TIC TAC TOE AI MODEL

Using reinforcement learning

NAME: SRI SAI VIJAYA ADITYA NITTALA

ROLL NO.: 177163

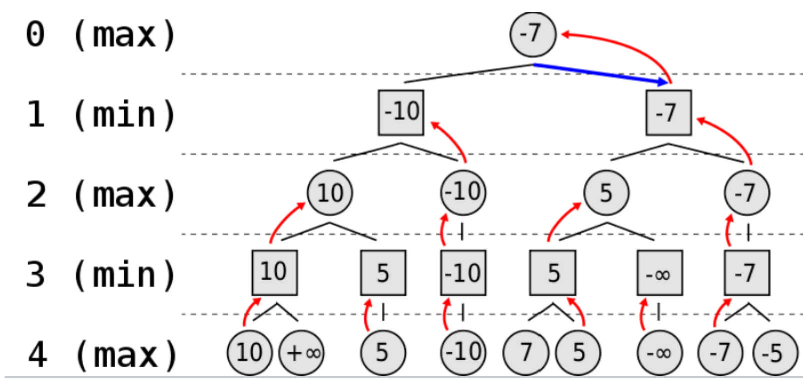
Section: A

The **minimax** algorithm is a recursive algorithm for choosing the next move in an n-player game (usually a 2-player game). A value is associated with each position or state of the game. This value is computed by means of an evaluation function that indicates how good it would be for a player to reach that position.

The pseudocode for the algorithm is given below:

```
function minimax(node, depth, maximizingPlayer) is
    if depth = 0 or node is a terminal node then
        return the heuristic value of node
    if maximizingPlayer then
        value := −∞
        for each child of node do
            value := max(value, minimax(child, depth − 1, FALSE))
        return value
    else (* minimizing player *)
        value := +∞
        for each child of node do
            value := min(value, minimax(child, depth − 1, TRUE))
        return value
```

With the image below, the algorithm is explained:



- Circles are for the maximizing player, that is the player running the algorithm.
- The squares are for the minimizing player.
- At each step, the maximizing player wants to maximize his/her winning and the minimizing player wants to minimize.
- Upon reaching the greatest depth, that is, final board state, the evaluation function is called to get a score on the board state. Depending on whose turn it was, the minimum or maximum value of two child nodes are returned.
- This continues until a value is returned to the current board state, which then helps the computer pick a move.

minimax() set-up for Tic Tac Toe game:

- Depth for a nxn tic tac toe board is when all nxn cells are evaluated.
- The terminal board state is when all cells of the board are filled.
- The **Human** is given a value of -1.
- The **Computer** is given a value of +1.
- Evaluation:
  - If **Computer** wins: +1
  - If **Human** wins: -1
  - If **draw**: 0

The pseudocode is given below:

```
function minimax(board, depth, isMaximizingPlayer):

    if current board state is a terminal state :
        return value of the board

    if isMaximizingPlayer :
        bestVal = -INFINITY
        for each move in board :
            value = minimax(board, depth+1, false)
            bestVal = max( bestVal, value)
        return bestVal

    else :
        bestVal = +INFINITY
        for each move in board :
            value = minimax(board, depth+1, true)
            bestVal = min( bestVal, value)
        return bestVal
```

NOTE:

The code attached below is for a **3x3** tic tac toe board but the same functions with a few modifications (mentioned as comments in the code) will work with a **4x4** board. Due to insufficient computing power, a game of 4x4 tic tac toe with minimax() on my computer took about 20 minutes.

```
from math import inf as infinity
from random import choice
import platform
import time
import numpy as np
```

Declaring the board below. Please uncomment 4x4 board to test for that size.
The functions written below work for nxn matrices but a lot of computation is involved.

```
COMP = 1
HUMAN = -1
'''BOARD = [
         [0, 0, 0, 0],
         [0, 0, 0, 0],
         [0, 0, 0, 0],
         [0, 0, 0, 0],
]'''
BOARD = [
         [0, 0, 0],
         [0, 0, 0],
         [0, 0, 0],
]


def isWin(state, player):
  '''
        Checks if either of the players have won the game
        to test for 4x4 uncomment the below list and comment the 3x3 list out
  '''
  '''win_states = [
               [state[0][0], state[0][1], state[0][2], state[0][3]],
               [state[1][0], state[1][1], state[1][2], state[1][3]],
               [state[2][0], state[2][1], state[2][2], state[2][3]],
               [state[3][0], state[3][1], state[3][2], state[3][3]],
               [state[0][0], state[1][0], state[2][0], state[3][0]],
               [state[0][1], state[1][1], state[2][1], state[3][1]],
               [state[0][2], state[1][2], state[2][2], state[3][2]],
               [state[0][3], state[1][3], state[2][3], state[3][3]],
               [state[0][0], state[1][1], state[2][2], state[3][3]],
               [state[0][3], state[1][2], state[2][1], state[3][0]],
  ]'''
  win_states = [
               [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[0][0], state[1][1], state[2][2]],
                [state[2][0], state[1][1], state[0][2]],
  ]

  win_array = []
  for i in range(len(BOARD)):
    win_array.append(player)

  if win_array in win_states:
    return True
  else:
    return False


def evaluate(state):
  if isWin(state, COMP):
    score = 1
  elif isWin(state, HUMAN):
    score = -1
  else:
    score = 0
  return score


def isGameOver(state):
  return isWin(state, COMP) or isWin(state, HUMAN)


def getEmptyCells(state):
  cells = []

  for i in range(len(state)):
    for j in range(len(state[i])):
      if state[i][j] == 0:
        cells.append([i, j])

  return cells


def isValidMove(x, y):
  if [x, y] in getEmptyCells(BOARD):
    return True
  else:
    return False


def setMove(x, y, player):
  if isValidMove(x, y):
    BOARD[x][y] = player
    return True
  else:
    return False
```

```
def minimax(state, depth, player):
  '''
        Minimax is a Game Theory based algorithm that is frequently used in Reinforcement Lea
        The computer considers all possible moves before a move is played. This ensures that
        optimal move is made always
  '''
  if player == COMP:
    best = [-1, -1, -infinity]
  else:
    best = [-1, -1, +infinity]

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

  for cell in getEmptyCells(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
    else:
      if score[2] < best[2]:
        best = score

  return best


def printBoard(c, h):
  piece = {
      1: c,
      -1: h,
      0: ' ',
  }

  print("-----"*len(BOARD))
  for i in range(len(BOARD)):
    for j in range(len(BOARD[i])):
      print("| {} |".format(piece[BOARD[i][j]]), end='')
    print()
    print("-----"*len(BOARD))


def ComputerTurn(c, h):
  print("Computer's turn : ")
  depth = len(getEmptyCells(BOARD))
  if depth == 0 or isGameOver(BOARD):
    return
```

```
  if depth == len(BOARD)*len(BOARD):
    x = np.random.randint(len(BOARD))
    y = np.random.randint(len(BOARD))
  else:
    move = minimax(BOARD, depth, COMP)
    x, y = move[0], move[1]

  setMove(x, y, COMP)
  printBoard(c, h)


def HumanTurn(c, h):
  print("Human's turn : ")
  depth = len(getEmptyCells(BOARD))
  if depth == 0 or isGameOver(BOARD):
    return

  emptyCells = getEmptyCells(BOARD)
  x = 0
  y = 0
  ''' print board here '''
  while True:
    x = int(input("Enter x coordinate : "))
    y = int(input("Enter y coordinate : "))

    if [x, y] in emptyCells:
      break

  setMove(x, y, HUMAN)
  printBoard(c, h)


def playGame(h_choice, c_choice, first):
  while len(getEmptyCells(BOARD)) > 0 and not isGameOver(BOARD):
    if first == 'N':
      ComputerTurn(c_choice, h_choice)
      first = ''

    HumanTurn(c_choice, h_choice)
    ComputerTurn(c_choice, h_choice)

  if isWin(BOARD, COMP):
    print("Computer Wins!")
  elif isWin(BOARD, HUMAN):
    print("Human Wins!")
  else:
    print("Draw!")


h_choice = input("HUMAN: What would you like to be? {X or O} : ").upper()

if h_choice == 'X':
```

```
    c_choice = 'O'
  else:
    c_choice = 'X'

  first = input("HUMAN: Would you like to go first? [Y/N2]: ").upper()
  playGame(h_choice, c_choice, first)
```

```
HUMAN: What would you like to be? {X or O} : x
HUMAN: Would you like to go first? [Y/N2]: n
Computer's turn :
---------------
| O ||   ||   |
---------------
|   ||   ||   |
---------------
|   ||   ||   |
---------------
Human's turn :
Enter x coordinate : 2
Enter y coordinate : 2
---------------
| O ||   ||   |
---------------
|   ||   ||   |
---------------
|   ||   || X |
---------------
Computer's turn :
---------------
| O ||   || O |
---------------
|   ||   ||   |
---------------
|   ||   || X |
---------------
Human's turn :
Enter x coordinate : 1
Enter y coordinate : 1
---------------
| O ||   || O |
---------------
|   || X ||   |
---------------
|   ||   || X |
---------------
Computer's turn :
---------------
| O || O || O |
---------------
|   || X ||   |
---------------
|   ||   || X |
---------------
Computer Wins!
```