INDIAN INSTITUTE OF INFORMATION TECHNOLOGY

ARTIFICIAL INTELLIGENCE PROJECT

Submitted by:

18BCS009 – ANUJ C S

18BCS026 – G K BHARATH BHUSHAN

18BCS102 – SUSHANTH B PATIL

18BCS108 – VARUN MAHESH AWATI

Submitted to:

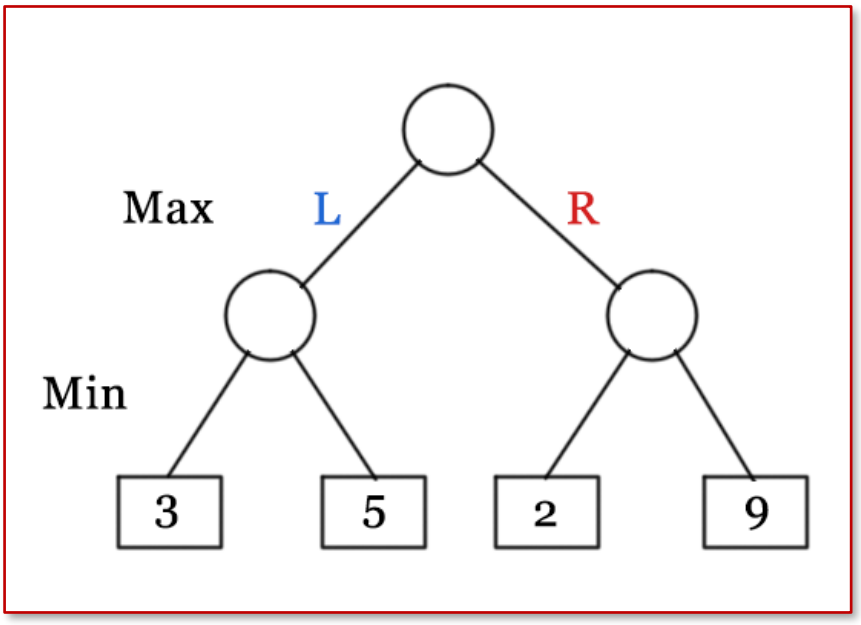
B JAYALAKSHMI

ABSTRACT

Tic-Tac-Toe is a two-player game. It involves two players (one player vs AI) placing their respective symbols in a 3X3 grid. The player who manages to place three of their symbols in horizontal/vertical/diagonal row wins the game. If either player fails to do so the game ends in a draw. If both the people always play their optimal strategies the game always ends in a draw. It is done using the Minimax Algorithm.

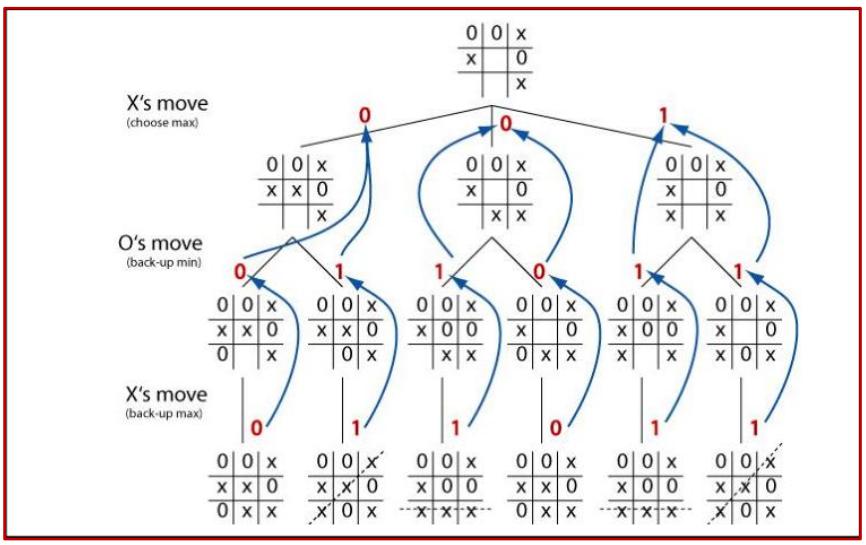
Minimax Algorithm is a decision rule formulated for 2 player zero-sum games (Tic-Tac-Toe, Chess, Go, etc.). This algorithm sees a few steps ahead and puts itself in the shoes of its opponent. It keeps playing and exploring subsequent possible states until it reaches a terminal state resulting in a draw, a win, or a loss. Being in any of these possible terminal states has some utility for the AI — such as being in a 'Win' state is good (utility is positive), being in a 'Loss' state

is bad (utility is negative) and being in a draw in neither good nor bad (utility is neutral).



In our execution of the Minimax algorithm for solving Tic-Tac-Toe, it works by visualizing all future possible states of the board and constructs it in the form of a tree. When the current board state is given to the algorithm (the root of the tree), it splits into 'n' branches (where n denotes the number of moves that can be chosen by the AI/number of empty cells where the AI can be placed). If any of these new states is a terminal state, no further splits are performed for this state and it is assigned a score the following way:

- score = +1 (if AI wins)
- score = -1 (if AI loses)
- score= 0 (If a draw happens)



If, however, there are no terminal states — each of these new states is considered as new root and they give rise to a tree of their own. But there's a catch — since this is a 2-player game and the players take turns alternatively, therefore whenever we go one layer deeper in the network, we need to change the player which would be placed in one of the empty cells. This way we can visualize what moves the other player would take as a result of our move. The algorithm evaluates the best move to take by choosing the move which has the maximum score when it is the AI's turn and choosing the minimum score when it is the Human's turn.

WORKING CODE:

```
from copy import deepcopy:



class TicTacToe:
    def __init__(self, other=None):
        self.board = [["   ", "   ", "   "] for i in range(3)]
        self.player = "X"
        self.opponent = "O"
        if other:
            self.__dict__ = deepcopy(other.__dict__)


    def move(self, inp):
        b = TicTacToe(self)
        b.board[inp // 3][inp % 3] = " " + b.player + " "
        (b.player, b.opponent) = (b.opponent, b.player)
        return b


    def tie(self):
        t = True
```

```
    for i in self.board:
        for k in i:
            if k == "   ":
                t = False
    return t

def check_game(self):
    if self.tie():
        return "TIE!"
    for i in range(3):
        if self.board[i][0] != "   " and \
                self.board[i][0] == self.board[i][1] and self.board[i][1] == self.board[i][2]:
            return f"{self.board[i][0][1:]}has won!"

        if self.board[0][i] != "   " and \
                self.board[0][i] == self.board[1][i] and self.board[1][i] == self.board[2][i]:
            return f"{self.board[0][i][1:]}has won!"

    if self.board[0][0] != "   " and \
            self.board[0][0] == self.board[1][1] and self.board[1][1] == self.board[2][2]:
        return f"{self.board[0][0][1:]}has won!"

    if self.board[0][2] != "   " and \
            self.board[0][2] == self.board[1][1] and self.board[1][1] == self.board[2][0]:
        return f"{self.board[0][2][1:]}has won!"

    return False

def won(self):
```

```
    for i in range(3):
        if self.board[i][0] == f" {self.opponent}" and \
                self.board[i][0] == self.board[i][1] and self.board[i][1] == self.board[i][2]:
            return True

        if self.board[0][i] == f" {self.opponent}" and \
                self.board[0][i] == self.board[1][i] and self.board[1][i] == self.board[2][i]:
            return True

    if self.board[0][0] == f" {self.opponent}" and \
            self.board[0][0] == self.board[1][1] and self.board[1][1] == self.board[2][2]:
        return True

    if self.board[0][2] == f" {self.opponent}" and \
            self.board[0][2] == self.board[1][1] and self.board[1][1] == self.board[2][0]:
        return True

    return False

def __str__(self):
    return "".join([x for i in self.board for x in ["".join([z for k in i for z in [k, "|"]][:-1]) +
                                                    "\n", "-----------" + "\n"]][:-1])

def best(self):
    return self._minimax(True)[1]

def _minimax(self, player):
    if self.won():
        if player:
```

```
            return (-1, None)
        else:
            return (1, None)

    elif self.tie():
        return (0, None)

    elif player:
        best = (-2, None)
        for i in range(9):
            if self.board[i // 3][i % 3] == "  ":
                value = self.move(i)._minimax(not(player))[0]
                if value > best[0]:
                    best = (value, i)
        return best

    else:
        best = (2, None)
        for i in range(9):
            if self.board[i // 3][i % 3] == "  ":
                value = self.move(i)._minimax(not(player))[0]
                if value < best[0]:
                    best = (value, i)
        return best


if __name__ == "__main__":
    t = TicTacToe()
    print(t)
```

```
while True:
    t = t.move(int(input(":")) - 1)
    m = t.best()
    t = t.move(m)
    print(t)
    # print(t.won())
```

SCREENSHOTS OF OUTPUT: