AI LAB WEEK 6

Implementation of minmax algorithm for an application

Aim: To implement minmax algorithm for an application

Code:

import random

class TicTacToe(object):

    winning\_combos = (

    [0, 1, 2], [3, 4, 5], [6, 7, 8],

    [0, 3, 6], [1, 4, 7], [2, 5, 8],

    [0, 4, 8], [2, 4, 6]

    )

    winners = ('X-win', 'Draw', 'O-win')

    def \_\_init\_\_(self, board=[]):

        '''

        Initialize the tic tac toe board

        :param board: 1-D list of board positions

        '''

        if len(board) == 0:

          self.board = [0 for i in range(9)]

        else:

          self.board = board

    def print\_board(self):

        '''

        Printing the tic tac toe board

        '''

        for i in range(3):

          print(

          "| " + str(self.board[i \* 3]) +

          " | " + str(self.board[i \* 3 + 1]) +

          " | " + str(self.board[i \* 3 + 2]) + " |"

          )

    def check\_game\_over(self):

        '''

        Check if the game is over or there is a winner

        '''

        if 0 not in [element for element in self.board]:

          return True

        if self.winner() != 0:

          return True

        return False

    def available\_moves(self):

        '''

        To check what all possible moves are remaining for a player

        '''

        return [index for index, element in enumerate(self.board) if element is 0]

    def available\_combos(self, player):

        '''

        To check what are the possible places to play for winning the game

        '''

        return self.available\_moves() + self.get\_acquired\_places(player)

    def X\_won(self):

        return self.winner() == 'X'

    def O\_won(self):

        return self.winner() == 'O'

    def is\_tie(self):

        return self.winner() == 0 and self.check\_game\_over()

    def winner(self):

        '''

        Checks for the winner of the game

        :return player: return 'X' or 'O' whoever has won the game

        else returns 0

        '''

        for player in ('X', 'O'):

            positions = self.get\_acquired\_places(player)

            for combo in self.winning\_combos:

              win = True

              for pos in combo:

                  if pos not in positions:

                      win = False

              if win:

                  return player

        return 0

    def get\_acquired\_places(self, player):

        '''

        To get the positions already acquired by a particular player

        :param player: 'X' or 'O'

        '''

        return [index for index, element in enumerate(self.board) if element == player]

    def make\_move(self, position, player):

        self.board[position] = player

    def minimax(self, node, player):

        '''

        Minimax algorithm for choosing the best possible move towards

        winning the game

        '''

        if node.check\_game\_over():

          if node.X\_won():

              return -1

          elif node.is\_tie():

              return 0

          elif node.O\_won():

              return 1

        best = 0

        for move in node.available\_moves():

          node.make\_move(move, player)

          val = self.minimax(node, get\_enemy(player))

          node.make\_move(move, 0)

          if player == 'O':

            if val > best:

              best = val

          else:

            if val < best:

              best = val

        return best

def determine(board, player):

        '''

        Driver function to apply minimax algorithm

        '''

        a = 0

        choices = []

        if len(board.available\_moves()) == 9:

            return 4

        for move in board.available\_moves():

            board.make\_move(move, player)

            val = board.minimax(board, get\_enemy(player))

            board.make\_move(move, 0)

            if val > a:

              a = val

              choices = [move]

            elif val == a:

              choices.append(move)

        try:

            return random.choice(choices)

        except IndexError:

            return random.choice(board.available\_moves())

def get\_enemy(player):

      if player == 'X':

        return 'O'

      return 'X'

board = TicTacToe()

print('Board positions are like this: ')

for i in range(3):

    print(

    "| " + str(i \* 3 + 1) +

    " | " + str(i \* 3 + 2) +

    " | " + str(i \* 3 + 3) + " |"

    )

print('Type in the position number you to make a move on..')

while not board.check\_game\_over():

  player = 'X'

  player\_move = int(input("Your Move: ")) - 1

  if player\_move not in board.available\_moves():

    print('Please check the input!')

    continue

  board.make\_move(player\_move, player)

  board.print\_board()

  print()

  if board.check\_game\_over():

    break

  print('Computer is playing.. ')

  player = get\_enemy(player)

  computer\_move = determine(board, player)

  board.make\_move(computer\_move, player)

  board.print\_board()

if board.winner() != 0:

  if board.winner() == 'X':

    print ("Congratulations you win!")

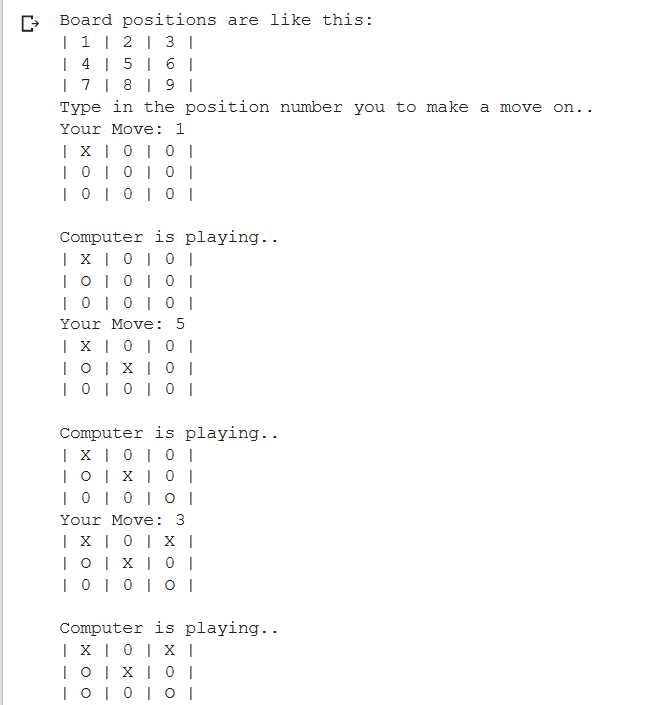
  else:

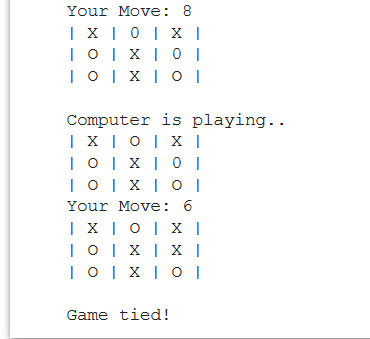
    print('Computer Wins!')

else:

  print("Game tied!")

Output:





Result:

Minmax algorithm for an application was implemented successfully.