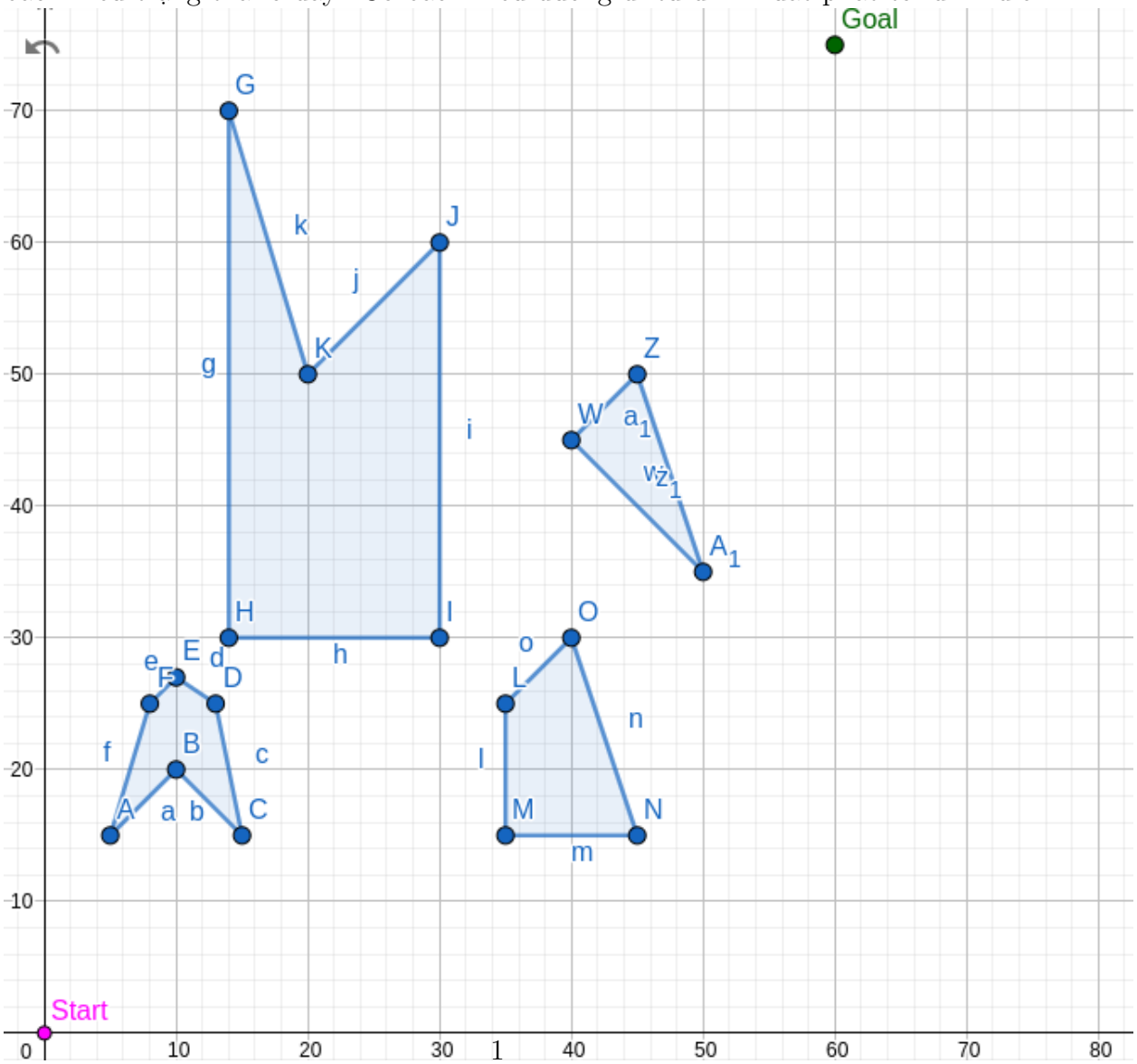
Report AI

Nguyễn Quốc Bảo

18110053

Lab04

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



Theo như dữ liệu ở file *input.txt* thì ta có 20 điểm với (x,y) là tọa độ của mỗi điểm và theo như hình thì ta có 4 đa giác, đa giác đầu tiên có 6 đỉnh, đa giác thứ 2 có 5 đỉnh, đa giác thứ 3 có 4 đỉnh, đa giác thứ 4 có 3 đỉnh và 2 đỉnh bắt đầu, kết thúc.

```
start: (0.0, 0.0, 'S')
goal (60.0, 75.0, 'G')
total polygon:  4
total 18 path
the shortest path from (0.0, 0.0, 'S') to (60.0, 75.0, 'G'):
(0.0, 0.0, 'S') --> (15.0, 15.0, 0) --> (30.0, 30.0, 1) --> (40.0, 45.0, 3) --> (60.0, 75.0, 'G')
Distance:  96.50967600315269
```

Theo như kết quả chạy ta biết được là có 18 đường đi từ đỉnh xuất phát.

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

   Vì nếu không bao gồm các đoạn thẳng nối thì sẽ rất khó để tìm đường đi ngắn nhất và hợp lý. Như hình ở ban đầu thì ta có thể thấy không gian trạng thái hiện tại có 20 điểm và 4 đa giác với số cạnh lần lượt là 6, 5, 4, 3. Và khi đi tìm đường ngắn nhất từ điểm bắt đầu đến kết thúc thì khi đi qua những điểm nó nhìn thấy được nó sẽ tạo ra một đoạn thẳng và biết được khoảng cách đó. Cứ thế thì cho đến khi tìm được khoảng cách ngắn nhất đến điểm kết thúc.

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

```
def all_edges(total_polygons, polygons):
    ''' get all edge of each polygon'''
    edges = []
    for i in range(total_polygons):
        pol = polygons[i]
        num_ver = len(polygons[i])
        for j in range(num_ver):
            edge = [pol[j % num_ver], pol[(j + 1) % num_ver]]
            edges.append(edge)
    return edges

def all_points(start,goal,polygons):
    ''' get all point of all polygon '''
    all_point = [start,goal]
    for i in range(len(polygons)):
        for point in polygons[i]:
            all_point.append(point)
    return all_point
```

Hàm *all-edges* là hàm dùng để lấy ra từng cặp đỉnh tạo ra các cạnh của đa giác. Hàm all-points là hàm dùng để lấy tất cả các đỉnh của đa giác và 2 đỉnh bắt đầu và kết thúc.

```
# calculation distance with other point
def distance(point_1, point_2):
    """ calculation distance with 2 point """
    dx = point_1[0] - point_2[0]
    dy = point_1[1] - point_2[1]
    return sqrt(dx**2 + dy**2)


# Check the relative position of the point
def Is_same_side(edge, point_1, point_2):
    ''' Check the relative position of the point'''
    x1 = edge[0][0]
    y1 = edge[0][1]
    x2 = edge[1][0]
    y2 = edge[1][1]
    d1 = (point_1[0] - x1)*(y2 - y1) - (point_1[1] - y1)*(x2 - x1)
    d2 = (point_2[0] - x1)*(y2 - y1) - (point_2[1] - y1)*(x2 - x1)
    return d1*d2 < 0
```

Hàm *distance* dùng để tính khoảng cách giữa 2 điểm bằng công thức Euclid. Hàm *Is-*

same-side là hàm dùng để kiểm tra xem vị trí tương đối của các điểm bằng cách xét những đường thẳng qua 2 điểm nếu những điểm thế vào có tích nhỏ hơn 0 sẽ không nhìn thấy được và ngược lại là nhìn thấy được.

```
# Check for the adjacent points of 1 point
def adjoining_points(total_polygons,polygons):
    ''' Return the adjacent points of each point in the polygon'''
    adjacent_points = defaultdict(list)
    edges = all_edges(total_polygons,polygons)
    for edge in edges:
        point1 = edge[0]
        point2 = edge[1]
        adjacent_points[point1].append(point2)
        adjacent_points[point2].append(point1)
    return adjacent_points
```

Hàm *adjoining-points* nhằm lấy ra các đỉnh kề của 1 đỉnh bất kì trong mỗi đa giác.

```
def successor(all_points, all_edges, adjacent_points):
    '''get all point can see and can't see of each point'''
    graph = defaultdict(list)
    not_see = defaultdict(list)
    for current_point in all_points:
        for edge in all_edges:
            for point in all_points:
                # check point and current point the same
                if (point == current_point) | (point in edge):
                    continue
                # check point and current point have the same polygon and edge
                if ((point[2] == current_point[2]) \
                    and (point not in adjacent_points[current_point])):
                    if point not in not_see[current_point]:
                        not_see[current_point].append(point)
                else:
                    edge1 = (current_point, edge[0])
                    edge2 = (current_point, edge[1])
                    # Check the relative position of the point
                    if (Is_same_side(edge1, edge[1], point)==False \
                        and Is_same_side(edge2, edge[0], point)==False\
                            and Is_same_side(edge, current_point, point)==True):
                        if point in graph[current_point]:
                            graph[current_point].remove(point)
                        if point not in not_see[current_point]:
                            not_see[current_point].append(point)
                    else:
                        if point not in not_see[current_point]:
                            if point not in graph[current_point]:
                                graph[current_point].append(point)
    return graph
```

Với Hàm này ta có thể lưu lại những đỉnh có thể nhìn thấy được bằng cách xét từng

điểm và từng cạnh.

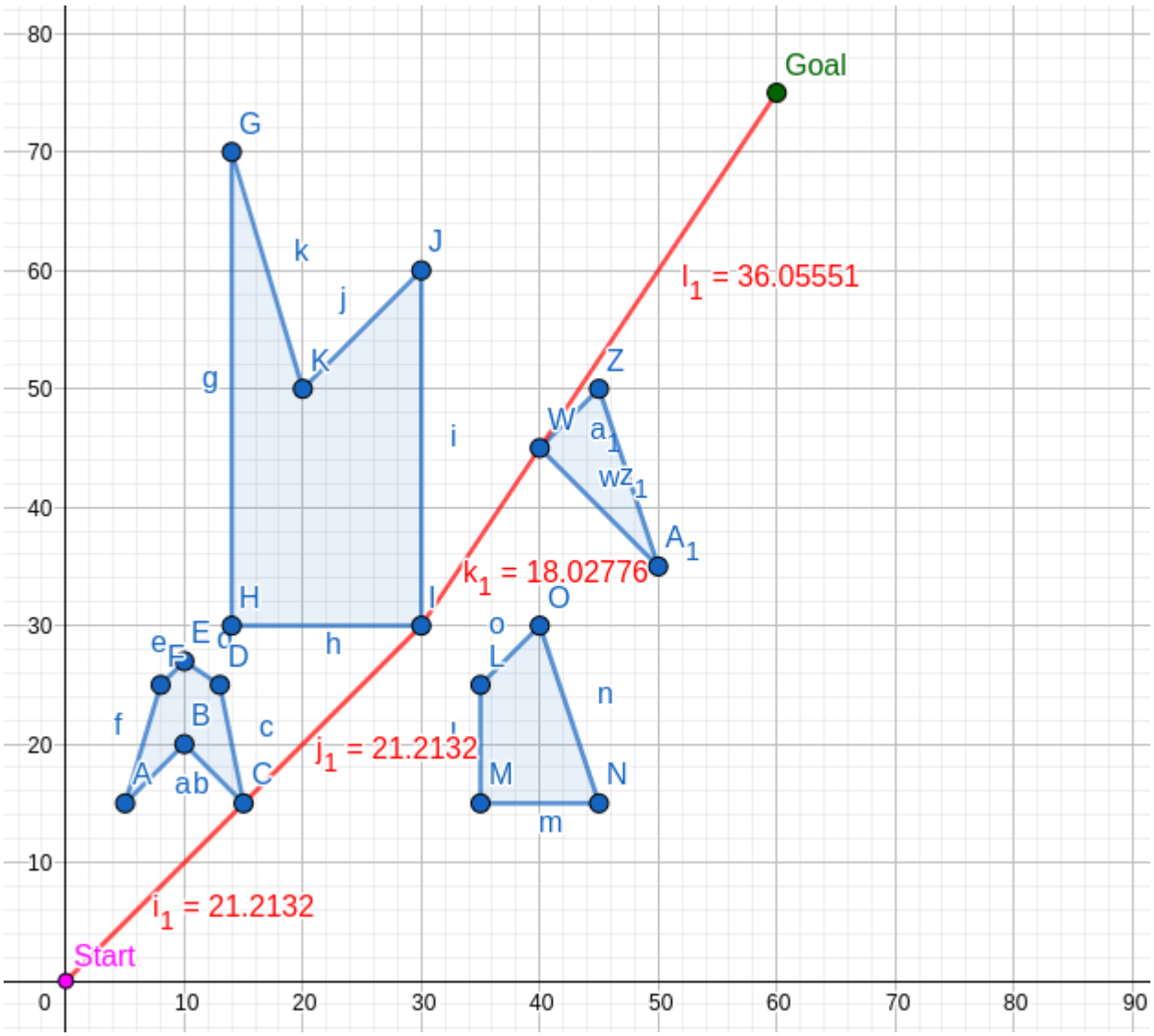
4. Áp dụng một thuật toán tìm kiếm để giải bài toán.

```
def shortest_path(graph, start, goal):
    ''' Return path shortest from start to goal'''
    explored = []
    queue = PriorityQueue()
    root = (0,[start])
    queue.put(root)
    if (start == goal):
        return "Start = goal"
    while True:
        if queue.empty():
            raise Exception("No way Exception")
        current_distance, path = queue.get()
        current_point = path[-1]
        if current_point == goal:
            print('total {} path'.format(queue.qsize()))
            return current_distance,path
        if current_point not in explored:
            if current_point not in graph:
                continue
            list_point = graph[current_point]
            # go through all point can see, construct a new path and
            # push it into the queue
            for point in list_point:
                d = distance(current_point,point)
                new_path = list(path)
                new_path.append(point)
                queue.put((current_distance+d,new_path))

            # mark point as explored
            explored.append(current_point)
    # in case there's no path between the 2 points
    return None,"There's no path between the 2 points"
```

Ta sử dụng thuật toán tìm kiếm giống với $A*$ duyệt các đỉnh và từ đỉnh đang xét mở ra các đỉnh có thể nhìn thấy được đã được chuẩn bị trước đó. Và lưu lại những điểm đã đi qua, đồng thời ta tính khoảng cách giữa những điểm đang xét và những điểm có thể đi qua và cộng với quãng đường đã đi được trước đó và chỉ lấy quãng đường ngắn nhất.

```python
def main():
    # Preprocessing
    file = open("input.txt","r")
    total_polygons, start, goal, polygons = read_file(file)
    all_edge = all_edges(total_polygons,polygons)
    all_point = all_points(start,goal,polygons)
    adjoining = adjoining_points(total_polygons,polygons)
    print('\n\n')
    print('start:',start)
    print('goal',goal)
    print('total polygon: ', total_polygons)
    # Set of point can see
    graph = successor(all_point,all_edge,adjoining)
    # Search shortest path and distance
    distance, path = shortest_path(graph,start,goal)
    print('the shortest path from {} to {}:'.format(start,goal))
    print(" --> ".join(str(i) for i in path))
    print('Distance: ', distance)
    print('\n\n')
```



Kết quả đã tính ra được.