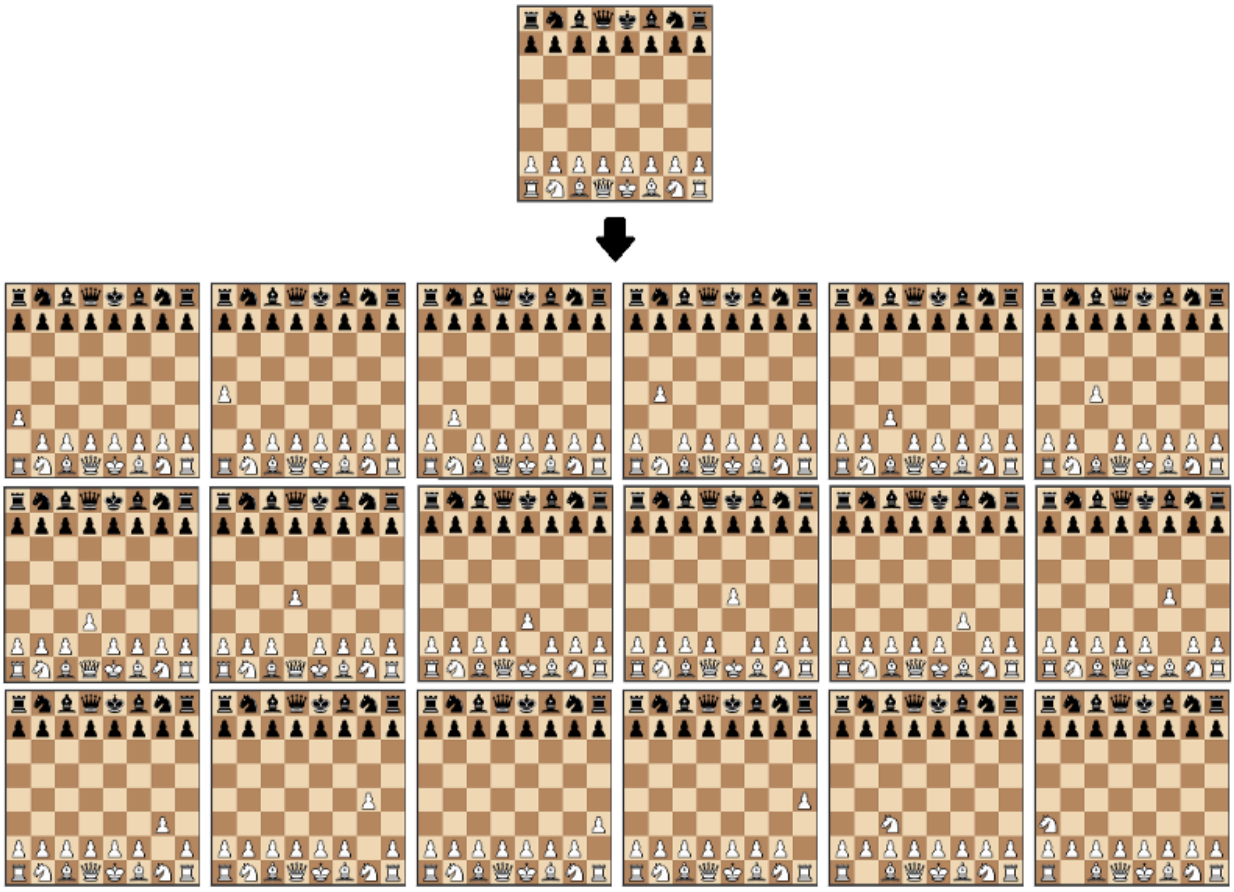
- move-generation
- board evaluation
- minimax
- and alpha beta pruning.

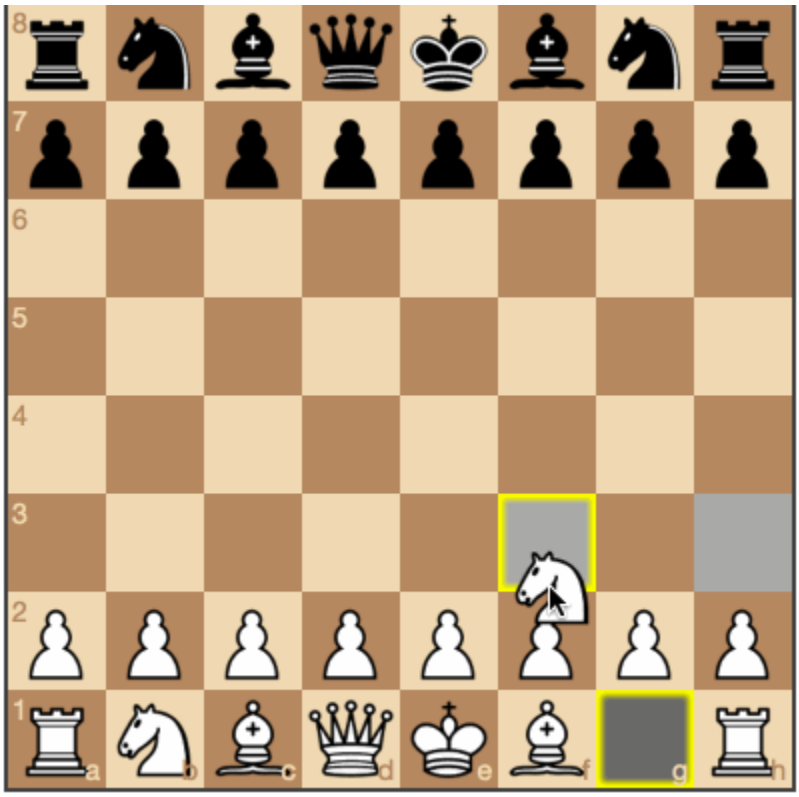
Step 1: Move generation and board visualization

We'll use the chess.js library for move generation, and chessboard.js for visualizing the board. The move generation library basically implements all the rules of chess. Based on this, we can calculate all legal moves for a given board state.



Using these libraries will help us focus only on the most interesting task: creating the algorithm that finds the best move. We'll start by creating a function that just returns a random move from all of the possible moves:

Although this algorithm isn't a very solid chess player, it's a good starting point, as we can actually play against it:



Step 2 : Position evaluation

Now let's try to understand which side is stronger in a certain position. The simplest way to achieve this is to count the relative strength of the pieces on the board using the following table:

With the evaluation function, we're able to create an algorithm that chooses the move that gives the highest evaluation:

The only tangible improvement is that our algorithm will now capture a piece if it can.

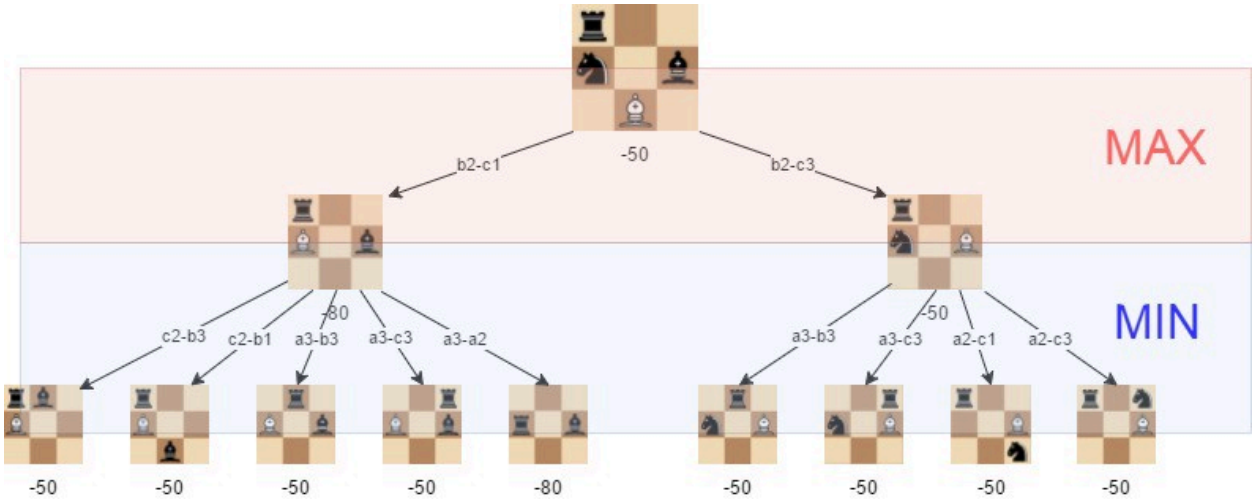
Step 3: Search tree using Minimax

Next we're going to create a search tree from which the algorithm can chose the best move. This is done by using the Minimax algorithm.

In this algorithm, the recursive tree of all possible moves is explored to a given depth, and the position is evaluated at the ending "leaves" of the tree.

After that, we return either the smallest or the largest value of the child to the parent node, depending on whether it's a white or

black to move. (That is, we try to either minimize or maximize the outcome at each level.)



A visualization of the minimax algorithm in an artificial position. The best move for white is **b2-c3**, because we can guarantee that we can get to a position where the evaluation is **-50**

With minimax in place, our algorithm is starting to understand some basic tactics of chess:

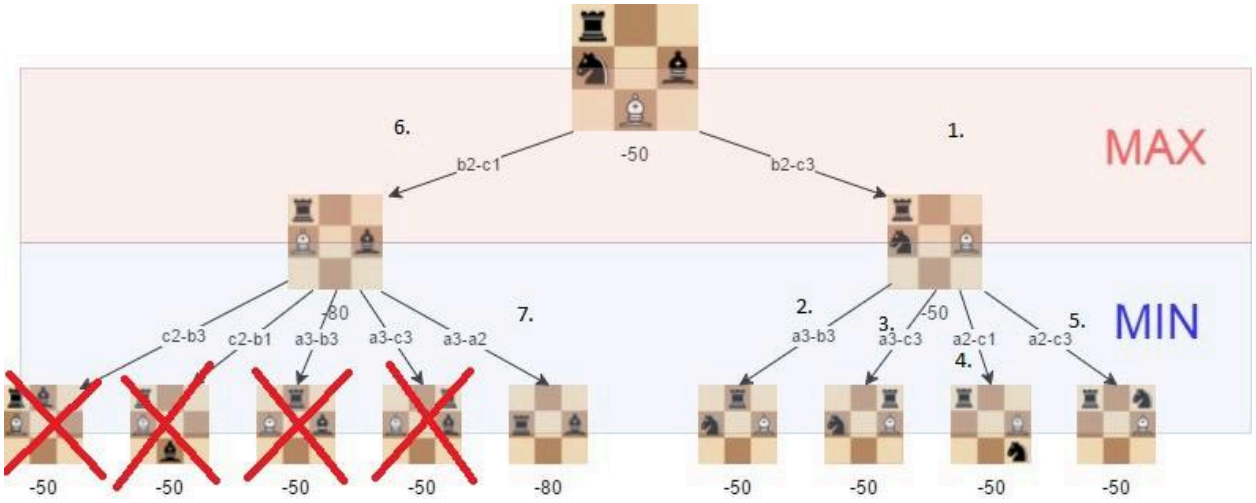
Step 4: Alpha-beta pruning

Alpha-beta pruning is an optimization method to the minimax algorithm that allows us to disregard some branches in the search tree. This helps us evaluate the minimax search tree much deeper, while using the same resources.

The alpha-beta pruning is based on the situation where we can stop evaluating a part of the search tree if we find a move that leads to a worse situation than a previously discovered move.

The alpha-beta pruning does not influence the outcome of the minimax algorithm — it only makes it faster.

The alpha-beta algorithm also is more efficient if we happen to visit **first** those paths that lead to good moves.



The positions we do not need to explore if alpha-beta pruning isused and the tree is visited in the described order.

With alpha-beta, we get a significant boost to the minimax algorithm, as is shown in the following example:

	Minimax	Minimax with alpha-beta
Positions	879750	61721

The number of positions that are required to evaluate if we want to perform a search with depth of 4 and the “root” position is the one that is shown.

Step 5: Improved evaluation function

The initial evaluation function is quite naive as we only count the material that is found on the board. To improve this, we add to the evaluation a factor that takes in account the position of the pieces. For example, a knight on the center of the board is better (because it has more options and is thus more active) than a knight on the edge of the board.

We’ll use a slightly adjusted version of piece-square tables that are originally described in the chess-programming-wiki.

```
[ -3.0, -4.0, -4.0, -5.0, -5.0, -4.0, -4.0, -3.0],
[ -3.0, -4.0, -4.0, -5.0, -5.0, -4.0, -4.0, -3.0],
[ -3.0, -4.0, -4.0, -5.0, -5.0, -4.0, -4.0, -3.0],
[ -3.0, -4.0, -4.0, -5.0, -5.0, -4.0, -4.0, -3.0],
[ -2.0, -3.0, -3.0, -4.0, -4.0, -3.0, -3.0, -2.0],
[ -1.0, -2.0, -2.0, -2.0, -2.0, -2.0, -2.0, -1.0],
[  2.0,  2.0,  0.0,  0.0,  0.0,  0.0,  2.0,  2.0 ],
[  2.0,  3.0,  1.0,  0.0,  0.0,  1.0,  3.0,  2.0 ]
```

```
[ -2.0, -1.0, -1.0, -0.5, -0.5, -1.0, -1.0, -2.0],
[ -1.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0, -1.0],
[ -1.0,  0.0,  0.5,  0.5,  0.5,  0.5,  0.0, -1.0],
[ -0.5,  0.0,  0.5,  0.5,  0.5,  0.5,  0.0, -0.5],
[  0.0,  0.0,  0.5,  0.5,  0.5,  0.5,  0.0, -0.5],
[ -1.0,  0.5,  0.5,  0.5,  0.5,  0.5,  0.0, -1.0],
[ -1.0,  0.0,  0.5,  0.0,  0.0,  0.0,  0.0, -1.0],
[ -2.0, -1.0, -1.0, -0.5, -0.5, -1.0, -1.0, -2.0]
```

```
[  0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0],
[  0.5,  1.0,  1.0,  1.0,  1.0,  1.0,  1.0,  0.5],
[ -0.5,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0, -0.5],
[ -0.5,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0, -0.5],
[ -0.5,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0, -0.5],
[ -0.5,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0, -0.5],
[ -0.5,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0, -0.5],
[  0.0,   0.0, 0.0,  0.5,  0.5,  0.0,  0.0,  0.0]
```

```
[ -2.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -2.0],
[ -1.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0, -1.0],
[ -1.0,  0.0,  0.5,  1.0,  1.0,  0.5,  0.0, -1.0],
[ -1.0,  0.5,  0.5,  1.0,  1.0,  0.5,  0.5, -1.0],
[ -1.0,  0.0,  1.0,  1.0,  1.0,  1.0,  0.0, -1.0],
[ -1.0,  1.0,  1.0,  1.0,  1.0,  1.0,  1.0, -1.0],
[ -1.0,  0.5,  0.0,  0.0,  0.0,  0.0,  0.5, -1.0],
[ -2.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -2.0]
```

```
[-5.0, -4.0, -3.0, -3.0, -3.0, -3.0, -4.0, -5.0],
[-4.0, -2.0,  0.0,  0.0,  0.0,  0.0, -2.0, -4.0],
[-3.0,  0.0,  1.0,  1.5,  1.5,  1.0,  0.0, -3.0],
[-3.0,  0.5,  1.5,  2.0,  2.0,  1.5,  0.5, -3.0],
[-3.0,  0.0,  1.5,  2.0,  2.0,  1.5,  0.0, -3.0],
[-3.0,  0.5,  1.0,  1.5,  1.5,  1.0,  0.5, -3.0],
[-4.0, -2.0,  0.0,  0.5,  0.5,  0.0, -2.0, -4.0],
[-5.0, -4.0, -3.0, -3.0, -3.0, -3.0, -4.0, -5.0]
```

```
[0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0],
[5.0,  5.0,  5.0,  5.0,  5.0,  5.0,  5.0,  5.0],
[1.0,  1.0,  2.0,  3.0,  3.0,  2.0,  1.0,  1.0],
[0.5,  0.5,  1.0,  2.5,  2.5,  1.0,  0.5,  0.5],
[0.0,  0.0,  0.0,  2.0,  2.0,  0.0,  0.0,  0.0],
[0.5, -0.5, -1.0,  0.0,  0.0, -1.0, -0.5,  0.5],
[0.5,  1.0, 1.0,  -2.0, -2.0,  1.0,  1.0,  0.5],
[0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0]
```

The visualized piece-square tables visualized. We can decrease or increase the evaluation, depending on the location of the piece.

With the following improvement, we start to get an algorithm that plays some “decent” chess, at least from the viewpoint of a casual player:

Conclusions

The strength of even a simple chess-playing algorithm is that it doesn't make stupid mistakes. This said, it still lacks strategic understanding.

With the methods I introduced here, we've been able to program a chess-playing-algorithm that can play basic chess. The "AI-part" (move-generation excluded) of the final algorithm is just 200 lines of code, meaning the basic concepts are quite simple to implement. You can check out the final version is on Chess AI (Github)

Some further improvements we could make to the algorithm would be for instance:

- move-ordering
- faster move generation
- and end-game specific evaluation.