DEPARTMENT OF INFORMATION TECHNOLOGY

COURSE CODE: DJ19ITL504 **DATE: 16/11/24**

COURSE NAME: Artificial Intelligence Laboratory **CLASS: TY-IT**

EXPERIMENT NO.09

CO/LO: Apply various AI approaches to knowledge intensive problem solving, reasoning, planning and uncertainty.

AIM / OBJECTIVE: Implement any AI based game: Wumpus world, Tic-tac-toe, 8-Queens Problem

DESCRIPTION OF EXPERIMENT:

Wumpus World

Wumpus World is a grid-based environment where an agent navigates to find gold while avoiding the Wumpus and pits. The agent receives sensory feedback: stench (near the Wumpus) and breeze (near a pit). The goal is to collect gold and exit the cave safely.

Procedure

1. Setup Environment:
    - Create a grid (4x4 is standard).
    - Place the Wumpus, pits, gold, and the agent in random positions, ensuring they do not overlap.
2. Define Rules:
    - If the agent moves into a cell with the Wumpus, it dies.
    - If it moves into a cell with a pit, it dies.
    - If it collects gold, the game is won.
3. Implement Agent Logic:
    - Use a basic rule-based system or a search algorithm (like A* or BFS) to navigate.
    - Incorporate the sensory feedback to make decisions.
4. Testing:
    - Run multiple simulations to ensure the agent can successfully navigate and collect gold without dying.

Tic-Tac-Toe

Tic-Tac-Toe is a two-player game played on a 3x3 grid. Players take turns placing their markers (X or O). The first player to align three markers vertically, horizontally, or diagonally wins. The game ends in a draw if the grid is filled without a winner.

Procedure

1. Setup Board:
    - Create a 3x3 matrix to represent the game board.
2. Define Game Logic:
    - Implement functions to check for a win, check for a draw, and switch turns between players.
3. Implement AI:
    - Use a simple algorithm (like Minimax) for the AI player to make optimal moves.
4. User Interface:
    - Develop a console-based or graphical interface for players to interact with the game.
5. Testing:
    - Play against the AI and against another player to verify the correctness of the implementation.

EXPLANATION / SOLUTIONS (DESIGN):

Code:

```
import random

# Initialize the grid size and entities
GRID_SIZE = 4
WUMPUS = 'W'
PIT = 'P'
GOLD = 'G'
AGENT = 'A'
EMPTY = '-'

class WumpusWorld:
    def __init__(self, size):
        self.size = size
        self.grid = [[EMPTY for _ in range(size)] for _ in range(size)]
        self.agent_position = (0, 0)
        self.gold_position = None
        self.place_entities()
```

```
    def place_entities(self):
        # Place agent
        self.grid[0][0] = AGENT

        # Place Wumpus, Gold, and Pits
        entities = [WUMPUS, GOLD] + [PIT] * (self.size - 1)
        for entity in entities:
            while True:
                x, y = random.randint(0, self.size - 1), random.randint(0,
self.size - 1)
                if self.grid[x][y] == EMPTY:
                    self.grid[x][y] = entity
                    if entity == GOLD:
                        self.gold_position = (x, y)
                    break

    def display_grid(self):
        for row in self.grid:
            print(" ".join(row))
        print()

class Agent:
    def __init__(self, world):
        self.world = world
        self.position = (0, 0)
        self.has_gold = False
        self.visited = set()
        self.visited.add(self.position)  # Mark the starting cell as
visited

    def move(self, direction):
        x, y = self.position
        if direction == 'UP' and x > 0:
            x -= 1
        elif direction == 'DOWN' and x < self.world.size - 1:
            x += 1
        elif direction == 'LEFT' and y > 0:
            y -= 1
        elif direction == 'RIGHT' and y < self.world.size - 1:
            y += 1
        else:
            return False  # Invalid move

        new_position = (x, y)
```

```
        if new_position in self.visited:
            return False  # Skip already visited cells
        self.position = new_position
        self.visited.add(new_position)
        self.check_cell()
        return True

    def check_cell(self):
        x, y = self.position
        cell_content = self.world.grid[x][y]
        if cell_content == WUMPUS or cell_content == PIT:
            print(f"Agent has died at {self.position}!")
            exit()
        elif cell_content == GOLD:
            print(f"Agent has found the gold at {self.position} and won!")
            self.has_gold = True
            exit()

    def decide_move(self):
        # Intelligent move logic
        for direction in ['UP', 'DOWN', 'LEFT', 'RIGHT']:
            if self.move(direction):
                return  # Stop after making a valid move
        print("No valid moves left. Agent is stuck!")
        exit()

# Main Execution
if __name__ == "__main__":
    world = WumpusWorld(GRID_SIZE)
    world.display_grid()

    agent = Agent(world)
    while not agent.has_gold:
        agent.decide_move()
```

Output:

```
A - - -
P P W -
G - - -
- - P -

Agent has died at (1, 0)!
Agent has found the gold at (2, 0) and won!
```

Questions:

1. Explain the algorithm used for pathfinding in Wumpus World.
2. Discuss the representation of the knowledge of the Wumpus World within the agent?
3. Explain the Minimax algorithm's role in your Tic-Tac-Toe implementation.

CONCLUSION:

The Wumpus World implementation demonstrates effective AI techniques like rule-based systems and pathfinding for knowledge-intensive problems, showcasing intelligent decision-making and navigation in uncertain environments to achieve defined goals.

REFERENCES:

[1] Stuart Russell and Peter Norvig, “Artificial Intelligence: A Modern Approach”, 2nd Edition,Pearson Education, 2010