上海交通大學
SHANGHAI JIAO TONG UNIVERSITY

Shortest path identification

Team Name ： 名字取的最好的组

Team Member:张黄灏、吴思澜、章甜、郑宜暄

LIST

Chapter 1

Chapter 2

Chapter 3

Chapter 4

Background:

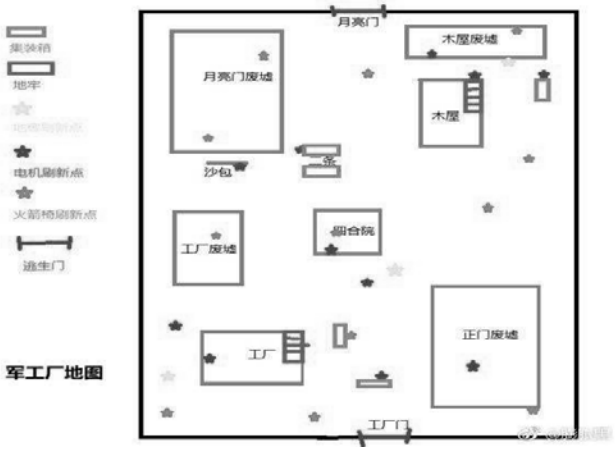
①Real life problems：

When we use map apps like baidu, gaode, etc, they will automatically recommend the shortest route for us. However, this kind of map apps can't cover some special area. For example, information on the terrain of an indoor environment is often vague. So we'd like to develop a program that can make up for this defect, which means you can input a map and ask the program to identify the shortest path for you.

②Game application：

What's more, except for the use we just mentioned, this program can also benefit a lot in the virtual world. Take the popular game 第五人格 as an example. The maps (one of them is shown below) are so complicated that it is hard for us to figure out the shortest escape route immediately. However, with this program, all you have to do is to prepare the specific map in advance and input your location when you want. Then you will get the best way leading to the gate as soon as possible.

For example:



Our Goal:

we'd like to develop a program that can make up for the defect we talked before, which means you can input a map like the specific locations of stores in a big mall, and ask the program to identify the shortest path for you.

What's more, except for the use we just mentioned, this program can also benefit a lot in the virtual world.

With this program, all you have to do is to prepare the specific map in advance and input your location when you want. Then you will get the best way leading to the gate as soon as possible.

Introduction

Main ideas:

Initialize all map cells with infinite distance

Initialize the start cell with 0 distance

step counter = 0

Add start cell to wavefront (active set)

While wavefront not empty:

select the first node with the minimum dist value from the wavefront

remove node from the wavefront

if node == goal:

break

for all 8 neighbors of the node:

if the neighbor is valid and not a wall:

increase step counter

neighDist = distance saved in node + distanceToNeighbor (1 or sqrt(2))

if neighDist < distance saved in neighbor:

update distance saved in neighbor with nieghDist

append neighbor to the wavefront (active set)

Create the path:

starting from the goalCell select the neighbor with the lowest distance saved

add the selection to the path and continue with the selection to save it's lowest neighbor till you reached the start point

Print the map with the path

Additional Info

①Boundary cases are not handeled with special code. So if the goal node is

outside of the map, for example, the the algorithm will still run as specified in the pseudocode till the wavefront is empty.

②Anyways, the only one special case is tested, and this is mainly about a special case when reading the map. Normally both start and goal point are within the map in unoccupied cells.

Input:

The input to the program is a text file. It contains comments, a start position, a goal position and a 2D map. Sentences begin with a dollar character($) are comment lines.

The first non-comment line of that text file encodes the start position of the robot and the second non-comment line is the goal position. Those are two integer numbers separated by spaces. They encode x, y, in that order.

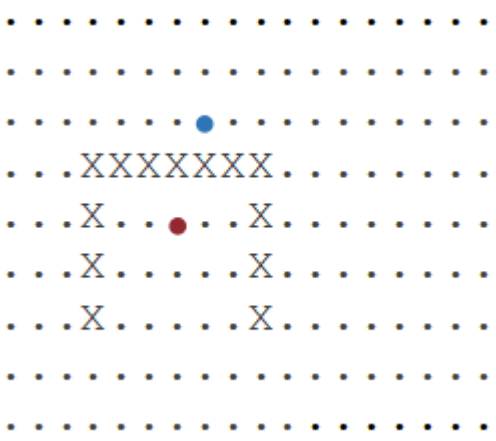
The second part of the input is the map. It consists of either '.' (a free cell) or 'X' (occupied cell = wall). The bottom left cell in the text file is located in the origin of the coordinate system (0, 0). The x-axis follows this line to the east (right) while the y-axis follows this column to the north (up).

The width of the map is defined by the number of characters in the line. The height of the map is defined by the number of lines in the map part.

```
COMMENT_MARKER = '$'
ROAD_MARKER = '.'
WALL_MARKER = 'X'
PATH_MARKER = 'P'
```

```
....................
....................
....................
...XXXXXXX..........
...X.....X..........
...X.....X..........
...X.....X..........
....................
....................
```

to



Process:

①define what we need

The distance of each point is infinitely large at first

```
INFINITE = float('inf')


map_row_inversed = []
```

This contains the point to be visited

```
wavefront = []
start_point = None
goal_point = None
```

Define the coordinate changes corresponding to the length and the direction of motion.

```
class Direction:
    def __init__(self, dx, dy):
        self.dx = dx # row
        self.dy = dy # column
BORDERING_DIRECTIONS = [
    Direction(dx=1, dy=0),  # Up
    Direction(dx=0, dy=1),  # Right
    Direction(dx=-1, dy=0),  # Down
    Direction(dx=0, dy=-1)  # Left
]

BORDERING_DIRECTION_TRAVEL_COST = 1
DIAGONAL_DIRECTIONS = [
    Direction(dx=1, dy=1),  # Upper right
    Direction(dx=-1, dy=1),  # Bottom right
    Direction(dx=-1, dy=-1),  # Bottom left
    Direction(dx=1, dy=-1)  # Upper left
]
DIAGONAL_DIRECTION_TRAVEL_COST = sqrt(2)


def get_travelling_cost(current_direction):
    for direction in BORDERING_DIRECTIONS:
        if direction.dx == current_direction.dx and direction.dy == current_direction.dy:
            return BORDERING_DIRECTION_TRAVEL_COST
    for direction in DIAGONAL_DIRECTIONS:
        if direction.dx == current_direction.dx and direction.dy == current_direction.dy:
            return DIAGONAL_DIRECTION_TRAVEL_COST
```

Returns the eight points around the middle point

```
def get_neighbour_point(point, direction):
    return Point(
        x=point.x + direction.dx,
        y=point.y + direction.dy
    )
```

Check if the point is conforming to our requirements.

```
def point_is_in_map(point, map):
    if point.x < 0 or point.x >= len(map) or point.y < 0 or point.y >= len(map[0]):
        return False
    return True


def point_is_free(point, map):
    return True if map[point.x][point.y].cell != WALL_MARKER else False
```

And Transform the coordinates of point into the corresponding coordinates on the map as well.

```
def get_row_inversed_point(point, map):
    return Point(
        len(map) - 1 - point.x,
        point.y
    )
```

Label points of the determined path on the map

```
def set_path_on_map(row_index, column_index, map):
    map[row_index][column_index].cell = PATH_MARKER


class Point:
    def __init__(self, x, y):
        self.x = x
```

```
        self.y = y

    def get_valid_neighbours(self, map):
        valid_neighbours = []
        for direction in BORDERING_DIRECTIONS:
            # Get the coordinates of bordering points
            neighbour_point = get_neighbour_point(self, direction)
            if point_is_in_map(neighbour_point, map):
                # If the point is not a wall, then the point is valid
                valid_neighbours.append(neighbour_point)
        for direction in DIAGONAL_DIRECTIONS:
            # Get the coordinates of diagonal points
            neighbour_point = get_neighbour_point(self, direction)
            if point_is_in_map(neighbour_point, map):
                # If the point is not a wall, then the point is valid
                valid_neighbours.append(neighbour_point)
        return valid_neighbours


class MapInfo:
    def __init__(self, cell, visited=False, distance_to_start=INFINITE):
        self.cell = cell
        self.visited = visited
        self.distance_to_start = distance_to_start
```

②Main procedure starts

```
current_row_index = 0
for line in sys.stdin.readlines():
    # Delete the empty line and judge whether it is valid information
    if not line.isspace() and line[0] != COMMENT_MARKER:
        input_content = line.split()
        # Note: The position of a point from input, is flipped when converted to row/column indexes on the map
        # For example, position (0, 1), is actually the first row and second column on the map, as opposed to second row and first column on the map
        if start_point is None:
            start_point = Point(
```

```
                x=int(input_content[1]),
                y=int(input_content[0])
            )  # Note: x, y flipped
        elif goal_point is None:
            goal_point= Point(
                x=int(input_content[1]),
                y=int(input_content[0])
            )  # Note: x, y flipped
        else:
            current_row = input_content[0]
            if len(map_row_inversed) == current_row_index:
                map_row_inversed.append([])
            for cell in current_row:
                map_row_inversed[current_row_index].append(
                    MapInfo(cell=cell)
                )
            current_row_index += 1
```

If it's an empty map, output the original empty map and quit the program

```
if map_row_inversed==[]:
    print("")
    os._exit(0)
```

Transform the coordinates of start point and goal point into the corresponding coordinates on the map

```
start_point = get_row_inversed_point(start_point, map_row_inversed)
goal_point = get_row_inversed_point(goal_point, map_row_inversed)
start_point_map_info = map_row_inversed[start_point.x][start_point.y]
start_point_map_info.visited = True
start_point_map_info.distance_to_start = 0
```

Confirm the start point on the map and take it as the first node

```
wavefront.append(start_point)
```

Dijkstra implementation

```
path_found = False
```

End the loop until no point can be used as a node or end point is found

```
while len(wavefront) > 0:
    # Take the first point as a node and delete it
    point_visiting = wavefront.pop(0)
    # Find valid points around the node
    current_neighbours = point_visiting.get_valid_neighbours(map_row_inversed)
    for current_neighbour in current_neighbours:
        current_neighbour_map_info =
map_row_inversed[current_neighbour.x][current_neighbour.y]
        # Check whether this point is "wall", or if it has been visited
        if not current_neighbour_map_info.visited and point_is_free(current_neighbour,
map_row_inversed):
            travelling_cost = get_travelling_cost(
                Direction(
                    dx=current_neighbour.x - point_visiting.x,
                    dy=current_neighbour.y - point_visiting.y
                )
            )
```

#Calculate the distance of one step

```
            new_total_distance=map_row_inversed[point_visiting.x][point_visiting.y].distance
_to_start + travelling_cost
            if new_total_distance < current_neighbour_map_info.distance_to_start:
                current_neighbour_map_info.distance_to_start = new_total_distance
                # Update distance saved in neighbor with the smaller distance
                wavefront.append(current_neighbour)
                wavefront.extend(current_neighbour.get_valid_neighbours(map_row_inversed))
        else:
            # Exclude the walls by marking them as visited
            map_row_inversed[current_neighbour.x][current_neighbour.y].visited = True
```

```
        #When the calculated node is goal point, the loop ends.
        if point_visiting.x == goal_point.x and point_visiting.y == goal_point.y:
            map_row_inversed[goal_point.x][goal_point.y].visited = True
            set_path_on_map(
                start_point.x,
                start_point.y,
                map_row_inversed
            )
            set_path_on_map(
                goal_point.x,
                goal_point.y,
                map_row_inversed
            )
            #Mark the start point and goal point on the map
            path_found = True
            break

if path_found:
    #Reset map to find path
    for row in map_row_inversed:
        for column in row:
            column.visited = False
    map_row_inversed[goal_point.x][goal_point.y].visited = True
    # Find the path from computed distances
    wavefront = [goal_point]
    while len(wavefront) > 0:
        point_visiting = wavefront.pop(0)
        # The loop ends until the start point is found
        if point_visiting.x == start_point.x and point_visiting.y == start_point.y:
            break
        # Find valid points around the node
        current_neighbours = point_visiting.get_valid_neighbours(map_row_inversed)
        current_neighbour_on_path = None
        for current_neighbour in current_neighbours:
            # Check whether this point is "wall"
```

```
            if point_is_free(current_neighbour, map_row_inversed):
                # Update distance saved in neighbor with the smaller distance
                if current_neighbour_on_path is None:
                    current_neighbour_on_path = current_neighbour
                elif
map_row_inversed[current_neighbour.x][current_neighbour.y].distance_to_start <
map_row_inversed[current_neighbour_on_path.x][current_neighbour_on_path.y].distance_to_start:
                    current_neighbour_on_path = current_neighbour
        map_row_inversed[current_neighbour_on_path.x][current_neighbour_on_path.y].cell =
PATH_MARKER
        wavefront.append(current_neighbour_on_path)
```

Output:

The program has to output the map with the found path marked with the letter P. The format follows exactly the input map format, except that the found path, including the start point and the goal point, are marked with the letter 'P' instead of a '.'

```
for row in map_row_inversed:
    row_to_print = ''
    for column in row:
        row_to_print += column.cell
    print(row_to_print)
```

For example:

```
....................
....................
.......PPP.........
...XXXXXXXP........
...X..P..XP........
...X...P.XP........
...X....PXP........
.........P.........
....................
```

Summary

THIS PROGRAM HELPS US:

①Offer point to point path planning ability for places that can’t be covered by the electronic map.

②Can be used in virtual world.

DIFICIENCY:

①It must start with a manual entry map.

②The position also need to be manually input.