# Undefeatable Tic-Tac-Toe Documentation

FEBRUARY 14

**PSUT**
**Authored by: Ameer Jamal**
**Supervised by Dr.Saleh Abu-Soud**

# Where it all starts

## About Tic-Tac-Toe

Tic-tac-toe (also known as noughts, crosses, or X's and O's) is a simple paper-and-pencil game played by two players, X and O, who take turns filling the spaces on a 3×3 grid. The player who succeeds in placing three of their marks in a horizontal, vertical, or diagonal row wins the game.

Tic-tac-toe is a zero-sum game, which means one person's gain is another person's loss, it's essentially a win-lose kind of game. Also, it's a perfect information game which means both players, when making any decision, are perfectly informed of all the events that have previously occurred, including the "initialization event" of the game.

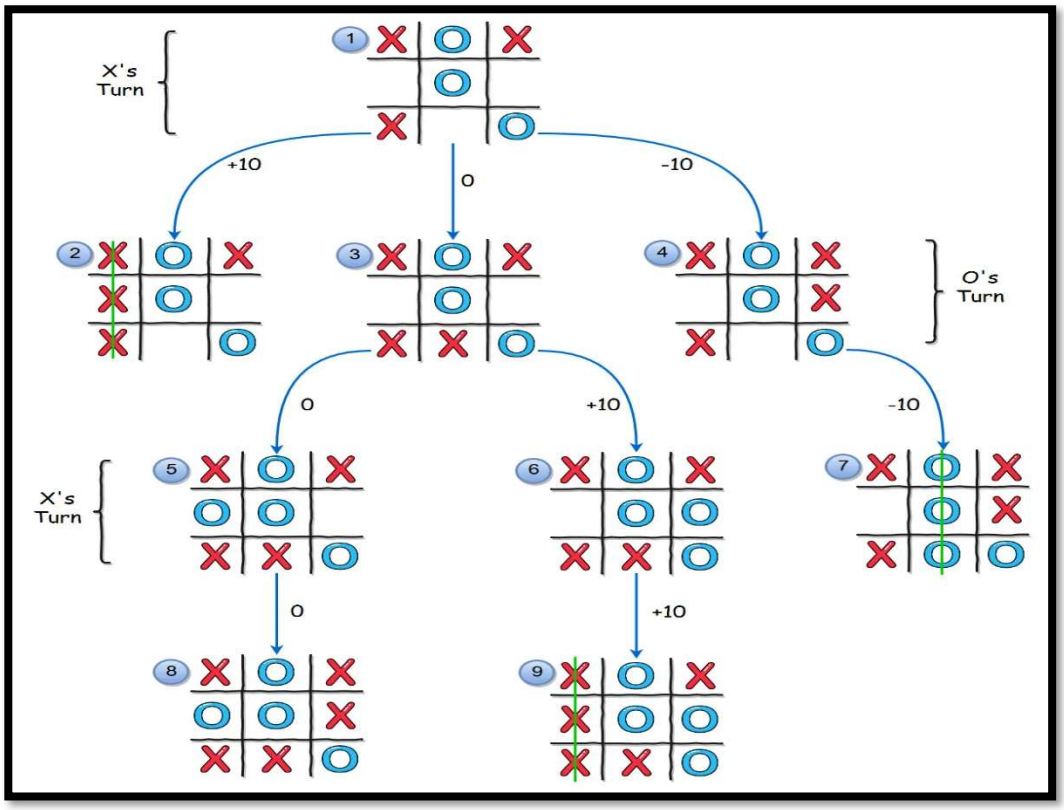
A quick summation of the idea: In a two-player (A vs B) game, if player A scores x points (utility units), player B loses x points. Total gains/losses always sum up to 0.

Keeping all that in mind, we used a minimax algorithm that suited this case. Before we get to the code here let's start with what the minimax algorithm even is.

> ***"The key to artificial intelligence has always been the representation."***
>
> ***—Jeff Hawkins***

# The Minimax Algorithm

Minimax is a kind of backtracking or recursive algorithm that is used in decision making and game theory to find the optimal move for a player, The ultimate goal of this algorithm is to minimize the maximum loss (minimize the worst-case scenario). Here is a small representation of how this would work in a mid-game tic-tac-toe situation




Of course for this algorithm to work some assumptions need to be taken place.

1-The human player should be playing optimally or trying to win. Making moves at random or trying to lose might interfere with the algorithm's effectiveness.

2-The game must be purely strategic and cannot incorporate any sort of chance component (i.e., Monopoly, Poker, Tetris). Note:

Variations of this algorithm can be used to account for the "luck factor".

The algorithm incorporates three basic functions: Maximize and Minimize, as well as a Utility Calculation function. The pseudocode looks something like this:

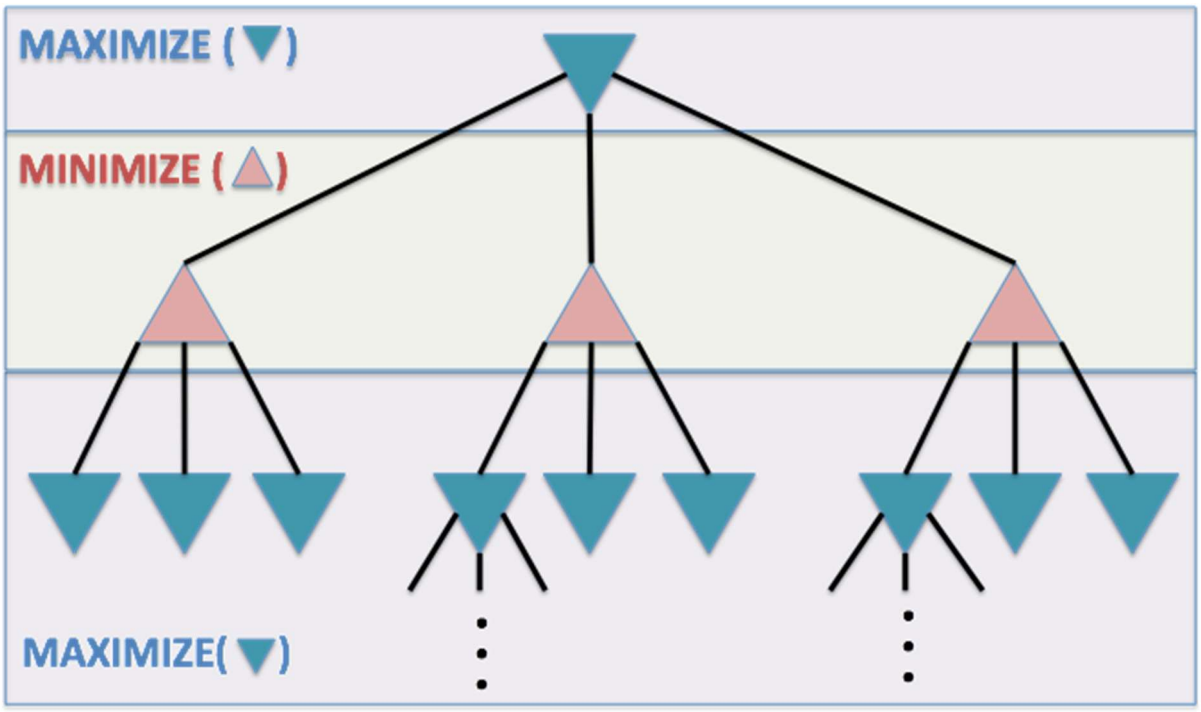
```
Define MAXIMIZE (board):
    Returns a board configuration & and its utility

    If board is in terminal state, or reached depth limit:

        return None, CALCULATE_UTILITY (board)

    maximum utility = -∞
    move with maximum utility = None

    for move possibility in board.children:

        ( _ , board) = MINIMIZE (move possibility):

        if utility > maximum utility:

            move with maximum utility = move possibility
            maximum utility = utility

    return move with maximum utility, maximum utility
```

```
Define MINIMIZE (board):
    Returns a board configuration & and its utility

    If board is in terminal state or reached depth limit:

        return None, CALCULATE_UTILITY (board)

    minimum utility = ∞
    move with minimum utility = None

    for move possibility in board.children:

        ( _ , board) = MAXIMIZE (move possibility):

        if utility < minimum utility:

            move with minimum utility = move possibility
            minimum utility = utility

    return move with minimum utility, minimum utility
```

The two key takeaways from this schematic are:

1. The algorithm is recursive in that Maximize Calls Minimize, and Minimize Calls Maximize.
2. We have conditions that break us out of the recursive loop. This would happen if, for instance, we reach a node in our search space where someone has won the game, the board is full, or perhaps most commonly, we have reached the pre-determined depth limit.

Returning to the scenario where it's the computer's turn, the computer would call the Maximize function on the current board. This would call Minimize on each child of the board, which calls Maximize on each grandchild, and so on and so forth… here is a representation of this:




# The Code Behind it All

THE JAVASCRIPT

Minimax.js

First, the bestMove function finds the best move for the AI to play by initializing a bestScore variable, going through all the positions in the tic tac toe game, then calling the minimax function to find the score of each one of the empty positions, then compares the score of each position to the bestScore variable to find the best move for the AI to play, saves its coordinates, plays it, and then gives the turn to the human player.

We set the scores to 10 for x, -10 for y, and 0 for a tie so that the minimax can calculate the scores for each move and know what's a good move and what's a bad move.

Then finally comes the minimax function which first checks if someone won the game or not, if there is a winner we return the score (this is the base condition to terminate the recursive function meaning we reached a terminal node and have found a winner which is either human or AI), if there is no winner we continue through the function which is divided into two sections, a maximizing player section (the AI) and a minimizing player section (the human), these two sections take turns (alternate) in order to generate the tree.

In the maximizing player section, we set the "bestScore" variable to the lowest value possible and look for the highest score value we can get, then we check if a position is empty, if it is empty we assume the AI played that position, we then recursively call the minimax function while increasing the depth by one - to generate the children nodes – and give the "isMaximizing" variable in the parameters a value of "false" in order to enter the else statement (meaning it's the minimizing player's turn now).

In the minimizing player section, we do the same thing we have done in the maximizing player section except that here we are

looking for the lowest score value (assuming the optimal choice the human player can choose in order to win).
The recursive function stops calling itself when we find a winner.

## Sketch.js and P5.js

The sketch.js file is what is running the drawing sequence of the board X's, O's, and the text that shows the final evaluation before the AI moves, it is made using the creative coding library P5.js which is a JavaScript library that has a ton of drawing functionality that is used to make developing 2D games nice and easy which is especially useful for tic-tac-toe.
In sketch.js you will find it has a setup() and draw() function, the concept is very simple, the setup is executed once, and draw is executed continuously until the termination of the program. By default the frame rate (The number of times the draw() function is called) is 60.

In our sketch.js we globally define the tic tac toe board to be empty at the beginning of the game, defining both its height and width. since the board only needs to be drawn once we place the board drawing in the setup() function  We then set the AI player as 'X' in the game and subsequently the human plays 'O's.

We use the starter function as a way to make the AI start first, this function only gets called if the user clicks on the 'AI starts first' button, the function works by checking if the start index is equal to

zero and if so we call the “bestMove”(which runs the AI’s best move), then we add one to the start index value if the value is already incremented meaning its equal to one that means that the AI has already started playing and will display a message accordingly, this will allow not let the game continuously let the button be pressed.

Then comes the “checkWinner” function which checks if anyone has won the game by checking if we have O’s or X’s that are aligned in a way that wins the game. It checks if their aligned horizontally, vertically, or diagonally by calling the equals3 function.

The “equals3” function is a helper of the “checkWinner” function, it compares the values (X, O, and the empty positions) in a vertical, horizontal, and diagonal manner, and if we get 3 consecutive positions that have the same value (either X or O) then we have a winner.

The “openSpots” function checks the number of open spots so that when there are no open spots left that would mean the board is full and no one has won and the outcome is a tie.

# THE STYLING

## introstyle.html

This code is written for the index page which takes you to the Tic-tac-toe game page itself.
It's a simple bit of code that first sets a background for the body of the page, chooses a font, a color, justifies the content, and aligns the text at the center of the page. Also, there is the animation property -in CSS- which is defined with the @keyframe to animate the background color which concludes a nice introduction to our game.

## Style.html

This code is the styling behind the Tic-tac-toe page. It starts with aligning the margin and the height of the body and adding a background. Then there is the button class, where we set the width, padding, color, background, fix the text size, font, and align the text to the center. This code ensures the centering of the board and alignment of the elements and also adds a nice dark but saturated display to play the game.