Rishab Kumar
15-Jun-2019

Tic - Tac - Toe

Perfect AI model by Minimax Algorithm

I recently built an unbeatable game of tic tac toe. In order to make the game unbeatable, it was necessary to create an algorithm that could calculate all the possible moves available for the computer player and use some metric to determine the best possible move. After extensive research it became clear that the Minimax algorithm was right for the job. It is a well known algorithm in Game Theory. It took a little while to really fundamentally understand the algorithm and implement it in my game. I found many code examples and explanations, but none that really walked a simpleton like me through the ins and outs of the process. I am writing this project file so that anyone can understand my approach and appreciate the elegance of this algorithm.

pseudocode is written in red color

Describing a Perfect Game of Tic Tac Toe

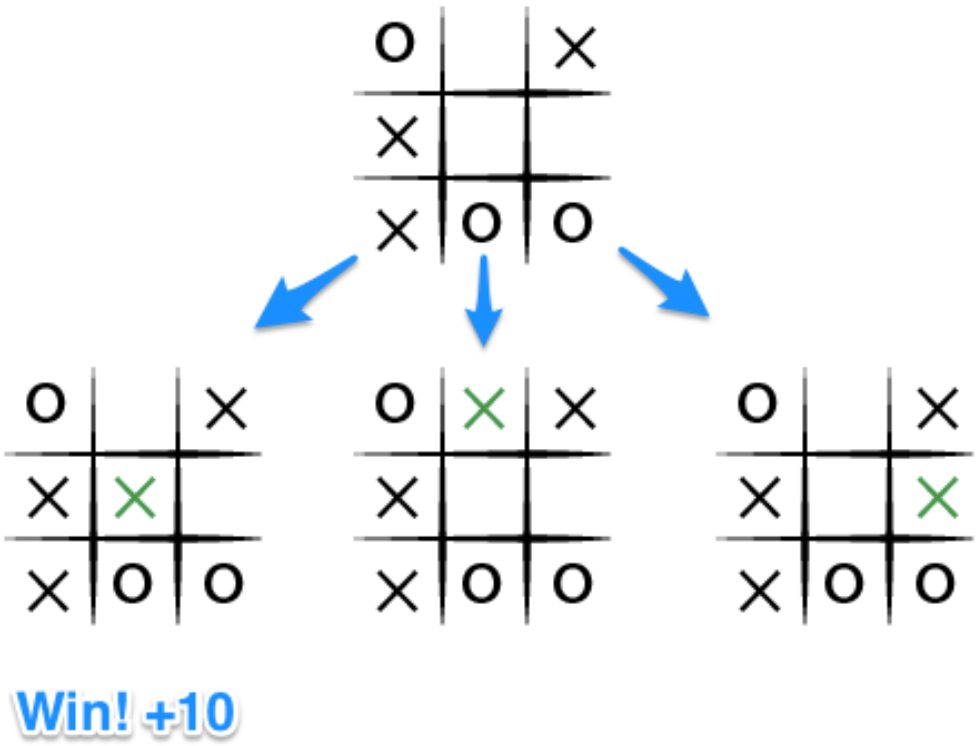
To begin, let's start by defining what it means to play a perfect game of tic tac toe:
If I play perfectly, every time I play I will either win the game, or I will draw the game. Furthermore if I play against another perfect player, I will always draw the game. How might we describe these situations 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

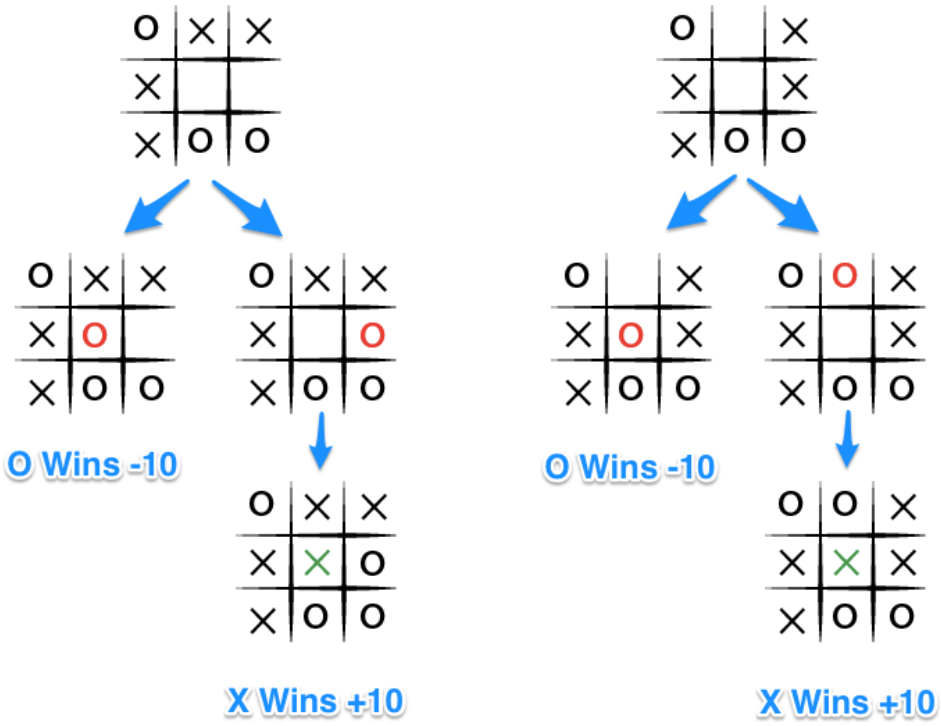
To apply this, let's take an example from near the end of a game, where it is my turn. I am X. My goal here, obviously, is to maximize my end game score.



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.

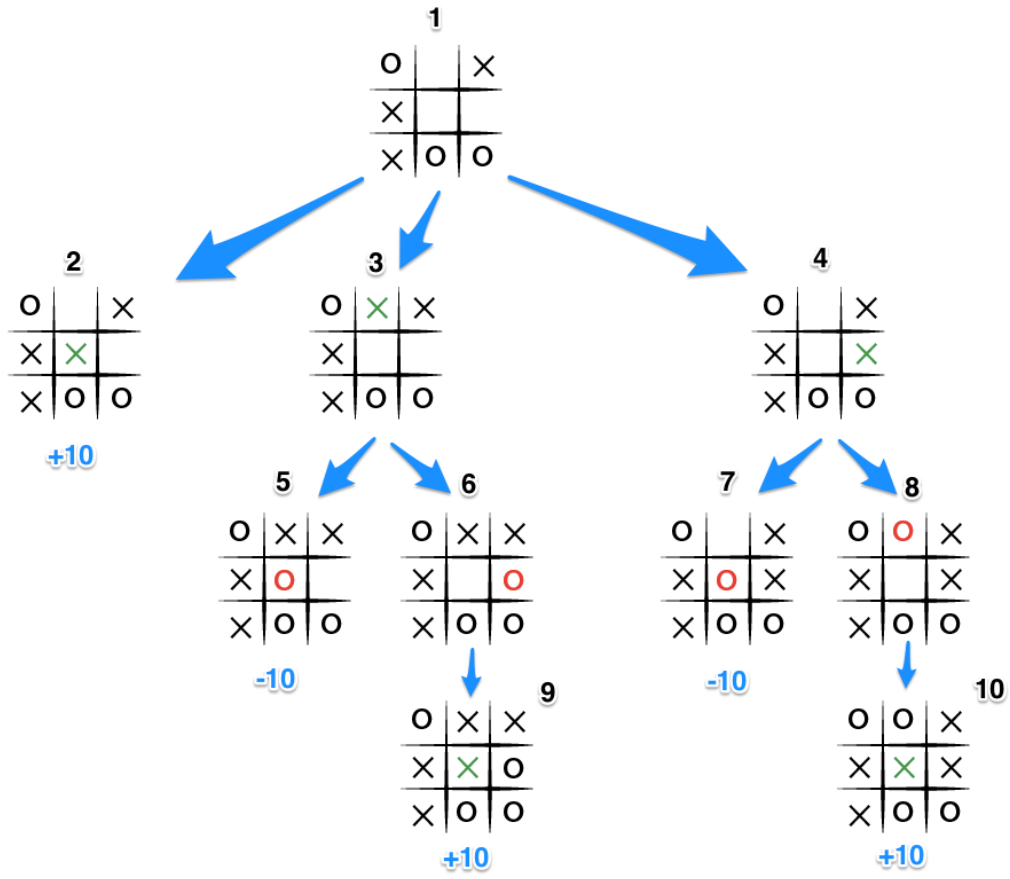
Describing Minimax

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
-

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.

That is certainly a lot to take in. And that is why we have a computer execute this algorithm.
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
    else if game.win?(@opponent)
        return -10
    else
        return 0
```

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:
    if win:
      return 10
    else if lose:
      return -10
    else if draw:
      return 0
    else if computer's turn:
      choose maximum of all children - depth
    else if player's turn:
      choose minimum of all children + depth
```

When this algorithm is run, the ultimate choice of best move is stored in the game tree, which is then used to return the new game state in which the current player has moved.

Pseudocode for gameplay:

```
continue until terminal state occurs:
    take input from user as state S
    find the maximum score node connected to S
    set S as this node and recurse
```

The program will take input from user and find the maximum score connected node to it (that will be the program's turn) and display it as its turn. The game will continue until a terminal state occurs.

Some relevant data from the code results:

Total states = 549946

If computer starts then:
  number of win states = 131184
  number of lose states= = 77904
  number of tie states = 46080

If player starts:
  number of win states = 77904
  number of lose states= = 131184
  number of tie states = 46080

Time complexity is O(number of states) which is less than 10^6 in tic-tac-toe