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 Álfaro
Rubén Rafael Rubio Cuéllar

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

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 Álfaro
Rubén Rafael Rubio Cuéllar

Convocatoria: *Junio 2025*

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

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

8 de abril de 2025

Dedication

To our younger selves, for knowing the art of chess

Acknowledgments

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

Resumen

AlphaDeepChess: motor de ajedrez basado en podas alpha-beta

Los motores de ajedrez han influido notablemente en el desarrollo de estrategias computacionales y algoritmos de juego desde mediados del siglo XX. Informáticos de la talla de Alan Turing y Claude Shannon sentaron las bases para el desarrollo de este campo. Posteriormente, las mejoras de hardware y software y la evolución de la heurística se asentarían sobre estos cimientos, incluida la introducción de la poda alfa-beta, una optimización del algoritmo minimax que reducía significativamente el número de nodos evaluados en un árbol de juego. Con el aumento de la potencia de cálculo, motores modernos como Stockfish o Komodo aprovechan no sólo las optimizaciones de búsqueda, sino también los avances en heurística y, en algunos casos, la inteligencia artificial mediante redes neuronales.

Palabras clave

motor de ajedrez, poda alfa-beta, algoritmo minimax, búsqueda árbol de juego, killer moves, multihilo, bitboards, tablas de transposición, zobrist hashing, optimización de búsqueda

Abstract

AlphaDeepChess: chess engine based on alpha-beta pruning

Chess engines have significantly influenced the development of computational strategies and game-playing algorithms since the mid-20th century. Computer scientists as renowned as Alan Turing and Claude Shannon set the foundations for the development of the field. Thereafter, hardware and software improvements and the evolution of heuristics would build upon these foundations, including the introduction of alpha-beta pruning, an optimization of the minimax algorithm that significantly reduced the number of nodes evaluated in a game tree. With increasing computational power, modern engines such as Stockfish or Komodo leverage not only search optimizations but also advancements in heuristics and, in some cases, artificial intelligence using neuronal networks.

Keywords

Contents

List of figures

List of tables

Chapter 1

Introduction

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

Chess, one of the oldest and most strategic 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 (GM), was present. It was only a matter of time before computation surpassed human computational capabilities.

In 1997, the chess engine Deep Blue made history by defeating the world champion at the time, Garry Kasparov, marking the first time a computer had defeated a reigning world champion in a six-game match under standard chess tournament time controls[1].

Since then, the development of chess engines has advanced rapidly, moving from rule-based systems to AI-driven models. However, classical search algorithms, such as alpha-beta pruning, continue to be fundamental to understanding the basics of efficient search and evaluation of game trees.

1.1. Where to begin?

Let's start from the beginning. What is chess? Chess is a board game where two players who take white pieces and black pieces respectively compete to first checkmate[2] the opponent.

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

---

[1]https://en.wikipedia.org/wiki/Deep_Blue_(chess_computer)
[2]https://en.wikipedia.org/wiki/Checkmate

referred to as UCI, that provides the interaction with chess engines through user interfaces.

In the following section 1.2, we will talk about the basic concepts of chess, but if you already have the knowledge we recommend you to advance directly to the chapter 2.

1.2. Basic concepts

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

1.2.1. Chessboard

A chessboard is a game board of 64 squares, 8 rows by 8 columns. To refer to each of the squares we mostly use **algebraic notation**[3] using the numbers from 1 to 8 and the letters from "a" to "h". There are also other notations like descriptive notation[4] which is obsolete or ICCF numeric notation[5] due to chess pieces have different abbreviations depending on language.
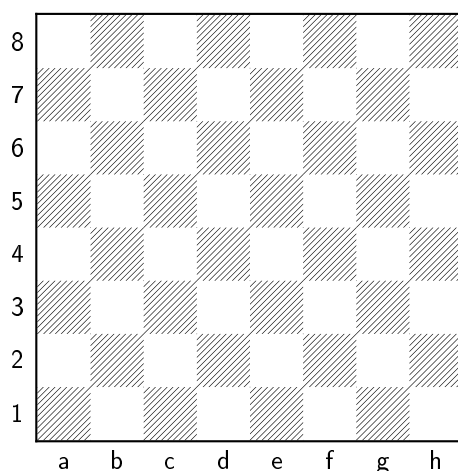

Figure 1.1: Empty chessboard.

---

[3]https://en.wikipedia.org/wiki/Algebraic_notation_(chess)
[4]https://en.wikipedia.org/wiki/Descriptive_notation
[5]https://en.wikipedia.org/wiki/ICCF_numeric_notation

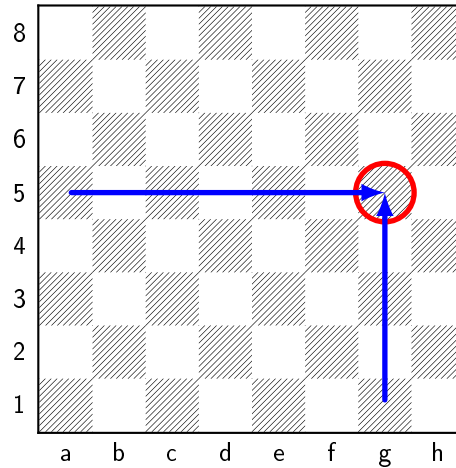For example, $g5$ refers to the following square:

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

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

1.2.2. Chess pieces

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

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

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

The starting position of the chess pieces on a chessboard is the following:
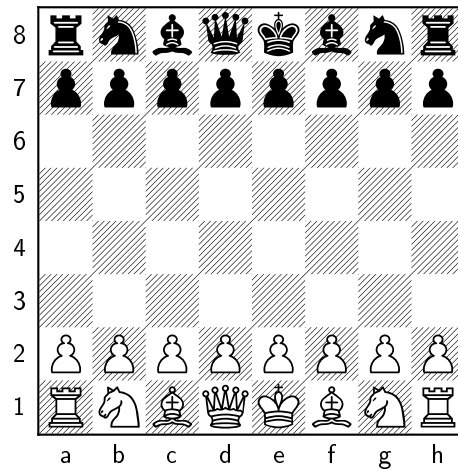
Figure 1.3: Starting position.

Notice that the queen and king are placed in the center columns. The queen is placed on a square of its color, while the king is placed on the remaining central column. The rest of the pieces are positioned symmetrically, as shown in Figure 1.3. This means that the chessboard is divided into two sides relative to the positions of the king and queen at the start of the game:
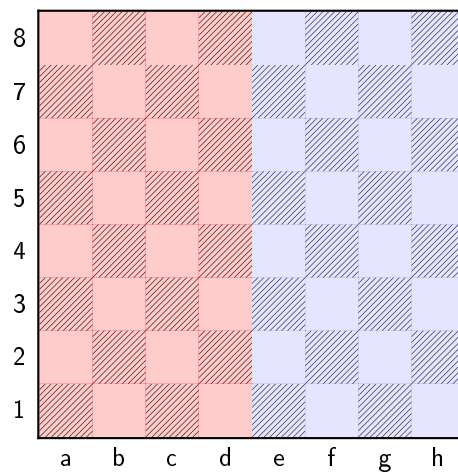
Figure 1.4: King's side (blue) and Queen's side (red).

1.2.3.   Movement of the pieces

1.2.3.1.   Pawn

The pawn can move one square forward, but it can only capture pieces one square diagonally. On its first move, the pawn has the option to move two squares forward.

If a pawn reaches the last row of the opponent's side, it can be promoted[6] to any other piece (except for a king).
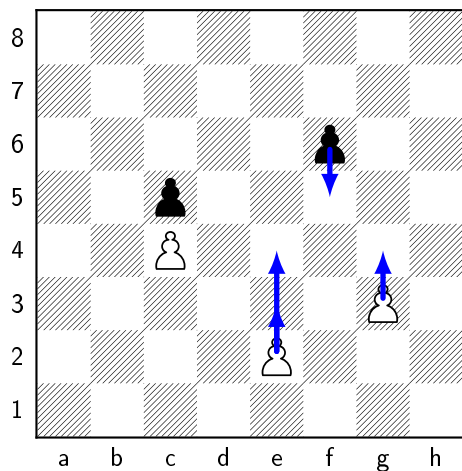
Figure 1.5: Pawn's movement.
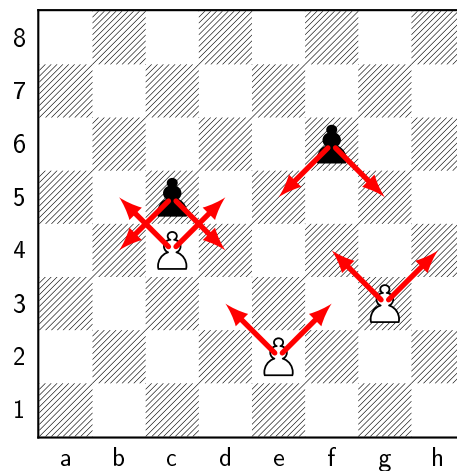
Figure 1.6: Pawn attack.
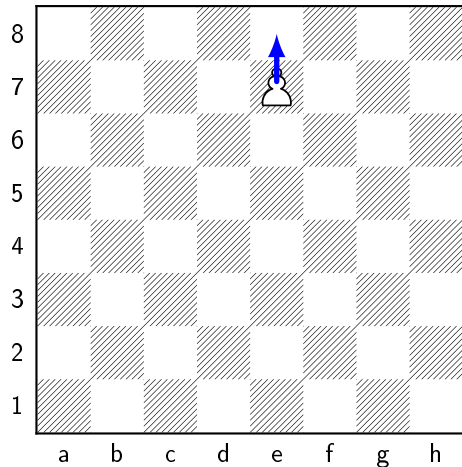
Figure 1.7: Promotion.
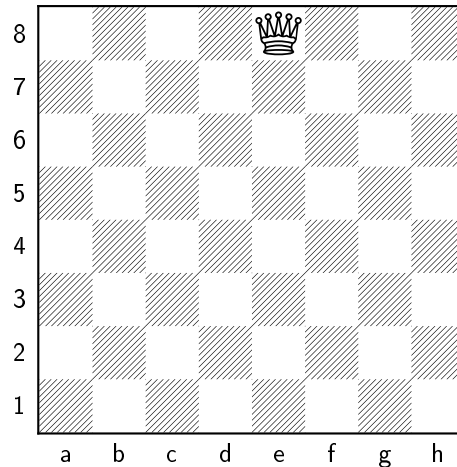
Figure 1.8: Pawn promotes to queen.

There is a specific capture movement which is **en passant** [7]. This move allows a pawn that has moved two squares forward from its starting position to be captured by an opponent's pawn as if it had only moved one square. The capturing pawn must be on an adjacent file and can only capture the en passant pawn immediately after it moves.

---

[6]https://en.wikipedia.org/wiki/Promotion_(chess)
[7]https://en.wikipedia.org/wiki/En_passant
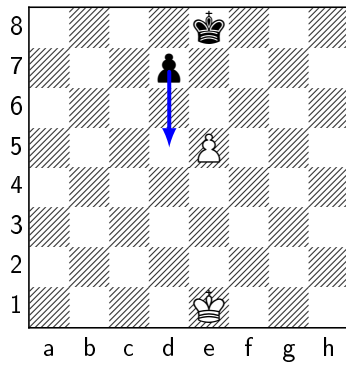
Figure 1.9: En passant (1).
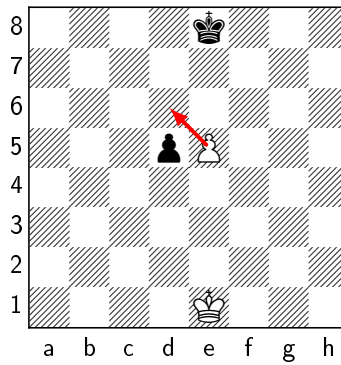
Figure 1.10: En passant (2).
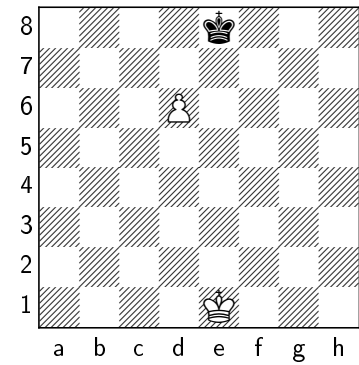
Figure 1.11: En passant (3).

1.2.3.2.   Rook

The rook can move any number of squares horizontally or vertically. It can also capture pieces in the same way.
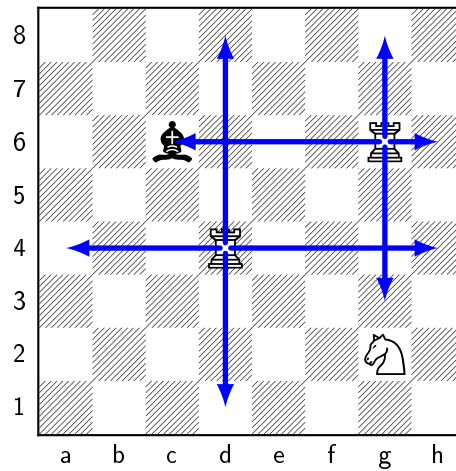
Figure 1.12: Rook's movement.

1.2.3.3.   Knight

The knight moves in an L-shape: two squares in one direction and then one square perpendicular to that direction. The knight can jump over other pieces, making it a unique piece in terms of movement. It can also capture pieces in the same way.
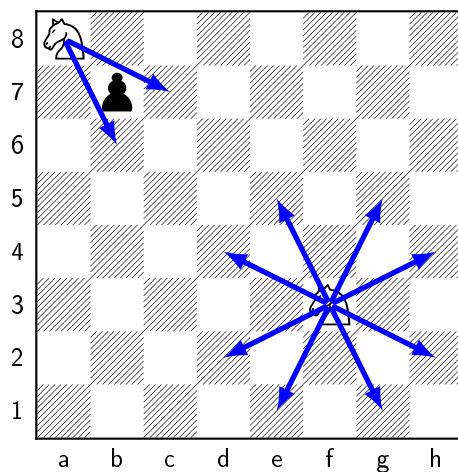
Figure 1.13: Knight's movement.

1.2.3.4.   Bishop

The bishop can move any number of squares diagonally. It can also capture pieces in the same way. Considering that each side has two bishops, one bishop moves on light squares and the other on dark squares.
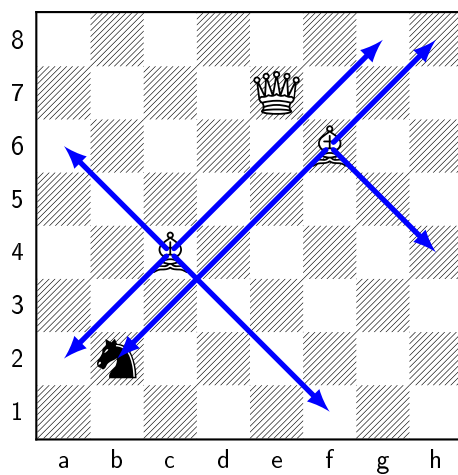
Figure 1.14: Bishop's movement.

1.2.3.5.   King

The king can move one square in any direction: horizontally, vertically, or diagonally. However, the king cannot move to a square that is under attack by an opponent's piece. The king can also capture pieces in the same way. The king is a crucial piece in chess, as the game ends when one player checkmates the opponent's king.
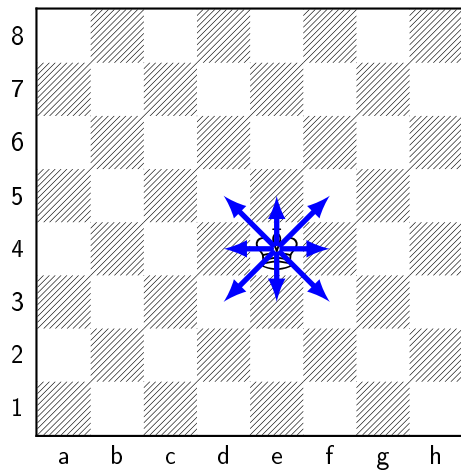
Figure 1.15: King's movement.
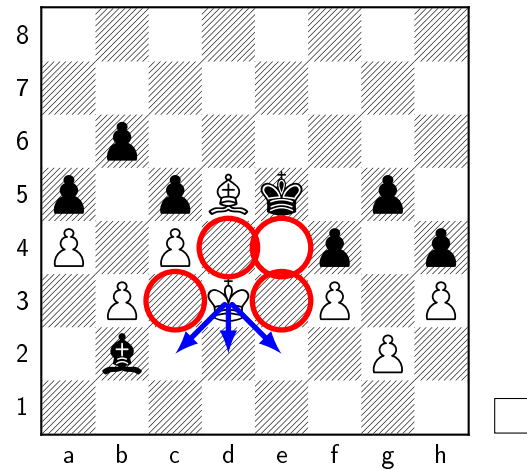
Figure 1.16: White King's movement in a game.

In Figure 1.16, the white king cannot move to $e4$ because the black king is attacking that square. When two kings are positioned close to each other, neither can move to a square adjacent to the other.

Additionally, the king can perform a special move called **castling**, which involves moving the king two squares towards a rook and moving the rook to the square next to the king. Castling has specific conditions which are:

- Neither the king nor the rook involved in castling must have moved previously.

- There must be no pieces between the king and the rook.

- The king cannot be in check, move through a square under attack, or end up in check.
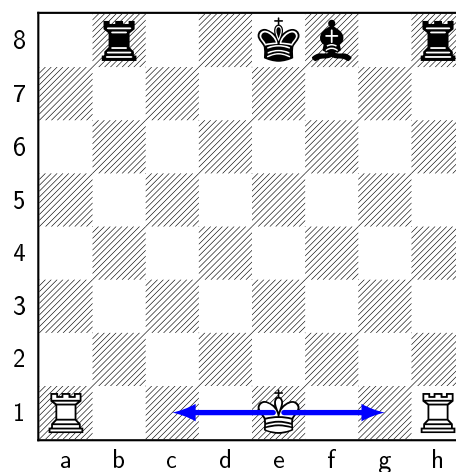
Figure 1.17: Castling

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

1.2.3.6.   Queen

The queen can move any number of squares in any direction: horizontally, vertically, or diagonally. It can also capture pieces in the same way.
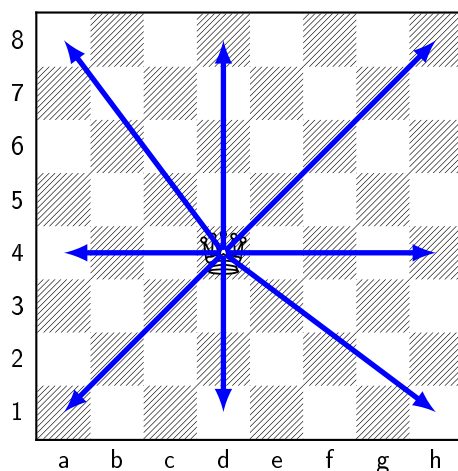
Figure 1.18: Queen's movement.

1.2.4.   Rules

Each player aims to checkmate the opponent's king, which means that 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[8]. A 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).

---

[8]`https://en.wikipedia.org/wiki/Draw_(chess)`

4. Fifty-move rule: if 50 consecutive moves are made by both players without a pawn move or a capture, the game can be declared a draw.

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

6. Dead position: a position where no legal moves can be made, and the game cannot continue. This includes cases like king vs king, king and knight vs king, or king and bishop vs king.
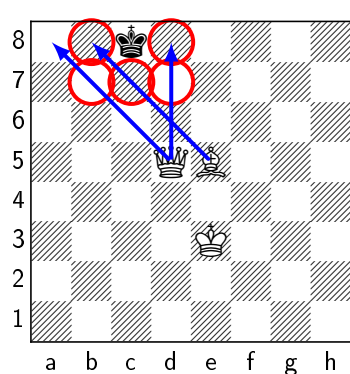
Figure 1.19: Stalemate.
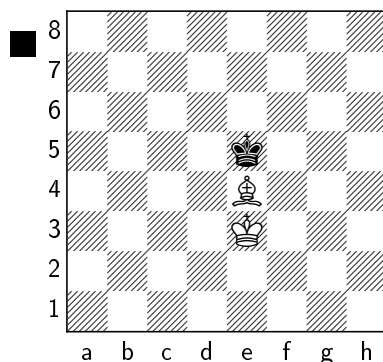
Figure 1.20: Insufficient material.
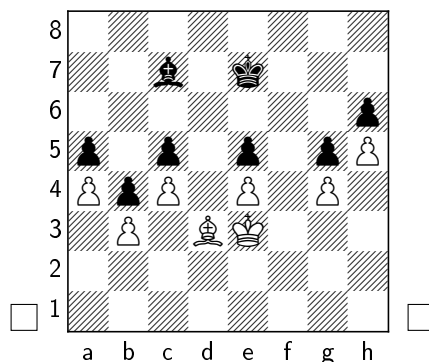
Figure 1.21: Dead position.

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

For more information about chess rules, refer to the Wikipedia page: `https://en.wikipedia.org/wiki/Rules_of_chess`.

1.2.5.   Notation

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

1.2.5.1.   Algebraic notation

In addition to the **algebraic notation** of the squares in section 1.2.1, each piece is identified by an uppercase letter, which may vary across different languages:

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

Table 1.2: Chess piece notation in English and Spanish.

Normal moves (not captures nor promoting)   It is written using the piece uppercase letter plus the coordinate of destination:
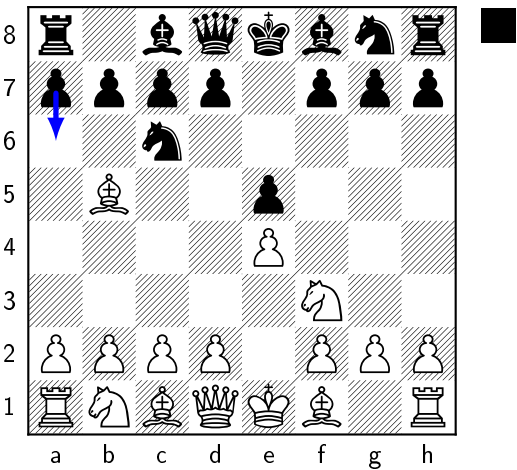
Figure 1.22: Pawn goes to a6.

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

Captures   They are written with an "x" between the piece uppercase letter and coordinate of destination/the captured piece coordinate:
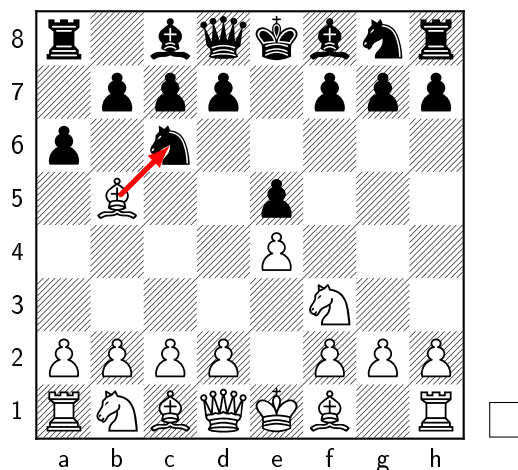
Figure 1.23: Bishop captures knight.

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

Pawn promotion It is written as the pawn's movement to the last row, followed by the piece to which it is promoted:
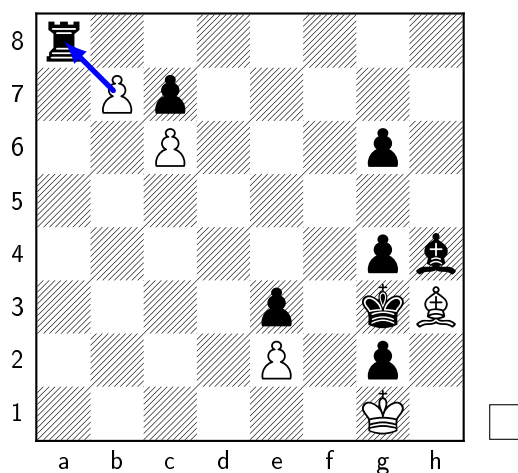
Figure 1.24: Pawn captures rook.

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

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

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

Figure 1.25: Black queen checkmates.

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

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

- **1-0**: White wins.

- **0-1**: Black wins.

- **1/2-1/2**: The game ends in a draw.

For more information about notation, refer to the Wikipedia page: `https://en.wikipedia.org/wiki/Algebraic_notation_(chess)`.

1.2.5.2. Forsyth–Edwards Notation (FEN)

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

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

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

For more information about FEN, refer to the Wikipedia page: `https://en.wikipedia.org/wiki/Forsyth%E2%80%93Edwards_Notation`.

1.2.5.3.   Portable Game Notation (PGN)

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

Listing 1.1: Example of a PGN file

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

For more information about PGN, refer to the Wikipedia page: https://en.wikipedia.org/wiki/Portable_Game_Notation.

1.3.   Objectives

- Develop a functional chess engine using alpha-beta pruning as the core search algorithm.

- Optimize search efficiency by implementing move ordering, quiescence search, and iterative deepening to improve pruning effectiveness.

- Implement transposition tables using Zobrist hashing to store and retrieve previously evaluated board positions efficiently.

- Implement multithreading to enable parallel search.

- Ensure modularity and efficiency so that the engine can be tested, improved, and integrated into chess-playing applications.

- Profile the engine to identify performance bottlenecks and optimize critical sections of the code.

- Compare performance metrics against other classical engines to evaluate the impact of implemented optimizations.

1.4.   Work plan

1. Research phase and basic implementation: understand the fundamentals of alpha-beta pruning with minimax and position evaluation. Familiarize with the UCI (Universal Chess Interface) and implement the move generator with its specific exceptions and rules.

2. Optimization: implement quiescence search and iterative deepening to improve pruning effectiveness.

3. Optimization: improve search efficiency using transposition tables and Zobrist hashing.

4. Optimization: implement multithreading to enable parallel search.

5. Profiling: use a profiler to identify performance bottlenecks and optimize critical sections of the code.

6. Comparation: use Stockfish to compare efficiency generating tournaments between chess engines.

7. Analyze the results and write the final report.

Chapter 2

State of the art

2.1. Board representation

The chessboard is where the game takes place and which serves as the foundation for all operations. In order to store a position setting with its pieces and other additional information like the side to move[1], the castling rights[2] or the fifty-move rule counter[3], we can encounter different types of representations: piece centric, square centric and hybrid solutions. We chose to use bitboards as the primary representation, complemented by a piece list to store the piece on each square (piece centric representation). Additionally, the game state is stored in a bit field.

A **bitboard**[4], also known as a bitset or bitmap, is a 64-bit word structure that efficiently represents a chessboard because it matches with the number of squares: every bit corresponds to a square of the chessboard.

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

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

For more information about board representation, refer to the Chess Programming page: $\texttt{https://www.chessprogramming.org/Board\_Representation}$.

---

[1] Corresponds to the colour that has the turn of movement.

[2] Refers to the possibilities for each side to castle both short and long. Castling is explained in 1.2.5.1.

[3] This rule is explained in 4.

[4] $\texttt{https://www.chessprogramming.org/Bitboards}$

2.2.   Move generation

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

There are two types of move generation:

- **Pseudolegal move generation**: generates all moves without considering whether the king is left in check after the move. It requires additional filtering to remove illegal moves.

- **Legal move generation**: generates only moves that are valid according to the chess rules, ensuring that the king is not left in check. The more accurate, the computationally more expensive it is.

We have preferred to use **legal move generation**. Although it is computationally more expensive than pseudolegal move generation, it simplifies the filtering process and ensures that the generated moves are correct.

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

For more information about move generation, refer to the Chess Programming page: https://www.chessprogramming.org/Move_Generation.

2.3.   Game trees

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

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

2.4. Search algorithms

There are different approaches to find the best move from a position. Some of these search algorithms are: Depth-First Search (DFS)[5], Best-First Search[6] (not to be confused with Breadth-First Search or BFS) and Parallel Search[7].

Depth-First Search refers to the process of exploring each branch of a tree or graph to its deepest level before backtracking[8].

Best-First Search refers to the way of exploring the most promising nodes first. They typically require significant memory resources, as they must store a search space[9] that grows exponentially.

Parallel Search refers to mulithreaded search, a technique used to accelerate search processes by leveraging multiple processors. The downside part is that by utilizing these additional processors effectively poses unique challenges, as searchtree traversal is inherently sequential.

Algorithm	Advantages	Disadvantages
Depth-First Search (DFS)	- Memory-efficient: it does not store the entire search space. - Simple to implement and compatible with alpha-beta pruning.	- May explore less promising branches before finding optimal solutions. - Sequential dependency: it is not easy to parallelize.
Best-First Search	- Explores the most promising nodes first, potentially finding solutions faster.	- Requires significant memory to store the search space, which grows exponentially.
Parallel Search	- Accelerates search by taking advantage of multiple processors. - Effective for modern multiprocessor systems.	- Challenging to manage synchronization and resource utilization. - Limited scalability due to the inherently sequential nature of search tree traversal.

Table 2.1: Comparison of search algorithms.

---

[5]https://www.chessprogramming.org/Depth-First

[6]https://www.chessprogramming.org/Best-First

[7]https://www.chessprogramming.org/Parallel_Search

[8]https://en.wikipedia.org/wiki/Backtracking

[9]Represents the collection of all potential solutions in search algorithms.

Based on this comparison, we selected Depth-First Search for its simplicity, low memory requirements, and compatibility with alpha-beta pruning.

2.4.1.   Minimax algorithm

The **minimax**[10] algorithm is a decision making algorithm that follows DFS principles. It is based on the assumption that both players play optimally, with one player (the maximizer) trying to maximize his score and the other player (the minimizer) trying to minimize his score. It explores the game tree to evaluate all possible moves and determines the best move for the current player.
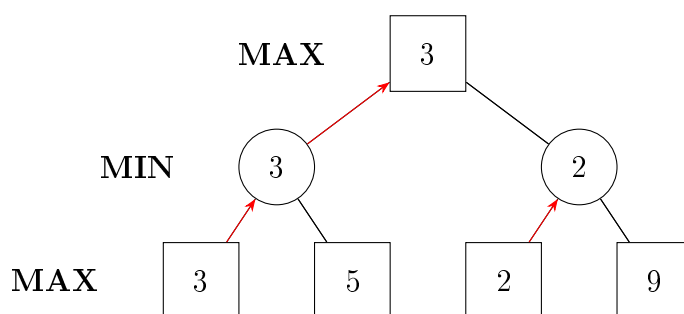
Figure 2.1: Example of minimax.

In this example[11], white is represented by square nodes and black by circle nodes. Each of them wants to maximize or minimize their respective final value in each position. For the leftmost pair of leaf nodes with values of 3 and 5, 3 is chosen because black tries to get the lowest score between them. Then, the other pair of leaf nodes with values of 2 and 9, 2 is chosen for the same reason. Lastly, at the root node, white selects 3 as the maximum number between 3 and 2.

Negamax   is a variant of minimax that simplifies the implementation by assuming that the gain of one player is the loss of the other. This allows the use of a single evaluation function with inverted values. In this project, this variant is not used.

Alpha-beta pruning   [12] is an optimization of minimax that reduces significantly the number of evaluated nodes in the game tree. It uses two values, alpha and beta, to discard branches that cannot influence the final decision improving the efficiency.

---

[10]https://www.chessprogramming.org/Minimax
[11]Note that this example is a binary tree, but there could be more moves or nodes in a real scenario.
[12]https://www.chessprogramming.org/Alpha-Beta
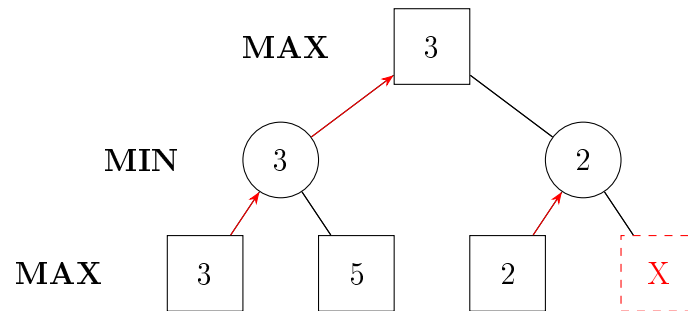
Figure 2.2: Example of alpha-beta pruning.

In this other example, the red dashed node is pruned because it cannot influence the final decision independently of its value. If its value is less than or equal to 2, it will never improve the previously analyzed value of 3. On the other hand, if its value is greater than 2, black will still choose 2 to minimize the score.

Another formal way to explain this is by using *alpha* and *beta* values:
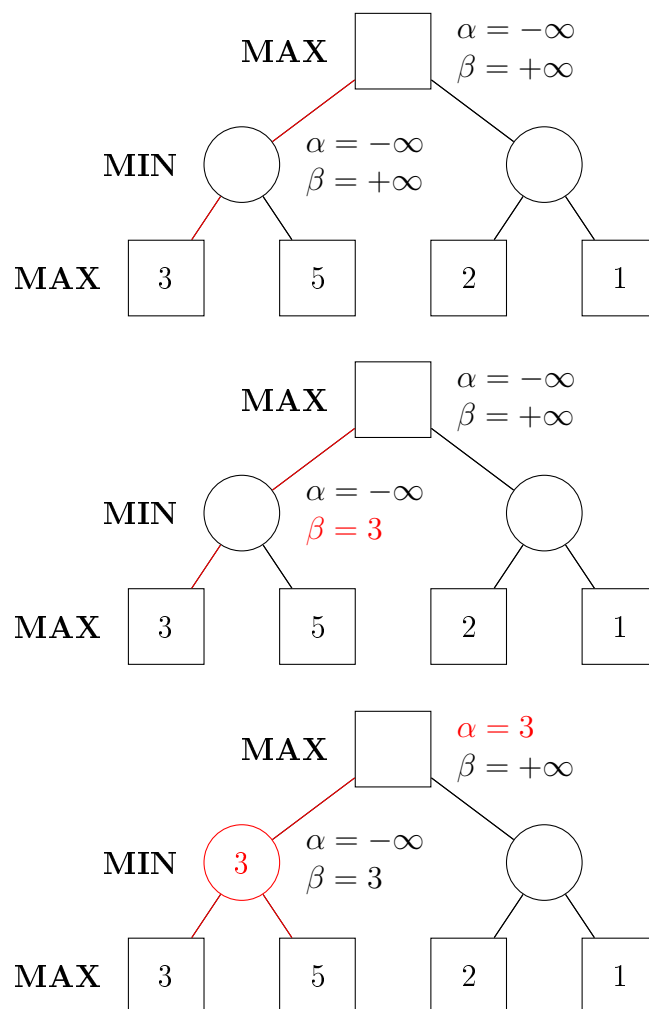
Figure 2.3: Example of alpha-beta pruning with α and β values (1).
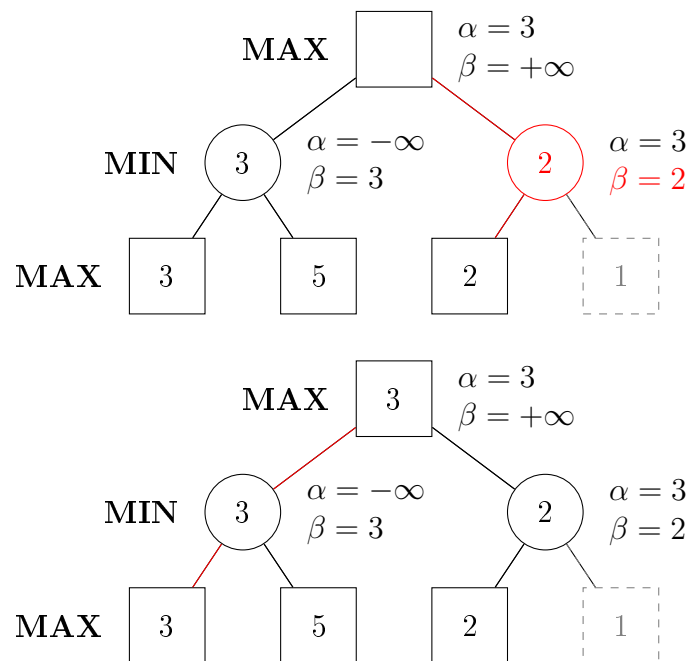
Figure 2.4: Example of alpha-beta pruning with α and β values (2).

2.4.1.1.   Alpha-Beta Enhancements

The alpha-beta algorithm has been further improved over time with various enhancements to increase the overall efficiency. Some of these are: Transposition Tables, Iterative Deepening, Aspiration Windows, Quiescence Search, Move Ordering. . .

Take into consideration that many positions can be reached in different ways. This is formally known as **transpositions**[13]. Just like in dynamic programming[14], the evaluations of different positions are stored in a structure, the **transposition tables**, to avoid repeating the process of searching and evaluating, which improves efficiency. Take this following example:
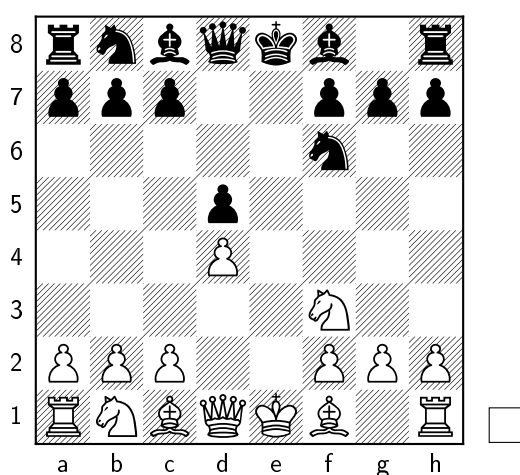
French Defense:
1. e4 e6 2. d4 d5 3. exd5 exd5 4. Nf3 Nf6

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

Figure 2.5: Example of transposition.

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

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

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

---

[13]`https://www.chessprogramming.org/Transposition`
[14]`https://en.wikipedia.org/wiki/Dynamic_programming`

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

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

> *Consider the situation where the last move you consider is QxP. If you stop there and evaluate, you might think that you have won a pawn. But what if you were to search one move deeper and find that the next move is PxQ? You didn't win a pawn, you actually lost a queen. Hence the need to make sure that you are evaluating only quiescent (quiet) positions.*[15]

Finally, alpha-beta algorithm could not perform well without **move ordering**[16]. This is important to ensure that best moves are searched first and to reduce the search space of the game tree. Some of the techniques for move ordering are: Most Valuable victim - Least Valuable Aggressor (MVV-LVA) for captures and killer moves for non-captures.

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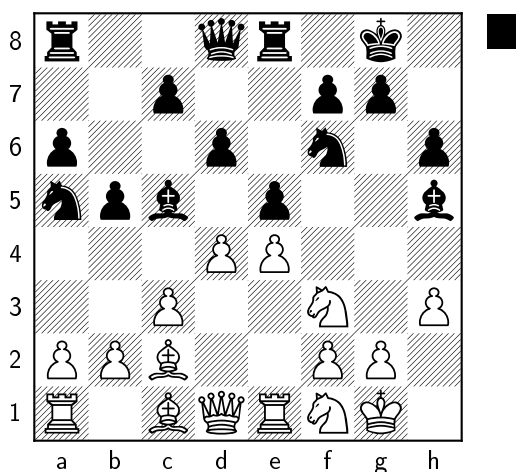

Figure 2.6: Example of MVV-LVA.

---

[15]https://www.chessprogramming.org/Quiescence_Search
[16]https://www.chessprogramming.org/Move_Ordering

In this position, it is black's turn to play after white has moved $d4$, black has the option to capture the pawn on $d4$ with the pawn on $e5$ or with the bishop on $c5$. Between the two capturing movements, the best option is to take $d4$'s pawn with $e5$'s pawn because the pawn has less value than the bishop. Then, after $exd4$, white can re-capture with the pawn on $c3$ or the knight on $f3$. If black had taken the pawn with the bishop, white would have won a bishop for a pawn. This simple heuristic that orders what is best in capturing movements for each side can efficiently evaluate tactical exchanges and focus on moves that are more likely to yield a favorable outcome.

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

2.5.   Evaluation

For each position, a numerical value is assigned representing how favorable the position is for one side (positive for white and negative for black). These values guide the engine in selecting the best moves during the search process. Generally, this value is expressed in centipawns (cp), which represent one hundredth of a pawn's value.

There are two ways to implement it:

- **Traditional hand-crafted evaluation**

- **Multi-layer neuronal networks**

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

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

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

Table 2.2: Standard values assigned to chess pieces in centipawns.

By summing up the piece values for each side, we end up with the concept of **material**. Over the time, there have been different point values for each piece. A

table can be found in `https://www.chessprogramming.org/Point_Value` in *Basic values* section.

There are other material considerations like *bonus for the bishop pair* depending on the position that can be advantageous to control squares of different color, or **insufficient material**, which refers to situations where neither side has enough pieces to deliver checkmate. This is also mentioned in Section 1.2.4.

However, the higher the level one aims to reach in chess, the greater the need to evaluate other aspects.

There are squares in which the pieces have less value because of their activity. Consider the following example:
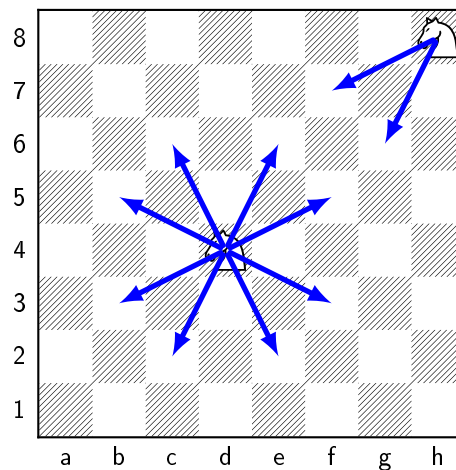
Figure 2.7: Knight's movement on corner square and center square.

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

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

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

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

For more information about piece-square tables, refer to the Chess Programming page in *Piece-Square Tables* section: `https://www.chessprogramming.org/Simplified_Evaluation_Function`.

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

- **Opening**: where players develop their minor pieces[17], control the center of the board and try to ensure king safety.

- **Middlegame**: where players try to create tactical opportunities and attack the opponent's position once they have developed their pieces and secured their king.

- **Endgame**: where players have fewer pieces on the board and the focus is on promoting pawns and achieving checkmate.

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

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

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

This score is calculated as the number of legal moves adjusted for certain factors like blocked moves by friendly pieces (pieces of the same color as the current piece) and enemy attacks[18].

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

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

All these ensures that positions with exposed kings are evaluated as less favorable. If there are open[19] or semi-open[20] files near the king, the position is penalized further, as these files provide attacking opportunities for enemy rooks and queens.

---

[17]Knights and bishops.

[18]Refer to the squares controlled by the opponent.

[19]Columns where there are no pawns of either color.

[20]Columns where there are no pawns of one color but at least one pawn of the opposite color.

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

This can be answered by playing against other engines and analyzing the results. The most common way to do this is by using the **Computer Chess Rating Lists**[21] which ranks chess engines based on their performance in various tournaments and matches. By the time being, we have chosen to compare different versions of the engine with Stockfish, currently ranked as the number one on the list.

---

[21]https://www.computerchess.org.uk/ccrl/

Chapter 3

Work description

3.1. Modules

3.1.1. Board

3.1.2. Move generator

3.1.3. Move ordering

3.1.4. Evaluation

3.1.5. Search

3.2. Code implementation

3.2.1. Data representation

Use of uint64_t as bitboards to store the board information and other code structures and classes justifying why is efficient.

3.2.2. Initialized memory

Some tables are memory initialized instead of computed, explain it.

3.3. Additional tools and work

3.3.1. Board visualizer using Python

3.3.2. Comparation using Cutechess and Stockfish engine

3.3.3. Profiling

4

Conclusions and Future Work

...

The next steps to be implemented would be the application of neural networks (NNUE[1]) which, although intended for CPUs, could be thought of as a streamlined evaluation with GPUs as performed by Leela Chess Zero.

---

[1]https://en.wikipedia.org/wiki/Efficiently_updatable_neural_network

Personal contributions

Student1

Al menos dos páginas con las contribuciones del estudiante 1.

Student 2

Al menos dos páginas con las contribuciones del estudiante 2.

Apéndice A

Título del Apéndice A

Los apéndices son secciones al final del documento en las que se agrega texto con el objetivo de ampliar los contenidos del documento principal.

Apéndice B

Título del Apéndice B

Se pueden añadir los apéndices que se consideren oportunos.

–¿Qué te parece desto, Sancho? – Dijo Don Quijote –
Bien podrán los encantadores quitarme la ventura,
pero el esfuerzo y el ánimo, será imposible.

Segunda parte del Ingenioso Caballero
Don Quijote de la Mancha
Miguel de Cervantes

–Buena está – dijo Sancho –; fírmela vuestra merced.
–No es menester firmarla – dijo Don Quijote–,
sino solamente poner mi rúbrica.

Primera parte del Ingenioso Caballero
Don Quijote de la Mancha
Miguel de Cervantes