CS5700 - Project Assignment 3

Adversarial Search

Name : Pavan Uppala
CRN : 21569
Student ID : 700754009
Github link :
https://github.com/Pavan7947/assignment1/tree/master/project_assignment_3

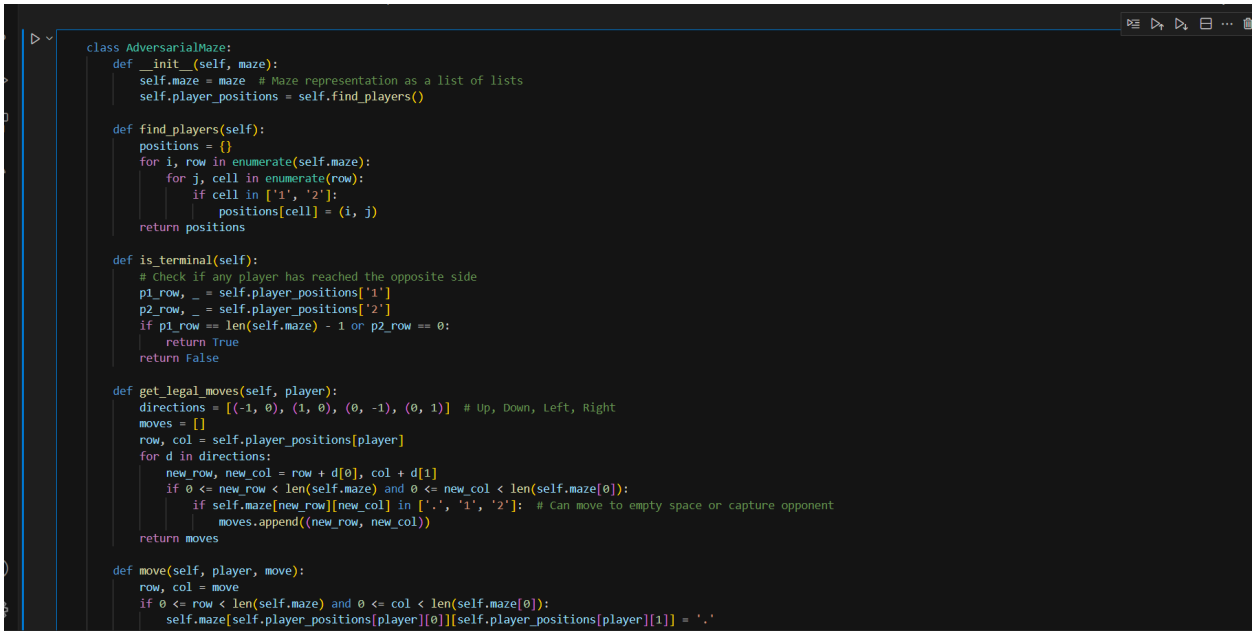
Code:

```
class AdversarialMaze:
    def __init__(self, maze):
        self.maze = maze  # Maze representation as a list of lists
        self.player_positions = self.find_players()

    def find_players(self):
        positions = {}
        for i, row in enumerate(self.maze):
            for j, cell in enumerate(row):
                if cell in ['1', '2']:
                    positions[cell] = (i, j)
        return positions

    def is_terminal(self):
        # Check if any player has reached the opposite side
        p1_row, _ = self.player_positions['1']
        p2_row, _ = self.player_positions['2']
        if p1_row == len(self.maze) - 1 or p2_row == 0:
            return True
        return False

    def get_legal_moves(self, player):
        directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]  # Up, Down, Left,
Right
        moves = []
```

```
        row, col = self.player_positions[player]
        for d in directions:
            new_row, new_col = row + d[0], col + d[1]
            if 0 <= new_row < len(self.maze) and 0 <= new_col <
len(self.maze[0]):
                if self.maze[new_row][new_col] in ['.', '1', '2']:  # Can
move to empty space or capture opponent
                    moves.append((new_row, new_col))
        return moves

    def move(self, player, move):
        row, col = move
        if 0 <= row < len(self.maze) and 0 <= col < len(self.maze[0]):

self.maze[self.player_positions[player][0]][self.player_positions[player][
1]] = '.'
            self.maze[row][col] = player
            self.player_positions[player] = move


    def evaluate(self):
        p1_row, _ = self.player_positions['1']
        p2_row, _ = self.player_positions['2']
        return p2_row - p1_row  # Difference in row positions

    def minimax(self, depth, maximizingPlayer):
        if depth == 0 or self.is_terminal():
            return self.evaluate()

        if maximizingPlayer:
            maxEval = float('-inf')
            for move in self.get_legal_moves('1'):
                self.move('1', move)
                eval = self.minimax(depth - 1, False)
                self.move('1', (self.player_positions['1'][0] - (move[0] -
self.player_positions['1'][0]),
                                self.player_positions['1'][1] - (move[1] -
self.player_positions['1'][1])))
                maxEval = max(maxEval, eval)
            return maxEval
```

```
        else:
            minEval = float('inf')
            for move in self.get_legal_moves('2'):
                self.move('2', move)
                eval = self.minimax(depth - 1, True)
                self.move('2', (self.player_positions['2'][0] - (move[0] -
self.player_positions['2'][0]),
                                self.player_positions['2'][1] - (move[1] -
self.player_positions['2'][1])))
                minEval = min(minEval, eval)
            return minEval

# Example maze setup
maze = [
    ['1', '.', '.', '.', '.', 'W', '.','.','.','.'],
    ['.', 'W', 'W', '.', '.', '.', '.','.','.','.'],
    ['.', '.', '.', 'W', '.', '.', 'W','.','W','.'],
    ['.', 'W', '.', '.', '.', '.', 'W','W','W','W'],
    ['.', '.', '.', 'W', '.', '.', '.','.','.','2'],

]
game = AdversarialMaze(maze)
print("Best outcome for Player 1:", game.minimax(4, True))
```

Screenshot:



```python
class AdversarialMaze:
    def __init__(self, maze):
        self.maze = maze  # Maze representation as a list of lists
        self.player_positions = self.find_players()

    def find_players(self):
        positions = {}
        for i, row in enumerate(self.maze):
            for j, cell in enumerate(row):
                if cell in ['1', '2']:
                    positions[cell] = (i, j)
        return positions

    def is_terminal(self):
        # Check if any player has reached the opposite side
        p1_row, _ = self.player_positions['1']
        p2_row, _ = self.player_positions['2']
        if p1_row == len(self.maze) - 1 or p2_row == 0:
            return True
        return False

    def get_legal_moves(self, player):
        directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]  # Up, Down, Left, Right
        moves = []
        row, col = self.player_positions[player]
        for d in directions:
            new_row, new_col = row + d[0], col + d[1]
            if 0 <= new_row < len(self.maze) and 0 <= new_col < len(self.maze[0]):
                if self.maze[new_row][new_col] in ['.', '1', '2']:  # Can move to empty space or capture opponent
                    moves.append((new_row, new_col))
        return moves

    def move(self, player, move):
        row, col = move
        if 0 <= row < len(self.maze) and 0 <= col < len(self.maze[0]):
            self.maze[self.player_positions[player][0]][self.player_positions[player][1]] = '.'
```

```python
    def move(self, player, move):
        row, col = move
        if 0 <= row < len(self.maze) and 0 <= col < len(self.maze[0]):
            self.maze[self.player_positions[player][0]][self.player_positions[player][1]] = '.'
            self.maze[row][col] = player
            self.player_positions[player] = move


    def evaluate(self):
        p1_row, _ = self.player_positions['1']
        p2_row, _ = self.player_positions['2']
        return p2_row - p1_row  # Difference in row positions

    def minimax(self, depth, maximizingPlayer):
        if depth == 0 or self.is_terminal():
            return self.evaluate()

        if maximizingPlayer:
            maxEval = float('-inf')
            for move in self.get_legal_moves('1'):
                self.move('1', move)
                eval = self.minimax(depth - 1, False)
                self.move('1', (self.player_positions['1'][0] - (move[0] - self.player_positions['1'][0]),
                                self.player_positions['1'][1] - (move[1] - self.player_positions['1'][1])))
                maxEval = max(maxEval, eval)
            return maxEval
        else:
            minEval = float('inf')
            for move in self.get_legal_moves('2'):
                self.move('2', move)
                eval = self.minimax(depth - 1, True)
                self.move('2', (self.player_positions['2'][0] - (move[0] - self.player_positions['2'][0]),
                                self.player_positions['2'][1] - (move[1] - self.player_positions['2'][1])))
                minEval = min(minEval, eval)
            return minEval

# Example maze setup
```

```
            maxEval = float('-inf')
            for move in self.get_legal_moves('1'):
                self.move('1', move)
                eval = self.minimax(depth - 1, False)
                self.move('1', (self.player_positions['1'][0] - (move[0] - self.player_positions['1'][0]),
                                self.player_positions['1'][1] - (move[1] - self.player_positions['1'][1])))
                maxEval = max(maxEval, eval)
            return maxEval
        else:
            minEval = float('inf')
            for move in self.get_legal_moves('2'):
                self.move('2', move)
                eval = self.minimax(depth - 1, True)
                self.move('2', (self.player_positions['2'][0] - (move[0] - self.player_positions['2'][0]),
                                self.player_positions['2'][1] - (move[1] - self.player_positions['2'][1])))
                minEval = min(minEval, eval)
            return minEval

# Example maze setup
maze = [
    ['1', '.', '.', '.', '.', 'W', '.','.','.','.','.'],
    ['.', 'W', 'W', '.', '.', '.', '.','.','.','.','.'],
    ['.', '.', '.', '.', 'W', '.', '.', 'W','.','W','.'],
    ['.', 'W', '.', '.', '.', '.', '.', 'W','W','W','W'],
    ['.', '.', '.', 'W', '.', '.', '.','.','.','.','2'],

]
game = AdversarialMaze(maze)
print("Best outcome for Player 1:", game.minimax(4, True))
```

1] ✓ 0.0s

```
Best outcome for Player 1: 4
```