<u>Title</u>: Implementation of search algorithms Depth First Search (DFS)

This code is a Python implementation of a maze solver using a depth-first search (DFS) algorithm. Here's a breakdown of the code:

Overview:

This code implements a maze solver in Python using a depth-first search (DFS) algorithm. It reads a maze from a text file, solves it, prints the solution, and creates an image of the maze with the solution path.

Maze Representation:

The maze is represented as a 2D grid, where each cell is either a wall (True) or an open space (False). It includes a starting point (A) and an endpoint (B).

Classes:

The code consists of four main classes:

Node: Represents a position in the maze, including its state, parent node, and the action taken to reach it.

StackFrontier: A stack-based structure that holds nodes to explore, with methods to add, check, and remove nodes.

QueueFrontier: A queue-based structure with similar methods to StackFrontier for exploring nodes.

Maze: Manages the maze, reading it from a file, solving it with DFS, and printing the solution.

Maze Solving:

The Maze class has a `solve` method that follows these steps:

1. Start with the initial node in the frontier.

2. While there are nodes to explore:

   - Remove a node from the frontier.

- If it’s the goal, trace back to create the solution path.
- Mark the node as explored.
- Add unexplored neighbors to the frontier.

3. If no solution is found and the frontier is empty, raise an exception.

Image Generation:

The Maze class also includes an `output_image` method that creates a visual representation of the maze with the solution path using the Pillow library, with each cell measuring 50x50 pixels.

Main Program:

The main program reads the maze file, creates a Maze object, solves the maze, and outputs the solution along with the number of explored states. It also generates an image of the maze with the solution path.

Overall, this code effectively implements a maze solver using DFS and provides a visual representation of the solution.

Source Code:

```
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("█", 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()
    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

    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")
```

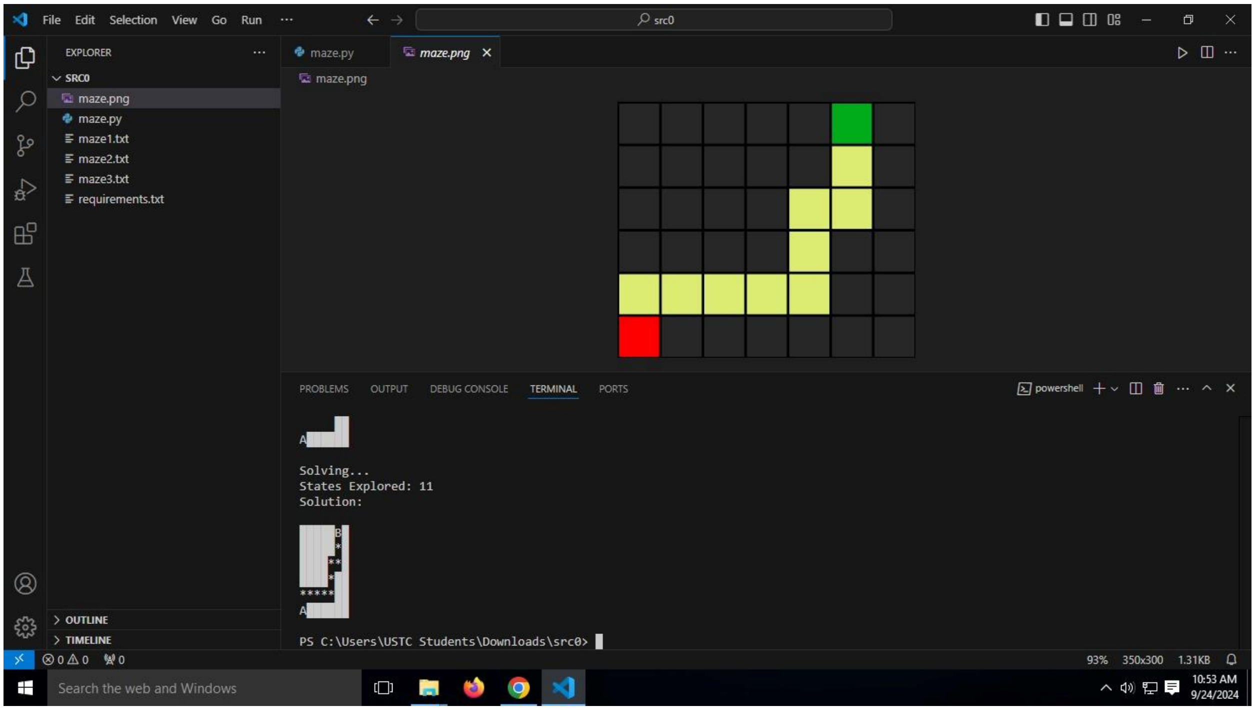
```
        # Choose a node from the frontier
        node = frontier.remove()
        self.num_explored += 1

        # 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
            actions.reverse()
            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)

    def output_image(self, filename, show_solution=True, show_explored=False):
```

```
from PIL import Image, ImageDraw
cell_size = 50
cell_border = 2

# Create a blank canvas
img = Image.new(
    "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)
```

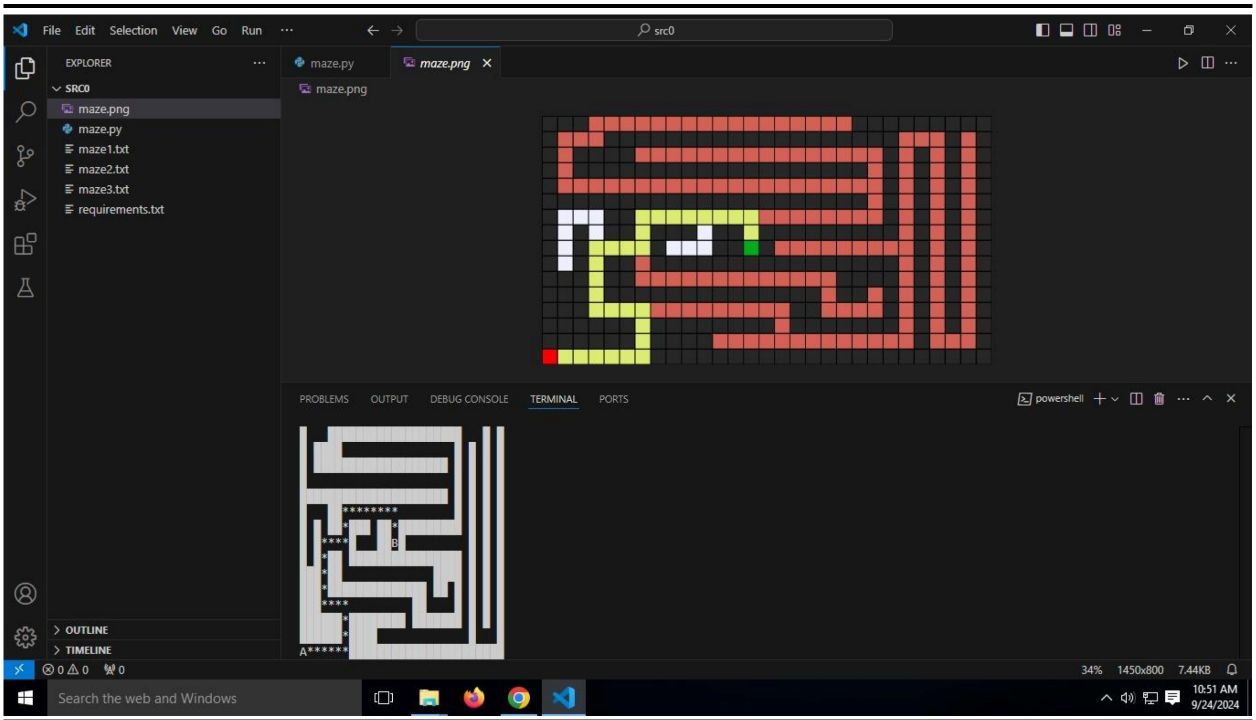
```
                # Solution
                elif solution is not None and show_solution and (i, j) in solution:
                    fill = (220, 235, 113)

                # 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)
```

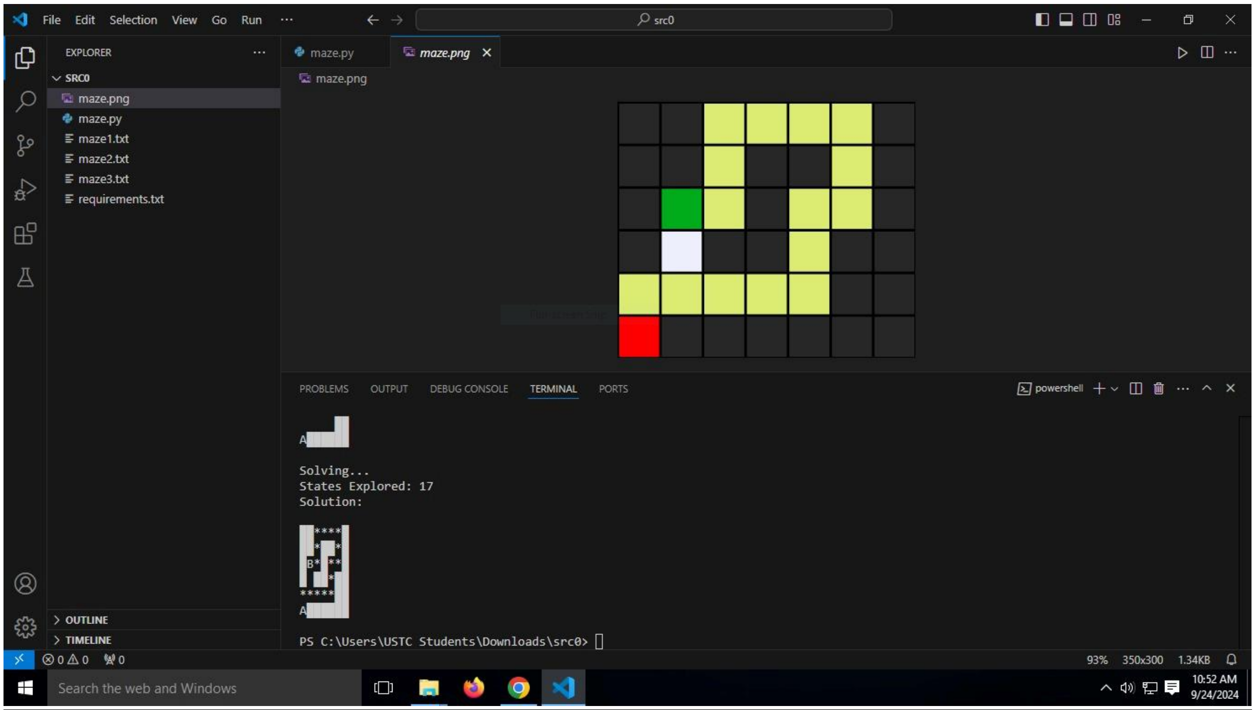
Output:



File Edit Selection View Go Run
src0
EXPLORER
SRC0
maze.png
maze.py
maze1.txt
maze2.txt
maze3.txt
requirements.txt
OUTLINE
TIMELINE
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
powershell
Solving...
States Explored: 11
Solution:
PS C:\Users\USTC Students\Downloads\src0>
93% 350x300 1.31KB
Search the web and Windows
10:53 AM
9/24/2024



File Edit Selection View Go Run
src0
EXPLORER
SRC0
maze.png
maze.py
maze1.txt
maze2.txt
maze3.txt
requirements.txt
OUTLINE
TIMELINE
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
powershell
34% 1450x800 7.44KB
Search the web and Windows
10:51 AM
9/24/2024


File Edit Selection View Go Run
src0
EXPLORER
SRC0
maze.png
maze.py
maze1.txt
maze2.txt
maze3.txt
requirements.txt
maze.py
maze.png
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
powershell
Solving...
States Explored: 17
Solution:
PS C:\Users\USTC Students\Downloads\src0>
OUTLINE
TIMELINE
93% 350x300 1.34KB
Search the web and Windows
10:52 AM
9/24/2024