Arnau Deu Marín 10/03/2023

PRUEBA EVALUACIÓN DAMAVIS

En este documento, explico la estructura del código entregado igual que algunos detalles de porqué lo he hecho de esta manera.

ESTRUCTURA

1. Mapa

Inicialmente, he definido el laberinto y el contenido de sus celdas. He creado un objeto laberinto y un objeto casilla.

El objeto laberinto sirve para poder almacenar todas las variables en relación a este, y a parte para indexar y consultar cada una de sus casillas (no se consulta ninguna casilla desde fuera del objeto laberinto).

El objeto casilla sirve para poder almacenar el tipo de casilla que es (si "." o "#") y para referenciar directamente a los objetos casilla que tiene alrededor suyo.

2. Camino

Para encontrar el camino óptimo se genera una función que itera sobre si misma junto con una función *starter*.

La función *starter* se encargará de iniciar la iteración con las posibles opciones.

La función iteradora se encargará de buscar todos los caminos posibles.

3. Evaluación Camino

Durante el proceso de iteración se analiza:

- Que el camino que se esté encontrando no haya sido encontrado, o que, si ya se ha encontrado, haya sido con una cantidad de pasos menor.
- Que el camino no haya realizado un movimiento ilegal.
- Que el camino haya llegado a su destino.

EXPLICACIÓN CÓDIGO

El código empieza con la definición de las dos clases:

```
class cell:
    def __init__(self, condition):
        self.block = condition
        self.top = False
        self.bottom = False
        self.left = False
        self.right = False

class maze():
    def __init__(self,labyrinth_map):
        self.maze = []
        self.rows = len(labyrinth_map)
        self.columns = len(labyrinth_map[0])
        self.dictionari={}
        self.minimum = float('inf')
        for line in labyrinth_map:
            temp = []
            for value in line:
                temp.append(cell(value))

            self.maze.append(temp.copy())

        for x in range(self.rows):
            for y in range(self.columns):
                node = self.maze[x][y]
                if x != 0:
                    node.top = self.maze[x-1][y]
                if x != (self.rows - 1):
                    node.bottom = self.maze[x+1][y]
                if y != 0:
                    node.left = self.maze[x][y-1]
                if y != (self.columns - 1):
                    node.right = self.maze[x][y+1]
        if self.maze[0][0].block == "#":
            print('Entrance blocked')
        if self.maze[-1][-1].block == "#":
            print('Exit blocked')
```

Se crea el laberinto y todas las celdas que hay en él, con su respectivo valor. Seguidamente se informa, dentro de las variables de cada celda, la celda que tienen arriba, abajo, a la derecha y a la izquierda. A parte, se mira que el laberinto tenga una entrada y salida abierta.

Dentro de la clase *maze*, se definen funciones para validar que:

- Las posiciones de cierto movimiento son legales dentro del laberinto
- El movimiento no ha sido usado ya y en caso de que haya sido usado por otra opción de recorrido, haya sido realizando menos pasos.
- Se ha encontrado la salida y se ha terminado el recorrido.

```python
def check_position(self,x,y,orientation, rotation=False):
    if orientation == "h" or rotation:
        if y <= 0 or y >= (self.columns - 1) or x >= self.rows or x <0:
            return False
        if self.maze[x][y].block == "#" or self.maze[x][y-1].block == "#" or self.maze[x][y+1].block == "#":
            return False
        if not rotation:
            return True

    if orientation == "v" or rotation:
        # print(x)
        # print(y)
        # print(self.columns)
        if x <= 0 or x >= (self.rows - 1) or y >= self.columns or y < 0:
            return False
        if self.maze[x][y].block == "#" or self.maze[x-1][y].block == "#" or self.maze[x+1][y].block == "#":
            return False
        if not rotation:
            return True

    if self.maze[x-1][y-1].block == "#" or self.maze[x-1][y+1].block == "#" or self.maze[x-1][y+1].block == "#" or self.maze[x+1][y+1].block == "#":
        return False
    return True

def check_end(self, x, y):
    if (x == (self.rows-1) and y == (self.columns-2)) or (x == (self.rows-2) and y == (self.columns-1)):
        return True
    return False
def step_dictionari(self, step, steps):

    if steps -1 > self.minimum:
        return False
    if step in self.dictionari.keys():
        if self.dictionari[step] > steps:
            self.dictionari[step], steps
            return True
        return False
    else:
        self.dictionari[step] = steps
        return True
```

Seguidamente pasamos a la función de iteración principal, donde teniendo en cuenta el laberinto, el movimiento que se ha realizado, la orientación del "palo", hacia donde ha sido el último movimiento y si este ha sido un movimiento de rotación, es capaz de analizar todos los posibles caminos.

Los pasos que sigue esta función son:

- Primero mira que la posición realizada sea "legal"
- Mira que el movimiento realizado no haya sido realizado ya y con una cantidad menor de pasos.
- Que no sea un movimiento final
- Llamada recursiva con los posibles movimientos desde esa coordenada.

```python
def iterator(maze,steps,x,y,orientation,last_move,has_rotated=False):
    # print(x,"-",y)
    if maze.check_position(x, y, orientation,has_rotated):

        step_made = str(x)+","+str(y)+"," + orientation
        # print(step_made)
        if maze.step_dictionari(step_made, len(steps)):
            steps.append(step_made)
            if maze.check_end(x,y):
                # for step in steps:
                #     print(step)

                # print("Total Steps: ", len(steps) -1)
                if maze.minimum > len(steps)-1:
                    maze.minimum = len(steps)-1
                return len(steps)-1
            else:


                if has_rotated:
                    if last_move == "right":
                        iterator(maze, steps.copy(), x, y-1, orientation, "right")
                        iterator(maze, steps.copy(), x-1, y, orientation, "down")
                        iterator(maze, steps.copy(), x+1, y, orientation, "up")
                    elif last_move == "left":
                        iterator(maze, steps.copy(), x, y+1, orientation, "left")
                        iterator(maze, steps.copy(), x-1, y, orientation, "down")
                        iterator(maze, steps.copy(), x+1, y, orientation, "up")
                    elif last_move == "down":
                        iterator(maze, steps.copy(), x-1, y, orientation, "down")
                        iterator(maze, steps.copy(), x, y+1, orientation, "left")
                        iterator(maze, steps.copy(), x, y-1, orientation, "right")
                    elif last_move == "up":
                        iterator(maze, steps.copy(), x+1, y, orientation, "up")
                        iterator(maze, steps.copy(), x, y+1, orientation, "left")
                        iterator(maze, steps.copy(), x, y-1, orientation, "right")
                    else:
                        iterator(maze, steps.copy(), x-1, y, orientation, "down")
                        iterator(maze, steps.copy(), x+1, y, orientation, "up")
                        iterator(maze, steps.copy(), x, y+1, orientation, "left")
                        iterator(maze, steps.copy(), x, y-1, orientation, "right")
                else:
                    if orientation == "h":
                        orientation2 = "v"
                    else:
                        orientation2 = "h"
                    if last_move == "right":
                        iterator(maze, steps.copy(), x, y-1, orientation, "right")
                        iterator(maze, steps.copy(), x-1, y, orientation, "down")
                        iterator(maze, steps.copy(), x+1, y, orientation, "up")
                        iterator(maze, steps.copy(), x, y, orientation2, False,has_rotated = True)
                    elif last_move == "left":
                        iterator(maze, steps.copy(), x, y+1, orientation, "left")
                        iterator(maze, steps.copy(), x-1, y, orientation, "down")
                        iterator(maze, steps.copy(), x+1, y, orientation, "up")
                        iterator(maze, steps.copy(), x, y, orientation2, False,has_rotated = True)
                    elif last_move == "down":
                        iterator(maze, steps.copy(), x-1, y, orientation, "down")
                        iterator(maze, steps.copy(), x, y+1, orientation, "left")
                        iterator(maze, steps.copy(), x, y-1, orientation, "right")
                        iterator(maze, steps.copy(), x, y, orientation2, False,has_rotated = True)
                    elif last_move == "up":
                        iterator(maze, steps.copy(), x+1, y, orientation, "up")
                        iterator(maze, steps.copy(), x, y+1, orientation, "left")
                        iterator(maze, steps.copy(), x, y-1, orientation, "right")
                        iterator(maze, steps.copy(), x, y, orientation2, False,has_rotated = True)

                    else:
                        iterator(maze, steps.copy(), x-1, y, orientation, "down")
                        iterator(maze, steps.copy(), x+1, y, orientation, "up")
                        iterator(maze, steps.copy(), x, y+1, orientation, "left")
                        iterator(maze, steps.copy(), x, y-1, orientation, "right")
        else:
            return float('inf')
    else:
        return float('inf')
```

Finalmente se muestra la función “starter”que, contemplando las dos posibles maneras de empezar un laberinto, lanza la función iterativa.

```
def solution(labyrinth):
    # map creation
    lab_map = maze(labyrinth)

    iterator(lab_map, [],0,1,'h',False)
    iterator(lab_map, [],1,0,'v',False)
    print("\n - Labyrinth")
    for row in labyrinth:
        print(row)
    if lab_map.minimum == float('inf'):
        print("\nMinimum steps required to complete the labyrinth: -1")
    else:
        print("\nMinimum steps required to complete the labyrinth: ", lab_map.minimum)

    return
```

OBSERVACIONES

Durante la programación del código se han contemplado diferentes estructuras. La más óptima ha sido almacenar el número de pasos realizados en una variable del laberinto ya que, si devolvía de cada iteración posible el valor de pasos para luego calcular el mínimo, en el último ejemplo de TEST 4 tardaba muchísimo.

Se han dejado unos *prints* comentados en la línea 89 por si se quiere ver que caminos ha encontrado el algoritmo.

Todo el código lo he programado yo, si hay alguna cosa que no entendéis cómo funciona o me queréis preguntar directamente porqué está hecho de esta manera, os responderé encantado.

Iba a crear un sistema de versiones en GitHub, pero al hacerlo directamente y sin pausa solo he guardado la última versión. Igualmente, esto está subido a mi GitHub personal de forma pública.