MAZE GAME REPORT

Ziad Essam Ibrahim

Code Explanation:

I have used OOP to structure the code and to make it more maintainable and easier to fix the code or add anything in the code in the future. I made 3 classes `MazeGenerator, `MazeToGraph`, `MazeSolver`, each one of them has methods that is essential to the game.

MazeGenerator Class:

```python
import random
class MazeGenerator:
    def __init__(self, row, col):
        self.row = row
        self.col = col
        self.maze = [['E' for _ in range(row)] for _ in range(col)]

    def generateMaze(self):

        self.maze[0][0] = "R"
        self.maze[self.row - 1][self.col - 1] = "T"
        obstaclesNumber = 0
        while obstaclesNumber < 18:
            x = random.randint(0, self.row - 1)
            y = random.randint(0, self.col - 1)
            if self.maze[x][y] == "E":
                self.maze[x][y] = "X"
                obstaclesNumber += 1
        return self.maze, (0, 0), (self.row - 1, self.col - 1)

    def printMaze(self):
        for row in self.maze:
            print(' '.join(row))
```

This class is used to generate a maze with fixed places for the robot and the treasure and places 18 obstacles in random places with the use of `generateMaze` method.

Also it print the maze as a 2D array with “E” representing empty spaces, “X” representing obstacles, “R” representing the robot and “T" representing the treasure.

MazeToGraph Class

```python
class MazeToGraph:
    def __init__(self):
        self.graph = {}
    def convertMazeToGraph(self, maze):
        for i in range(len(maze)):
            for j in range(len(maze[i])):
                neighbors = []
                if i > 0 and maze[i - 1][j] != "X":
                    neighbors.append((i - 1, j))
                if i < len(maze) - 1 and maze[i + 1][j] != "X":
                    neighbors.append((i + 1, j))
                if j > 0 and maze[i][j - 1] != "X":
                    neighbors.append((i, j - 1))
                if j < len(maze[i]) - 1 and maze[i][j + 1] != "X":
                    neighbors.append((i, j + 1))
                self.graph[(i, j)] = neighbors
        return self.graph
```

The *MazeToGraph* class is used to convert the maze to a graph, it has a method called *convertMazeToGraph* that takes the maze as a parameter and return a dictionary that acts as our graph. Converting the maze to graph is not an essential step as we can deal with the maze directly , but dealing with graph makes the implementation of the A* algorithm way easier.

MazeSolver Class

```python
import heapq
class MazeSolver:

    def manhattanHeuristic(self,a, b):
        return abs(a[0] - b[0]) + abs(a[1] - b[1])

    def aStarSolution(self,graph,start,goal):
        # Priority queue to store the node with the lowest cost + heuristic value
        frontier = []
        heapq.heappush(frontier, (0, start))

        # Dictionary to store the each node in the path and the previous node in the pa
        cameFrom = {}
        cameFrom[start] = None
        # Dictionary to store the cost to reach a node
        costOfThePath = {}
        costOfThePath[start] = 0

        while frontier:
            # Pop the node with the f(n) value from the frontier
            currentCost, current = heapq.heappop(frontier)

            # If we've reached the goal, reconstruct the path and return it
            if current == goal:
                path = []
                while current:
                    path.append(current)
                    current = cameFrom[current]
                path.reverse()
                return path

            for neighbor in graph[current]:
                # consider that the cost of moving from one node to another is 1
                gOfTheNode = costOfThePath[current] + 1
                if neighbor not in costOfThePath or gOfTheNode < costOfThePath[neighbor
                    costOfThePath[neighbor] = gOfTheNode
                    fOfTheNode = gOfTheNode + self.manhattanHeuristic(neighbor, goal)
                    heapq.heappush(frontier, (fOfTheNode, neighbor))
                    cameFrom[neighbor] = current
        return None
```

This class is used to solve the maze by using the method aStarSolution that takes three parameters, the graph, start and the goal and returns the path if there is one and None if the maze is not solvable. Also it has another method called manhattanHeuristic that use the Manhattan distance as the

heuristic function as the Manhattan distance is the best heuristic for our case.

A* algorithm implementation

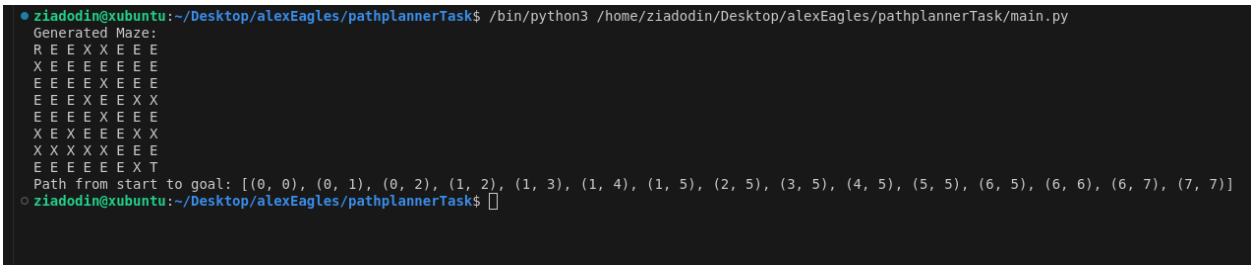
- First the variable frontier will be the priority queue that from we will pop the node with the lowest f(n) with the help of heapq module that will make it easier for us to use the priority queue efficiently as its pop and push methods has time complexity of log(n).

- Then there is two dictionary comeFrom and costOfThePath, comeFrom help us to know where each node come from, by come from I mean the node that led us to another node, so it can be helpful to trach the path to the goal, costOfThePath help us know how it cost to reach each node in the graph.

- Then we start looping over the frontier items until its empty or we found the goal, first we pop from the frontier the node with the smallest f(n),if it is the goal then we start to construct the path by the help of comeFrom dictionary that has all of the paths to every node in the graph, if it is not the goal we proceed by exploring the neighbors if the node and to get the best paths for them and store it in the comeFrom dictionary.

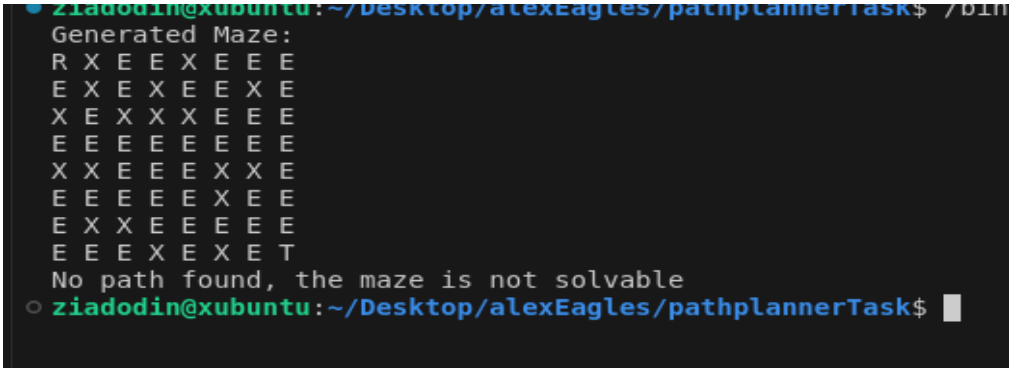
Example Usage:

1. If the maze is solvable



If the maze can be solved, then a path is printed from the start to the goal by printing the cells where the robot needs to move. Please note that “R” is the robot, “T” is the treasure, “X” is the obstacles, and “E” is the empty cells that the robot can move freely to.

2. If the maze is not solvable



If the maze cannot be solved, for example there are many obstacles around the robot so he cant reach the treasure, then no path is printed and no path found is declared.