Report: Maze Game (A*)

I didn't have time to adapt it to the two maze exercises done previously as I didn't know what I had to change on the deadline day. Then I used the pyamaze library importing maze, agent, and a label for the text. The main idea of this module, pyamaze, is to assist in creating customizable random mazes and be able to work on that, like applying the search algorithm with much ease. By using this module, you don't need to program the GUI and also you don't need the Object-Oriented Programming since the module will provide you the support. This module uses the Tkinter GUI framework which is built-in in Python and you don't need to install any framework to use this module. In addition, I used the library queue.

```
from pyamaze import maze, agent, textLabel
from queue import PriorityQueue
```

This function simply defines the cell of the maze.

```
def h(cell1, cell2):
    x1, y1 = cell1
    x2, y2 = cell2
    return abs(x1 - x2) + abs(y1 - y2)
```

Starting from the initial cell, the cost of the initial cell is 0 because the cost to reach the initial cell from the initial cell is obviously zero. Algorithm A* will choose the cell with the lowest cost for the next move. The algorithm defined below will apply the same operations for all other cells until it reaches the end, i.e. the green square. Since we must choose the cell with the least cost, we will use the priority queue of the data structure to implement the A* algorithm.

Unlike the queue which works on the FIFO (First In First Out) principle, the items in a priority queue are checked according to priority. The priority can be the value of the item (highest or lowest). In Python, we have a priority queue available in the queue module and the priority is the lowest value, and therefore is more suitable for implementing A*.

The Maze arguments that will be presented are: rows, where m.rows will return the number of rows in the maze; cols, where m.cols will return the number of columns in the maze; grid, where m.grid will return a list of all cells in the maze; maze_map, where m.maze_map will return the information of the open and closed walls of the maze as a dictionary. The keys will be the cell and the value will be another dictionary with the information of four walls in the East, West, North and South directions ("ESNW" in the for).

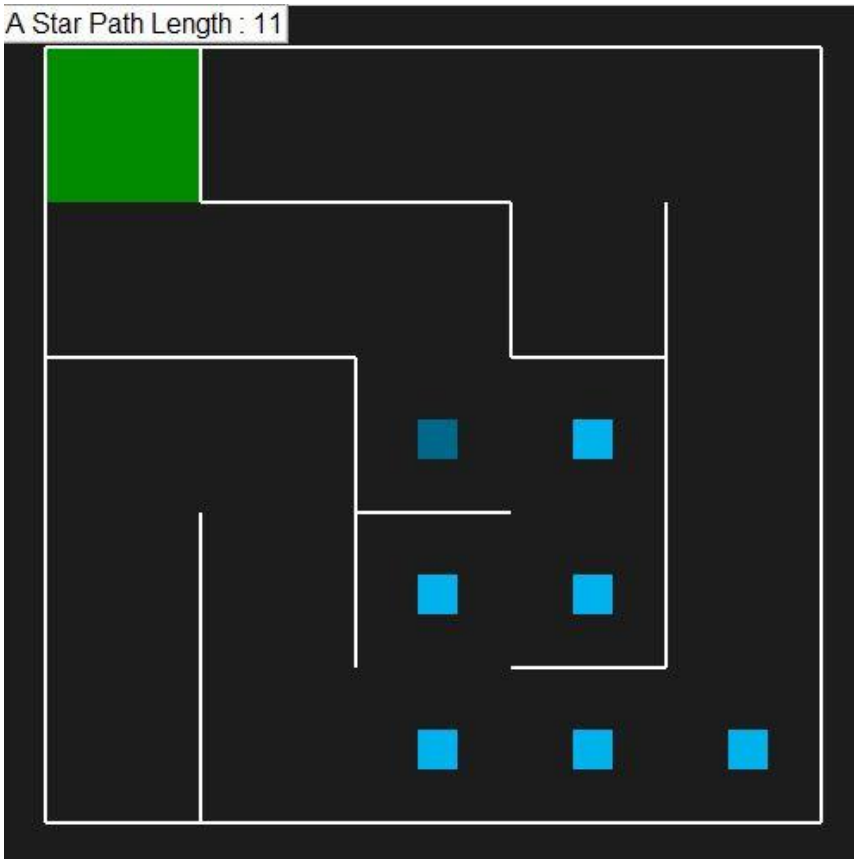
In addition, the length of the path from the start point to the end point is calculated.

```
def aStar(m):
    start = (m.rows, m.cols)
    g_score = {cell: float('inf') for cell in m.grid}
    g_score[start] = 0
    f_score = {cell: float('inf') for cell in m.grid}
    f_score[start] = h(start, (1, 1))

    open = PriorityQueue()
    open.put((h(start, (1, 1)), h(start, (1, 1)), start))
    aPath = {}
    while not open.empty():
        currCell = open.get()[2]
        if currCell == (1, 1):
            break
        for d in 'ESNW':
            if m.maze_map[currCell][d] == True:
                if d == 'E':
                    childCell = (currCell[0], currCell[1] + 1)
                if d == 'W':
                    childCell = (currCell[0], currCell[1] - 1)
                if d == 'N':
                    childCell = (currCell[0] - 1, currCell[1])
                if d == 'S':
                    childCell = (currCell[0] + 1, currCell[1])
            temp_g_score = g_score[currCell] + 1
            temp_f_score = temp_g_score + h(childCell, (1, 1))

            if temp_f_score < f_score[childCell]:
                g_score[childCell] = temp_g_score
                f_score[childCell] = temp_f_score
                open.put((temp_f_score, h(childCell, (1, 1)), childCell))
                aPath[childCell] = currCell
fwdPath = {}
cell = (1, 1)
while cell != start:
    fwdPath[aPath[cell]] = cell
    cell = aPath[cell]
return fwdPath
```

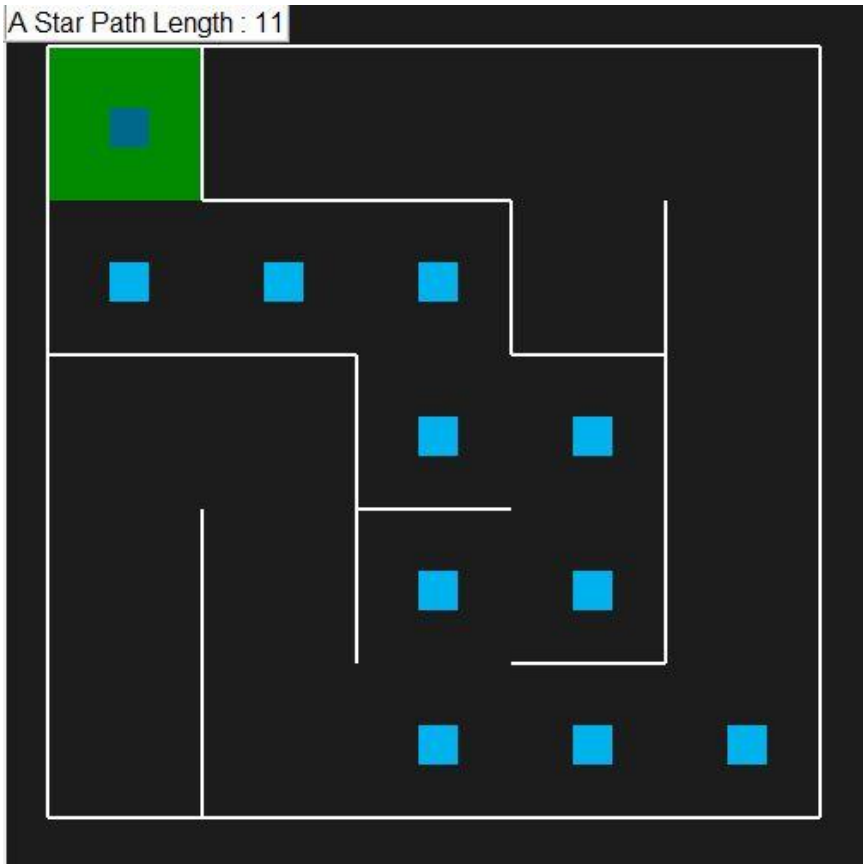
In the main, to simply generate a maze, you need to create the maze object and then apply the CreateMaze function. The last statement will be applying the function run to run the simulation. The top left cell is the goal of the maze and there are ways you can change the goal in any cell. Also, by default, a perfect maze is generated, which means that all cells in the maze are accessible and there is only one path from any cell to the target cell. Therefore, any cell can be treated as a starting cell and therefore not highlighted. Internally the last cell, i.e., the last row and the last cell of the column, is set as the starting cell. So, the general task will be to find the way from the lower right cell to the upper left cell. You can change the size of the maze as you create it. After creating a maze and an agent (one or more) inside the maze, you can make the agent move along a specific path. The best will be to move the agent on the path attribute of the maze. For this, we have a method in the maze class called tracePath which takes a dictionary as an input argument. The dictionary key is the agent, and the value is the path you want that agent to follow. The tracePath method will simulate the agent walking the path.

```
if __name__ == '__main__':
    m = maze(5, 5)
    m.CreateMaze()
    path = aStar(m)

    a = agent(m, footprints=True)
    m.tracePath({a: path})
    l = textLabel(m, 'A Star Path Length', len(path) + 1)

    m.run()
```

Once the maze is created, the starting point is in the lower right corner and the ending point is marked with a green square in the upper left.



Finally, the final path from the start blue dot to the green dot is marked in blue dot street.



Statistics A*:

The time that the algorithm takes to find the solution since the program is run, the labyrinth is built, and the final solution has been found is 00:06.80.

The time that the algorithm takes to find the solution with the maze already built and therefore the final solution is 00:03.50.