Minimax and Alpha/Beta Pruning for Tic-Tac-Toe

Frederik Blem - cph-fb114@cphbusiness.dk
Simon Schønberg Bojesen - cph-sb339@cphbusiness.dk

May 6, 2021

Copenhagen Business Academy
Supervisor: Martin Vestergaard

Abstract

This is where an abstract will be in 4 sentences! What is the problem? Why is the problem interesting? What is the solution? What are the implications of the solution?

Contents

1 Introduction **3**

2 Minimax Algorithm **4**
- 2.1 Introduction to Minimax . . . 4
- 2.2 Explanation of Minimax . . . 4
  - 2.2.1 About Minimax . . . 4
  - 2.2.2 Why is it called Minimax . . . 4
- 2.3 Implementation of Minimax . . . 6

3 Alpha/Beta Pruning **9**
- 3.1 Introduction to Alpha/Beta Pruning . . . 9
- 3.2 Explanation of Alpha/Beta Pruning . . . 9
- 3.3 Implementation of Alpha/Beta Pruning . . . 10

4 Discussion **12**
- 4.1 Compare results . . . 12
- 4.2 Reflection . . . 12
- 4.3 Conclusion . . . 12

List of Figures

1 An example of Alpha Beta Pruning . . . 9

3 An example of branches being pruned . . . 9

List of Tables

1 Introduction

This paper is about the Minimax algorithm and Alpha/Beta pruning, which make up the core of our former assignment "Tic-Tac-Toe using Minimax" for the subject Data Science. We chose to cover this project over other projects because the topics were among the more interesting and we wanted to research these further.
One of the objective of this paper would be to shed some light on the Minimax Algorithm, what it is good for, why this is interesting and different ways to implement it. Another objective is showing how to optimize this algorithm further by implementing Alpha/Beta pruning.
We will do this by describing the logic behind the topics and showing implementations from the Tic-Tac-Toe assignment, that we will be modifying for the purpose of this paper.

2 Minimax Algorithm

2.1 Introduction to Minimax

The Minimax algorithm is a recursive algorithm which is used in turn-based games like chess and tic-tac-toe, to make the AI player consider all possible moves and choose the best one.

2.2 Explanation of Minimax

2.2.1 About Minimax

First, it is important to understand the meaning behind recursive code. When code is recursive, it simply means that you are writing a function that calls itself with updated arguments until it reaches some condition the function wants to return. In the Minimax algorithm we want to make the function call itself to keep looking at more possible future moves, and ultimately decide which move is the best move to make on the AI's current board.

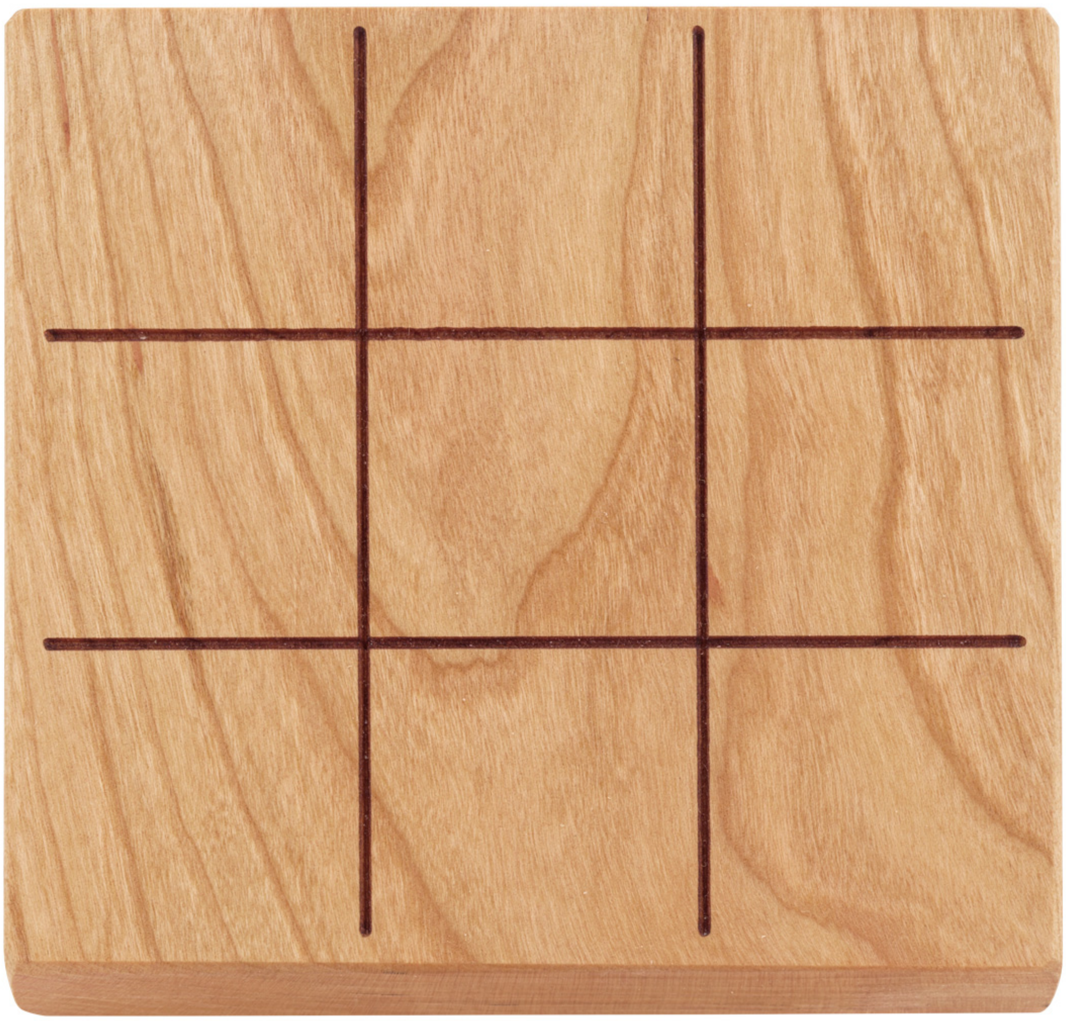
2.2.2 Why is it called Minimax

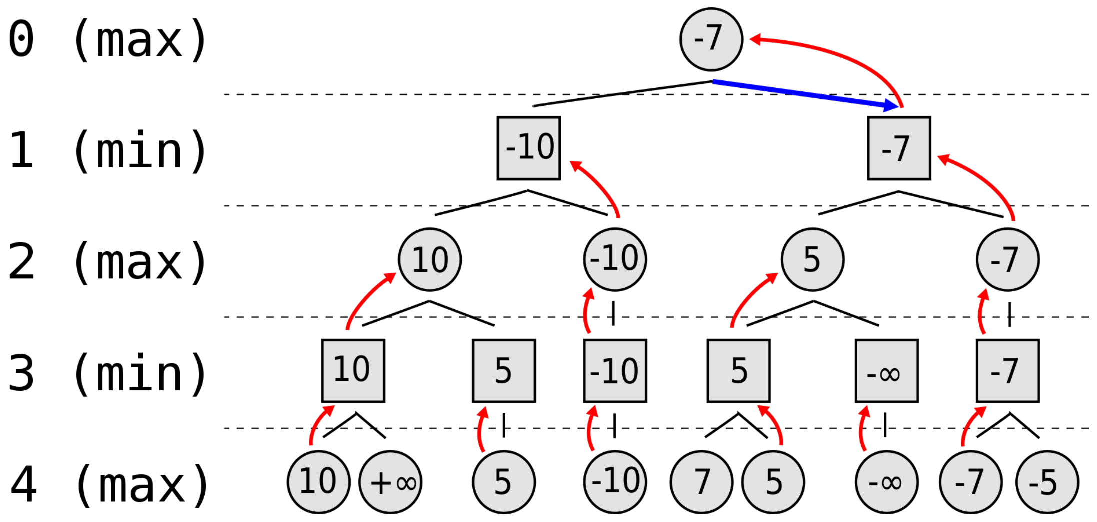
The idea behind minimax is that the algorithm expects the game to be a turn-based 2-player game, and the algorithm defines these 2 players as the minimizer and the maximizer. This simply means that the algorithm always expects the minimizer to pick the move with the lowest score, and the maximizer to do the opposite. Here is an example: Let us say the maximizer is the human player and the minimizer is the AI player in this scenario.

Look at this Tic-Tac-Toe board:

In this scenario the AI player goes first as "X" and tries to figure out the most rational move of the 9 moves possible. It does this by predicting what the opposing player, maximizer would then do on his next move and so on until it has found the fastest way to win for that one move. Basically, the algorithm runs a simulated game on each of the current board's possible moves, against the maximizer, and when it is done with one move it backtracks back to the current board with that moves score value, to run a new simulation on the next move. When it has simulated how to win by all the possible moves it will then make the move that will help it win the fastest.

Let us see another example:





If you look at the tree above it shows an example of how the minimax works. In this scenario the maximizer goes first, and he has 2 possible moves. He would therefore explore the first move first, marked -10. At this point the algorithm has not simulated this move yet and therefore does not know that value yet. So first he just tries the moves leftmost in the tree until he hits the depth 4 which is the bottom in this scenario. He would end up with a value of 10 which would be good for the maximizer. But we need to take the minimizer into account, so we backtrack once to see the other move which is +infinity, which we can say for sure the minimizer would never pick that route, which means we keep the score 10 in mind while we backtrack to the next point where the minimizer has a move. As is seen this route only gives both players one choice on each of their respective turns. The last move has a -10 score so we backtrack with that score once again and end up 10 and -10 on the 2 moves the minimizer can make. If the maximizer chooses this route the point value of the move would therefore be -10 as the minimizer always goes for the lesser score.
The algorithm then does the same thing for the other move and finds that this move will serve him better as it will yield -7 move value which is still bad for the maximizer but less bad then a -10 score.

2.3 Implementation of Minimax

Listing 1: python example

```
    # AI makes a move based on minmax algorithmic search for the
       most rational move to make
    def ai_move(board):
```

```python
        best_move = board
        best_value = float('-inf')
        for move in all_possible_moves(board, AI):
            #print(move)
            move_val = minimax(move, 0, False)
            #print(move_val)
            if move_val > best_value or best_value == None:
                best_move = move
                best_value = move_val
        return best_move
```

The function ai_move gets called when its the AI's turn to play. The function generates a list of all possible moves from the current board, which the function recieves from the arguments. We then iterate over this list, to run the minimax recursion on each of these possible moves. When we get the result move_val back for the current move, we check to see if this value is higher or equal to the stored best_value. If it is, we then override that stored value with the new best value.

Listing 2: python example

```python
    def minimax(board, depth, isMax):
    score = evaluate(board)

    if score == 10:
        return score
    if score == -10:
        return score
    if board.count(EMPTY) == 0:
        return 0

    if isMax:
        best = float('-inf')
        for move in all_possible_moves(board, AI):
            best = max(best, minimax(move, depth + 1, not isMax))
        return best
    else:
        best = float('inf')
        for move in all_possible_moves(board, HUMAN):
            best = min(best, minimax(move, depth + 1, not isMax))
        return best
```

This is the implementation made in Simon's Original Jupyter Notebook. The function always start by checking the board with another function called evaluate. This function checks for win conditions and returns either 10 if the AI has a winning board and likewise -10 for the Human player. If noone has won yet it returns 0.

Next we check the score we get back, to find out if we should end the recursion for this move. We do this if the score is -10, 10 or 0, and as you can image the move will not be considered as best if the score is -10 as the AI does not want the human player to win.

The next part of the implementation is the interesting part. First we check to see if the current simulated move is the maximizer (Human) or the minimizer (AI). Depending on the result of this if we set up the recursion quite oppositely. As can be seen we start a "best" variable at the worst value for the player. -infinity for the maximizer and infinity for the minimizer. We do this to ensure that there is a value to overwrite on the very first run of the algorithm. We then create a list of possible for this current simulated board, and iterates over it. This is where the recursion happens. We call the minimax function within it self on the current move of the list. This ensures that we hit all possible moves for this new current board as well, because every time the function returns "best" it backtracks to check the next move. Remember the tree example explained in **??**.

Figure 1: An example of Alpha Beta Pruning

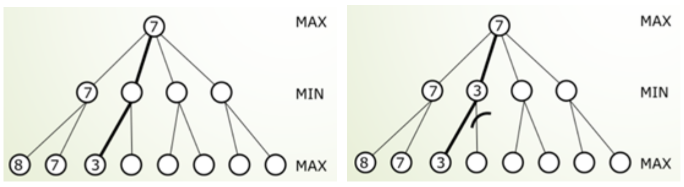
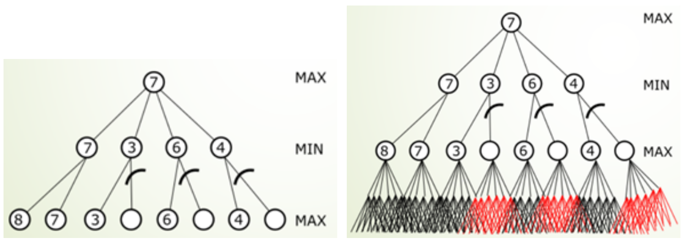


Figure 3: An example of branches being pruned



3 Alpha/Beta Pruning

3.1 Introduction to Alpha/Beta Pruning

Alpha/Beta pruning is the way by which the Minimax Algorithm is made more efficient. This happens by stopping consideration of possible moves, that will never be better than previously considered moves.

3.2 Explanation of Alpha/Beta Pruning

Alpha Beta Pruning is an optimization method for the Minimax algorithm that eliminates branches or entire subtrees. It is so named because it introduces the two variables or values Alpha and Beta into the Minimax algorithm. The variable Alpha is set to be the biggest value the Maximizer can guarantee at the current level being explored or above it. The variable Beta is likewise set to be the smallest value the Minimizer can guarantee.

Figure 1 shows how first the leftmost branches are explored and the minimizer picks "7" which is stored in the variable Alpha. After that the second leftmost branch is explored and the Minimizer is presented with a choice that is smaller than 7. Given that there is a choice smaller than Alpha present already, and it is the Minimizer's turn to choose, meaning that it will undoubtedly have a chance to pick a value smaller than Alpha, it would not be worth it for the Maximizer to explore the rest of the branch and the branch is pruned as seen in figure 1.

Alpha Beta Pruning can potentially prune a lot of branches and thus make the examination of choices or moves through Minimax considerably faster than if it simply went through every single branch.

3.3 Implementation of Alpha/Beta Pruning

In this section we will show you our changes to the minimax function and more, which we added to better see the results of the pruning. First thing we did was to add a counter to the minimax function. We did this to both V3 and V4 versions of the code. This counter increments every time we look at a new future move. The idea was to show a count of how many moves the AI had considered before making the most rational choice. This way we could compare the number of considered moves on both versions of the code: one without pruning and one where we implemented it.

Listing 3: python example

```python
def minimax(board, depth, alpha, beta, isMax):
    global count
    score = evaluate(board)
    count += 1
    if score == 10:
        return score
    if score == -10:
        return score
    if board.count(EMPTY) == 0:
        return 0

    if isMax:
        maxEval = float('-inf')
        for move in all_possible_moves(board, AI):
            evaluation = minimax(move, depth + 1, alpha,
                beta, not isMax)
            maxEval = max(maxEval, evaluation)
            alpha = max(alpha, evaluation)
            if beta <= alpha:
                break
        return maxEval
    else:
        minEval = float('inf')
        for move in all_possible_moves(board, HUMAN):
            evaluation = minimax(move, depth + 1, alpha,
                beta, not isMax)
            minEval = min(minEval, evaluation)
            beta = min(beta, evaluation)
            if beta <= alpha:
                break
        return minEval
```

To implement Alpha Beta Pruning we had to add 2 more parameters to the function: alpha and beta. These start with values -infinity and infinity. The recursive call was then updated to have include arguments. For the maximizer we set the alpha value to whichever is greater between the current alpha value and the evaluation. Likewise it is done for the minimizer between which is lesser of beta's current value and the evaluation. This means that every time the minimax function is run, we update the alpha value if it is the maximizing players turn. Likewise we update the beta value on the minimizing players turn. No matter whose turn it is we check to see if beta is less then or equal to alpha, and if this is true, we prune the rest of the moves.

4 Discussion

4.1 Compare results

This subsection will be used to show the results of the pruning.

In the notebook without pruning (V3) on turn 1 the AI considered 549945 different moves. In the notebook with pruning (V4) on turn 1 the AI considered 30709 different moves.

This means that on turn one we managed to prune away around 93% of the moves it would consider without pruning.

On turn 3 this went from 7331 down to 1519 which is again a significant decrease of around 80% These prunings of course become less and less as the game has less possible moves, and at turn 7 it would no longer prune anything.

These findings can be seen better by running files MinimaxV3.ipynb and MinimaxV4.ipynb in the code folder of this project.

4.2 Reflection

4.3 Conclusion