# Computer Science NEA - Chess Engine

Jonathan Kasongo

March 2 2024

# Contents

# Chapter 1

# Analysis of the problem

## 1.1 Problem Identification

The game of chess has skyrocketed in terms of popularity recently, so much so that half of my school now spend their break times playing each other on chess.com. Chess is a strategy board game with the end goal being to checkmate the opponent's king. [5] This means that capture of the opponent's king is inevitable upon the next move. The game also involves **no** elements of luck and the outcome of the game is soley dependent on the actions of the player. Moreover, the game of chess is known to be very hard to master with many of the best chess *Grandmasters* starting training from the ages of 7-8 [8]. The game of chess has an average of 35 moves [6] per position. This means that if one wants to think three moves ahead of his opponent he must consider $42,875$ positions in total! This is simply not possible for a human, however for a computer this task is something that could be done in less than 1 second. By leveraging the high computational power of modern computers, I aim to write a chess engine that is able to beat an average human chess player 9 times out of 10.

Whilst chess prodigies and Grandmasters dedicate their entire lifes to improving their chess abilities, using high order thinking processes, experience and strategical tactics to play the best move in a position we may simply use a brute-force style of play, in which we consider all legal moves from a given position and simply choose the one that gives the most advantageous position even if our opponent doesn't make any mistakes.

## 1.2 Stakeholders

One of the students at my school who plays chess regularly is John Arco. John Arco is a 17 year old male with a passion for chess. John has a rating of roughly 1000 ELO but wishes to improve to a higher rating and beat all of his classmates. John is also very competitive and wishes to **guarentee** that none of his classmates can beat him. The use of a strong chess engine is one method to ensure that John Arco always beats his classmates and requires little to no effort on his part, all he has to do is replicate the moves played by his opponent on the engine's board and he will simply replicate the computer's moves. [1] Moreover using a chess engine can also be highly educational as we may learn new ideas or moves from the engine that we may have never considered previously. Even Magnus Carlsen has openly said that he has learnt new ideas from chess engines. [4] This means the engine is to serve 2 purposes, the first is ensure that John Arco remains undefeated against his classmates, and the second is to improve John Arco's chess ability by exposing him to new and unique tactics that he wouldn't have thought of otherwise. The construction of a strong chess engine will be able to solve both problems effectively, providing both educational benefits and competitive benefits also.

[1] I do realise that this is considered cheating, however we intend to use this engine completely offline in unrated friendly games against close friends. I do not advocate cheating in any way shape or form.

## 1.3 Research the problem

To begin research it is first nescessary to get a higher level understanding of how a chess engine works. To learn about this topic I made use of resources like https://www.chessprogramming.org/Main_Page and https://www.talkchess.com/forum/index.php, citations will be given accordingly. The following subsections will act to be a brief summary of the research I conducted on understanding how to write a chess engine.

Any chess engine must be comprised of these 3 fundamental components:

- *Legal move generation*
- *Evaluation functions*
- *Searching algorithms*

We will explore each of these components in detail, however if you have never come across the term "bitboards" in relation to chess programming, I strongly encourage you to read the next subsection.

### Bitboards

To understand the following algorithms it is nescessary to have an adequate understanding on ***bitboards***. If you already understand this concept please skip this subsection entirely, otherwise I will provide a brief introduction to the idea here. Some helpful resources can be found here [2].

Every chess engine needs a way to represent the state of the chess board. Bitboards are one such way to represent the state of the chess board with 64 bit integers. Consider the following chess position.

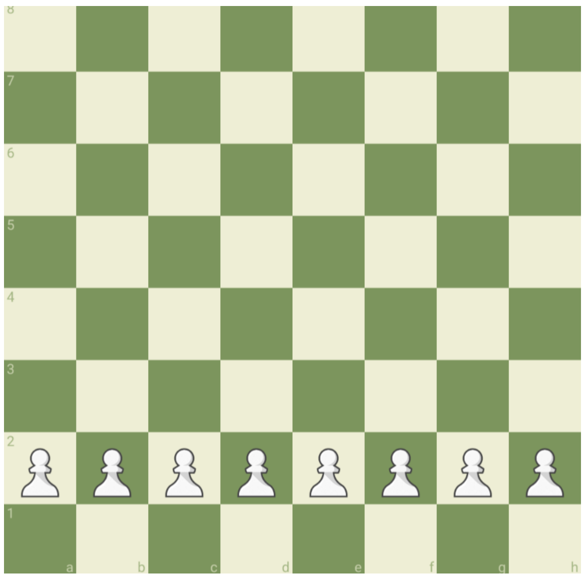

Figure 1.1: Starting position for white pawns

Immediately we may notice that a chess board has dimensions (8 x 8) and 64 squares. Furthermore, each of the squares in figure 1.1 exists in one of these two states: There either is a white pawn on this square or there is not. Does this remind you of a familiar concept in computer science? This innate similarity to the binary numbering system motivates one to consider the use of binary in order to represent a chess board. We can take a 64 bit unsigned integer and have each 0 represent the lack of a piece and similarly have each 1 represent the existance of a piece on this square.

Consider the following code snippet.[2]

```
# For the rest of this paper i64 will refer to the
# unsigned 64 bit integer
i64 = np.uint64
WhitePawn = i64(0b        # Dots represent 0,
                00000000  # . . . . . . . .
                00000000  # . . . . . . . .
                00000000  # . . . . . . . .
                00000000  # . . . . . . . .
                00000000  # . . . . . . . .
                00000000  # . . . . . . . .
                11111111  # 1 1 1 1 1 1 1 1
                00000000) # . . . . . . . .
```

Each bit in the `WhitePawn` variable represents the state of a square like we saw previously, this allows us to store the state of the board with 12, 64 bit numbers (6 piece types in chess, and 2 players). Modern computers typically have register sizes of 64 bits or greater, meaning that we may easily and quickly manipulate these bitboards in order to generate legal moves for a position. We will consider how we may leverage bitboards for legal move generation in the following subsection. [3]

## Legal move generation

Legal move generation is the first step to writing a strong chess engine, in this component we wish to find a way to feed in a position to a computer program and have it output to us all of the possible legal moves available in this position. The study of move generation algorithms in the chess programming world is still very nascant, with one of the newest algorithms being discovered in 2017 [1]. The two algorithms I decided to spend time researching were *Hyperbola quintessence* and *Magic bitboards* because they are the standard accepted algorithms for the top chess engines [7]. Both these algorithms are used to generate moves for sliding pieces [4].

## Magic bitboards

Magic bitboards were discovered in 2006 by Lasse Hansen [3], and was heavily influenced by Gerd Isenberg's *"Kindergarten"* bitboards. Both techniques use the same core idea: we will access moves from a pre-initialised moves array/table instead of calculating the required move set on the fly. Magic bitboards involves the usage of a *perfect* hash function to map all possible board occupancies to all their corresponding move sets. By occupancy I mean some bitboard of all other pieces that are able to block the movement of our sliding piece. For instance consider a rook on the A1 square, if there is another piece on the D1 square the rook will not be able to move past D1 anymore. After we hash our occupancy bitboard, we will use it to index into a pre-calculated attack array that will give us a bitboard of the correct legal moves in $O(1)$ time and space complexity. Examples are often the best way to explain concepts so let's go through a simple one. Let's use our rook that was on A1. It's bitboard will look like this:

[2]The importing of the numpy library has been omitted for clarity.
[3]We assume reverse little endian indexing for our boards throughout.
[4]That is the queen, bishop and rook.

```
WhiteRook = i64(0b        # Dots represent 0,
                00000000  # 8 |. . . . . . . .
                00000000  # 7 |. . . . . . . .
                00000000  # 6 |. . . . . . . .
                00000000  # 5 |. . . . . . . .
                00000000  # 4 |. . . . . . . .
                00000000  # 3 |. . . . . . . .
                00000000  # 2 |. . . . . . . .
                10000000) # 1 |1 . . . . . . .
                          #    _______________
                          #    A B C D E F G H
```

The technique is no doubt fast, we are simply accessing an array, the concern with this technique is rather it's memory consumption.

## Hyperbola quintessence

# List of Figures

# Bibliography

[1] *Black magic bitboards.* URL: https://www.talkchess.com/forum/viewtopic.php?t=64790.

[2] *Chess programming wiki: Bitboards.* URL: https://www.chessprogramming.org/Bitboards.

[3] *Fast(er) bitboard move generator.* URL: https://web.archive.org/web/20221113174017/http://www.open-aurec.com/wbforum/viewtopic.php?t=5015.

[4] Lex Fridman. *Lex Fridman podcast.*

[5] Tim Just and Daniel B. Burg. *U.S. Chess Federation's official rules of chess.* 5th ed. ISBN: 0-8129-3559-4.

[6] Alan Levinovitz. "The Mystery of Go, the Ancient Game That Computers Still Can't Win". In: (2014).

[7] *Stockfish 16.1.* URL: https://stockfishchess.org/.

[8] *Wikipedia: Magnus Carlsen.* URL: https://en.wikipedia.org/wiki/Magnus_Carlsen.