Programming a Chess AI

Matura Thesis by Oliver Graf
Kantonsschule Hohe Promenade Zürich
Supervised by Clemens Pohle and Stefan Müller
Informatics
2020

My Goals

Being a passionate chess player myself, I've always wondered how computers miraculously calculate the best moves and nowadays beat human players with ease. Thus, I set as my goal to code a chess AI from scratch that is able to defeat me as a human player.

For my chess engine, I followed the following guidelines:

- It has to be possible to understand my code from an outsider's perspective.
- The chess programming concepts used should be so close to an actual chess board as possible without sacrificing performance.
- Anyone should be able to play against *Mockfish*. To ensure that, I created a GUI application which can be downloaded with the QR code at the bottom right of this poster.

How does *Mockfish* work?

A chess engine needs several parts that have to work together seamlessly. The most important ones are the following:

- **Square representation** In order to store a single square with only one byte, I gave every bit in a byte a distinct role (see Table 1). That makes it easy for the computer to read and write information.
- **Board representation** After knowing how to store a single square, there has to be a way to store an entire board. There are several different ways which have different strengths and weaknesses (see Table 2).
- **Move generation** A chess engine also requires a function which can create all legal moves from a certain position. The shape of this function heavily relies on the board representation design.
- **Heuristic evaluation** Another vital function is the one that estimates the value of a position, whereas a positive number means that white has an advantage while a negative number means that black has an edge. It's not possible to create a perfect function – otherwise, chess programming would be much easier – but it has to be able to guess the value of a position more or less accurately (see Image 1).
- **Search tree** Because the heuristic evaluator is never precise enough, it makes sense to postpone the evaluation to some moves after the position that you actually want to evaluate. This can be done using a so-called search tree which contains all positions starting from the root (the first positions) and then moves downwards up to a certain level (see Image 2).
- **α-ß pruning** In order to find the best result when analyzing a search tree, the best bet is the so-called α-ß algorithm. Its advantage is that it doesn't have to evaluate all positions at the bottom of the tree because some don't have an impact on the final result (see Image 2).
- **Sorting** The α-ß algorithm can be improved by sorting the search tree's positions before applying the algorithm. This allows the algorithm to evaluate fewer positions. In Mockfish, I implemented sorting using the heuristic evaluator.
- **Multi-threading** After creating the search tree with only one level, *Mockfish* creates threads that split up the workload of going through the search tree.

1 Byte				
Bits 7-6	Bit 5	Bit 4	Bit 3	Bits 2-0
Unused	Color	En Passant Flag	Castling Flag	Piece Type
always set to 00	0 = black, 1 = white	0 = no en passant right, 1 = en passant right	0 = no castling right, 1 = castling right	000 = empty, 001 = pawn, 010 = knight, etc.

Table 1 (Square Representation)

- `00 1 0 1 110` corresponds to a white king that can still castle
- `00 0 1 0 001` corresponds to a black pawn that can be captured en passant

-	-	-	-	-	-	-	-
50	60	65	65	65	65	60	50
25	45	50	50	50	50	45	35
0	10	20	25	25	20	10	0
-10	-5	5	10	10	5	-5	-10
-15	-10	-5	-10	-10	-5	-10	-15
-20	-15	-15	-20	-20	-15	-15	-20
-	-	-	-	-	-	-	-

Image 1 (Heuristic evaluation)

This image shows the graphical visualization of a so-called lookup table. *Mockfish*'s evaluator uses 18 tables – one for each piece type for each game phase (opening, middle game and endgame). Depending on where each piece stands, it is assigned a different value. All those values are then added up as the placement coefficient of the evaluator's result. The other half of the result is made up by the material coefficient. This one is only important if one player has more pieces than the other.

This specific table shows the table for the pawn in the endgame. This table doesn't have values for the first and last ranks because there, pawns cannot exist. This table encourages pawns to move forward and capture towards the center, where the evaluation's results are generally better.

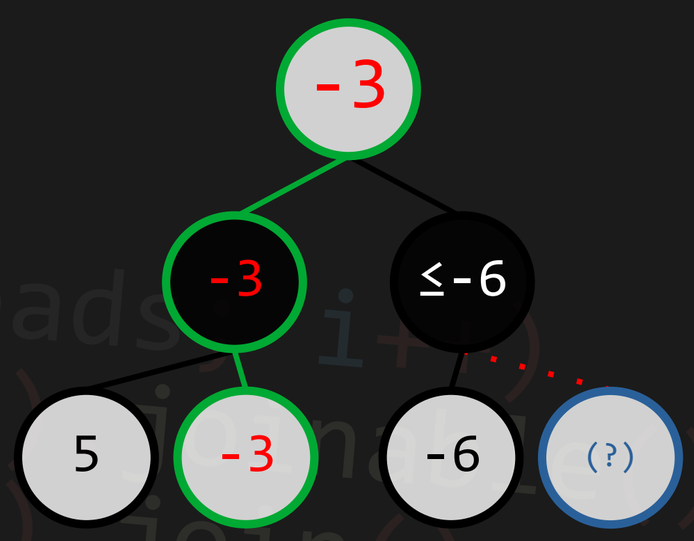

Image 2 (α-ß algorithm)

In order to evaluate a search tree as shown in this image, the evaluator only has to evaluate the positions at the bottom level. After evaluating the first two ones, the value of the first position in which it's black's turn is already known: Since the player with the black pieces wants a result as small as possible, -3 is favorable for her/him. Then, after evaluating the third positions, the result is already known without even looking at the fourth position.

Black would always decide for -6 or a lower result in the second half of the search tree. As white already has a better alternative (-3), the fourth position doesn't have an impact on the result and can thus be omitted.

xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx
xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx
xxxx	0c	02	03	05	0e	03	02	0c	xxxx
xxxx	01	01	01	01	01	01	01	01	xxxx
xxxx									xxxx
xxxx									xxxx
xxxx									xxxx
xxxx									xxxx
xxxx	21	21	21	21	21	21	21	21	xxxx
xxxx	2c	22	23	25	2e	23	22	2c	xxxx
xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx
xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx

Table 2 (Board Representation)

This type of board representation is called the 10×12 array. This type of list stores all squares in a single file, but it adds some flagged squares around the central board. This allows a much more efficient move generation algorithm because all moves that would overshoot the board just land on a flagged square. Thus, the move generation algorithm doesn't have to check whether the target coordinates are legal: It just has to find out whether the target square is flagged.