AlphaDeepChess: motor de ajedrez basado en podas alpha-beta
AlphaDeepChess: chess engine based on alpha-beta pruning

Trabajo de Fin de Grado
Curso 2024–2025

Autores
Juan Girón Herranz
Yi Wang Qiu

Directores
Ignacio Fábregas Alfaro
Rubén Rafael Rubio Cuéllar

Grado en Ingeniería de Computadores
Grado en Desarrollo de Videojuegos
Facultad de Informática
Universidad Complutense de Madrid

AlphaDeepChess: motor de ajedrez basado en podas alpha-beta
AlphaDeepChess: chess engine based on alpha-beta pruning

Trabajo de Fin de Grado en Ingeniería de Computadores
Trabajo de Fin de Grado en Desarrollo de Videojuegos

Autores
Juan Girón Herranz
Yi Wang Qiu

Directores
Ignacio Fábregas Alfaro
Rubén Rafael Rubio Cuéllar

Convocatoria: *Junio 2025*

Grado en Ingeniería de Computadores
Grado en Desarrollo de Videojuegos
Facultad de Informática
Universidad Complutense de Madrid

13 de mayo de 2025

Dedication

To our younger selves, for knowing the art of chess

Acknowledgments

To our family members for their support and for taking us to chess tournaments to compete.

Abstract

AlphaDeepChess: chess engine based on alpha-beta pruning

Chess engines have played a fundamental role in the advancement of artificial intelligence applied to chess since the mid-20th century. Pioneers such as Alan Turing and Claude Shannon established the theoretical principles that laid the foundation for this field. Building upon these foundations, the evolution of hardware and the refinement of search techniques have enabled significant advancements, such as alpha-beta pruning, an optimization of the minimax algorithm that drastically reduces the number of nodes evaluated in the game tree. Today, Stockfish, the most powerful and open-source chess engine, continues to rely on alpha-beta pruning but also incorporates deep learning techniques and neural networks.

The goal of this project is to develop a chess engine capable of competing against both other engines and human players, using alpha-beta pruning as its core. Additionally, we will analyze the impact of other classical algorithmic techniques such as transposition tables, iterative deepening, and a move generator based on magic bitboards.

The chess engine has finally been uploaded to Lichess platform, where AlphaDeepChess achieved an ELO rating of 1900 while running on a Raspberry Pi 5 equipped with a 2TB transposition table.

Keywords

chess, chess engine, alpha-beta pruning, iterative deepening, quiescence search, move ordering, transposition table, zobrist hashing, pext instruction, magic bitboards

Resumen

AlphaDeepChess: motor de ajedrez basado en podas alpha-beta

Los motores de ajedrez han desempeñado un papel fundamental en el avance de la inteligencia artificial aplicada al ajedrez desde mediados del siglo XX. Pioneros como Alan Turing y Claude Shannon establecieron los principios teóricos que sentaron las bases de este campo. Sobre estos cimientos, la evolución del hardware y el perfeccionamiento de técnicas de búsqueda permitieron importantes avances, como la poda alfa-beta, una optimización del algoritmo minimax que reduce drásticamente el número de nodos evaluados en el árbol de juego. Hoy en día, Stockfish, el motor de ajedrez más potente y de código abierto, sigue basándose en técnicas algorítmicas clásicas, pero también incorpora deep learning y redes neuronales.

El objetivo de este proyecto es desarrollar un motor de ajedrez capaz de competir tanto contra otros motores como contra jugadores humanos, utilizando la poda alfa-beta como núcleo del algoritmo. Además, se analizará el impacto de otras técnicas algorítmicas clásicas, como las tablas de transposición, la búsqueda en profundidad iterativa y un generador de movimientos basado en bitboards mágicos.

Finalmente, el motor de ajedrez ha sido subido a la plataforma de Lichess, donde AlphaDeepChess ha alcanzado una puntuación ELO de 1900, ejecutándose en una Raspberry Pi 5 con una tabla de transposiciones de 2TB.

Palabras clave

ajedrez, motor de ajedrez, poda alfa-beta, búsqueda en profundidad iterativa, búsqueda quiescente, ordenación de movimientos, tabla de transposiciones, zobrist hashing, instrucción pext, bitboards mágicos

Contents

List of figures

List of tables

Chapter 1

Introduction

"The most powerful weapon in chess is to have the next move"
— David Bronstein

Chess, one of the oldest strategy games in human history, has long been a domain for both intellectual competition and computational research. The pursuit of creating a machine that could compete with the best human players, chess Grandmasters, was present. It was only a matter of time before computation surpassed human capabilities.

In 1997, the chess engine `Deep Blue` made history by defeating the reigning world champion at the time, Garry Kasparov, marking the first time a computer had defeated a sitting world chess champion.

Today, we find ourselves in an era where chess engines have reached unprecedented strength. This has been achieved through a combination of classical techniques like alpha-beta pruning, and modern advancements such as deep learning and neural networks.

1.1. Objectives

The objectives of this project are the following:

- Develop a chess engine based on alpha-beta pruning that follows the UCI protocol (Stefan-Meyer Kahlen (2004)). The engine will be a console application capable of playing chess against humans or other engines, as well as analyzing and evaluating positions to determine the best legal move.
- Implement various optimization techniques, including move ordering, quiescence search, iterative deepening, transposition tables, multithreading, and a move generator based on magic bitboards.
- Measure the impact of these optimization techniques and profile the engine to identify performance bottlenecks.

- Upload the engine to `lichess.org` and compete against other chess engines.

1.2. Work plan

The project will be divided into several phases, each focusing on specific aspects of the engine's development. The timeline for each phase is as follows:

1. Research phase and basic implementation: understand the fundamentals of alpha-beta pruning with minimax and position evaluation. Familiarize with the UCI (Universal Chess Interface) and implement the move generator with its specific exceptions and rules.
2. Optimization: implement quiescence search and iterative deepening to improve pruning effectiveness.
3. Optimization: improve search efficiency using transposition tables and Zobrist hashing.
4. Optimization: implement multithreading to enable parallel search.
5. Profiling: use a profiler to identify performance bottlenecks and optimize critical sections of the code.
6. Testing: use Stockfish to compare efficiency generating tournaments between chess engines and estimate the performance of the engine. Also, compare different versions of the engine to evaluate the impact of optimizations.
7. Analyze the results and write the final report.

In the following Section 1.3, we will talk about the basic concepts of chess, but if you already have the knowledge we recommend you to advance directly to the next chapter 2.

1.3. Basic concepts

Chess is a board game where two players who take white pieces and black pieces respectively compete to first checkmate the opponent. Checkmate occurs when the king is under threat of capture (known as check) by a piece or pieces of the enemy, and there is no legal way to escape or remove the threat.

What about a chess engine? A chess engine consists of a software program that analyzes chess positions and returns optimal moves depending on its configuration. In order to help users to use these engines, chess community agreed on creating an open communication protocol called **Universal Chess Interface** or commonly referred to as UCI, that provides the interaction with chess engines through user interfaces.

A chess game takes place on a chessboard with specific rules governing the movement and interaction of the pieces. This section introduces the fundamental concepts necessary to understand how chess is played.

1.3.1. Chessboard

A chessboard is a game board of 64 squares arranged in 8 rows and 8 columns. To refer to each of the squares we mostly use **algebraic notation** using the numbers from 1 to 8 and the letters from "a" to "h". There are also other notations like descriptive notation (now obsolete) or ICCF numeric notation due to chess pieces have different abbreviations depending on the language.

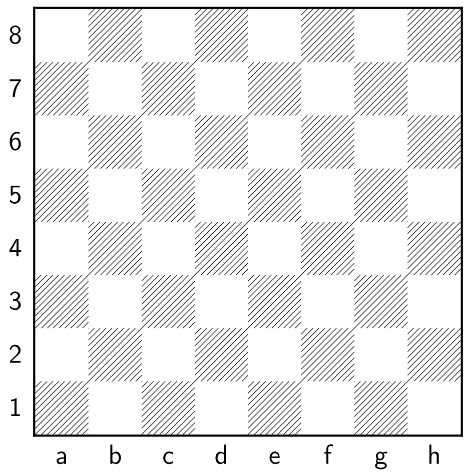


Figure 1.1: Empty chessboard.

For example, $g5$ refers to the following square:

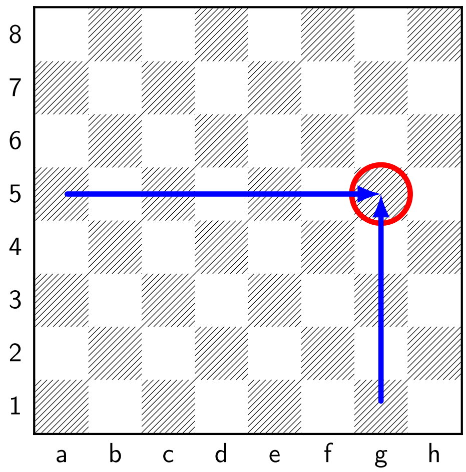


Figure 1.2: Example: square $g5$ highlighted and arrows pointing to it.

It is important to know that when placing a chessboard in the correct orientation, there should always be a white square in the bottom-right corner or a black square in the bottom-left corner.

1.3.2. Chess pieces

There are 6 types of chess pieces: king, queen, rook, bishop, knight and pawn, and each side has 16 pieces:

Piece	White Pieces	Black Pieces	Number of Pieces
King	♔	♚	1
Queen	♕	♛	1
Rook	♖ ♖	♜ ♜	2
Bishop	♗ ♗	♝ ♝	2
Knight	♘ ♘	♞ ♞	2
Pawn	♙ ♙ ♙ ♙ ♙ ♙ ♙ ♙	♟ ♟ ♟ ♟ ♟ ♟ ♟ ♟	8

Table 1.1: Number of chess pieces by type and color.

The starting position of the chess pieces on a chessboard is the following:

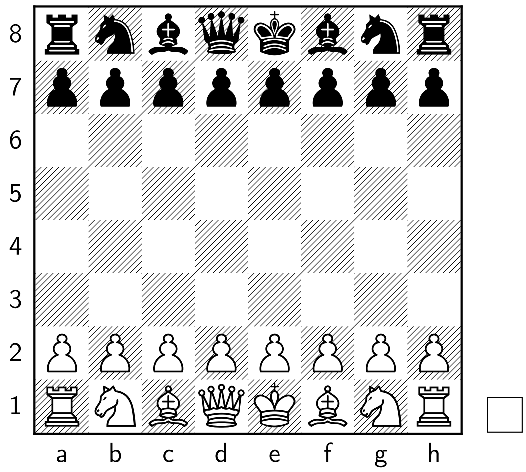


Figure 1.3: Starting position.

The smaller white square next to the board indicates which side is to move in the current position. If the square is white, it means it is white's turn to move; if the square is black, it means it is black's turn to move. This visual indicator helps clarify which player has the next move in the game. Notice that the queen and king are placed in the center columns. The queen is placed on a square of its color, while the king is placed on the remaining central column. The rest of the pieces are positioned symmetrically, as shown in Figure 1.3. This means that the chessboard is divided into two sides relative to the positions of the king and queen at the start of the game:

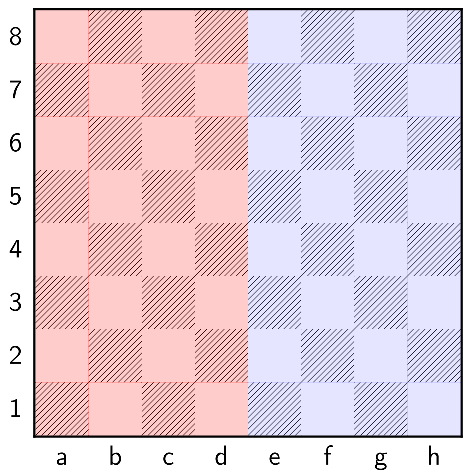


Figure 1.4: King's side (blue) and Queen's side (red).

1.3.3. Movement of the pieces

1.3.3.1. Pawn

The pawn can move one square forward, but it can only capture pieces one square diagonally. On its first move, the pawn has the option to move two squares forward. If a pawn reaches the last row of the opponent's side, it promotes to any other piece (except for a king). Promotion is a term to indicate the mandatory replacement of a pawn with another piece, usually providing a significant advantage to the player who promotes.

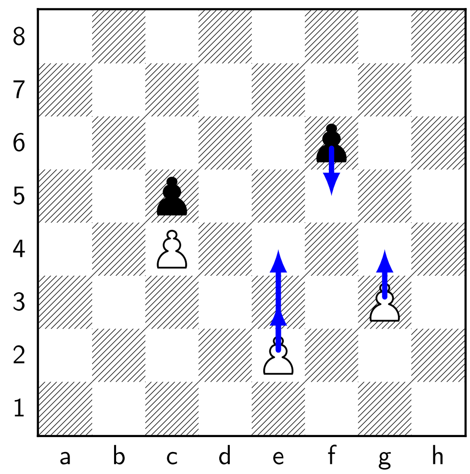


Figure 1.5: Pawn's movement.

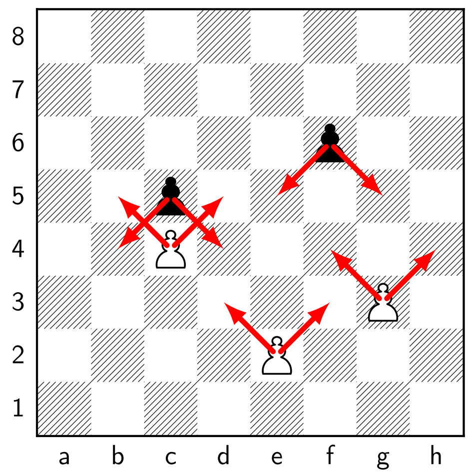


Figure 1.6: Pawn attack.

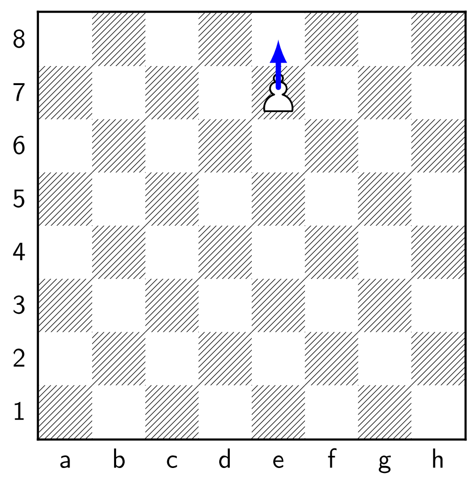


Figure 1.7: Promotion.

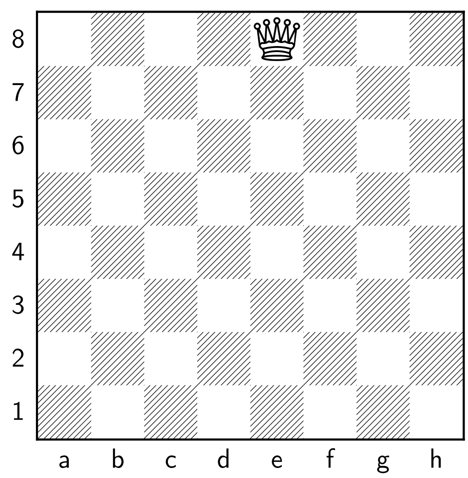


Figure 1.8: Pawn promotes to queen.

There is a specific capture movement which is **en passant**. This move allows a pawn that has moved two squares forward from its starting position to be captured by an opponent's pawn as if it had only moved one square. The capturing pawn must be on an adjacent file and can only capture the en passant pawn immediately after it moves.

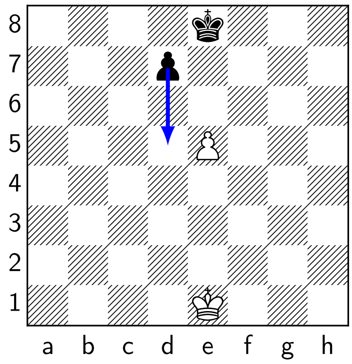


Figure 1.9: En passant (1).

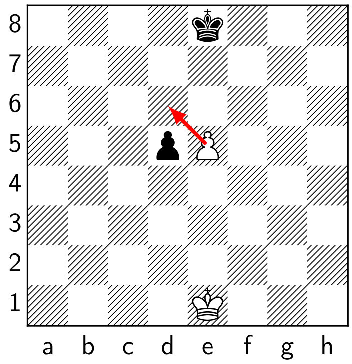


Figure 1.10: En passant (2).

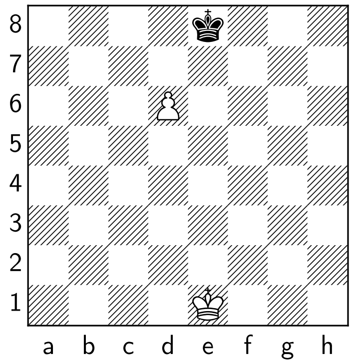


Figure 1.11: En passant (3).

1.3.3.2. Rook

The rook can move any number of squares horizontally or vertically. It can also capture pieces in the same way.

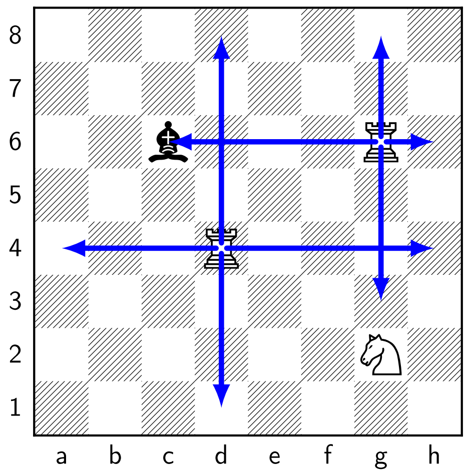


Figure 1.12: Rook's movement.

1.3.3.3. Knight

The knight moves in an L-shape: two squares in one direction and then one square perpendicular to that direction. The knight can jump over other pieces, making it a unique piece in terms of movement. It can also capture pieces in the same way.

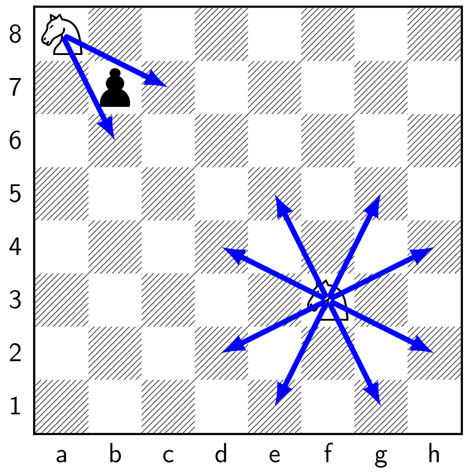


Figure 1.13: Knight's movement.

1.3.3.4. Bishop

The bishop can move any number of squares diagonally. It can also capture pieces in the same way. Considering that each side has two bishops, one bishop moves on light squares and the other on dark squares.

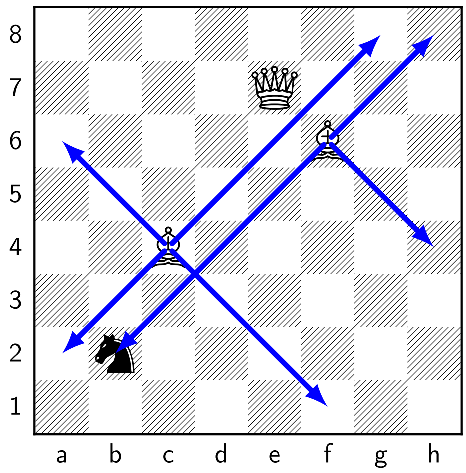


Figure 1.14: Bishop's movement.

1.3.3.5. King

The king can move one square in any direction: horizontally, vertically, or diagonally. However, the king cannot move to a square that is under attack by an opponent's piece. The king can also capture pieces in the same way. The king is a crucial piece in chess, as the game ends when one player checkmates the opponent's king.

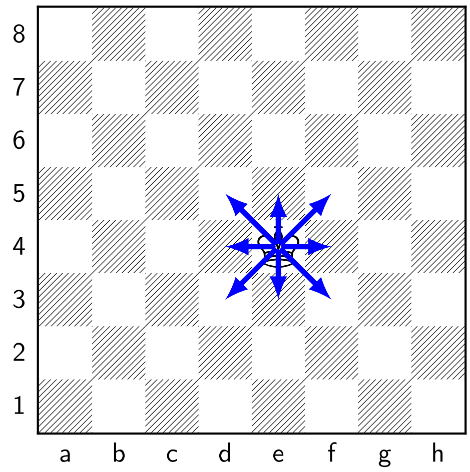


Figure 1.15: King's movement.

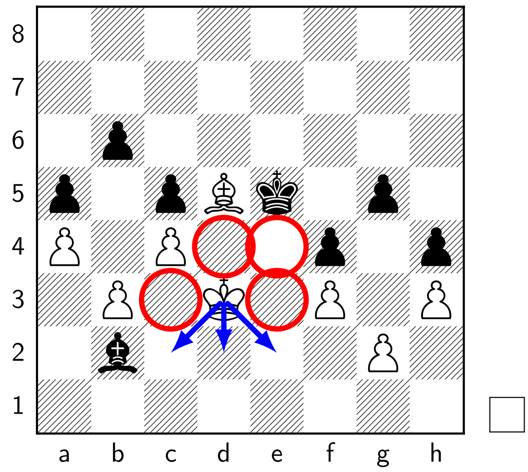


Figure 1.16: White King's movement in a game.

In Figure 1.16, the white king cannot move to $e4$ because the black king is attacking that square.

Castling is a special move which involves moving the king two squares towards a rook and moving the rook to the square next to the king. Castling has specific conditions which are:

- Neither the king nor the rook involved in castling must have moved previously.
- There must be no pieces between the king and the rook.
- The king cannot be in check, move through a square under attack, or end up in check.

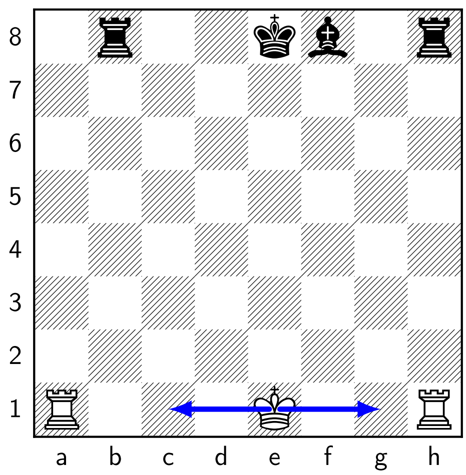


Figure 1.17: Castling

In Figure 1.17, the white king can castle on either the king's side or the queen's side as long as the rooks have not been moved from their starting position, but the black king cannot castle because there is a bishop on $f8$ interfering with the movement and the rook on the queen's side has been moved to $b8$.

1.3.3.6. Queen

The queen can move any number of squares in any direction: horizontally, vertically, or diagonally. It can also capture pieces in the same way.

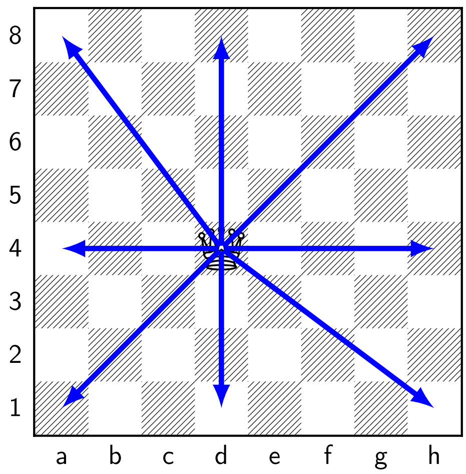


Figure 1.18: Queen's movement.

1.3.4. Rules

The rules of chess follow the official regulations established by FIDE Fédération Internationale des Échecs (2023). The objective of each player is to checkmate the opponent's king, meaning the king is under attack and cannot escape.

In every game, white starts first, and the possible results of each game can be win for white, win for black or draw. A draw or tie could be caused by different conditions:

1. Stalemate: the player whose turn it is to move has no legal moves, and their king is not in check.
2. Insufficient material: neither player has enough pieces to checkmate. Those cases are king vs king, king and bishop vs king, king and knight vs king, and king and bishop vs king and bishop with the bishops on the same color.
3. Threefold repetition: it occurs when same position happens three times during the game, with the same player to move and the same possible moves (including castling and en passant).
4. Fifty-move rule: if 50 consecutive moves are made by both players without a pawn move or a capture, the game can be declared a draw.

5. Mutual agreement: both players can agree to a draw at any point during the game.

6. Dead position: a position where no legal moves can be made, and the game cannot continue. This includes cases like king vs king, king and knight vs king, or king and bishop vs king.

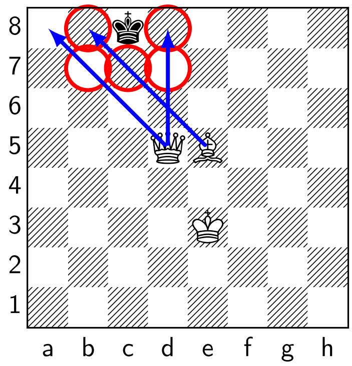


Figure 1.19: Stalemate.

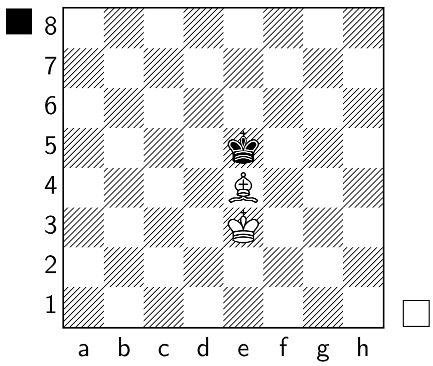


Figure 1.20: Insufficient material.

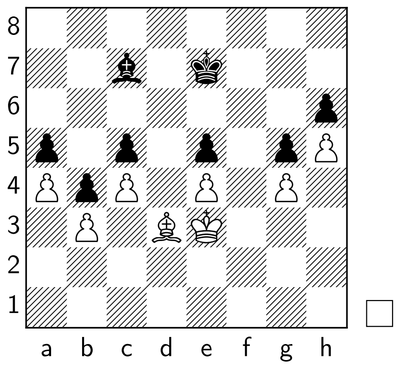


Figure 1.21: Dead position.

Players can also resign at any time, conceding victory to the opponent. Also, if a player runs out of time in a timed game, they lose unless the opponent does not have enough material to checkmate, in which case the game is drawn.

1.3.5. Notation

Notation is important in chess to record moves and analyze games.

1.3.5.1. Algebraic notation

In addition to the algebraic notation of the squares in Section 1.3.1, each piece is identified by an uppercase letter, which may vary across different languages:

Piece	English Notation	Spanish Notation
Pawn	P	P (peón)
Rook	R	T (torre)
Knight	N	C (caballo)
Bishop	B	A (alfil)
Queen	Q	D (dama)
King	K	R (rey)

Table 1.2: Chess piece notation in English and Spanish.

Normal moves (not captures nor promoting) are written using the piece uppercase letter plus the coordinate of destination. In the case of pawns, it can be written only with the coordinate of destination:

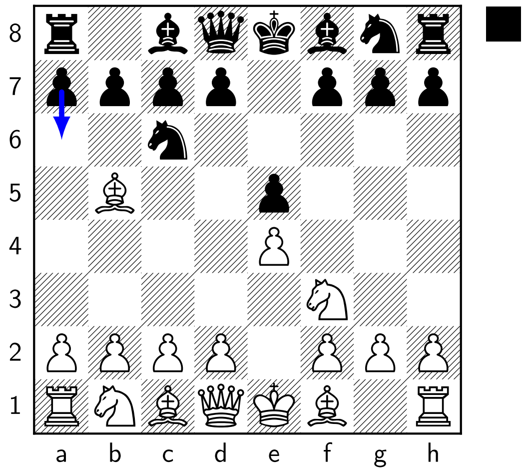


Figure 1.22: Pawn goes to a6.

In Figure 1.22, the pawn's movement is written as $Pa6$ or directly as $a6$.

Captures are written with an "x" between the piece uppercase letter and coordinate of destination or the captured piece coordinate. In the case of pawns, it can be written with the column letter of the pawn that captures the piece. Also, if two pieces of the same type can capture the same piece, the piece's column or row letter is added to indicate which piece is moving:

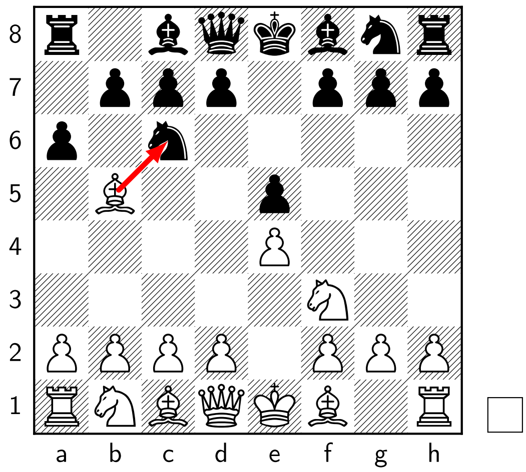


Figure 1.23: Bishop captures knight.

In Figure 1.23, the white bishop capturing the black knight is written as $Bxc6$. If it were black's turn, the pawn on $a6$ could capture the white bishop, and it would be written as $Pxb5$ or simply $axb5$, indicating the pawn's column.

Pawn promotion is written as the pawn's movement to the last row, followed by the piece to which it is promoted:

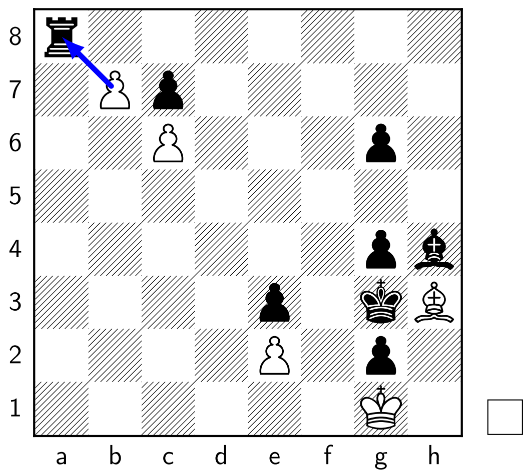


Figure 1.24: Pawn captures rook.

In Figure 1.24, white pawn capturing and promoting in $a8$ to a queen is written as $bxa8Q$ or $bxa8 = Q$.

Castling depending on whether it is on the king's side or the queen's side, it is written as $0-0$ and $0-0-0$, respectively.

Check and checkmate are written by adding a + sign for check or ++ for checkmate, respectively.

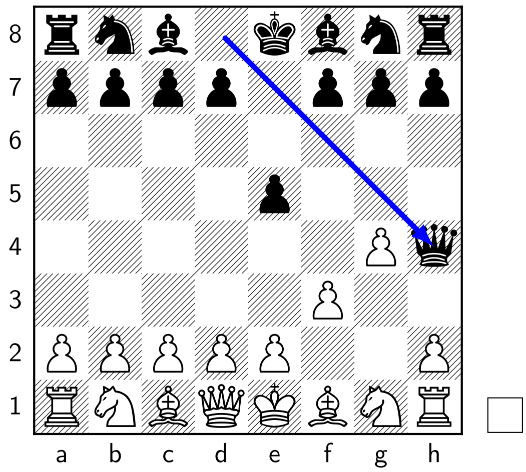


Figure 1.25: Black queen checkmates.

In Figure 1.25, black queen movement checkmates and it is written as $Dh4++$.

The end of game notation indicates the result of the game. It is typically written as:

- *1-0*: White wins.
- *0-1*: Black wins.
- *1/2-1/2*: The game ends in a draw.

1.3.5.2. Forsyth–Edwards Notation (FEN)

This is a notation that describes a specific position on a chessboard. It includes 6 fields separated by spaces: the piece placement, whose turn it is to move, castling availability, en passant target square, halfmove clock, and fullmove number. For example, the FEN for the starting position is:

```
rnbqkb1r/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1
```

Keep in mind this notation is important for the engine to understand the position of the pieces on the board.

1.3.5.3. Portable Game Notation (PGN)

This notation is mostly used for keeping information about the game and a header section with metadata: the name of the event, site, date of play, round, color and name of each player, and result. For example, the PGN for a game could look like this:

Listing 1.1: Example of a PGN file

```
[Event "XX␣Gran␣Torneo␣Internacional␣Aficionado"]
[Site "?"]
[Date "2024.06.25"]
[Round "1"]
[White "Tejedor␣Barber,␣Lorenzo"]
[Black "Giron␣Herranz,␣Juan"]
[Result "0-1"]
1. e4 c6 2. d4 d5 3. exd5 cxd5 4. Bd3 Nc6 5. c3 Nf6 6. Bf4 Bg4
    7. Qb3 Qd7 8. h3 Bh5 9. Nd2 e6 10. Ngf3 a6 11. O-O Be7 12.
    Rfe1 O-O 13. Re3 b5 14. Ne5 Qb7 15. Nxc6 Qxc6 16. Nf3 Nd7
   17. a3 Bg6 18. Bxg6 hxg6 19. Rae1 a5 20. Qd1 b4 21. axb4
   axb4 22. h4 bxc3 23. bxc3 Ra3 24. Qd3 Rc8 25. Rc1 Bb4 26.
   h5 gxh5 27. Ng5 Nf6 28. Be5 Rxc3 29. Rxc3 Qxc3 30. Qe2 Qc1+
    31. Kh2 Bd2 32. Bxf6 Bxe3 33. fxe3 gxf6 34. Nh3 Qb1 35.
   Nf4 Rc1 36. Nxh5 Rh1+ 37. Kg3
```

Chapter 2

State of the art

In this chapter, we will explore the fundamental concepts and techniques on which our chess engine is based. This includes board representation, move generation, game trees, etc. Each section will provide an overview of the concepts and techniques used by our engine, and additional tools.

2.1. Board representation

The chessboard is where the game takes place and which serves as the foundation for all operations. In order to store a position setting with its pieces and other additional information like the side to move (the colour that has the turn of movement), the castling rights (the possibilities for each side to castle both short and long, explained in Section 1.3.5.1) or the fifty-move rule counter (explained in Section 1.3.4 Item 4), we can encounter different types of representations: piece centric, square centric and hybrid solutions. We chose to use bitboards as the primary representation, complemented by a piece list to store the piece on each square (piece centric representation). Additionally, the game state is stored in a bit field.

Bitboard also known as a bitset or bitmap, is a 64-bit data structure that efficiently represents a chessboard.

Bitboards can be used to represent different aspects of the board:

- All pieces: a bit is set to 1 for every square occupied by a piece, regardless of its type or color.
- Pieces by color: a bit is set to 1 for every square occupied by a piece of a specific color.
- Specific piece types: a bit is set to 1 for every square occupied by a specific type of piece.

This structure is highly efficient for operations such as move generation and attack detection, as bitwise operations (e.g., AND, OR, XOR, shifts) can be used to manipulate and query the board state quickly.

Bit field in constrast to a bitboard, uses a fixed number of bits within an integer to store multiple small values or flags.

For example, using a 64-bit bitfield, we can allocate 6 bits to store the number of pieces on the board, as $2^6 = 64$ possibilities. This means the bits would occupy an interval from X to $X + 5$ inclusive.

2.2. Move generation

An essential part of any chess engine is move generation. It involves generating all possible legal moves from a given position ensuring chess rules.

There are two types of move generation:

- Pseudolegal move generation: generates all moves without considering whether the king is left in check after the move. It requires additional filtering to remove illegal moves which is more computationally expensive than legal move generation.
- Legal move generation: generates only moves that are valid according to the chess rules, ensuring that the king is not left in check. The more accurate, the computationally more expensive it is.

We have preferred to use legal move generation because it ensures that the generated moves are correct.

Additionally, we have chosen to implement **magic bitboards** for this type of move generator, particularly for sliding pieces such as rooks, bishops, and queens. Magic bitboards use precomputed attack tables and bitwise operations. This approach significantly reduces the computational cost of move generation, enabling the engine to explore deeper levels of the game tree while maintaining accuracy and performance.

2.3. Game trees

Sequential games, such as chess or tic-tac-toe, where players take turns alternately, unlike simultaneous games, can be represented in a game tree or graph. In this representation, the root node is the main position from which we look for the best move, and each subsequent node is a possible option or game state, forming a tree-like structure. This tree has a height or depth that refers to the number of levels or layers in the tree, starting from the root node (the initial game state) and extending to the leaf nodes.

The depth of a chess game tree is important because it determines the extent to which it will be analysed and evaluated. A depth of 1 represents all possible moves for the current player or side to move, while a depth of 2 includes the opponent's responses to those moves. As the depth increases, the tree grows exponentially, making it computationally expensive to explore all possible states.

2.4. Search algorithms

There are different approaches to find the best move from a position. Some of these search algorithms are: Depth-First Search (DFS), Best-First Search (not to be confused with Breadth-First Search or BFS but they are related) and Parallel Search.

Note that these search algorithms are the foundation of more advanced and practical algorithms used today. However, explaining them is essential to understand the underlying principles.

Depth-First Search refers to the process of exploring each branch of a tree or graph to its deepest level before backtracking. Unfortunately, in chess, this cannot be possible because the number of possible moves grows exponentially with the depth of the search tree, leading to the so-called combinatorial explosion. To address this, depth-first search is often combined with techniques like alpha-beta pruning (discussed below) to reduce the number of nodes evaluated, making the search more efficient while still exploring the tree deeply. The following pseudocode illustrates the working of the DFS algorithm:

Listing 2.1: Pseudocode of the Depth-First Search algorithm.

```
1 Procedure DepthFirstSearch(Graph G, Node v):
2     Mark v as visited
3     For each neighbor w of v in G.adjacentEdges(v):
4         If w is not visited:
5             Recursively call DFS(G, w)
```

DFS visits nodes by marking them as visited (line 2) and recursively explores all adjacent nodes until no unvisited nodes remain (lines 3 to 5). It has a worst-case performance of $O(|V| + |E|)$ and worst-case space complexity of $O(|V|)$, with $|V| =$ number of nodes and $|E| =$ number of edges.

Best-First Search refers to the way of exploring the most promising nodes first. It is similar to a breadth-first search but prioritizes some nodes before others. They typically require significant memory resources, as they must store a search space (the collection of all potential solutions in search algorithms) that grows exponentially.

Listing 2.2: Pseudocode of the Best-First Search algorithm.

```
Procedure BestFirstSearch(Graph G, Node start, Node goal):
    Create an empty priority queue PQ
    Add start to PQ with priority 0
    Mark start as visited

    While PQ is not empty:
        Node current = Remove the node with the highest
            priority from PQ
        If current is the goal:
            Return the path to the goal
```

```
        For each neighbor w of current in G.adjacentEdges(
            current):
            If w is not visited:
                Calculate priority for w (e.g., using a
                    heuristic)
                Add w to PQ with the calculated priority
                Mark w as visited
```

In this case, the priority queue contains nodes along with their associated priorities, which are determined by a heuristic function.

Parallel Search refers to mulithreaded search, a technique used to accelerate search processes by leveraging multiple processors.

Next, we will explore some of the most used search algorithms in chess engines.

2.4.1. Minimax algorithm

The **minimax** algorithm is a decision making algorithm that follows DFS principles. It is based on the assumption that both players play optimally, with one player (the maximizer) trying to maximize his score and the other player (the minimizer) trying to minimize his score. It explores the game tree to evaluate all possible moves and determines the best move for the current player.

Listing 2.3: Pseudocode of the Minimax algorithm.

```
Procedure Minimax(Node position, Integer depth, Boolean
    maximizingPlayer):
    If depth == 0 or position is a terminal node:
        Return the evaluation of the position

    If maximizingPlayer:
        Integer maxEval = -Infinity
        For each child of position:
            Integer eval = Minimax(child, depth - 1, False)
            maxEval = max(maxEval, eval)
        Return maxEval
    Else: // minimizingPlayer
        Integer minEval = +Infinity
        For each child of position:
            Integer eval = Minimax(child, depth - 1, True)
            minEval = min(minEval, eval)
        Return minEval
```

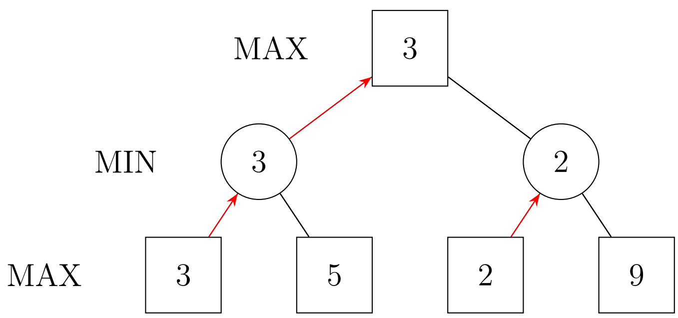


Figure 2.1: Example of minimax.

In this example, white is represented by square nodes and black by circle nodes. Note that this example is a binary tree, but there might be more moves or nodes in a real scenario. Each of them wants to maximize or minimize their respective final value in each position. For the leftmost pair of leaf nodes with values of 3 and 5, 3 is chosen because black tries to get the lowest score between them. Then, the other pair of leaf nodes with values of 2 and 9, 2 is chosen for the same reason. Lastly, at the root node, white selects 3 as the maximum number between 3 and 2.

2.4.2. Alpha-beta pruning

Alpha-beta pruning is an optimization of minimax that reduces significantly the number of evaluated nodes in the game tree. It uses two values, alpha and beta, to discard branches that cannot influence the final decision, improving the efficiency.

Listing 2.4: Pseudocode of the Alpha-Beta Pruning algorithm.

```
Procedure AlphaBeta(Node position, Integer depth, Integer alpha, Integer beta, Boolean maximizingPlayer):
    If depth == 0 or position is a terminal node:
        Return the evaluation of the position
    If maximizingPlayer:
        Integer maxEval = -Infinity
        For each child of position:
            Integer eval = AlphaBeta(child, depth - 1, alpha, beta, False)
            maxEval = max(maxEval, eval)
            alpha = max(alpha, eval)
            If beta <= alpha:
                Break // Beta cutoff
        Return maxEval
    Else: // minimizingPlayer
        Integer minEval = +Infinity
        For each child of position:
            Integer eval = AlphaBeta(child, depth - 1, alpha, beta, True)
            minEval = min(minEval, eval)
            beta = min(beta, eval)
            If beta <= alpha:
                Break // Alpha cutoff
        Return minEval
```

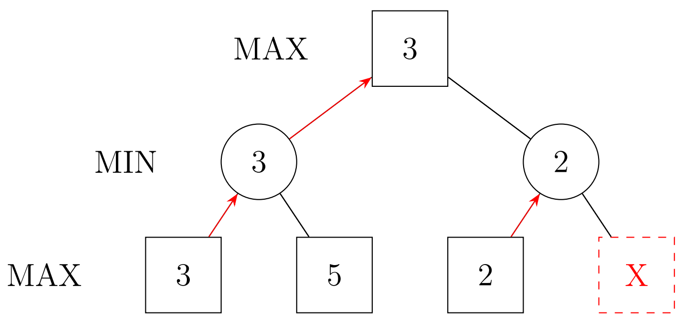


Figure 2.2: Example of alpha-beta pruning.

In this other example, the red dashed node is pruned because it cannot influence the final decision independently of its value. If its value is less than or equal to 2, it will never improve the previously analyzed value of 3. On the other hand, if its value is greater than 2, black will still choose 2 to minimize the score. Another formal way to explain this is by using *alpha* and *beta* values:

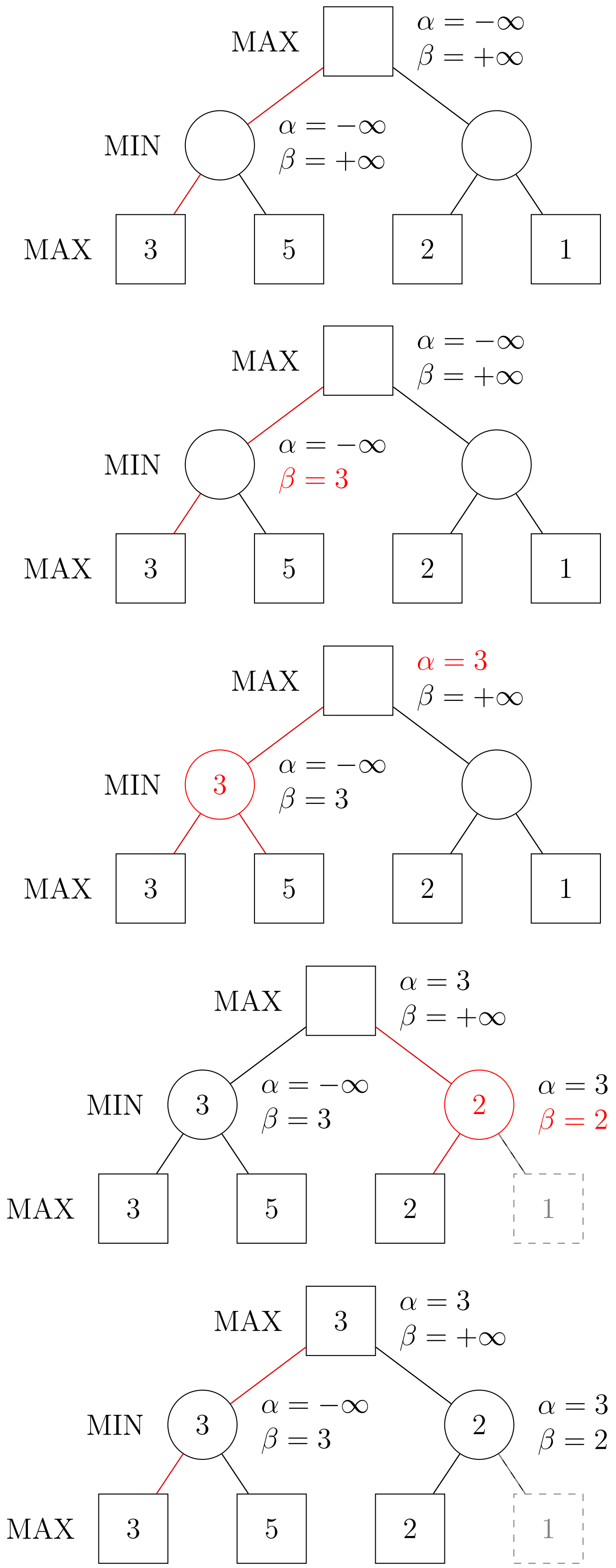
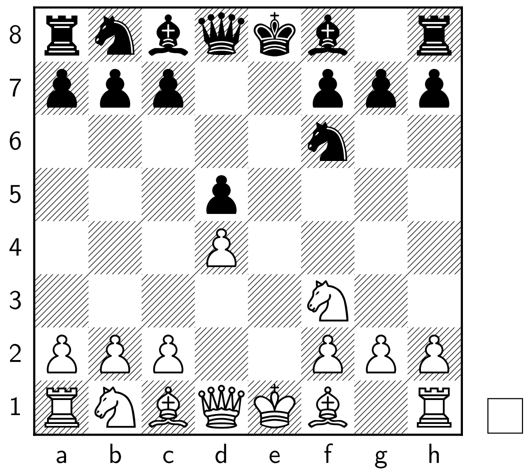


Figure 2.3: Example of alpha-beta pruning with α and β values.

2.4.2.1. Alpha-Beta Enhancements

The alpha-beta algorithm has been further improved over time with various enhancements to increase the overall efficiency. Some of these are: transposition tables, iterative deepening, aspiration windows, quiescence search, move ordering...

Take into consideration that many positions can be reached in different ways. This is formally known as transpositions. Just like in dynamic programming, the evaluations of different positions are stored in a structure, the **transposition tables**, to avoid repeating the process of searching and evaluating, which improves efficiency. Take the following example:



French Defense:
1. e4 e6 2. d4 d5 3. exd5 exd5 4. Nf3 Nf6

Petrov Defense:
1. e4 e5 2. Nf3 Nf6 3. Nxe5 d6 4. Nf3 Nxe4 5. d3 Nf6 6. d4 d5

Figure 2.4: Example of transposition.

Both games reach to the same position, although they involve a different number of moves.

In order to store these different positions and access them in a map or dictionary structure, there is a need for a unique and efficient way to index positions: **Zobrist hashing**. Zobrist hashing maps a large number of possible positions to a fixed-size hash value, which can lead to collisions, as different positions may produce the same hash. To handle these collisions, storing additional information like the depth is used to verify the correctness of the entry. In some cases, overwriting the older or less relevant entries can be also useful.

As it is mentioned in `https://www.chessprogramming.org/Search`, *«Depth-first algorithms are generally embedded inside an iterative deepening framework for time control and move ordering issues.»*. **Iterative deepening** refers to the combination of DFS with limited depth searches. It performs successive searches by increasing the depth limit at each iteration, allowing you to obtain partial results quickly and improve accuracy over time.

An important concept related to iterative deepening is the use of **aspiration windows**. Their main objective is to reduce the search space by narrowing the search bounds. In other words, by adjusting *alpha* and *beta* values in each iteration of the iterative deepening. If the value of the evaluation in the iteration falls outside this range or window, a re-search is performed with a wider window to ensure accuracy.

Another critical concept is **quiescence search**. This is a search technique used to address the horizon effect at the end of the search. Simply stopping the search at a fixed or desired depth and evaluating the position can lead to inaccuracies, as critical tactical moves, such as captures, are often overlooked.

> *Consider the situation where the last move you consider is QxP. If you stop there and evaluate, you might think that you have won a pawn. But what if you were to search one move deeper and find that the next move is PxQ? You didn't win a pawn, you actually lost a queen. Hence the need to make sure that you are evaluating only quiescent (quiet) positions.*[1]

Finally, alpha-beta algorithm could not perform well without **move ordering**. It is important to ensure that best moves are searched first and to reduce the search space of the game tree. Some of the techniques for move ordering are: Most Valuable victim - Least Valuable Aggressor (MVV-LVA) for captures and killer moves for non-captures.

MVV-LVA is a heuristic that prioritizes capturing moves by evaluating the value of the piece being captured (the victim) and the value of the piece performing the capture (the aggressor). The goal is to maximize the gain while minimizing the risk. The following example reflects this:

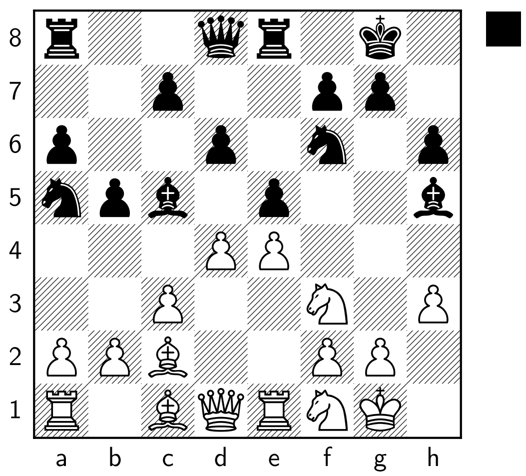


Figure 2.5: Example of MVV-LVA.

In this position, it is black's turn to play after white has moved $d4$, black has the option to capture the pawn on $d4$ with the pawn on $e5$ or with the bishop on $c5$.

[1] `https://www.chessprogramming.org/Quiescence_Search`

Between the two capturing movements, the best option is to take $d4$'s pawn with $e5$'s pawn because the pawn has less value than the bishop. Then, after $exd4$, white can re-capture with the pawn on $c3$ or the knight on $f3$. If black had taken the pawn with the bishop, white would have won a bishop for a pawn. This simple heuristic that orders what is best in capturing movements for each side can efficiently evaluate tactical exchanges and focus on moves that are more likely to yield a favorable outcome.

Killer moves is a heuristic that considers moves that produced cutoffs or pruning while searching. When the engine encounters a similar position at the same depth later in the search, it will prioritize the last move that caused a cutoff, potentially leading to faster pruning.

Young Brothers Wait Concept is a parallel search algorithm designed to optimize the distribution of work among multiple threads. This is particularly effective in alpha-beta pruning, where the search tree is explored selectively. It is divided into two phases: the principal variation move and the wait concept. The principal variation is searched sequentially by the main thread which ensures that the most promising move is evaluated first. Then, once the first move is evaluated, the remaining moves are distributed among multiple threads for parallel evaluation.

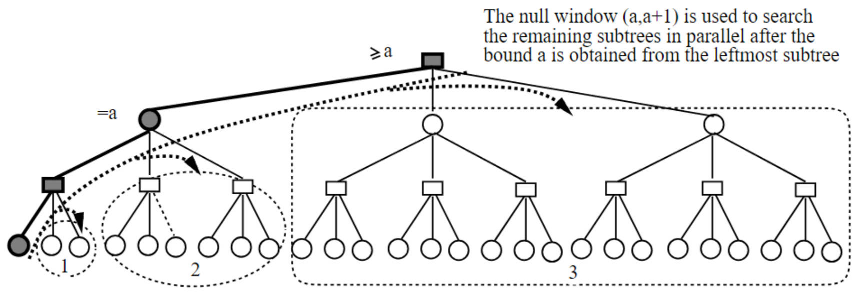


Figure 2.6: Principal variation splitting. Gao and Marsland (1996)

2.5. Evaluation

For each position, a numerical value is assigned representing how favorable the position is for one side: positive (+) for white and negative (-) for black. These symmetric values were first formulated by Shannon (1950) and which guide the engine in selecting the best moves during the search process. Generally, this value is expressed in centipawns (cp) that represents one hundredth of a pawn's value. There are two ways to implement it:

- Traditional hand-crafted evaluation
- Multi-layer neuronal networks

The use of neural networks is beyond the scope of the project, so traditional evaluation has been implemented and discussed below.

When teaching new chess players how to evaluate their position, assigning a simple value to each piece on board is an effective approach.

Piece	Value
Pawn	100
Knight	320
Bishop	330
Rook	500
Queen	950
King	∞

Table 2.1: Standard values assigned to chess pieces in centipawns.

Over the time, there have been different point values for each piece. A table can be found in `https://www.chessprogramming.org/Point_Value` in *Basic values* section.

By summing up the piece values for each side, we end up the concept of material. For example, if white has 1 knight and 1 bishop, while black has 2 bishops, the material calculation is as follows:

$$(+)(1 \times \text{knightValue} + 1 \times \text{bishopValue}) - (2 \times \text{bishopValue}) \tag{2.1}$$

Substituting the standard piece values in centipawns (cp):

$$(1 \times 320 + 1 \times 330) - (2 \times 330) = -10\,\text{cp} \tag{2.2}$$

This result indicates that black has a slight material advantage of 10 centipawns.

There are other material considerations like *bonus for the bishop pair* depending on the position that can be advantageous to control squares of different color, or insufficient material, which refers to situations where neither side has enough pieces to deliver checkmate. This is also mentioned in Section 1.3.4.

However, the higher the level one aims to reach in chess, the greater the need to evaluate other aspects. For instance, there are squares in which the pieces have less value because of their activity.

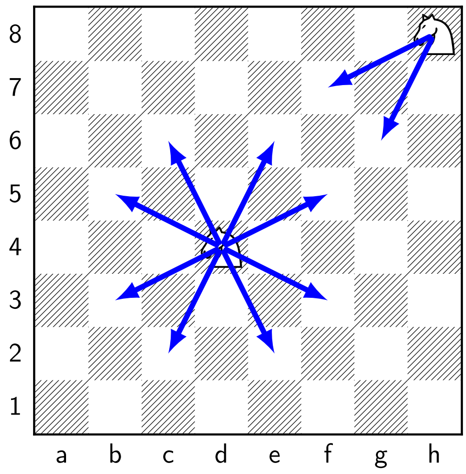


Figure 2.7: Knight's movement on corner square and center square.

A knight on a corner has only 2 moves, while one in the center can reach up to 8.

A way to add this to the evaluation is to have precomputed piece-square tables like the following one for the bishop:

-20	-10	-10	-10	-10	-10	-10	-20
-10	0	0	0	0	0	0	-10
-10	0	5	10	10	5	0	-10
-10	5	5	10	10	5	5	-10
-10	0	10	10	10	10	0	-10
-10	10	10	10	10	10	10	-10
-10	5	0	0	0	0	5	-10
-20	-10	-10	-10	-10	-10	-10	-20

Figure 2.8: Chessboard with precomputed piece-square values for the bishop.

It is important to consider the game phase to select these piece-square tables, especially in the endgame phase. These phases are:

- Opening: where players develop their minor pieces (knights and bishops), control the center of the board and try to ensure king safety.
- Middlegame: where players try to create tactical opportunities and attack the opponent's position once they have developed their pieces and secured their king.
- Endgame: where players have fewer pieces on the board and the focus is on promoting pawns and achieving checkmate.

In the endgame phase, when there are no queens, king activity becomes extremely significant in supporting the advancement of the pawns.

In relation to the above and Section 2.2, the mobility score measures the number of legal moves a piece can make from its current position. As it is mentioned in `https://www.chessprogramming.org/Mobility`,

> *. . . the more choices you have at your disposal, the stronger your position.*

This score is calculated as the number of legal moves adjusted for certain factors like blocked moves by friendly pieces (pieces of the same color as the current piece) and enemy attacks (the squares controlled by the opponent).

During opening and middlegame, king safety is an important factor to consider. When the king has castled, it is crucial to maintain the pawns nearby to shield it from attacks. Generally, it is preferable to keep the pawns unmoved or advanced by only one square. This is called king's shield and a bonus is awarded for each pawn that still has not moved from the castling area.

On the contrary, king safety penalization can also mean to punish the player based on the number and type of enemy attacks targeting the area around the king.

All these ensures that positions with exposed kings are evaluated as less favorable. If there are open (no pawns of either color) or semi-open (no pawns of one color but at least one pawn of the opposite color) files near the king, the position is penalized further, as these files provide attacking opportunities for enemy rooks and queens.

2.6. How can we determine the strength of our engine?

This can be answered by playing against other engines and analyzing the results. The most common way to do this is by using the Computer Chess Rating Lists which ranks chess engines based on their performance in various tournaments and matches. By the time being, we have chosen to compare different versions of the engine with Stockfish, currently ranked as the number one on the list. Continue reading to learn about the used tools and read about the work behind it in Section 3.4.

2.6.1. Profiler

TODO: explain

2.6.2. CustomTkinter: Python UI-library

TODO: explain

2.6.3. Cutechess

TODO: explain

2.6.4. Stockfish

TODO: explain

2.6.5. GitHub Actions and workflows

TODO: explain

Chapter 3

Work description

We designed a simple structure to organize all the scripts. As we are implementing different versions of the engine, there are specific modules that are compulsory to have well distinguished and possibly selected between them. Those modules are presented in the following Section 3.1.

Next, the key point is how to represent each required structure, including the board, chess pieces, precomputed tables, transposition tables, etc., as discussed in Section 3.2.

3.1. Modules

Each module is responsible for a specific aspect of the chess engine's functionality. The good part about this modular design is that it ensures clarity, maintainability, and the ability to test and improve individual components independently even more so when it comes to development with more people.

3.1.1. Board

This module handles the representation of the chessboard, as previously mentioned in Section 2.1, and the state of the game. It includes:

- Representation of the chessboard with its pieces using bitboards for efficiency.
- Functions to make and unmake moves, including special moves like castling and en passant.
- Updating game state variables, such as castling rights or the half-move counter.

This module is vital, as it provides the data structures and operations required by other modules.

3.1.2. Evaluation

This module returns the final evaluation, a positive or negative number, depending on the board.

3.1.2.1. Move Generator

This module generates the list of legal moves generated depending on the board while updating relevant information such as king's danger masks, pinned pieces or checking pieces, and calculating each piece moves. All chess rules are taken into consideration.

3.1.3. Move ordering

This module uses a heuristic to return a prioritized list of moves to improve the efficiency of the search.

3.1.4. Search

This module must return a set of search results, each containing the best move and the achieved evaluation at a specific depth.

3.1.5. UCI

This module is responsible for processing commands to invoke each module, following UCI standards.

3.2. Code implementation

Although the previous algorithms were presented in pseudocode, from this point onward, all code implementations will be written in C++.

3.2.1. Data representation

3.2.1.1. Square

There are 64 possible squares on a chessboard so the first thought is to use a 6-bit structure which seem like a perfect match ($2^6 = 64$). However, modern processors are optimized to work with data sizes aligned to multiples of 8 bits (1 byte). Then, using 6 bits would require packing the data into more complex structures, which could introduce additional overhead in terms of bit manipulation and memory access. We preferred clarity and performance over micro-optimizations that complicate readability so we used a `uint8_t` to describe a square.

Moreover, masks are extremely useful for efficiently identifying and manipulating the squares on a chessboard using bitwise operations. Some of these masks are defined as constants in the code and others are calculated during compilation time. For example, they can be used to identify the column of a square or simply placing a new piece on board.

```
class Square {
    uint8_t sq_value;

    constexpr bool is_valid() const {
        return sq_value < 64U;
    }

    constexpr uint64_t mask() const {
        return is_valid() ? 1ULL << sq_value : 0ULL;
    }

    // Calculating the column of a square (sq_value % 8)
    constexpr int col() const {
        return is_valid() ? sq_value & 7U : COL_INVALID;
    }
}

// Placing a piece by setting the bit corresponding to the
// square to 1
const uint64_t mask = square.mask();
bitboard_all |= mask;
```

Just to clarify, `sq_value` must be a value between 0 and 63, inclusive, for the 64 squares on a chessboard. To calculate the column of a square, an AND operation is applied: `sq_value & 7U` extracts the 3 least significant bits, which correspond to the column of the square. Columns are enumerated from 0 (A) to 7 (H).

3.2.1.2. Piece and PieceType

They are simply enumerations where each piece and piece type correspond to an integer number to improve code readability.

```
enum class Piece : int
{
    W_PAWN = 0,
    W_KNIGHT = 1,
    W_BISHOP = 2,
    ...
    B_QUEEN = 10,
    B_KING = 11,
    EMPTY = 12,
    NUM_PIECES = 13
};

enum class PieceType : int
{
    PAWN = 0,
```

```
    KNIGHT = 1,
    BISHOP = 2,
    ROOK = 3,
    QUEEN = 4,
    KING = 5,
    EMPTY = 6,
    NUM_PIECES = 7
};
```

3.2.1.3. Move and MoveType

There are four types of moves: normal, promotion, en passant, and castling. These are represented using an integer-based enumeration:

```
enum class MoveType
{
    NORMAL = 0,
    PROMOTION = 1,
    EN_PASSANT = 2,
    CASTLING = 3
};
```

Meanwhile, moves are represented as a combination of two squares (the origin square and the destination square), 2 bits for the promotion piece, and 2 bits for the move type, all encoded in a `uint16_t`. In this case, each square is represented using 6 bits, resulting in a total of 16 bits ($6 + 6 + 2 + 2 = 16$ bits). Each bit field has a unique mask and its specific shift, which must remain unchanged throughout the development.

3.2.1.4. Game State

The game state must store important information during the game, including the Zobrist hash key of the current position, the number of moves, the en passant square, the castling rights for each side and color, the side to move, the last captured piece, the fifty-move rule counter, the number of pieces, and a flag indicating whether the attacks are updated.

There are two `uint64_t` variables: one for the Zobrist hash key and the other for the remaining bit fields:

```
class GameState {
    uint64_t zobrist_key;

    // 50 : attacks_updated : 1 if updated, 0 if not
    // 43-49 : num_pieces : 0 to 64 pieces
    // 35-42 : fifty_move_rule_counter : if counter gets to
    // 100 then game is a draw.
    // 32-34 : last_captured_piece : PieceType::Empty if last
    // move was not a capture.
    // 31 : side_to_move : 0 if white, 1 if black.
```

```
    // 30 : castle_king_white : 1 if available, 0 if not.
    // 29 : castle_queen_white : 1 if available, 0 if not.
    // 28 : castle_king_black : 1 if available, 0 if not.
    // 27 : castle_queen_black : 1 if available, 0 if not.
    // 26-20 : en_passant_square : 0-63 if available, >=64 if
    // not available
    // 19-0 : move_number : 0-1048575 number of moves in the
    // game.
    uint64_t state_register;
}
```

3.2.1.5. Board

For the chessboard, as previously mentioned, bitboards are represented as 64-bit structures using `uint64_t`. Since the sign is not relevant, an unsigned type is used.

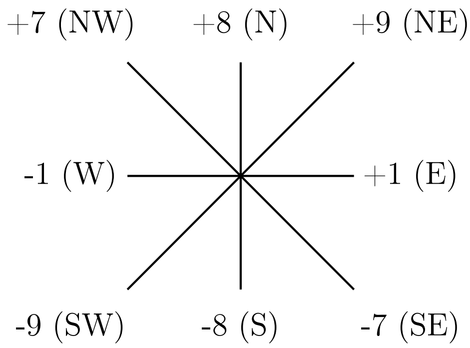
The bitboard representation follows the Little-Endian Rank-File Mapping convention also called as LERF. In this mapping, each bit in the 64-bit integer corresponds to a square on the chessboard where the least significant bit (0) is square `A1`, and the most significant bit (63) is square `H8`.

8	56	57	58	59	60	61	62	63
7	48	49	50	51	52	53	54	55
6	40	41	42	43	44	45	46	47
5	32	33	34	35	36	37	38	39
4	24	25	26	27	28	29	30	31
3	16	17	18	19	20	21	22	23
2	8	9	10	11	12	13	14	15
1	0	1	2	3	4	5	6	7
	A	**B**	**C**	**D**	**E**	**F**	**G**	**H**

Figure 3.1: Little-Endian Rank-File Mapping with Coordinates.

There are bitboards for all pieces (`bitboard_all`), for each piece color (`bitboard_color[0]` and `bitboard_color[1]`), and for each piece type. (`bitboard_piece[Piece]` like `bitboard_piece[Piece::W_QUEEN]`)

To identify ray directions on the board, we used the compass rose:



This means that, to get the numerical value that identifies the square to the north-east of a given square, you only need to add 9. For example, given the square $f6$ (45), the north-east square $g7$ has a value of 54 ($45 + 9 = 54$). It is really effective for sliding pieces to calculate their attacks.

3.2.1.6. Transposition table

The transposition table contains a list of entries. These entries are defined as a storage of information about a specific chess position, including its Zobrist key, evaluation score, best move, node type, and search depth.

The use of this will be discussed in Section 3.3.1.

3.2.1.7. History

It contains a circular array (`uint64_t positions[HISTORY_MAX_SIZE]` with a position index that runs circularly `int next_position_index`) to store the hashes of previous board positions, ensuring efficient memory usage and quick access. This is essential for detecting threefold repetitions. Additionally, it provides functions to add new positions, remove the most recent ones, and clear the history when needed. The array is implemented with a fixed size (`HISTORY_MAX_SIZE`) of 128, defined as a power of two, to optimize indexing operations.

The detection of threefold repetition is performed by comparing the hash of the current position with the hashes stored in the history. If the same hash appears at least twice, it indicates that the position has been repeated three times.

3.2.1.8. Move generator information

TODO

3.2.1.9. Precomputed data

TODO

3.2.2. Evaluation

In addition to the basic information that is mentioned in Section 2.5, in order to calculate the game phase, we can perform a dynamic evaluation (tampering evaluation): evaluate twice, once as if we were in the middlegame and once as if we were in the endgame. The final evaluation will be the sum of both, but each with a different weight. To do this, we calculate the middlegame percentage (24 pieces means 100 % middlegame) and the endgame percentage (0 pieces = 100 % endgame).

$$\text{Evaluation} = \mathit{middlegame}\,\% \cdot \mathit{eval\_middlegame} + \mathit{endgame}\,\% \cdot \mathit{eval\_endgame}$$

TODO: insert diagram of pawn PST in middlegame and in the endgame to compare
TODO: insert code only with piece counting evaluation and adding king safety and mobility

3.2.3. Move generator

Calculating the legal moves in a chess position is a more difficult and tedious task than it might seem, mainly due to the unintuitive rules of en passant and castling, and it is also difficult to restrict the moves of pinned pieces. Jones (2023)

To create an efficient move generator, we will be using bitboards. The steps to generate legal movements efficiently are the following:

- Calculate the bitboard of attacked squares by the waiting side.
- Calculate bitboard of pinned pieces.
- Generate the legal moves of each piece of the side whose turn is, knowing that the king cannot move to any attacked square and that pinned pieces can only move in the direction of the pin.
- Generate the special legal moves like en passant and castling.

TODO: insert some of the code of move generator

3.2.4. Move ordering

Order the legal moves from most to least likely to be the best move in the position. The sooner we explore the best move, the more branches of the tree will be pruned. To do this, we use the MVV-LVA heuristic (most valuable victim, least valuable aggressor). We give higher scores to capturing a low-value piece over a higher-value piece. Capturing a queen with a pawn scores highly. We also give a bonus to piece promotions.

TO DO: insert code of move ordering

3.2.4.1. Killer moves

A killer move is a non-capturing (quiet) move that previously caused a beta-cutoff during the search in a sibling node or any other branch at the same depth in the game tree. These moves are often strong candidates, as they have previously led to pruning in similar positions. Promoting them early in the move ordering increases the chances of early cutoffs, which improves search efficiency.

To take advantage of this heuristic we store up to two killer moves for each search depth. During move ordering, these killer moves are given a bonus score, allowing them to be explored before other quiet moves.

TODO: insert code of killer moves

3.2.5. The Core: Search Algorithm

The core of the chess engine is its search algorithm, in this case alpha-beta pruning algorithm. The entire game tree is generated up to a selected maximum depth. At each node, the next player evaluates the position and during execution, the values of alpha and beta are updated. Pruning is performed when a branch of the tree is detected as irrelevant because the evaluation being examined is worse than the current value of alpha or beta for MAX or MIN, respectively.

Therefore, the following events happen at each node of the tree:

- Check if we are at an end node because of a checkmate, a draw by triple repetition, the 50-move rule, or because we have reached the maximum selected depth.
- Evaluate the position: a positive value (+) means that White has an advantage, and a negative value (-) means that Black has an advantage. A limit is set that represents mate in one; we have arbitrarily chosen 3,200,000.
- Generate legal moves: create a list of every possible legal move in the position.
- Order the legal moves: from greatest to least intuition of being the best move for the position. The sooner we explore the best move, the more branches of the tree will be pruned.
- Explore each of the legal moves from the position in order, update the evaluation, the value of alpha and beta, and check if we can perform pruning.

TODO: insert code of alpha beta pruning

3.2.5.1. Search: iterative deepening

At what depth do we decide to search? Actually, the simplest thing is to perform an infinite search, first searching at depth 1, then 2, then 3... to infinity. The engine will update the evaluation and the best move for the position in each iteration. Simply by signaling *stop* the search will stop.

TODO: insert code of iterative deepening

3.2.5.2. Search: horizon effect, quiescence search

What happens if, upon reaching maximum depth, we evaluate the position in the middle of a piece exchange? For example, if a queen captures a pawn. It will seem like we have won a pawn, but on the next move, another pawn captures the queen, and now we lose a queen. This is known as the horizon effect. To avoid this, when we reach the end of the tree at maximum depth, we must extend the search to include only piece captures. This is known as quiescence search.

The purpose of this search is only to stop the search and evaluate quiet positions, where there is no capture or tactical movement.

TODO: insert code of quiescence search

3.3. Improvements

Some improvements and new structures were later added for different versions: transposition tables, Zobrist hashing, table entries, PEXT instructions, search multithread and search reductions that will be discussed and analysed below.

3.3.1. Transposition Table

The basic implementation of the chess engine generates a large amount of redundant calculations due to transpositions: situations in which the same board position is reached through different sequences of moves in the game tree.

Taking advantage of the concept of dynamic programming, we are going to create a look-up table of chess positions and its evaluation. So if we encounter the same position again, the evaluation is already precalculated. However, we ask ourselves the following question: how much space does the look-up table take up if there are an astronomical amount of chess positions? What we can do is assign a hash to each position and make the table index the last bits of the hash. The larger the table, the less likely access collisions will be. We also want a hash that is fast to calculate and has collision-reducing properties; for this, we will use the Zobrist hashing technique in the following subsection.

TODO: insert code of transposition table

3.3.1.1. Zobrist Hashing

Zobrist Hashing (Zobrist (1970)) is a technique to transform a board position of arbitrary size into a number of a set length, with an equal distribution over all possible numbers invented by Albert Zobrist.

To generate a 64-bit hash for a position, the following steps are followed:

- There are 12 different types of chess pieces. For each of the 64 squares on the board, we generate 12 random 64-bit integers. That is, each piece-square combination is assigned a unique random value. This initialization step is performed only once when the program starts.
- The hash value for a given position is computed by performing the XOR operation between the hash accumulator and the random value corresponding to each piece on its square.
- In addition to the pieces, we also include:
  - A random value for the side to move (white or black),
  - One random value per square to account for the possibility of an *en passant* capture.
- These random values are carefully chosen so that even slightly different positions produce very different hash values. This greatly reduces the chance of collisions.
- The XOR operation is used not only because it is computationally inexpensive, but also because it is reversible. This means that when a move is made or undone, we can update the hash incrementally by applying XOR only to the affected squares, without needing to recompute the entire hash.

TODO: insert code of zobrist hashing

3.3.1.2. Table Entry

Each entry in the transposition table stores the following information:

- **Zobrist Hash**: The full 64-bit hash of the position. This is used to verify that the entry corresponds to the current position and to detect possible index collisions in the table.
- **Evaluation**: The numerical evaluation of the position, as computed by the evaluation function.
- **Depth**: The depth at which the evaluation was calculated. A deeper search could potentially yield a more accurate evaluation, so this value helps determine whether a new evaluation should overwrite the existing one.
- **Node Type**: Indicates the type of node stored:

- `EXACT` the evaluation is precise for this position.
- `UPPERBOUND` the evaluation is an upper bound, typically resulting from an alpha cutoff.
- `LOWERBOUND` the evaluation is a lower bound, typically resulting from a beta cutoff.

TODO: insert code of table entry

3.3.1.3. Collisions

As discussed earlier, index collisions in the transposition table are handled by verifying the full Zobrist hash stored in the entry. However, it is still theoretically possible for a full hash collision to occur, that is two different positions producing the same hash.

This scenario is extremely rare. With 64-bit hashes, there are 2^{64} possible unique values, which is more than sufficient for practical purposes. In the unlikely event of a true hash collision, it could result in an incorrect evaluation being reused for a different position.

3.3.1.4. Analysis

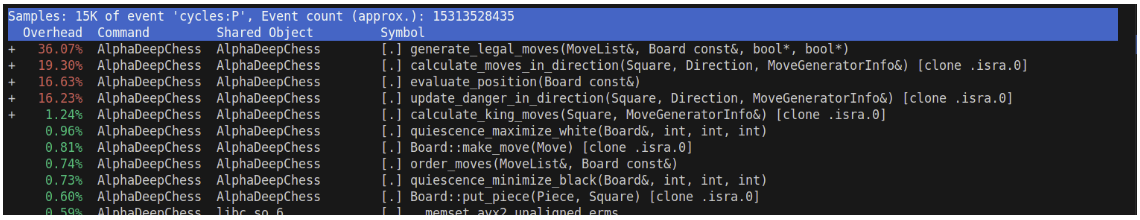
To evaluate the improvement introduced by the transposition table, we conducted a 100-game tournament against the basic version of the engine. We selected 50 random starting positions from an opening book and played each position twice, alternating colors to ensure fairness. Each bot has 4 seconds to think per move.

64MB Transposition Table bot vs basic bot

We see a substantial improvement by adding the transposition table with 46 wins versus 32 losses.

3.3.2. Move generator with Magic Bitboards and PEXT instructions

To identify potential performance bottlenecks, we performed profiling on the engine.



Profiling results show that most part of the total execution time is spent in the generate legal moves function. Therefore, optimizing this component is expected to lead to significant performance improvements.

3.3.2.1. Magic bitboards

We can create a look up table of all the rook and bishop moves for each square on the board and for each combination of pieces that blocks the path of the slider piece (blockers bitboard). Basically we need a hash table to store rook and bishop moves indexed by square and bitboard of blockers. The problem is that this table could be very big. Kannan (2007)

Magic bitboards technique used to reduce the size of the look up table. We cut off unnecesary information in the blockers bitboard, excluding the board borders and the squares outside its attack pattern.

A **magic number** is a multiplier to the bitboard of blockers with the following properties:

- Preserves relevant blocker information: The nearest blockers along a piece's movement direction are preserved. *Example:* Consider a rook with two pawns in its path:

  Rook –> –> –> [Pawn1] [Pawn2]

  In this case, only 'Pawn1' blocks the rook's movement, while 'Pawn2' is irrelevant.

- Compresses the blocker bitboard, pushing the important bits near the most significant bit.

- The final multiplication must produce a unique index for each possible blocker configuration. The way to ensure the uniqueness is by brute force testing.

As illustrated in Figure 3.2, we aim to compute the legal moves of the white rook in the given position. In practice, the only pieces that truly block the rook's path are those marked with a red circle.

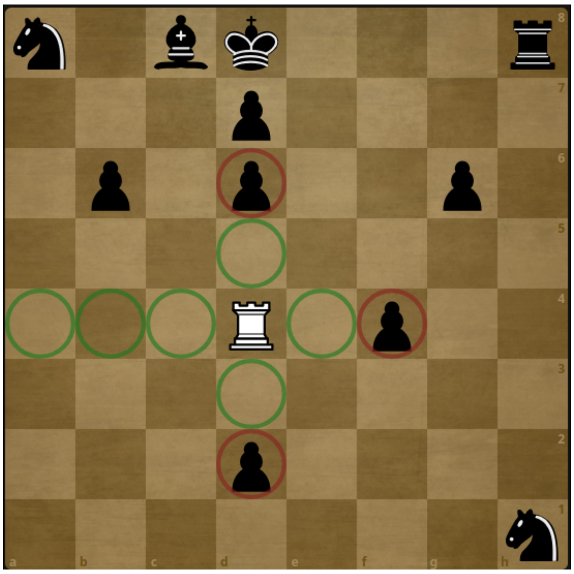
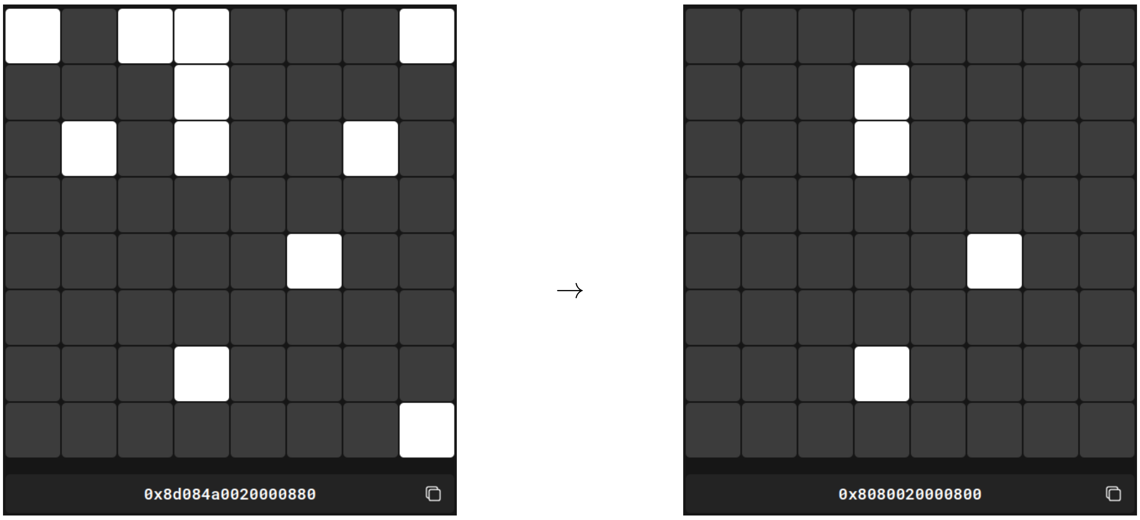


Figure 3.2: Initial chess position with white rook and blockers

First, we mask out all pieces outside the rook's attack pattern or on the board borders, as shown in Figure 3.3.

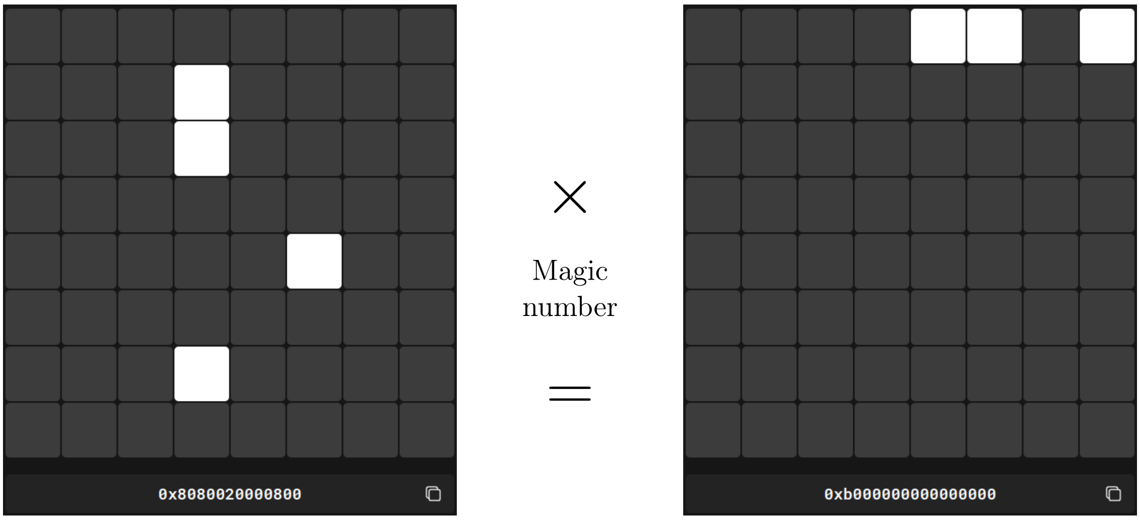


Original blockers bitboard

Masked blockers bitboard

Figure 3.3: Pre-processing of the blockers bitboard

As illustrated in Figure 3.4, the masked blockers bitboard is then multiplied by the magic number. The result retains only the three relevant pawns that obstruct the rook's movement, pushing them toward the most significant bits.



Masked blockers bitboard Multiplied blockers bitboard

Figure 3.4: Multiplication by magic number to produce an index

Next, we compress the index toward the least significant bits by shifting right by 64−`relevant_squares`. The number of relevant squares varies per board square; Figure 3.5 shows this for the rook:

```
/**
 * @brief ROOK_RELEVANT_SQUARES
 *
 * number of squares where the rook could move from
 * each square minus the board border squares.
 */
static constexpr int ROOK_RELEVANT_SQUARES[NUM_SQUARES] = {
    12, 11, 11, 11, 11, 11, 11, 12,
    11, 10, 10, 10, 10, 10, 10, 11,
    11, 10, 10, 10, 10, 10, 10, 11,
    11, 10, 10, 10, 10, 10, 10, 11,
    11, 10, 10, 10, 10, 10, 10, 11,
    11, 10, 10, 10, 10, 10, 10, 11,
    11, 10, 10, 10, 10, 10, 10, 11,
    12, 11, 11, 11, 11, 11, 11, 12
};
```

Figure 3.5: Number of relevant squares for each rook square

The final index is thus computed as

$$\text{index} = (\texttt{bitboard\_of\_blockers} \times \texttt{magic\_number}) \gg (64 - \texttt{relevant\_squares}) .$$

3.3.2.2. PEXT instruction

The `PEXT` (Parallel Bits Extract) instruction—available on modern x86_64 CPUs—extracts bits from a source operand according to a mask and packs them into the lower bits of the destination operand. Hilewitz and Lee (2006) It is ideally suited for computing our table index.

Figure 3.6 illustrates how `PEXT` works: it selects specific bits from register `r2`, as specified by the mask in `r3`, and packs the result into the lower bits of the destination register `r1`.

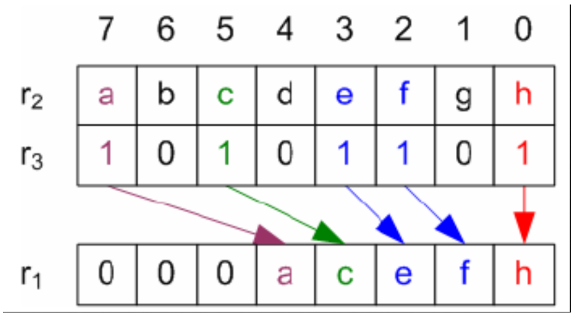
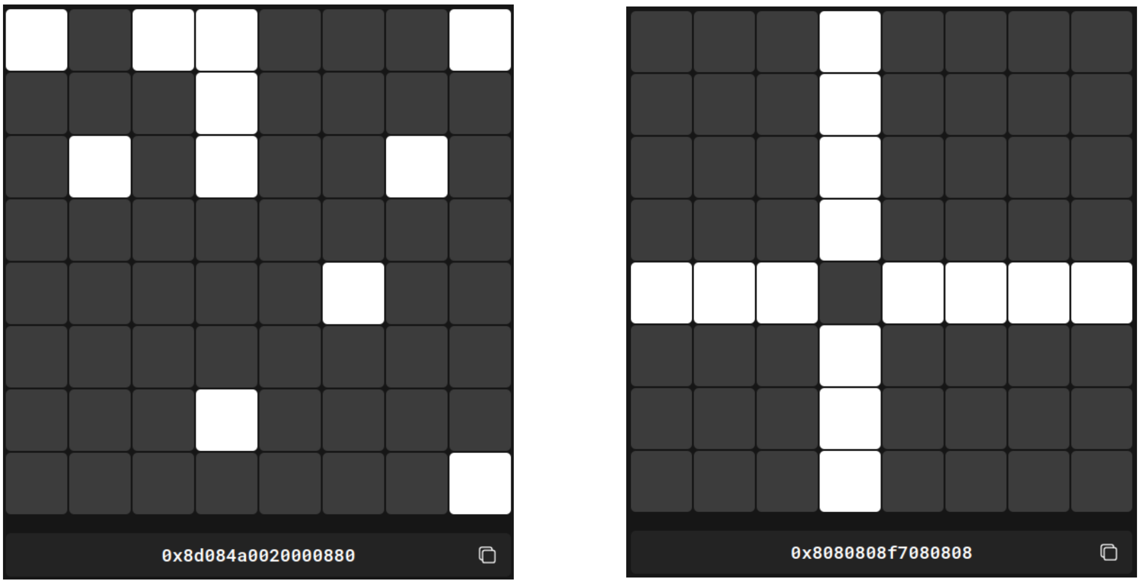


Figure 3.6: Example of the `PEXT` instruction: extracting bits from `r2` using `r3` as a mask, and storing the result in `r1`

For our previous example 3.2 we just need the full bitboard of blockers and the attack pattern of the rook. 3.7



Blockers bitboard

Rook attack mask

Figure 3.7: Blockers and attack mask for PEXT extraction

The final index used to access the lookup table is calculated using the `pext` instruction as follows:

$$\text{index} = \texttt{\_pext\_u64}(\texttt{blockers}, \texttt{attack\_pattern})\,.$$

To maintain compatibility and performance across different hardware platforms, we provide two implementations:

- If `PEXT` support is detected at compile time, the engine uses it to compute the index directly.
- Otherwise, the engine falls back to the Magic Bitboards approach using multiplication and bit shifts.

3.3.2.3. Analysis

To evaluate the improvement in the move generator, we conducted the same 100 game match vs the basic bot version.

Move generator with PEXT instructions bot vs basic bot

Huge improvement with 64 wins versus 22 losses.

...

3.3.3. Evaluation with King Safety and piece mobility

It is often beneficial to evaluate additional aspects of a position beyond simply counting material. We introduce the following positional evaluation parameters:

1. King Shield Bonus: The king is typically safer when protected by friendly pawns in front of it. We assign a bonus in the evaluation score for each allied pawn positioned directly in front of the king.

2. King Safety Penalty: For each square within a 3×3 area surrounding the king that is attacked by enemy pieces, we apply a penalty to reflect increased vulnerability.

3. Piece Mobility: Greater piece mobility is generally indicative of a stronger position. Each piece receives a bonus for every available move to a square that is not attacked by enemy pawns.

3.3.3.1. Analysis

To evaluate the improvement in the new evaluation, we conducted the same 100 game match vs the basic bot version.

King Safety and Piece mobility evaluation bot vs basic bot

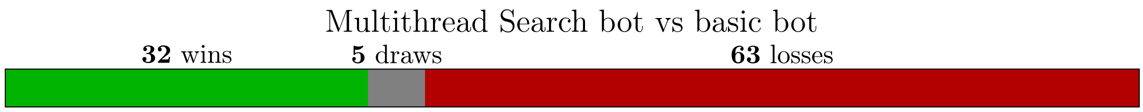
The results are slightly worse compared to the match using the material-only evaluation, with 8 more losses than before. This may be due to the increased computational cost of evaluating these additional parameters. Furthermore, although these are abstract concepts commonly used by humans to assess positions, the engine may struggle to find a clear correlation between them and actual positional strength.

3.3.4. Search Multithread

TODO: mention YBWC
TODO: insert code of search multithread

3.3.4.1. Analysis

To evaluate the improvement in the new evaluation, we conducted the same 100 game match vs the basic bot version.



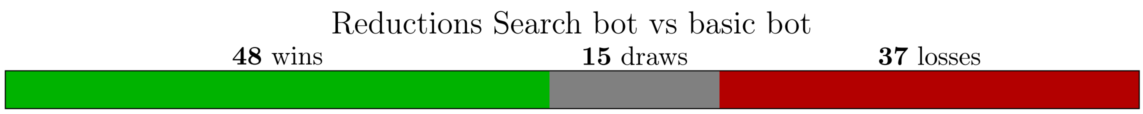
The results are slightly worse compared to the match using the material-only evaluation, with 8 more losses than before. This may be due to the increased computational cost of evaluating these additional parameters. Furthermore, although these are abstract concepts commonly used by humans to assess positions, the engine may struggle to find a clear correlation between them and actual positional strength.

3.3.5. Search reductions

TODO: mention reductions
TODO: insert code of search reductions

3.3.5.1. Analysis

To evaluate the improvement in the new evaluation, we conducted the same 100 game match vs the basic bot version.



The results are slightly worse compared to the match using the material-only evaluation, with 8 more losses than before. This may be due to the increased computational cost of evaluating these additional parameters. Furthermore, although these are abstract concepts commonly used by humans to assess positions, the engine may struggle to find a clear correlation between them and actual positional strength.

3.4. Additional tools and work

TODO: mention gui (customtkinter), cutechess, stockfish, github actions and workflows

3.4.1. Board visualizer using Python

TODO: explain

3.4.2. Testing engine strength with cutechess, stockfish, and actions

TODO: explain

Chapter 4

Conclusions and Future Work

...
The next steps to be implemented would be the application of neural networks (NNUE) which, although intended for CPUs, could be thought of as a streamlined evaluation with GPUs as performed by Leela Chess Zero.

Personal contributions

Juan Girón Herranz

Al menos dos páginas con las contribuciones del estudiante 1.
TODO

- **Move generator**
  - Explain.
  - Explain.
- **Transposition table**
  - Explain.
  - Explain.
- **Move ordering**
  - Explain.
  - Explain.
- **Evaluation**
  - Explain.
  - Explain.
- **Killer moves**
  - Explain.
  - Explain.
- **Triple repetition detection, history**
  - Explain.
  - Explain.
- **UCI Protocol Support:**

- Explain.
- Explain.

- **Testing of incremental features with Cutechess**
  - Explain.
  - Explain.
- **Unit testing**
  - Explain.
  - Explain.
- **Python GUI chess board**
  - Explain.
  - Explain.
- **Profiling**
  - Explain.
  - Explain.
- **Lichess-bot in Raspberry-pi**
  - Explain.
  - Explain.

...

Yi Wang Qiu

Al menos dos páginas con las contribuciones del estudiante 2.
TODO

- **Multithread search**
  - Explain.
  - Explain.
- **Alpha beta pruning**
  - Explain.
  - Explain.
- **Aspiration Window**
  - Explain.

    - Explain.

- **Evaluation**
    - King safety, mobility...
    - Explain.
- **Chess position bitboard representation:**
    - Explain.
    - Explain.
- **UCI Protocol Support:**
    - Explain.
    - Explain.
- **Testing of incremental features with Cutechess**
    - Explain.
    - Explain.
- **Unit testing**
    - Explain.
    - Explain.
- **Github Actions**
    - Explain.
    - Explain.
- **Python GUI chess board**
    - Explain.
    - Explain.

...

Bibliography

Fédération Internationale des Échecs. Fide laws of chess. Online, 2023. Avail. at `https://handbook.fide.com/chapter/E012023` (last access, May, 2025).

Gao, Y. and Marsland, T. A. Multithreaded pruned tree search in distributed systems. University of Alberta, 1996. Avail. at `https://webdocs.cs.ualberta.ca/~tony/RecentPapers/icci.pdf` (last access, March, 2025).

Hilewitz, Y. and Lee, R. B. Fast bit compression and expansion with parallel extract and parallel deposit instructions. Princeton University, 2006. Avail. at `http://palms.ee.princeton.edu/PALMSopen/hilewitz06FastBitCompression.pdf` (last access, May, 2025).

Jones, P. E. Generating legal chess moves efficiently. Online, 2023. Avail. at `https://peterellisjones.com/posts/generating-legal-chess-moves-efficiently` (last access, May, 2025).

Kannan, P. Magic move-bitboard generation in computer chess. Online, 2007. Avail. at `http://pradu.us/old/Nov27_2008/Buzz/research/magic/Bitboards.pdf` (last access, May, 2025).

Shannon, C. E. Programming a computer for playing chess. Computer History Museum Archive, 1950. Avail. at `https://archive.computerhistory.org/projects/chess/related_materials/text/2-0%20and%202-1.Programming_a_computer_for_playing_chess.shannon/2-0%20and%202-1.Programming_a_computer_for_playing_chess.shannon.062303002.pdf` (last access, November, 2024).

Stefan-Meyer Kahlen. Description of the universal chess interface (uci). Online, 2004. Avail. at `https://www.wbec-ridderkerk.nl/html/UCIProtocol.html` (last access, May, 2025).

Zobrist, A. L. A new hashing method with application for game playing. The University of Wisconsin, 1970. Avail. at `https://research.cs.wisc.edu/techreports/1970/TR88.pdf` (last access, May, 2025).