Chess Engine

COMP4121

Joshua Shim

November 2024

Abstract

Chess is a simple game that is played on a 8×8 grid board, with 6 different types of pieces that each move differently. Intuitively it is a simple game. However due to the permutational complexity of this game, a large number of humans have dedicated their lives to chess and yet chess remains 'unsolved'. Chess remains a highly theoretical board game where it remains subjective in many cases as to which move is the best.

By the late 1980s computer chess programs had begun being competitive with top human players and 25 years later, the last known win against a top-performing computer by a human player was recorded in the Ponomariov vs Fritz game.

Since then, as hardware and chess algorithmic theory have had noticeable improvements, the delta between human chess and computer chess programs have only increased.

For the COMP4121 major project, I have chosen to make a chess engine from scratch. Coming from a pure computer science background with limited mathematical knowledge, I believe that the creation of a chess engine would allow me to actively learn about both the deterministic and randomised algorithms that allow chess engines to overcome the human brain. Hence I came up with the goal of creating a chess engine that would be able to beat the average human player.

Initially, I had planned to create a chess engine that primarily utilised the *Monte Carlo Tree Search* algorithm to search for the best moves. However due to time constraints and the inability for me to feasibly tune the *Monte Carlo Tree Search* using either deep learning meant that a *MCTS* chess engine would be significantly less impressive than a more standard chess engine that utilises the *Min-Max Algorithm*.

The chess bot that I have created is named **COMP4121bot** and uses a *Universal Chess Interface (UCI)* interface to communicate with the front-end. I have a raspberry pi 3 model b setup that runs the chess engine so that it exists as a bot on *lichess.org* where it is available for play. I will keep it active on the website until 31/12/2024 and highly encourage you to play a game against it. Note that as it is running on lower end hardware, it's performance is significantly lower than if it were to run on my desktop. The following game modes are available for my bot:

Contents

Chapter 1

Board Representation

A chess program requires an internal board representation to maintain the current locations of each piece and also additional information to identify whose turn it is to play, castling rights and whether an en passant move is possible. Furthermore, more advanced engines require additional data structures to keep track of previous moves in order to avoid *threefold repetition* and the *fifty-move rule*.

1.1 Array

The most intuitive board representation would be a 8×8 2-dimensional array of 1 byte words. This board representation was the first design that had come to my mind. This design would also be quite space efficient, which was a big consideration for me as the *Monte Carlo Tree Search* algorithm requires a tree data structure to be kept, with each *node* containing a full board representation.

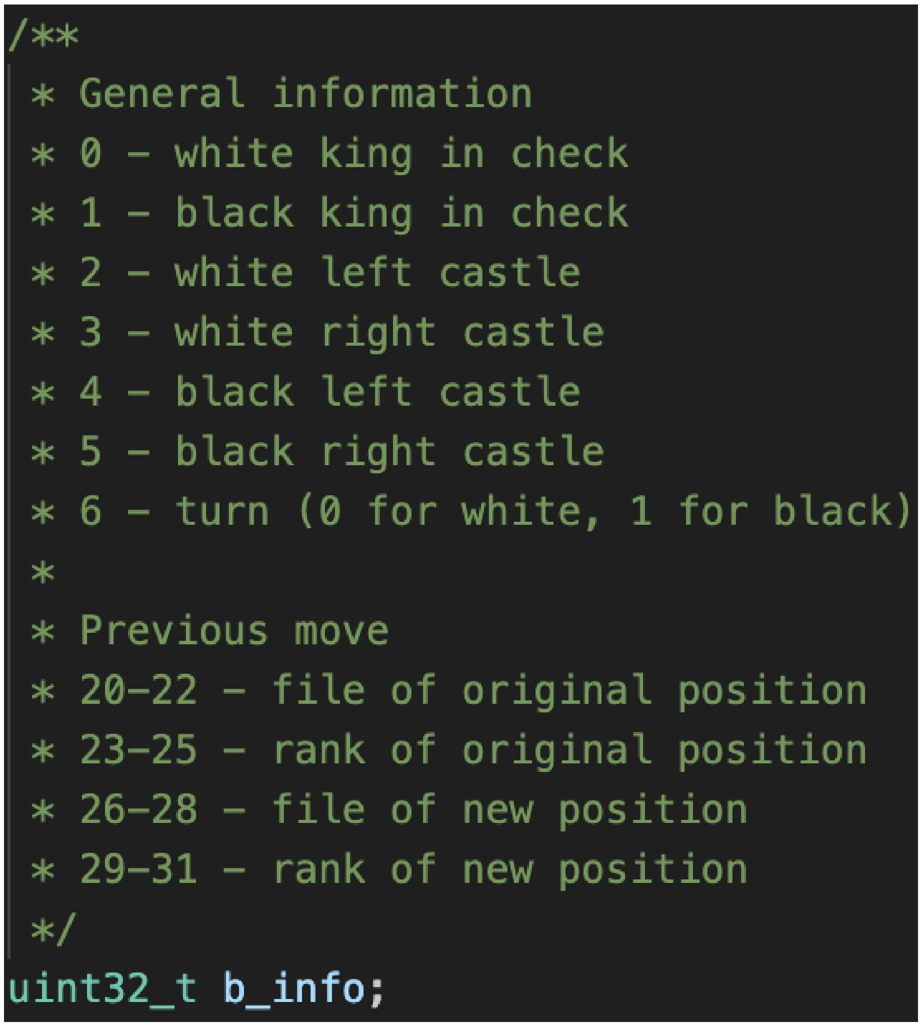
This particular board representation would allow for the whole board to be represented by **64 bytes** in total, as there are 64 slots of 1 byte each. As there are only 12 different types of pieces, only 4 bits are required to determine the colour of the piece and the type of the piece. This allows for the board to be represented with a lower byte count of **32 bytes** if we were to use 4 bit words for each cell of the array. Whereas this would be faster for systems that are optimised for 4 bit words, most modern systems and hardware is designed to handle 8-64 bit words, I would have chosen to take the larger sized array for higher speed. Furthermore, more bits would be required to hold the additional information about the state of the game.

Despite the small size of the board representation, speed which is arguably a greater concern when designing chess engines is much slower as opposed to using bitboards. This is because when we generate possible moves, it is necessary to loop through each cell, identify whether a piece exists within the cell and create copies of the board for each move that can be made.

1.2 Bitboards

Bitboards are an array of 64-bit words that can represent the chess board. Because there are 2 colours and 6 different types of pieces, the usual bitboard representation consists of an array of 12 64-bit words, each *bitboard* being a representation of the locations of their respective pieces. 1-bits inside the 64-bit bitboard represents a location where the piece exists, where the most significant bit represents $a8$ and the least significant bit represents $h1$. This results in a relatively dense board representation consisting of $12 \times 8 = 96$ total bytes per board representation alongside additional bits for turns, castling rights and en passant.

Due to the balance of high performance and memory density of a bitboard representation, I decided to use bitboards as the internal board representation of my chess engine. The first 6 indices of my array are reserved for white pieces and the next 6 for the black pieces. They represent in order: pawns, knights, bishops, rooks, queens and the king. Aside from the 12 length array of 64-bit words, I have an additional 32 bit word to contain additional information about the game state as shown below.



```
/**
 * General information
 * 0 - white king in check
 * 1 - black king in check
 * 2 - white left castle
 * 3 - white right castle
 * 4 - black left castle
 * 5 - black right castle
 * 6 - turn (0 for white, 1 for black)
 *
 * Previous move
 * 20-22 - file of original position
 * 23-25 - rank of original position
 * 26-28 - file of new position
 * 29-31 - rank of new position
 */
uint32_t b_info;
```

Chapter 2

Move Generation

Chess engines aim to choose the best possible move that can be taken given the game state. Hence, chess engines typically generate all possible moves for the side whose turn it is. This provides the possibilities for the engine to pick moves from. Further moves are generated recursively from the newly generated moves in order to analyse how certain choices may lead to certain outcomes in the future, and to evaluate which move would be the best to take in the present. Here I will explain how moves are efficiently generated using bitboards as implemented in my chess engine.

2.1 Precomputations and Optimisations

A downside of the bitboard board representation is that it is computationally inefficient in finding whether a piece exists in any given square, whether that piece is black or white and which type of piece it is. Hence, an optimisation I have made is to precompute at the start of every move generation two 64-bit-masks, the first to indicate the squares that are occupied by the white pieces and the second for the black pieces.

Furthermore, another optimisation that I have made was to generate all *pseudo-legal* moves. Legal moves are moves that can be legally taken as per the rules of chess. The moves that my engine generates include illegal moves such as when the king is in check, moves that don't uncheck the king are not removed in the list of moves returned by my move generation function. The logic behind this is that if the king is in check and an illegal move is made, either the MCTS or min-max algorithm, if correctly implemented would avoid taking that move as it would result in an automatic loss. If we were to generate moves to a depth of 2 in order to check for illegal moves as described, the time complexity of move generation would become $O(n^2)$ as opposed to $O(n)$, where n is the largest number of possible moves. This has greater implications in the search function as the move evaluation function would be called recursively, compounding the increase in latency.

2.2 Special Moves

There are two special moves that can be made in chess: Castling and En passant. Specific rules apply to these moves and therefore we must use additional bits allocated to our board representation.

2.2.1 Castling

Castling is a special move that involves the king and the rook. Castling requires both the king and the rook to be castled with to not have moved, the king to not be in check and there to not exist pieces between the king and the rook. Hence for each side, we must reserve two bits each to indicate whether a respective rook has moved or not and we can set both bits to not allow castling if the king has moved.

2.2.2 En Passant

En Passant is a special move that translates to *"on passing"* and involves the pawns. If playing as white and one of your pawns is on rank 5 and black does *double push* with one of its pawns on an adjacent file, you may take the recently moved pawn as if it had done a single push. The same applies for black on rank 4. Hence it is necessary to keep track of the most recently moved piece as En Passant is only a possible move if the enemy pawn had passed your pawn in the most recent move. My solution to this problem was to keep track of the new position and old position of the most recently moved piece. As there are a total of 8 ranks and files, $3 \times 2 \times 2 = 12$ bits in total were required to store the old position's rank and file and the new position's rank and file.

2.3 Non-Sliding Moves

Non-Sliding moves in chess are moves that have limited range but can happen regardless of enemy obstruction. This includes moves from pawns, knights and kings. Bitboards allow for very efficient non-sliding move generation as we could generate a non-sliding move for every piece in one cpu instruction then mask out the unavailiable moves through another instruction. For example, we could create every possible *double push* for all the white pawns by left-shifting the white pawn bitboard by 16 bits. Then we could use two *&*= instructions to mask out possible destinations where the pieces are obstructed by either enemy or friendly pieces.

After generating all of the possible destinations, I then loop through each bit in the newly created destination bitboard by selecting the least significant 1 bit and then creating a copy of the original board where the the piece moves to the new destination and push the new board/move to the list of generated moves. The selection of the least significant bit is done using the negation operator -, then then and operator *&*. This works as the negative of a number is represented by the two's complement, where the negation is the NOT of the number + 1. Hence the least significant bit is preserved whilst all the higher significance bits are masked out. After the copy is created and pushed into the list, an xor operation ˆ with the destinations bitboard allows me to remove the used destination. A similar process is used to generate the new boards for sliding moves.

2.4 Sliding Moves

Sliding moves in chess are moves that have unlimited range until obstructed by another piece or by the border of the board. Examples include the rook/queen's horizontal and vertical movements and the bishop/queen's diagonal movements. I implemented sliding moves in my engine by hardcoding each possible direction that a piece could move, appropriately shifting the previous location and if not obstructed using the or — operation to add to a destination bitboard. This is repeated until the direction is obstructed by a friendly piece. if obstructed by an enemy piece, the — operation is done for the destination bitboard but the loop is broken as to not generate moves where the enemy piece is passed.

Chapter 3

Move Evaluation

The move evaluation function is one of the most important aspects of a chess engine as it is what essentially determines the move that is to be selected by the chess engine. With perfect heuristics, the search function loses relevancy as it would be possible to determine the best move/position through the move evaluation alone.

3.1 Piece Value

My original move evaluation function began as a simple addition subtraction function that attributed points to each type of piece, multiplied the number of existing pieces by these points and took the sum of these points as the total points for each side. Then the advantage could be calculated through the difference between the sum of each side.

$$\text{white\_advantage} = wp\cdot 100+wn\cdot 300+wb\cdot 325+wr\cdot 500+wq\cdot 900-(bp\cdot 100+bn\cdot 300+bb\cdot 325+br\cdot 500+bq\cdot 900)$$

Despite being a rather simple evaluation function, it is possible to create a relatively strong chess engine using such a simple evaluation function. An advantage of this evaluation function is that it is very fast to compute and may allow for more time and potentially deeper searches. Conversely, this approach fails to be effective at the start of the game as simple searchs usually do not allow for nuanced positioning and piece development that may aid the engine later on. A similar issue is faced during the endgame for similar reasons. However it is possible to complement the evaluation function and move generation to avoid these problems. For example a trie database of Grandmaster Chess games could be relied on at the start to allow for the chess engine to play more naturally.

3.2 Positional Value

A more complicated and nuanced approach is to consider the value that is to be had from occupying certain positions on the chess board. For example a knight in the centre of the board is arguably more valuable than a knight on the edge or corner of the board as it allows for more options for the knight and furthermore allows the knight to control more squares. Likewise pawns that are closer to being promoted are more valuable than pawns that are far from being promoted as these pawns have to possibility to become the most valuable piece in the game.

Positional values are especially more important in endgames where computer programs may not be able to plan ahead for checkmates with limited pieces. The Manhattan Distance

$$d = \sum_{i=1}^{n} |x_i - y_i|$$

is especially useful as we may use it to give bonus advantage points when the enemy king is nearer to the corners of the board for easier checkmates.

My final implementation used the both the piece value alongside the positional values with the addition of the manhattan distance to aid in endgames. Due to a lack of time and hardware required for tuning for good positional value heatmaps, credit for the heatmaps go to Romain Goussault's Deepov Chess Engine: *https://github.com/RomainGoussault/Deepov.git*

Chapter 4

Monte Carlo Tree Search