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

22 de mayo de 2025

Dedication

To our younger selves, for knowing the art of chess

Acknowledgments

To the Chess programming Wiki for providing an extensive collection of high-level resources about chess engines.

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. In addition to classical search algorithms, AI is applied to chess through neural networks for position evaluation and reinforcement learning.

Pioneers such as Alan Turing and Claude Shannon laid the theoretical foundations of the field, such as modeling chess as a search tree, the minimax algorithm, evaluation functions and techniques to reduce the search space. Building on these foundations, the evolution of hardware and the refinement of search techniques have enabled significant advances, such as alpha-beta pruning, an optimization of the minimax algorithm that drastically reduces the number of evaluated nodes in the game tree. Today, *Stockfish*, the most powerful open source chess engine, still relies on alpha-beta pruning, but also incorporates deep learning and neural network techniques.

The goal of this project is to develop a chess engine capable of competing against both other engines and human players, using minimax with alpha-beta pruning as its core. Additionally, we 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 been uploaded to the Lichess platform, where AlphaDeepChess achieved an ELO rating of 1900 while running on a Raspberry Pi 5 equipped with a 2GB 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. Además de los algoritmos clásicos de búsqueda, la IA se aplica al ajedrez mediante redes neuronales para evaluar posiciones y aprendizaje por refuerzo.

Pioneros como Alan Turing y Claude Shannon sentaron las bases teóricas del campo, como el modelado del ajedrez como un árbol de búsqueda, el algoritmo minimax, las funciones de evaluación y técnicas para reducir el espacio de búsqueda. Sobre estos cimientos, la evolución del hardware y el perfeccionamiento de las técnicas de búsqueda han permitido avances significativos, como la poda alfa-beta, una optimización del algoritmo minimax que reduce drásticamente el número de nodos evaluados en el árbol del juego. Hoy en día, *Stockfish*, el motor de ajedrez más potente y de código abierto, sigue basándose en la poda alfa-beta, pero también incorpora técnicas de aprendizaje profundo 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 analiza el impacto de otras técnicas clásicas, como las tablas de transposición, la búsqueda en profundidad iterativa y un generador de movimientos basado en bitboards mágicos.

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

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

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.

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.

In recent years, *Stockfish*, one of the strongest and most widely used open-source chess engines in the world, has incorporated neural network evaluation (NNUE), combining classical search with deep learning techniques. Inspired by *Stockfish*, we started this project: AlphaDeepChess, a chess engine based on minimax with alpha-beta pruning that relies solely on classical techniques and human-designed heuristics.

Beyond its technical contributions, this project also narrates the process of building a chess engine, evaluating each version along the way. This includes profiling and benchmarking different versions, as well as implementing improved heuristics and efficiency enhancements based on current knowledge about chess engines. The project even reaches the point where the engine can be played against on online platforms or observed competing with other live chess engines, making it valuable for any chess enthusiast. Notably, the engine has achieved an ELO rating of 1900 on *Lichess*, demonstrating its competitive strength among online chess engines.

The source code of our engine is available at:
`https://github.com/LauraWangQiu/AlphaDeepChess`

1.1. Objectives

The objectives of this project are the following:

- Develop a chess engine based on minimax search with alpha-beta pruning that follows the UCI protocol [11]. The engine will be a console application capable of playing chess against humans or other engines, analyzing and evaluating positions to determine the best legal move.
- Implement various known 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 was divided into several phases, each focusing on a specific aspect of the engine's development. The timeline for each phase is as follows:

1. Research phase and basic implementation: understand the fundamentals of minimax with alpha-beta pruning 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. *Benchmarking*: 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 explain 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. Chess Fundamentals

Chess is a board game where two players, taking white pieces and black pieces respectively, compete to be the first to checkmate the opponent. In chess, to *capture* means to move one of your pieces to a square occupied by an opponent's piece, thereby removing it from the board. 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.

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, the chess community agreed on creating an open communication protocol called **Universal Chess Interface** (commonly referred to as UCI), that allows interacting with chess engines through user interfaces.

A chess game takes place on a chessboard with specific rules governing the movements and interactions 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, as shown in Figure 1.1. 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, which solves the problem of chess pieces having different abbreviations depending on the language.

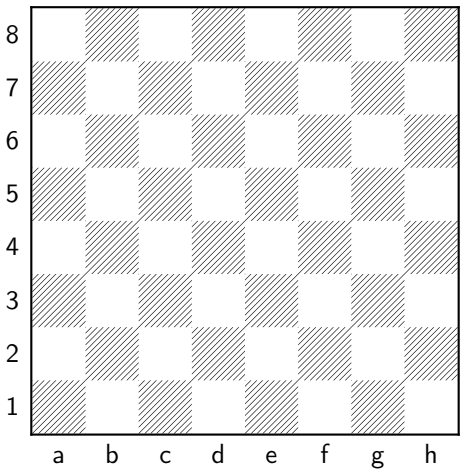
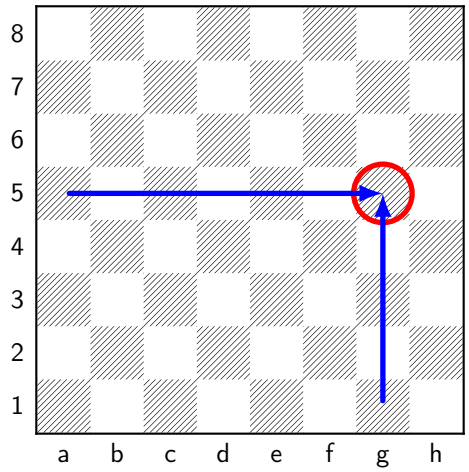


Figure 1.1: Empty chessboard.

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



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.

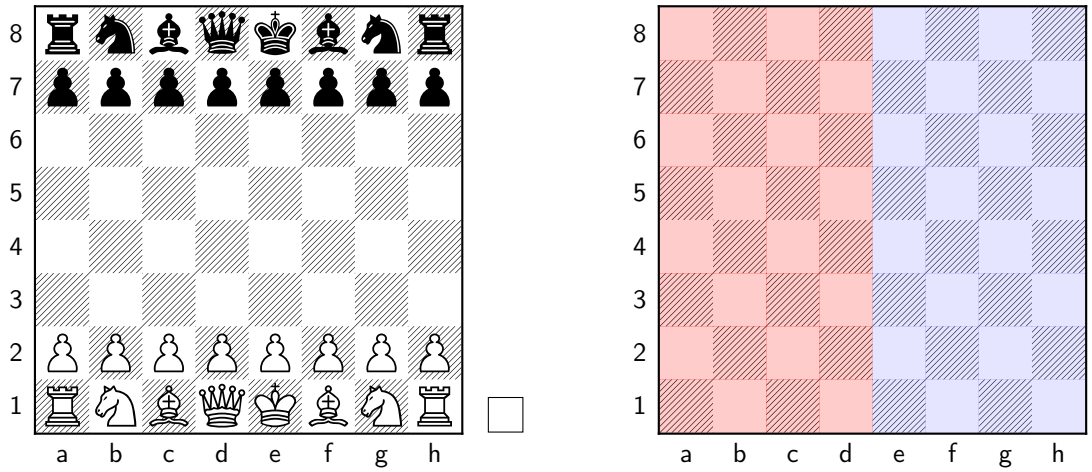


Figure 1.2: Starting position.

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, as shown in Table 1.1.

The starting position of the chess pieces on a chessboard is described in Figure 1.2. 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. 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 1.2. 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. Red is queen's side and blue is king's side.

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.

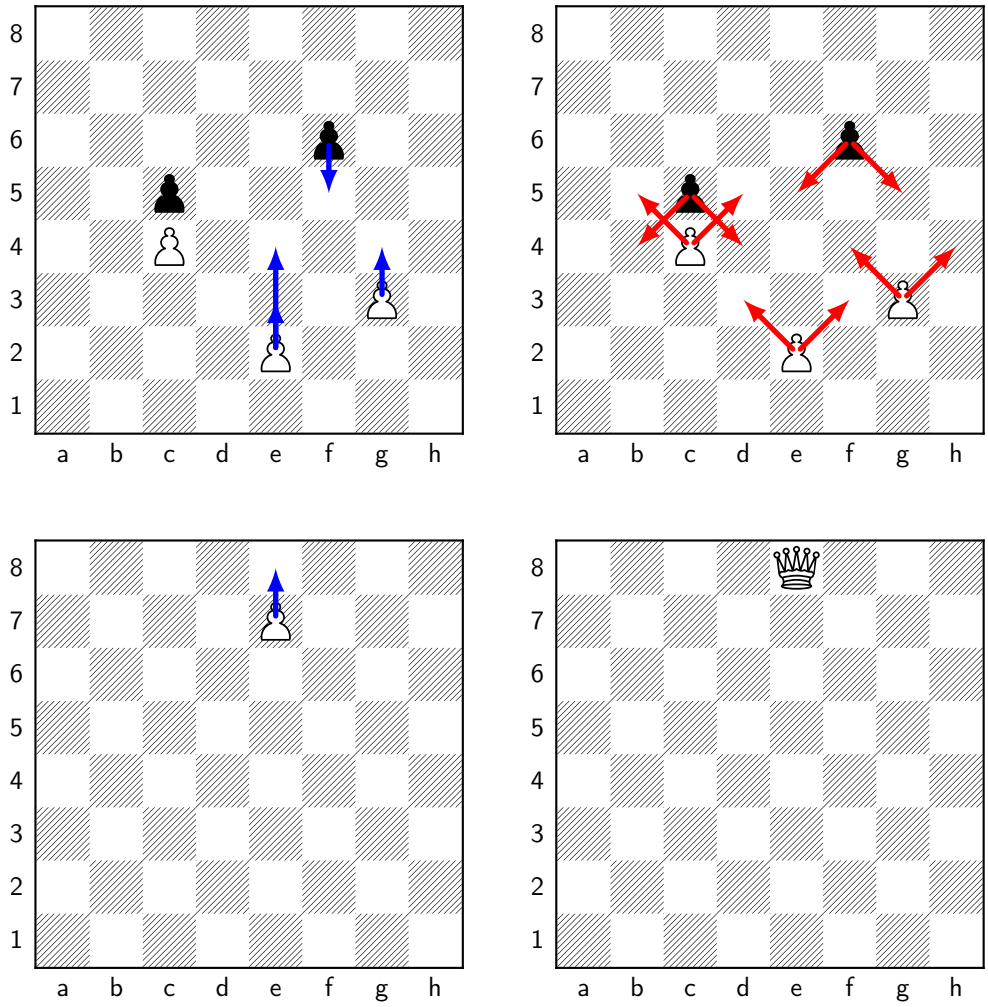


Figure 1.3: Pawn's movement, attack, and promotion.

1.3.3. Movement of the pieces

In the following sections, we describe the movement rules for each type of piece, including their unique abilities and special moves. These are fundamental to understand playing and analyzing the game.

1.3.3.1. Pawn

The pawn can move one square forward, but it can only capture pieces one square diagonally, as shown in the top right chessboard in Figure 1.3. 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. This is also illustrated in the two lower boards of Figure 1.3.

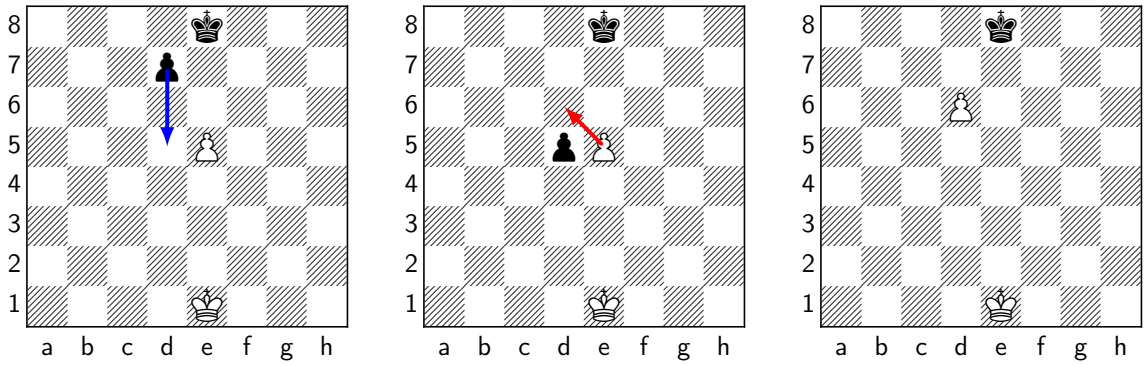


Figure 1.4: *En passant*.

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, as shown in Figure 1.4. The capturing pawn must be on an adjacent file and can only capture the *en passant* pawn immediately after it moves.

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, as shown in Figure 1.5.

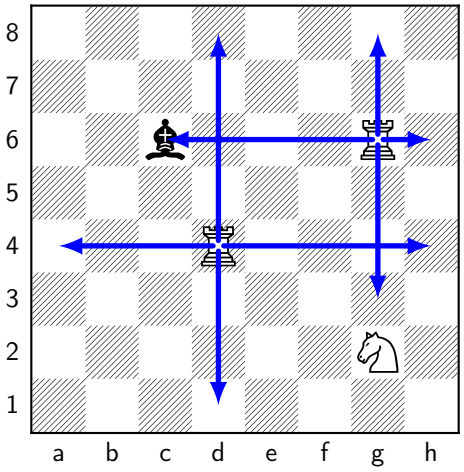


Figure 1.5: Rook's movement.

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.

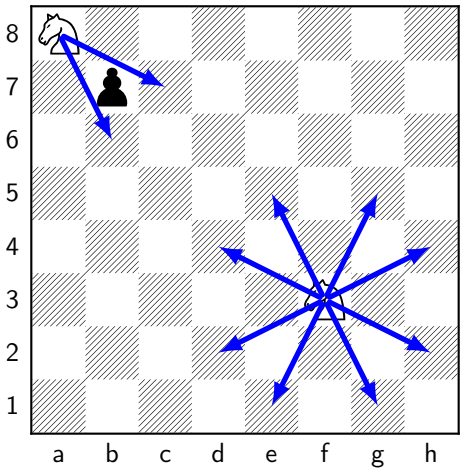


Figure 1.6: Knight's movement.

1.3.3.4. Bishop

The bishop can move any number of squares diagonally, as shown in Figure 1.7. 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.

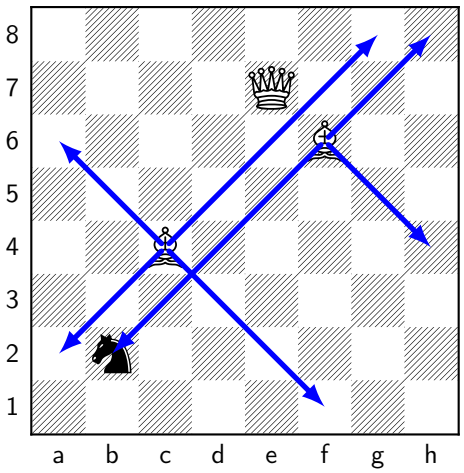


Figure 1.7: Bishop's movement.

1.3.3.5. King

The king can move one square in any direction: horizontally, vertically, or diagonally. As shown in Figure 1.8, 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.

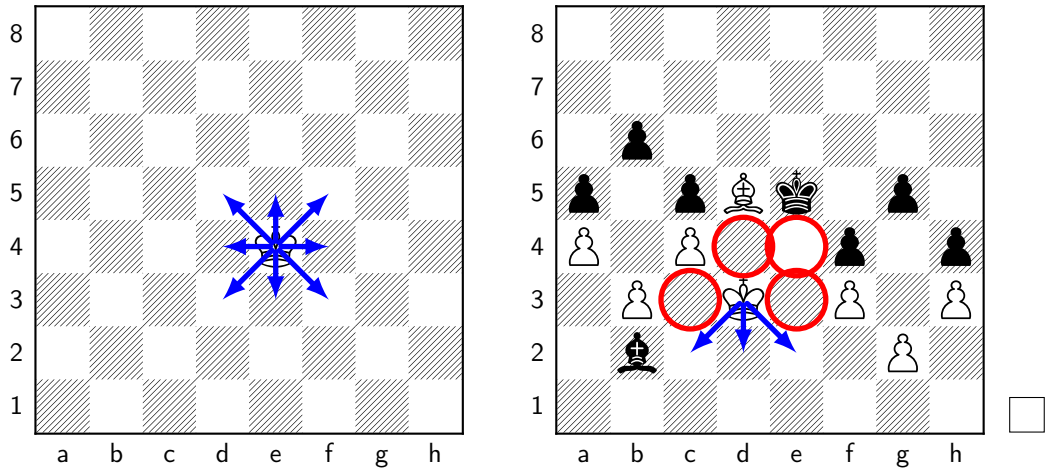


Figure 1.8: King's movement.

In Figure 1.8, the white king cannot move to $c3$, $d4$, $e3$ or $e4$ because those squares are being attacked by black pieces:

- $c3$ is attacked by the black bishop on $b2$.
- $d4$ is attacked by the black pawn on $c5$ and black king on $e5$.
- $e3$ is attacked by the black pawn on $f4$.
- $e4$ is attacked by the black king on $e5$.

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.

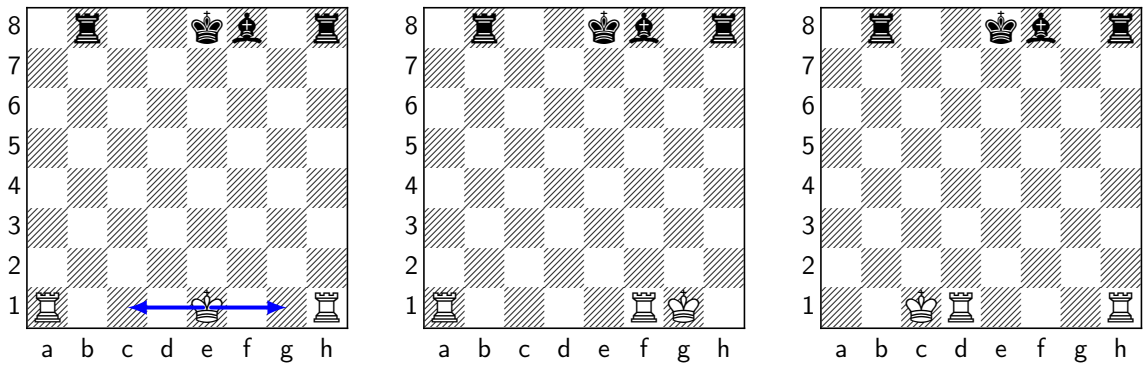


Figure 1.9: Castling situation.

In Figure 1.9, the white king can castle on either the king's side (short castling) or the queen's side (long castling) 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, as shown in Figure 1.10. It can also capture pieces in the same way.

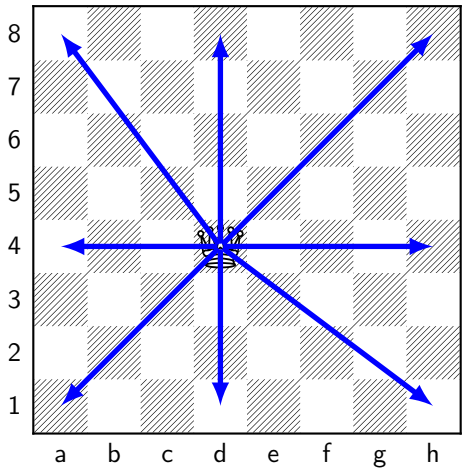


Figure 1.10: Queen's movement.

1.3.4. Rules

The rules of chess follow the official regulations established by FIDE [3]. As we have already said, 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*).

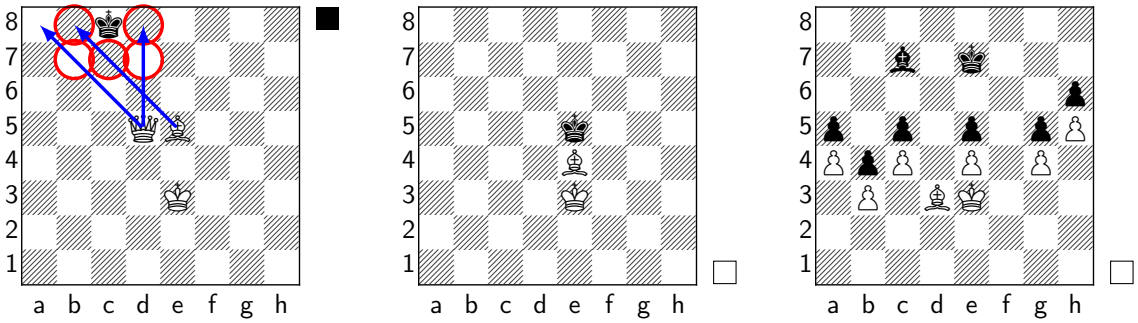


Figure 1.11: Stalemate, insufficient material, and dead position.

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 in which neither player can achieve checkmate by any series of legal moves, making further progress impossible. This includes the case of insufficient material, where it is not possible to checkmate the opponent regardless of the moves played. In these situations, the game is immediately declared a draw because the main objective to checkmate cannot be accomplished.

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, as shown in Table 1.2.

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, as shown in Figure 1.12.

In Figure 1.12, the pawn's movement is written as $Pa6$ or directly as $a6$.

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.

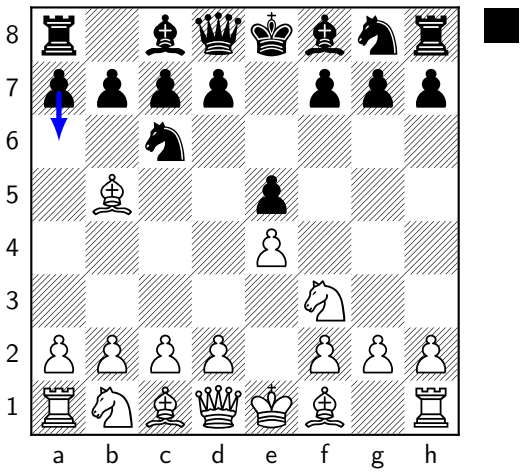


Figure 1.12: Pawn goes to a6.

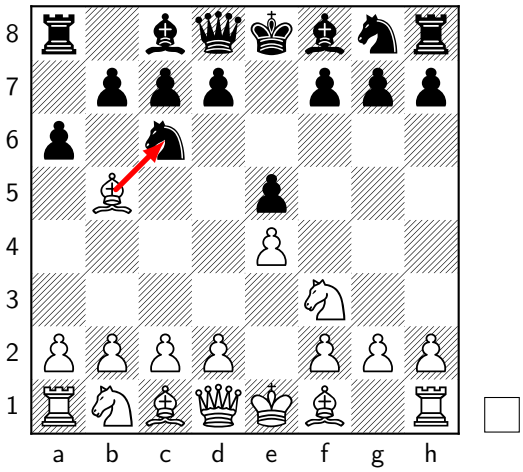


Figure 1.13: Bishop captures knight.

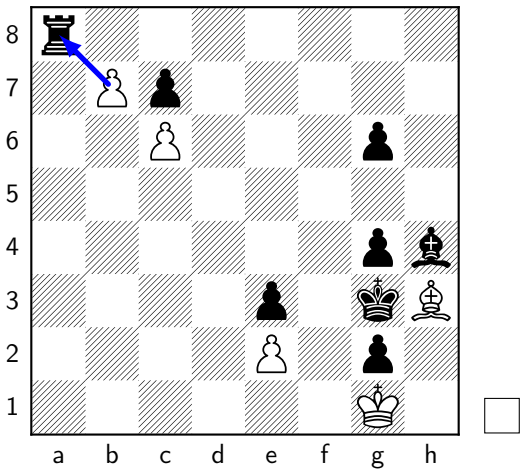


Figure 1.14: Pawn captures rook.

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.

In Figure 1.13, 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. In Figure 1.14, white pawn capturing and promoting in $a8$ to a queen is written as $bxa8Q$ or $bxa8 = Q$.

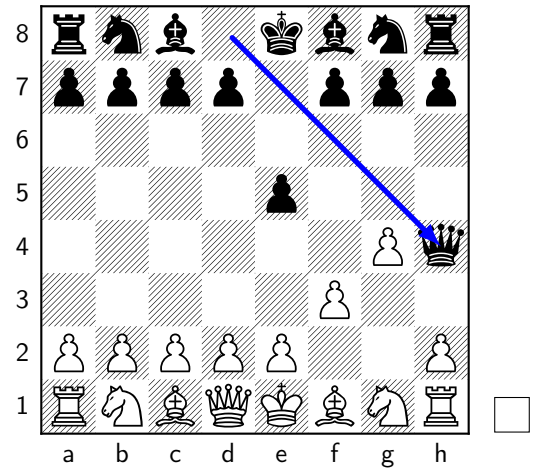


Figure 1.15: Black queen checkmates.

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. In Figure 1.15, black queen movement checkmates and it is written as $Qh4++$.

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 notation 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
```

The fields of the FEN for the starting position are:

- `rnbqkb1r/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR`: Piece placement on the board, from rank 8 to rank 1. Uppercase letters represent White pieces, lowercase letters represent Black pieces. Numbers indicate consecutive empty squares.

- `w`: Indicates which side is to move (`w` for White, `b` for Black).
- `KQkq`: Castling availability. `K` (White kingside), `Q` (White queenside), `k` (Black kingside), `q` (Black queenside). If no castling is available, a hyphen (`-`) is used.
- `-`: En passant target square. If there is no en passant possibility, a hyphen is used.
- `0`: Halfmove clock, counting the number of halfmoves since the last pawn move or capture (for the fifty-move rule).
- `1`: Fullmove number, incremented after each Black move.

Keep in mind this notation is very important for chess engines, including Alpha-DeepChess, because it provides all the necessary information to reconstruct any chess position, including all the mentioned variables above on the current board without needing to replay all the moves from the beginning of the game. This makes it highly efficient for testing, analysis, and interoperability with other chess software.

1.3.5.3. Portable Game Notation (PGN)

This notation is a widely used text-based format for recording chess games. Its primary purpose is to store complete game records, making it ideal for building large databases of chess games for study, analysis, and sharing. These PGN files not only include the sequence of moves, but also a header section with metadata such as the name of the event, site, date of play, color and name of each player, result and possible comments and variations. For example, PGN for a game could look like this:

```
[Event "Rated␣blitz␣game"]
[Site "https://lichess.org/8USBsBgk"]
[Date "2025.05.19"]
[White "Zimbabwean_chessbot"]
[Black "AlphaDeepChess"]
[Result "0-1"]
[GameId "8USBsBgk"]
[UTCDate "2025.05.19"]
[UTCTime "05:41:30"]
[WhiteElo "2865"]
[BlackElo "1904"]
[WhiteRatingDiff "-32"]
[BlackRatingDiff "+12"]
[WhiteTitle "BOT"]
[BlackTitle "BOT"]
[Variant "Standard"]
[TimeControl "180+2"]
[ECO "B12"]
[Opening "Caro-Kann␣Defense:␣Advance␣Variation,␣Short␣
   Variation"]
[Termination "Time␣forfeit"]
```

```
1. e4 c6 2. d4 d5 3. e5 Bf5 4. Nf3 e6 5. Be2 Nd7 6. O-O h6 7.
   a4 Ne7 8. c3 a6 9. Nbd2 g5 10. b4 Bg7 11. Nb3 O-O 12. Ra2
   f6 13. exf6 Bxf6 14. h4 g4 15. Ne1 h5 16. g3 Qc7 17. Ng2 e5
    18. f3 gxf3 19. Bxf3 exd4 20. Bf4 Ne5 21. Nxd4 Nxf3+ 22.
   Qxf3 Qb6 23. Qxh5 Bxd4+ 24. cxd4 Qxd4+ 25. Raf2 Bg6 26. Qg5
    Qxb4 27. Be5 Rxf2 28. Rxf2 Rf8 29. Qh6 Rf6 30. Rxf6 Qb6+
   0-1
```

Listing 1.1: Example of a PGN file

These files are generated both during the testing phases of the chess engine and when playing games on online chess platforms.

Chapter 2

State of the Art

In this chapter, we will explore the fundamental concepts and classical techniques on which our chess engine is based. This includes game trees and search algorithms. Each section will provide an overview of the concepts and tools used by our engine, and additionally we will explain the workflow we followed to get better versions of our engine.

2.1. 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 corresponds to the initial position from which we search for the best move, and each subsequent node represents a possible option or game state, forming a tree-like structure. The tree has a height or depth, which refers to the number of turns alternating between black and white from the root node to the leaf nodes.

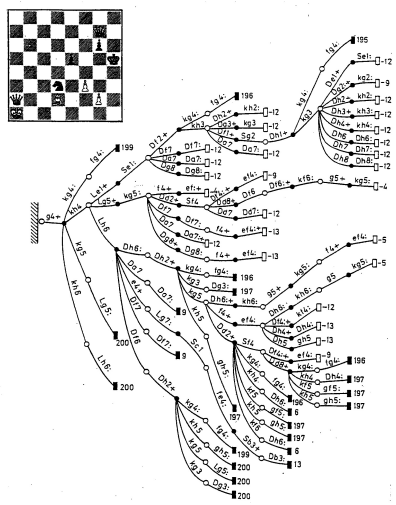


Figure 2.1: Illustration of a game tree for a chess position [1].

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.

In Figure 2.1, the shaded cluster of lines on the left represents the part of the game tree corresponding to previous moves leading up to the current position. From there, only a subset of possible continuations is drawn, illustrating the immense complexity of chess. Claude Shannon estimated that the number of possible chess games is around 10^{120}, a value known as the *Shannon number* [13]. This astronomical figure makes it computationally infeasible to represent or analyze the entire game tree, motivating the use of search algorithms and pruning techniques to explore only the most relevant branches, which are explained in the following section.

2.2. Search algorithms

There are different approaches to analyze and 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. That's why 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 as mentioned in the last section. This is 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 to reduce the number of nodes evaluated, making the search more efficient while still exploring the tree deeply. Listing 2.1 illustrates the working of the DFS algorithm.

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

```
1 Procedure DFS(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.

In practice, especially in chess, a bounded or depth-limited version of DFS is often used. In bounded DFS, the search is restricted to a maximum depth, preventing the algorithm from exploring the entire tree. This approach allows the algorithm to focus on the most relevant positions.

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.

```
1  Procedure BestFirstSearch(Graph G, Node start, Node goal):
2      Create an empty priority queue PQ
3      Add start to PQ with priority 0
4      Mark start as visited
5
6      While PQ is not empty:
7          Node current = PQ.pop()
8          If current is the goal:
9              Return the path to the goal
10
11         For each neighbor w of current in G.adjacentEdges(
               current):
12             If w is not visited:
13                 Calculate priority for w (e.g., using a
                       heuristic)
14                 Add w to PQ with the calculated priority
15                 Mark w as visited
```

In this case, the priority queue contains nodes along with their associated priorities, which are determined by a heuristic function. This process is commonly referred to as branch and bound, where the algorithm explores branches of the search tree according to their heuristic value.

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

Next, we will discuss one of the most widely used search algorithms in chess engines: the minimax algorithm.

2.2.1. Minimax algorithm

The **minimax** algorithm is a decision making algorithm that follows Depth-First Search (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 has the pseudocode of the algorithm:

Listing 2.3: Pseudocode of the Minimax algorithm.

```
1  Procedure Minimax(Node position, Integer depth, Boolean
       maximizingPlayer):
2      If depth == 0 or position is a terminal node:
3          Return the evaluation of the position
4
5      If maximizingPlayer:
6          Integer maxEval = -Infinity
7          For each child of position:
8              Integer eval = Minimax(child, depth - 1, False)
9              maxEval = max(maxEval, eval)
10         Return maxEval
11     Else: // minimizingPlayer
12         Integer minEval = +Infinity
13         For each child of position:
14             Integer eval = Minimax(child, depth - 1, True)
15             minEval = min(minEval, eval)
16         Return minEval
```

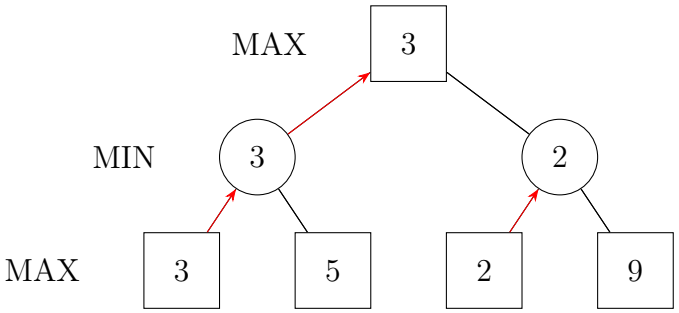


Figure 2.2: Example of minimax.

For example, let's say that white always wants the highest value (maximizing the result), while black wants the lowest value (minimizing the result). In Figure 2.2, white is represented by square nodes and black by circle nodes. Note that this example is a binary tree, but in a real scenario there would likely be more moves or nodes, because in chess each position usually allows a wide range of legal moves, not just two as in a binary tree. 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, for 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 value between 3 and 2.

The effectiveness of search algorithms such as minimax, and their optimizations (which are carefully explained in Chapter 3), is crucial for the playing strength of a chess engine. The deeper and more efficiently the engine can search the game tree, the better its move selection and overall performance.

Achieving this level of efficiency and quality requires a well-structured development process. For this reason, we adopted a systematic methodology to guide the implementation and continuous improvement of our engine.

2.3. Methodology

Once the basic foundations are established with an initial version of the essential components or modules (which will be described later), our workflow follows an iterative process: first, we search for existing information on each topic, analyze it, implement a solution, and then profile the implementation to identify bottlenecks. After locating performance issues, we optimize the relevant parts, and finally, compare the new version with the previous one to assess improvements.

Then, at a given moment, we can decide to take action and try to determine the strength of the engine with the last functional version.

2.3.1. Profiler

First, in order to analyze the performance of our chess engine and identify potential bottlenecks, we used the `perf` tool available on Linux systems. `perf` provides robust profiling capabilities by recording CPU events, sampling function execution, and collecting stack traces.

Our profiling goal is to identify which parts of the code consume the most execution time. We run the engine under `perf` using the following commands:

Listing 2.4: Profiling AlphaDeepChess with perf

```
# Record performance data with function stack traces
sudo perf record -g ./build/release/AlphaDeepChess

# Display interactive report
sudo perf report -g --no-children
```

After recording, `perf report` opens an interactive terminal interface where functions are sorted by CPU overhead. This allows us to prioritize which functions to optimize.

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

Once the profiling and optimization steps have been completed with the last decided version, we are ready to try to determine the strength or level of our engine.

The most common way to measure the strength of a chess engine is by playing games against other engines and analyzing the results. To quantify this strength,

the Elo rating system is used. Elo is a statistical rating system originally developed for chess, which assigns a numerical value to each player (or engine) based on their game results against opponents of known strength. When an engine wins games against higher-rated opponents, its Elo increases; if it loses, its Elo decreases. This allows for an objective comparison of playing strength between different engines.

But which engine should we use as a reference for comparison? One approach is to consult the Computer Chess Rating Lists, which rank chess engines based on their performance in various tournaments and matches. We have chosen to compare different versions of our engine with *Stockfish*, currently ranked as the number one engine on the list. In addition, we have also competed against other engines online to further evaluate our engine's performance in diverse environments.

2.4.1. Stockfish

Stockfish [5] is an open-source chess engine and command-line program available for multiple platforms, including Windows, macOS, Linux, Android, and iOS. It provides a wide range of versions optimized for different hardware configurations, such as specific CPU instruction sets, to maximize performance. For instance, the AVX2 version is recommended for most users with Intel processors from 2013 onwards or AMD processors from 2015 onwards, as it utilizes advanced vectorization instructions.

Although it is not strictly mandatory for chess engines to follow a communication standard, adopting one greatly facilitates interoperability and development. In this case, *Stockfish* uses the Universal Chess Interface communication protocol, which we have also implemented in our engine.

2.4.2. UCI

Universal Chess Interface, also known as UCI, is an open communication protocol whose specifications are independent of the operating system and which enables chess engines to communicate with graphical user interfaces and with each other.

The move format is in long algebraic notation which means sending two squares coordinates like `e2e4` or `b1c3` independently of the type of piece because the engine must be the one checking that the movement is legal.

Some of the most important and used commands are the following:

- `position [fen <fenstring>| startpos | actualpos] moves <move1>... <movei>`: Sets the current position of the board to the FEN string, or applies the list of moves from the starting position or current position.
- `go`: Starts evaluating the current position. Some important subparameters are:
  - `depth <x>`: Specifies the number of `x` plies to search.
  - `movetime <x>`: Specifies the number of `x` seconds to search.

- `stop`: Stops evaluation if it is running.
- `diagram`: Shows a basic terminal chessboard with the current position.

A ply consists of a single move made by one player in a turn-based game like chess. In chess terminology, one ply refers to one move by either White or Black, not both. Therefore, a full turn (when both White and Black have moved) consists of two plies.

These commands are considered the most important because they form the core of the interaction between the chess engine and the user interface or testing tools. The `position` command allows setting up any board state, which is essential for starting games, analyzing specific positions, or resuming play. The `go` command initiates the engine's search for the best move, with parameters like `depth` and `movetime` providing control over the search process. The `stop` command is necessary to halt the engine's calculation when required, ensuring responsiveness. Finally, the `diagram` command provides a quick visual representation of the current position, which is especially useful for debugging and analysis. Together, these commands enable flexible, efficient, and interactive communication with the chess engine.

Then, to conduct fair and automated matches between Stockfish and our engine, it is necessary to use a tool that can arbitrate the games, ensuring consistent conditions and accurate results.

2.4.3. Cutechess

After considering different options, *Cutechess* proved to be the best fit for our needs.

Cutechess [4] is an open-source tool designed to perform automated games between chess engines. It is widely used in the chess programming community to test and compare engines, evaluate their performance, and analyze games.

It provides both command-line interface (CLI) and a graphic user interface (GUI), with cross-platform compatibility for Windows, macOS, and Linux. For our purposes, we utilized the CLI version to automate the tests with Python scripts and commands, integrating it into a CI/CD workflow.

Mainly, this tool is responsible for sending commands to both selected engines. For example, when both Stockfish and our engine implement UCI, *Cutechess* knows in advance the commands to send, as well as parameters such as the search time and depth for each engine, the number of games to play, the time control, or even specific openings to use. Providing specific openings introduces randomization between games, which is beneficial for later evaluation.

Then, it checks the active status of each engine and manages the start of the games by setting up the board on both engines with the `position` command. Afterwards, it sends the `go` command and stops the search with `stop` when the search time or depth is reached, extracting the best move provided by the engine whose turn it is.

This move is then passed to the other engine, alternating turns so that both engines play against each other automatically. Listing 2.5 we show a *Cutechess* log file, we directly configured the starting positions from a list of FENs in the `positions.fen` file:

Listing 2.5: Example of *Cutechess*

```
Running test (8) with the following configuration:
Games: 1, Search Time: 5, Depth: 5
PGN File: results.pgn, EPD File: results.epd, Log File: results.log
Engines: ['AlphaDeepChess', 'Stockfish']
Options: {'Stockfish': {'UCI_LimitStrength': 'true', 'UCI_Elo': '2000'}}
Book:
Positions: positions.fen
...
 <Stockfish(1): uciok
 >Stockfish(1): setoption name UCI_Elo value 2000
 >Stockfish(1): setoption name UCI_LimitStrength value true
 >Stockfish(1): isready
 <Stockfish(1): readyok
Started game 1 of 1 (AlphaDeepChess vs Stockfish)
 >AlphaDeepChess(0): ucinewgame
 >AlphaDeepChess(0): position fen rnbqkb1r/1p2pppp/p2p1n2/8/3NP3/2N5/PPP2PPP/
     R1BQKB1R w KQkq - 0 6
 >Stockfish(1): ucinewgame
 >Stockfish(1): setoption name Ponder value false
 >Stockfish(1): position fen rnbqkb1r/1p2pppp/p2p1n2/8/3NP3/2N5/PPP2PPP/R1BQKB1R w
     KQkq - 0 6
 >AlphaDeepChess(0): isready
 <AlphaDeepChess(0): readyok
 >AlphaDeepChess(0): go movetime 5000 depth 5
 <AlphaDeepChess(0): info depth 1 score cp 130 bestMove c1e3
 <AlphaDeepChess(0): info depth 2 score cp 85 bestMove c1e3
 <AlphaDeepChess(0): info depth 3 score cp 80 bestMove c1e3
 <AlphaDeepChess(0): info depth 4 score cp 62 bestMove c1g5
 <AlphaDeepChess(0): info depth 5 score cp 79 bestMove c1g5
 <AlphaDeepChess(0): bestmove c1g5
 >Stockfish(1): position fen rnbqkb1r/1p2pppp/p2p1n2/8/3NP3/2N5/PPP2PPP/R1BQKB1R w
     KQkq - 0 6 moves c1g5
 >Stockfish(1): isready
 <Stockfish(1): readyok
 >Stockfish(1): go movetime 5000 depth 5
 <Stockfish(1): info depth 1 seldepth 6 multipv 1 score cp -52 ...
 <Stockfish(1): info depth 2 seldepth 3 multipv 1 score cp -45 ...
 <Stockfish(1): info depth 3 seldepth 3 multipv 1 score cp -39 ...
 <Stockfish(1): info depth 4 seldepth 3 multipv 1 score cp -39 ...
 <Stockfish(1): info depth 5 seldepth 3 multipv 1 score cp -27 ...
 <Stockfish(1): bestmove e7e6
 >AlphaDeepChess(0): position fen rnbqkb1r/1p2pppp/p2p1n2/8/3NP3/2N5/PPP2PPP/
     R1BQKB1R w KQkq - 0 6 moves c1g5 e7e6
```

In Listing 2.5, in addition to setting the number of games to 1, the search time to 5 seconds, and the search depth to 5, we provided a list of starting positions, from which one is chosen at random. Each line starting with `position fen ...` establishes the FEN position and continues by adding the moves played.

Note that both engines stop searching when depth of 5 is reached and are verified to be ready after `isready` command.

Depending on the number of games, search time, and time control, these tournaments between engines can take a long time to process. For this reason, we decided to use GitHub Actions and workflows to separate our development environment from the execution of performance and strength tests. This is explained later in **??**.

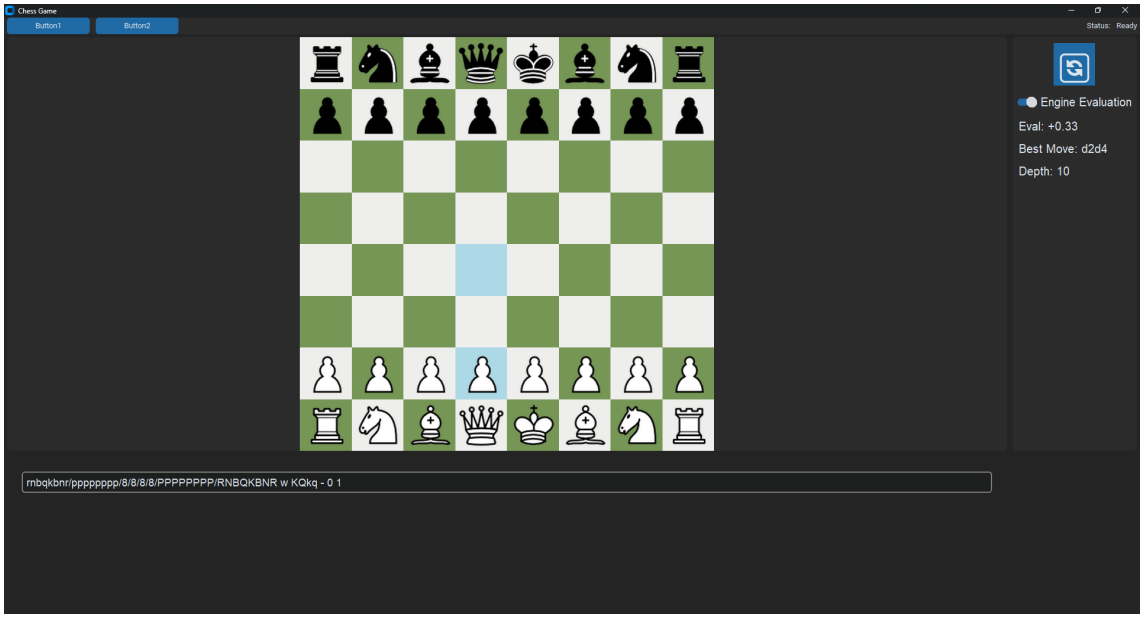


Figure 2.3: AlphaDeepChess GUI

Although UCI implements the `diagram` command that draws the current position through standard output, making moves and showing the evaluation is somewhat a time-consuming task when debugging and testing while programming. We resorted to using an interface to help us do this job and a really fast solution was to use Python to make a GUI. In this case, one of the most used UI libraries was *CustomTkinter* and it was used to build a friendly interface from scratch for bridging between executable and command sending. This tool can be used after compiling the engine and executing `AlphaDeepChessGUI.py` with Python.
In Figure 2.3, the engine evaluation is enabled and displays the current evaluation value, the best move found, and the calculated search depth. In this way, we also ensure a more user-friendly experience.

In the following chapters, we will present more extensive and detailed information on building the engine.

Chapter 3

Engine Architecture

This chapter documents the development process of the chess engine. The project is organized into the following modules:

- *Board*: Data structures to represent the chess board.
- *Evaluation*: Assign a score to a chess position
- *Move_ generator*: Create a list of all the legal moves in a position.
- *Move_ ordering*: Sorts moves by estimated quality.
- *Search*: Contains the algorithm to search the best move.
- *UCI*: Universal Chess Interface implementation.

First, we describe the implementation of the basic parts of the chess engine, then we introduce and explain in detail the algorithmic techniques developed to improve the engine's performance.

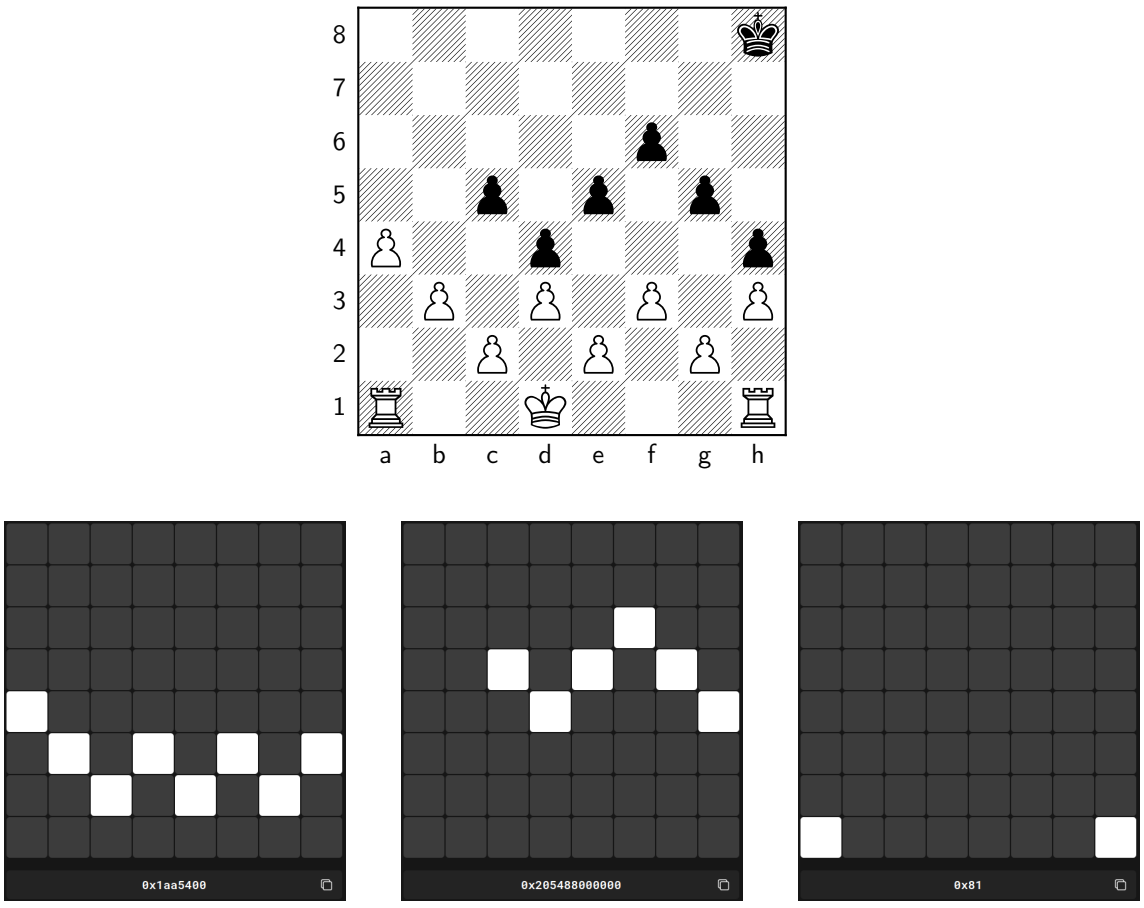
We begin by examining the fundamental data structure used for chess position representation.

3.1. Chessboard Representation: Bitboards

The chessboard is represented using a list of *bitboards*. A bitboard is a 64-bit variable in which each bit corresponds to a square on the board. A bit is set to `1` if a piece occupies the corresponding square and `0` otherwise. The least significant bit (LSB) represents the `a1` square, while the most significant bit (MSB) corresponds to `h8` [18].

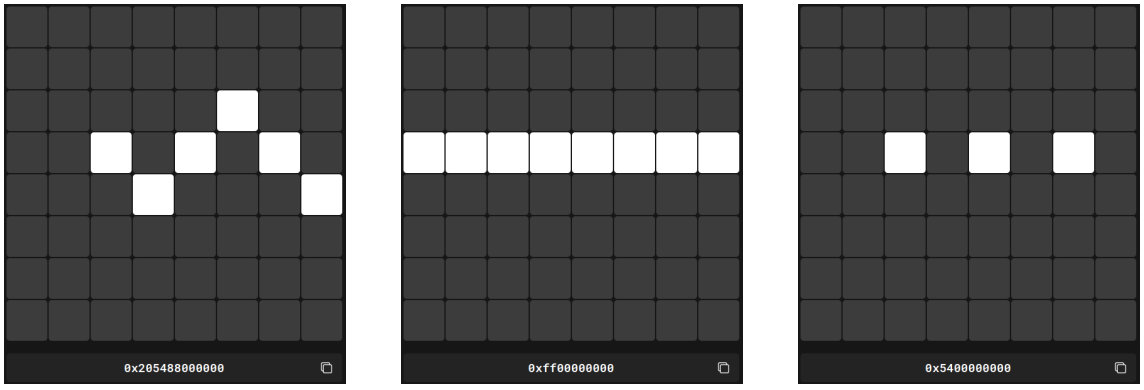
The complete implementation can be found in the `Board` class file in `include\board\board.hpp` and `src\board\board.cpp`.

A list of twelve bitboards is used, one for each type of chess piece. Figure 3.1 illustrates this concept with a chess position example.
The main advantages of bitboards is that we can operate on multiple squares simultaneously using bitwise operations. For example, we can determine if there are



Bitboard of white pawns. Bitboard of black pawns. Bitboard of white rooks.

Figure 3.1: List of bitboards data structure example.



Bitboard of black pawns | Fifth rank mask | Pawn's bitboard & mask

Figure 3.2: Bitboard mask operation example.

any black pawns on the fifth rank by performing a bitwise AND operation with the corresponding mask. Figure 3.2 illustrates this concept.

3.1.1. Game State

In addition, we need to store the game state information. We designed a compact 64-bit structure to encapsulate all relevant data, enabling efficient copying of the complete game state through a single memory operation. The structure contains the following key fields:

1. Total number of moves played in the game (bit 0-19).
2. En passant target square (if applicable) (bits 20-26).
3. Black's queenside castling availability (bit 27).
4. Black's kingside castling availability (bit 28).
5. White's queenside castling availability (bit 29).
6. White's kingside castling availability (bit 30).
7. Current side to move (white or black) (bit 31).
8. Fifty-move rule counter (moves since a capture or pawn move) (bits 35-42).

Having described the data structures that represent chess positions, we can now present the engine's core component: the search algorithm.

3.2. Search Algorithm

The search algorithm implemented is an enhanced minimax with alpha-beta pruning (see Section 2.2.1) where white acts as the maximizing player and black as the

minimizing player. The entire game tree is generated up to a selected maximum depth. At each node, the active player evaluates the position, while the alpha and beta values are dynamically updated during execution. 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 (for MAX) or beta (for MIN).

The complete implementation is available in `src\search\search_basic.cpp`.

The following events happen at each node of the tree:

1. Terminal node verification: Check for game termination conditions including checkmate, threefold repetition, the fifty-move rule, or reaching maximum search depth.
2. Position evaluation: A positive value indicates White's advantage, while a negative value favors Black. We establish 3,200,000 as the mate-in-one threshold value.
3. Legal move generation: create a list of every possible legal move in the position.
4. Move ordering: Sort moves by estimated quality (best to worst). The sooner we explore the best move, the more branches of the tree will be pruned.
5. Move exploration: Iterate through each of the legal moves from the position in order, update the position evaluation, the value of alpha and beta, and performing pruning when possible.

Figure 3.3 demonstrates the alpha-beta search process by using *alpha* (α) and *beta* (β) values.

First, we obtain the leftmost and deepest number, which is 3, and β is set to this number on the node above, representing the current lowest bound. Then, 3 is compared with the following node on the same depth, 5, and the MIN node selects the minimum value between 3 and 5, which is 3.

Next, this β value is passed up to the MAX node above, updating its α value to 3. This means that the upper bound for that node must be a number less than or equal to 3.

Then, the last calculated α value is passed to the MIN node in the right subtree. To determine its value, we continue down to the next depth, where the first node has a value of 2. This value updates the β value of the MIN node above, with the passed α value. Now, this MIN node can only take a value between 3 and 2, meaning the node with a value of 1 cannot be a solution and is pruned.

Throughout this process, branches that cannot possibly influence the final decision are pruned, significantly reducing the number of nodes that need to be evaluated. This demonstrates the efficiency of alpha-beta pruning in minimizing the search space while guaranteeing the same result as a full minimax search.

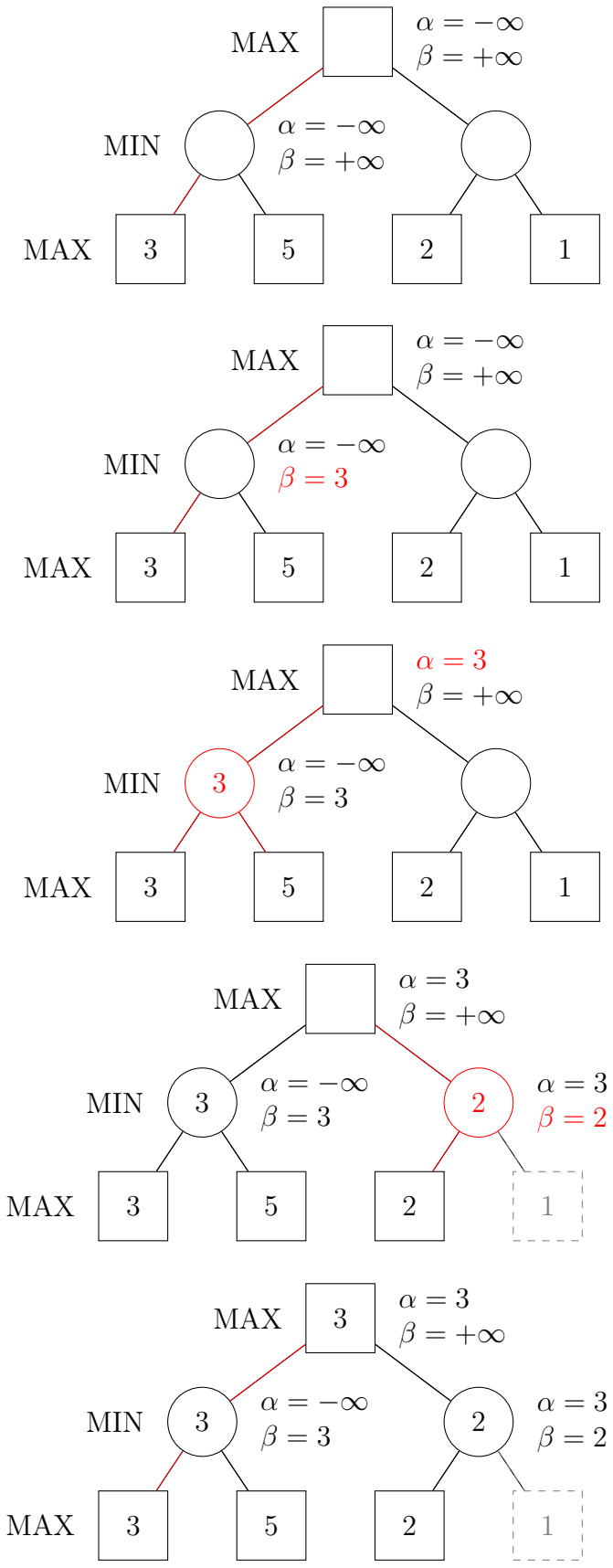


Figure 3.3: Example of alpha-beta pruning with α and β values.

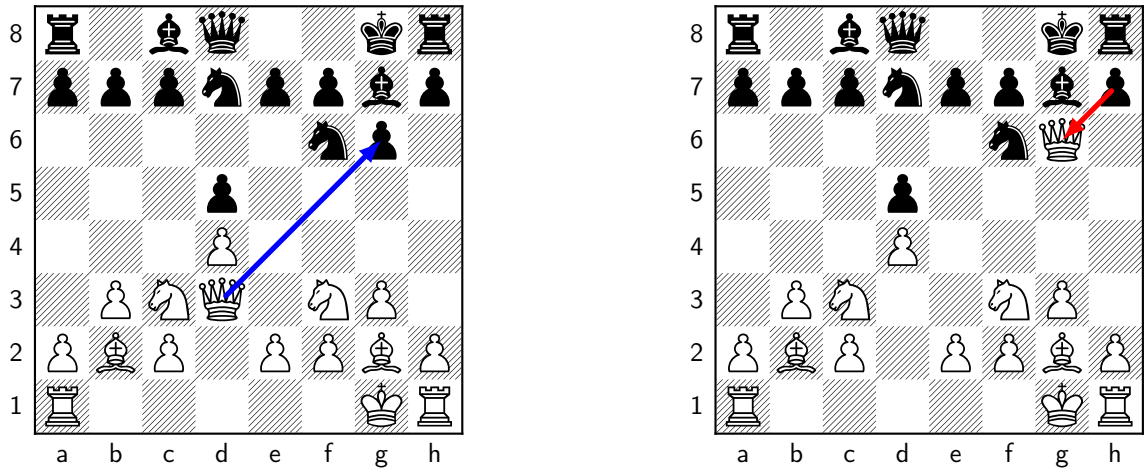


Figure 3.4: Horizon effect position example.

Therefore, one might think to what extent or depth should the search be carried out. This is where iterative deepening comes into play.

3.2.1. Iterative deepening

What is the optimal depth at which to stop the search? In practice, the most straightforward approach is to perform an iterative deepening search, first searching at depth 1, then 2, then 3... to infinity [16]. The engine will update the evaluation and the best move for the position in each iteration and the search can be halted at any point by issuing the *stop* command. In our implementation, this is handled using two threads: one dedicated to reading input from the command line, and the other performing the search. When the `stop` command is received, the input thread sets an atomic stop flag, which the search thread checks to terminate its execution.

It is important to note that, in each iteration, all computations from the previous depth are repeated from scratch. This approach is inherently inefficient so in subsequent sections, we will introduce techniques to tackle this inefficiency.

3.2.2. Horizon effect problem, quiescence search

What happens if, upon reaching maximum depth, we evaluate the position in the middle of a piece exchange? For example, the Figure 3.4 illustrates a position where if the search is stopped when the queen captures the 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 [7].

To avoid this, when we reach the end of the tree at maximum depth, we must extend the search but only considering capture moves until no captures are available. This is known as quiescence search [21].

The purpose of this technique is to stop the search only in quiet positions, where there is no capture or tactical movement. Efficiency in this search extension is paramount, as some positions may lead to long sequences of capture moves. To prevent excessive depth, we select a limit of 64 plies, in order to stop the search when we explore to that threshold depth.

The following events occur in a quiescence node:

1. *Terminal node verification*: Check for game termination conditions due to checkmate, threefold repetition, the fifty-move rule or reaching a maximum ply.

2. *Standing pat evaluation*: Also known as static evaluation, this step assigns a preliminary score to the position. This score can serve as a lower bound and is immediately used to determine whether alpha-beta pruning can be applied.

3. *Selective legal move generation*: Create a list of every possible legal move excluding moves that are not captures.

4. *Move ordering*: Sort capture moves by estimated quality (best to worst).

5. *Move exploration*: Iterate through each of the capture legal moves from the position in order, update the position evaluation, the value of alpha and beta, and check if we can perform pruning.

3.2.3. Aspiration Window

Continuing with another improvement, aspiration window's main objetive is to reduce the number of nodes to explore by simply restricting the range of *alpha* and *beta* values (window). A search is performed with this narrow window. If the position evaluation falls within the window, it is accepted as valid and additional node exploration is avoided. However, if the evaluation is outside the limits of the window (when a fail-low or fail-high occurs), the window is expanded to the extreme values ($-\infty$ and $+\infty$) and a new search is performed to obtain an accurate evaluation [6].

3.3. Evaluation: Materialistic approach

In this section we present how our evaluation function works. For each position, a numerical value is assigned representing how favorable the position is for one side: positive (+) for white and negative (−) for black. The values are typically expressed in centipawns (cp), where one centipawn equals one hundredth of a pawn [13].

The full implementation can be found in `src\evaluation\evaluation_dynamic.cpp`.

Table 3.1 shows the standard centipawn values assigned to each piece type

Piece	Value (cp)
Pawn	100
Knight	320
Bishop	330
Rook	500
Queen	950
King	500

Table 3.1: Standard values assigned to chess pieces in centipawns.

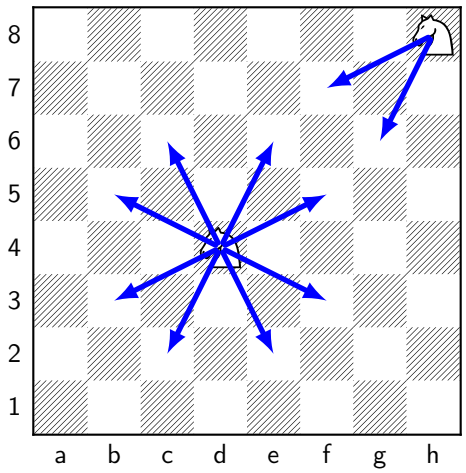


Figure 3.5: Knight's movement on corner vs in center.

The evaluation of a position can be computed by summing the values of all white pieces on the board and subtracting the values of all black pieces, as shown in the next equation:

$$\text{Evaluation(position)} = \sum_{w \in \text{WhitePieces}} V(w) - \sum_{b \in \text{BlackPieces}} V(b)$$

Where $V(x)$ denotes the value of piece x.

3.3.1. Piece Square Tables (PST's)

The basic material evaluation described earlier has a significant limitation, it does not consider the fact that a piece could have more power in different squares of the board. For instance, as illustrated in Figure 3.5, a knight placed in the center can control up to eight squares, while a knight positioned in the corner can reach only two.

The solution is to add a bonus or a penalization to the piece depending on the square it occupies. This is called a piece square table (PST). For each piece type, a PST assigns a positional bonus based on the square it occupies. These tables are typically implemented as arrays indexed by square and piece type [20].

-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 3.6: Piece Square Table for the bishop.

An example PST for the bishop is shown in Figure 3.6, where the bishop receives a positional bonus for occupying central squares and a penalty for being placed in the edges of the board.

3.3.2. Tapered evaluation

A chess game typically consists of three phases: the opening, where pieces are developed to more effective squares; the middlegame, where tactical and strategic battles take place; and the endgame, where usually the pawns aim to promote and the side that has the advantage tries to corner the enemy king to mate it.

It is clearly suboptimal to assign the same piece-square table (PST) bonuses to pieces like the pawn or king during both the middlegame and the endgame. To address this, we implement *tapered evaluation*, a technique that computes two separate evaluations, one for the middlegame/opening and another for the endgame, then interpolates between the two scores to produce a final evaluation [17].

First we calculate the percentage of middlegame and the percentage of endgame:

1. 100 % Middlegame: The position includes at least all of the initial minor pieces (2 bishops and 2 knights per side), 2 rooks per side, and both queens.

2. 100 % Endgame: there are zero minor pieces, zero rooks and zero queens.

The final tapered evaluation score is computed as a weighted average of the middlegame and endgame evaluations. This is formalized in the following equation:

$$\text{Eval}(\text{position}) = \alpha \cdot \text{middlegameEval} + (1 - \alpha) \cdot \text{endgameEval}$$

Where α represents the proportion of middlegame.

We now need two PST's for each piece, in the following Figure 3.7 there is an example of the middlegame bonus and the endgame bonus for the pawn, as we can see, the pawns in the endgame receive a bonus for being near the promotion squares.

0	0	0	0	0	0	0	0
50	50	50	50	50	50	50	50
10	10	20	30	30	20	10	10
5	5	10	25	25	10	5	5
0	0	0	20	20	0	-5	0
5	-5	-10	0	0	-10	-5	5
5	10	10	-20	-20	10	10	5
0	0	0	0	0	0	0	0

Pawn middlegame PST

0	0	0	0	0	0	0	0
80	80	80	80	80	80	80	80
50	50	50	50	50	50	50	50
30	30	30	30	30	30	30	30
20	20	20	20	20	20	20	20
10	10	10	10	10	10	10	10
10	10	10	10	10	10	10	10
0	0	0	0	0	0	0	0

Pawn endgame PST

Figure 3.7: Tapered Piece Square Tables for pawn.

3.4. 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 [10].

The full implementation of the move generator can be found in `src\move_generator\move_generator_basic.cpp`.

Our move generator uses bitboard operations to quickly and efficiently compute all legal moves available in any given chess position.

3.4.1. Precomputed Attacks

The first step is to precompute attack patterns for each piece type on every square of the board. These patterns are stored as bitboards, typically in arrays indexed by both piece type and square.

For instance, Figure 3.8 illustrates the precomputed attack bitboard for a bishop positioned on the $d4$ square.

3.4.2. Bitboard of Danger Squares

Using the previously precomputed attack patterns, we generate a bitboard representing all the squares currently attacked by the waiting side (the side not having the move). This bitboard is referred to as the danger bitboard. It includes all squares that are unsafe for the king of the side to move, as moving the king to any of these squares would result in an illegal position. Figure 3.9 illustrates an example of this concept.

The danger bitboard is constructed by performing a bitwise OR operation across all legal attacks from the opponent's pieces. For non-sliding pieces such as pawns,

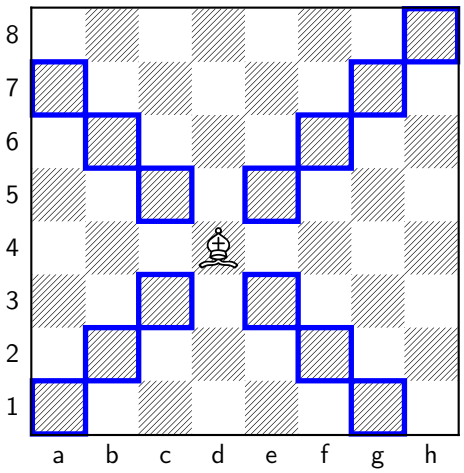
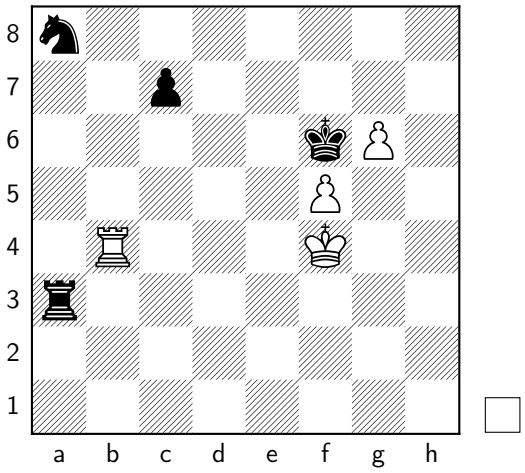
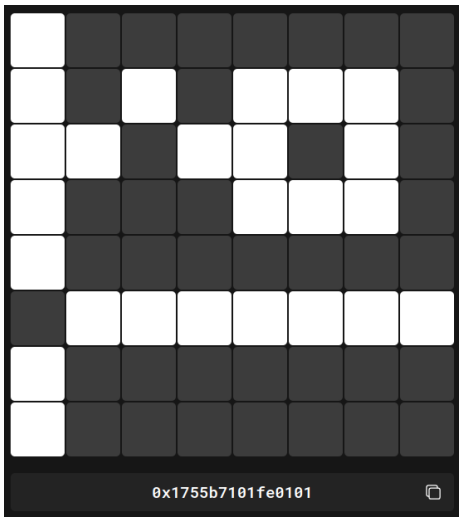


Figure 3.8: Precomputed attack for the bishop on the d4 square.



White is the side to move.



Danger bitboard for the black (waiting) side.

Figure 3.9: Example of a danger bitboard squares attacked by the black side.

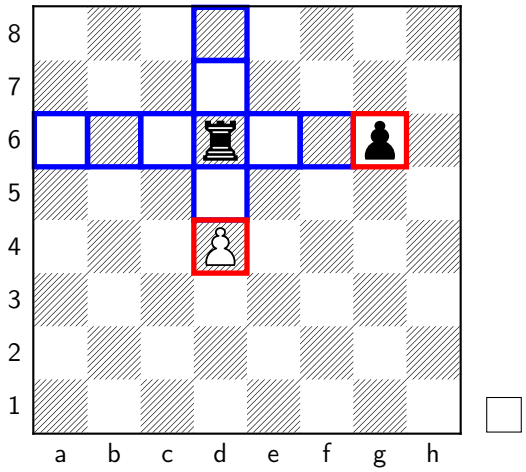


Figure 3.10: Example of blocking pieces.

knights, and kings, their legal attack bitboards match exactly with their precomputed attack patterns.

However, sliding pieces as rooks, bishops, and queens require additional handling. Their attacks depend on the presence of blockers in their movement paths. Figure 3.10 shows an example where the attacks of sliding pieces are limited due to blocking pawns.

In Figure 3.10, the rook sliding attack is being blocked by the pawns.

Currently, calculating legal attacks for sliding pieces involves iterating along orthogonal and diagonal directions until a blocking piece is encountered. This approach is relatively inefficient. In subsequent sections, we present optimization techniques that address this issue.

3.4.3. Bitboard of Pinned pieces

The next challenging aspect of legal move generation is handling pinned pieces. A pinned piece cannot move freely, as doing so would expose its king to check, making the move illegal. An example of a pinned piece is shown in Figure 3.11.

In Figure 3.11, the black rook pins the white knight. If the knight moves, the white king could be captured, making the move illegal.

We must compute a bitboard that contains all pinned pieces on the board. This allows us to restrict their movement accordingly during move generation. However, identifying pinned pieces is computationally expensive, as it requires iterating through the attack patterns of sliding pieces and checking for alignment with the king and potential blockers.

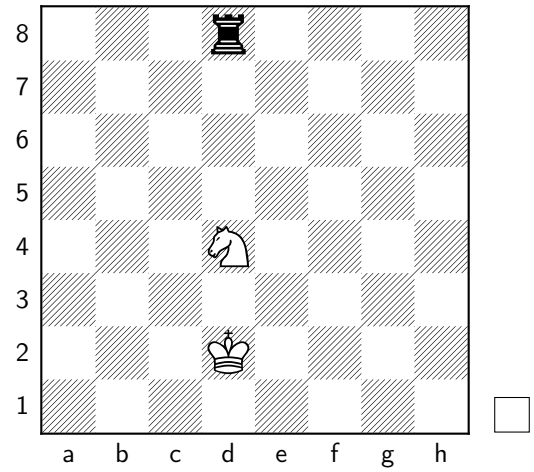
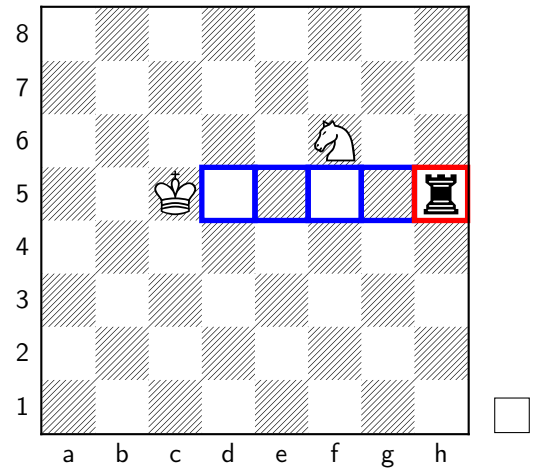


Figure 3.11: Pinned piece.

3.4.4. Capture and push mask

The final bitboards required for handling checks are the *capture mask* and the *push mask*. The capture mask identifies the squares occupied by the checking pieces, while the push mask includes the squares in between the king and the checking piece along the line of attack.



The capture mask is shown in red, and the push mask is shown in blue.

These masks are applied during check situations. In such cases, the only legal responses are:

- Capturing the checking piece (a move to a square in the capture mask).
- Blocking the check (a move to a square in the push mask).
- Moving the king to a safe square outside the danger bitboard.

3.4.5. Legal Move Computation

The final step is to calculate the legal moves of the side to move using the previously calculated information: bitboard of attacks, dangers, pinned pieces, and push and capture masks.

We begin by determining the number of checking pieces, which can be deduced from the number of bits set in the capture mask. Based on this, three main scenarios must be handled:

- *Double check*: If there are two or more checkers, the only legal option is to move the king to a square that is not under attack, outside the danger bitboard. No other piece can legally move in this case.
- *No check*: In the absence of any checks, we iterate through all the pieces belonging to the side to move and generate their legal moves. If a piece is pinned (inside pinned piece bitboard), its movement is constrained to the direction of the pin.
- *Single check*: If exactly one checker is present, we again iterate over all pieces, but the only moves available are to capture the checker (captures inside the capture mask) or to block the check (moves inside the push mask).

Finally, the special moves of castling and *en passant* are handled explicitly by looking for the castling rights and the *en passant* target square in the game state.

3.4.6. Testing the move generator: Perft test

To ensure the correctness of our move generator, we perform what is known as a *Perft test* (performance test, move path enumeration) [22].

Perft is a debugging function in which we generate the entire game tree for a specific position up to a given depth and count all the resulting nodes. We can then compare our results with those of other engines, such as Stockfish. Since Stockfish is widely regarded as highly accurate, it serves as a reliable reference for validating move generation.

In Table 3.2, we present our Perft results alongside those of Stockfish for seven well-known test positions at depth 6. The identical node counts in all cases confirm the correctness of our move generator.

3.5. Move Ordering: MVV-LVA

Once having explained the move generator function, in this chapter we detail the implementation of the move ordering heuristic.

The complete implementation can be found in
`src\move_ordering\move_ordering_MVV_LVA.cpp`.

FEN NAME	*Stockfish Nodes*	*AlphaDeepChess Nodes*
`FEN KIWIPETE`	8031647685	8031647685
`FEN EDWARDS2`	6923051137	6923051137
`FEN STRANGEMOVES`	5160619771	5160619771
`FEN TALKCHESS`	3048196529	3048196529
`FEN TEST4`	706045033	706045033
`FEN TEST4 MIRROR`	706045033	706045033
`FEN START POS`	119060324	119060324

Table 3.2: Perft results at depth 6: comparison between Stockfish and AlphaDeepChess [23].

Victim \ *Attacker*	*P*	*N*	*B*	*R*	*Q*	*K*	*EMPTY*
P	15	14	13	12	11	10	0
N	25	24	23	22	21	20	0
B	35	34	33	32	31	30	0
R	45	44	43	42	41	40	0
Q	55	54	53	52	51	50	0
K	0	0	0	0	0	0	0
EMPTY	0	0	0	0	0	0	0

Table 3.3: MVV-LVA heuristic table: Rows = Victims, Columns = Attackers.

During the search process, the earlier we explore the best move in a position, the better the algorithm performs. In the best-case scenario, if the first move explored is indeed the optimal one, the remaining branches of the tree can be pruned. To achieve this, we sort the legal moves by estimated quality, from best to worst [19].

3.5.1. Most Valuable Victim - Least Valuable Aggressor

The heuristic we implemented is the Most Valuable Victim - Least Valuable Aggressor (MVV-LVA). In this approach, a move receives a high score if it captures a valuable piece using a less valuable one. For example, capturing a queen with a pawn is considered a very strong move [15].

We implemented this heuristic using a look-up table indexed by the moving piece and the captured piece, as shown in Table 3.3. Capturing a queen with a pawn receives a score of 55, while doing the opposite receives 11 points.

3.5.2. Killer moves

The main limitation of MVV-LVA is that it only applies to capture moves. In fact, assigning meaningful scores to non-capturing (quiet) moves is a challenging task. To address this, we implemented the *killer move* heuristic, which assigns high scores to certain quiet moves.

A *killer move* is a quiet, non-capturing move which can cause a cutoff in different

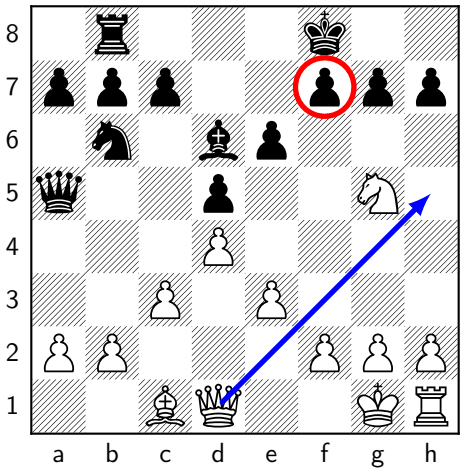


Figure 3.12: Killer move example.

branches of the tree at the same depth [14].

In Figure 3.12, the queen moves to h5, threatening checkmate on f7. This quiet move prunes all other moves that do not respond to the threat.
These moves are remembered and prioritized during move ordering, as they have proven effective in position at the same depth in the search tree. We implemented a table where we store two moves that causes a cutoff per search depth. There could be more than two killer moves, our replacement policy is to always maintain the older killer move found in one slot, and in the other slot store the least recently found.

If a quiet move being evaluated matches one of the killer moves stored at the current search depth, we increase its score by 70 points.

In the following **??**, we created a benchmark to measure the effectiveness of each technique by playing matches with 100 games versus a baseline engine implementation, and we analyze the results of these games at the end of the process to assess the impact of each improvement.

Chapter 4

Improvement Techniques

This chapter documents the implementation of the following techniques used to improve the chess engine:

- Transposition tables with zobrist hashing.
- Move generator with magic bitboards and PEXT instructions.
- Evaluation with king safety and piece mobility parameters.
- Multithread search.
- Search with Late move Reductions.

4.1. Transposition Table

As discused in the previous chapter (see Section 3.2.1), the basic implementation of the chess engine generates a large amount of redundant calculations due to the iterative deepening approach and also the concept of transpositions: situations in which the same board position is reached through different sequences of moves in the game tree. Figure 4.1 illustrates a position that can arise through multiple move orders. Where the white king could go to the g3 square from multiple paths.

Taking advantage of dynamic programming, we 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. [2]

The complete implementation can be found in
`include\utilities\transposition_table.hpp`.

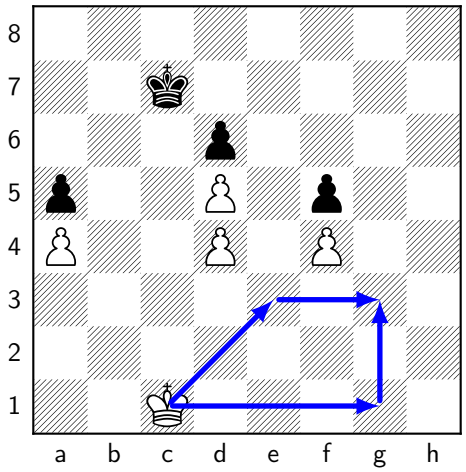


Figure 4.1: Lasker-Reichhelm Position, transposition example

4.1.1. Zobrist Hashing

Zobrist Hashing 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 [24].

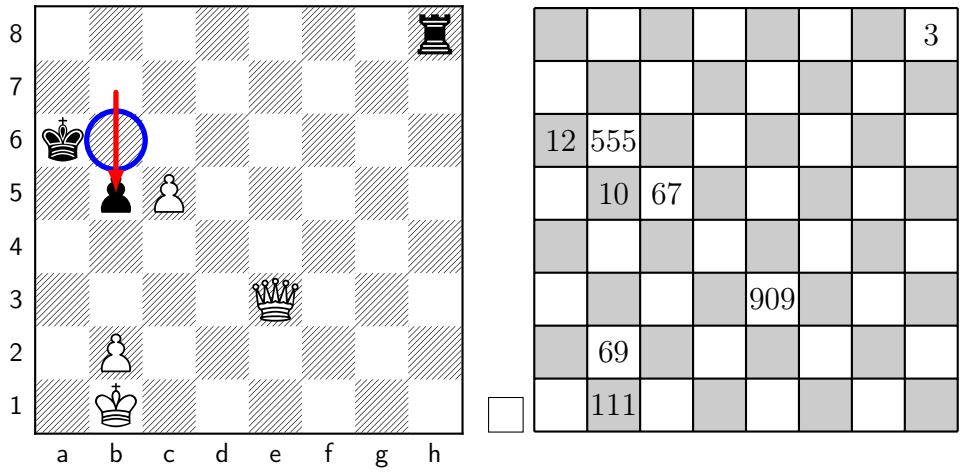
To generate a 64-bit hash for a position, the following steps are followed:

1. Pseudorandom 64-bit numbers are generated for each possible feature of a position:
   a) One number for each piece type on each square — 12 pieces x 64 squares = 768 numbers.
   b) One number to indicate the side to move is black.
   c) Four numbers to represent castling rights (kingside and queenside for both white and black).
   d) Eight numbers to represent the file of an available *en passant* square.
2. The final hash is computed by XOR-ing together all the random numbers corresponding to the features present in the current position.

These random values ensure 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.

The position shown in Figure 4.2 illustrates an example of how the Zobrist hash is computed. The hash value is calculated by XORing the random values associated



Side to move is white. The last move was a pawn advancing from b7 to b5, making en passant available on the b6 square.

Random values corresponding to each piece and the en passant square. The value for Black to move is 62319.

Figure 4.2: Zobrist hash calculation example.

with each element of the position. Since the side to move is White, we do not XOR the value associated with Black to move. The resulting hash is computed as follows:

$$111 \oplus 69 \oplus 909 \oplus 10 \oplus 67 \oplus 12 \oplus 555 \oplus 3 = 458$$

Where $\oplus$ denotes XOR.

4.1.2. Table Entry

Each entry in the transposition table stores the following information:

1. *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.
2. *Evaluation*: The numerical evaluation of the position, as computed by the evaluation function.
3. *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.
4. *Node Type*: Indicates the type of node stored:
   a) *EXACT* the evaluation is precise for this position.
   b) *UPPERBOUND* the evaluation is an upper bound, typically resulting from an alpha cutoff.

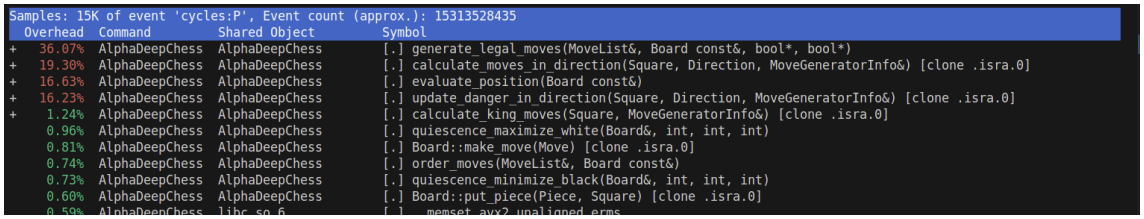


Figure 4.3: Profiling results.

c) *LOWERBOUND* the evaluation is a lower bound, typically resulting from a beta cutoff.

d) *FAILED* entry is empty or with invalid information.

4.1.3. 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.

4.2. Move generator with Magic Bitboards and PEXT instructions

To identify potential performance bottlenecks, we performed profiling on the engine, as shown in Figure 4.3.

The profiling results indicate that the majority of the total execution time is spent in the legal move generation function. Therefore, optimizing this component is expected to yield significant performance improvements.

4.2.1. 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 [12].

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.

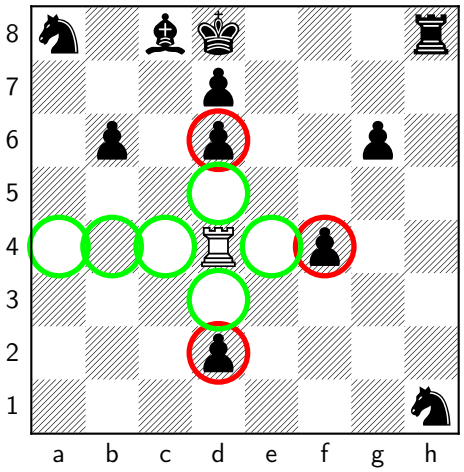


Figure 4.4: Initial chess position with white rook and blockers

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:

  $$\text{Rook} \rightarrow\rightarrow\rightarrow [\text{Pawn1}][\text{Pawn2}]$$

  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 4.4, 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.

First, we mask out all pieces outside the rook's attack pattern or on the board borders, as shown in Figure 4.7.

As illustrated in Figure 4.10, 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.

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 4.11 shows this for the rook:

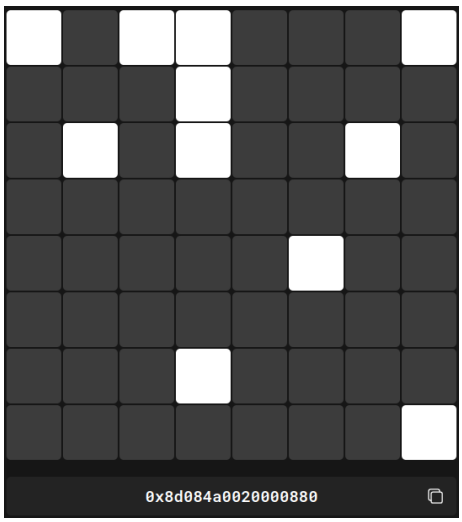


Figure 4.5: Original blockers bitboard

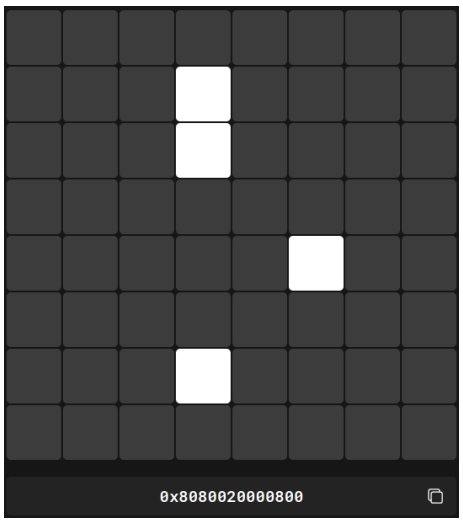


Figure 4.6: Masked blockers bitboard

Figure 4.7: Pre-processing of the blockers bitboard

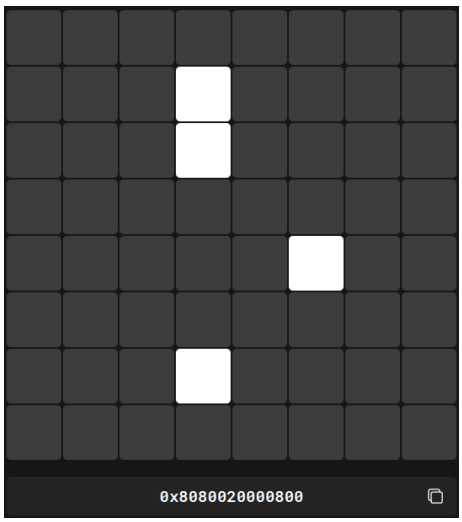


Figure 4.8: Masked blockers bitboard

Magic number

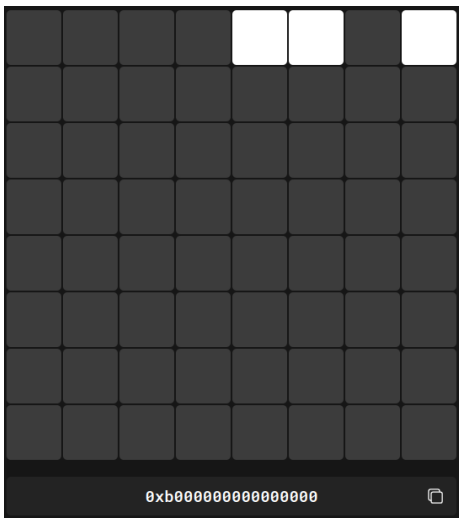


Figure 4.9: Multiplied blockers bitboard

Figure 4.10: Multiplication by magic number to produce an index

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 4.11: Relevant squares for rook piece.

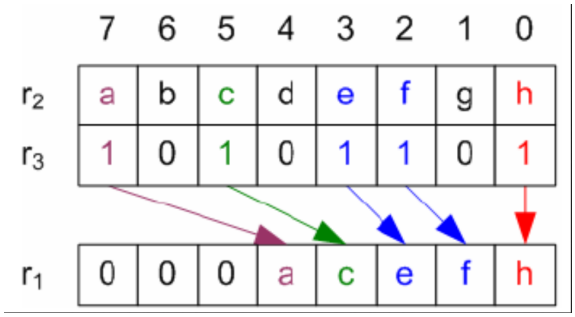


Figure 4.12: Example of the `PEXT` instruction.

The final index is thus computed as:

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

4.2.2. 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 [9]. It is ideally suited for computing our table index.

Figure 4.12 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`.

For our previous example (see Figure 4.4), we only need the full bitboard of blockers and the rook's attack pattern (excluding the borders to reduce space), as illustrated in Figure 4.16.

The final index used to access the lookup table is calculated using the `pext` instruc-

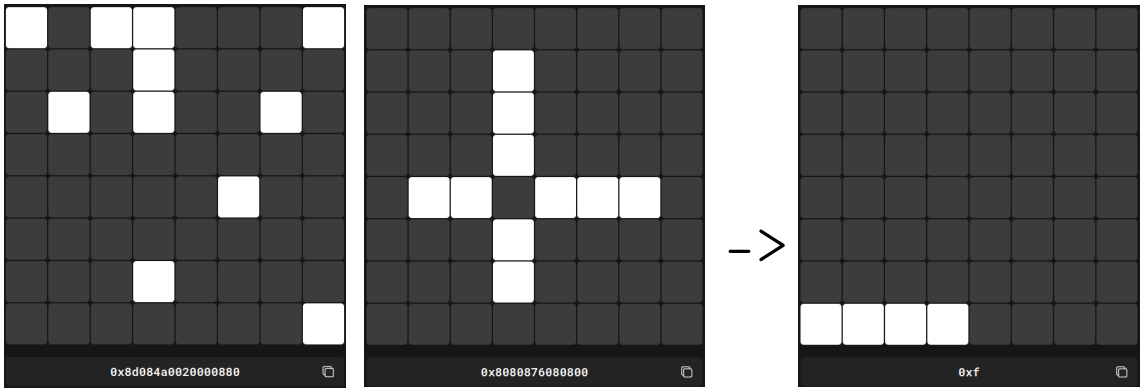


Figure 4.13: Blockers bitboard

Figure 4.14: Rook attack mask

Figure 4.15: Final extracted index

Figure 4.16: index extraction with Pext example

tion as follows:

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

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.

4.3. 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.

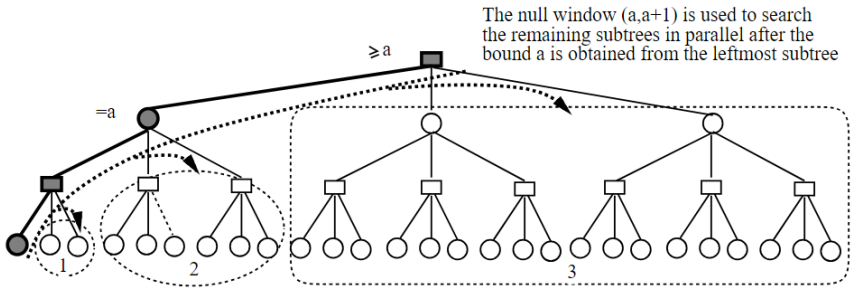


Figure 4.17: Principal variation splitting [8].

4.4. Multithreaded Search

This version of search follows Young Brothers Wait Concept, which 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, ensuring that the most promising move is evaluated first. If this move turns out to be the best and pruning is applied, the search can directly return the final node evaluation. Otherwise, once the first move has been evaluated, the remaining moves are distributed among multiple threads for parallel evaluation. The wait concept refers to these threads that remain idle until the main thread finishes searching the principal variation.

In Figure 4.17, the gray circle and square nodes are considered part of the principal variation, while the remaining white nodes are processed in parallel by multiple threads.

4.5. Late Move Reductions

TODO

Chapter 5

Evaluation of playing strenght

This chapter documents the implementation of the following techniques used to improve the chess engine:

- Transposition tables with zobrist hashing.
- Move generator with magic bitboards and PEXT instructions.
- Evaluation with king safety and piece mobility parameters.
- Multithread search.
- Search with Late move Reductions.

All of these improvements have been compared with each other using *CuteChess*, and the best version was also evaluated against *Stockfish*. These comparisons were conducted on machines provided by GitHub Actions.

5.1. GitHub Actions and Workflows

GitHub Actions is a CI/CD tool integrated into GitHub that allows developers to automate tasks such as building, testing, and deploying code. Workflows are defined in YAML files and specify the tasks to be executed, the jobs or events that trigger them, and the environment in which they run.

In this project, since it is public in a GitHub repository, we used GitHub Actions to automate the testing and evaluation of the chess engine with *Stockfish* and *CuteChess*. A workflow was configured (located at `.github\workflows\manual-workflow.yml`) to compile the engine and run automated games using *CuteChess* between different versions of the engine or against *Stockfish*.

At the end of each section, we provide the results of a 100-game match between the improved engine and a baseline version. The baseline includes only the core techniques discussed in the previous chapter on engine architecture. The purpose of these matches is to measure the improvement in playing strength introduced by each new implementation.

Component	*Transposition Table Bot*	*Basic Bot*
Search	Alpha-beta With Transposition Table	Basic Alpha-beta
Evaluation Function	Materialistic	Materialistic
Move Generator	Basic implementation	Basic implementation
Move Ordering	MVV-LVA	MVV-LVA

Table 5.1: Match configuration: Transposition Table Bot vs Basic Bot

All matches are conducted using the tournament manager *CuteChess* [4] with the following configuration:

- 100 games per match.
- 50 unique random starting positions, each played twice with alternating colors.
- 4 seconds of thinking time per move.
- A 150-move limit per game, after which the game is declared a draw.

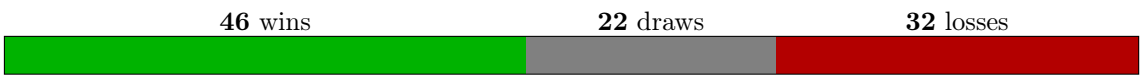
The matches were executed on virtual machines provided by GitHub Actions, using the `ubuntu-latest` runner. At the time of testing, this environment provided 4 CPUs, 16 GB of RAM, a 14 GB SSD, and a 64-bit architecture.

5.2. Transposition Table

5.2.1. Analysis

To evaluate the impact of introducing the transposition table, we conducted the 100-game tournament against the baseline version of the engine. The Table 5.1 details the implementation used for each bot.

As illustrated in following result bar, we see a substantial improvement by adding the transposition table with 46 wins versus 32 losses. The remaining 22 games ended in a draw.



Component	*PEXT instructions Bot*	*Basic Bot*
Search	Alpha-beta With Transposition Table	Basic Alpha-beta
Evaluation Function	Materialistic	Materialistic
Move Generator	PEXT implementation	Basic implementation
Move Ordering	MVV-LVA	MVV-LVA

Table 5.2: Match configuration: PEXT instructions Bot vs Basic Bot

5.3. Move generator with Magic Bitboards and PEXT instructions

5.3.1. Analysis

To evaluate the impact of introducing the move generator accelerated with PEXT instructions, we conducted the 100-game tournament against the baseline version of the engine. The Table 5.2 details the implementation used for each bot.

As illustrated in the following result bar, we achieved a significant performance improvement by adding the PEXT instructions with 46 wins versus 22 losses. The remaining 14 games ended in a draw.

5.4. Evaluation with King Safety and piece mobility

5.4.1. Analysis

To evaluate the impact of introducing the new parameters in the evaluation, we conducted the 100-game tournament against the baseline version of the engine. The Table 5.3 details the implementation used for each bot.

As illustrated in the following result bar, the results are slightly worse compared to the match using the material-only evaluation shown in the following result bar, with 62 wins and 30 losses. This decline may be attributed to the additional computational overhead introduced by evaluating the new parameters. Moreover, while concepts such as king safety and piece mobility are intuitively valuable to human players, the engine may struggle to consistently associate them with actual positional strength.

Component	*PEXT instructions Bot*	*Basic Bot*
Search	Alpha-beta With Transposition Table	Basic Alpha-beta
Evaluation Function	Safety and Mobility	Materialistic
Move Generator	PEXT implementation	Basic implementation
Move Ordering	MVV-LVA	MVV-LVA

Table 5.3: Match configuration: King Safety and Piece mobility eval Bot vs Basic Bot

5.5. Multithreaded Search

5.5.1. Analysis

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

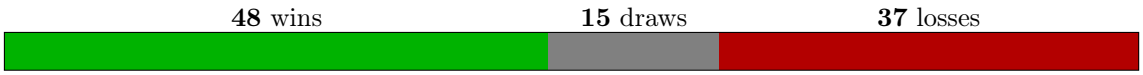
The results are slightly worse compared to the match using the material-only evaluation, with 8 more losses than before. This may be due to the increased computational cost of evaluating these additional parameters. Furthermore, although these are abstract concepts commonly used by humans to assess positions,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

5.6. Late Move Reductions

We experiment with the use of late move pruning, under the assumption that if our move ordering is good, the best move in the position should be among the first explored moves. We reduce the depth by one unit starting from the tenth movement.

Analysis

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



The results are worse, with 15 more losses than the version without this aggresive pruning. **??** This could be because our move ordering is not that strong, and the best move in the position sometimes is in the last positions.

Chapter 6

Conclusions and Future Work

We have successfully developed a competitive chess engine based on an enhanced minimax search with alpha-beta pruning, a straightforward materialistic evaluation, and a Most Valuable Victim-Least Valuable Aggressor (MVV-LVA) move-ordering heuristic.

Building on this basic implementation, we have researched and analyzed the following algorithmic techniques to further improve the bot's playing strength:

- *Transposition tables with Zobrist hashing*: provided an increase in performance by avoiding redundant position evaluations.
- *Move generator with magic bitboards and PEXT instructions*: substantial improvement in move generation.
- *Evaluation with king safety and piece mobility parameters*: showed no clear improvement, likely due to the added computational cost and the difficulty of correlating these factors directly with positional strength.
- *Multithreaded search*: requires further optimization of data structures and algorithms for concurrent access to yield tangible gains.
- *Search with Late Move Reductions*: did not improve performance, as our current move-ordering heuristic is not strong enough to support such aggressive pruning.

The engine achieved an ELO rating of 1900 on Lichess while running on a Raspberry Pi 5 with a 2GB transposition table, demonstrating its efficiency even on resource-constrained hardware.

The engine's Lichess profile can be found at:
`https://lichess.org/@/AlphaDeepChess`

6.1. Future Work

Potential avenues for further improvement include:

- *Neural-network evaluation and move ordering*: Application of neural networks for the evaluation function and the move ordering heuristic, following the steps of *Stockfish*, the actual best chess engine. This could unlock more aggressive pruning strategies.
- *Advanced pruning techniques*: With a stronger evaluation and ordering heuristic, revisit and tune the actual Late Move Reductions, and implement null move pruning technique.
- *Multithreading optimizations*: Refine concurrent search algorithm and redesign shared data structures like transposition tables to support concurrent access.

Chapter 7

Personal Contributions

The following section details the individual contributions made by each team member, Juan and Yi, throughout the development of the AlphaDeepChess project. Each contribution is listed to provide transparency regarding the division of work, highlight specific areas of responsibility, and acknowledge the expertise and effort invested by each author.

Juan Girón Herranz

- Contributed to the alpha-beta pruning algorithm. Taking part in the research and implementation of the iterative deepening and the algorithm core.
- Designed and implemented the bitboard-based move generator, then optimized the calculation of the slider pieces moves with magic bitboards technique and the PEXT hardware instruction. Additionally, integrated of the move generator in the search algorithm.
- Involved in the implementation of the board data structure, with emphasis on the game state bit field design.
- Design and Developed the move data structure, which was also optimized as a bit field to reduce space consumption.
- Designed and implemented the auxiliary data structures for rows, columns, diagonals, and directions. These structures played a key role in simplifying and optimizing bitboard masking operations, enabling more efficient move generation and attack pattern calculations.
- Researched, designed and implemented the transposition table using Zobrist hashing, with total integration in the alpha-beta search.
- Introduced MVV-LVA (Most Valuable Victim, Least Valuable Aggressor), then research and enhanced the algorithm with killer-move heuristics.
- Researched and developed the quiescence search enhancement to avoid the horizon effect in the alpha-beta pruning.
- Developed and tested the search with the late move reductions technique.

- Developed the tampering evaluation, adjusting the weight for the middlegame and endgame evaluations, also contributed to optimizations in the implementation for king safety and piece mobility.
- Created the algorithm to detect threefold repetition using the game's position history and its implementation in the search.
- Implemented part of the UCI command parsing and communication for engine integration.
- Conducted multiple 100-game matches using CuteChess between engine versions to measure the impact of each optimization.
- Created of hundreds of unit tests, which have been a fundamental part of finding bugs and ensuring code quality. This includes Perft testing of the move generator, a standard technique that counts all possible legal positions up to a certain depth to ensure the correctness of move generation in the chess engine.
- Contributed to the development of a helper GUI in Python to facilitate interactive testing of the engine, with support for the UCI protocol.
- Used Linux's `perf` tool, analyzed the CPU overhead of the different parts of the chess engine.
- Compiled and deployed the engine on a Raspberry Pi 5, configuring it as a Lichess.org bot. Running under limited hardware resources, the engine achieved competitive ELO ratings while demonstrating our code's efficiency and portability.

Yi Wang Qiu

- Responsible for the architectural design and full implementation of the alpha-beta pruning algorithm, established as the foundational search technique of the engine. This algorithm was enhanced through iterative refinement, such as aspiration windows, and theoretical benchmarking, enabling effective traversal of the game tree while significantly reducing the computational overhead associated with brute-force minimax strategies.
- Developed an optimized multithreaded search version incorporating the Young Brothers Wait Concept (YBWC), a parallelization paradigm specifically tailored for game tree evaluation.
- Engineered the parsing and command interpretation system compliant with the Universal Chess Interface (UCI) protocol. This subsystem ensures seamless bidirectional communication between the chess engine and external graphical user interfaces, testing suites, and benchmarking frameworks.
- Designed and implemented the core engine abstractions, `Square` and `Board` classes, which supports the representation and manipulation of chess positions. These classes encapsulate critical logic such as coordinate translation

or position translation from FEN, piece tracking, castling rights, and *en passant* possibilities, all integrated with a bitboard backend. This design allows high-level readability while preserving low-level computational performance.

- Constructed the internal board representation model using 64-bit bitboards. This representation supports highly efficient binary operations such as masking, shifting, and logical conjunctions to simulate piece movement and board updates.
- Developed a modular and extensible evaluation system capable of quantifying chess positions through multiple heuristic lenses. The implemented strategies range from basic material balance (expressed in centipawns) to more sophisticated models that incorporate positional features such as game phase, piece activity, mobility scoring, and king vulnerability or safety. These heuristics were designed to be dynamically weighted depending on the stage of the game (opening, middlegame, or endgame).
- Integrated and calibrated the precomputed positional data structures, including piece-square tables to accelerate the static evaluation of positions.
- Designed and authored a suite of automated Python scripts for orchestrating engine versus engine tournaments and performance benchmarking using *Cutechess CLI*. These scripts included configurable match parameters like search time, depth of search, number of games, book of openings or initial positions. They were essential in enabling the reproducibility of experiments, comparison of successive versions of the engine, and quantification of the impact of algorithmic refinements.
- Established a robust continuous integration and delivery (CI/CD) pipeline using GitHub Actions. This infrastructure automated the build, deployment, and testing stages of the engine. Used the above Python scripts to automate the tournaments in an independent machine to avoid wasting time and computation capacity while still developing.
- Contributed to the frontend layer of the project by prototyping a graphical user interface (GUI) in Python, designed to allow interactive execution of the engine in a visual environment. The GUI included subprocess communication features, move display, and optional positional evaluations. Although later iterations focused on headless execution, this interface was key during early debugging and demonstration phases.
- Authored detailed and structured online documentation describing the engine's internal architecture, modular hierarchy, function-level responsibilities, and usage guidelines. The documentation was designed not only as an educational resource for future contributors, but also as a formal exposition of the system's logic for academic evaluation purposes like this exact document. It includes illustrative diagrams or graphs, code references, and configuration examples to support transparency and reproducibility.

Bibliography

[1] M. M. Botvinnik. *Computers, Chess and Long-Range Planning*. Springer New York, NY, 1970. Translated by Kenneth P. Neat.

[2] J. v. d. H. Dennis Breuker, Jos Uiterwijk. Information in transposition tables. University of Limburg, 1997.

[3] F. I. des Échecs. Fide laws of chess. Online, 2023.

[4] C. Developers. Cutechess repository. Github, 2024.

[5] S. Developers. Stockfish chess engine. Online, 2025.

[6] D. Eppstein. 1999 variants of alpha-beta search. UC Irvine, 1999.

[7] D. Eppstein. Which nodes to search? full-width vs. selective search. UC Irvine, 1999.

[8] Y. Gao and T. A. Marsland. Multithreaded pruned tree search in distributed systems. University of Alberta, 1996.

[9] Y. Hilewitz and R. B. Lee. Fast bit compression and expansion with parallel extract and parallel deposit instructions. Princeton University, 2006.

[10] P. E. Jones. Generating legal chess moves efficiently. Online, 2023.

[11] S.-M. Kahlen. Description of the universal chess interface (uci). Online, 2004.

[12] P. Kannan. Magic move-bitboard generation in computer chess. Online, 2007.

[13] C. E. Shannon. Programming a computer for playing chess. Computer History Museum Archive, 1950.

[14] M. Vanthoor. Killer move heuristic. Online, 2024.

[15] M. Vanthoor. Mvv-lva. Online, 2024.

[16] C. P. Wiki. Iterative deepening. Online, 2019.

[17] C. P. Wiki. Tapered evaluation. Online, 2021.

[18] C. P. Wiki. Bitboards. Online, 2022.

[19] C. P. Wiki. Move ordering. Online, 2022.

[20] C. P. Wiki. Piece-square tables. Online, 2022.

[21] C. P. Wiki. Quiescence search. Online, 2024.

[22] C. P. Wiki. Perft. Online, 2025.

[23] C. P. Wiki. Perft results. Online, 2025.

[24] A. L. Zobrist. A new hashing method with application for game playing. The University of Wisconsin, 1970.