King Abdulaziz University
Faculty of Computing and Information Technology
Software Engineering

CPCS-331 Artificial Intelligence (I)
AI Group Project: Solving Maze Problems
Section: CS2

Team members:

Name	ID	Email
Mohammed Al Qurayshah	2035620	mmohammedalqurayshah@stu.kau.edu.sa
Ahmed Saleh	2047450	aahmedakhtarulzaman@stu.kau.edu.sa
Rawad Al Ghamdi	2135424	2135424@stu.kau.edu.sa
Hashem Halawani	2043554	hghazihalawani@stu.kau.edu.sa
Tamim Al Humaydhi	2037095	Taalhumaydhi@stu.kau.edu.sa

Contents

Introduction

This report presents the collaborative efforts of our group in tackling an Artificial Intelligence (AI) assignment. The primary goal of this project is to develop a program in **python** to provide a detailed report on solving real maze problems using Depth-First Search (**DFS**) and Breadth-First Search (**BFS**) algorithms. Our task involves navigating through a maze environment from a start state to a goal state.

Problem Description

Maze Environment: The maze environment is a 20x20 grid with four cardinal directions: North (N), South (S), East (E), and West (W). Our agent's actions are limited to moving in these directions, allowing for left, right, or forward movement. Walls within the maze represent illegal states that the agent must avoid. The ultimate objective for the agent is to devise a plan leading from the start state to the goal state within the maze while circumventing any obstacles.

Assignment Components

Formal Analysis

PEAS (Performance Measure, Environment type, Actuators, and Sensors):

- **Performance Measure**: The agent's performance is evaluated based on the efficiency of finding a path from the start to the goal state. This is quantified by the number of steps taken, with shorter paths indicating better performance.

- **Environment Type**: The agent operates in a:
  - Partially observable: because the agent can only perceive adjacent cells.
  - Deterministic environment: because actions have predictable outcomes.

- **Actuators**: Actuators control the agent's movement in the four cardinal directions (N, S, E, W) and update its internal state and path plan.

- **Sensors**: Proximity sensors detect the presence of walls and obstacles in adjacent cells, providing the agent with information to make informed decisions.

Formulation of the Search Problem

- **Initial State**: represents the agent's starting position within the maze which is red cell.

- **Goal State**: represents the destination point the agent aims to reach, which is green cell.

- **State Space**: represents all possible positions within the maze cells, considering the presence of walls and obstacles.

- **Operators**: represents permissible actions for the agent to transition between states. These actions include moving in the North, South, East, or West direction while avoiding collisions with walls.

- **Relevant Assumptions**: We assume that the agent possesses knowledge of its current position, orientation within the maze, and the locations of walls and obstacles in adjacent cells.

Agent Program Implementation:

```
import tkinter as tk

import time


# DFS algorithm

def dfs(maze, start, end):

    stack = [start]

    visited = set()

    while stack:

        x, y = stack.pop()

        if (x, y) == end:

            return True


        if (x, y) not in visited:

            visited.add((x, y))

            canvas.create_rectangle(y * 25, x * 25, (y + 1) * 25, (x + 1) * 25, fill="red")

            root.update()

            time.sleep(0.05)  # Slow it down so you can see the progress


        for dx, dy in [(0, -1), (1, 0), (0, 1), (-1, 0)]:  # up, right, down, left (clockwise)

            nx, ny = x + dx, y + dy

            if (0 <= nx < len(maze)) and (0 <= ny < len(maze[0])) and (maze[nx][ny] != 1)
and ((nx, ny) not in visited):

                stack.append((nx, ny))
```

```
    return False


# BFS algorithm
def bfs(maze, start, end):
    queue = [start]
    visited = set()

    while queue:
        x, y = queue.pop(0)
        if (x, y) == end:
            return True

        if (x, y) not in visited:
            visited.add((x, y))
            canvas.create_rectangle(y * 25, x * 25, (y + 1) * 25, (x + 1) * 25, fill="blue")
            root.update()
            time.sleep(0.05)  # Slow it down so you can see the progress

        for dx, dy in [(0, -1), (1, 0), (0, 1), (-1, 0)]:  # up, right, down, left (clockwise)
            nx, ny = x + dx, y + dy
            if (0 <= nx < len(maze)) and (0 <= ny < len(maze[0])) and (maze[nx][ny] != 1) and ((nx, ny) not in visited):
                queue.append((nx, ny))
```

```
    return False



# Draw the initial state of the maze
def draw_maze(maze):
    for i in range(len(maze)):
        for j in range(len(maze[i])):
            color = "white"
            if maze[i][j] == 1:
                color = "black"
            elif maze[i][j] == 2:
                color = "red"
            elif maze[i][j] == 3:
                color = "green"
            canvas.create_rectangle(j * 25, i * 25, (j + 1) * 25, (i + 1) * 25, fill=color)
```

```
# Your maze definition should go here
maze = \
   [
      [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
      [1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 3, 1],
      [1, 0, 1, 0, 1, 0, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 0, 1],
      [1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 1, 0, 1],
      [1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 0, 1, 0, 1],
      [1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1],
      [1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 1],
      [1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1],
      [1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1],
      [1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1],
      [1, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1],
      [1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1],
      [1, 0, 1, 0, 1, 1, 1, 1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 0, 1],
      [1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1],
      [1, 0, 1, 1, 1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1],
      [1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 1],
      [1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 0, 1],
      [1, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1],
      [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]]
```

```
# The start and end coordinates are in (row, column) format
start = (17, 1)
end = (1, 17)

# Create the tkinter canvas
root = tk.Tk()
canvas = tk.Canvas(root, width=len(maze[0]) * 25, height=len(maze) * 25)
canvas.pack()

# Draw the initial state of the maze
draw_maze(maze)

# Call the DFS and BFS functions here:
dfs(maze, start, end)
bfs(maze, start, end)

root.mainloop()
```

References

Christian. (2017). Making a maze.

Ghallab, M. N. (n.d.). *Automated planning: Theory and practice.* Morgan Kaufmann.

Russell, S. J. (n.d.). *Artificial intelligence: A modern approach (3rd ed.).* Pearson Education.

Salviati, J. (2021). *Maze Generation Algorithms with Matrices in Python.* Python in Plain English.

Zekai, O. (2020). Fun With Python #1: Maze Generator.