CƠ SỞ TRÍ TUỆ NHÂN TẠO

ĐỒ ÁN 1 - TÌM KIẾM

YÊU CẦU 2 - ROBOT TÌM ĐƯỜNG

- **GV: LÊ HOÀI BẮC**
- **GV: LÊ NGỌC THÀNH**
- **GV: NGUYỄN NGỌC THẢO**
- **GV: NGUYỄN HẢI MINH**

Sinh viên thực hiện:

• **Họ tên:** • **MSSV:** • **Email:** • **SĐT:**	➢ **Cao Nguyễn Minh Hiếu** ➢ **1512157** ➢ 1512157@students.hcmus.edu.vn ➢ **0165.259.2239**

I. Các lưu ý để chạy chương trình thành công:

_ Chương trình được viết bằng ngôn ngữ Python, phiên bản Python 3.6 - 64bit, và chỉ gói gọn trong một file duy nhất là *main.py*.

_ Source code của Yêu cầu 2 có import 2 thư viện chuẩn là *math* và *time*, và 1 thư viện cài đặt thêm của python, hỗ trợ việc vẽ hình, là *matplotlib*.

_ Để chạy chương trình, Thầy/Cô mở tập tin *main.py* trong thư mục Yêu cầu 2 bằng IDE của Python và chạy.

_ Python 3.6 tải tại: https://www.python.org/downloads/

_ Để cài đặt thư viện matplotlib, Thầy/Cô vào command line và chạy dòng lệnh sau: *python -m pip install matplotlib* . Đợi command line cài đặt mất khoảng vài giây. Nếu Thầy/Cô dùng phiên bản Python3 thì thay từ *"python"* trong dòng lệnh trên thành *"python3"* .

❖ *Lưu ý:* Thư viện *matplotlib* yêu cầu máy đã cài đặt một thư viện khác là *numpy*. Nếu chưa cài đặt *numpy*, Thầy/Cô thực hiện tương tự như trên, tức vào command line chạy *python -m pip install numpy* (lưu ý phiên bản python), sau đó mới cài matplotlib. Để cài đặt thành công, máy tính phải đang kết nối Internet.

II. Cấu trúc dữ liệu:

1. Cấu trúc tập tin input.txt:

● Chương trình đọc input từ các tập tin *input1.txt, input2.txt*, …, đặt cùng cấp với tập tin *main.py*. Mỗi input là mỗi trường hợp đặc biệt mà chương trình phải xử lí.

● Trong tập tin *input.txt*:

_ Dòng đầu tiên chứa một số nguyên n (n ≥ 1) thể hiện số đa giác.

_ Dòng thứ hai chứa bốn số nguyên, có dạng *"x_1,y_1 x_2,y_2"* (giữa y_1 và x_2 là dấu khoảng trắng), lần lượt biểu diễn tọa độ của 2 đỉnh START và GOAL.

_ n dòng tiếp theo chứa n*2 số nguyên trên mỗi dòng, có dạng *"x_1,y_1 x_2,y_2 x_2,y_2 x_3,y_3 …"* , lần lượt biểu diễn tọa độ các đỉnh của đa giác tương ứng.

● Ví dụ:

Cấu trúc tập tin *input0.txt*	Ý nghĩa:
3 4,7 4,1 1,8 2,8 2,5 1,5	Có 3 đa giác Đỉnh Start = (4,7) , đỉnh Goal = (4,1) Đa giác 1 có 4 đỉnh là (1,8), (2,8), (2,5), (1,5)

6,8 7,8 7,5 3,5 5,5 5,4 3,4	Đa giác 2 có 3 đỉnh là (6,8), (7,8), (7,5) Đa giác 3 có 4 đỉnh là (3,5), (5,5), (5,4), (3,4)

❖ *Lưu ý:* Input phải đảm bảo cấu trúc như trên và tính đúng đắn của số liệu. Nếu số đa giác là âm hoặc lớn hơn số đa giác thực sự, hoặc tọa độ chứa kí tự không phải số, thì chương trình sẽ báo lỗi khi biên dịch. Mặt khác, nếu số liệu hợp lệ, nhưng thứ tự đỉnh của đa giác không tạo thành đa giác lồi, hoặc có đa giác nào đó có dưới 3 đỉnh, thì chương trình vẫn chạy, nhưng kết quả sẽ không như mong muốn. Đây là một khuyết điểm của em trong việc xử lí các ngoại lệ của input.

2. Cấu trúc dữ liệu lưu trữ đa giác và điểm trong chương trình:

2.1 Cấu trúc dữ liệu lưu trữ một điểm: MyPoint

_ Em đã thử tìm cấu trúc Point trong thư viện matplotlib để kế thừa nhưng không tìm thấy. Nên em quyết định tự cài đặt lớp MyPoint để lưu các thông tin một điểm trong đồ thị.

2.1.1 Thuộc tính và phương thức:

Thuộc tính - Phương thức	Ý nghĩa

`def __init__(self, x, y):`
`  self.x = x`
`  self.y = y`
`  self.G = 0`
`  self.H = 0`
`  self.parent = None`
	▸ __init__ : hàm khởi tạo đối tượng trong Python
	▸ x, y: tọa độ điểm này
	▸ G: giá trị chi phí đi từ điểm Start tới điểm này, khởi tạo bằng 0.
	▸ H: giá trị heuristic của điểm này, khởi tạo bằng 0
	▸ parent: điểm cha của điểm này trên cây đường đi.

`def __eq__ (self, other):`	Phương thức định nghĩa toán tử so sánh bằng giữa 2 điểm. Hai điểm là bằng nhau khi chúng có cùng tọa độ.
`def movable_nodes(self, polygons):`	Phương thức phát sinh danh sách các điểm lân cận với điểm hiện tại, và thỏa điều kiện không nằm trong đa giác nào trong danh sách đa giác *polygons*. Cách hoạt động sẽ được trình bày bên dưới.
`def Distance(self, other):`	Phương thức tính khoảng cách Euclide từ điểm đang xét đến một điểm *other* khác.
`def move_cost(self, other):`	Phương thức tính chi phí đi từ điểm này tới điểm lân cận nó. Ở đây, ta coi đi chéo và đi ngang, dọc là như nhau nên

	phương thức này trả về giá trị 1. Ngược lại, đi chéo là căn 2 và đi ngang, dọc là 1.
`def Heuristic(self, goal):`	Phương thức tính giá trị Heuristic của điểm này. Cách tính sẽ được trình bày ở phần bên dưới.

2.1.2 Phương thức *movable_nodes(self, polygons):*

_ Chức năng: Phát sinh danh sách các điểm lân cận với điểm hiện tại, và thỏa điều kiện không nằm trong đa giác nào trong danh sách đa giác *polygons*.

_ Quanh một đỉnh có 8 đỉnh lân cận với nó. Với mỗi đỉnh lân cận đó, phương thức sẽ kiểm tra đỉnh đó có nằm trong đa giác nào hay không. Cuối cùng trả về danh sách những đỉnh lân cận mà không nằm trong đa giác nào.

_ Mã giả:

```
method movable_nodes(Đỉnh_đang_xét , Danh_sách_đa_giác) return Danh_sách_đỉnh_có_thể_mở
    Danh_sách_đỉnh_lân_cận ← Khởi_tạo_danh_sách_8_đỉnh_xung_quanh(Đỉnh_đang_xét)
    Danh_sách_đỉnh_có_thể_mở ← Tập_rỗng
    for Đỉnh_lân_cận in Danh_sách_đỉnh_lân_cận:
        if Đỉnh_lân_cận.Hoành_độ < 0 or Đỉnh_lân_cận.Tung_độ < 0 then:
            continue
        Đỉnh_này_có_thể_mở ← True
        for Đa_giác in Danh_sách_đa_giác:
            if Đa_giác.Chứa (Đỉnh_lân_cận) then:
                Đỉnh_này_có_thể_mở ← False
                break
        if Đỉnh_này_có_thể_mở = True then:
            Danh_sách_đỉnh_có_thể_mở ← Thêm (Đỉnh_lân_cận)
```

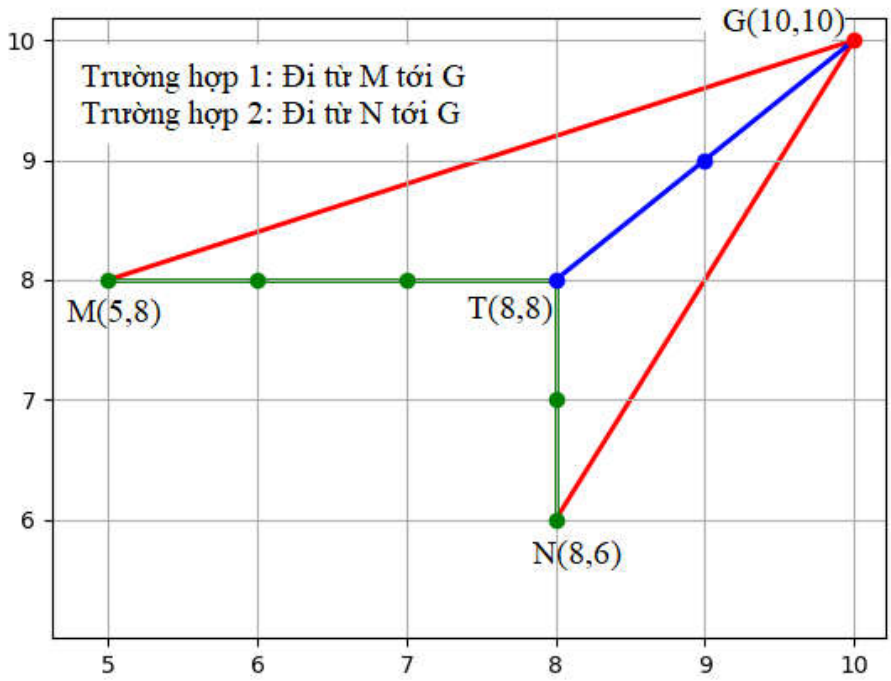
_ Trong phương thức này có gọi phương thức Chứa(*Đỉnh_lân_cận*) của lớp *Đa_giác*, sẽ được trình bày trong mục **2.2**.

2.1.3 *Phương thức Heutistic(self, goal):*

_ Chức năng: Tính giá trị Heuristic cho đỉnh đang xét.

_ Ý tưởng ban đầu: Heuristic đơn giản là khoảng cách Euclide từ đỉnh đang xét tới đỉnh goal. Tuy nhiên, em nhận thấy rằng, để đi từ một đỉnh tới Goal, vì mỗi

lần đi chỉ đi 1 ô, nên ta không thể đi theo đường chim bay, mà phải đi theo một trong 2 trường hợp:



Nhận xét: Trong cả 2 trường hợp, không thể đi trực tiếp theo các đường màu đỏ, mà phải đi theo các đường màu xanh lục tới điểm T, rồi mới từ T đến G theo các đường chéo màu xanh dương.

_ Từ nhận xét trên, thay vì tính giá trị heuristic là độ dài đường màu đỏ, em cho nó bằng tổng độ dài đường màu xanh lục và xanh dương. Nói cách khác, em tìm một điểm T sao cho từ T có thể đi chéo tới Goal, và tính tổng khoảng cách từ điểm đang xét tới T, và từ T tới Goal.

_ Mã giả:

```
method Heuristic (Đỉnh_đang_xét, Goal) return Giá_trị_Heuristic
    if Hiệu_x(Đỉnh_đang_xét, Goal) ≤ Hiệu_y(Đỉnh_đang_xét, Goal) then:
        if Đỉnh_đang_xét.x ≤ Goal.x then:
            Đỉnh_T.x ← Goal.x - Hiệu_y (Đỉnh_đang_xét, Goal)
        else:
            Đỉnh_T.x ← Goal.x + Hiệu_y (Đỉnh_đang_xét, Goal)
        Đỉnh_T.y ← Đỉnh_đang_xét.y
    else:
        if Đỉnh_đang_xét.y ≤ Goal.y then:
            Điểm_T.y ← Goal.y - Hiệu_x (Đỉnh_đang_xét, Goal)
```

else:

Điểm_T.y ← *Goal.y* + Hiệu_x (*Đỉnh_đang_xét, Goal*)

Đỉnh_T.y ← *Đỉnh_đang_xét.x*

Giá_trị_Heuristic ← Khoảng_cách(*Đỉnh_đang_xét, Đỉnh_T*) + Khoảng_cách(*Đỉnh_T, Goal*)

(Các hàm Hiệu_x, Hiệu_y kể trên trả về giá trị tuyệt đối của hiệu hoành độ và tung độ)

_ Cách tính này cho đường đi ngắn hơn so với cách tính heuristic bằng khoảng cách đường chim bay.

2.2 Cấu trúc dữ liệu lưu trữ một đa giác: MyPolygon

2.2.1 Thuộc tính và phương thức:

Thuộc tính - Phương thức	Ý nghĩa

`def __init__(self, vertices):`
`  self.vertices = vertices`
`  self.edges = self.getEdges()`
`  self.boundingbox = self.calculate_boundingbox()`
	▸ __init__ : hàm khởi tạo đối tượng trong Python
	▸ vertices: danh sách các đỉnh, mỗi đỉnh có kiểu là MyPoint
	▸ edges: danh sách các bộ 3 số thực (a,b,c) , là hệ số phương trình (có dạng ax + by + c = 0) chứa các cạnh của đa giác.
	▸ boundingbox: danh sách 2 đỉnh Phải_Trên và Trái_Dưới của hình chữ nhật bao quanh đa giác.

`def getEdges(self):`	Phương thức tính các hệ số (a,b,c) của phương trình chứa các cạnh của đa giác.
`def calculate_boundingbox(self):`	Phương thức tính 2 đỉnh của boundingbox.
`def contains_point(self, u, v):`	Phương thức kiểm tra đỉnh v có nằm trong đa giác này không, biết đỉnh u nằm ngoài đa giác.

2.2.2 *Phương thức getEdges(self):*

_ Chức năng: Tính các hệ số (a,b,c) của phương trình chứa các cạnh giác. Phương thức trả về mảng n dòng 3 cột, với n là số cạnh (cũng là số đỉnh), và 3 cột là 3 số thực a,b,c (vì phương trình đường thẳng có dạng *ax + by + c = 0*)

_ Lí do: Để kiểm tra một đỉnh có nằm trong đa giác không, có nhiều giải thuật. Em sử dụng giải thuật Ray Casting (tham khảo tại: https://en.wikipedia.org/wiki/Point_in_polygon). Giải thuật này cần phải tính

phương trình các cạnh của đa giác. Để khi chạy giải thuật, việc tính toán này không phải thực hiện lại nhiều lần, phương thức này được viết để tính luôn một lần ngay khi khởi tạo và lưu lại trong thuộc tính edges.

_ Ta đã biết tọa độ các đỉnh, nên việc viết phương trình đường thẳng đi qua các cạnh, mỗi cạnh chứa 2 đỉnh, là hoàn toàn đơn giản. Vì vậy em xin phép không trình bày mã giả của phương thức này.

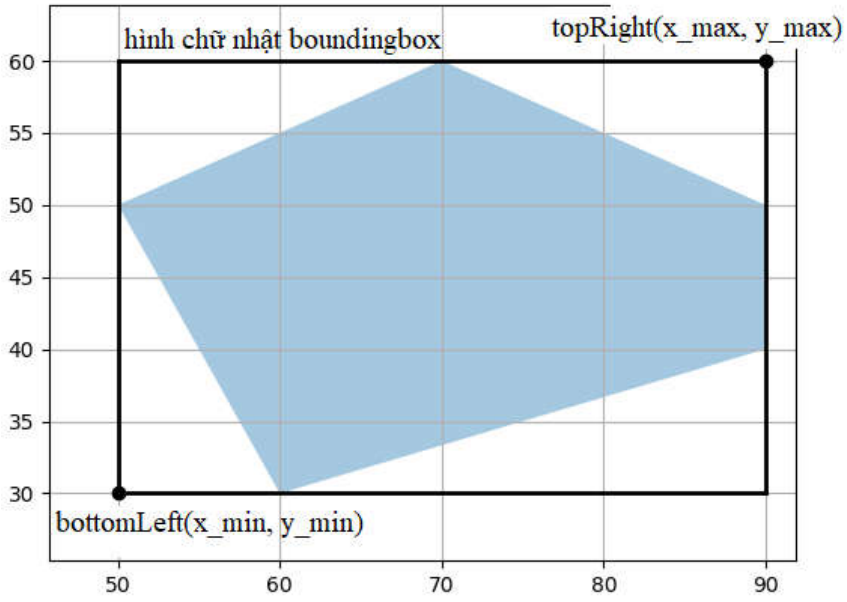
2.2.3 *Phương thức calculate_boundingbox(self):*

_ Chức năng: Tính 2 đỉnh của boundingbox, và lưu vào thuộc tính boundingbox.

_ Boundingbox là gì:

- Khi kiểm tra đỉnh nằm trong đa giác, có những đỉnh nằm xa đa giác, chắc chắn nó nằm ngoài đa giác, mà ta phải mất thời gian duyệt hết các cạnh mới biết nó nằm ngoài => boundingbox ra đời giúp tăng tốc việc kiểm tra này.

- Boundingbox là hình chữ nhật bao quanh đa giác, được biểu diễn bởi 2 đỉnh topRight và bottomLeft chứa tọa độ max và min của đa giác:



Nhận xét: Để kiểm tra một điểm nằm trong hay ngoài đa giác, trước khi duyệt các cạnh theo giải thuật Ray Casting, ta kiểm tra nếu x > x_max hoặc x < x_min hoặc y > y_max hoặc y < y_min , thì điểm đó nằm ngoài đa giác, không cần mất thời gian duyệt các cạnh.

_ Việc tính toán các tọa độ x_max, x_min, y_max, y_min được thực hiện ngay khi khởi tạo đa giác và lưu vào thuộc tính boundingbox, và chỉ đơn giản là duyệt các đỉnh, nên em sẽ không trình bày mã giả của phương thức này.

2.2.4 *Phương thức contains_point(self, u, v):*

_ Chức năng: Kiểm tra đỉnh v có nằm trong đa giác này không, biết đỉnh u nằm ngoài đa giác.

_ Lí do: Để mở tiếp các đỉnh lân cận của đỉnh đang xét khi tìm kiếm đường đi từ Start tới Goal, ta phải kiểm tra xem các đỉnh lân cận đó có nằm trong đa giác nào hay không. Để làm việc đó, ta duyệt tất cả đa giác, mỗi đa giác gọi phương thức contains_point của nó, nhận vào đỉnh u đã biết nằm ngoài và đỉnh v chưa biết, và trả về kết quả.

_ Ý tưởng ban đầu: Ban đầu em định sử dụng phương thức **contains_point** có sẵn của lớp **Path** của thư viện *matplotlib*. Tuy nhiên phương thức này cho kết quả False khi đỉnh v nằm trên một cạnh của đa giác, điều mà không đúng với yêu cầu đồ án. Vì vậy em tự cài đặt theo giải thuật Ray Casting (tham khảo tại: https://en.wikipedia.org/wiki/Point_in_polygon).

_ Mã giả:

```
method contains_point (Đa_giác, Đỉnh_u, Đỉnh_v) return True hoặc False
    if Nằm_ngoài_Boundingbox(Đỉnh_v) then:
        return False
    for Cạnh in Đa_giác.Danh_sách_cạnh:
        t_u ← Cạnh[0] * Đỉnh_u.x + Cạnh[1] * Đỉnh_u.y + Cạnh[2]
        t_v ← Cạnh[0] * Đỉnh_v.x + Cạnh[1] * Đỉnh_v.y + Cạnh[2]
        if t_v = 0 then:
            return True // v nằm trên cạnh đa giác
        if t_u * t_v > 0 then:
            continue
        a , b , c ← Phương_trình_đường_thẳng_qua ( Đỉnh_u, Đỉnh_v)
        t_A ← a * Cạnh.Đỉnh_1.x + b * Cạnh.Đỉnh_1.y + c
        t_B ← a * Cạnh.Đỉnh_2.x + b * Cạnh.Đỉnh_2.y + c
        if t_A * t_B ≤ 0 then:
            return True
    return False
```

(*Ghi chú:* Biến *Cạnh* nói trên là bộ mảng 3 phần tử đại diện cho 3 hệ số a, b, c đã trình bày ở mục **2.2.2**)

III. Giải thuật tìm kiếm:

1. Giải thuật tìm kiếm A*:

_ Chương trình sử dụng Giải thuật A* Search để tìm đường đi ngắn nhất từ Start tới Goal trên đồ thị.

_ Mã giả:

```
function A_Star_Search (Ds_đa_giác, Start, Goal) return Đường_đi
    Tập_đỉnh_đã_mở ← Tập_rỗng
    Tập_đỉnh_đã_thăm ← Tập_rỗng
    Start.H = Start.Heuristic(Goal)   // phương thức Heuristic đã được trình bày ở mục II.2.1.3
    Tập_đỉnh_đã_mở ← Thêm (Start)
    while Tập_đỉnh_đã_mở chưa_rỗng do:
        u ← Lấy_Đỉnh_Có_F_Min (Tập_đỉnh_đã_mở) // với F(u)  = G(u) + H(u)
        if u = Goal then:
            // trả về đường đi tìm được
            Đường_đi ← Tập_rỗng
            while u != NULL do:
                Đường_đi ← Thêm (u)
                u = u.Nút_Cha
            return Đường_đi
        Tập_đỉnh_đã_mở  ← Xóa (u)
        Tập_đỉnh_đã_thăm ← Thêm (u)
        Expand_ASS (Tập_đỉnh_đã_mở, Tập_đỉnh_đã_thăm, Ds_đa_giác, u, Goal)
    return NULL // không có đường đi thì trả về NULL
```

_ Trong đó, hàm Expand_ASS mở các đỉnh lân cận của đỉnh u, sao cho các đỉnh đó không nằm trong đa giác nào của *Ds_đa_giác*. Các đỉnh này được thêm vào *Tập_đỉnh đã_mở*. Hàm có mã giả như sau:

```
function Expand_ASS (Tập_đỉnh_đã_mở, Tập_đỉnh_đã_thăm, Ds_đa_giác, u, Goal)
    for v in u.movable_nodes (Ds_Đa_giác):
        // phương thức movable_nodes đã được trình bày ở mục II.2.1.2
        if v in Tập_đỉnh_đã_thăm then:
            continue
        if v in Tập_đỉnh_đã_mở then:
            Giá_trị_G_mới_của_v ←  u.G + u.move_cost(v)
            if Giá_trị_G_mới_của_v < v.G then:
                v.G ←  Giá_trị_G_mới_của_v
```

v.Nút_cha ← *u*

else:

v.G ← *u.G* + *u.*move_cost*(v)*

v.H ← *v.*Heutistic*(Goal)*

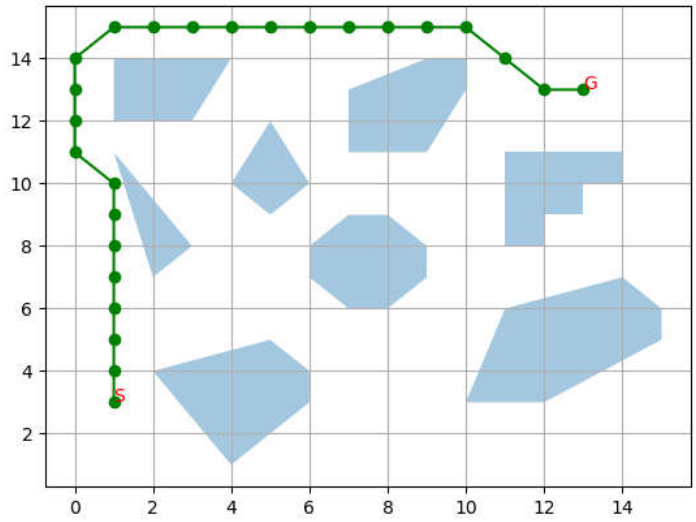
v.Nút_cha ← *u*

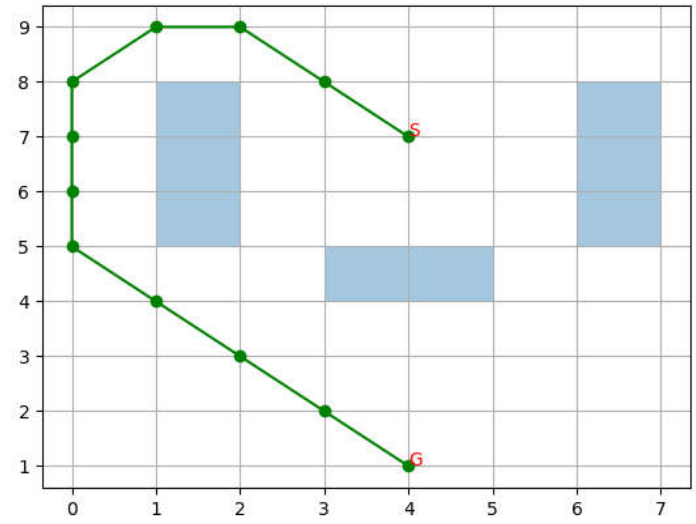
Tập_đỉnh_đã_mở ← Thêm (v)

2. Các ví dụ:

Ví dụ 1:

_ Với đồ thị kích thước nhỏ, giải thuật A* chạy tốt về mặt đường đi ngắn nhất và cả thời gian chạy.

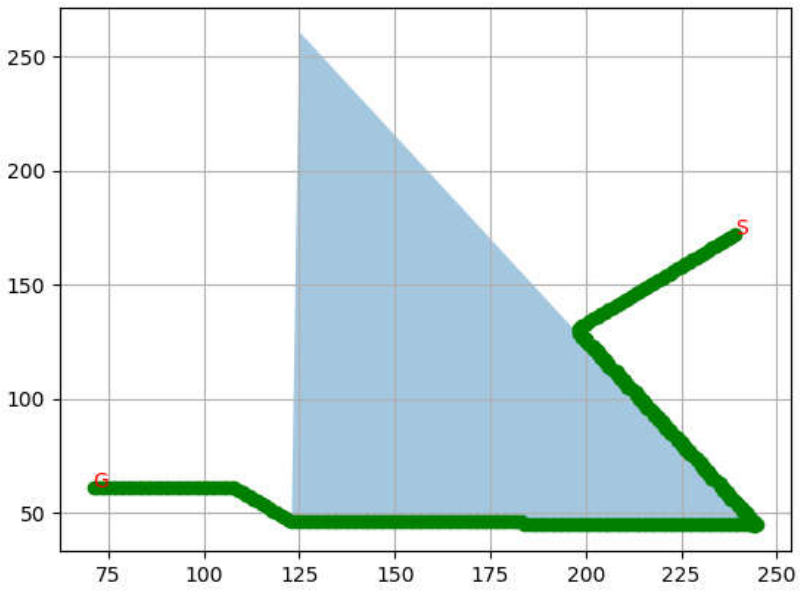
Input	Kết quả
8 1,3 13,13 1,14 4,14 3,12 1,12 7,13 9,14 10,14 10,13 9,11 7,11 5,12 6,10 5,9 4,10 11,11 14,11 14,10 13,10 13,9 12,9 12,8 11,8 1,11 3,8 2,7 7,9 8,9 9,8 9,7 8,6 7,6 6,7 6,8 5,5 6,4 6,3 4,1 2,4 11,6 14,7 15,6 15,5 12,3 10,3 Độ dài đường đi = 25.6568	

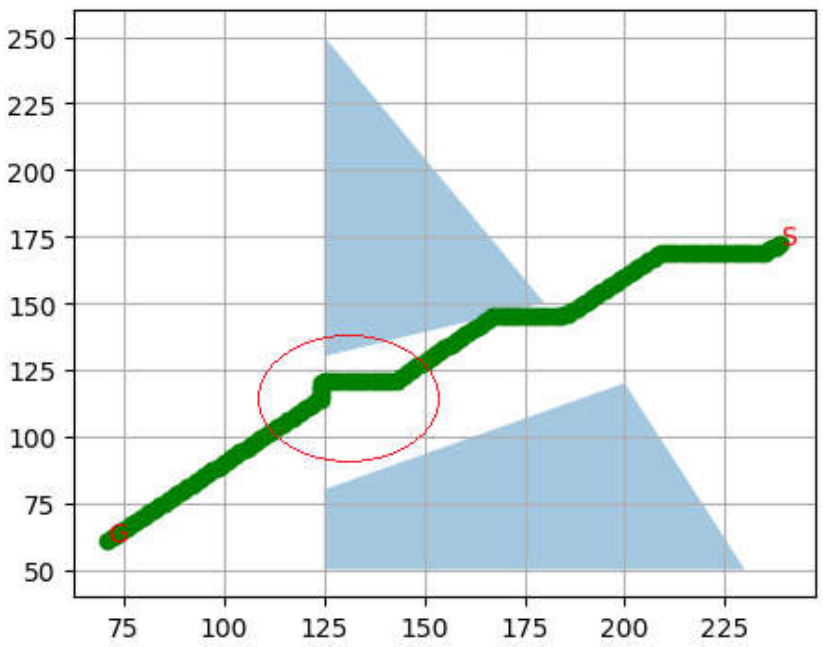
3 4,7 4,1 1,8 2,8 2,5 1,5 6,8 7,8 7,5 6,5 3,5 5,5 5,4 3,4 Độ đài đường đi = 13.9	

Ví dụ 2:

_ Với đồ thị có kích thước lớn, A* chạy chậm hẳn, do phải kiểm tra và mở quá nhiều đỉnh không cần thiết.

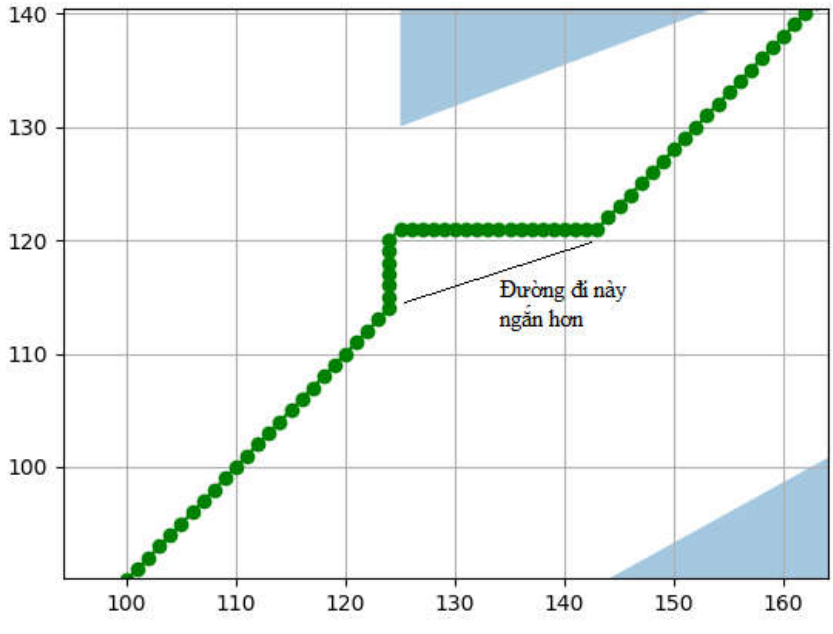
_ Với đồ thị lớn và có đa giác có kích thước lớn chắn ngang đường đi từ Start tới Goal, A* sẽ còn chạy chậm hơn rất nhiều vì phải đi đường vòng.

Input	Kết quả
1 239,172 71,61 123,47 244,45 125,261 Độ dài đường đi = 344.9066. Thời gian chạy: 452.2197 (s)	

2 239,172 71,61 125,250 125,130 180,150 125,80 125,50 230,50 200,120 Độ dài đường đi = 217.4924. Thời gian chạy: 28.872 (s) ** Vùng được khoanh đỏ sẽ được khảo sát ở Ví dụ 3*	

Ví dụ 3:

_ Giải thuật cho đường đi chưa tối ưu trong một số trường hợp. Xét lại input trên:

Input (lấy lại input trên)	Kết quả (như trên, nhưng phóng to vùng khoanh đỏ)
2 239,172 71,61 125,250 125,130 180,150 125,80 125,50 230,50 200,120 Độ dài đường đi: 82.81 Như ta thấy trong kết quả bên, trên đường đi kết quả có một đoạn không tối ưu, tức giải thuật cho đường đi ngắn hơn đường đi ngắn nhất thực sự.	

_ Em vẫn chưa tìm được hướng khắc phục tình huống này.

IV. Nhận xét - Mở rộng:

1. Nhận xét:

_ Giải thuật A* tỏ ra chậm chạp khi đồ thị có kích thước lớn hớn 100x100, và có đa giác lớn chắn ngang giữa đường đi từ Start tới Goal.

_ Em đã thử dùng Tham lam, cho tốc độ nhanh hơn nhiều , nhưng không tối ưu, tức đường đi mà Tham lam tìm được không phải đường đi ngắn nhất trong một số trường hợp.

_ Em cũng đã tìm hiểu về Bidirectional Search và áp dụng cho A* search, trong source code em có cài đặt, tuy nhiên Bi-AStar cho kết quả không ổn định, tức với đồ thị nhỏ thì nhanh hơn A* , nhưng với đồ thị lớn thì ngược lại.

2. Mở rộng:

Em sẽ cố gắng tìm hiểu thêm về Bidirectional Search và MultiProcessor để tăng tốc cho A*, cũng như tìm cách khắc phục tình huống ở ví dụ 3.

---------- HẾT ----------