CSE 401: Artificial Intelligence
Tic-Tac-Toe Game using MiniMax Algorithm

Utkarsh Gupta
A2305217557
7CSE 8Y

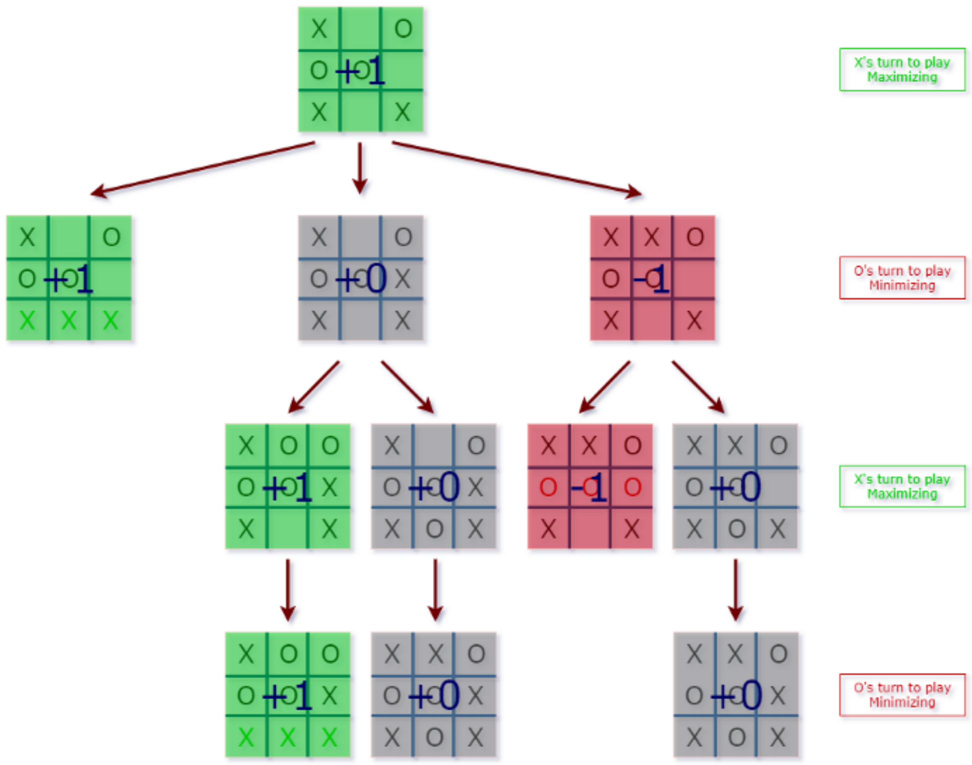
August 25th, 2020

1 What is MiniMax?

Minimax is a artifical intelligence applied in two player games, such as tic-tac-toe, checkers, chess and go. This games are known as zero-sum games, because in a mathematical representation: one player wins (+1) and other player loses (-1) or both of anyone not to win (0).

Minimax is a type of adversarial search algorithm for generating and exploring game trees. It is mostly used to solve zero-sum games where one side's gain is equivalent to other side's loss, so adding all gains and subtracting all losses end up being zero.

Adversarial search differs from conventional searching algorithms by adding opponents into the mix. Minimax algorithm keeps playing the turns of both player and the opponent optimally to figure out the best possible move.


X's turn to play Maximizing
O's turn to play Minimizing
X's turn to play Maximizing
O's turn to play Minimizing
+1
+1
+0
-1
+1
+0
-1
+0
+1
+0
+0

2 How does it work?

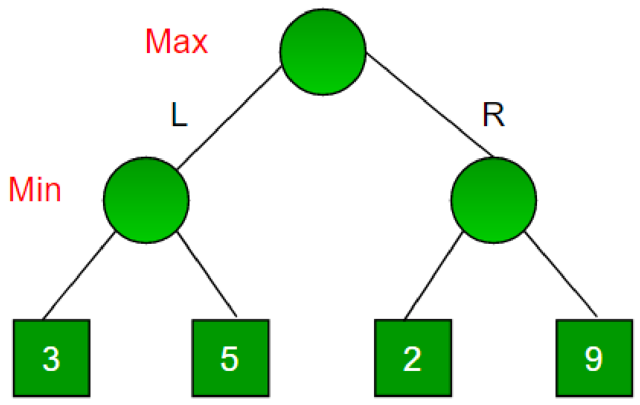
The algorithm search, recursively, the best move that leads the Max player to win or not lose (draw). It consider the current state of the game and the available moves at that state, then for each valid move it plays (alternating min and max) until it finds a terminal state (win, draw or lose).

Minimax is a kind of backtracking algorithm that is used in decision making and game theory to find the optimal move for a player, assuming that your opponent also plays optimally. It is widely used in two player turn-based games such as Tic-Tac-Toe, Backgammon, Mancala, Chess, etc.

In Minimax the two players are called maximizer and minimizer. The maximizer tries to get the highest score possible while the minimizer tries to do the opposite and get the lowest score possible.

Every board state has a value associated with it. In a given state if the maximizer has upper hand then, the score of the board will tend to be some positive value. If the minimizer has the upper hand in that board state then it will tend to be some negative value. The values of the board are calculated by some heuristics which are unique for every type of game.

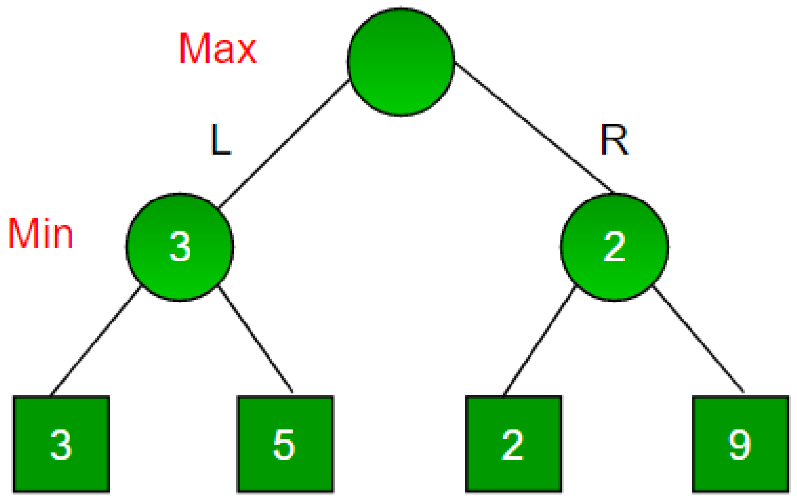
Example: Consider a game which has 4 final states and paths to reach final state are from root to 4 leaves of a perfect binary tree as shown below. Assume you are the maximizing player and you get the first chance to move, i.e., you are at the root and your opponent at next level. Which move you would make as a maximizing player considering that your opponent also plays optimally?



Since this is a backtracking based algorithm, it tries all possible moves, then backtracks and makes a decision.

Maximizer goes LEFT: It is now the minimizers turn. The minimizer now has a choice between 3 and 5. Being the minimizer it will definitely choose the least among both, that is 3 Maximizer goes RIGHT: It is now the minimizers turn. The minimizer now has a choice between 2 and 9. He will choose 2 as it is the least among the two values. Being the maximizer you would choose the larger value that is 3. Hence the optimal move for the maximizer is to go LEFT and the optimal value is 3.

Now the game tree looks like below :



The above tree shows two possible scores when maximizer makes left and right moves.

Note: Even though there is a value of 9 on the right subtree, the minimizer will never pick that. We must always assume that our opponent plays optimally.

---

3 Code

[1]:
```
#Import the necessary libraries
import numpy as np
from math import inf as infinity
```

[2]:
```
#Set the Empty Board
game_state = [[' ',' ',' '],
              [' ',' ',' '],
              [' ',' ',' ']]
#Create the Two Players as 'X'/'O'
players = ['X','O']
```

[3]:
```
#Method for checking the correct move on Tic-Tac-Toe
def play_move(state, player, block_num):
    if state[int((block_num-1)/3)][(block_num-1)%3] == ' ':
        state[int((block_num-1)/3)][(block_num-1)%3] = player
    else:
        block_num = int(input("Block is not empty, ya blockhead! Choose again:␣
 ↪"))
        play_move(state, player, block_num)
        None
```

[4]:
```
#Method to copy the current game state to new_state of Tic-Tac-Toe
def copy_game_state(state):
    new_state = [[' ',' ',' '],[' ',' ',' '],[' ',' ',' ']]
    for i in range(3):
        for j in range(3):
            new_state[i][j] = state[i][j]
    return new_state
```

[5]:
```
#Method to check the current state of the Tic-Tac-Toe
def check_current_state(game_state):
    draw_flag = 0
    for i in range(3):
        for j in range(3):
            if game_state[i][j] == ' ':
                draw_flag = 1

    if draw_flag == 0:
        return None, "Draw"

    # Check horizontals in first row
    if (game_state[0][0] == game_state[0][1] and game_state[0][1] ==␣
 ↪game_state[0][2] and game_state[0][0] != ' '):
        return game_state[0][0], "Done"
    if (game_state[1][0] == game_state[1][1] and game_state[1][1] ==␣
↪game_state[1][2] and game_state[1][0] != ' '):
        return game_state[1][0], "Done"
    if (game_state[2][0] == game_state[2][1] and game_state[2][1] ==␣
↪game_state[2][2] and game_state[2][0] != ' '):
        return game_state[2][0], "Done"

    # Check verticals in first column
    if (game_state[0][0] == game_state[1][0] and game_state[1][0] ==␣
↪game_state[2][0] and game_state[0][0] != ' '):
        return game_state[0][0], "Done"
    # Check verticals in second column
    if (game_state[0][1] == game_state[1][1] and game_state[1][1] ==␣
↪game_state[2][1] and game_state[0][1] != ' '):
        return game_state[0][1], "Done"
    # Check verticals in third column
    if (game_state[0][2] == game_state[1][2] and game_state[1][2] ==␣
↪game_state[2][2] and game_state[0][2] != ' '):
        return game_state[0][2], "Done"

    # Check left diagonal
    if (game_state[0][0] == game_state[1][1] and game_state[1][1] ==␣
↪game_state[2][2] and game_state[0][0] != ' '):
        return game_state[1][1], "Done"
    # Check right diagonal
    if (game_state[2][0] == game_state[1][1] and game_state[1][1] ==␣
↪game_state[0][2] and game_state[2][0] != ' '):
        return game_state[1][1], "Done"

    return None, "Not Done"
```

[6]:
```
#Method to print the Tic-Tac-Toe Board
def print_board(game_state):
    print('----------------')
    print('| ' + str(game_state[0][0]) + ' || ' + str(game_state[0][1]) + ' || '␣
↪+ str(game_state[0][2]) + ' |')
    print('----------------')
    print('| ' + str(game_state[1][0]) + ' || ' + str(game_state[1][1]) + ' || '␣
↪+ str(game_state[1][2]) + ' |')
    print('----------------')
    print('| ' + str(game_state[2][0]) + ' || ' + str(game_state[2][1]) + ' || '␣
↪+ str(game_state[2][2]) + ' |')
    print('----------------')
```

[7]:
```
#Method for implement the Minimax Algorithm
def getBestMove(state, player):
    winner_loser , done = check_current_state(state)

    if done == "Done" and winner_loser == 'O':
        return 1
    elif done == "Done" and winner_loser == 'X':
        return -1
    elif done == "Draw":
        return 0

    moves = []
    empty_cells = []

    #Append the block_num to the empty_cells list
    for i in range(3):
        for j in range(3):
            if state[i][j] == ' ':
                empty_cells.append(i*3 + (j+1))

    for empty_cell in empty_cells:
        move = {}

        move['index'] = empty_cell

        new_state = copy_game_state(state)

        play_move(new_state, player, empty_cell)

        #if player is computer
        if player == 'O':
            result = getBestMove(new_state, 'X')
            move['score'] = result
        else:
            result = result = getBestMove(new_state, 'O')
            move['score'] = result

        moves.append(move)

    # Find best move
    best_move = None
    if player == 'O':
        best = -infinity
        for move in moves:
            if move['score'] > best:
                best = move['score']
                best_move = move['index']
    else:
        best = infinity
        for move in moves:
            if move['score'] < best:
                best = move['score']
                best_move = move['index']

    return best_move
```

[8]:
```
# Now PLaying the Tic-Tac-Toe Game
play_again = 'Y'
while play_again == 'Y' or play_again == 'y':
    #Set the empty board for Tic-Tac-Toe
    game_state = [[' ',' ',' '],
                  [' ',' ',' '],
                  [' ',' ',' ']]
    #Set current_state as "Not Done"
    current_state = "Not Done"
    print("\nNew Game!")

    #print the game_state
    print_board(game_state)

    #Select the player_choice to start the game
    player_choice = input("Choose which player goes first - X (You) or␣
 ↪O(Computer): ")

    #Set winner as None
    winner = None

    #if player_choice is ('X' or 'x') for humans else for computer
    if player_choice == 'X' or player_choice == 'x':
        current_player_idx = 0
    else:
        current_player_idx = 1

    while current_state == "Not Done":
        #For Human Turn
        if current_player_idx == 0:
            block_choice = int(input("Your turn please! Choose where to place (1␣
 ↪to 9): "))
            play_move(game_state ,players[current_player_idx], block_choice)

        else:   # Computer turn
            block_choice = getBestMove(game_state, players[current_player_idx])
            play_move(game_state ,players[current_player_idx], block_choice)
            print("AI plays move: " + str(block_choice))
        print_board(game_state)
        winner, current_state = check_current_state(game_state)
```

```python
        if winner is not None:
            print(str(winner) + " won!")
        else:
            current_player_idx = (current_player_idx + 1)%2

        if current_state == "Draw":
            print("Draw!")

    play_again = input('Wanna try again?(Y/N) : ')
    if play_again == 'N':
        print('Thank you for playing Tic-Tac-Toe Game!!!!!!!')
```

```
New Game!
----------------
|   ||   ||   |
----------------
|   ||   ||   |
----------------
|   ||   ||   |
----------------
Choose which player goes first - X (You) or O(Computer): X
Your turn please! Choose where to place (1 to 9): 1
----------------
| X ||   ||   |
----------------
|   ||   ||   |
----------------
|   ||   ||   |
----------------
AI plays move: 2
----------------
| X || O ||   |
----------------
|   ||   ||   |
----------------
|   ||   ||   |
----------------
Your turn please! Choose where to place (1 to 9): 3
----------------
| X || O || X |
----------------
|   ||   ||   |
----------------
|   ||   ||   |
----------------
AI plays move: 6
```

```
----------------
| X || O || X |
----------------
|   ||   || O |
----------------
|   ||   ||   |
----------------
Your turn please! Choose where to place (1 to 9): 9
----------------
| X || O || X |
----------------
|   ||   || O |
----------------
|   ||   || X |
----------------
AI plays move: 5
----------------
| X || O || X |
----------------
|   || O || O |
----------------
|   ||   || X |
----------------
Your turn please! Choose where to place (1 to 9): 8
----------------
| X || O || X |
----------------
|   || O || O |
----------------
|   || X || X |
----------------
AI plays move: 7
----------------
| X || O || X |
----------------
|   || O || O |
----------------
| O || X || X |
----------------
Your turn please! Choose where to place (1 to 9): 4
----------------
| X || O || X |
----------------
| X || O || O |
----------------
| O || X || X |
----------------
Draw!
```

```
Wanna try again?(Y/N) : N
Thank you for playing Tic-Tac-Toe Game!!!!!!!
```

4 Bonus Question

[9]:
```python
# Initial values of Aplha and Beta
MAX, MIN = 1000, -1000

# Returns optimal value for current player
#(Initially called for root and maximizer)
def aplha_beta_pruning(depth, nodeIndex, maximizingPlayer,
        values, alpha, beta):

    if depth == 3:
        return values[nodeIndex]

    if maximizingPlayer:
        best_move = MIN
        for i in range(0, 2):
            val = aplha_beta_pruning(depth + 1, nodeIndex * 2 + i, False,␣
 ↪values, alpha, beta)
            best_move = max(best_move, val)
            alpha = max(alpha, best_move)

            if beta <= alpha:
                break
        return best_move

    else:
        best_move = MAX
        for i in range(0, 2):
            val = aplha_beta_pruning(depth + 1, nodeIndex * 2 + i,True, values,␣
 ↪alpha, beta)
            best_move = min(best_move, val)
            beta = min(beta, best_move)

        # Alpha Beta Pruning
            if beta <= alpha:
                break
    return best_move
```