ARTIFICIAL INTELLIGENCE LAB (CSL5402)

Name: Lakhan Kumawat

Roll: 1906055

Program: B.Tech CSE
(5th Sem JUL-DEC 2021)

Assignment - 10

1. Write a program to implement the Tic-Tac-Toe problem with the help of Minimax algorithm.

```
from math import inf as infinity
from random import choice
import platform
import time
from os import system




HUMAN = -1
COMP = +1
board = [
    [0, 0, 0],
    [0, 0, 0],
    [0, 0, 0],
]


def evaluate(state):
    """
    Function to heuristic evaluation of state.
    :param state: the state of the current board
    :return: +1 if the computer wins; -
1 if the human wins; 0 draw
    """
    if wins(state, COMP):
        score = +1
    elif wins(state, HUMAN):
        score = -1
    else:
        score = 0
```

```
    return score


def wins(state, player):
    """
    This function tests if a specific player wins. Possib
ilities:
    * Three rows    [X X X] or [O O O]
    * Three cols    [X X X] or [O O O]
    * Two diagonals [X X X] or [O O O]
    :param state: the state of the current board
    :param player: a human or a computer
    :return: True if the player wins
    """
    win_state = [
        [state[0][0], state[0][1], state[0][2]],
        [state[1][0], state[1][1], state[1][2]],
        [state[2][0], state[2][1], state[2][2]],
        [state[0][0], state[1][0], state[2][0]],
        [state[0][1], state[1][1], state[2][1]],
        [state[0][2], state[1][2], state[2][2]],
        [state[0][0], state[1][1], state[2][2]],
        [state[2][0], state[1][1], state[0][2]],
    ]
    if [player, player, player] in win_state:
        return True
    else:
        return False


def game_over(state):
    """
    This function test if the human or computer wins
    :param state: the state of the current board
    :return: True if the human or computer wins
    """
    return wins(state, HUMAN) or wins(state, COMP)
```

```python
def empty_cells(state):
    """
    Each empty cell will be added into cells' list
    :param state: the state of the current board
    :return: a list of empty cells
    """
    cells = []

    for x, row in enumerate(state):
        for y, cell in enumerate(row):
            if cell == 0:
                cells.append([x, y])

    return cells


def valid_move(x, y):
    """
    A move is valid if the chosen cell is empty
    :param x: X coordinate
    :param y: Y coordinate
    :return: True if the board[x][y] is empty
    """
    if [x, y] in empty_cells(board):
        return True
    else:
        return False


def set_move(x, y, player):
    """
    Set the move on board, if the coordinates are valid
    :param x: X coordinate
    :param y: Y coordinate
    :param player: the current player
    """
```

```
    if valid_move(x, y):
        board[x][y] = player
        return True
    else:
        return False


def minimax(state, depth, player):
    """
    AI function that choice the best move
    :param state: current state of the board
    :param depth: node index in the tree (0 <= depth <= 9
),
    but never nine in this case (see iaturn() function)
    :param player: an human or a computer
    :return: a list with [the best row, best col, best sc
ore]
    """
    if player == COMP:
        best = [-1, -1, -infinity]
    else:
        best = [-1, -1, +infinity]

    if depth == 0 or game_over(state):
        score = evaluate(state)
        return [-1, -1, score]

    for cell in empty_cells(state):
        x, y = cell[0], cell[1]
        state[x][y] = player
        score = minimax(state, depth - 1, -player)
        state[x][y] = 0
        score[0], score[1] = x, y

        if player == COMP:
            if score[2] > best[2]:
                best = score  # max value
```

```
        else:
            if score[2] < best[2]:
                best = score  # min value

    return best


def clean():
    """
    Clears the console
    """
    os_name = platform.system().lower()
    if 'windows' in os_name:
        system('cls')
    else:
        system('clear')


def render(state, c_choice, h_choice):
    """
    Print the board on console
    :param state: current state of the board
    """

    chars = {
        -1: h_choice,
        +1: c_choice,
        0: ' '
    }
    str_line = '---------------'

    print('\n' + str_line)
    for row in state:
        for cell in row:
            symbol = chars[cell]
            print(f'| {symbol} |', end='')
        print('\n' + str_line)
```

```python
def ai_turn(c_choice, h_choice):
    """
    It calls the minimax function if the depth < 9,
    else it choices a random coordinate.
    :param c_choice: computer's choice X or O
    :param h_choice: human's choice X or O
    :return:
    """
    depth = len(empty_cells(board))
    if depth == 0 or game_over(board):
        return

    clean()
    print(f'Computer turn [{c_choice}]')
    render(board, c_choice, h_choice)

    if depth == 9:
        x = choice([0, 1, 2])
        y = choice([0, 1, 2])
    else:
        move = minimax(board, depth, COMP)
        x, y = move[0], move[1]

    set_move(x, y, COMP)
    time.sleep(1)


def human_turn(c_choice, h_choice):
    """
    The Human plays choosing a valid move.
    :param c_choice: computer's choice X or O
    :param h_choice: human's choice X or O
    :return:
    """
    depth = len(empty_cells(board))
    if depth == 0 or game_over(board):
        return
```

```
    # Dictionary of valid moves
    move = -1
    moves = {
        1: [0, 0], 2: [0, 1], 3: [0, 2],
        4: [1, 0], 5: [1, 1], 6: [1, 2],
        7: [2, 0], 8: [2, 1], 9: [2, 2],
    }

    clean()
    print(f'Human turn [{h_choice}]')
    render(board, c_choice, h_choice)

    while move < 1 or move > 9:
        try:
            move = int(input('Use numpad (1..9): '))
            coord = moves[move]
            can_move = set_move(coord[0], coord[1], HUMAN
)

            if not can_move:
                print('Bad move')
                move = -1
        except (EOFError, KeyboardInterrupt):
            print('Bye')
            exit()
        except (KeyError, ValueError):
            print('Bad choice')


def main():
    """
    Main function that calls all functions
    """
    clean()
    h_choice = ''  # X or O
    c_choice = ''  # X or O
```

```
    first = ''  # if human is the first

    # Human chooses X or O to play
    while h_choice != 'O' and h_choice != 'X':
        try:
            print('')
            h_choice = input('Choose X or O\nChosen: ').u
pper()
        except (EOFError, KeyboardInterrupt):
            print('Bye')
            exit()
        except (KeyError, ValueError):
            print('Bad choice')

    # Setting computer's choice
    if h_choice == 'X':
        c_choice = 'O'
    else:
        c_choice = 'X'

    # Human may starts first
    clean()
    while first != 'Y' and first != 'N':
        try:
            first = input('First to start?[y/n]: ').upper
()
        except (EOFError, KeyboardInterrupt):
            print('Bye')
            exit()
        except (KeyError, ValueError):
            print('Bad choice')

    # Main loop of this game
    while len(empty_cells(board)) > 0 and not game_over(b
oard):
        if first == 'N':
            ai_turn(c_choice, h_choice)
```

```
            first = ''

        human_turn(c_choice, h_choice)
        ai_turn(c_choice, h_choice)

    # Game over message
    if wins(board, HUMAN):
        clean()
        print(f'Human turn [{h_choice}]')
        render(board, c_choice, h_choice)
        print('YOU WIN!')
    elif wins(board, COMP):
        clean()
        print(f'Computer turn [{c_choice}]')
        render(board, c_choice, h_choice)
        print('YOU LOSE!')
    else:
        clean()
        render(board, c_choice, h_choice)
        print('DRAW!')

    exit()


if __name__ == '__main__':
    main()
```

OUTPUT:

```
Choose X or O
Chosen: X
First to start?[y/n]: y
Human turn [X]

---------------
|   ||   ||   |
---------------
|   ||   ||   |
---------------
|   ||   ||   |
```

Name : Lakhan Kumawat Roll No : 1906055 Course : CSL5402

```
---------------
Use numpad (1..9): 1
Computer turn [O]

---------------
| X ||   ||   |
---------------
|   ||   ||   |
---------------
|   ||   ||   |
---------------
Human turn [X]

---------------
| X ||   ||   |
---------------
|   || O ||   |
---------------
|   ||   ||   |
---------------
Use numpad (1..9): 7
Computer turn [O]

---------------
| X ||   ||   |
---------------
|   || O ||   |
---------------
| X ||   ||   |
---------------
Human turn [X]

---------------
| X ||   ||   |
---------------
| O || O ||   |
---------------
| X ||   ||   |
---------------
Use numpad (1..9): 6
Computer turn [O]

---------------
| X ||   ||   |
---------------
| O || O || X |
---------------
| X ||   ||   |
---------------
Human turn [X]

---------------
| X || O ||   |
---------------
```

```
| O || O || X |
---------------
| X ||   ||   |
---------------
Use numpad (1..9): 8
Computer turn [O]

---------------
| X || O ||   |
---------------
| O || O || X |
---------------
| X || X ||   |
---------------
Human turn [X]

---------------
| X || O ||   |
---------------
| O || O || X |
---------------
| X || X || O |
---------------
Use numpad (1..9): 3

---------------
| X || O || X |
---------------
| O || O || X |
---------------
| X || X || O |
---------------
DRAW!
```

Q2 Write a program to implement the 8-puzzle problem using DFS.

```
class puzzle_8:
    def __init__(self):
        self.board=[[6,5,2],[1,7,3],[4,0,8]]#initial board position
        self.target=[[1,2,3],[4,5,6],[7,8,0]]#target board position
        self.visited=set() #set containing the visited states
```

```
          self.success=False #variable indicating if the
puzzle is solved
          self.path=[] #list containing the moves used in
 solving the puzzle
          for i in range(3):#setting the position of empt
y space
              for j in range(3):
                  if self.board[i][j]==0:
                      self.row_pos=i
                      self.col_pos=j
    def swap(self,curr,dest):
        t=self.board[curr[0]][curr[1]]
        self.board[curr[0]][curr[1]]=self.board[dest[0]][
dest[1]]
        self.board[dest[0]][dest[1]]=t
    def move_up(self):#moving the empty space up
          if self.row_pos==0:
              return False
          self.swap((self.row_pos,self.col_pos),(self.row
_pos-1,self.col_pos))
          self.row_pos-=1
          return True
    def move_down(self):#moving the empty space down
          if self.row_pos==2:
                return False
          self.swap((self.row_pos,self.col_pos),(self.row
_pos+1,self.col_pos))
          self.row_pos+=1
          return True
    def move_left(self):#moving the empty space left
            if self.col_pos==0:
                return False
            self.swap((self.row_pos,self.col_pos),(self.r
ow_pos,self.col_pos-1))
            self.col_pos-=1
            return True
```

```
    def move_right(self):#moving the empty space right
            if self.col_pos==2:
                return False
            self.swap((self.row_pos,self.col_pos),(self.r
ow_pos,self.col_pos+1))
            self.col_pos+=1
            return True
    def dfs_solve(self,depth,depthlimit):#function to sol
ve the puzzle using dfs
             if self.success==True:
                 return
             elif depth==depthlimit:
                 return
             elif self.board==self.target:
                 print('success')
                 self.success=True
             elif str(self.board) in self.visited:
                 return
             else:
                 self.visited.add(str(self.board))
                 if self.move_up():
                     self.dfs_solve(depth+1,depthlimit)
                     self.move_down()
                     if self.success:
                          self.path.append('up')
                          return
                 if self.move_left():
                   self.dfs_solve(depth+1,depthlimit)
                   self.move_right()
                   if self.success:
                      self.path.append('left')
                      return
                 if self.move_down():
                    self.dfs_solve(depth+1,depthlimit)
                    self.move_up()
                    if self.success:
                      self.path.append('down')
```

```
                        return
                    if self.move_right():
                        self.dfs_solve(depth+1,depthlimit)
                        self.move_left()
                        if self.success:
                            self.path.append('right')
                            return

    def print_path(self):
        for i in range(len(self.path)-1,-1,-1):
            print(self.board)
            if self.path[i]=='up':
              self.move_up()
            elif self.path[i]=='down':
              self.move_down()
            elif self.path[i]=='left':
              self.move_left()
            elif self.path[i]=='right':
              self.move_right()
        print(self.board)

if __name__=='__main__':
   p=puzzle_8()
   for i in range(0,1000):
      p.dfs_solve(0,i)
      p.visited=set()
      if p.success:
          print('the path followed is:')
          for j in range(len(p.path)-1,-1,-1):
              print(p.path[j],end=' ')
          print('')
          p.print_path()
          break;
   if not p.success:
      print('unable to solve')
```

OUTPUT:

```
success
the path followed is:
up up left down down right up up right down left down right
[[6, 5, 2], [1, 7, 3], [4, 0, 8]]
[[6, 5, 2], [1, 0, 3], [4, 7, 8]]
[[6, 0, 2], [1, 5, 3], [4, 7, 8]]
[[0, 6, 2], [1, 5, 3], [4, 7, 8]]
[[1, 6, 2], [0, 5, 3], [4, 7, 8]]
[[1, 6, 2], [4, 5, 3], [0, 7, 8]]
[[1, 6, 2], [4, 5, 3], [7, 0, 8]]
[[1, 6, 2], [4, 0, 3], [7, 5, 8]]
[[1, 0, 2], [4, 6, 3], [7, 5, 8]]
[[1, 2, 0], [4, 6, 3], [7, 5, 8]]
[[1, 2, 3], [4, 6, 0], [7, 5, 8]]
[[1, 2, 3], [4, 0, 6], [7, 5, 8]]
[[1, 2, 3], [4, 5, 6], [7, 0, 8]]
[[1, 2, 3], [4, 5, 6], [7, 8, 0]]
```

Q3 Write a program to implement the 8-queens problem using hill climbing algorithm.*

```
import copy
import numpy as np
import chess
import sys
from chess import svg
import matplotlib.pyplot as plt
%matplotlib inline


def exists(i, j):
    # Checks if square exists within boundary

    return (i >= 0 and i < 8 and j >= 0 and j < 8)


def contains(i, j, l, m, queen_pairs):
```

```
    # Check if the two pair of queens have already been i
ncluded in count

    if ((i, j, l, m) in queen_pairs) or ((l, m, i, j) in
queen_pairs):
        return True
    return False


def save_board_as_png(fen):
    # Converts the FEN format notation to an SVG chess bo
ard

    board = chess.Board(fen)
    print(board)



def create_board(board):
    # Creates a python-chess board for the matrix board

    chess_board = chess.Board()
    chess_board.clear()

    for i in range(8):
        for j in range(8):
            if board[i][j]:
                chess_board.set_piece_at(chess.square(
                    i, j), chess.Piece(5, chess.WHITE))

    return chess_board.fen()


def position_queens_row_wise(board):
    """Place a single queen on every row. If there are mo
re than
    two quueens in one row, it places them on other rows"
""
```

```
    for row in board:
        while row.count(1) > 1:
            # More than one 1s so distribute to other row
s

            for i in range(8):
                if board[i].count(1) == 0:
                    j = row.index(1)
                    board[i][j] = 1
                    row[j] = 0
                    break

    return board


def heuristic_value(board):
    # Calculates the heuristic value h of the current sta
te of board
    # Number of pairs of queens attacking each other dire
ctly or indirectly

    h = 0
    queen_pairs = []
    for i in range(8):
        for j in range(8):

            if board[i][j]:

                # Calculate horizontal attacks
                for k in range(8):
                    if board[i][k] == 1 and k != j and no
t contains(i, j, i, k, queen_pairs):
                        queen_pairs.append((i, j, i, k))
                        h += 1

                # Calculate vertical attacks
```

```
                for k in range(8):
                    if board[k][j] == 1 and i != k and no
t contains(i, j, k, j, queen_pairs):
                        queen_pairs.append((i, j, k, j))
                        h += 1

                # Calculate / diagonal attacks
                # First go up the diagonal
                l, m = i-1, j+1
                while exists(l, m):
                    if board[l][m] == 1 and not contains(
i, j, l, m, queen_pairs):
                        queen_pairs.append((i, j, l, m))
                        h += 1
                    l, m = l-1, m+1

                # Now go down the diagonal
                l, m = i+1, j-1
                while exists(l, m):
                    if board[l][m] == 1 and not contains(
i, j, l, m, queen_pairs):
                        queen_pairs.append((i, j, l, m))
                        h += 1
                    l, m = l+1, m-1

                # Calculate \ diagonal attacks
                # First go up the diagonal
                l, m = i-1, j-1
                while exists(l, m):
                    if board[l][m] == 1 and not contains(
i, j, l, m, queen_pairs):
                        queen_pairs.append((i, j, l, m))
                        h += 1
                    l, m = l-1, m-1

                # Now go down the diagonal
                l, m = i+1, j+1
```

```
                while exists(l, m):
                    if board[l][m] == 1 and not contains(
i, j, l, m, queen_pairs):
                        queen_pairs.append((i, j, l, m))
                        h += 1
                    l, m = l+1, m+1

    return h


def hill_climbing(board):
    # Find the least cost successor for the given board s
tate

    min_board = board
    min_h = 999999
    global n_side_moves, n_steps

    n_steps += 1

    # Check if number of side moves has reached a limit
    if n_side_moves == 100:
        return -1

    sideway_move = False

    for i in range(8):
        # Find index of queen in current row
        queen = board[i].index(1)

        board[i][queen] = 0

        for k in range(8):
            # Place queen at different positions and calc
ulate new score

            if k != queen:
```

```
                board[i][k] = 1

                h = heuristic_value(board)

                if h < min_h:
                    min_h = h
                    min_board = copy.deepcopy(board)
                if h == min_h:
                    min_h = h
                    min_board = copy.deepcopy(board)
                    sideway_move = True

                board[i][k] = 0

        board[i][queen] = 1

    if sideway_move:
        n_side_moves += 1

    if min_h == 0:
        print("Number of steps required: {}".format(n_ste
ps))
        return min_board

    return hill_climbing(min_board)


if __name__ == "__main__":
    # 8x8 chess board
    board = [[0,1,0,0,0,0,0,0],[0,0,1,0,0,0,0,0],[1,0,0,0
,0,0,0,0],[0,1,0,0,0,0,0,0],[0,1,0,0,0,0,0,0],[0,1,0,0,0,
0,0,0],[0,1,0,0,0,0,0,0],[0,1,0,0,0,0,0,0]]
    n_side_moves = 0
    n_steps = 0
```

```
    print("Current position's heuristic value: ", heuristic_value(board))

    board = position_queens_row_wise(board)
    min_board = hill_climbing(board)

    if min_board != -1:
        fen = create_board(min_board)
        save_board_as_png(fen)

    else:
        print("Could not solve")
```

OUTPUT:

```
Current position's heuristic value:  17
Number of steps required: 5
. . . Q . . . .
. . . . . . . Q
Q . . . . . . .
. . . . Q . . .
. . . . . . Q .
. Q . . . . . .
. . . . . Q . .
. . Q . . . . .
```

End Of Assignment