point là một trong các khóa của self.graph), nó trả về giá trị tương ứng với point từ self.graph.

Nếu point không tồn tại trong self.graph, nó trả về một tập hợp rỗng set().

Cách hoạt động này cho phép ta truy cập các giá trị tương ứng với một point trong đồ thị thông qua toán tử []. Ví dụ, nếu point là một đối tượng Point, ta có thể truy cập các giá trị liên kết với point trong đồ thị bằng cách sử dụng graph[point].

Nếu point tồn tại trong self.graph, nó trả về giá trị tương ứng với point, có thể là một tập hợp chứa các cạnh hoặc các giá trị khác liên kết với point trong đồ thị. Nếu point không tồn tại, nó trả về một tập hợp rỗng.

```
151     def __str__(self):
152         res = ""
153         for point in self.graph:
154             res += "\n" + str(point) + ": "
155             for edge in self.graph[point]:
156                 res += str(edge)
157         return res
```

Phương thức __str__ trong đoạn mã trên được định nghĩa để trả về một chuỗi biểu diễn cho đồ thị.

“ res = “” ”: Khởi tạo một chuỗi rỗng res.

“for point in self.graph”: Duyệt qua mỗi point trong self.graph.

Đối với mỗi point, nó thêm chuỗi biểu diễn của point vào res (str(point)), sau đó duyệt qua tất cả các edge liên kết với point và thêm chuỗi biểu diễn của edge vào res.

Cuối cùng, phương thức trả về chuỗi res làm biểu diễn của đồ thị. Phương thức này tạo ra một chuỗi biểu diễn cho đồ thị bằng cách liệt kê các điểm và các cạnh mà chúng kết nối. Chuỗi này chứa thông tin về các điểm trong đồ thị và các cạnh mà chúng tham gia, được phân tách nhau bởi dấu xuống dòng (\n).

```
159     def __repr__(self):
160         return self.__str__()
161
```

Phương thức __repr__ được định nghĩa để trả về một chuỗi biểu diễn của đối tượng đồ thị khi được gọi hàm repr().

Trong trường hợp này, phương thức __repr__ đơn giản chỉ trả về kết quả của phương thức __str__. Điều này làm cho __repr__ có cùng đầu ra với __str__.

__repr__ thường được sử dụng để xác định một biểu diễn của đối tượng để debug hoặc mô tả chi tiết hơn về đối tượng trong khi __str__ thường được sử dụng để hiển thị thông tin cho người dùng. Trong trường hợp này, vì __repr__ chỉ gọi __str__, nó trả về cùng một chuỗi mô tả cho đối tượng đồ thị.

```
162     def h(self, point):
163         heuristic = getattr(self, 'heuristic', None)
164         if heuristic:
165             return heuristic[point]
166         else:
167             return -1
168
```

Phương thức h trong đoạn mã này là một phương thức heuristic (ước lượng) được sử dụng trong việc tìm kiếm. Nó nhận một điểm point và cố gắng trả về một giá trị heuristic tương ứng với point.

Trước hết, nó sử dụng hàm getattr để lấy giá trị của thuộc tính heuristic từ đối tượng hiện tại. Nếu thuộc tính này tồn tại, nó sẽ trả về giá trị tương ứng với point trong từ điển heuristic.

Nếu không tìm thấy thuộc tính heuristic hoặc point không có giá trị heuristic tương ứng, nó trả về một giá trị mặc định, trong trường hợp này là -1.

Nếu trong đồ thị có sẵn một phương thức heuristic đã được định nghĩa và chứa các ước lượng cho các điểm trong đồ thị, phương thức h sẽ trả về giá trị heuristic tương ứng với point. Nếu không, nó sẽ trả về -1 làm giá trị mặc định. Điều này có thể được sử dụng khi không có ước lượng heuristic cụ thể cho điểm được cung cấp.

3. Trình bày Euclid_distance Function.

```
169 def euclid_distance(point1, point2):
170     return round(float(math.sqrt((point2.x - point1.x)**2 + (point2.y - point1.y)**2)), 3)
171
```

Hàm euclid_distance tính khoảng cách Euclid (Euclidean distance) giữa hai điểm point1 và point2 trong không gian hai chiều.

Sử dụng công thức Euclid để tính toán khoảng cách:

$$\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Trả về giá trị khoảng cách tính được, là kết quả của phép tính căn bậc hai của tổng bình phương của độ chênh lệch theo chiều x và chiều y giữa hai điểm. Kết quả được làm tròn đến 3 chữ số thập phân bằng hàm round với 3 chữ số sau dấu phẩy.

Công thức này phổ biến trong việc tính khoảng cách giữa hai điểm trong không gian hai chiều theo đường thẳng. Trong trường hợp này, hàm euclid_distance được sử dụng để tính toán khoảng cách giữa các điểm trong một đồ thị hoặc không gian hai chiều.

4. Trình bày Search Function.

```
1 usage
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
```

“closed = set()”: Khởi tạo một tập hợp closed để lưu trữ các nút đã duyệt. Cấu trúc dữ liệu set() chứa các phần tử không có thứ tự, duy nhất tức không cho phép lặp lại.

“queue = PriorityQueue()”: Khởi tạo một hàng đợi ưu tiên (PriorityQueue) để lưu trữ các nút chưa duyệt, sắp xếp theo độ ưu tiên.

“queue.put((0 + func(graph, start), start))”: Đưa nút ban đầu start vào hàng đợi với ưu tiên được tính dựa trên tổng của chi phí hiện tại và giá trị được tính bởi hàm func. Hàm func sẽ là hàm lambda của các thuật toán như A*, Greedy Best First Search, BFS, DFS, UCS, … tùy ta chọn.

“while not queue.empty():”: Bắt đầu vòng lặp while, với mỗi vòng tiếp theo sau đó, kiểm tra hàng đợi có không trống hay không.

“cost, node = queue.get()”:Bởi queue là hàng đợi ưu tiên xét theo trọng số nên câu lệnh sẽ thực hiện lấy nút ở đầu hàng đợi với chi phí thấp nhất và gán lần lượt cho cost và node.

“if node == goal:”: Kiểm tra nếu nút này là nút mục tiêu (goal), trả về nút này “return node” và kết thúc hàm search.

Vòng lặp “for i in graph.can_see(node)”: Duyệt qua các nút i mà node có thể nhìn thấy trong đồ thị (graph.can_see(node)).

“new_cost = node.g + euclid_distance(node, i)”: Trong đoạn mã đang tính toán chi phí mới từ node đến i bằng cách cộng chi phí hiện tại của node (từ đầu start đến node đang xét) với khoảng cách Euclid giữa node và i. Điều này giúp cập nhật chi phí mới khi di chuyển từ node đến i trong quá trình tìm kiếm đường đi.

Sau đó kiểm tra xem liệu nút đó đã được duyệt hay chưa. Nếu chưa hoặc chi phí mới thấp hơn chi phí hiện tại của nút đó, cập nhật thông tin cho nút đó và đưa nó vào hàng đợi với ưu tiên được tính bằng hàm func.

Kết thúc vòng lặp và trả về nút mà thuật toán đưa ra sau khi duyệt xong.

```
191    a_star = lambda graph, i: i.g + graph.h(i)
192    greedy = lambda graph, i: graph.h(i)
193
```

Cả object được định danh lần lượt là a_start và greedy thông qua “hàm ẩn danh” (lambda function). Với cú pháp của Lambda Function là: “lambda arguments: expression”.

“a_start = lambda graph, i: i.g + graph.h(i)”: Khởi tạo “hàm ẩn danh” (lambda function) a_start với đối số đầu vào là graph (đồ thị được sử dụng) và i (một đỉnh trong đồ thị). Khi đó, biểu thức trả về là “i.g + graph.h(i)”.

+ “i.g”: Là chi phí (cost) để đi từ đỉnh gốc đến đỉnh i.

+ “graph.h(i)”: Đại diện cho hàm heuristic (là một loại ước lượng) để ước tính chi phí từ đỉnh i đến đích.

Trong thuật toán A*, h(i) thường là ước tính từ đỉnh i đến đích dựa trên heuristic. Công thức tổng cộng i.g + graph.h(i) được sử dụng để ước tính tổng chi phí từ đỉnh gốc tới đỉnh i và từ i đến đích.

5. Trình bày Main Function.

```
1 usage
def main():
    n_polygon = 0
    poly_list = list(list())
    x = list()
    y = list()
```

"def main():": Định nghĩa một hàm có tên main. Trong Python, các hàm được xác định bằng từ khóa def, theo sau là tên hàm và dấu ngoặc đơn (). Dấu hai chấm: biểu thị sự bắt đầu của khối chức năng.

"n_polygon = 0": Tạo một biến có tên n_polygon và gán cho nó giá trị 0. Biến này mang thông tin số lượng đa giác có trong biểu đồ.

"poly_list = list(list())": Khởi tạo một biến có tên poly_list dưới dạng danh sách trống. list() tạo danh sách trống và list(list()) tạo danh sách chứa danh sách trống. Được sử dụng để lưu trữ danh sách đại diện cho các đa giác khác nhau.

"x = list()": Khởi tạo một biến có tên x dưới dạng danh sách trống. Nhằm mục đích lưu trữ tọa độ x của các điểm trong đa giác.

"y = list()": Tương tự như như list x, dòng này khởi tạo một biến có tên y dưới dạng một danh sách trống, nhằm mục đích lưu trữ tọa độ y của các điểm trong đa giác.

```
    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])
```

Tiếp theo đây, ta sẽ mở file Input.txt. Sẽ có bước tiền xử lý dòng trong file để chuẩn hóa dữ liệu cho việc đọc, sau đó thực hiện các thao tác gán để trả về số lượng đa giác trong biểu đồ, điểm bắt đầu start, điểm kết thúc end.

“with open('Input.txt', 'r') as f:”: Mở tệp 'Input.txt' để đọc bằng lệnh with. Với mode ‘r’ xử lý việc mở để đọc, chỉ cho phép đọc file, nếu file không tồn tại thì trả về lỗi. “as f”, f là object file được tạo ra sau khi mở file. Vì ta mở file bằng lệnh with thì file sẽ tự động đóng sau khi thực hiện hết các khối lệnh tạo ra trong with, nên ta không cần dùng tới lệnh đóng file “f.close()” trong trường hợp này.

“line = f.readline()”: Đọc dòng đầu tiên từ tệp và lưu nó vào biến line. Ví dụ xét với dòng đầu tiên xuyên suốt đoạn code ngắn của file “Input.txt”. Lúc này line = “8 2 4 38 21”.

“line = line.strip()”: Trả về một bản sao của chuỗi ban đầu là line trong đó loại bỏ khoảng trắng từ dòng vừa đọc được line để loại bỏ các khoảng trống không cần thiết ở đầu hoặc cuối dòng. Lúc này, line = “8 2 4 38 21”.

“line = line.split()”: Tách line thành một danh sách các chuỗi con dựa trên khoảng trắng (mặc định). Lúc này, line = [“8”, “2”, “4”, “38”, “21”].

“line = list(map(int, line))”: Chuyển đổi mỗi phần tử trong danh sách line thành số nguyên bằng cách sử dụng map(int, line) và sau đó chuyển kết quả này thành một danh sách. Hàm map() sẽ duyệt từng phần tử trong list line và thực hiện thao tác với từng phần tử đó, ở đây là ép kiểu thành số nguyên và trả về danh sách chứa các số nguyên đó. Lúc này, line = [8, 2, 4, 38, 21].

“n_polygon = line[0]”: Gán giá trị đầu tiên của danh sách (sau khi chuyển đổi thành số nguyên) cho biến n_polygon. Lúc này, n_polygon = 8.

“start = Point(line[1], line[2])”: Tạo một đối tượng Point mới sau đó, python gọi constructor của class Point với đối số truyền vào là phần tử thứ hai line[1] và thứ ba line[2] của danh sách line sau khi chuyển đổi thành số nguyên, và đối số mặc định polygon_id được gán cho giá trị là -1.

Tương tự, ta cũng có “goal = Point(line[3], line[4])”: Tạo một đối tượng Point mới sau đó, python gọi constructor của class Point với đối số truyền vào là phần tử thứ tư line[3] và thứ năm line[4] của danh sách line sau khi chuyển đổi thành số nguyên, và đối số mặc định polygon_id được gán cho giá trị là -1.

“poly_list.append([start])”: Thêm một danh sách mới chứa đối tượng start vào danh sách poly_list.

```
209         for line in f:
210             point_list = list()
211             line = line.split()
212             n_vertex = int(line[0])
213             for j in range(0, 2*n_vertex, 2):
214                 point_list.append(Point(int(line[j + 1]), int(line[j + 2])))
215             poly_list.append(point_list[:])
216         poly_list.append([goal])
217         graph = Graph(poly_list)
218         graph.heuristic = {point: point.heuristic(goal) for point in graph.get_points()}
219
```

“for line in f:”: Đọc từng dòng trong file Input (từ dòng thứ 2 trở đi vì ta đã xét dòng đầu tiên trước đó rồi).

“point_list = list()”: Khởi tạo một danh sách để lưu trữ danh sách các điểm.

“line = line.split()”: Tách dòng hiện tại thành một danh sách các chuỗi con dựa trên khoảng trắng.

“n_vertex = int(line[0])”: Lấy số đỉnh của đa giác từ dòng hiện tại (là phần tử đầu tiên sau khi chuyển thành số nguyên).

Tiếp theo, vòng lặp “for j in range(0, 2*n_vertex, 2):” sẽ lặp qua các đỉnh của đa giác. Để dễ hiểu, ta xét dòng thứ 2 trong file Input.txt. Ta có dòng thứ 2 là “4 4 2 4 7 20 7 20 2”. Xét biến j chạy từ 0 cho đến 2*n_vertex – 2 = 2* 4 – 2 = 6, bước nhảy là 2. Khi đó, chỉ số cặp line[j+1] và line[j+2] sẽ chạy từ (line[1], line[2]) đến (line[7], line[8]).

“point_list.append(Point(int(line[j + 1]), int(line[j + 2])))”: Tạo đối tượng Point mới từ từng cặp hoành độ và tung độ được cung cấp trong dòng và thêm nó vào point_list.

Sau khi thêm toàn bộ đối tượng chứa đối số là các cặp tọa độ của các đỉnh thuộc đa giác ứng với từng dòng, ta dùng lệnh “poly_list.append(point_list[:])” để thêm một bản sao của danh sách point_list vào danh sách poly_list. Lặp lại cho đến khi vòng for duyệt hết các dòng trong file Input.txt.

“poly_list.append([goal])”: Thêm một danh sách chứa điểm goal vào danh sách poly_list. Điều này tạo ra một đa giác mới chứa chỉ một điểm, điểm này thường được coi là điểm đích mà một thuật toán tìm đường đi sẽ cố gắng đến.

“graph = Graph(poly_list)”: Tạo một đối tượng Graph từ danh sách các đa giác trong poly_list.

“graph.heuristic = {point: point.heuristic(goal) for point in graph.get_points()}:” Trong đó: graph.get_points() trả về danh sách các điểm trong đồ thị. Dòng code này tạo một từ điển (dictionary) trong đó mỗi điểm trong đồ thị được liên kết với giá trị heuristic của nó, được tính bằng cách gọi phương thức heuristic(goal) của từng điểm. Điều này có thể đo lường khoảng cách, số bước cần thiết hoặc một ước lượng chi phí từ điểm hiện tại đến điểm đích (goal).

```
220         a = search(graph, start, goal, a_star)
221
222         result = list()
223
224         while a:
225             result.append(a)
226             a = a.pre
227         result.reverse()
228         print_res = [[point, point.polygon_id] for point in result]
229         print(*print_res, sep=' -> ')
230         plt.figure()
231         plt.plot( *args: [start.x], [start.y], 'ro')
232         plt.plot( *args: [goal.x], [goal.y], 'ro')
```

“a = search(graph, start, goal, a_star)”: Gọi hàm search để tìm đường đi từ start đến goal trong đồ thị graph bằng thuật toán tìm đường đi được chỉ định (a_star). Kết quả của việc tìm đường sẽ được lưu vào a.

“result = list()”: Khởi tạo một danh sách result để lưu trữ các điểm trên đường đi từ start đến goal. Trước mắt là sẽ lưu ngược, từ điểm goal về điểm ban đầu của bài toán là start, sau đó dùng lệnh reverse() để đảo ngược lại.

“while a:”: Một vòng lặp while được sử dụng để lấy các điểm trên đường đi từ a (điểm cuối cùng trên đường đi) về start. Các điểm này được thêm vào danh sách result. Điều kiện dừng vòng lặp là a mang giá trị None. Thật vậy, khi chạy vòng lặp, sẽ luôn cập nhật lại a.pre, mà khi node đang xét là node start, thì a.pre = None, khi đó vòng lặp sẽ dừng lại để trả về list đường đi.

“result.append(a)”: Thêm điểm a vào danh sách result.

“a = a.pre”: Di chuyển a đến điểm trước đó trên đường đi, được lưu trong thuộc tính pre của a.

“result.reverse()”: Đảo ngược thứ tự của các điểm trong result để có thứ tự từ start đến goal.

“print_res = [[point, point.polygon_id] for point in result]”: Tạo một danh sách mới print_res chứa các cặp point và polygon_id của chúng trên đường đi.

“print(*print_res, sep=' -> ')”: In ra các điểm trên đường đi cùng với polygon_id của chúng, sắp xếp theo thứ tự từ start đến goal, mỗi cặp được phân cách bởi “->”.

“plt.figure(), plt.plot([start.x], [start.y], 'ro')”: Sử dụng thư viện matplotlib để vẽ một hình vẽ mới và thêm điểm start vào đồ thị bằng biểu đồ scatter với ký hiệu 'ro' (đỏ).

“plt.plot([goal.x], [goal.y], 'ro')”: Thêm điểm goal vào biểu đồ cũng với ký hiệu 'ro' (đỏ).

```
234         for point in graph.get_points():
235             x.append(point.x)
236             y.append(point.y)
237         plt.plot( *args: x, y, 'ro')
238
```

Vòng lặp “for point in graph.get_points():” duyệt qua từng điểm trong đồ thị graph. Mỗi điểm này được thêm hoành độ x vào danh sách x và tung độ y vào danh sách y.

“plt.plot(x, y, 'ro')”: Sau khi đã thu thập hoành độ và tung độ của tất cả các điểm trong đồ thị, mã vẽ biểu đồ scatter plot sử dụng plt.plot() với các điểm có hoành độ là x, tung độ là y, và ký hiệu 'ro' để đánh dấu các điểm này bằng hình tròn màu đỏ ('ro' trong matplotlib tượng trưng cho màu đỏ - 'r' - và hình dạng hình tròn - 'o').

```
239         for i in range(1, len(poly_list) - 1):
240             coord = list()
241             for point in poly_list[i]:
242                 coord.append([point.x, point.y])
243             coord.append(coord[0])
244             xs, ys = zip(*coord) #create lists of x and y values
245             plt.plot( *args: xs, ys)
```

Vòng lặp “for i in range(1, len(poly_list) - 1):” duyệt qua từng phần tử trong poly_list bắt đầu từ phần tử thứ hai đến phần tử thứ kế cuối (trừ đi phần tử cuối cùng). Vì poly_list chứa len(poly_list) – 2 phần tử là các điểm tọa độ của đa giác, còn phần tử đầu và cuối là point start và point goal.

Lúc này, mỗi đa giác được biểu diễn bởi danh sách các điểm trong poly_list[i].

Vòng lặp tiếp theo “for point in poly_list[i]”: duyệt qua từng điểm trong từng đa giác.

Tạo danh sách coord chứa các cặp tọa độ [x, y] của các điểm trong đa giác. Sau đó, dùng lệnh “coord.append(coord[0])” để thêm điểm đầu tiên của đa giác vào cuối danh sách coord nhằm mục đích kết nối điểm cuối cùng với điểm đầu tiên, tạo thành hình đa giác đóng.

“xs, ys = zip(*coord)”: Tách tọa độ x và y ra từ danh sách coord để tạo thành các danh sách xs và ys thông qua zip(). Vì tham số được truyền là list nên giá trị trả về cho xs, ys cũng là list

“plt.plot(xs, ys)”: Vẽ hình đa giác dựa trên danh sách xs (hoành độ) và ys (tung độ) đã tạo ra từ các điểm trong đa giác. Điều này sẽ tạo ra các đường nối các điểm để hiển thị hình dạng của đa giác trên biểu đồ.

```
246
247         x = list()
248         y = list()
249
250         for point in result:
251             x.append(point.x)
252             y.append(point.y)
253
254         plt.plot( *args: x, y, 'b', linewidth=2.0)
255         plt.show()
256
```

Hai danh sách x và y được khởi tạo để lưu trữ các tọa độ x và y của các điểm trong result.

Vòng lặp “for point in result:” duyệt qua từng điểm trong danh sách result.

Từ mỗi điểm, tọa độ x và y được lấy ra và được thêm vào danh sách tương ứng x và y.

“plt.plot(x, y, 'b', linewidth=2.0)”: Tạo đồ thị đường bằng cách sử dụng plt.plot() với các tọa độ x và y từ x và y đã thu thập được. Điều này sẽ vẽ một đường liên tục theo thứ tự các điểm trong result. Tham số 'b' chỉ định màu của đường (màu xanh), và linewidth=2.0 chỉ định độ dày của đường.

Cuối cùng là lệnh “plt.show()” để hiển thị đồ thị đã vẽ lên màn hình.

II. BÀI TOÁN TRÊN THUẬT TOÁN BFS, DFS VÀ UCS.

1. Bài toán trên thuật toán BFS.

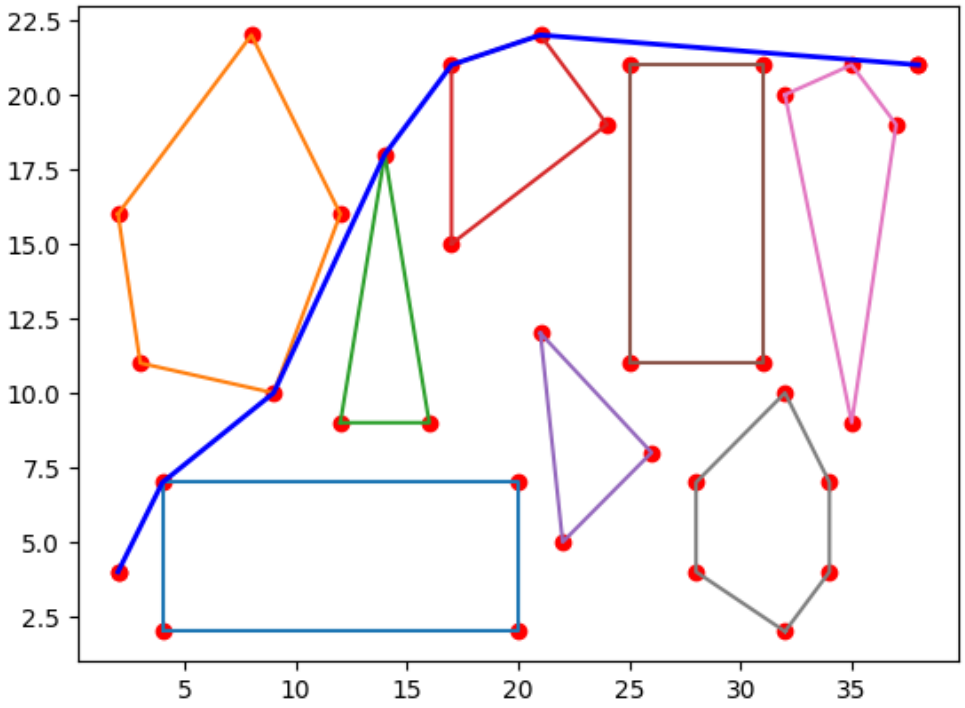
Để ưu tiên hạn chế chỉnh sửa đoạn mã cho sẵn, ta sẽ chỉ thêm vào các hàm lambda, tránh thao tác chỉnh sửa trên hàm search. Nhìn vào hàm search ta thấy rằng chúng như dạng tổng quát của thuật toán BFS. Hơn nữa, ta biết rằng trong hàm search dùng hàng đợi ưu tiên để lấy các point ra xét, khi đó ta chỉ cần tạo hàm lambda bfs với hằng số 0, tức mức độ ưu tiên của hàng đợi ưu tiên theo trọng số của từng point là như nhau. Hàng đợi ưu tiên lúc này như hàng đợi bình thường. Từ đó ta dễ dàng có được đoạn mã giải quyết bài toán dựa trên thuật toán BFS.

Để cài đặt và chạy thử, ta thêm cho chương trình đoạn mã bfs = lambda graph, i: 0. Có thể cho phía dưới 2 hàm lambda a_star và greedy cho dễ kiểm soát.

Lúc này, để chạy đoạn mã giải quyết bài toán với thuật toán BFS. Ta chỉ cần sửa dòng “a = search(graph, start, goal, a_start)” trong main function thành “a = search(graph, start, goal, bfs)”.

Đoạn mã cho ta kết quả:

```
C:\Users\PC-LENOVO\AppData\Local\Programs\Python\Python310\python.exe C:\Users\PC-LENOVO\Downloads\Pycharm_code\node_cansee.py
C:\Users\PC-LENOVO\Downloads\Pycharm_code\node_cansee.py:93: DeprecationWarning: NotImplemented should not be used in a boolean context
  return list(filter(None.__ne__, [edge.get_adjacent(point) for edge in self.edges]))
[(2, 4), -1] -> [(4, 7), 0] -> [(9, 10), 1] -> [(14, 18), 2] -> [(17, 21), 3] -> [(21, 22), 3] -> [(38, 21), -1]

Process finished with exit code 0
```



2. Bài toán trên thuật toán DFS.

Ý tưởng ban đầu để từ hàng đợi ưu tiên trong search có thể vận hành tương tự Stack là LIFO, ta sẽ đưa trọng số giảm dần theo bậc -1, -2, -3, …Tuy nhiên việc đó không khả thi nếu không thay đổi hàm search.

Do đó, ta sẽ tạo 1 hàm searchByDFS dựa trên hàm search có sẵn. Ta sẽ chỉ thêm vào level = 0 sau đó mỗi vòng lặp for sẽ giảm đi -1 và gán giá trị level này cho new_cost. Điều này sẽ tạo ra trọng số giảm dần cho từng phần từ được đưa vào, khi đó phần tử nào vào cuối cùng (Last In) sẽ có trọng số bé nhất và được hàng đợi ưu tiên đưa ra trước để xét (First Out) đúng với việc cài đặt thuật toán DFS. Và để giống với số lượng đối số truyền vào như hàm search là 4 và các hàm thuật toán đều là lambda thì ta sẽ viết hàm dfs = lambda graph, i: 0. Ở đây hàm lambda dfs chỉ tác động vào giá trị trọng số của start trong hàng đợi. Vì nó luôn được gọi ra đầu tiên để xét nên việc đặt hằng bằng giá trị nào không quan trọng lắm, do đó ta có thể đặt bằng 0 cho tiện xử lý.

Lúc này, để chạy đoạn mã giải quyết bài toán với thuật toán DFS. Ta chỉ cần sửa dòng “a = search(graph, start, goal, a_start)” trong main function thành comment (để sau này có dùng lại hàm này cho thuật toán

khác thì không cần nhập lại) và viết thêm hàm “a = searchByDFS(graph, start, goal, dfs)”.

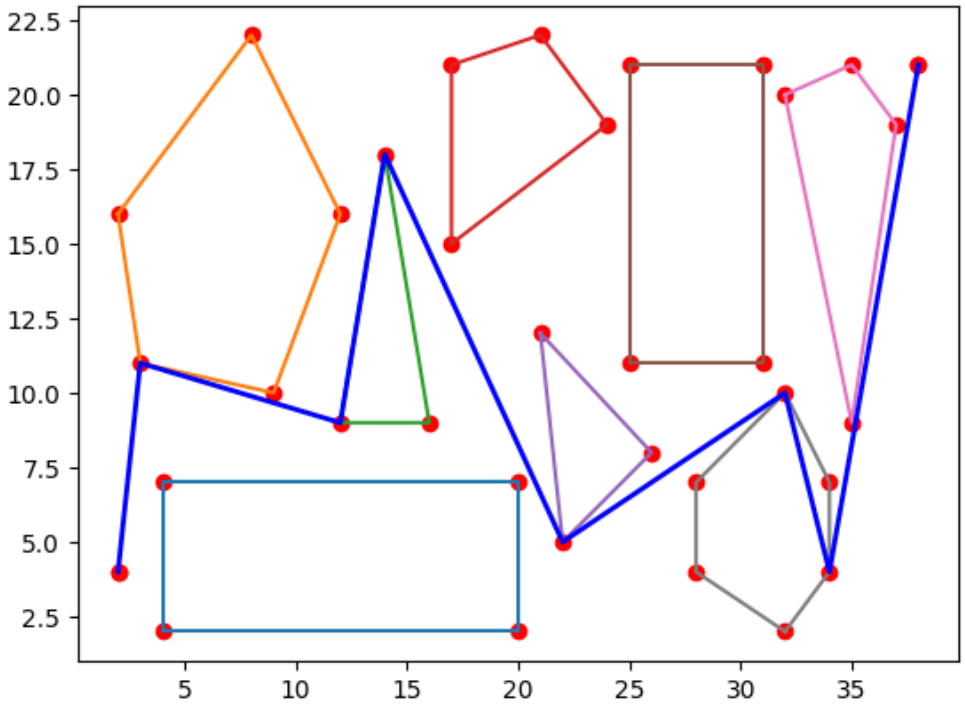
Phần đoạn mã được thêm vào:

```
196     def searchByDFS (graph, start, goal, func):
197         level = 0
198         closed = set()
199         queue = PriorityQueue()
200         queue.put((0 + func(graph, start), start))
201         if start not in closed:
202             closed.add(start)
203         while not queue.empty():
204             cost, node = queue.get()
205             if node == goal:
206                 return node
207             for i in graph.can_see(node):
208                 level -= 1
209                 new_cost = node.g + euclid_distance(node, i)
210                 if i not in closed or new_cost < i.g:
211                     closed.add(i)
212                     i.g = new_cost
213                     i.pre = node
214                     new_cost = level
215                     queue.put((new_cost, i))
216         return node
217     dfs = lambda graph, i: 0
```

```
244             #a = search(graph, start, goal, bfs)
245             a = searchByDFS(graph, start, goal, dfs)
246             result = list()
```

Kết quả thu được:

```
C:\Users\PC-LENOVO\AppData\Local\Programs\Python\Python310\python.exe C:\Users\PC-LENOVO\Downloads\Pycharm_code\node_cansee.py
C:\Users\PC-LENOVO\Downloads\Pycharm_code\node_cansee.py:93: DeprecationWarning: NotImplemented should not be used in a boolean context
  return list(filter(None.__ne__, [edge.get_adjacent(point) for edge in self.edges]))
[(2, 4), -1] -> [(3, 11), 1] -> [(12, 9), 2] -> [(14, 18), 2] -> [(22, 5), 4] -> [(32, 10), 7] -> [(34, 4), 7] -> [(38, 21), -1]

Process finished with exit code 0
```



3. Bài toán trên thuật toán UCS.

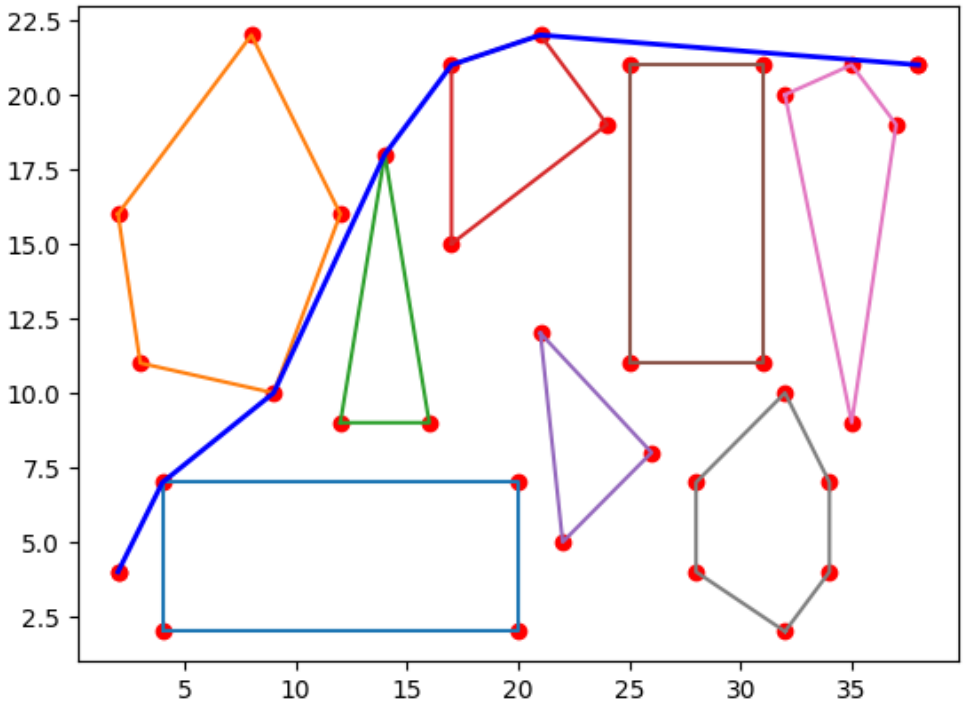
Để ưu tiên hạn chế chỉnh sửa đoạn mã cho sẵn, ta sẽ chỉ thêm vào các hàm lambda, tránh thao tác chỉnh sửa trên hàm search. Quan sát phần Cài đặt thuật toán UCS trong bài thực hành tuần 1. Ta có thể thấy search function với thuật toán UCS đều có điểm tương đồng khá rõ khi đều cùng hàng đợi ưu tiên nhưng thay vì cập nhật là tính cost từ current_node đến node kế tiếp, trong đoạn mã này có sẵn cho ta chính là giá trị i.g.

Để cài đặt và chạy thử, tương tự bfs ta thêm cho chương trình đoạn mã "ucs = lambda graph, i: i.g". Nên để ở phía dưới 2 hàm lambda a_star và greedy để cho dễ kiểm soát

Lúc này, để chạy đoạn mã giải quyết bài toán với thuật toán UCS. Ta chỉ cần sửa dòng "a = search(graph, start, goal, ucs)" trong main function thành "a = search(graph, start, goal, ucs)".

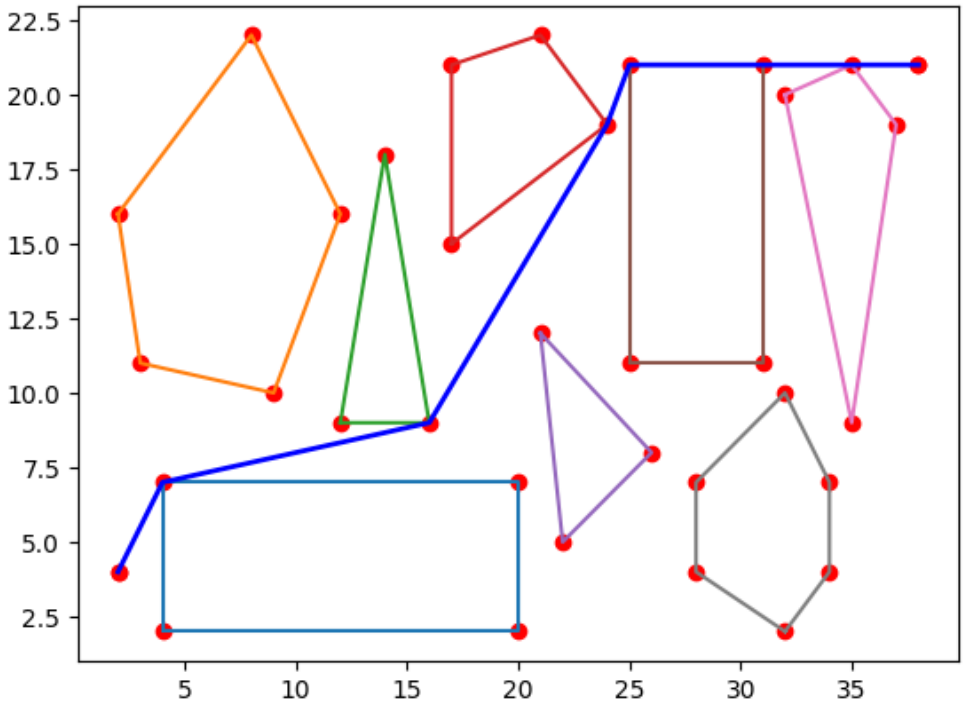
Đoạn mã cho ta kết quả:

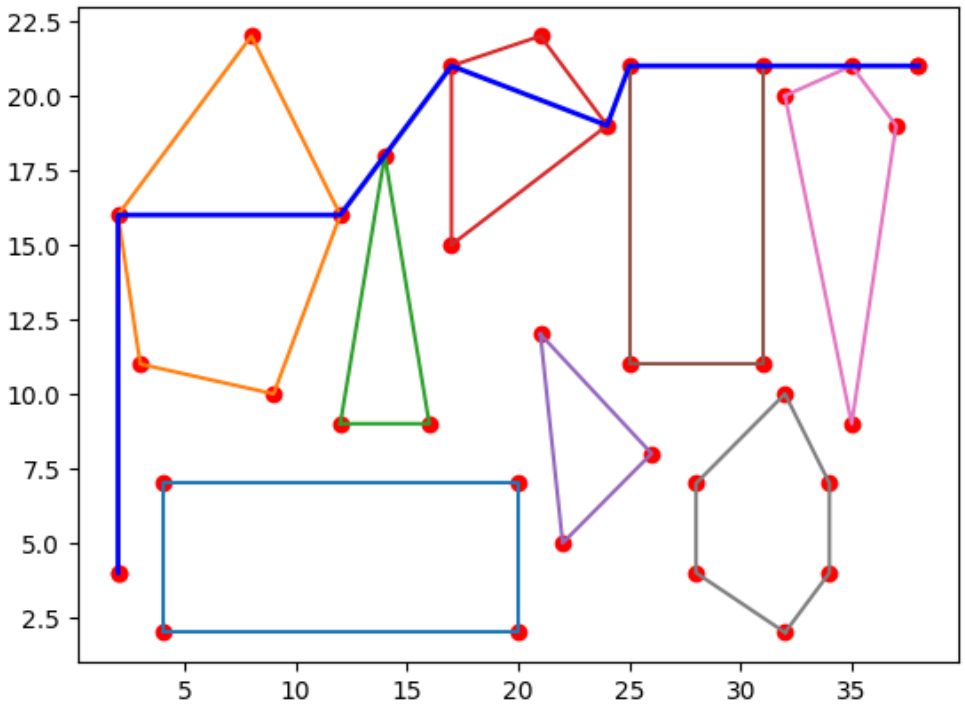
```
C:\Users\PC-LENOVO\AppData\Local\Programs\Python\Python310\python.exe C:\Users\PC-LENOVO\Downloads\Pycharm_code\node_cansee.py
C:\Users\PC-LENOVO\Downloads\Pycharm_code\node_cansee.py:93: DeprecationWarning: NotImplemented should not be used in a boolean context
  return list(filter(None.__ne__, [edge.get_adjacent(point) for edge in self.edges]))
[(2, 4), -1] -> [(4, 7), 0] -> [(9, 10), 1] -> [(14, 18), 2] -> [(17, 21), 3] -> [(21, 22), 3] -> [(38, 21), -1]

Process finished with exit code 0
```



III. NHẬN XÉT.

Xem thêm kết quả khi giải quyết bài toán lần lượt bằng thuật toán A* và GBFS:


22.5
20.0
17.5
15.0
12.5
10.0
7.5
5.0
2.5
5
10
15
20
25
30
35


22.5
20.0
17.5
15.0
12.5
10.0
7.5
5.0
2.5
5
10
15
20
25
30
35

Từ các kết quả trên, ta có thể thấy tìm kiếm tốt nhất thu được là A*, kế đó là BFS. Ở GBFS vì khoảng cách ở đây là đường chim bay nên kết quả thu được sẽ thấy phần đường kẻ màu xanh đi xuyên qua các vật cản là các đa giác. BFS và UCS thu được kết quả giống nhau.