24/09/20

Practical - 1

Aim: Implement and analyze Breadth first search on a specific problem.

Theory:

Graph Traversal Algorithm-
Graph traversal refers to the process of visiting each vertex in a graph. Such traversals are classified by the order in which they are visited. They can also be used to find out whether a node is reachable from given node or not.

Breadth First Search:
The Breadth First Search (BFS) is an algorithm for traversing or searching tree or graph data structures. It explores all the nodes at the present depth before moving on to the nodes at the next depth level. It is implemented by using queue data structure.

Algorithm:

Step 1: Pick a node and enqueue all its adjacent nodes into a queue.

Step 2: Dequeue a node from the queue, mark it as visited and enqueue all its adjacent nodes.

Step 3: Repeat until queue is empty or goal is reached.

Problem:
Maze Path Finding:
A maze is a path or collection of paths, typically from start to goal.

The problem is to find a path (possibly the best/shortest one) which will take the agent from start to goal position.

Code:

maze.py

```python
import sys

class Node():
    def __init__(self, state, parent, action):
        self.state = state
        self.parent = parent
        self.action = action

class StackFrontier():
    def __init__(self):
        self.frontier = []

    def add(self, node):
        self.frontier.append(node)

    def contains_state(self, state):
        return any(node.state == state for node in self.frontier)

    def empty(self):
        return len(self.frontier) == 0

    def remove(self):
        if self.empty():
            raise Exception("empty frontier")
        else:
            node = self.frontier[-1]
            self.frontier = self.frontier[:-1]
            return node

class QueueFrontier(StackFrontier):

    def remove(self):
        if self.empty():
            raise Exception("empty frontier")
        else:
            node = self.frontier[0]
            self.frontier = self.frontier[1:]
            return node

class Maze():

    def __init__(self, filename):

        # Read file and set height and width of maze
        with open(filename) as f:
            contents = f.read()

        # Validate start and goal
        if contents.count("A") != 1:
            raise Exception("maze must have exactly one start point")
        if contents.count("B") != 1:
            raise Exception("maze must have exactly one goal")

        # Determine height and width of maze
        contents = contents.splitlines()
        self.height = len(contents)
        self.width = max(len(line) for line in contents)

```

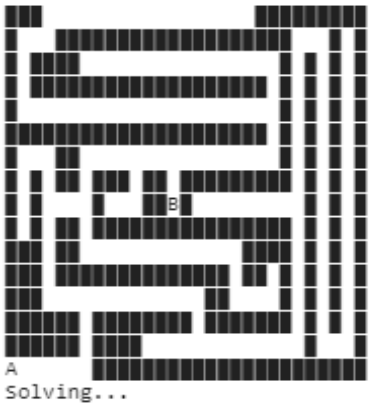
```
        # Keep track of walls
        self.walls = []
        for i in range(self.height):
            row = []
            for j in range(self.width):
                try:
                    if contents[i][j] == "A":
                        self.start = (i, j)
                        row.append(False)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

        self.solution = None


    def print(self):
        solution = self.solution[1] if self.solution is not None else None
        print()
        for i, row in enumerate(self.walls):
            for j, col in enumerate(row):
                if col:
                    print("\u2588", end="")
                elif (i, j) == self.start:
                    print("A", end="")
                elif (i, j) == self.goal:
                    print("B", end="")
                elif solution is not None and (i, j) in solution:
                    print("*", end="")
                else:
                    print(" ", end="")
            print()


    def neighbors(self, 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 < self.height and 0 <= c < self.width and not self.walls[r][c]:
                result.append((action, (r, c)))
        return result
```

```
115
116     def solve(self):
117         """Finds a solution to maze, if one exists."""
118
119         # Keep track of number of states explored
120         self.num_explored = 0
121
122         # Initialize frontier to just the starting position
123         start = Node(state=self.start, parent=None, action=None)
124         frontier = QueueFrontier()
125         frontier.add(start)
126
127         # Initialize an empty explored set
128         self.explored = set()
129
130         # Keep looping until solution found
131         while True:
132
133             # If nothing left in frontier, then no path
134             if frontier.empty():
135                 raise Exception("no solution")
136
137             # Choose a node from the frontier
138             node = frontier.remove()
139             self.num_explored += 1
140
141             # If node is the goal, then we have a solution
142             if node.state == self.goal:
143                 actions = []
144                 cells = []
145                 while node.parent is not None:
146                     actions.append(node.action)
147                     cells.append(node.state)
148                     node = node.parent
149                 actions.reverse()
150                 cells.reverse()
151                 self.solution = (actions, cells)
152                 return
153
154             # Mark node as explored
155             self.explored.add(node.state)
156
157             # Add neighbors to frontier
158             for action, state in self.neighbors(node.state):
159                 if not frontier.contains_state(state) and state not in self.explored:
160                     child = Node(state=state, parent=node, action=action)
161                     frontier.add(child)
162
163     def output_image(self, filename, show_solution=True, show_explored=False):
164         from PIL import Image, ImageDraw
165         cell_size = 50
166         cell_border = 2
167
168         # Create a blank canvas
169         img = Image.new(
170             "RGBA",
```

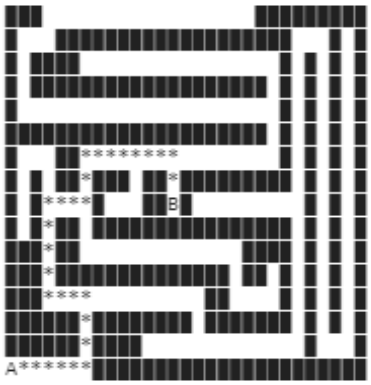
```
            (self.width * cell_size, self.height * cell_size),
            "black"
        )
        draw = ImageDraw.Draw(img)

        solution = self.solution[1] if self.solution is not None else None
        for i, row in enumerate(self.walls):
            for j, col in enumerate(row):

                # Walls
                if col:
                    fill = (40, 40, 40)

                # Start
                elif (i, j) == self.start:
                    fill = (255, 0, 0)

                # Goal
                elif (i, j) == self.goal:
                    fill = (0, 171, 28)

                # Solution
                elif solution is not None and show_solution and (i, j) in solution:
                    fill = (220, 235, 113)

                # Explored
                elif solution is not None and show_explored and (i, j) in self.explored:
                    fill = (212, 97, 85)

                # Empty cell
                else:
                    fill = (237, 240, 252)

                # Draw cell
                draw.rectangle(
                    ([(j * cell_size + cell_border, i * cell_size + cell_border),
                      ((j + 1) * cell_size - cell_border, (i + 1) * cell_size - cell_border)]),
                    fill=fill
                )

        img.save(filename)

if len(sys.argv) != 2:
    sys.exit("Usage: python maze.py maze.txt")

m = Maze(sys.argv[1])
print("Maze:")
m.print()
print("Solving...")
m.solve()
print("States Explored:", m.num_explored)
print("Solution:")
m.print()
m.output_image("maze.png", show_explored=True)
```

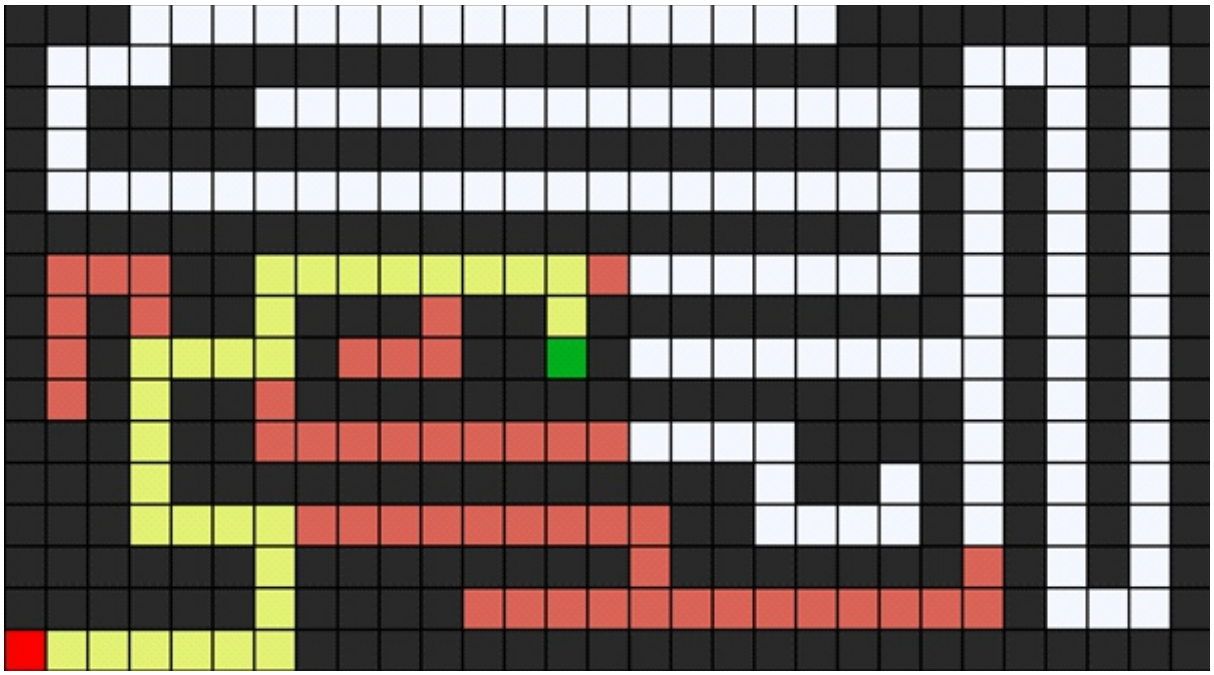
Output:

```
!python /maze.py /maze2.txt

Maze:
```



```
A
Solving...
States Explored: 77
Solution:
```





Conclusion: Hence, successfully implemented Breadth first Search and solved Maze Path finding Problem.