Business analyst at mind, developer and designer at heart. http://abdolsa.com/
Feb 18 · 8 min read

How to make your Tic Tac Toe game unbeatable by using the minimax algorithm

I struggled for hours scrolling through tutorials, watching videos, and banging my head on the desk trying to build an unbeatable Tic Tac Toe game with a reliable Artificial Intelligence. So if you are going through a similar journey, I would like to introduce you to the Minimax algorithm.

Like a professional chess player, this algorithm sees a few steps ahead and puts itself in the shoes of its opponent. It keeps playing ahead until it reaches a terminal arrangement of the board (**terminal state**) resulting in a tie, a win, or a loss. Once in a terminal state, the AI will assign an arbitrary positive score (+10) for a win, a negative score (-10) for a loss, or a neutral score (0) for a tie.

At the same time, the algorithm evaluates the moves that lead to a terminal state based on the players' turn. It will choose the move with maximum score when it is the AI's turn and choose the move with the minimum score when it is the human player's turn. Using this strategy, Minimax avoids losing to the human player.

Try it for yourself in the following game.

A Minimax algorithm can be best defined as a recursive function that does the following things:

> return a value if a terminal state is found (+10, 0, -10)
>
> go through available spots on the board
>
> call the minimax function on each available spot (recursion)
>
> evaluate returning values from function calls
>
> and return the best value

If you are new to the concept of recursion, I recommend watching this video from Harvard's CS50.

To completely grasp the Minimax's thought process, let's implement it in code and see it in action in the following two sections.

Minimax in Code

For this tutorial you will be working on a near end state of the game which is shown in figure 2 below. Since minimax evaluates every state of the game (hundreds of thousands), a near end state allows you to follow up with minimax's recursive calls easier(9).

For the following figure, assume the AI is X and the human player is O.

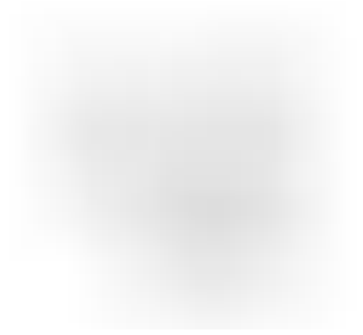


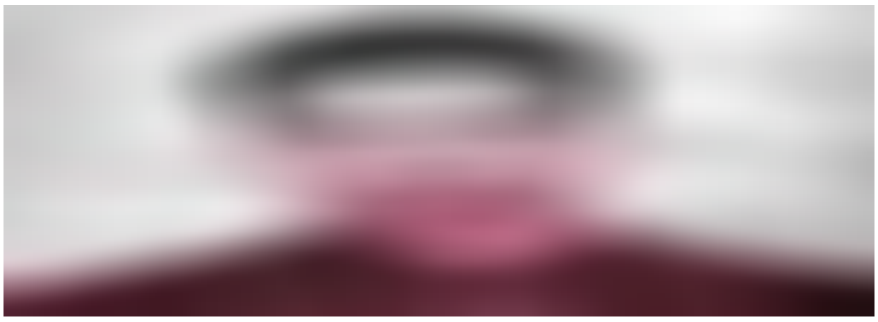
figure 2 sample of game state

To work with the Ti Tac Toe board easier, you should define it as an array with 9 items. Each item will have its index as a value. This will come handy later on. Because the above board is already populated with some X and Y moves, let us define the board with the X and Y moves already in it (*origBoard*).

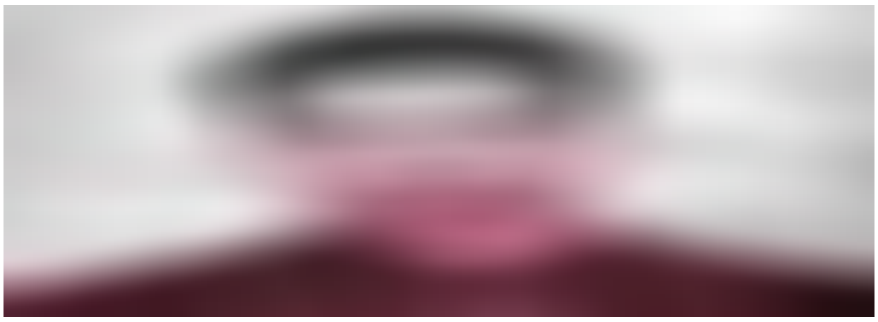
```
var origBoard = ["O",1,"X","X",4,"X",6,"O","O"];
```

Then declare *aiPlayer* and *huPlayer* variables and set them to "X" and "O" respectively.

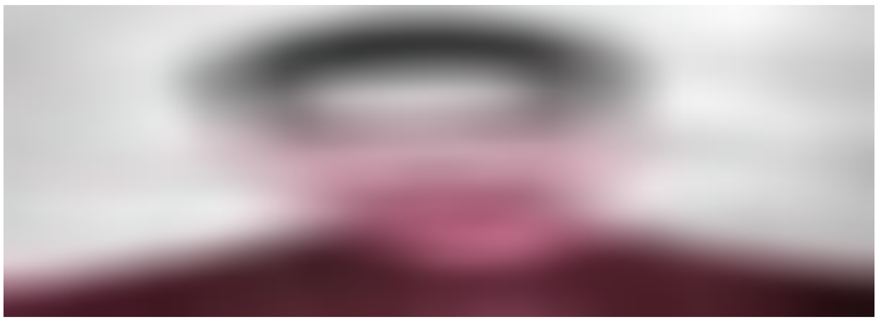
Additionally, you need a function that looks for winning combinations and returns true if it finds one, and a function that lists the indexes of available spots in the board.



Now let's dive into the good parts by defining the Minimax function with two arguments *newBoard* and *player*. Then, you need to find the indexes of the available spots in the board and set them to a variable called *availSpots*.



Also, you need to check for terminal states and return a value accordingly. If O wins you should return -10, if X wins you should return +10. In addition, if the length of the *availableSpots* array is zero, that means there is no more room to play, the game has resulted in a tie, and you should return zero.



Next, you need to collect the scores from each of the empty spots to evaluate later. Therefore, make an array called *moves* and loop through empty spots while collecting each move's index and score in an object called *move*.

Then, set the index number of the empty spot that was stored as a number in the *origBoard* to the index property of the *move* object. Later, set the empty spot on the *newboard* to the current player and call the *minimax* function with other player and the newly changed *newboard*. Next, you should store the object resulted from the *minimax* function call that includes a *score* property to the *score* property of the *move* object.

> If the minimax function does not find a terminal state, it keeps recursively going level by level deeper into the game. This recursion happens until it reaches a terminal state and returns a score one level up.

Finally, Minimax resets *newBoard* to what it was before and pushes the *move* object to the *moves* array.

```
// an array to collect all the objects
  var moves = [];

  // loop through available spots
  for (var i = 0; i < availSpots.length; i++){
    //create an object for each and store the index of that
    var move = {};
        move.index = newBoard[availSpots[i]];

    // set the empty spot to the current player
    newBoard[availSpots[i]] = player;

    /*collect the score resulted from calling minimax
      on the opponent of the current player*/
    if (player == aiPlayer){
      var result = minimax(newBoard, huPlayer);
      move.score = result.score;
    }
    else{
```

Then, the minimax algorithm needs to evaluate the best *move* in the *moves* array. It should choose the *move* with the highest score when AI is playing and the *move* with the lowest score when the human is playing. Therefore, If the *player* is *aiPlayer*, it sets a variable called *bestScore* to a very low number and loops through the *moves* array, if a *move* has a higher *score* than *bestScore*, the algorithm stores that *move*.

In case there are moves with similar score, only the first one will be stored.

The same evaluation process happens when *player* is *huPlayer*, but this time *bestScore* would be set to a high number and Minimax looks for a move with the lowest score to store.

At the end, Minimax returns the object stored in *bestMove*.

```
// if it is the computer's turn loop over the moves and cho
  var bestMove;
  if(player === aiPlayer){
    var bestScore = -10000;
    for(var i = 0; i < moves.length; i++){
      if(moves[i].score > bestScore){
        bestScore = moves[i].score;
        bestMove = i;
      }
    }
  }else{

// else loop over the moves and choose the move with the lo
    var bestScore = 10000;
    for(var i = 0; i < moves.length; i++){
      if(moves[i].score < bestScore){
        bestScore = moves[i].score;
```

> That is it for the minimax function. :) you can find the above algorithm on github and codepen. Play around with different boards and check the results in the console.

In the next section, let's go over the code line by line to better understand how the minimax function behaves given the board shown in figure 2.

Minimax in action

Using the following figure, let's follow the algorithm's function calls (**FC**) one by one.

Note: In figure 3, large numbers represent each function call and levels refer to how many steps ahead of the game the algorithm is playing.

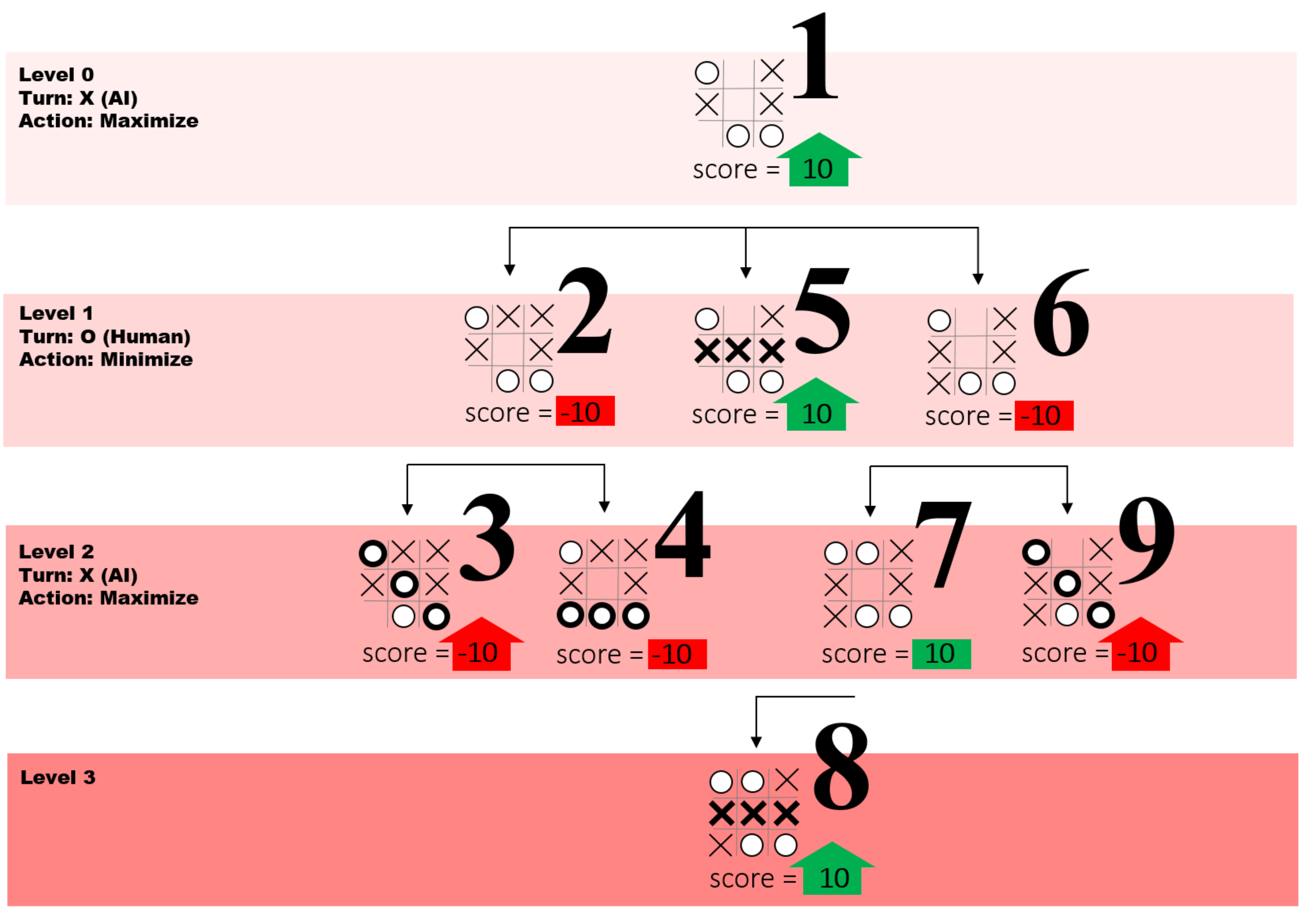


Figure 3 Minimax function call by function call

1.*origBoard* and *aiPlayer* is fed to the algorithm. The algorithm makes a list of the three empty spots it finds, checks for terminal states, and loops through every empty spot starting from the first one. Then, it changes the *newBoard* by placing the *aiPlayer* in the first empty spot. After that, it calls itself with *newBoard* and the *huPlayer* and waits for the FC to return a value.

2. While the first FC is still running, the second one starts by making a list of the two empty spots it finds, checks for terminal states, and loops through the empty spot starting from the first one. Then, it changes the *newBoard* by placing the *huPlayer* in the first empty spot. After that it calls itself with *newBoard* and the *aiPlayer* and waits for the FC to return a value.

3. Finally the algorithm makes a list of the empty spots, and finds a win for the human player after checking for terminal states. Therefore, it returns an object with a score property and value of -10.

> Since the second FC listed two empty spots, Minimax changes the *newBoard* by placing *huPlayer* in the second empty spot. Then, it calls itself with the new board and the *aiPlayer*.

4. The algorithm makes a list of the empty spots, and finds a win for the human player after checking for terminal states. Therefore, it returns an object with a score property and value of -10.

> On the second FC, the algorithm collects the values coming from lower levels (3rd and 4th FC). Since *huPlayer*'s turn resulted in the two values, the algorithm chooses the lowest of the two values. Because both of the values are similar, it chooses the first one and returns it up to the first FC.

> At this point the first FC has evaluated the score of moving *aiPlayer* in the first empty spot. Next, it changes the *newBoard* by placing *aiPlayer* in the second empty spot. Then, it calls itself with the *newBoard* and the *huPlayer*.

5. On the fifth FC, The algorithm makes a list of the empty spots, and finds a win for the human player after checking for terminal states. Therefore, it returns an object with a score property and value of +10.

> After that, the first FC moves on by changing the *newBoard* and placing *aiPlayer* in the third empty spot. Then, it calls itself with the new board and the *huPlayer*.

6. The 6th FC starts by making a list of two empty spots it finds, checks for terminal states, and loops through the two empty spots starting from the first one. Then, it changes the *newBoard* by placing the *huPlayer* in the first empty spot. After that, it calls itself with *newBoard* and the *aiPlayer* and waits for the FC to return a score.

7. Now the algorithm is two level deep into the recursion. It makes a list of the one empty spot it finds, checks for terminal states, and changes the *newBoard* by placing the *aiPlayer* in the empty spot. After that, it calls itself with *newBoard* and the *huPlayer* and waits for the FC to return a score so it can evaluate it.

8. On the 8th FC, the algorithm makes an empty list of empty spots, and finds a win for the *aiPlayer* after checking for terminal states. Therefore, it returns an object with score property and value of +10 one level up (7th FC).

> The 7th FC only received one positive value from lower levels (8th FC). Because *aiPlayer's turn* resulted in that value, the algorithm needs to return the highest value it has received from lower levels. Therefore, it returns its only positive value (+10) one level up (6th FC).

> Since the 6th FC listed two empty spots, Minimax changes *newBoard* by placing *huPlayer* in the second empty spot. Then, calls itself with the new board and the *aiPlayer*.

9. Next, the algorithm makes a list of the empty spots, and finds a win for the *aiPlayer* after checking for terminal states. Therefore, it returns an object with score properties and value of +10.

> At this point, the 6 FC has to choose between the score (+10)that was sent up from the 7th FC (returned originally from from the 8 FC) and the score (-10) returned from the 9th FC. Since *huPlayer*'s turn resulted in those two returned values, the algorithm finds the minimum score (-10) and returns it upwards as an object containing score and index properties.

> Finally, all three branches of the first FC have been evaluated (-10, +10, -10). But because aiPlayer's turn resulted in those values, the algorithm returns an object containing the highest score (+10) and its index (4).

In the above scenario, Minimax concludes that moving the X to the middle of the board results in the best outcome. :)

The End!

By now you should be able to understand the logic behind the Minimax algorithm. Using this logic try to implement a Minimax algorithm yourself or find the above sample on github or codepen and optimize it.

Thanks for reading! If you liked this story, please recommend it by clicking the ❤️ button on the side and sharing it on social media.

Special thanks to Tuba Yilmaz, Rick McGavin, and Javid Askerov for reviewing this article.