Lab 6

23/07/2024

Brief Summary:

The script addresses the following tasks and problems:

I. Puzzle Game:

- **Initialization**:
    - The board is flattened into a tuple for easy manipulation and stored in a dictionary (state_distances) with its distance from the start state initialized to 0.

- **Breadth-First Search (BFS)**:
    - The get_paths function performs BFS by exploring all possible next states from the current state.
    - At each step, the algorithm checks if the current state is the goal state (0, 1, 2, 3, 4, 5, 6, 7, 8).
    - If the goal state is reached, the algorithm returns the number of moves taken to reach the goal.

- **State Transitions**:
    - The find_next function generates all possible next states by moving the empty tile (0) to its adjacent positions.
    - For each valid move, a new state is created and added to the list of possible next moves.

- **Main Loop**:
    - In the get_paths function, the BFS loop runs until the goal state is found or all possible states are explored.
    - At each level of BFS (each distance count), the algorithm prints the possible next moves and updates the state_distances dictionary.

- **Sleep and Print**:
    - The sleep(2) and print(next_moves) lines are added to visualize the state transitions with a 2-second delay between each step.

- **Execution Flow**

- The `solve` function initializes the state and calls `get_paths` to start the BFS process.
- The `get_paths` function repeatedly explores the current level's nodes and generates their next possible states.
- The algorithm prints the next states at each step and checks if any of them match the goal state.
- If the goal state is reached, the function returns the number of moves taken. If no solution is found, it returns -1.

II. Tic-Tac-Toe Game

• **Board Creation**:

- The create_board function initializes a 3x3 board filled with zeros, representing an empty board.

• **Finding Empty Places**:

- The possibilities function identifies and returns the list of empty positions on the board where a move can be made.

• **Making Random Moves**:

- The random_place function selects a random empty position from the list of possibilities and places the player's mark there.

• **Win Conditions**:

- The row_win, col_win, and diag_win functions check if a player has won by having three of their marks in a row, column, or diagonal, respectively.

• **Game Evaluation**:

- The evaluate function determines the game's outcome. It checks if any player has won or if the board is full, resulting in a tie.

• **Game Loop**:

- The play_game function manages the game loop. Players take turns making random moves until there is a winner or a tie.
- After each move, the board is printed, and the game pauses for 2 seconds to visualize the moves.

• **Execution Flow**

- The game starts with an empty board, no winner, and a move counter set to 1.

- Players 1 and 2 take turns making random moves.
- After each move, the board is printed, and the state is evaluated to check for a winner or a tie.
- The loop continues until a winner is found or the board is full.
- If a player wins, the game announces the winner.
- If the board is full without a winner, the game ends in a tie.
-

Code Puzzle:

```
from time import sleep

class Solution:
    def solve(self, board):
        # Dictionary to store the states and their respective distances from
the start state
        state_distances = {}

        # Flatten the 2D board to a tuple
        flatten = []
        for row in board:
            flatten += row
        flatten = tuple(flatten)

        # Initialize the dictionary with the start state
        state_distances[flatten] = 0

        # Check if the start state is the goal state
        if flatten == (0, 1, 2, 3, 4, 5, 6, 7, 8):
            return 0

        # Perform BFS to find the shortest path to the goal state
        return self.get_paths(state_distances)

    def get_paths(self, state_distances):
        cnt = 0
        while True:
            # Get all nodes at the current distance
            current_nodes = [
                node for node in state_distances if state_distances[node] ==
cnt
            ]

            # If there are no more nodes to explore, return -1 (no solution)
            if not current_nodes:
                return -1
```

```
            for node in current_nodes:
                # Get all possible next moves from the current node
                next_moves = self.find_next(node)
                sleep(2)
                print(next_moves)

                for move in next_moves:
                    if move not in state_distances:
                        # Assign the distance to the next move
                        state_distances[move] = cnt + 1

                        # Check if the next move is the goal state
                        if move == (0, 1, 2, 3, 4, 5, 6, 7, 8):
                            return cnt + 1
            cnt += 1

    def find_next(self, node):
        # Possible moves for each position on the board
        moves = {
            0: [1, 3],
            1: [0, 2, 4],
            2: [1, 5],
            3: [0, 4, 6],
            4: [1, 3, 5, 7],
            5: [2, 4, 8],
            6: [3, 7],
            7: [4, 6, 8],
            8: [5, 7],
        }

        # Find the position of the empty tile (represented by 0)
        pos_0 = node.index(0)
        results = []

        # Generate new states by swapping the empty tile with its adjacent
tiles
        for move in moves[pos_0]:
            new_node = list(node)
            new_node[move], new_node[pos_0] = new_node[pos_0],
new_node[move]
            results.append(tuple(new_node))
        return results

# Initialize the puzzle and solve it
ob = Solution()
matrix = [[3, 1, 2], [4, 7, 5], [6, 8, 0]]
print(ob.solve(matrix))
```

Output Puzzle:

```
PS D:\CSE449 - Artificial Intelligence> & "C:/Program Files/Python312/python.exe" "d:/CSE449 - Artificial Intelligenc
[(3, 1, 2, 4, 7, 0, 6, 8, 5), (3, 1, 2, 4, 7, 5, 6, 0, 8)]
[(3, 1, 0, 4, 7, 2, 6, 8, 5), (3, 1, 2, 4, 0, 7, 6, 8, 5), (3, 1, 2, 4, 7, 5, 6, 8, 0)]
[(3, 1, 2, 4, 0, 5, 6, 7, 8), (3, 1, 2, 4, 7, 5, 0, 6, 8), (3, 1, 2, 4, 7, 5, 6, 8, 0)]
[(3, 0, 1, 4, 7, 2, 6, 8, 5), (3, 1, 2, 4, 7, 0, 6, 8, 5)]
[(3, 0, 2, 4, 1, 7, 6, 8, 5), (3, 1, 2, 0, 4, 7, 6, 8, 5), (3, 1, 2, 4, 7, 0, 6, 8, 5), (3, 1, 2, 4, 8, 7, 6, 0, 5)]
[(3, 0, 2, 4, 1, 5, 6, 7, 8), (3, 1, 2, 0, 4, 5, 6, 7, 8), (3, 1, 2, 4, 5, 0, 6, 7, 8), (3, 1, 2, 4, 7, 5, 6, 0, 8)]
[(3, 1, 2, 0, 7, 5, 4, 6, 8), (3, 1, 2, 4, 7, 5, 6, 0, 8)]
[(0, 3, 1, 4, 7, 2, 6, 8, 5), (3, 1, 0, 4, 7, 2, 6, 8, 5), (3, 7, 1, 4, 0, 2, 6, 8, 5)]
[(0, 3, 2, 4, 1, 7, 6, 8, 5), (3, 2, 0, 4, 1, 7, 6, 8, 5), (3, 1, 2, 4, 0, 7, 6, 8, 5)]
[(0, 1, 2, 3, 4, 7, 6, 8, 5), (3, 1, 2, 4, 0, 7, 6, 8, 5), (3, 1, 2, 6, 4, 7, 0, 8, 5)]
[(3, 1, 2, 4, 0, 7, 6, 8, 5), (3, 1, 2, 4, 8, 7, 0, 6, 5), (3, 1, 2, 4, 8, 7, 6, 5, 0)]
[(0, 3, 2, 4, 1, 5, 6, 7, 8), (3, 2, 0, 4, 1, 5, 6, 7, 8), (3, 1, 2, 4, 0, 5, 6, 7, 8)]
[(0, 1, 2, 3, 4, 5, 6, 7, 8), (3, 1, 2, 4, 0, 5, 6, 7, 8), (3, 1, 2, 6, 4, 5, 0, 7, 8)]
4
```

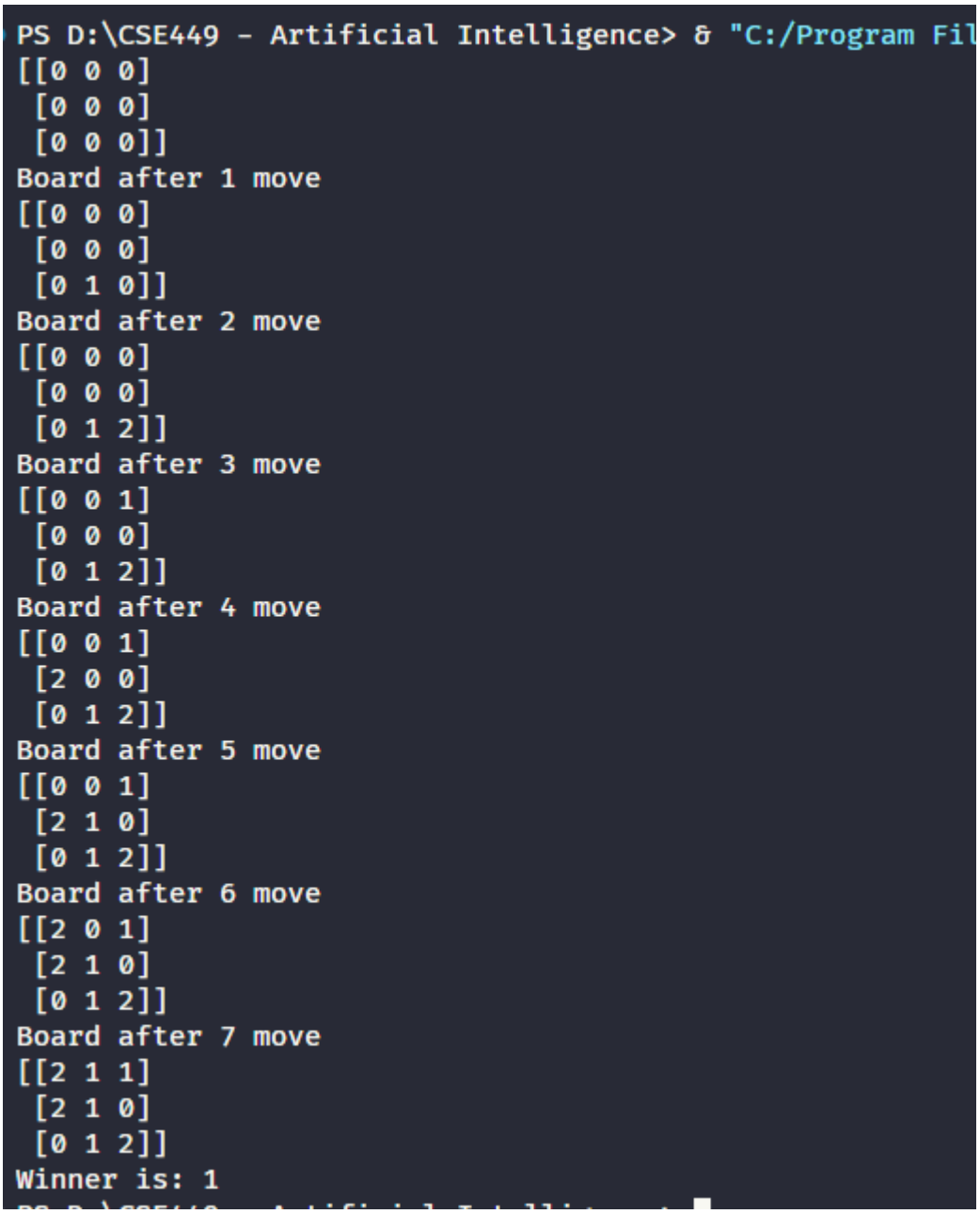
Code TicTacToe:

```
import numpy as np
import random
from time import sleep

# Creates an empty board
def create_board():
    return np.array([[0, 0, 0], [0, 0, 0], [0, 0, 0]])

# Check for empty places on board
def possibilities(board):
    l = []
    for i in range(len(board)):
        for j in range(len(board)):
            if board[i][j] == 0:
                l.append((i, j))
    return l

# Select a random place for the player
def random_place(board, player):
    selection = possibilities(board)
    current_loc = random.choice(selection)
    board[current_loc] = player
    return board

# Checks whether the player has three
# of their marks in a horizontal row
def row_win(board, player):
    for x in range(len(board)):
        win = True
        for y in range(len(board)):
            if board[x, y] != player:
```

```
                win = False
                break
        if win:
            return True
    return False

# Checks whether the player has three
# of their marks in a vertical row
def col_win(board, player):
    for x in range(len(board)):
        win = True
        for y in range(len(board)):
            if board[y][x] != player:
                win = False
                break
        if win:
            return True
    return False

# Checks whether the player has three
# of their marks in a diagonal row
def diag_win(board, player):
    win = True
    for x in range(len(board)):
        if board[x, x] != player:
            win = False
            break
    if win:
        return True
    win = True
    for x in range(len(board)):
        if board[x, len(board) - 1 - x] != player:
            win = False
            break
    return win

# Evaluates whether there is a winner or a tie
def evaluate(board):
    winner = 0
    for player in [1, 2]:
        if row_win(board, player) or col_win(board, player) or
diag_win(board, player):
            winner = player
    if np.all(board != 0) and winner == 0:
        winner = -1
    return winner

# Main function to start the game
```

```
def play_game():
    board, winner, counter = create_board(), 0, 1
    print(board)
    sleep(2)
    while winner == 0:
        for player in [1, 2]:
            board = random_place(board, player)
            print("Board after " + str(counter) + " move")
            print(board)
            sleep(2)
            counter += 1
            winner = evaluate(board)
            if winner != 0:
                break
        if winner != 0:
            break
    return winner

# Driver Code
print("Winner is: " + str(play_game()))
```

Output TicTacToe:


PS D:\CSE449 - Artificial Intelligence> & "C:/Program Fil
[[0 0 0]
 [0 0 0]
 [0 0 0]]
Board after 1 move
[[0 0 0]
 [0 0 0]
 [0 1 0]]
Board after 2 move
[[0 0 0]
 [0 0 0]
 [0 1 2]]
Board after 3 move
[[0 0 1]
 [0 0 0]
 [0 1 2]]
Board after 4 move
[[0 0 1]
 [2 0 0]
 [0 1 2]]
Board after 5 move
[[0 0 1]
 [2 1 0]
 [0 1 2]]
Board after 6 move
[[2 0 1]
 [2 1 0]
 [0 1 2]]
Board after 7 move
[[2 1 1]
 [2 1 0]
 [0 1 2]]
Winner is: 1