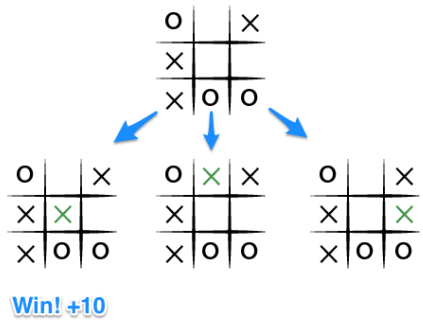
# EX-06 Implement Minimax Search Algorithm for a Simple TIC-TAC-TOE game

## Aim:

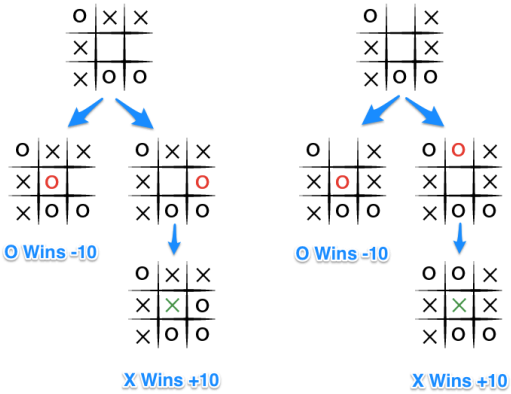
Implement Minimax Search Algorithm for a Simple TIC-TAC-TOE game.

## Theory and Procedure:

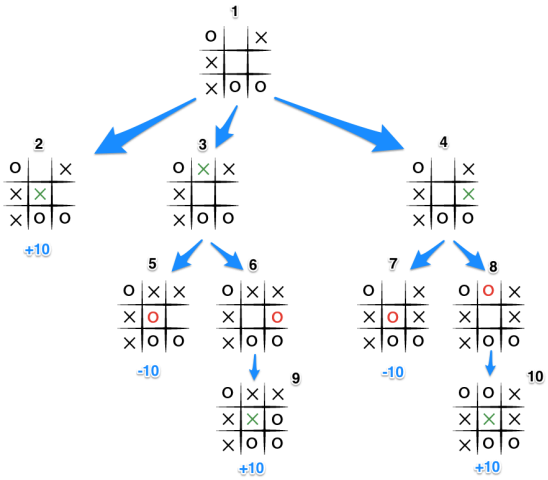
To begin, let's start by defining what it means to play a perfect game of tic tac toe: If I play perfectly, every time I play I will either win the game, or I will draw the game. Furthermore if I play against another perfect player, I will always draw the game. How might we describe these situations quantitatively? Let's assign a score to the "end game conditions:" I win, hurray! I get 10 points! I lose, shit. I lose 10 points (because the other player gets 10 points) I draw, whatever. I get zero points, nobody gets any points. So now we have a situation where we can determine a possible score for any game end state. Looking at a Brief Example To apply this, let's take an example from near the end of a game, where it is my turn. I am X. My goal here, obviously, is to maximize my end game score.




If the top of this image represents the state of the game I see when it is my turn, then I have some choices to make, there are three places I can play, one of which clearly results in me wining and earning the 10 points. If I don't make that move, O could very easily win. And I don't want O to win, so my goal here, as the first player, should be to pick the maximum scoring move. But What About O? What do we know about O? Well we should assume that O is also playing to win this game, but relative to us, the first player, O wants obviously wants to chose the move that results in the worst score for us, it wants to pick a move that would minimize our ultimate score. Let's look at things from O's perspective, starting with the two other game states from above in which we don't immediately win:



The choice is clear, O would pick any of the moves that result in a score of -10. Describing Minimax The key to the Minimax algorithm is a back and forth between the two players, where the player whose "turn it is" desires to pick the move with the maximum score. In turn, the scores for each of the available moves are determined by the opposing player deciding which of its available moves has the minimum score. And the scores for the opposing players moves are again determined by the turn-taking player trying to maximize its score and so on all the way down the move tree to an end state. A description for the algorithm, assuming X is the "turn taking player," would look something like:




If the game is over, return the score from X's perspective. Otherwise get a list of new game states for every possible move Create a scores list For each of these states add the minimax result of that state to the scores list If it's X's turn, return the maximum score from the scores list If it's O's turn, return the minimum score from the scores list You'll notice that this algorithm is recursive, it flips back and forth between the players until a final score is found. Let's walk through the algorithm's execution with the full move tree, and show why, algorithmically, the instant winning move will be picked:

- It's X's turn in state 1. X generates the states 2, 3, and 4 and calls minimax on those states.
- State 2 pushes the score of +10 to state 1's score list, because the game is in an end state.
- State 3 and 4 are not in end states, so 3 generates states 5 and 6 and calls minimax on them, while state 4 generates states 7 and 8 and calls minimax on them.
- State 5 pushes a score of -10 onto state 3's score list, while the same happens for state 7 which pushes a score of -10 onto state 4's score list.
- State 6 and 8 generate the only available moves, which are end states, and so both of them add the score of +10 to the move lists of states 3 and 4.
- Because it is O's turn in both state 3 and 4, O will seek to find the minimum score, and given the choice between -10 and +10, both states 3 and 4 will yield -10.
- >Finally the score list for states 2, 3, and 4 are populated with +10, -10 and -10 respectively, and state 1 seeking to maximize the score will chose the winning move with score +10, state 2. ##A Coded Version of Minimax Hopefully by now you have a rough sense of how th e minimax algorithm determines the best move to play. Let's examine my implementation of the algorithm to solidify the understanding: Here is the function for scoring the game:

```
# @player is the turn taking player
def score(game)
    if game.win?(@player)
        return 10
    elsif game.win?(@opponent)
        return -10
    else
        return 0
    end
end
```

Simple enough, return +10 if the current player wins the game, -10 if the other player wins and 0 for a draw. You will note that who the player is doesn't matter. X or O is irrelevant, only who's turn it happens to be. And now the actual minimax algorithm; note that in this implementation a choice or move is simply a row / column address on the board, for example [0,2] is the top right square on a 3x3 board.

```
def minimax(game)
    return score(game) if game.over?
    scores = [] # an array of scores
    moves = []  # an array of moves
    # Populate the scores array, recursing as needed
    game.get_available_moves.each do |move|
        possible_game = game.get_new_state(move)
        scores.push minimax(possible_game)
        moves.push move
    end
    # Do the min or the max calculation
    if game.active_turn == @player
        # This is the max calculation
        max_score_index = scores.each_with_index.max[1]
        @choice = moves[max_score_index]
        return scores[max_score_index]
    else
        # This is the min calculation
        min_score_index = scores.each_with_index.min[1]
        @choice = moves[min_score_index]
        return scores[min_score_index]
    end
end
```

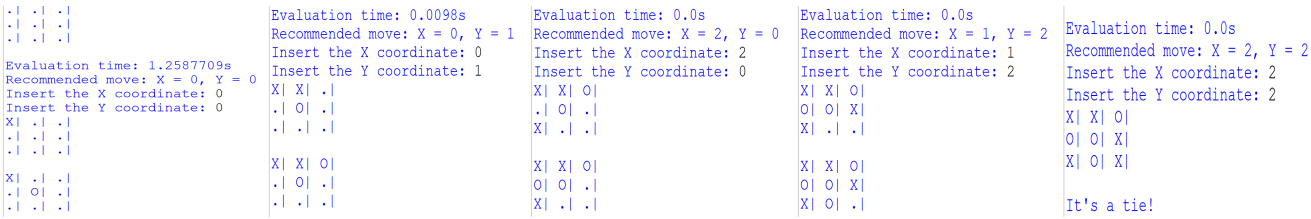
## Program:

```
import time                # Developed By: Ragul A C
class Game:                # Register No : 212221240042
    def __init__(self):
        self.initialize_game()
    def initialize_game(self):
        self.current_state = [['.','.','.'],['.','.','.'],['.','.','.']]
        self.player_turn = 'X' # Player X always plays first
    def draw_board(self):
        for i in range(0, 3):
            for j in range(0, 3):
                print('{}|'.format(self.current_state[i][j]), end=" ")
            print()
        print()
    def is_valid(self, px, py):
        if px < 0 or px > 2 or py < 0 or py > 2:
            return False
        elif self.current_state[px][py] != '.':
            return False
        else:
            return True
    def is_end(self):
        for i in range(0, 3): # Vertical win
            if (self.current_state[0][i] != '.' and
                self.current_state[0][i] == self.current_state[1][i] and
                self.current_state[1][i] == self.current_state[2][i]):
                return self.current_state[0][i]
        for i in range(0, 3): # Horizontal win
            if (self.current_state[i] == ['X', 'X', 'X']):
                return 'X'
            elif (self.current_state[i] == ['O', 'O', 'O']):
                return 'O'
        if (self.current_state[0][0] != '.' and # Main diagonal win
            self.current_state[0][0] == self.current_state[1][1] and
            self.current_state[0][0] == self.current_state[2][2]):
            return self.current_state[0][0]
        if (self.current_state[0][2] != '.' and # Second diagonal win
            self.current_state[0][2] == self.current_state[1][1] and
            self.current_state[0][2] == self.current_state[2][0]):
            return self.current_state[0][2]
        for i in range(0, 3): # Is the whole board full?
            for j in range(0, 3): # There's an empty field, we continue the game
                if (self.current_state[i][j] == '.'):
                    return None
        return '.' # It's a tie!
    def max(self): # Possible values for maxv are: -1-loss,0-tie,1-win
        maxv = -2 # We're initially setting it to -2 as worse than the worst case:
        px,py = None,None
        result = self.is_end() # If the game came to an end,the function needs to
        # the evaluation function of the end. That can be:  -1-loss,0-tie,1-win
        if result == 'X':
            return (-1, 0, 0)
        elif result == 'O':
```

```
            return (1, 0, 0)
        elif result == '.':
            return (0, 0, 0)
        for i in range(0, 3):
            for j in range(0, 3):
                if self.current_state[i][j]=='.': # On the empty field player 'O'
                    self.current_state[i][j] = 'O' # That's one branch of the game
                    (m, min_i, min_j) = self.min() # Fixing the maxv value if need
                    if m > maxv:
                        px,py,maxv = i,j,m # Setting back the field to empty
                    self.current_state[i][j] = '.'
        return (maxv, px, py)
    def min(self): # Possible values for minv are: -1-win,0-tie,1-loss
        minv = 2 # We're initially setting it to 2 as worse than the worst case:
        qx,qy = None,None
        result = self.is_end()
        if result == 'X':
            return (-1, 0, 0)
        elif result == 'O':
            return (1, 0, 0)
        elif result == '.':
            return (0, 0, 0)
        for i in range(0, 3):
            for j in range(0, 3):
                if self.current_state[i][j] == '.':
                    self.current_state[i][j] = 'X'
                    (m, max_i, max_j) = self.max()
                    if m < minv:
                        qx,qy,minv = i,j,m
                    self.current_state[i][j] = '.'
        return (minv, qx, qy)
    def play(self):
        while True:
            self.draw_board()
            self.result=self.is_end() #Printing the appropriate message when game
            if self.result != None:
                if self.result == 'X':
                    print('The winner is X!')
                elif self.result == 'O':
                    print('The winner is O!')
                elif self.result == '.':
                    print("It's a tie!")
                self.initialize_game()
                return
            if self.player_turn == 'X': # If it's player's turn
                while True:
                    start = time.time()
                    (m, qx, qy) = self.min()
                    end = time.time()
                    print('Evaluation time: {}s'.format(round(end - start, 7)))
                    print('Recommended move: X = {}, Y = {}'.format(qx, qy))
                    px = int(input('Insert the X coordinate: '))
```

```
                    py = int(input('Insert the Y coordinate: '))
                    (qx, qy) = (px, py)
                    if self.is_valid(px, py):
                        self.current_state[px][py] = 'X'
                        self.player_turn = 'O'
                        break
                    else:
                        print('The move is not valid! Try again.')
            else: # If it's AI's turn
                (m,px,py) = self.max()
                self.current_state[px][py] = 'O'
                self.player_turn = 'X'
def main():
    g = Game()
    g.play()
if __name__ == "__main__":
    main()
```

## Sample Input and Output:




## Result:

Thus,Implementation of Minimax Search Algorithm for a Simple TIC-TAC-TOE game wasa done successfully.