Minimax Algorithm

The Minimax algorithm is a decision-making process designed for two-player zero-sum games (a game where one player's gain is equal to the other player's loss). It helps the computer evaluate all possible future moves and will choose the one that minimizes the potential loss or maximizes the potential win.

Overview

Minimax constructs a game tree where each node represents a board state and each branch is a possible move. The tree extends from the current board state down to 'terminal' states - positions where the game is over (win, loss, or draw).

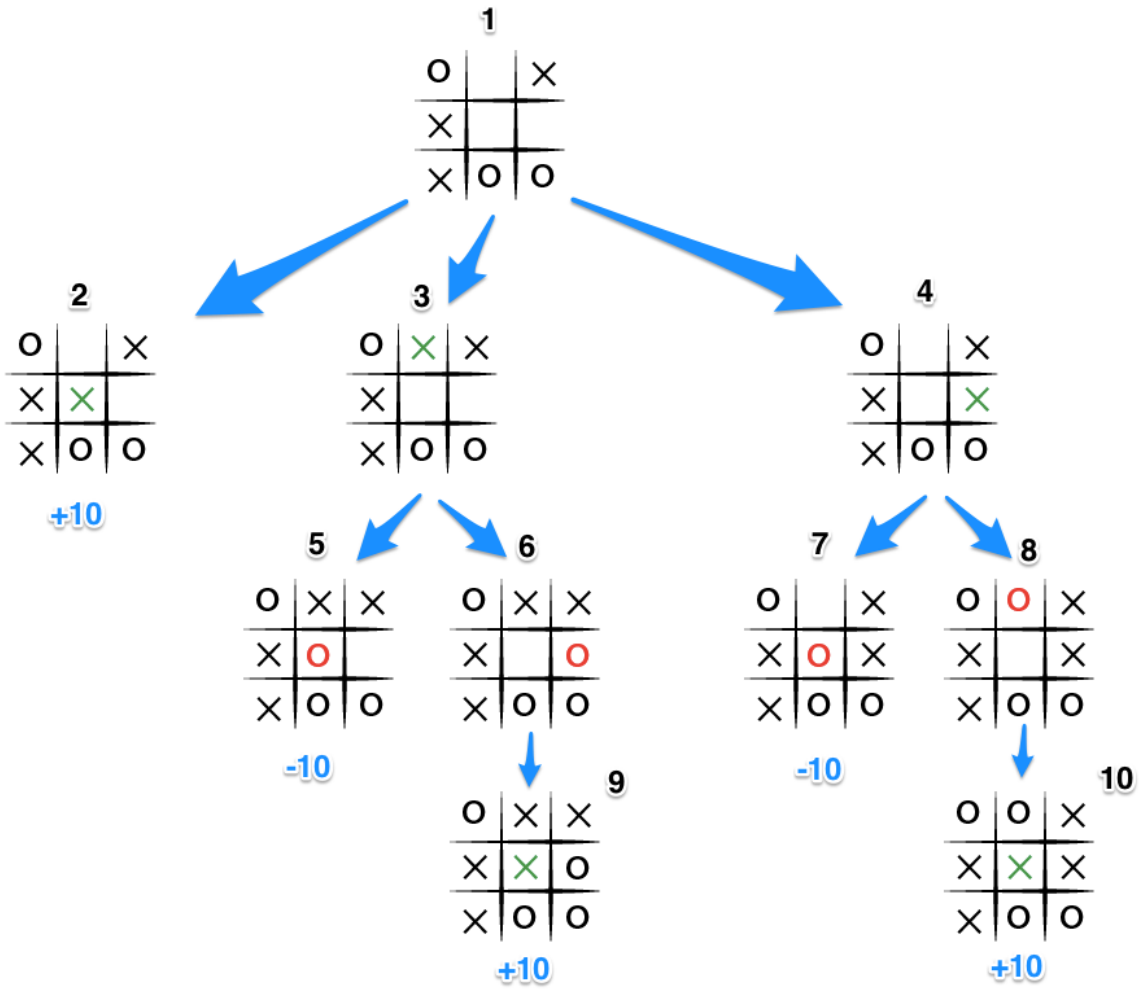
There are two roles - the Maximizer and the Minimizer:

- Maximizer (Computer): Tries to maximize the score. In Tic-Tac-Toe, the computer uses the mark 'O' and tries to achieve the highest score possible (+10 for a win).
- Minimizer (Player): Tries to minimize the score. The player uses an 'X' and the algorithm assumes that the player will choose moves to minimize the computer's score (-10 for a computer loss).

When the algorithm reaches a terminal node (win, loss, or draw) it assigns a score. If the computer wins, the node gets a +10. If the player wins, the node gets a -10. If it's a draw, the node gets a 0.

Starting from these terminal nodes, minimax will work backwards through the tree. At each point, if it's the computer's turn, it picks the move with the highest score. If it's the player's turn, it picks the move with the lowest score. This is assuming that the player will play optimally.

Here's an example of what minimax does in Tic-Tac-Toe



Step-by-step

Using the Tic-Tac-Toe.jsx file as an example, here's what we're doing at each step.

1. The `evaluateBoard` function checks the board for a win by either player. The outcome determines the score for that terminal state (+10 for a computer win, -10 for a player win). A draw returns 0.

2. The `minimax` function simulates every possible move from the current board state. For every empty cell, it temporarily places an 'X' or an 'O' depending on whose turn it is, and then runs the function again with that new board state.

    - When it's the computer's turn, the algorithm tries to maximize the score.
    - When it's the player's turn, the algorithm tries to minimize the score.

    The algorithm also factors 'depth' into the equation. Depth is the amount of moves it takes to reach the terminal state. This means that if a win can be achieved in fewer moves (less depth), it will favor that move.

3. The `computerTurn` function tests each possible move, calls `minimax` to evaluate the resulting board, and then selects the move with the highest score.

If implemented correctly, the player will never win since our functions are constantly simulating every possible move by both players and choosing the best move for the computer (the one with the highest score). At best, there will be a draw.

Code

Let's take the main `minimax` function from our Tic-Tac-Toe game:

```
function minimax(board, depth, isMaximizing) {
    const score = evaluateBoard(board);

    if (score == 10) return score - depth;
    if (score == -10) return score + depth;
    if (!isMovesLeft(board)) return 0;

    if (isMaximizing) {
        let best = -Infinity;
        for (let i = 0; i < board.length; i++) {
            for (let j = 0; j < board[i].length; j++) {
                if (board[i][j] == null) {
                    board[i][j] = 'O';
                    best = Math.max(best, minimax(board, depth + 1, false));
                    board[i][j] = null;
                }
            }
        }
        return best;
    } else {
        let best = Infinity;
        for (let i = 0; i < board.length; i++) {
            for (let j = 0; j < board[i].length; j++) {
                if (board[i][j] == null) {
                    board[i][j] = 'X';
                    best = Math.min(best, minimax(board, depth + 1, true));
                    board[i][j] = null;
                }
            }
        }
        return best;
    }
}
```

1. Evaluating the board state:

```
```js
const score = evaluateBoard(board);
```
```

- This checks if the current board state is in a terminal state (i.e., a win for either player) and returns a score.

2. Terminal state cases

```
```js
if (score == 10) return score - depth;
if (score == -10) return score + depth;
if (!isMovesLeft(board)) return 0;
```
```

- When the computer wins `(score == 10)`, returning `score - depth` ensures that winning in fewer moves (lower depth) results in a higher score.
- When the player wins `(score == -10)`, returning `score + depth` lowers the score for moves that allow an early loss.
- If there are no moves left, the game is a draw, so the function returns 0.

These are base cases for the recursion; when one is met, it doesn't need to simulate further.

3. Recursion

The algorithm now decides based on whose turn it is.

```
if (isMaximizing) {
    let best = -Infinity;
    for (let i = 0; i < board.length; i++) {
        for (let j = 0; j < board[i].length; j++) {
            if (board[i][j] == null) {
                board[i][j] = 'O';
                best = Math.max(best, minimax(board, depth + 1, false));
                board[i][j] = null;
            }
        }
    }
    return best;
}
```

This is the 'maximizer', which wants to choose the move that results in the highest score possible.

Process:

- Iterate over every cell in the board.
- For each empty cell (null), simulate placing an 'O' (the computer's mark).
- Call minimax again (recursively) with:
    - Increased `depth` to track the number of moves made.
    - Set `isMaximizing` to false to switch the turn to the player (so it will 'minimize' next turn).
- After evaluating that move, reset the cell back to null so that the board is unchanged for other simulations.
- The `Math.max` function ensures that the highest score among all simulated moves is chosen.
- Return the best move.

```
else {
    let best = Infinity;
    for (let i = 0; i < board.length; i++) {
        for (let j = 0; j < board[i].length; j++) {
            if (board[i][j] == null) {
                board[i][j] = 'X';
                best = Math.min(best, minimax(board, depth + 1, true));
                board[i][j] = null;
            }
        }
    }
    return best;
}
```

This is the 'minimizer', which wants to choose the move with the lowest score (the worst outcome for the computer).

Process:

- Iterate over every cell in the board.
- For each empty cell (null), simulate placing an 'X' (the player's mark).
- Call minimax again (recursively) with:
  - Increased `depth`.
  - Set `isMaximizing` to true to switch the turn to the computer (so it will 'maximize' next turn).
- After evaluating that move, reset the cell back to null so that the board is unchanged for other simulations.
- The `Math.min` function ensures that the lowest score among all simulated moves is chosen.
- Return the best move.

Effectively, this function recursively evaluates every possible sequence of moves until a terminal state is reached. Then, it chooses the move with the highest score, effectively making the optimal play.

Computer's Turn

Let's also look at the code for when it's the computer's turn.

```
function computerTurn(board) {
    let bestVal = -Infinity;
    let bestMove = null;

    for (let i = 0; i < board.length; i++) {
        for (let j = 0; j < board[i].length; j++) {
            if (board[i][j] == null) {
                board[i][j] = 'O';
                const moveVal = minimax(board, 0, false);
                board[i][j] = null;
                if (moveVal > bestVal) {
                    bestVal = moveVal;
                    bestMove = { i, j };
```

```
                }
            }
        }
    }

    if (bestMove) {
        const newBoard = board.map(row => [...row]);
        newBoard[bestMove.i][bestMove.j] = 'O';
        setBoard(newBoard);
        setTurn(true);
    }
}
```

1. Initialization

```
```js
let bestVal = -Infinity;
let bestMove = null;
```
```

- `bestVal` stores the highest score found from each possible move. It starts at `-Inifnity` so that any move that's evaluated first will be higher.
- `bestMove` stores the coordinates `{ i , j }` of the move that has the highest score according to our minimax function.

2. Iteration

```
```js
for (let i = 0; i < board.length; i++) {
    for (let j = 0; j < board[i].length; j++) {
        if (board[i][j] == null) {
            board[i][j] = 'O';
            const moveVal = minimax(board, 0, false);
            board[i][j] = null;
            if (moveVal > bestVal) {
                bestVal = moveVal;
                bestMove = { i, j };
            }
        }
    }
}
```
```

- The nested loops iterate over every cell on the board.
- If it reaches an empty cell on the board:
    - Temporarily place an 'O' in that cell.

- Call the minimax function with that temporary 'O', starting depth of 0, and `isMaximizing` to false because the next turn would be the player's (minimizing) turn.
- This minimax function will recursively evaluate the results of that move, returning a score that reflects how optimal that move is for the computer.
- After the evaluation, change the cell back to 'null', since we don't actually want to place anything on the board.
- Then, it compares the returned score (moveVal) with the best score (bestVal)
- If the current move is a higher score, it updates our bestVal with the move's coordinates.

3. Choosing the Best Move

```
```js
if (bestMove) {
    const newBoard = board.map(row => [...row]);
    newBoard[bestMove.i][bestMove.j] = 'O';
    setBoard(newBoard);
    setTurn(true);
}
```
```

- We create a copy of the board to avoid directly modifying the current state.
- The chosen move from `bestMove` is applied by setting that cell's coordinates to 'O'.
- We then update our board state and change turns to the player.