BÁO CÁO VỀ CHƯƠNG TRÌNH TÌM KIẾM ĐƯỜNG ĐI NGẮN NHẤT GIỮA HAI ĐIỂM TRONG BẢN ĐỒ

Cao Quang Nhật Khoa - 12 Tin - 10

Trường Phổ Thông Năng Khiếu Cơ Sở 1

I. Yêu cầu

- Tổng quát: Sử dụng ngôn ngữ thư viện Tkinter của Python để tạo ra một chương trình giao diện đồ họa tìm đường đi ngắn nhất giữa 2 điểm trong đa giác

- Cụ thể:

+ Tạo ra một đa giác và hiển thị được khu vực của nó trên Canvas (bảng vẽ)
+ Vẽ ra được những đường đi giữa 2 đỉnh bất kỳ nằm trong khu vực đa giác ấy. Đồng thời có nút bấm để ẩn/hiện chúng
+ Chọn được 2 điểm bắt đầu, kết thúc trong đa giác và tính được quãng đường ngắn nhất bằng thuật toán Dijkstra
+ Xuất ra được thời gian chạy của các thuật toán và độ dài quãng đường giữa 2 điểm bắt đầu, kết thúc
+ Tạo ra ít nhất 2-3 đa giác khác nhau trên bảng vẽ.

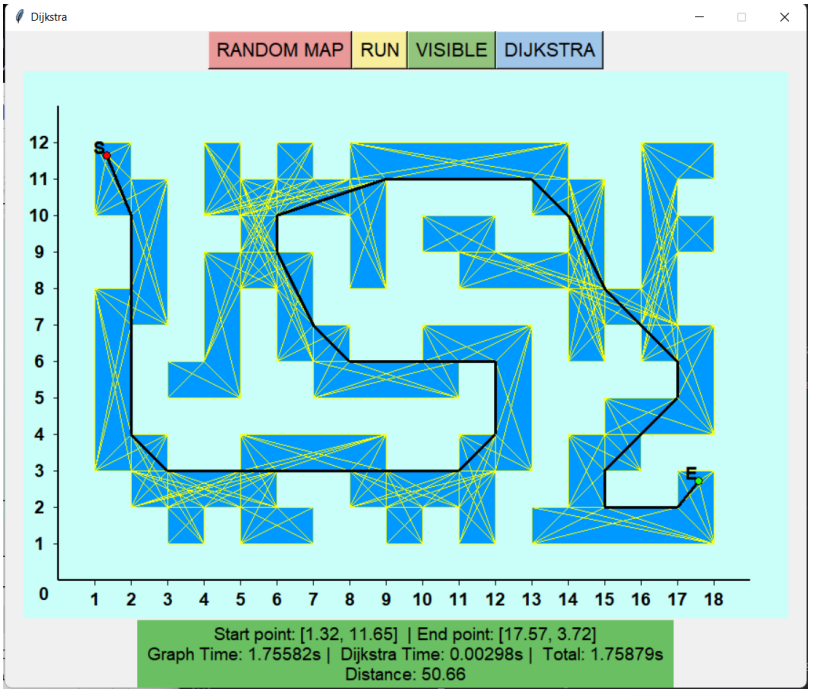
- Báo cáo: Sau 3 tuần, chương trình đã thỏa được các điều trên. Ở mục cuối cùng, thay thế việc tạo ra 3 đa giác có sẵn, chương trình này có khả năng tạo ra vô tận loại bản đồ khác nhau một cách ngẫu nhiên (12 x 18)

II. Sơ lược về chương trình

a. Môi trường - Thư viện sử dụng

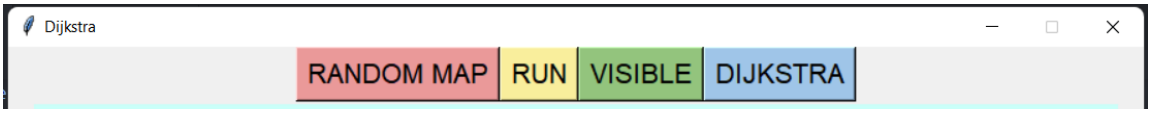
- Chương trình được viết bằng ngôn ngữ lập trình Python.
- Các thư viện được sử dụng:
    + Tkinter: Thư viện tạo giao diện đồ họa người dùng (GUI)
    + Math: Thư viện cung cấp các phép tính phức tạp có sử dụng trong chương trình như sqrt(),...
    + Time: Thư viện dùng để đo thời gian chạy các thuật toán như Dijkstra, vẽ bản đồ,...
    + Queue: Thư viên cung cấp cấu trúc dữ liệu Hàng đợi ưu tiên (Priority Queue) cho thuật toán Dijkstra
    + Shapely: thư viện tạo các điểm, đường thẳng, hình đa giác,... Ở chương trình này, thư viện dùng chủ yếu để kiểm tra đường thẳng có cắt qua hình đa giác không.

b. Giao diện



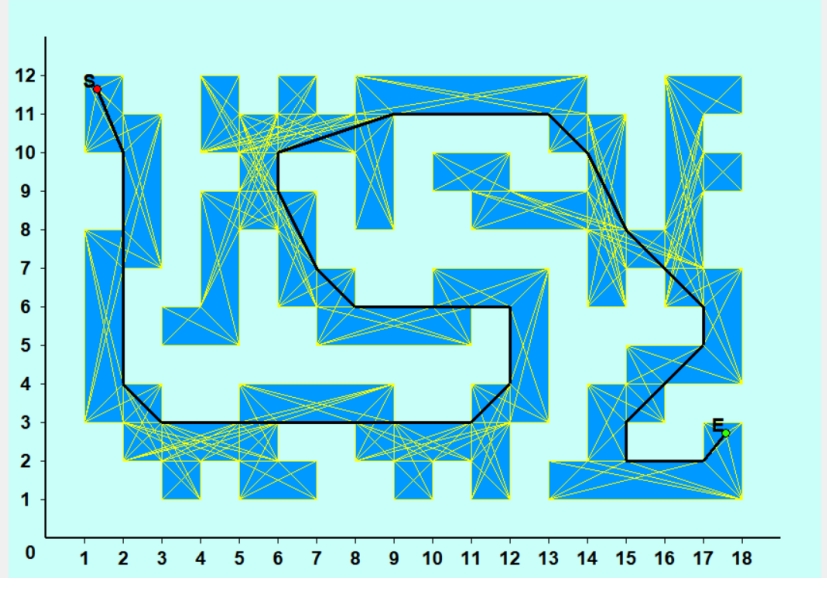
Giao diện chương trình hoàn chỉnh

- Giao diện chương trình được chia ra thành 3 phần:
  + Các nút bấm event:



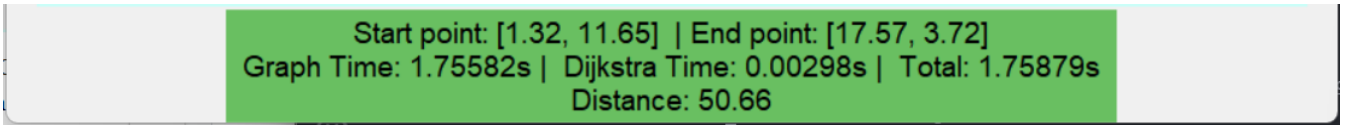
- RANDOM MAP: tạo bản đồ mê cung mới ngẫu nhiên
- RUN: Dựng các đường chạy giữa 2 đỉnh bất kỳ nằm trong khu vực mê cung
- VISIBLE: ẩn/hiện đồ thị trên
- DIJKSTRA: chạy thuật toán Dijkstra tìm đường đi ngắn nhất giữa điểm bắt đầu và kết thúc

  + Bảng vẽ:



- Bao gồm bản đồ (mê cung), đường đi giữa 2 điểm bất kỳ trong bản đồ, điểm bắt đầu và kết thúc (người dùng tùy ý chọn vị trí của chúng)

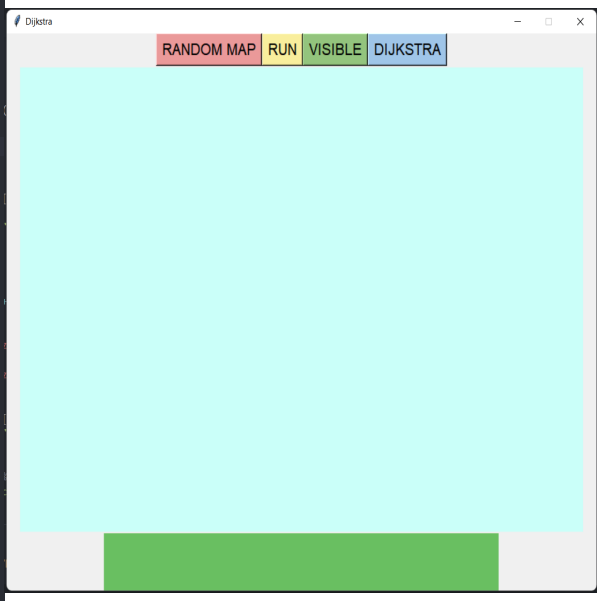
+ Bảng thông tin:



- Start/End point: tọa độ điểm bắt đầu/kết thúc
- Graph time: thời gian để tạo đồ thị đường giữa 2 đỉnh nằm trong bản đồ
- Dijkstra time: thời gian chạy thuật toán Dijkstra
- Total: tổng thời gian để tạo ra một chương trình hoàn chỉnh và vẽ ra đường đi ngắn nhất từ Start point đến End point
- Distance: quãng đường ngắn nhất từ Start point đến End point mà Dijkstra tìm ra

- Chương trình sử dụng Frame của Tkinter để chứa các Widget(Button, Label,...) cũng như các hàm pack() để vẽ ra Widget lên màn hình chính ngay khi ta cho chạy chương trình.

```
class myGUI():
    def __init__(self,app):
        top_frame = tkinter.Frame(app)
        bot_frame = tkinter.Frame(app)
        app.title("Dijkstra")
        # setting window size
        width=880
        height=720
        size_str = '%dx%d' % (width, height)
        app.geometry(size_str)
        app.resizable(width=False, height=False)

        self.graph_time, self.dijkstra_time, self.distance = -1,-1,-1
        self.res = []

        but_map = tkinter.Button(top_frame,text="RANDOM MAP",font=tkinter.font.Font(size=16),command=self.draw_graph, bg = '#ea9999')
        but_run = tkinter.Button(top_frame,text="RUN",font=tkinter.font.Font(size=16),command=self.VisibleMode, bg = '#f9ee9c')
        but_dji = tkinter.Button(top_frame,text="DIJKSTRA",font=tkinter.font.Font(size=16),command=self.dijkstra, bg = '#9fc5e8')

        but_vis = tkinter.Button(top_frame,text="VISIBLE",font=tkinter.font.Font(size=16),command=self.checkVisible,bg = '#93c47d')
        self.visible = False

        self.canvas = tkinter.Canvas(bot_frame,width=840,height=600,background='#CAFFF9')

        self.canvas.bind("<Button-1>", self.addStart)
        self.canvas.bind("<Button-3>", self.addEnd)
        self.checkStart, self.checkEnd, self.EinP,self.SinP,self.checkGraph, self.checkMap = False,False,False,False,False,False

        self.infoText = tkinter.Label(bot_frame, font=tkinter.font.Font(size=14),text="", bg="#69BF61", width=53, height=25)

        but_map.pack(side="left", fill="both", expand=True)
        but_run.pack(side="left", fill="both", expand=True)
        but_vis.pack(side="left", fill="both", expand=True)
        but_dji.pack(side="left", fill="both", expand=True)
        self.canvas.pack()
        top_frame.pack()
        bot_frame.pack()
```



Code khởi tạo giao diện người dùng (GUI) khi chạy chương trình

- Chương trình sử dụng Canvas để vẽ ra đa giác, vẽ ra đường đi được và đánh dấu điểm bắt đầu, kết thúc và và đường đi ngắn nhất. Cụ thể hơn, mỗi khi người dùng có tác động làm thay đổi bảng vẽ (VD thay thế điểm bắt đầu, tạo map mới,...), chương trình sử dụng **canvas.delete("all")** để xóa toàn bộ hình vẽ hiện tại và vẽ lại theo đúng ý của

người dùng. Nếu không có hàm này thì bảng vẽ sẽ bị đè bởi nhiều giao diện khác nhau khiến giao diện trở nên chằng chịt, thậm chí có thể sập chương trình nếu đè quá nhiều.

```
def draw_graph(self, redraw = True):
    self.canvas.delete("all")
    if redraw: self.gen_map()
    else: self.canvas.create_polygon(maze.pointsW[2:len(maze.pointsW)],fill = "#0099FF")
    self.draw_line(0, 0, 0, nrow, "black", 2)
    self.draw_line(0, nrow, ncolumn, nrow, "black", 2)

    if (self.visible):
        for i in self.vertices:
            for j in self.vertices.get(i):
                self.draw_line(maze.points[i][0] + 1, maze.points[i][1] + 1, maze.points[j][0] + 1, maze.points[j][1] + 1, "yellow", 1.25)

    if (self.checkStart):
        self.draw_point(maze.points[0][0] + 1, maze.points[0][1] + 1, "red", "S")
    if (self.checkEnd):
        self.draw_point(maze.points[1][0] + 1, maze.points[1][1] + 1, "#33FF00", "E")
    if self.res:
        for i in range(0,len(self.res)-1):
            x1, y1, x2, y2 = maze.points[self.res[i]][0]+1, maze.points[self.res[i]][1]+1, maze.points[self.res[i+1]][0]+1, maze.points[self.res[i+1]][1]+1
            self.draw_line(x1, y1, x2, y2, "black", 3)

    for i in range(1, nrow):
        #self.draw_point(0, i, color="black", t="%d" % (nrow-i))
        self.draw_line(0, i, -0.15, i, color="black", w = 1, n = nrow - i)
    self.canvas.create_text(-0.15 * 40 + 30, nrow * 40 + 55, text= 0, font=tkinter.font.Font(size=14, weight='bold'))
    for i in range(1, ncolumn):
        #self.draw_point(i, nrow, color="black", t="%d" % i)
        self.draw_line(i, nrow, i, nrow + 0.15, color="black", w = 1, n = i)
```

Hàm draw_graph() dùng để xóa đi vẽ lại GUI theo sự kiện người dùng đã chọn

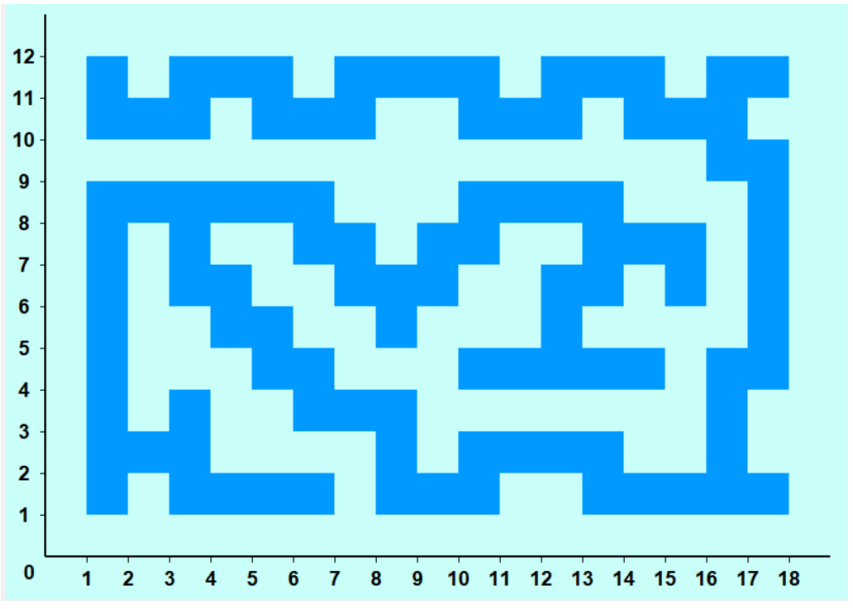
c. Bản đồ (Hình đa giác)

Bản đồ là 1 phần của Canvas (bảng vẽ), là phần quan trọng nhất vì nó cần thiết để thể hiện các thông tin chính của chương trình như tường, tọa độ các đỉnh...

Ban đầu, chương trình sử dụng thuật toán Randomized DFS (code nằm ở mục II) để vẽ một bản đồ dạng bit map, với 0 là tường và 1 là đường đi của bản đồ. Sau đó, hàm switch_to_tkinter() sẽ lấy bitmap ấy và tạo ra points[], một List những đỉnh (một list chứa 2 tọa độ x, y) tạo thành một đa giác.

Sau đó, sử dụng Canvas.create_polygon(), các đỉnh của đa giác ấy sẽ được vẽ lên bảng vẽ một cách dễ dàng (như hình dưới)

```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 1 0 1 1 1 0 1 1 1 1 0 1 1 1 0 1 1 0
0 1 1 1 0 1 1 1 0 0 1 1 1 0 1 1 1 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0
0 1 1 1 1 1 1 0 0 0 1 1 1 1 0 0 0 1 0
0 1 0 1 0 0 1 1 0 1 1 0 0 1 1 1 0 1 0
0 1 0 1 1 0 0 1 1 1 0 0 1 1 0 1 0 1 0
0 1 0 0 1 1 0 0 1 0 0 0 1 0 0 0 0 1 0
0 1 0 0 0 1 1 0 0 0 1 1 1 1 1 0 1 1 0
0 1 0 1 0 0 1 1 1 0 0 0 0 0 0 0 1 0 0
0 1 1 1 0 0 0 0 1 0 1 1 1 1 0 0 1 0 0
0 1 0 1 1 1 1 0 1 1 1 0 0 1 1 1 1 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```



d. Điểm đầu/cuối

Để vẽ ra điểm đến và đi trong bản đồ từ cú nhập chuột trái/phải, chương trình sử dụng Canvas.bind() để gán một hàm vẽ ra điểm tròn và tên điểm ra bản đồ khi sự kiện cụ thể xảy ra (Button-1: chuột phải được nhấp, Button-3: chuột trái được nhấp)

```
def addStart(self, event):
    if self.checkMap:
        print(event.x, event.y)
        self.res.clear()
        maze.pointsW[0] = [event.x,event.y]
        location = [(event.x-40)/40 - 1, (event.y-40)/40 - 1]
        maze.points[0] = location
        self.checkStart = True
        self.UpdateInfo()
        self.VisibleMode(1)

def addEnd(self, event):
    if self.checkMap:
        print(event.x, event.y)
        self.res.clear()
        maze.pointsW[1] = [event.x,event.y]
        location = [(event.x-40)/40 - 1, (event.y-40)/40 - 1]
        maze.points[1] = location
        self.checkEnd = True
        self.UpdateInfo()
        self.VisibleMode(1)
```

```
self.canvas.bind("<Button-1>", self.addStart)
self.canvas.bind("<Button-3>", self.addEnd)
```

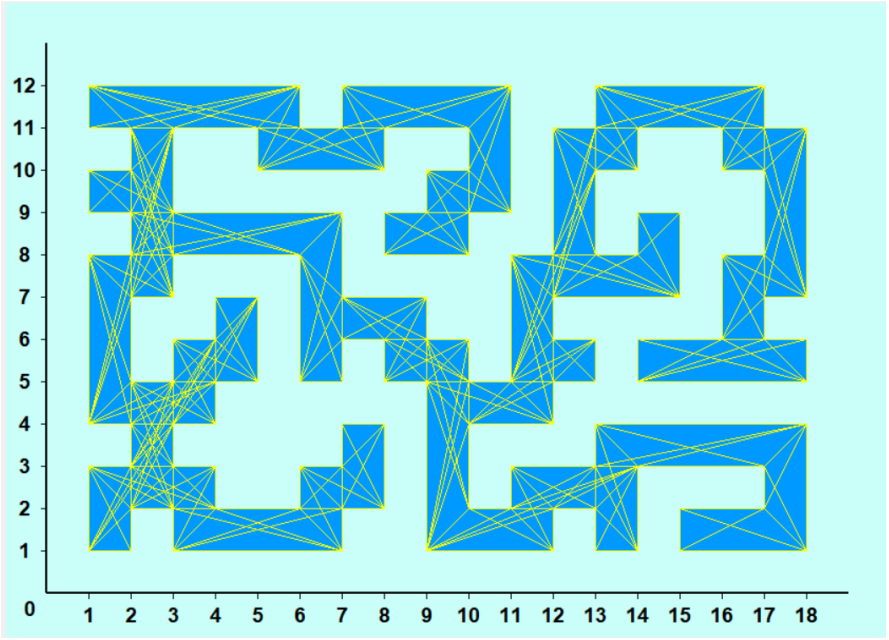
e. Đồ thi các đường đi được trong bản đồ

Trước khi tìm đường đi ngắn nhất, ta cần phải tìm các đường đi giữa 2 đỉnh nằm trong đa giác. Sở dĩ làm vậy là vì ta biết đường đi ngắn nhất chắc chắn sẽ nằm trong các đường đi ấy. Với List tọa độ các đỉnh trong đa giác, ta có thể kiểm tra ngay nếu đường nối giữa 2 đỉnh bất kỳ nằm trong đa giác như sau:

+ Sử dụng thư viện ngoài Shapely và các hàm của thư viện là Polygon(), Point(), Line() để tạo ra hình đa giác và các đường nối giữa 2 đỉnh bất kỳ trong môi trường Shapely.
+ Sử dụng hàm crossses (VD path.crosses(main_polygon)) để kiểm tra đường đi có cắt qua cạnh nào của đa giác không. Trường hợp 2 điểm kề nhau thì mặc định đúng
+ Sau khi chạy xong, sử dụng Canvas.create_line(), ta sẽ vẽ ra được những đường đi được trong đa giác

```
for i in range(s,e):
    for j in range(i+1,l):
        path = LineString([maze.pointsW[i],maze.pointsW[j]])
        point = path.interpolate(0.5)
        if not path.crosses(main_polygon) and (main_polygon.contains(point) or (j-i==1 and i!=0 and i!=1) or (i == 2 and j == l-1)):
                vertLen = [j, math.sqrt((maze.points[i][0]-maze.points[j][0])**2 + (maze.points[i][1]-maze.points[j][1])**2)]
                self.graph[i].append(vertLen)
                vertLen = [i, math.sqrt((maze.points[i][0]-maze.points[j][0])**2 + (maze.points[i][1]-maze.points[j][1])**2)]
                self.graph[j].append(vertLen)
                #if (i==0 and j==1): self.linkSE = True
                #if (i==0):
                #    print(j,self.graph[j])
                #self.draw_line(maze.points[i][0]+1, maze.points[i][1]+1, maze.points[j][0]+1, maze.points[j][1]+1, "yellow", 1.25)
                self.vertices[i].append(j)
```

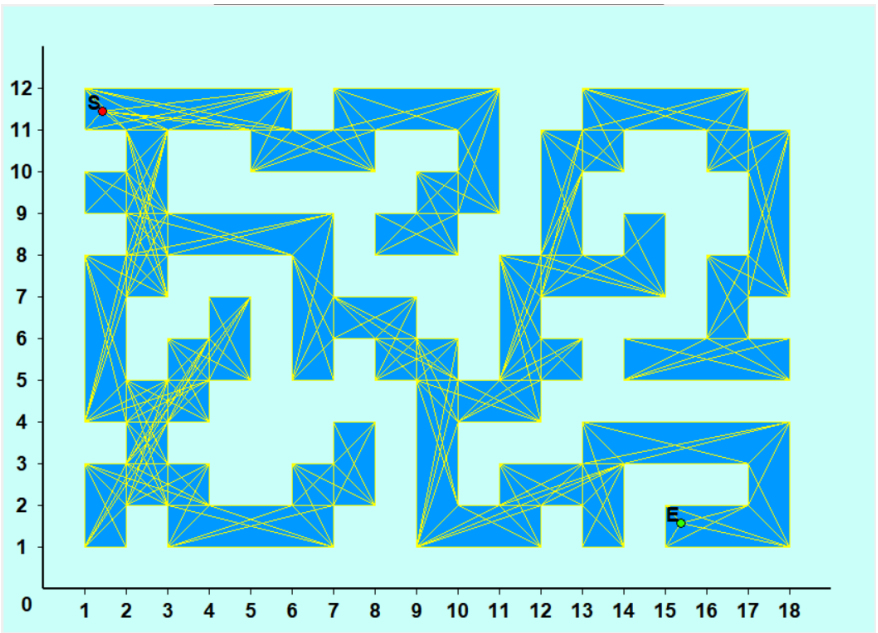
Code kiểm tra nếu đường thẳng 2 đỉnh bất kỳ có giao với cạnh nào của polygon



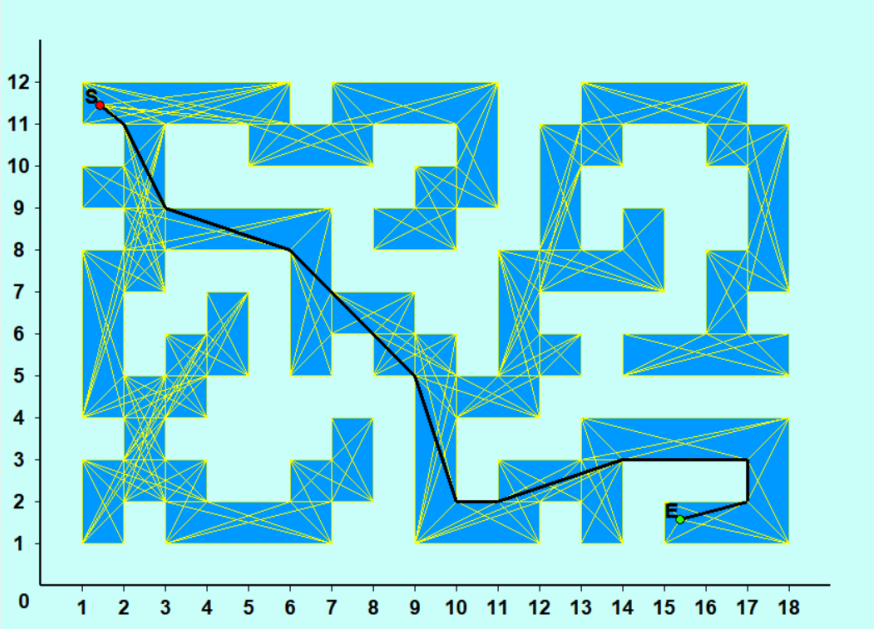
Kết quả của hàm trên

f. 2 điểm bắt đầu, kết thúc

Như mục trên, ta cũng sẽ phải tìm đường nối giữa điểm bắt đầu/kết thúc với các điểm của đa giác (kể cả điểm giữa bắt đầu và kết thúc) với phương pháp tương tự:



Sau khi có được đồ thị liên kết, ta có thể sử dụng thuật toán Dijkstra (code nằm ở mục II) để tìm đường đi ngắn nhất:



Thuật toán Dijkstra tìm đường đi ngắn nhất từ S đến E

g. Thời gian chạy và quãng đường

Thời gian chạy thuật toán Dijkstra và Graph và độ dài quãng đường đều được hiện lên ở phần thông tin nhờ sử dụng tkinter.Text() được gắn từ nhiều chuỗi string khác nhau. Qua đó mỗi khi ta cần cập nhật một thứ gì đó, như thời gian hay địa điểm Start/End point, ta chỉ việc gọi hàm UpdateInfo().

```
def UpdateInfo(self):
    text = ""
    if self.checkStart:
        text = "Start point: [" + ("%.2f" % (maze.points[0][0]+1)) + (", %.2f" % (nrow - 1 - maze.points[0][1])) + "]  |"
    if self.checkEnd:
        text += "End point: [" + ("%.2f" % (maze.points[1][0]+1)) + (", %.2f" % (nrow - 1 - maze.points[1][1]+1)) + "]\n"
    if self.graph_time != -1:
        text += "Graph Time: " + ("%.5f" % self.graph_time) + "s |"
    if self.dijkstra_time != -1:
        text += "  Dijkstra Time: " + ("%.5f" % self.dijkstra_time) + "s |"
    if self.dijkstra_time != -1 and self.graph_time != -1:
        text += "  Total: " + ("%.5f" % (self.dijkstra_time + self.graph_time)) + "s\n"
    if self.distance != -1:
        text += "Distance: " + ("%.2f" % self.distance)
    self.infoText["text"] = text
```

Và để tính thời gian chạy của các thuật toán, ta sử dụng thư viện **time** của Python và gọi chúng ở đầu và cuối thuật toán, từ đó hiệu của chúng sẽ là thời gian chạy thuật toán ấy (tính bằng giây).

```
def dijkstra(self):
    timeStart = time.time()
    if (self.checkStart and self.checkEnd and self.check
```

```
    self.dijkstra_time = time.time() - timeStart
```

Hàm gọi thời gian ở đầu và cuối hàm Dijkstra

III. Thuật toán hình học

a. Tạo bản đồ ngẫu nhiên

Để tạo ra một bản đồ ngẫu nhiên với số cột và hàng theo người dùng muốn:

- Tạo một bản đồ ban đầu bằng mảng kí tự qua thuật toán DFS ngẫu nhiên:
    1. Tạo một mảng list 2 chiều tên **map** (số dòng x số cột) gồm toàn bộ số 0. Chọn ngẫu nhiên 1 điểm và bỏ nó vào list các điểm đi qua (hoạt động giống Stack)
    2. Xét các điểm bên cạnh nó mà chưa được đi qua và chọn ngẫu nhiên 1 điểm. Bỏ nó vào list các điểm đi qua.
    3. Tiếp tục lặp lại bước 2 với điểm vừa chọn. Trong trường hợp cả 4 phía của điểm đều đã đi qua, bỏ nó ra khỏi Stack.
    4. Bước 2-3 lặp lại đến khi list các điểm đi qua không còn phần tử.

→ map là bản đồ ban đầu được thể hiện dạng kí tự

```
def set_up_maze(self):
    scr = randint(1, nrow-2)
    scc = randint(1, ncolumn-2)
    self.start_color = 'Green'
    ccr, ccc = scr, scc

    map[ccr][ccc] = '1'
    finished = False
    while not finished:
        visitable_neighbours = self.check_neighbours(ccr, ccc)
        if len(visitable_neighbours) != 0:
            d = randint(1, len(visitable_neighbours))-1
            ncr, ncc = visitable_neighbours[d]
            map[ncr][ncc] = '1'
            visited_cells.append([ncr, ncc])
            ccr, ccc = ncr, ncc
        if len(visitable_neighbours) == 0:
            try:
                ccr, ccc = visited_cells.pop()
            except:
                finished = True
    self.create()
```

b. Thuật toán tìm đường đi ngắn nhất Dijkstra

Xét đồ thị G =(X,E) với các cạnh có trọng số không âm.

- Dữ liệu nhập cho thuật toán là ma trận trọng số L (với qui ước L(h,k) = +∞ nếu không có cạnh nối từ đỉnh h đến đỉnh k) và hai đỉnh x, y (là điểm Start, End point) cho trước.

- Dữ liệu xuất là đường đi ngắn nhất từ x đến y.

```
def dijkstra(self):
    timeStart = time.time()
    if (self.checkStart and self.checkEnd and self.checkGraph and self.SinP and self.EinP):
        l = len(maze.points)
        D = [float('inf') for v in range(l+1)]
        D[0] = 0
        visited = [0 for i in range(l+1)]
        trace = [-1 for i in range(l+1)]
        trace[0] = 0
        pq = queue.PriorityQueue()
        pq.put((0, 0))
        while not pq.empty():
            (dist, current_vertex) = pq.get()
            visited[current_vertex] = 1
            if current_vertex == 1:
                break
            for i in self.graph:
                if (i == current_vertex):
                    for j in self.graph.get(i):
                        neighbor = j[0]
                        distance = j[1]

                        if visited[neighbor] == 0:
                            old_cost = D[neighbor]
                            new_cost = D[current_vertex] + distance
                            if new_cost < old_cost:
                                pq.put((new_cost, neighbor))
                                D[neighbor] = new_cost
                                trace[neighbor] = current_vertex
```

IV. Nguồn tham khảo

- Randomized DFS (tạo maze ngẫu nhiên): Maze generation algorithm - Wikipedia
- Thuật toán Dijkstra: Dijkstra's algorithm - Wikipedia và Graphs in Python: Dijkstra's Algorithm
- Shapely: The Shapely User Manual