Assignment 7

Name : Sanket Kele- SE-B-B2-34

Implementing a Maze Solver using AI Search Algorithms (BFS & DFS)

- **Objective:** To solve AI search problems using Graph Search Algorithms like Breadth-First Search (BFS) and Depth-First Search (DFS)[1].

Theory and Explanation

A maze can be interpreted as a graph where open paths are nodes and the possible moves between them are edges. AI search algorithms like BFS and DFS are excellent tools for navigating such graphs to find a path from a starting point to an end point.

Maze Representation

The maze is structured as a 2D grid (a list of lists in Python) where different characters represent different elements of the maze

- '#': Represents a wall or an obstacle.
- ' ': Represents an open, traversable path.
- 'S': Marks the starting point
- 'E': Marks the ending or goal point

Common Concepts for BFS & DFS

Both search algorithms utilize a few core components to navigate the maze:

1. **Finding Start and End:** The initial step is to scan the grid to find the coordinates of the 'S' and 'E' markers

2. **Defining Movements:** A set of possible moves is defined, typically up, down, left, and right. This can be represented as a list of coordinate changes:
   [(0, 1), (0, -1), (1, 0), (-1, 0)]

3. **Validation Check (is_valid):** A function is needed to check if a potential next move is valid. A move is valid if it is within the maze boundaries, is not a wall (

#), and has not been visited before[7]. This prevents the algorithm from going in circles or out of bounds.

Breadth-First Search (BFS)

BFS is an algorithm that explores a graph layer by layer. It's guaranteed to find the

shortest path from the start to the end in an unweighted graph, which makes it ideal for solving standard maze.

- **Data Structure:** BFS uses a **queue** (First-In, First-Out) to manage the nodes to. In Python,

  collections.deque is an efficient implementation of a queue.

- **Process:**
  1. Begin at the start node 'S' and add it to the queue.
  2. Mark the start node as visited to avoid cycles.
  3. While the queue is not empty, remove the first node and explore all its valid, unvisited neighbors.
  4. Add these neighbors to the queue and mark them as visited.
  5. The algorithm stops when it reaches the end node 'E'. Since it explores level by level, the first time it finds 'E', it will be via the shortest possible path.

Depth-First Search (DFS) 🧭

DFS explores a graph by going as deep as possible down one path before backtracking. It will find a path if one exists, but it

does not guarantee it will be the shortest one.

- **Data Structure:** DFS uses a **stack** (Last-In, First-Out) to manage the nodes to visit. A standard Python list can be used as a stack with
  append() to add items and pop() to remove them.

- **Process:**
  1. Begin at the start node 'S' and push it onto the stack.
  2. Mark the start node as visited.
  3. While the stack is not empty, pop a node and explore one of its valid, unvisited neighbors.

4. Push this neighbor onto the stack and mark it as visited.
5. The algorithm continues down a single path until it hits a dead end or the goal. If it hits a dead end, it backtracks by popping from the stack to try a different branch.

Python Implementation

Here is the Python code that implements both BFS and DFS to solve a given maze.

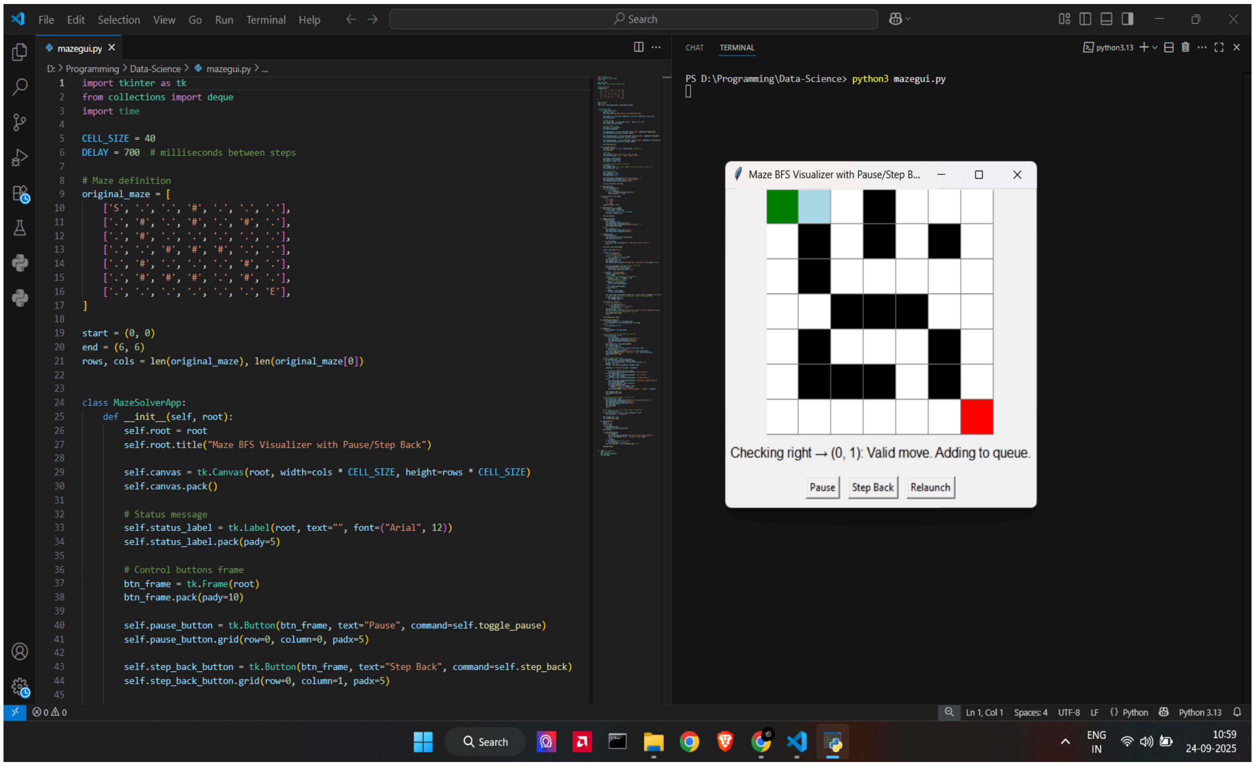
Python

```python
import collections

# Define the maze layout
maze = [
    "S #     #",
    "  # ###### #",
    "  #  #  #",
    "### # ### #",
    "# #  #  #",
    "# ###### ###",
    "#     # E"
]

def find_start_end(maze):
    """Finds the 'S' (start) and 'E' (end) coordinates in the maze."""
    for r, row in enumerate(maze):
        for c, val in enumerate(row):
            if val == 'S':
                start = (r, c)
            elif val == 'E':
                end = (r, c)
    return start, end

def print_path(maze, path):
    """Prints the maze with the found path marked by '*'."""
    maze_with_path = [list(row) for row in maze]
    # Skip the start and end points when marking the path
    for r, c in path[1:-1]:
        maze_with_path[r][c] = '*'
    for row in maze_with_path:
        print("".join(row))
```

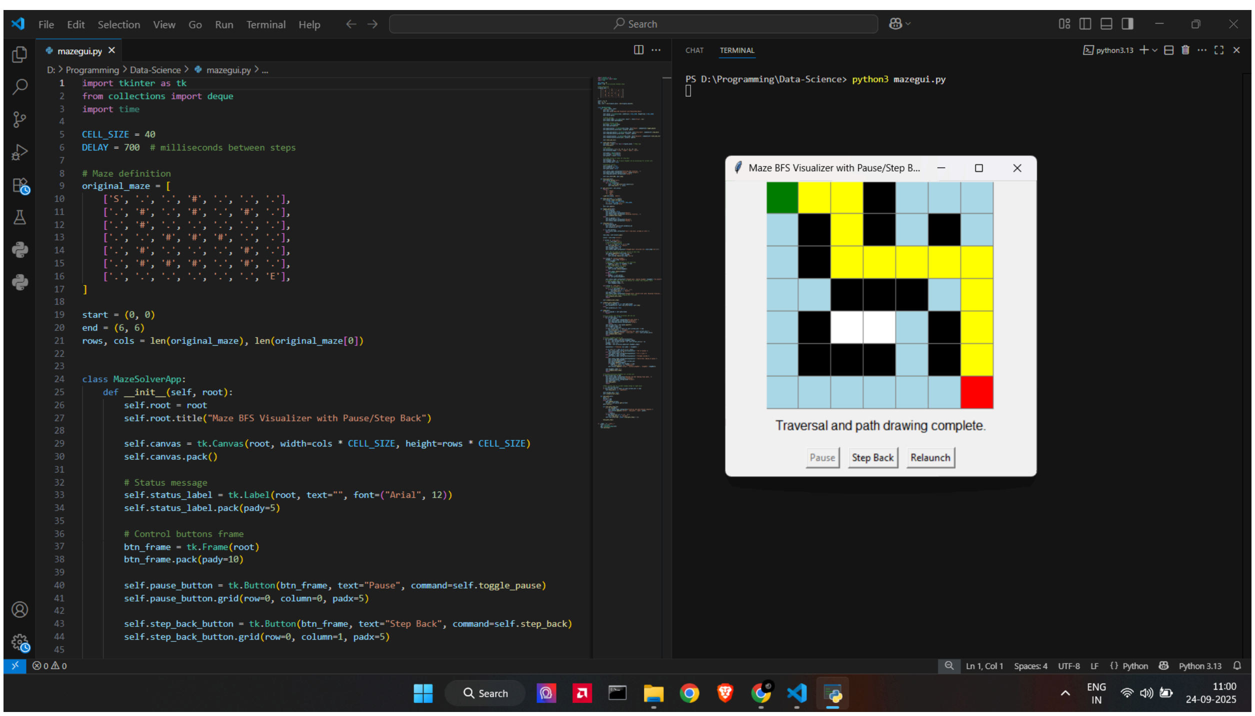
```python
def solve_maze(maze, algorithm='bfs'):
    """
    Solves the maze using either BFS or DFS.

    Args:
        maze (list of str): The maze grid.
        algorithm (str): 'bfs' or 'dfs'.

    Returns:
        list of tuples: The path from start to end, or None if no path exists.
    """
    rows, cols = len(maze), len(maze[0])
    start, end = find_start_end(maze)

    # Directions: Right, Left, Down, Up
    directions = [(0, 1), (0, -1), (1, 0), (-1, 0)]

    # Data structure for the search
    if algorithm == 'bfs':
        # Use a deque (queue) for BFS
        q = collections.deque([(start, [start])])
    else:
        # Use a list (stack) for DFS
        q = [(start, [start])]

    visited = {start}

    while q:
        if algorithm == 'bfs':
            (r, c), path = q.popleft()
        else: # dfs
            (r, c), path = q.pop()

        if (r, c) == end:
            return path

        for dr, dc in directions:
            nr, nc = r + dr, c + dc

            # Check if the new position is valid
            if (0 <= nr < rows and 0 <= nc < cols and
                    maze[nr][nc] != '#' and (nr, nc) not in visited):
```

```
                visited.add((nr, nc))
                new_path = path + [(nr, nc)]

                if algorithm == 'bfs':
                    q.append(((nr, nc), new_path))
                else: # dfs
                    q.append(((nr, nc), new_path))

    return None # No path found

# --- Main Execution ---
if __name__ == "__main__":
    print("Original Maze:")
    for row in maze:
        print(row)
    print("-" * 20)

    # Solve with BFS
    print("Solving with BFS (Shortest Path)...")
    bfs_path = solve_maze(maze, algorithm='bfs')
    if bfs_path:
        print("Path Found!")
        print_path(maze, bfs_path)
    else:
        print("No path found with BFS.")
    print("-" * 20)

    # Solve with DFS
    print("Solving with DFS (A Path)...")
    dfs_path = solve_maze(maze, algorithm='dfs')
    if dfs_path:
        print("Path Found!")
        print_path(maze, dfs_path)
    else:
        print("No path found with DFS.")
```

Results



mazegui.py
D: > Programming > Data-Science > mazegui.py > ...
import tkinter as tk
from collections import deque
import time
CELL_SIZE = 40
DELAY = 700 # milliseconds between steps
# Maze definition
original_maze = [
['S', '.', '.', '#', '.', '.', '.'],
['.', '#', '.', '#', '.', '#', '.'],
['.', '#', '.', '.', '.', '.', '.'],
['.', '.', '#', '#', '#', '.', '.'],
['.', '#', '.', '.', '.', '#', '.'],
['.', '#', '#', '#', '.', '#', '.'],
['.', '.', '.', '.', '.', '.', 'E'],
]
start = (0, 0)
end = (6, 6)
rows, cols = len(original_maze), len(original_maze[0])
class MazeSolverApp:
def __init__(self, root):
self.root = root
self.root.title("Maze BFS Visualizer with Pause/Step Back")
self.canvas = tk.Canvas(root, width=cols * CELL_SIZE, height=rows * CELL_SIZE)
self.canvas.pack()
# Status message
self.status_label = tk.Label(root, text="", font=("Arial", 12))
self.status_label.pack(pady=5)
# Control buttons frame
btn_frame = tk.Frame(root)
btn_frame.pack(pady=10)
self.pause_button = tk.Button(btn_frame, text="Pause", command=self.toggle_pause)
self.pause_button.grid(row=0, column=0, padx=5)
self.step_back_button = tk.Button(btn_frame, text="Step Back", command=self.step_back)
self.step_back_button.grid(row=0, column=1, padx=5)
PS D:\Programming\Data-Science> python3 mazegui.py
Maze BFS Visualizer with Pause/Step B...
Checking right → (0, 1): Valid move. Adding to queue.
Pause
Step Back
Relaunch
Ln 1, Col 1 Spaces: 4 UTF-8 LF Python Python 3.13
10:59
24-09-2025



mazegui.py
PS D:\Programming\Data-Science> python3 mazegui.py
Maze BFS Visualizer with Pause/Step B...
Traversal and path drawing complete.
Pause
Step Back
Relaunch
11:00
24-09-2025

Conclusion

In this experiment, we successfully implemented and compared two fundamental graph search algorithms, BFS and DFS, to solve a maze.

- **BFS** is generally preferred for maze-solving because it guarantees the **shortest path** by exploring all possible paths at a given depth before moving deeper.

- **DFS**, while often simpler to implement (especially recursively), explores one path to its absolute conclusion before backtracking. This can result in finding a solution quickly, but it is often a longer, more convoluted path than the one found by BFS.

Both algorithms demonstrated their ability to find a path from the start 'S' to the end 'E', showcasing different strategies for graph traversal. The choice between them depends entirely on the problem's requirements: **shortest path (BFS)** versus **any path (DFS)**.