Chess engine with an integrated AI

Fredrik Johansson[1]

Abstract
This report describes the theory and implementation of a chess engine written from scratch in C++ with an integrated Artificial Intelligence(AI). The AI makes decisions by simulating future board states and evaluating the positions using heuristics given by chess experts. The player can choose to play against the AI or let the AI play against itself.
Source code: https://github.com/FredrikErikJohansson/chess-ai

Authors
[1] *Media Technology Student at Linköping University, frejo851@student.liu.se*

Keywords: Chess engine — Alpha-beta pruning — Artificial Intelligence

Contents

1. Introduction

Chess is one of the oldest games in the world, played by millions of people worldwide. Chess involves no hidden information and can thus be described as a deterministic system. This property has made chess a popular testing ground for artificial intelligence researchers. However, chess is still an unsolvable game due to the large number of possible board states which can not be computed with the current available hardware.

Chess bots simulates the board state for an arbitrary number of moves with a search function. For each state an evaluation function calculates a score based on heuristics given by chess experts. In theory, a deeper search would yield a better move and it is therefore essential to design an efficient search function to reach further depths. Since the search function is highly coupled with the chess engine, a chess engine were implemented from scratch during this project.

1.1 Related work

The two most well known chess bots are Stockfish and AlphaZero [1, 2]. Stockfish is an open source chess engine based on traditional artificial intelligence methods with the solely purpose to play chess. Whilst AlphaZero is designed for generic problems using a neural network as a complement to the search function.

1.2 Limitations

Due to the large number of board states, a chess bot based on a neural network is unfeasible for this project as it would require immense computing power and time for the training phase. Instead, the project aims to implement some of the core search features of Stockfish.

2. Theory

This section describes the theory behind the core functionality of the chess engine, including the board representation and move generation. Furthermore, it describes the theory behind the search and evaluation function.

2.1 Board representation

A chess state consists of 64 squares, 6 types of pieces and 2 colors, and can be described by bitboards [3]. A bitboard

is a 64-bit binary representation of a number where each bit represents a square of the board. By combining 12 bitboards, any state can be represented. Bitboards are essential for the performance of the engine since Single Instruction Multiple Data(SIMD) operations can be utilized. The bitboards for the initial state of the board is illustrated in Figure 1.

2.2 Moves

A move in the chess community generally refers to a ply which denotes a half-move, that is a move of one side only. The moves are generated differently for the sliding pieces (rook, bishop, queen) and the non-sliding pieces (pawn, king, knight). The attack-sets of the latter pieces can simply be calculated from the bitboards using bitwise AND and NOT operations. The move-sets for the sliding pieces are more complex since they are highly dependent on the board state. To generate these moves a method called magic bitboards were used [4]. Essentially, all the possible moves for the bishops and the rooks are stored in two separate tables. The tables are of the sizes 64x4096 and 64x512 in respective order, where the first dimension is the position of the attacking piece and the second dimension is a index-key. The index-key can be computed using various methods, the only requirement is that each key is unique. Once the key has been computed, the attack-sets can be derived from the current board state. The queen's attack-set is the bitwise OR operation of the rooks and bishops attack-sets.

The engine handles moves by bitwise operations on the board-state. The post-move board state is computed using bitwise XOR operations where each move is stored in a stack. For each move the possible consequences has to be considered. For instance, if there was an capture, the captured piece's bitboard has to be updated. While unmaking a move, the last inserted element of the stack is removed and all the consequences of that move is undone.

2.3 Evaluation

The bot uses an evaluation function to determine its next move. The eveulation function returns a score which is calculated using material and positional score based on the current board-state. The material score is the sum of the values of the players pieces subtracted by the opponents pieces. Each piece type has a unique value assigned to it, the values can be seen in Table 1.

Table 1. Table of material score

Piece	Score
Pawn	100
Knight	320
Bishop	330
Rook	500
Queen	900
King	20000

The positional score is given by the board squares that the pieces occupy and it is piece type specific since their preferred positions differs. The positional score for the white knights can be seen in Figure 2.

2.4 Minimax

In order to simulate the board states a search function has to be used. Minimax is a decision rule that minimizes the possible loss for a worst case scenario. It is implemented as a depth-first recursive function where the recursive calls alters between the maximum player and the minimum player. The algorithm requires a predetermined maximum depth to be able to execute within a reasonable time frame since the only terminal nodes are checkmates or stalemates. The algorithm can be visualized by a tree structure where each node represents a board state. This results in a branching factor equal to the number of available moves from each node. In chess, the branching factor is commonly said to be 35, and an analysis of 2.5 million games revealed an average of 31 moves per state [5]. This results in the complexity $O(b^d)$, where b is the branching factor and d is the search depth. For instance, the search space for a branching factor of 35 with a depth of 6 equals to 1.8 billion states. It is unfeasible to search through this space within a reasonable time frame with the minimax algorithm, even on a high-end computer.

2.5 Negamax

Negamax is a variant of minimax used to simplify its implementation. Negamax relies on the zero-sum property of a two player game. The zero-sum property states that the gain or loss of each player's score is exactly balanced by the gain or loss of the other player's score.

2.6 Alpha-beta pruning

Alpha-beta pruning is a technique to reduce the search space by pruning branches that does not affect the result. The current maximum and minimum results are stored in two variables called alpha and beta respectively. If the move ordering is optimal, meaning that the best moves are always stored first, the complexity is reduced to $O(\sqrt{b^d})$. When reaching further depths the majority of the nodes can be pruned which gives a large performance boost to the bot. Figure 3 illustrates a simple example with a chronological description of the steps in alpha-beta pruning of a small tree with a branching factor of 2 from the white's perspective.

2.7 Quiescence search

By having a predetermined depth, a problem arises. The problem can occur when the algorithm reaches a terminal node that is not an checkmate or stalemate and evaluates the node to something in favour of the current player. The algorithm will then choose the action that leads to that terminal node. However, if the search algorithm would have continued one step deeper in the search it could have found that the terminal node is in fact in favour of the opponent. The problem is commonly referred to as the horizon effect and it mostly

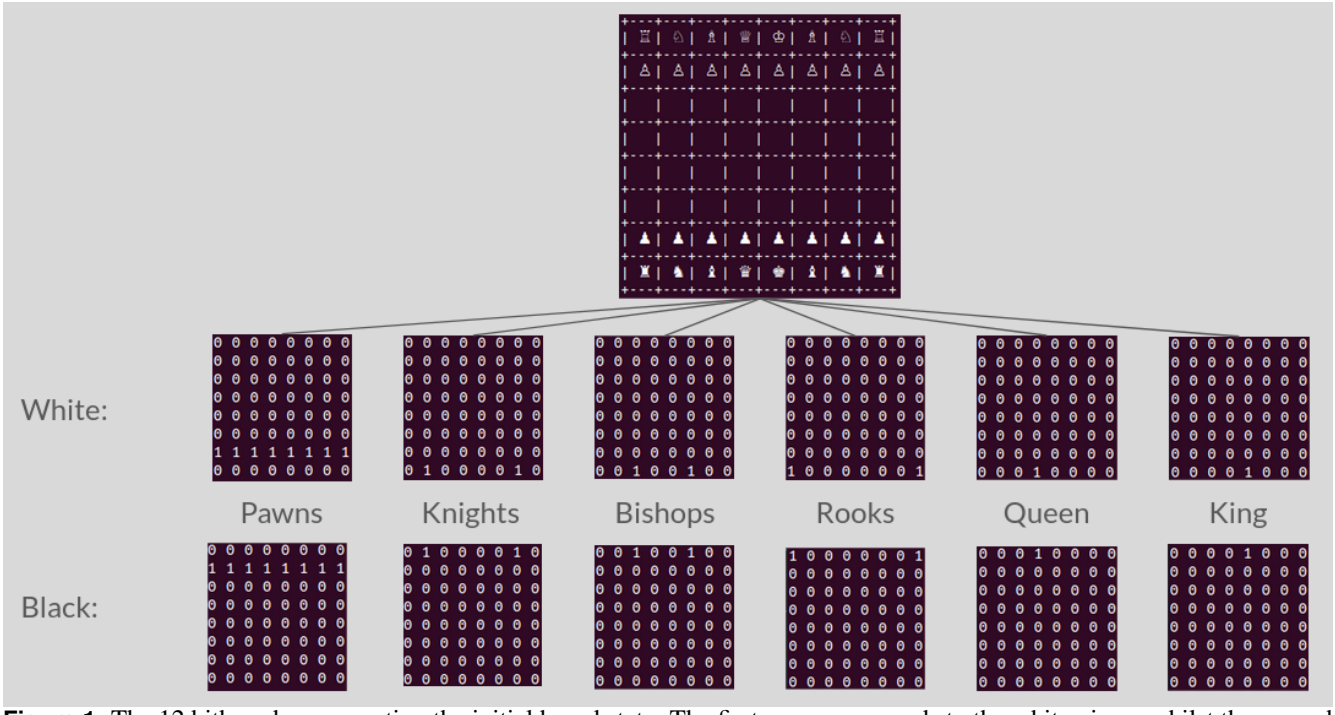


Figure 1. The 12 bitboards representing the initial board state. The first row corresponds to the white pieces whilst the second row corresponds to the black pieces. Each column represent a different piece type.

```
-50,-40,-30,-30,-30,-30,-40,-50,
-40,-20,  0,  0,  0,  0,-20,-40,
-30,  0, 10, 15, 15, 10,  0,-30,
-30,  5, 15, 20, 20, 15,  5,-30,
-30,  0, 15, 20, 20, 15,  0,-30,
-30,  5, 10, 15, 15, 10,  5,-30,
-40,-20,  0,  5,  5,  0,-20,-40,
-50,-40,-30,-30,-30,-30,-40,-50
```

Figure 2. The positional score for the white knights.

happens when the opponent has the potential to capture a piece after the predefined depth. To reduce the risk of the horizon effect occurring, quiescence search can be used. Quiescence search continues the search with a new predefined depth but it only considers the captures which reduces the branching factor. Some captures are more likely to be good or bad so the move ordering is essential to prune as many nodes as possible. A common move ordering technique is called Most Valuable Victim - Least Valuable Aggressor(MVV-LVA). It orders the move according to the relative score gain of the capture.

3. Method

The engine and the AI is implemented in C++ from scratch using the theory described in the previous section. The engine is shell based and uses utf-8 unicode to represent the pieces.

3.1 Bitboards

The initial board position bitboards and some initial puzzle positions are stored as constant 64-bit unsigned integers. Some bitboard masks are also stored as constant arrays to simplfy some bitboard computations. The magic bitboards generation is based on a third-party library called magic-bits [6]. The library were modified to store the bitboards in a single source file and integrated properly into the project.

3.2 Chess rules

Some chess rules were simplified or excluded. The excluded rules are:

- En pessant
- Repetition rules
- Dead position

The simplified rules are:

- Promotion
- Castle rights

The pawns are always promoted to queens to simplify the implementation. The castle rights works properly for the player since the moves are sequential. But since the bot simulates many board states by making and unmaking moves it may remove its castling rights. All the remaining rules are implemented and works properly.

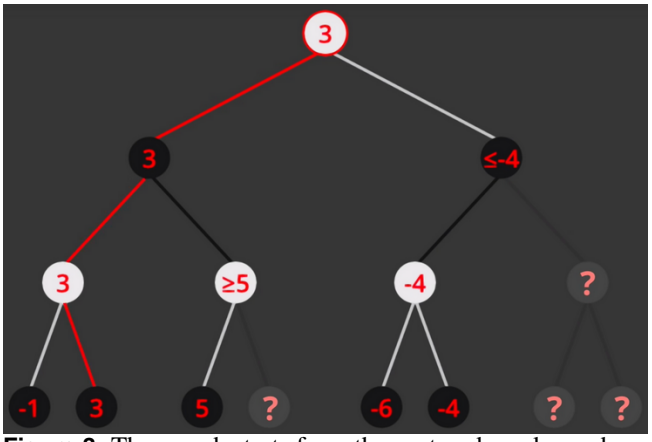


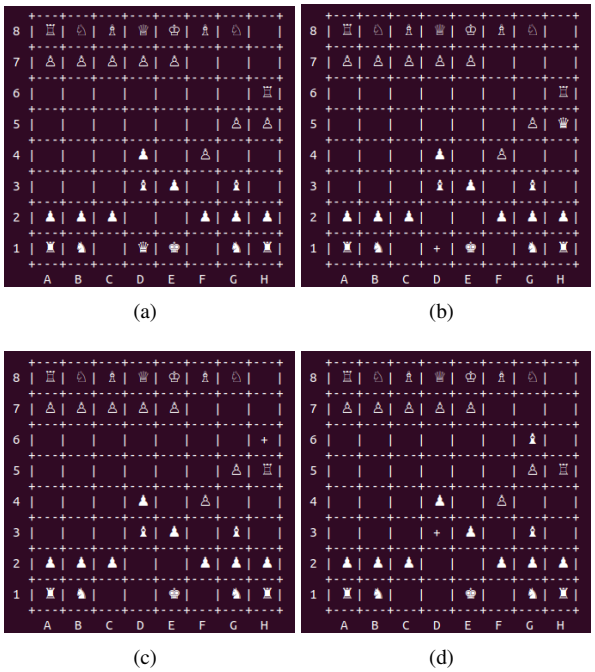
Figure 3. The search starts from the root node and searches the tree using a depth-first search. Once the the algorithm reaches the maximum depth or a terminal node, the node is evaluated. In this case the node is given the score -1. The search continues and the next terminal node is evaluated as 3. Since this search is from white's perspective, white is the maximum player and the white node is therefore evaluated to 3 since it is the maximum of -1 and 3. The next terminal node evaluates to 5 giving the white node a value of ≥ 5. This means that the minimum player always chooses 3 over ≥ 5, thus one node is pruned. The next two terminal nodes evaluates to -6 and -4 resulting in -4 for the white node. This gives the black node a value of ≤ -4. The maximum of 3 and ≤ -4 is 3 and thus the last nodes are pruned. White will therefore choose the move corresponding to the red branch from the root node. Picture by Sebastian Lague.

3.3 Search function

For every position the legal moves are computed and stored in a vector. The vector is sorted based on the material score and captures. The captures are also sorted using MVV-LVA. Then negamax is recursively called for each move and for each move the consequential moves are computed and sorted. Once the predetermined depth has been reached, quiescence search is recursively called for a new predetermined depth. Quiescence search, searches the captures using MVV-LVA until a terminal node has been reached. The terminal node is evaluated using a evaluation function to determine its score. The score propagates through the alpha and beta variables further up the call stack and for each step the move is unmade. Once the call stack is empty, the final score of the root node has been computed and the board is back to its former state. The root node with the highest score is the move chosen by the bot.

4. Result

One simple way to identify if the logic of the bot works is to let it play some puzzles. Figure 4 and 5 illustrates two puzzles where white has checkmate in two moves. The performance of the bot is highly dependent on what hardware it runs on, how many legal moves there are, and how good the move ordering is. However, on an average CPU (AMD Ryzen 7 3700U), depths up to 4+4 and 5+2 (Negamax+Quiescence depths) executes within a reasonable time in a rapid game of chess where each player has a total of ten minutes to make moves.



(a) (b) (c) (d)

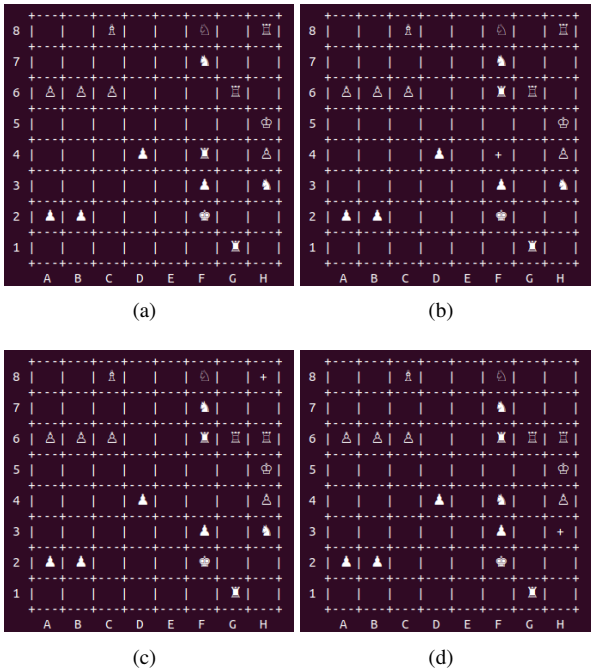
Figure 4. (a) The initial position of the puzzle, white's to move. (b) white moves queen from D1 to H5 putting the black king in check. (c) Black takes queen by moving rook from H6 to H5. (d) White checkmates black by moving bishop from D3 to G6.

5. Discussion

This section introduces the shortcomings of the AI and the future work that could solve these issues. Chess programming is a vast field with many small optimizations that overall improves the performance of the bot. It is hard for a single person to implement all these optimizations in a short time frame.

5.1 Transposition table

Currently, the search algorithm has no memory of previous searches. Therefore, many nodes will be searched multiple times leading to unnecessary computations. A transposition table(TT) is a hash table where the score and the corresponding search depth of each searched node is stored [7]. This means that the algorithm can prune even more nodes by a constant look-up in the TT, resulting in better performance. It can also be used to optimize the move ordering by storing the Principal Variation(PV) which is a sequence of moves that the program considers to be the best [8]. An unsuccessful attempt to implement a TT where made using Zobrist hashing, which is a technique to transform a board position into a number[9].



(a) (b) (c) (d)

Figure 5. (a) The initial position of the puzzle, white's to move. (b) White moves rook from F4 to F6. (c) Black moves rook from H8 to H6. (d) White moves knight from H3 to F4 putting the black king in checkmate.

5.2 Chess rules

For completion it would be nice to incorporate the excluded rules and complete the simplified rules described in the previous section. Some code and comments for en pessant is already included in the source.

6. Conclusion

In hindsight its fair to say that I misjudged the complexity of the chess engine's implementation which reduced my time to implement the AI. If I would have used a library for the engine in another programming language I may have been able to increase the search depth by focusing more on the AI's implementation. However, my commitment to the engine's implementation resulted in a fast engine which is the core that the AI runs on. With a working TT in place, I believe that the bot's performance would drastically improve. In conclusion, I exceeded my initial goal of reaching a depth larger than five.

References

[1] Stockfish 12. https://stockfishchess.org/. Accessed: 2020-12-26.

[2] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy P. Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *CoRR*, abs/1712.01815, 2017.

[3] Bitboards. https://www.chessprogramming.org/Bitboards. Accessed: 2020-12-26.

[4] Magic bitboards. https://www.chessprogramming.org/Magic_Bitboards. Accessed: 2020-12-28.

[5] What is the average number of legal moves per turn? https://chess.stackexchange.com/questions/23135/what-is-the-average-number-of-legal-moves-per-turn/24325#24325. Accessed: 2020-12-26.

[6] magic-bits. https://github.com/goutham/magic-bits. Accessed: 2020-12-28.

[7] Transposition table. https://www.chessprogramming.org/Transposition_Table. Accessed: 2020-12-28.

[8] Principal variation. https://www.chessprogramming.org/Principal_Variation. Accessed: 2020-12-28.

[9] Zobrist hashing. https://www.chessprogramming.org/Zobrist_Hashing. Accessed: 2020-12-28.