

The minimax algorithm is used to implement the AI player.

## Explanation of the code

The `minimax` method is a recursive function that tries to find the optimal move for the computer player given the current state of the game.

Here's a step-by-step explanation of the `minimax` function:

1. The function first checks if there's a winner on the board by calling `self.check_winner` function from class TicTacToe. If the current player has won, the function returns the score -1 (for the human player) or 1 (for the computer player) and the current position.
2. The function then checks if there's a tie by calling `self.is_board_full` function from the class TicTacToe. If the board is full and there's no winner, the function returns the score 0 and the current position.
3. If the game is not over, the function creates an empty list `moves` to store the score and position for each possible move.
4. The function then loops through each empty position on the board (indicated by "-") and sets the position to the current player's symbol.
5. The function then calls itself recursively with the new board and the other player's symbol. This will simulate the other player's move, and the function will return the score and position for that move.
6. The function then sets the position on the board back to "-" to simulate that the move was not made.
7. The function appends the score and position for the current move to the `moves` list.
8. After all possible moves have been explored, the function uses the `player` argument to determine whether to return the maximum or minimum score and position.
9. If the current player is the computer player, the function returns the move with the highest score.
10. If the current player is the human player, the function returns the move with the lowest score.

The minimax function uses the minimax algorithm to determine the best move for the computer given the current situation on the board. It returns two values: the best score and the best move.

Assuming that the board is represented as a list of strings, where each string represents a position on the board, and that the players are represented as the strings " O " and " X ", here is an example of possible values for the best score and best move variables:

- best\_score = 1, best\_move = 4: This means that the best move for the computer is to place an " O " in position 4 on the board, and the resulting score for this move is 1 (indicating a win for the computer).
- best\_score = -1, best\_move = 2: This means that the best move for the computer is to place an " O " in position 2 on the board, and the resulting score for this move is -1 (indicating a loss for the computer).
- best\_score = 0, best\_move = 8: This means that the best move for the computer is to place an " O " in position 8 on the board, and the resulting score for this move is 0 (indicating a tie game).

#Function to Generate an AI move (optimal move given the current situation on the board)

```
def minimax(self, board, player):
```

```
    """
```

```
    Minimax function to determine the best move for the computer.
    Returns the best score and best move.
    """
```

```
    # Define the player and opponent markers
```

```
    if player == " O ":
        opponent = " X "
```

```
    else:
        opponent = " O "
```

```
    # Define the base cases
```

```
    if self.check_winner(" O "):
        return (1, None)
```

```
    elif self.check_winner(" X "):
        return (-1, None)
```

```
    elif self.is_board_full():
        return (0, None)
```

```
    # Initialize the best_score and best_move variables
```

```
    if player == " O ":
        best_score = -float("inf")
```

```
    else:
        best_score = float("inf")
```

```
    best_move = None
```

```
    # Iterate through all possible moves and determine the best score and best move
```

```
    for i, pos in enumerate(board):
```

```
        if pos == " - ":
```

```
            board[i] = player
```

```
            score, _ = self.minimax(board, opponent)
```

```
            board[i] = " - "
```

```
            if player == " O ":
```

```
                if score > best_score:
                    best_score = score
                    best_move = i
```

```
            else:
```

```
                if score < best_score:
                    best_score = score
                    best_move = i
```

```
    return (best_score, best_move)
```

This is a Python function named `minimax` that is designed to implement the minimax algorithm to determine the optimal move for the computer player in a game of Tic Tac Toe.

Here is a detailed breakdown of the code:

```
def minimax(self, board, player):
```

This is the function definition, which takes in three arguments: `self`, `board`, and `player`. The `self` parameter is used to refer to the object that the function belongs to (in object-oriented programming), while `board` represents the current state of the game board (i.e., a list of strings representing the current marks on the board). `player` represents the current player's mark (either "O" or "X").

python

```
if player == " O ":
    opponent = " X "
else:
    opponent = " O "
```

Here, the function checks which player is the current player by comparing the `player` argument with the string `" O "` (with spaces). If the current player is "O", then `opponent` is set to "X", and vice versa.

```
if self.check_winner(" O "):  
    return (1, None)  
elif self.check_winner(" X "):  
    return (-1, None)  
elif self.is_board_full():  
    return (0, None)
```

These are the base cases for the recursive minimax algorithm. If either player has won the game, the function returns 1 or -1 (respectively) as the best score and None as the best move. If the game board is full (i.e., no more moves can be made), then the function returns 0 as the best score and None as the best move.


python

```
if player == " O ":  
    best_score = -float("inf")  
else:  
    best_score = float("inf")  
best_move = None
```

This initializes the `best\_score` and `best\_move` variables based on the current player. If the current player is "O", then `best\_score` is initialized to negative infinity, and vice versa.

```
for i, pos in enumerate(board):
    if pos == " - ":
        board[i] = player
        score, _ = self.minimax(board, opponent)
        board[i] = " - "
        if player == " 0 ":
            if score > best_score:
                best_score = score
                best_move = i
        else:
            if score < best_score:
                best_score = score
                best_move = i
```

This is the main recursive part of the minimax algorithm. The function loops through all possible moves on the game board and determines the score associated with each move. The score is determined by recursively calling the `minimax` function on the updated game board with the opponent as the current player. The `score` variable represents the best score that the opponent can achieve on their turn. The function then updates `best\_score` and `best\_move` if the current move is better than the previously best move.

 Copy code

```
return (best_score, best_move)
```

Finally, the function returns a tuple containing the best score and the corresponding best move that the computer player should make.

Point X Lite