Artificial Intelligence

Practical-1: Write a program to implement Tic-Tac-Toe game problem.

- **Code in Prolog:**

```
play :-
my_turn([]).

               my_turn(Game) :-
                   valid_moves(ValidMoves, Game, x),
                   any_valid_moves(ValidMoves, Game).

               any_valid_moves([], _) :-
                   write('It is a tie'), nl.
               any_valid_moves([_|_], Game) :-
                   findall(NextMove, game_analysis(x, Game, NextMove), MyMoves),
                   do_a_decision(MyMoves, Game).

               % This can only fail in the beginning.
               do_a_decision(MyMoves, Game) :-
                   not(MyMoves = []),
                   length(MyMoves, MaxMove),
                   random(0, MaxMove, ChosenMove),
                   nth0(ChosenMove, MyMoves, X),
                   NextGame = [X | Game],
                   print_game(NextGame),
                   (victory_condition(x, NextGame) ->
                       (write('I won. You lose.'), nl);
                       your_turn(NextGame), !).

               your_turn(Game) :-
                   valid_moves(ValidMoves, Game, o),
                   (ValidMoves = [] -> (write('It is a tie'), nl);
                    (write('Available moves:'), write(ValidMoves), nl,
                     ask_move(Y, ValidMoves),
                     NextGame = [Y | Game],
                     (victory_condition(o, NextGame) ->
                       (write('I lose. You win.'), nl);
```

```
        my_turn(NextGame), !))).

ask_move(Move, ValidMoves) :-
    write('Give your move:'), nl,
    read(Move), member(Move, ValidMoves), !.

ask_move(Y, ValidMoves) :-
    write('not a move'), nl,
    ask_move(Y, ValidMoves).

movement_prompt(X, Y, ValidMoves) :-
    write('Give your X:'), nl, read(X), member(move(o, X, Y), ValidMoves),
!,
    write('Give your Y:'), nl, read(Y), member(move(o, X, Y), ValidMoves).

% A routine for printing games.. Well you can use it.
print_game(Game) :-
    plot_row(0, Game), plot_row(1, Game), plot_row(2, Game).

plot_row(Y, Game) :-
    plot(Game, 0, Y), plot(Game, 1, Y), plot(Game, 2, Y), nl.

plot(Game, X, Y) :-
    (member(move(P, X, Y), Game), ground(P)) -> write(P) ; write('.').

% This system determines whether there's a perfect play available.
game_analysis(_, Game, _) :-
    victory_condition(Winner, Game),
    Winner = x. % We do not want to lose.
    % Winner = o. % We do not want to win. (egostroking mode).
    % true. % If you remove this constraint entirely, it may let you win.
game_analysis(Turn, Game, NextMove) :-
    not(victory_condition(_, Game)),
    game_analysis_continue(Turn, Game, NextMove).

game_analysis_continue(Turn, Game, NextMove) :-
    valid_moves(Moves, Game, Turn),
    game_analysis_search(Moves, Turn, Game, NextMove).

% Comment these away and the system refuses to play,
% because there are no ways to play this without a possibility of tie.
game_analysis_search([], o, _, _). % Tie on opponent's turn.
game_analysis_search([], x, _, _). % Tie on our turn.
```

```
game_analysis_search([X|Z], o, Game, NextMove) :- % Whatever opponent does,
    NextGame = [X | Game],                          % we desire not to lose.
    game_analysis_search(Z, o, Game, NextMove),
    game_analysis(x, NextGame, _), !.

game_analysis_search(Moves, x, Game, NextMove) :-
    game_analysis_search_x(Moves, Game, NextMove).

game_analysis_search_x([X|_], Game, X) :-
    NextGame = [X | Game],
    game_analysis(o, NextGame, _).
game_analysis_search_x([_|Z], Game, NextMove) :-
    game_analysis_search_x(Z, Game, NextMove).

% This thing describes all kinds of valid games.
valid_game(Turn, Game, LastGame, Result) :-
    victory_condition(Winner, Game) ->
        (Game = LastGame, Result = win(Winner)) ;
        valid_continuing_game(Turn, Game, LastGame, Result).

valid_continuing_game(Turn, Game, LastGame, Result) :-
    valid_moves(Moves, Game, Turn),
    tie_or_next_game(Moves, Turn, Game, LastGame, Result).

tie_or_next_game([], _, Game, Game, tie).
tie_or_next_game(Moves, Turn, Game, LastGame, Result) :-
    valid_gameplay_move(Moves, NextGame, Game),
    opponent(Turn, NextTurn),
    valid_game(NextTurn, NextGame, LastGame, Result).

% Victory conditions for tic tac toe.
victory(P, Game, Begin) :-
    valid_gameplay(Game, Begin),
    victory_condition(P, Game).

victory_condition(P, Game) :-
    (X = 0; X = 1; X = 2),
    member(move(P, X, 0), Game),
    member(move(P, X, 1), Game),
    member(move(P, X, 2), Game).

victory_condition(P, Game) :-
```

```
    (Y = 0; Y = 1; Y = 2),
    member(move(P, 0, Y), Game),
    member(move(P, 1, Y), Game),
    member(move(P, 2, Y), Game).

victory_condition(P, Game) :-
    member(move(P, 0, 2), Game),
    member(move(P, 1, 1), Game),
    member(move(P, 2, 0), Game).

victory_condition(P, Game) :-
    member(move(P, 0, 0), Game),
    member(move(P, 1, 1), Game),
    member(move(P, 2, 2), Game).

% This describes a valid form of gameplay.
% Which player did the move is disregarded.
valid_gameplay(Start, Start).

valid_gameplay(Game, Start) :-
    valid_gameplay(PreviousGame, Start),
    valid_moves(Moves, PreviousGame, _),
    valid_gameplay_move(Moves, Game, PreviousGame).

valid_gameplay_move([X|_], [X|PreviousGame], PreviousGame).
valid_gameplay_move([_|Z], Game, PreviousGame) :-
    valid_gameplay_move(Z, Game, PreviousGame).

% The set of valid moves must not be affected by the decision making
% of the prolog interpreter.
% Therefore we have to retrieve them like this.
% This is equivalent to the (∀x∈0..2)(∀y∈0..2)(....
% uh wait.. There's no way to represent this using those quantifiers.
valid_moves(Moves, Game, Turn) :-
    valid_moves_column(0, M1,    [], Game, Turn),
    valid_moves_column(1, M2,    M1, Game, Turn),
    valid_moves_column(2, Moves, M2, Game, Turn).

valid_moves_column(X, M3, M0, Game, Turn) :-
    valid_moves_cell(X, 0, M1, M0, Game, Turn),
    valid_moves_cell(X, 1, M2, M1, Game, Turn),
    valid_moves_cell(X, 2, M3, M2, Game, Turn).
```

```
valid_moves_cell(X, Y, M1, M0, Game, Turn) :-
    member(move(_, X, Y), Game) -> M0 = M1 ; M1 = [move(Turn,X,Y) | M0].

% valid_move(X, Y, Game) :-
%      (X = 0; X = 1; X = 2),
%      (Y = 0; Y = 1; Y = 2),
%      not(member(move(_, X, Y), Game)).

opponent(x, o).
opponent(o, x).
```

Output:

```
% c:/Users/Arjun Vankani/Desktop/prolog-tic-tac-toe-master/tictactoe.pro compiled 0.00 sec, 0 clauses
?-  play.
...
..x
...
Available moves:[move(o,2,2),move(o,2,0),move(o,1,2),move(o,1,1),move(o,1,0),move(o,0,2),move(o,0,1),move(o,0,0)]
Give your move:
|: move(o, 1,1).
...
.ox
..x
Available moves:[move(o,2,0),move(o,1,2),move(o,1,0),move(o,0,2),move(o,0,1),move(o,0,0)]
Give your move:
|: move(o, 0, 0).
ox.
.ox
..x
Available moves:[move(o,2,0),move(o,1,2),move(o,0,2),move(o,0,1)]
Give your move:
|: move(o, 0, 1).
ox.
oox
x.x
Available moves:[move(o,2,0),move(o,1,2)]
Give your move:
|: move(o, 2, 0).
oxo
oox
xxx
I won. You lose.
true.

?-
```

CODE in Python:

```
import pygame

from pygame.locals import *


Board_width = 3  # number of columns in the board

Board_height = 3 # number of rows in the board

Tile_size = 100

Window_width = 480

Window_height = 480
```

```
FPS = 30  # Frames per second
Blank = None



#               R    G    B
Black =        (  0,   0,   0)
White =        (255, 255, 255)
Green =        (  0,  204,  0)
Dark_turquoise = (  3,  54,  73)
Magenta =      (  255, 0, 255)

Background_color = Dark_turquoise
Tile_color = Magenta
Text_color = White
Border_color = Green
Font_size = 20

Button_color = White
Button_text_color = Black
Message_color = White



Blank = 10
Player_O = 11
Player_X = 21



Player_O_win = Player_O * 3
Player_X_win = Player_X * 3
```

```
Continue_Game  = 10
Draw_Game      = 20
Quit_Game      = 30

X_margin = int((Window_width - (Tile_size * Board_width + (Board_width - 1))) / 2)
Y_margin = int((Window_height - (Tile_size * Board_height + (Board_height - 1))) / 2)

choice = 0

def Check_Winner(board):
    def Check_Draw():
        return sum(board)%10 == 9

    def check_horizontal(player):   # Horizontal Win
        for i in [0, 3, 6]:
            if sum(board[i:i+3]) == 3 * player:
                return player

    def check_vertical(player):   # Vertical Win
        for i in range(3):
            if sum(board[i::3]) == 3 * player:
                return player

    def check_diagonals(player):   # Main Diagonal Win
        if (sum(board[0::4]) == 3 * player) or (sum(board[2:7:2]) == 3 * player):
            return player

    for player in [Player_X, Player_O]:
```

```
        if any([check_horizontal(player), check_vertical(player), check_diagonals(player)]):
            return player

    return Draw_Game if Check_Draw() else Continue_Game



def unit_score(winner, depth):
    if winner == Draw_Game:
        return 0
    else:
        return 10 - depth if winner == Player_X else depth - 10




def get_available_move(board):
    return [i for i in range(9) if board[i] == Blank]




def minimax(board, depth):
    global choice
    result = Check_Winner(board)
    if result != Continue_Game:
        return unit_score(result, depth)

    depth += 1  # index of the node in the game tree
    scores = []   # an array of scores
    steps = []   # an array of moves(steps)
```

```
    for step in get_available_move(board):
        score = minimax(update_state(board, step, depth), depth)
        scores.append(score)
        steps.append(step)

    if depth % 2 == 1:
        max_value_index = scores.index(max(scores))
        choice = steps[max_value_index]
        return max(scores)
    else:
        min_value_index = scores.index(min(scores))
        choice = steps[min_value_index]
        return min(scores)


def update_state(board, step, depth):
    board = list(board)
    board[step] = Player_X if depth % 2 else Player_O
    return board

def update_board(board, step, player):
    board[step] = player


def change_to_player(player):
    if player == Player_O:
        return 'O'
    elif player == Player_X:
        return 'X'
```

```
    elif player == Blank:
        return '-'


def Draw_Board(board, message):
    displaySurf.fill(Background_color)
    if message:
        textSurf, textRect = makeText(message, Message_color, Background_color, 5, 5)
        displaySurf.blit(textSurf, textRect)


    for tile_x in range(3):
        for tile_y in range(3):
            if board[tile_x*3+tile_y] != Blank:
                drawTile(tile_x, tile_y, board[tile_x*3+tile_y])


    left, top = get_Left_Top_Of_Tile(0, 0)
    width = Board_width * Tile_size
    height = Board_height * Tile_size
    pygame.draw.rect(displaySurf, Border_color, (left - 5, top - 5, width + 11, height + 11), 4)


    displaySurf.blit(New_surf, New_rect)
    displaySurf.blit(New_surf2, New_rect2)



def get_Left_Top_Of_Tile(tile_X, tile_Y):
    left = X_margin + (tile_X * Tile_size) + (tile_X - 1)
    top = Y_margin + (tile_Y * Tile_size) + (tile_Y - 1)
    return (left, top)
```

```
def makeText(text, color, bgcolor, top, left):
    textSurf = Basic_font.render(text, True, color, bgcolor)
    textRect = textSurf.get_rect()
    textRect.topleft = (top, left)
    return (textSurf, textRect)


def drawTile(tile_x, tile_y, symbol, adj_x=0, adj_y=0):
    left, top = get_Left_Top_Of_Tile(tile_x, tile_y)
    pygame.draw.rect(displaySurf, Tile_color, (left + adj_x, top + adj_y, Tile_size, Tile_size))
    textSurf = Basic_font.render(symbol_to_str(symbol), True, Text_color)
    textRect = textSurf.get_rect()
    textRect.center = left + int(Tile_size / 2) + adj_x, top + int(Tile_size / 2) + adj_y
    displaySurf.blit(textSurf, textRect)


def symbol_to_str(symbol):
    if symbol == Player_O:
        return 'O'
    elif symbol == Player_X:
        return 'X'


def get_spot_clicked(x, y):
    for tile_X in range(3):
        for tile_Y in range(3):
            left, top = get_Left_Top_Of_Tile(tile_X, tile_Y)
            tileRect = pygame.Rect(left, top, Tile_size, Tile_size)
```

```
            if tileRect.collidepoint(x, y):
                return (tile_X, tile_Y)
    return None



def board_to_step(spot_x, spot_y):
    return spot_x * 3 + spot_y


def check_valid_move(coords, board):
    step = board_to_step(*coords)
    return board[step] == Blank



def main():
    global FPS_clock, displaySurf, Basic_font, New_surf, New_rect, New_surf2, New_rect2
    two_player = False #by default false
    pygame.init()
    FPS_clock = pygame.time.Clock()
    displaySurf = pygame.display.set_mode((Window_width, Window_height))
    pygame.display.set_caption('Tic Tac Toe')
    Basic_font = pygame.font.Font('freesansbold.ttf', Font_size)
    New_surf, New_rect = makeText('vs AI', Text_color, Tile_color, Window_width - 120, Window_height - 60)
    New_surf2, New_rect2 = makeText('vs Human', Text_color, Tile_color, Window_width - 240, Window_height - 60)
    board = [Blank] * 9
    game_over = False
    x_turn = True
    msg = "Welcome to this game"   # Contains the message to show in the upper left corner.
```

```
    Draw_Board(board, msg)
    pygame.display.update()   # pygame.display.update() is called to draw the display Surface object on the actual computer screen

    while True:
        coords = None
        for event in pygame.event.get():   # event handling loop
            if event.type == MOUSEBUTTONUP:   # If the type of event was a MOUSEBUTTONUP event (that is, the player had released a mouse button somewhere over the window), then we pass the mouse coordinates to our getSpotClicked() function which will return the board coordinates of the spot on the board the mouse release happened. The event.pos[0] is the X coordinate and event.pos[1] is the Y coordinate.
                coords = get_spot_clicked(event.pos[0], event.pos[1])
                if not coords and New_rect.collidepoint(event.pos):
                    board = [Blank] * 9
                    game_over = False
                    msg = "Welcome to this game"
                    Draw_Board(board, msg)
                    pygame.display.update()
                    two_player = False
                if not coords and New_rect2.collidepoint(event.pos):
                    board = [Blank] * 9
                    game_over = False
                    msg = "Welcome to this game"
                    Draw_Board(board, msg)
                    pygame.display.update()
                    two_player = True
        if coords and check_valid_move(coords, board) and not game_over:
            if two_player:
                next_step = board_to_step(*coords)
```

```
            if x_turn:
                update_board(board, next_step, Player_X)
                x_turn = False
            else:
                update_board(board, next_step, Player_O)
                x_turn = True
            Draw_Board(board, msg)
            pygame.display.update()
        if not two_player:
            next_step = board_to_step(*coords)
            update_board(board, next_step, Player_X)
            Draw_Board(board, msg)
            pygame.display.update()
            minimax(board, 0)
            update_board(board, choice, Player_O)
        result = Check_Winner(board)
        game_over = (result != Continue_Game)
        if result == Player_X:
            msg = "The winner of this game is X"
        elif result == Player_O:
            msg = "The winner of this game is O"
        elif result == Draw_Game:
            msg = "Draw Game"
        Draw_Board(board, msg)
        pygame.display.update()


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

Output: