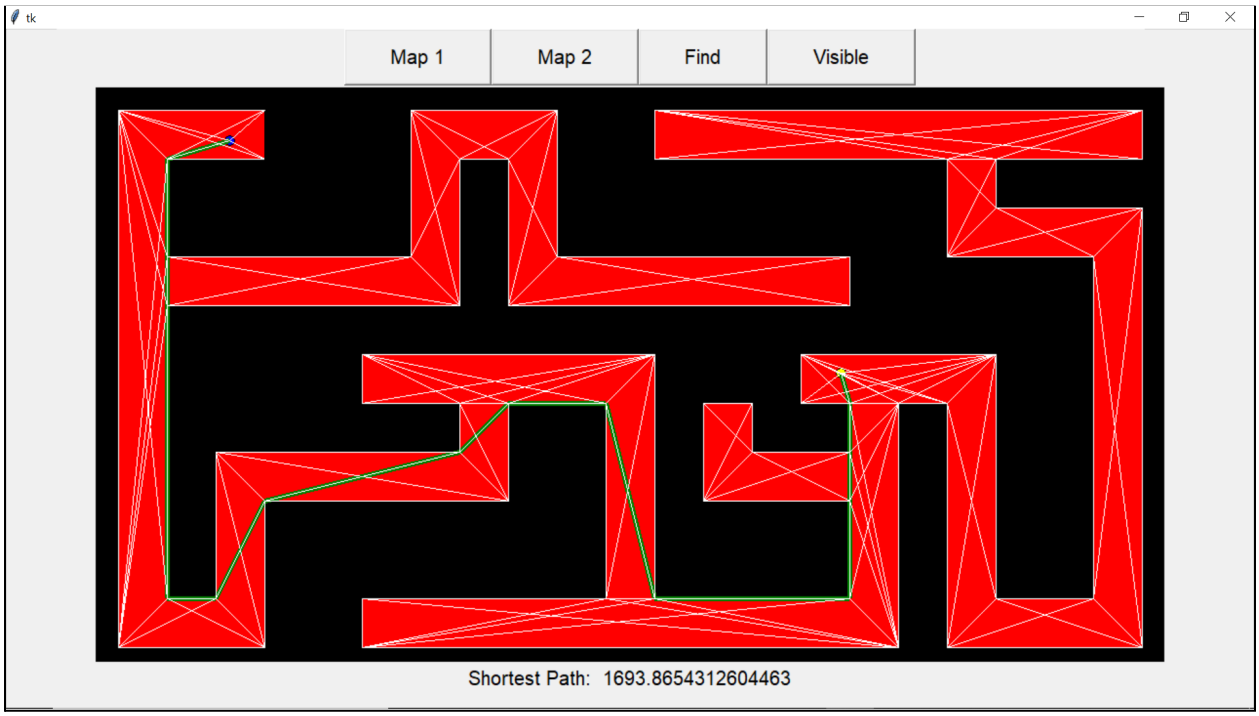
TẠO CHƯƠNG TRÌNH MÔ PHỎNG THUẬT TOÁN DIJKSTRA SỬ DỤNG PYTHON VÀ TKINTER

I. YÊU CẦU BÀI TOÁN

Sử dụng thư viện Tkinter của Python để tạo ra một giao diện đồ họa người dùng (GUI) mô phỏng thuật toán Dijkstra với các yêu cầu:

- Biểu hiện độ dài và hành trình của đường đi ngắn nhất giữa hai điểm cho trước hợp lệ trong bản đồ.
- Biểu hiện được những đường đi hợp lệ trong khu vực đi được.

II. TỔNG QUAN



Hình ảnh chương trình hiện tại

1) GIAO DIỆN:

a) <u>Bốn nút (Button):</u>

Cả bốn nút đều có kích cỡ là **40 pixels x 10 pixels**, kích cỡ chữ là 12, phông chữ mặc định và cùng một frame.

- Map 1: Biểu hiện bản đồ thứ nhất của chương trình
- Map 2: Biểu hiện bản đồ thứ hai của chương trình

- Find: Tìm hành trình đường đi ngắn nhất giữa 2 điểm đã nhập (đường xanh)
- Visible: Biểu hiện những đường đi hợp lệ, hay còn là đồ thị để thực hiện thuật toán Dijkstra (đường trắng)

b) Canvas:

- Canvas có hình nền màu đen, kích cỡ là **1096 pixels x 588 pixels**.
- Canvas là nơi chứa bản đồ, điểm đầu, điểm cuối, và các đường đi do các hoạt động vẽ và biểu hiện hình học phẳng chỉ có trong canvas.
- Bản đồ màu đỏ, hai điểm có màu xanh dương và vàng, hành trình đường đi ngắn nhất giữa hai điểm có màu xanh lá, đường đi hợp lệ (đồ thị thực hiện thuật toán Dijkstra) có màu trắng.

c) Kết quả:

Ghi lại độ dài đường đi ngắn nhất giữa 2 điểm.

2) NỘI DUNG:

a) Thư viện chính:

- Thư viện Tkinter: Thư viện lập trình GUI tiêu chuẩn của Python.
- Thư viện shapely: Thư viện lập trình phục vụ cho các bài toán liên quan đến hình học phẳng.

b) Bản đồ:

- Bản đồ là thành phần đầu tiên được vẽ trên canvas vì là nơi để tạo ra và chứa đựng các đỉnh nhập vào hoặc các đường đi.
- Bản đồ được lưu lại dưới dạng một list gồm các tọa độ đỉnh, tạo thành một đa giác. Các đỉnh của đa giác được sắp xếp theo thứ tự vẽ có thể được vẽ trên Canvas như sau

```
polygon_data = canvas.create_polygon([x1, y1, x2, y2, ...])
```

- Bên cạnh đó, để thực hiện dễ dàng hơn các bài toán liên quan tới việc lập đồ thị, ta sẽ lưu hình đa giác theo dạng **Polygon** trong shapely theo cú pháp

```
polygon = shapely.geometry.polygon.Polygon([x1, y1], [x2, y2], ...)
```

- Sau này, để xóa đi hình đa giác trên canvas, ta sẽ xóa như sau

```
canvas.delete(polygon_data)
```

c) Hai điểm được nhập vào:

- Hai điểm được nhập vào bằng chuột trái và chuột phải. Để thực hiện điều này, ta sử dụng ***bind()*** để gán một hàm hoặc thủ tục nhất định và thực hiện nó khi có một sự kiện xảy ra

```
main_window.bind('Sự kiện', hàm)
```

Cụ thể ở trường hợp này:

```
self.main_window.bind('<Button-1>', self.pressed1)
self.main_window.bind('<Button-3>', self.pressed3)
```

Trong đó:

★ <Button-1> thể hiện điều kiện là chuột phải được nhấn

★ <Button-3> thể hiện điều kiện là chuột trái được nhấn

★ pressed1, pressed3 là 2 thủ tục nhập điểm mới vào canvas

- Để biểu hiện hai điểm trên canvas, ta tạo thành hình oval trong tkinter:

```
canvas.create_oval(x - R, y - R, x + R, y + R)
```

- Để kiểm tra đỉnh được nhập có nằm bên trong bản đồ hay không, ta biến đổi đỉnh về kiểu **Point** trong tkinter, rồi sử dụng hàm **contains**.

```
this_point = shapely.geometry.Point(x, y)
check = polygon.contains(this_point)
```

d) Đường đi:

- Đường đi gồm hai loại: Hành trình của đường đi ngắn nhất và đường đi hợp lệ trong bản đồ.
- Để tạo ra những đường đi, ta sử dụng hàm trong tkinter:

```
line_data.append(canvas.create_line([x1, y1], [x2, y2]))
```

Trong đó, line_data là một list chứa đựng những thông tin của đường thẳng trên canvas. Việc chứa đựng như thế là để có thể xóa các đường đi khi tắt chế độ Visible.

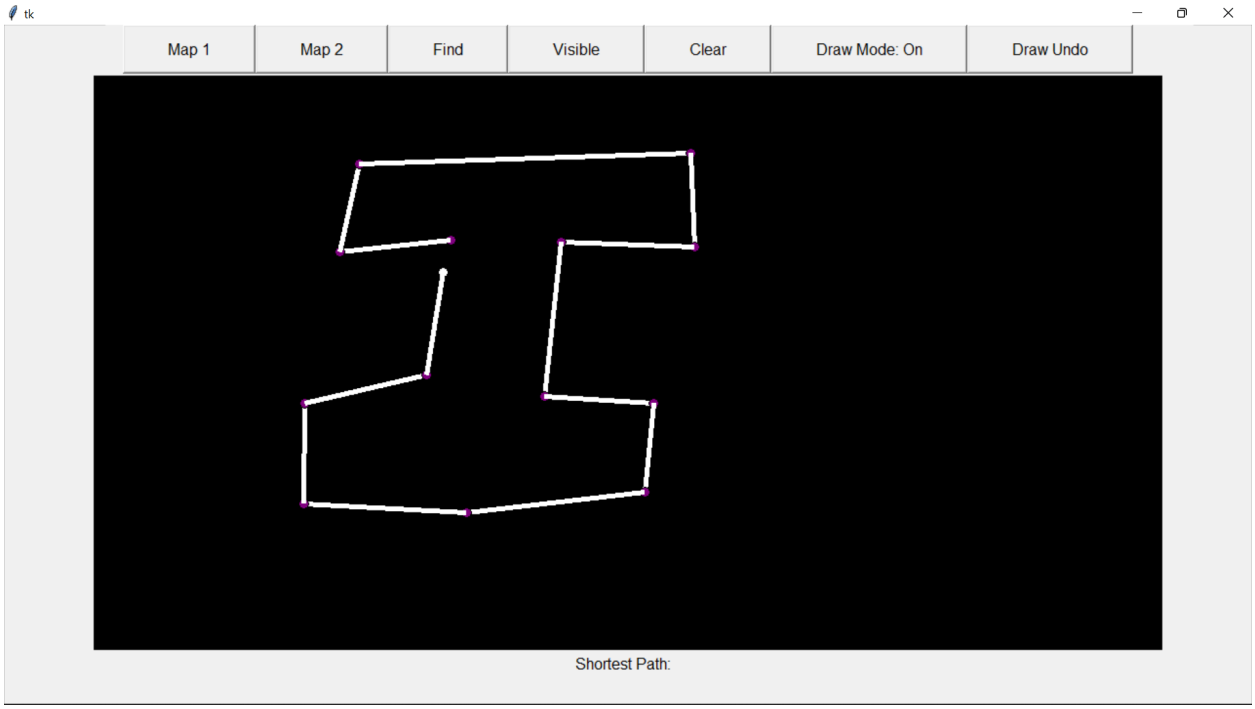
e) Mục ghi kết quả:

- Kết quả là một Widget Label trong đó có text (văn bản) là một biến có giá trị là **value**. Để thay đổi giá trị được in ra, ban đầu ta gán kết quả là **value**, sau đó với mỗi lần cập nhật ta sử dụng hàm **set()**

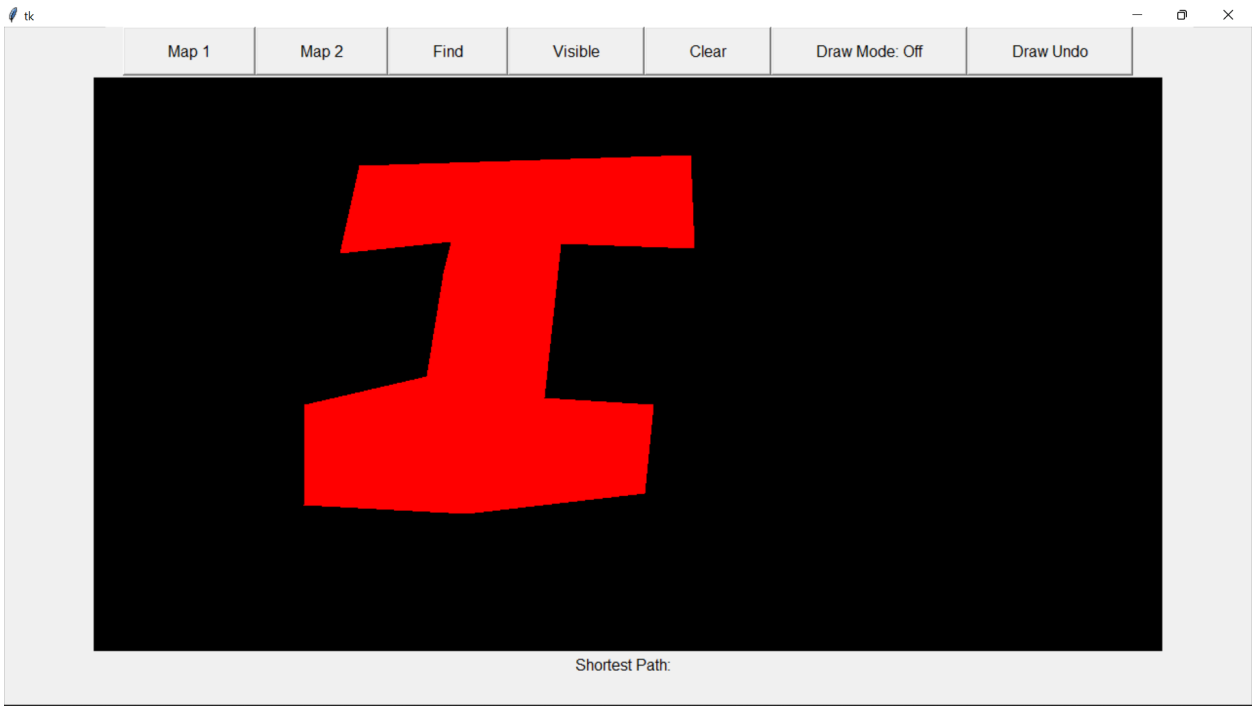
```
value = tkinter.StringVar()
Result = tkinter.Label(<frame>, textvariable=value,...)
```

f) <u>Chế độ vẽ:</u>

- Chế độ vẽ tạo điều kiện cho người dùng vẽ một mê cung mà họ muốn.
- Để kích hoạt chức năng, ta sẽ nhấn vào **Draw Mode: Off**. Chế độ vẽ sẽ biểu hiện những điểm được thêm vào và hình đa giác ta tạo ra dưới dạng các đường thẳng trong như thế nào.
- Để tạo ra bản đồ, ta nhấn vào **Draw Mode: On**. Bản đồ ta vẽ sẽ được tạo ra.
- Nếu phát hiện có điểm đầu và điểm cuối cắt đa giác, chương trình sẽ tự động thông báo cho người dùng.



Thao tác vẽ bản đồ



Bản đồ sau khi được vẽ xong và tắt chế độ vẽ sẽ tự động nối điểm đầu và cuối

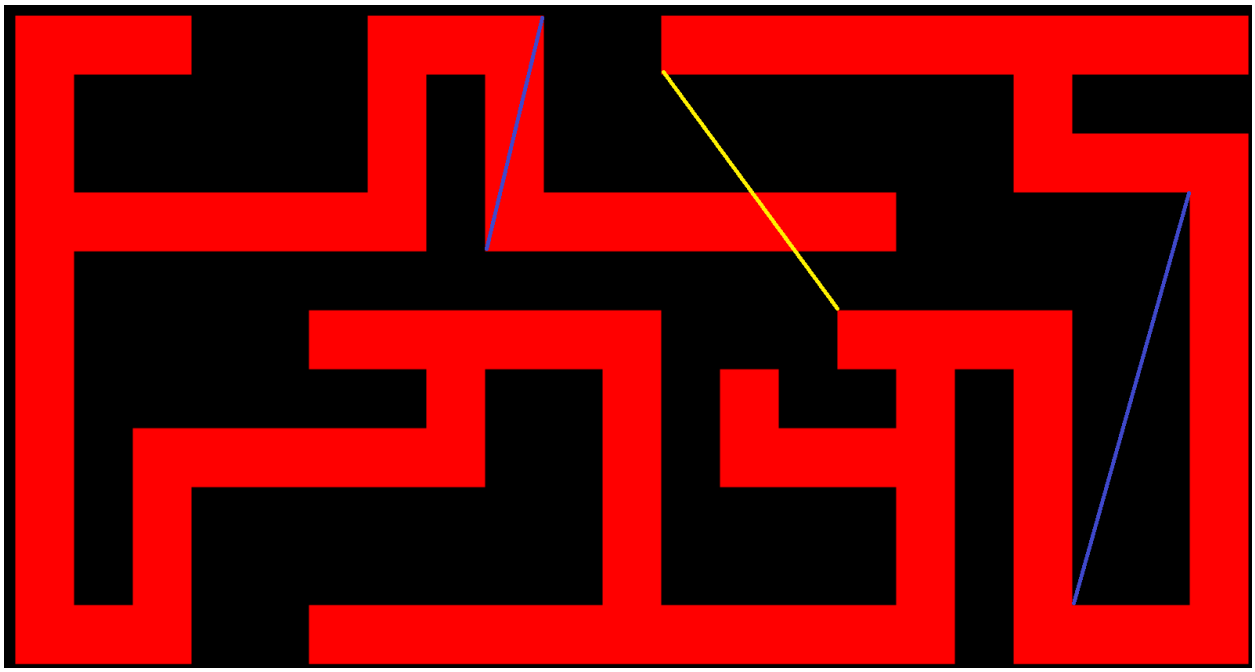
III. THUẬT TOÁN

1) XÂY DỰNG ĐỒ THỊ

- Để thực hiện thuật toán Dijkstra, việc xây dựng đồ thị là cần thiết. Ở đây, ta sẽ quy ước mỗi đường đi hợp lệ là một cạnh của đồ thị và mỗi điểm ở các góc là một cạnh.
- Đồ thị được lưu trữ dưới dạng danh sách kề vì đây là phương pháp lưu trữ tiện lợi nhất về mặt không gian và thời gian.
- Để thêm cạnh vào đồ thị, ta thực hiện việc kiểm tra hai đỉnh có tạo thành một đường đi nằm hoàn toàn trong bản đồ hay không. Một đường đi như thế có tính chất là
    + Đường đi này không cắt ngang qua bản đồ
    + Mọi điểm trên đường đi này nằm trên bản đồ
- Để kiểm tra đường đi có cắt ngang qua bản đồ hay không, ta sẽ lưu đường đi dưới dạng cấu trúc **LineString** trong shapely và sử dụng hàm **crosses()**

```
line = shapely.geometry.LineString([x1, y1], [x2, y2])
check = line.crosses(polygon)
```

- Để kiểm tra xem mọi điểm trên đường đi có nằm trên bản đồ hay không, ta đơn giản sử dụng một điểm bất kỳ trên đường đi rồi kiểm tra xem nó có nằm trong đa giác bằng hàm **contains()** như đề cập ở trên.
- Chứng minh tính đúng đắn:



- Sau khi kiểm tra xem các đường thẳng có giao nhau hay không, ta sẽ loại được hết các trường hợp đường đi màu vàng như trên hình, chỉ còn lại màu xanh. Đường đi màu xanh sẽ gồm hai loại: đường đi hoàn toàn bên trong và hoàn toàn bên ngoài bản đồ. Vì thế, ta có thể lấy một điểm bất kỳ (như trung điểm) để kiểm tra và luôn cho ra đồ thị chính xác.
- Khi thêm một điểm (nhập) vào bản đồ, ta có thể sử dụng thuật toán trên để tạo đường nối từ điểm input tới các đỉnh của đồ thị. Khi xóa một điểm input thì ta cũng sẽ xóa nó trên đồ thị.

Tổng kết: Như thế, việc tạo ra đồ thị được thực hiện như sau

```
    for i in range(1, len(self.p)):
    for j in range(i+1):
        if not LineString([self.p[i], self.p[j]]).crosses(self.polygon):
            midpoint = Point((self.p[i][0] + self.p[j][0]) / 2, (self.p[i][1] +
self.p[j][1]) / 2)
            if self.polygon.contains(midpoint):
```

```
                self.Line_List.append([j, i])
        for i in range(len(self.p) - 1):
            self.Line_List.append([i, i + 1])
        for line in self.Line_List:
            self.G.addEdge(line[0], line[1], ((self.p[line[0]][0] - self.p[line[1]][0]) **
2 + (self.p[line[0]][1] - self.p[line[1]][1]) ** 2) ** (1 / 2))
```

2) THUẬT TOÁN DIJKSTRA

Cho một đồ thị vô hướng có trọng số, tìm đường đi ngắn nhất giữa đỉnh start và end bất kỳ.

Ở đây, ta sử dụng các biến sau:

dist[x]: là khoảng cách giữa đỉnh start tới đỉnh x

Trace[x]: là đỉnh phía trước đỉnh x trong đường đi ngắn nhất

- Khởi tạo dist[start] = 0 và Trace[start] = start
- Chọn đỉnh u chưa được đánh dấu và có khoảng cách nhỏ nhất.
- Xét các đỉnh v kề với đỉnh u, nếu đường đi mới có khoảng cách nhỏ hơn đường đi cũ thì ta sẽ lưu trữ đường đi này

```
if (dist[v] > dist[u] + w):
        dist[v] = dist[u] + w
        Trace[v] = u
```

- Nếu còn đỉnh chưa được đánh dấu, hoặc là còn cập nhật được, thì quay lại bước 2.
- Để lưu lại đường đi, ta sẽ truy vết theo Trace. Ta sẽ trace cho tới khi ta gặp đỉnh start.

```
path = [end]
      while end != Trace[end]:
         end = Trace[end]
         path.append(end)
      path.reverse()
```

<u>Ngoài lề:</u>

- Để có thể thực hiện bước 2 nhanh hơn, thay vì tìm kiếm tuần tự xem đỉnh nào có ***dist*** nhỏ nhất, ta có thể sử dụng cấu trúc dữ liệu ***priority queue*** để

lấy ra giá trị lớn nhất, nhỏ nhất trong $O(logN)$. Sử dụng cấu trúc này có thể lấy ra các đỉnh bị dư (do việc cập nhật được thực hiện nhiều lần trước khi được xét), nhưng ta có thể đơn giản bỏ qua những đỉnh phía sau.

- Nếu sử dụng ***priority queue*** như là một ***max heap***, ta có thể lưu khoảng cách là -***dist*** để có thể lấy ra giá trị ***dist*** nhỏ nhất. Điều này là do dist càng nhỏ thì -***dist*** càng lớn.

Tổng kết: Như vậy, thuật toán Dijkstra là

```
def Dijkstra(self, start, end):
    dist = [1e9] * self.N
    Trace = [-1] * self.N
    dist[start] = 0
    Trace[start] = start
    pq = PriorityQueue()
    pq.put([0, start])
    while not pq.empty():
        [d, u] = pq.get()
        if dist[u] < -d:
            continue
        for edge in self.graph[u]:
            [v, w] = edge
            if dist[v] > dist[u] + w:
                Trace[v] = u
                dist[v] = dist[u] + w
                pq.put([-dist[v], v])
    E = end
    path = [end]
    while end != Trace[end]:
        end = Trace[end]
        path.append(end)
    path.reverse()
    return [dist[E], path]
```

IV. BÁO CÁO TIẾN ĐỘ

Tuần 1 (15/2 - 21/2)	• Nghiên cứu cách sử dụng Tkinter và lập trình một giao diện cơ bản

Tuần 2 (22/2 - 1/3)	• Biết về thư viện Shapely và lập trình các chức năng
Ngày 7/3/2022	• Thêm Chế Độ Vẽ

V. HẠN CHẾ/ HƯỚNG PHÁT TRIỂN

- Bản đồ hiện tại chỉ có hình dạng cây, không có lỗ ở giữa
- Chưa có chức năng tự vẽ bản đồ hay sinh bản đồ ngẫu nhiên
- Giao diện còn quá đơn giản