Artificial Intelligence (18CSC305J)

Faculty : Helen Victoria A

Ex- 6 : Team Tesla 2.0

Name	Regno.	Email Id
Abhighyan B	RA1911033010002	ab2134@srmist.edu.in
Sanjana N B	RA1911033010016	sn8740@srmist.edu.in
Roehit Ranganathan	RA1911033010017	rr9344@srmist.edu.in
Venkata Naga Sai Ram Nomula	RA1911033010021	vn4903@srmist.edu.in
K. Dushyant Reddy	RA1911033010029	kr7119@srmist.edu.in

Experiment 6
Implementation of Minimax Algorithm

Problem Statement:

Mini-max algorithm is a recursive or backtracking algorithm which is used in decision-making and game theory. It provides an optimal move for the player assuming that the opponent is also playing optimally.

Algorithm:

1. Use a dictionary to create a board.
2. Create a print board function
3. Initiate a Loop with a range of '10' to get input from users. I.e, user with 'X' and a user with 'O'
4. Check if the user input slot is free, if not inform the user before taking input from the next user.

5. Check for winning conditions with every user input after a minimum of five inputs, there are a total of 8 combinations.
6. Print ‘GAME OVER’ if winning combination is achieved
7. Break the loop and ask the user whether they wanna play again.

Code:

```
import math
import time
import random


class Player():
    def __init__(self, letter):
        self.letter = letter

    def get_move(self, game):
        pass


class HumanPlayer(Player):
    def __init__(self, letter):
        super().__init__(letter)

    def get_move(self, game):
        valid_square = False
        val = None
        while not valid_square:
            square = input(self.letter + '\'s turn. Input move (0-9): ')
            try:
                val = int(square)
                if val not in game.available_moves():
                    raise ValueError
                valid_square = True
            except ValueError:
                print('Invalid square. Try again.')
        return val
```

```
class RandomComputerPlayer(Player):
    def __init__(self, letter):
        super().__init__(letter)

    def get_move(self, game):
        square = random.choice(game.available_moves())
        return square


class SmartComputerPlayer(Player):
    def __init__(self, letter):
        super().__init__(letter)

    def get_move(self, game):
        if len(game.available_moves()) == 9:
            square = random.choice(game.available_moves())
        else:
            square = self.minimax(game, self.letter)['position']
        return square

    def minimax(self, state, player):
        max_player = self.letter  # yourself
        other_player = 'O' if player == 'X' else 'X'

        # first we want to check if the previous move is a winner
        if state.current_winner == other_player:
            return {'position': None, 'score': 1 *
(state.num_empty_squares() + 1) if other_player == max_player else -1 * (
                        state.num_empty_squares() + 1)}
        elif not state.empty_squares():
            return {'position': None, 'score': 0}

        if player == max_player:
            best = {'position': None, 'score': -math.inf}  # each score
should maximize
        else:
            best = {'position': None, 'score': math.inf}  # each score
should minimize
        for possible_move in state.available_moves():
```

```
            state.make_move(possible_move, player)
            sim_score = self.minimax(state, other_player)  # simulate a game after making that move

            # undo move
            state.board[possible_move] = ' '
            state.current_winner = None
            sim_score['position'] = possible_move  # this represents the move optimal next move

            if player == max_player:  # X is max player
                if sim_score['score'] > best['score']:
                    best = sim_score
            else:
                if sim_score['score'] < best['score']:
                    best = sim_score
        return best


class TicTacToe():
    def __init__(self):
        self.board = self.make_board()
        self.current_winner = None

    @staticmethod
    def make_board():
        return [' ' for _ in range(9)]

    def print_board(self):
        for row in [self.board[i*3:(i+1) * 3] for i in range(3)]:
            print('| ' + ' | '.join(row) + ' |')

    @staticmethod
    def print_board_nums():
        # 0 | 1 | 2
        number_board = [[str(i) for i in range(j*3, (j+1)*3)] for j in range(3)]
        for row in number_board:
            print('| ' + ' | '.join(row) + ' |')
```

```python
    def make_move(self, square, letter):
        if self.board[square] == ' ':
            self.board[square] = letter
            if self.winner(square, letter):
                self.current_winner = letter
            return True
        return False

    def winner(self, square, letter):
        # check the row
        row_ind = math.floor(square / 3)
        row = self.board[row_ind*3:(row_ind+1)*3]
        # print('row', row)
        if all([s == letter for s in row]):
            return True
        col_ind = square % 3
        column = [self.board[col_ind+i*3] for i in range(3)]
        # print('col', column)
        if all([s == letter for s in column]):
            return True
        if square % 2 == 0:
            diagonal1 = [self.board[i] for i in [0, 4, 8]]
            # print('diag1', diagonal1)
            if all([s == letter for s in diagonal1]):
                return True
            diagonal2 = [self.board[i] for i in [2, 4, 6]]
            # print('diag2', diagonal2)
            if all([s == letter for s in diagonal2]):
                return True
        return False

    def empty_squares(self):
        return ' ' in self.board

    def num_empty_squares(self):
        return self.board.count(' ')

    def available_moves(self):
        return [i for i, x in enumerate(self.board) if x == " "]
```

```python
def play(game, x_player, o_player, print_game=True):

    if print_game:
        game.print_board_nums()

    letter = 'X'
    while game.empty_squares():
        if letter == 'O':
            square = o_player.get_move(game)
        else:
            square = x_player.get_move(game)
        if game.make_move(square, letter):

            if print_game:
                print(letter + ' makes a move to square
{}'.format(square))
                game.print_board()
                print('')

            if game.current_winner:
                if print_game:
                    print(letter + ' wins!')
                return letter  # ends the loop and exits the game
            letter = 'O' if letter == 'X' else 'X'  # switches player

        time.sleep(.8)

    if print_game:
        print('It\'s a tie!')



if __name__ == '__main__':
    x_player = SmartComputerPlayer('X')
    o_player = HumanPlayer('O')
    t = TicTacToe()
    play(t, x_player, o_player, print_game=True)
```

Output:

```
PS C:\Users\ranga> python -u "d:\SRM\SEM 6\AI lab\EXP-6\exp6.py"
| 0 | 1 | 2 |
| 3 | 4 | 5 |
| 6 | 7 | 8 |
X makes a move to square 2
|   |   | X |
|   |   |   |
|   |   |   |

O's turn. Input move (0-9): 5
O makes a move to square 5
|   |   | X |
|   |   | O |
|   |   |   |

X makes a move to square 0
| X |   | X |
|   |   | O |
|   |   |   |

O's turn. Input move (0-9): 1
O makes a move to square 1
| X | O | X |
|   |   | O |
|   |   |   |

X makes a move to square 4
| X | O | X |
|   | X | O |
|   |   |   |

O's turn. Input move (0-9): 2
Invalid square. Try again.
O's turn. Input move (0-9): 8
O makes a move to square 8
| X | O | X |
|   | X | O |
|   |   | O |

X makes a move to square 6
| X | O | X |
|   | X | O |
| X |   | O |

X wins!
PS C:\Users\ranga>
```

Time Complexity and Space Complexity :

The time complexity of minimax is **O(b^m)** and the space complexity is **O(bm)**, where b is the number of legal moves at each point and m is the maximum depth of the tree.

Real World Solution:

- It is used for game theory in sports.

Result: Min max algorithm is successfully implemented.