Maze

Contents

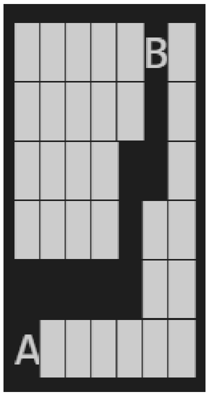
1 Introduction

You will build an AI agent to solve maze problem by using frontier search algorithm. We will start with couple of Python programming exercises, and then we will work on maze game.

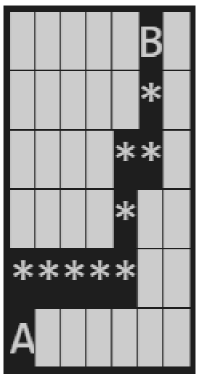
2 Submission Requirement

Once your finish your code, you need to submit source code and screenshots of result similar to this:

```
PS C:\PWA-L4\1.Search> python maze.py maze1.txt
Maze:
```




```
Solving...
['up', 'up', 'right', 'up', 'up', 'right', 'right', 'right', 'right', 'up']
['up', 'right', 'right', 'right', 'right', 'up', 'up', 'right', 'up', 'up']
States Explored: 11
Solution:
```



3 Maze

3.1 Array

Maze can be modeled as array.

1. Write the following code.

```
contents = '''
#####B#
##### #
####  #
#### ##
     ##
A######'''
contents = contents.splitlines()
print(contents)

print(f"contents[1]:{contents[1]}")
print(f"contents[1][5]:{contents[1][5]}")
```

Expected Output is

```
['', '#####B#', '##### #', '####  #', '#### ##', '     ##', 'A######']
contents[1]:#####B#
contents[1][5]:B
```

3.2 Maze Height and Width

We need to convert maze to data model, such as height, width, start point. These parameters will be used in search algorithm.

3.2.1 What to do

```
contents = '''
#####B#
##### #
####  #
#### ##
     ##
A######
'''
contents = contents.splitlines()
print(contents)
height = len(contents)

#print('length of each row')
width = max(len(line) for line in contents)
```

3.2.2 Expected Result

It should be some thing similar to the following:

```
['', '#####B#', '##### #', '####  #', '#### ##', '     ##', 'A######']
Height 7 Width 7
```

3.3 Maze Walls

We will create data model for start point, target point and walls.

3.3.1 What to do

1. Define the maze:

```
# Track walls
contents = '''
#####B#
##### #
####  #
#### ##
     ##
A######
'''
contents = contents.splitlines()
```

2. Create a function to parse the maze to walls, start point and target point. The function takes maze as input, and return start point, target point and walls data structure.

```
def get_walls(maze):
    print(contents)
    height = len(contents)
    width = max(len(line) for line in contents)
    start = ()
    goal = ()
    walls = []

    for i in range(height):
        row = []
        for j in range(width):
            try:
                if contents[i][j] == "A":
                    start = (i, j)
                    row.append(False)
                elif contents[i][j] == "B":
                    goal = (i, j)
                    row.append(False)
                elif contents[i][j] == " ":
                    row.append(False)
                else:
                    row.append(True)
            except IndexError:
                row.append(False)
        walls.append(row)

    print(f"start {start}")
    print(f"goal {goal}")
    return height, width, walls
```

3. Test the function.

```
if __name__ == '__main__':
     # execute only if run as a script
    height, width, walls = get_walls(contents)
    print(walls)
```

3.3.2 Expected Result

You should get the result similar to this:

```
['', '#####B#', '##### #', '####  #', '#### ##', '     ##', 'A######']
start (6, 0)
goal (1, 5)
[[False, False, False, False, False, False, False], [True, True, True, True, True,
False, True], [True, True, True, True, True, False, True], [True, True, True, True,
False, False, True], [True, True, True, True, False, True, True], [False, False,
False, False, False, True, True], [False, True, True, True, True, True, True]]
```

3.4 Neighbors

Each cell may have zero, one or more neighbor cells that it can move to. You will create a function to return all neighbors of a given cell.

3.4.1 What to do

1. Import get_walls function that you created in the previous step. You may need to replace a7_alls with your file name.

```
# Neighbors of a state
from a7_walls import get_walls
```

2. Define a function neighbours. It will take a state as input, and return valid neighbors.

```
def neighbors(state):
    row, col = state
    candidates = [
        ("up", (row - 1, col)),
        ("down", (row + 1, col)),
        ("left", (row, col - 1)),
        ("right", (row, col + 1))
    ]

    result = []
    for action, (r, c) in candidates:
        if 0 <= r < height and 0 <= c < width and not walls[r][c]:
            result.append((action, (r, c)))

    return result
```

3. Test the neighbors function. It uses a sample state (3,5).

```
if __name__ == '__main__':
    contents = '''
#####B#
##### #
####  #
#### ##
     ##
A######
'''

    height, width, walls = get_walls(contents)
    state = (3,5)
    result = neighbors(state)
    print(result)
```

3.4.2 Expected Result

You should get the result similar to this:

```
[('up', (2, 5)), ('left', (3, 4))]
```

4 Solve Maze

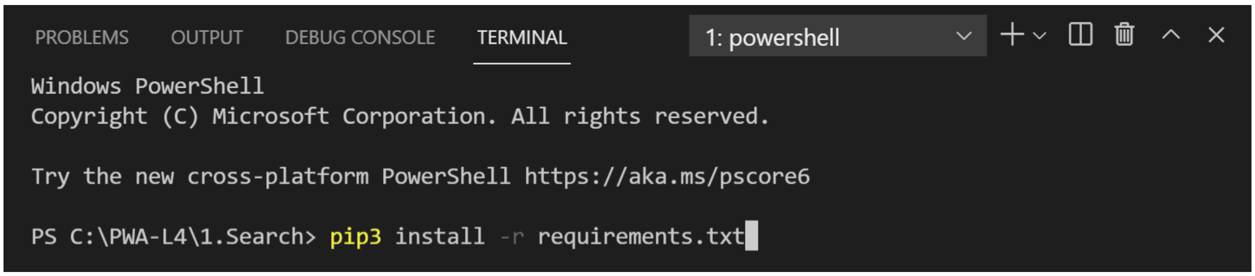
You will need to download maze.py. It has `Node` class, `StackFrontier` class, `QueueFroniter` class and `Maze` class. You have already created several functions in the previous steps. Now you will need to work the key function `solve` in `Maze` class.

4.1.1 Download Files

You will need to download maze.zip file and unzip it.

4.1.2 Install Pillow

1. Open Visual Studio Code, open the folder that contains unzipped maze.py file.
2. Open a terminal, type in
3. pip3 install -r requirements.txt



4.1.3 What to do

Open maze.py file, and finish the `solve` function in `Maze` class.

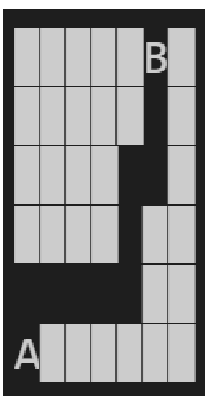
```
def solve(self):
        """Finds a solution to maze, if one exists."""

        # Keep track of number of states explored
        self.num_explored = 0

        # Initialize frontier to just the starting position
        start = Node(state=self.start, parent=None, action=None)
        frontier = StackFrontier()
        frontier.add(start)

        # Initialize an empty explored set
        self.explored = set()

        # Keep looping until solution found
        while True:

            # If nothing left in frontier, then no path
            if frontier.empty():
                raise Exception("no solution")

            # Choose a node from the frontier
            node = frontier.remove()
            self.num_explored += 1

            # If node is the goal, then we have a solution
            if node.state == self.goal:
                actions = []
                cells = []
                while node.parent is not None:
                    actions.append(node.action)
```

```
                    cells.append(node.state)
                    node = node.parent
                print(actions)
                actions.reverse()
                print(actions)
                cells.reverse()
                self.solution = (actions, cells)
                return

            # Mark node as explored
            self.explored.add(node.state)

            # Add neighbors to frontier
            for action, state in self.neighbors(node.state):
                if not frontier.contains_state(state) and state not in self.explored:
                    child = Node(state=state, parent=node, action=action)
                    frontier.add(child)
```

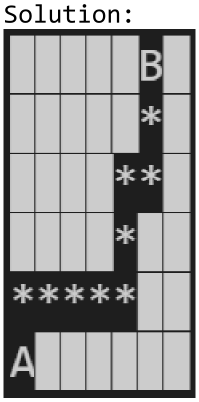
4.2 Test

Open a terminal in Visual Studio Code, type:

```
PS C:\PWA-L4\1.Search> python maze.py maze1.txt
Maze:
```




```
Solving...
['up', 'up', 'right', 'up', 'up', 'right', 'right', 'right', 'right', 'up']
['up', 'right', 'right', 'right', 'right', 'up', 'up', 'right', 'up', 'up']
States Explored: 11
Solution:
```



Once it's working, you can also test other mazes by changing the maze file name, such as:

```
PS C:\PWA-L4\1.Search> python maze.py maze2.txt
```

```
Or you can ask your family member to make a maze, and you use your little AI agent to
solve it!
```

Big Congratulation!! Now it's your time to lay back and ask your family member to play it!