
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

18 de mayo de 2025

Dedication

*To our younger selves, for knowing the art of
chess*

Acknowledgments

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

Abstract

AlphaDeepChess: chess engine based on alpha-beta pruning

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

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

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

Keywords

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

Resumen

AlphaDeepChess: motor de ajedrez basado en podas alpha-beta

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

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

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

Palabras clave

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

Contents

1. Introduction	1
1.1. Objectives	1
1.2. Work plan	2
1.3. Basic concepts	2
1.3.1. Chessboard	3
1.3.2. Chess pieces	4
1.3.3. Movement of the pieces	6
1.3.4. Rules	10
1.3.5. Notation	11
2. State of the art	17
2.1. Game trees	17
2.2. Search algorithms	17
2.2.1. Minimax algorithm	20
2.3. How can we determine the strength of our engine?	21
2.3.1. Profiler	21
2.3.2. UCI	21
2.3.3. CustomTkinter	22
2.3.4. Cutechess	22
2.3.5. Stockfish	22
2.3.6. GitHub Actions and workflows	23
3. Engine Development	25
3.1. Chessboard Representation: Bitboards	26
3.2. Search Algorithm: The Engine Core	28
3.3. Evaluation	32
3.4. Move Ordering	35
3.5. Improvements	37
3.5.1. Transposition Table	37
3.5.2. Move generator with Magic Bitboards and PEXT instructions	40
3.5.3. Evaluation with King Safety and piece mobility	45
3.5.4. Search Multithread	45

3.5.5. Late Move Reductions	47
3.6. Additional tools and work	48
3.6.1. Interactive board visualizer using CustomTkinter	48
3.6.2. Testing engine strength with Cutechess, Stockfish, and GitHub Actions	49
4. Conclusions and Future Work	51
Personal contributions	53
Bibliography	57

List of figures

1.1. Empty chessboard.	3
1.2. Example: square <i>g5</i> highlighted and arrows pointing to it.	4
1.3. Starting position.	5
1.4. King's side (blue) and Queen's side (red).	5
1.5. Pawn's movement.	6
1.6. Pawn attack.	6
1.7. Promotion.	6
1.8. Pawn promotes to queen.	6
1.9. En passant (1).	7
1.10. En passant (2).	7
1.11. En passant (3).	7
1.12. Rook's movement.	7
1.13. Knight's movement.	8
1.14. Bishop's movement.	8
1.15. King's movement.	9
1.16. White King's movement in a game.	9
1.17. Castling	9
1.18. Queen's movement.	10
1.19. Stalemate.	11
1.20. Insufficient material.	11
1.21. Dead position.	11
1.22. Pawn goes to <i>a6</i>	12
1.23. Bishop captures knight.	12
1.24. Pawn captures rook.	13
1.25. Black queen checkmates.	14
2.1. Example of minimax.	20
3.1. List of bitboards data structure example	26
3.2. Bitboard mask operation example	27
3.3. Example of alpha-beta pruning with α and β values.	29
3.4. Horizon effect position example	30

3.5. Materialistic eval formula. Where $V(x)$ denotes the value of piece x . .	32
3.6. Knight's movement on corner vs in center.	33
3.7. Piece Square Table for the bishop.	33
3.8. Tapered evaluation formula, where α represents the proportion of middlegame.	34
3.9. Tapered Piece Square Tables for pawn.	34
3.10. Killer move example. The queen moves to h5, threatening checkmate on f7. This quiet move prunes all other moves that do not respond to the threat.	36
3.11. Lasker-Reichhelm Position, transposition example	37
3.12. Zobrist hash calculation example	39
3.13. 64MB Transposition Table bot vs basic bot	40
3.14. Profiling results	40
3.15. Initial chess position with white rook and blockers	41
3.16. Pre-processing of the blockers bitboard	42
3.17. Multiplication by magic number to produce an index	42
3.18. Relevant squares for rook piece.	43
3.19. Example of the PEXT instruction: extracting bits from <code>r2</code> using <code>r3</code> as a mask, and storing the result in <code>r1</code> . [Yedidya Hilewitz and Ruby B. Lee] (2006)	43
3.20. index extraction with Pext example	44
3.21. Move generator with PEXT instructions bot vs basic bot	44
3.22. King Safety and Piece mobility evaluation bot vs basic bot	45
3.23. Principal variation splitting. [Yaoqing Gao and T. A. Marsland] (1996)	46
3.24. Multithread Search bot vs basic bot	47
3.25. Reductions Search bot vs basic bot	48
3.26. AlphaDeepChess GUI	49

List of tables

1.1. Number of chess pieces by type and color.	4
1.2. Chess piece notation in English and Spanish.	11
3.1. Standard values assigned to chess pieces in centipawns.	32
3.2. MVV-LVA heuristic table: Rows = Victims, Columns = Attackers. . .	35

Chapter 1

Introduction

“The most powerful weapon in chess is to have the next move”

— David Bronstein

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

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

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

1.1. Objectives

The objectives of this project are the following:

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

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

1.2. Work plan

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

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

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

1.3. Basic concepts

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

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

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

1.3.1. Chessboard

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

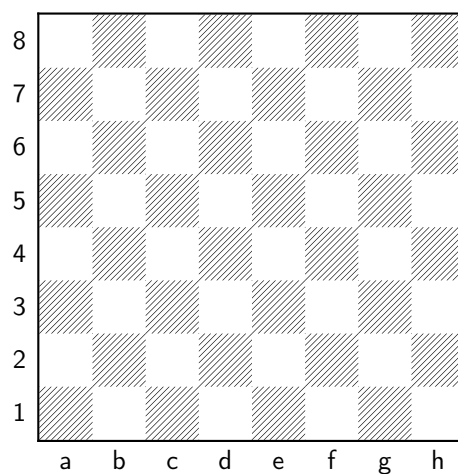


Figure 1.1: Empty chessboard.

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

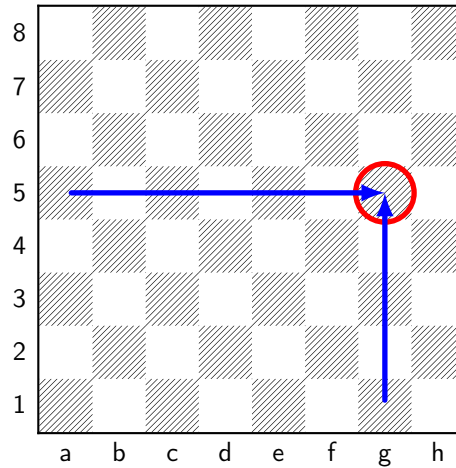


Figure 1.2: Example: square $g5$ highlighted and arrows pointing to it.

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

1.3.2. Chess pieces

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

































Piece	White Pieces	Black Pieces	Number of Pieces
King			1
Queen			1
Rook	 	 	2
Bishop	 	 	2
Knight	 	 	2
Pawn	       	       	8

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

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

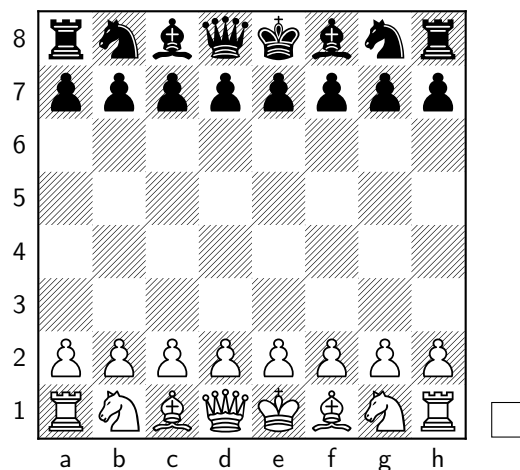


Figure 1.3: Starting position.

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

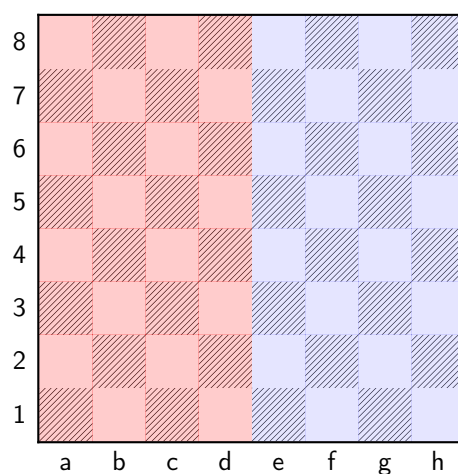


Figure 1.4: King's side (blue) and Queen's side (red).

1.3.3. Movement of the pieces

1.3.3.1. Pawn

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

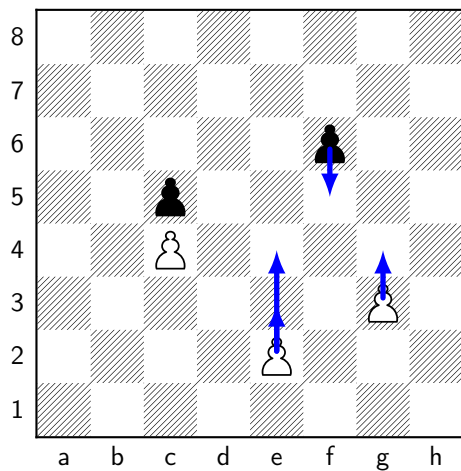


Figure 1.5: Pawn's movement.

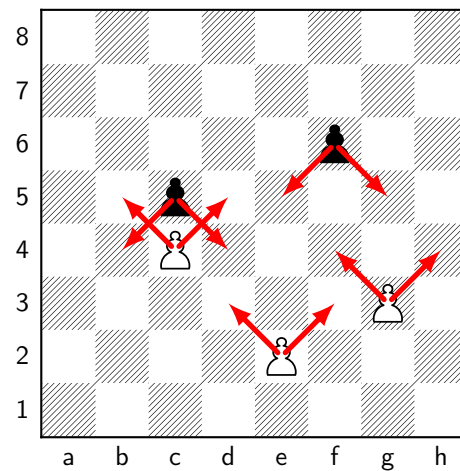


Figure 1.6: Pawn attack.

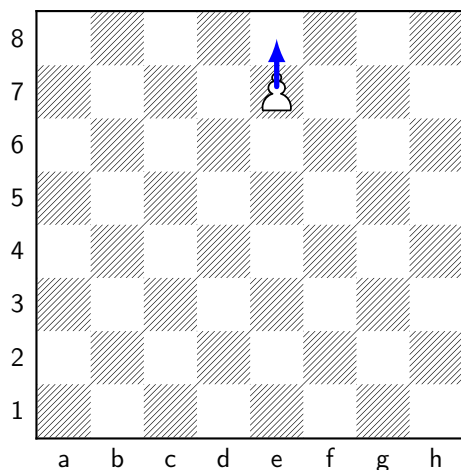


Figure 1.7: Promotion.

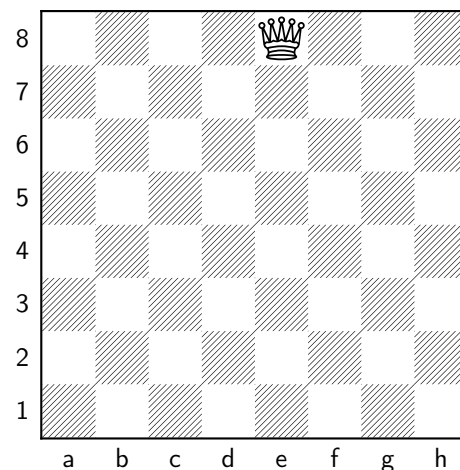


Figure 1.8: Pawn promotes to queen.

There is a specific capture movement which is **en passant**. This move allows a pawn that has moved two squares forward from its starting position to be captured by an

opponent's pawn as if it had only moved one square. The capturing pawn must be on an adjacent file and can only capture the en passant pawn immediately after it moves.

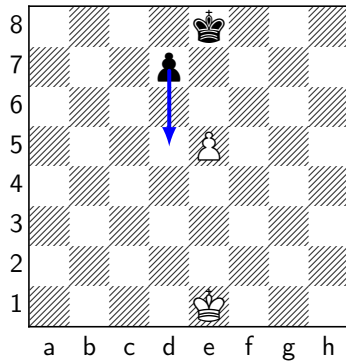


Figure 1.9: En passant (1).

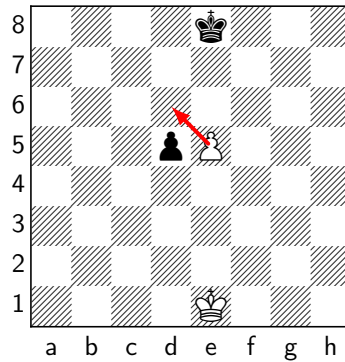


Figure 1.10: En passant (2).

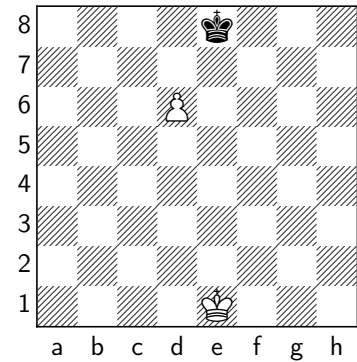


Figure 1.11: En passant (3).

1.3.3.2. Rook

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

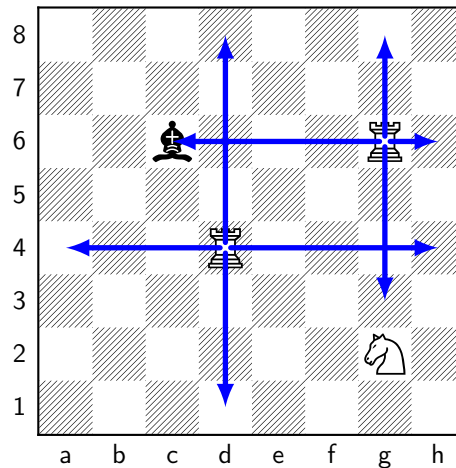


Figure 1.12: Rook's movement.

1.3.3.3. Knight

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

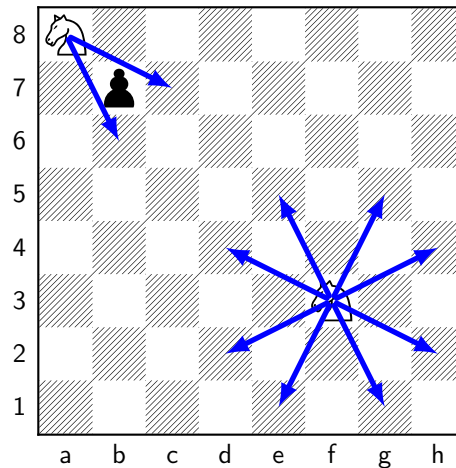


Figure 1.13: Knight's movement.

1.3.3.4. Bishop

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

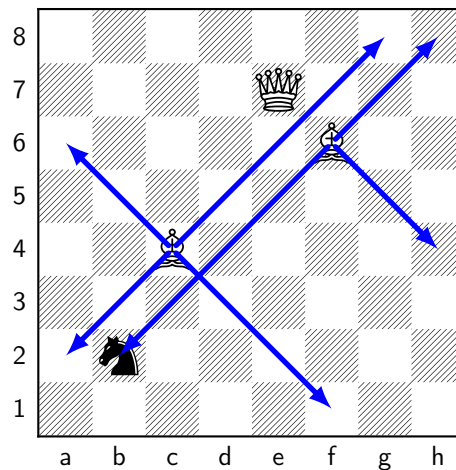


Figure 1.14: Bishop's movement.

1.3.3.5. King

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

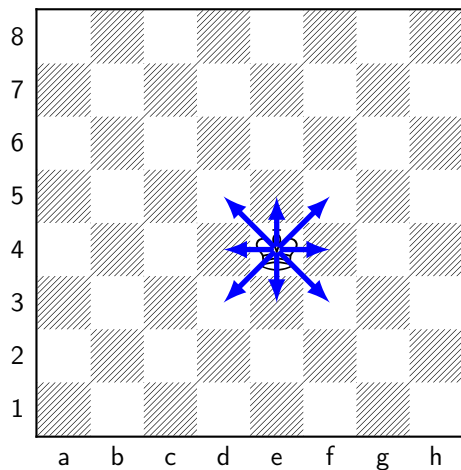


Figure 1.15: King's movement.

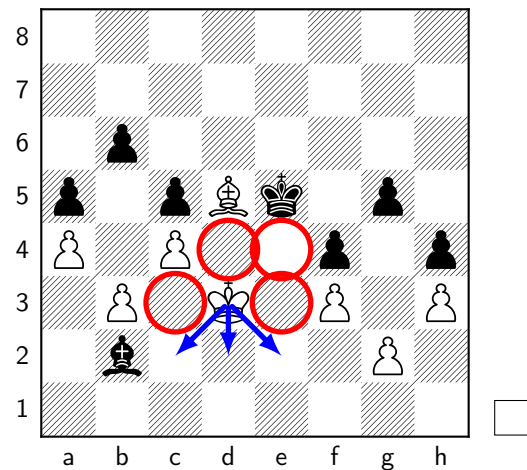


Figure 1.16: White King's movement in a game.

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

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

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

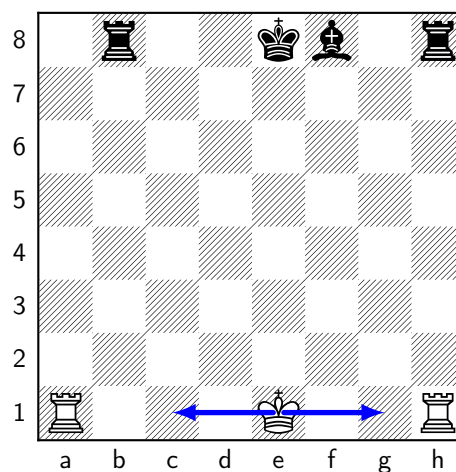


Figure 1.17: Castling

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

1.3.3.6. Queen

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

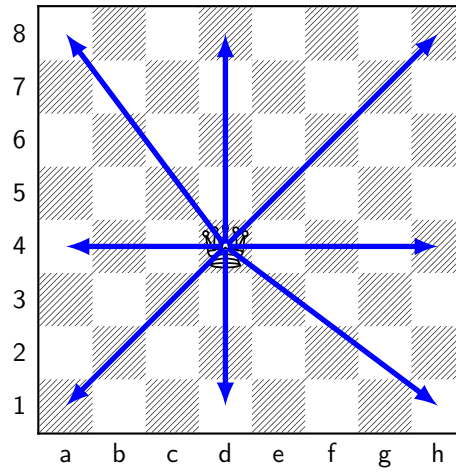


Figure 1.18: Queen's movement.

1.3.4. Rules

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

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

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

5. Mutual agreement: both players can agree to a draw at any point during the game.
6. Dead position: a position where no legal moves can be made, and the game cannot continue. This includes cases like king vs king, king and knight vs king, or king and bishop vs king.

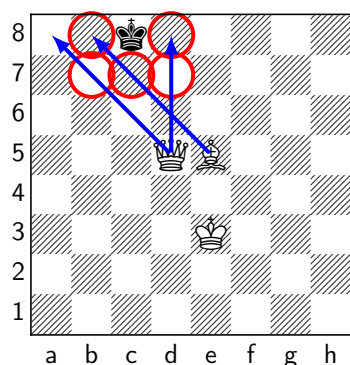


Figure 1.19: Stalemate.

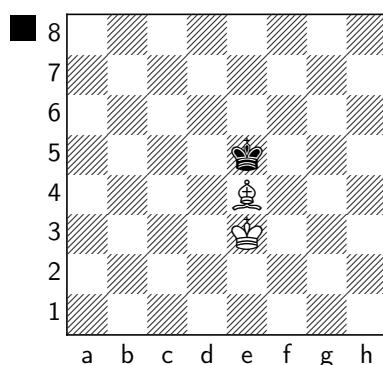


Figure 1.20: Insufficient material.

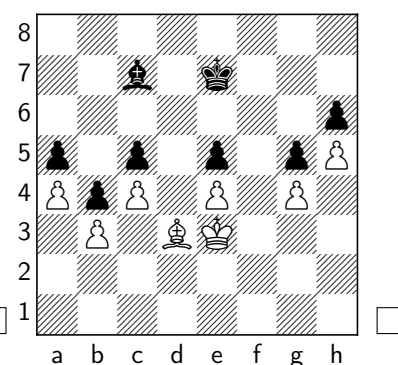


Figure 1.21: Dead position.

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

1.3.5. Notation

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

1.3.5.1. Algebraic notation

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

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

Table 1.2: Chess piece notation in English and Spanish.

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

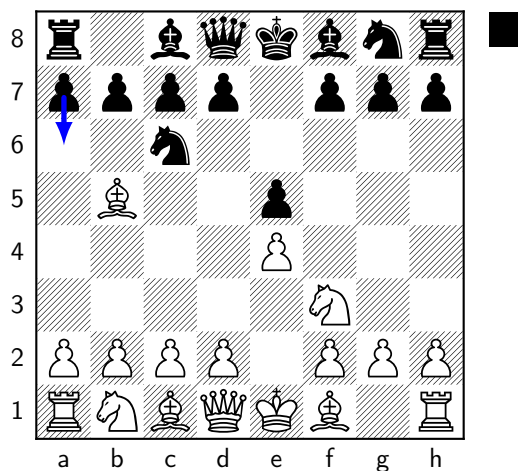


Figure 1.22: Pawn goes to a6.

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

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

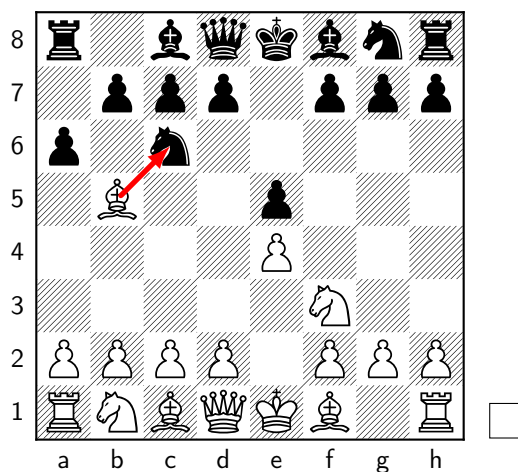


Figure 1.23: Bishop captures knight.

In Figure 1.23, the white bishop capturing the black knight is written as *Bxc6*. If it

were black's turn, the pawn on *a6* could capture the white bishop, and it would be written as *Pxb5* or simply *axb5*, indicating the pawn's column.

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

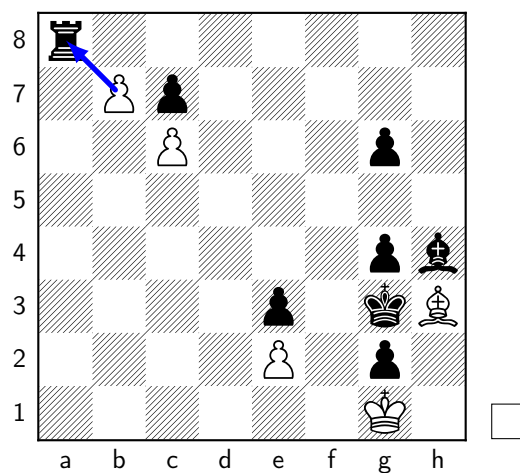


Figure 1.24: Pawn captures rook.

In Figure 1.24, white pawn capturing and promoting in *a8* to a queen is written as *bx a8Q* or *bx a8 = Q*.

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

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

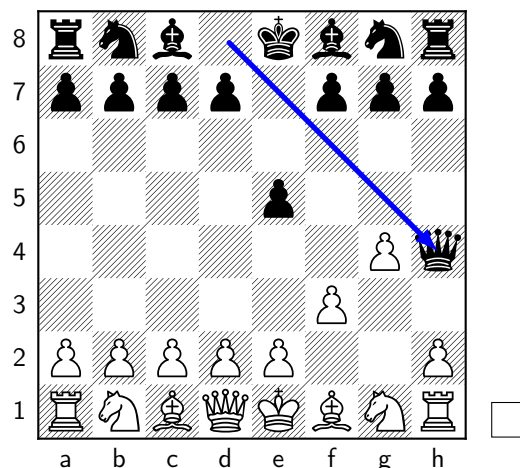


Figure 1.25: Black queen checkmates.

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

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

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

1.3.5.2. Forsyth–Edwards Notation (FEN)

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

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

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

1.3.5.3. Portable Game Notation (PGN)

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

Listing 1.1: Example of a PGN file

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


Chapter 2

State of the art

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

2.1. 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.

2.2. Search algorithms

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

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

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

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

```

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

```

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

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

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

```

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 = Remove the node with the highest
8       priority from PQ
9     If current is the goal:
10       Return the path to the goal
11
12     For each neighbor w of current in G.adjacentEdges(
13       current):
14       If w is not visited:
15         Calculate priority for w (e.g., using a
            heuristic)
16         Add w to PQ with the calculated priority
17         Mark w as visited

```

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

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

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

2.2.1. Minimax algorithm

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

Listing 2.3: Pseudocode of the Minimax algorithm.

```

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

```

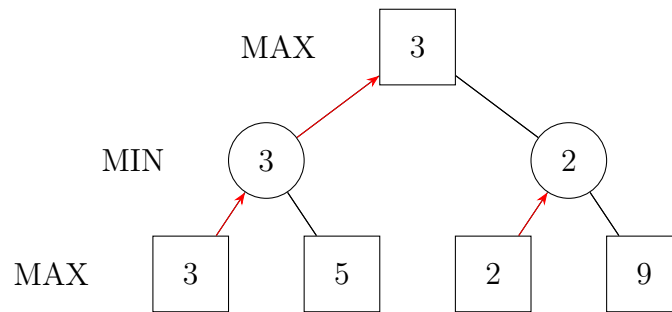


Figure 2.1: Example of minimax.

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

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

This can be answered by playing against other engines and analyzing the results. The most common way to do this is by using the Computer Chess Rating Lists which ranks chess engines based on their performance in various tournaments and matches. By the time being, we have chosen to compare different versions of the engine with Stockfish, currently ranked as the number one on the list. Continue reading to learn about the used tools or directly read about the work behind it in Section 3.6.

2.3.1. Profiler

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.

The engine must be capable of continuously processing input from standard input, even during evaluation. Moreover, if an unknown command is received, it should be ignored.

2.3.2. UCI

Universal Chess Interface specifications are independent of the operating system. To ensure the synchronization of the engine with the GUI, a `isready` command is sent and engine should respond with `readyok`.

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 commands are the following:

```
■ position [fen <fenstring>| startpos | actualpos] moves :
  <move1>...<movei>
```

sets the current position of the board to the FEN string or make the list of moves from 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.

2.3.3. CustomTkinter

CustomTkinter is a modern UI-library for Python that extends from Tkinter module and considers both OOP (Object Oriented Programming) and simple implementations.

Although UCI implements a 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.

2.3.4. Cutechess

Cutechess 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.

2.3.5. Stockfish

Stockfish is also an open-source tool and command-line program and chess engine with which we will compare with our engine. It is available for multiple platforms (Windows, macOS, Linux, Android, and iOS).

It is necessary to highlight that it provides multiple versions optimized for different hardware structures. These versions leverage specific CPU instruction sets to improve 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.

2.3.6. 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, GitHub Actions was used to automate the testing and evaluation of the chess engine using the last two mentioned tools. A workflow was configured to compile the engine, run automated games using Cutechess between different versions of the engine or our engine versus Stockfish.

Chapter 3

Engine Development

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*: in charge of sort moves by estimated quality.
- *Search*: Contains the algorithm to search the best move.
- *UCI*: Universal Chess Interface implementation.

The engine's source code is available on GitHub:

<https://github.com/LauraWangQiu/AlphaDeepChess>.

First we will describe the implementation of the basic parts of the chess engines, then we introduce and explain in detail the algorithm techniques developed to improve the chess engine playing strenght. We also created a benchmark to measure the effectiveness of each technique by playing matches with 100 games versus a baseline engine implementation.

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. [Chess Programming Wiki] (2022a)

The complete implementation can be found in the Board class file:

<https://github.com/LauraWangQiu/AlphaDeepChess/blob/main/include/board/board.hpp>.

A list of twelve bitboards is used, one for each type of chess piece. Figure 3.1 illustrates this concept with an example chess position.

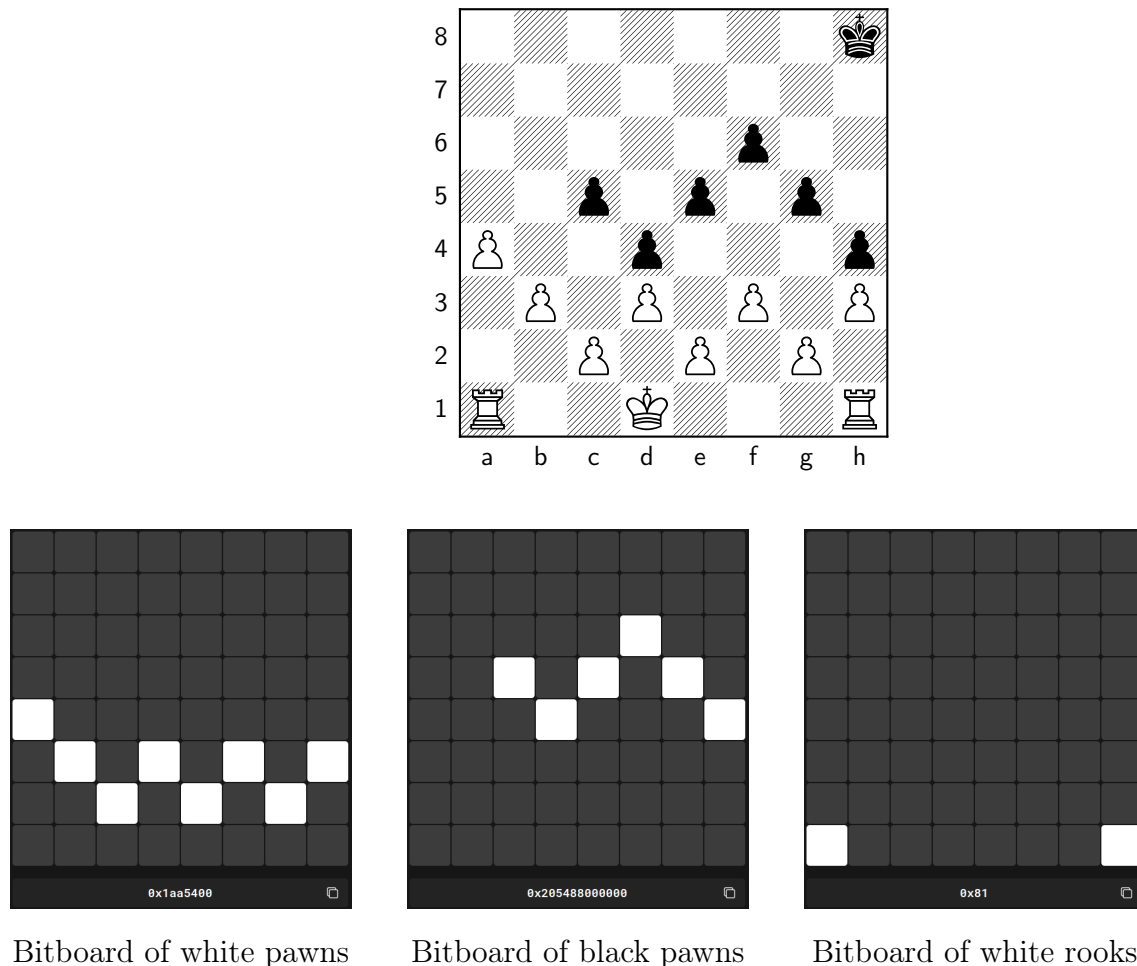


Figure 3.1: List of bitboards data structure example

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

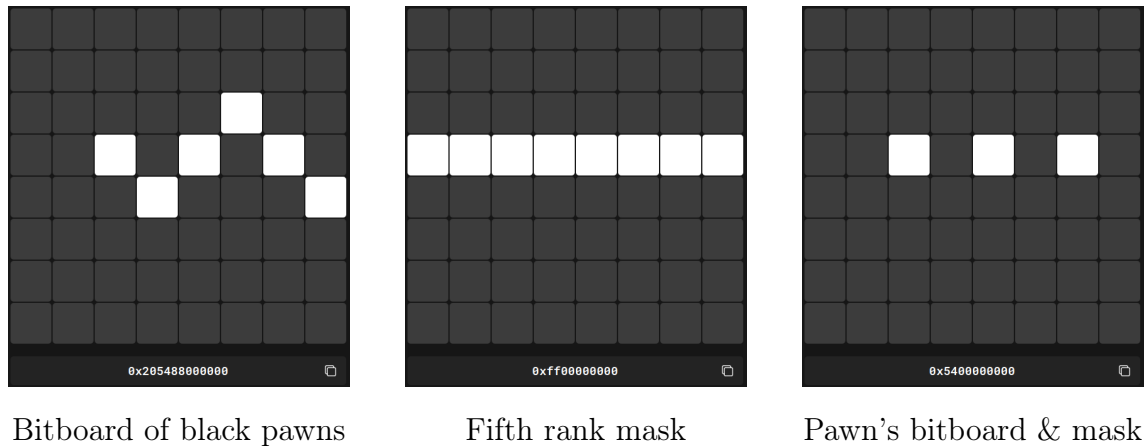


Figure 3.2: Bitboard mask operation example

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 for chess position representation, we now present the engine's core component, the search algorithm.

3.2. Search Algorithm: The Engine Core

The search algorithm implemented is minimax enhanced with alpha-beta pruning 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).

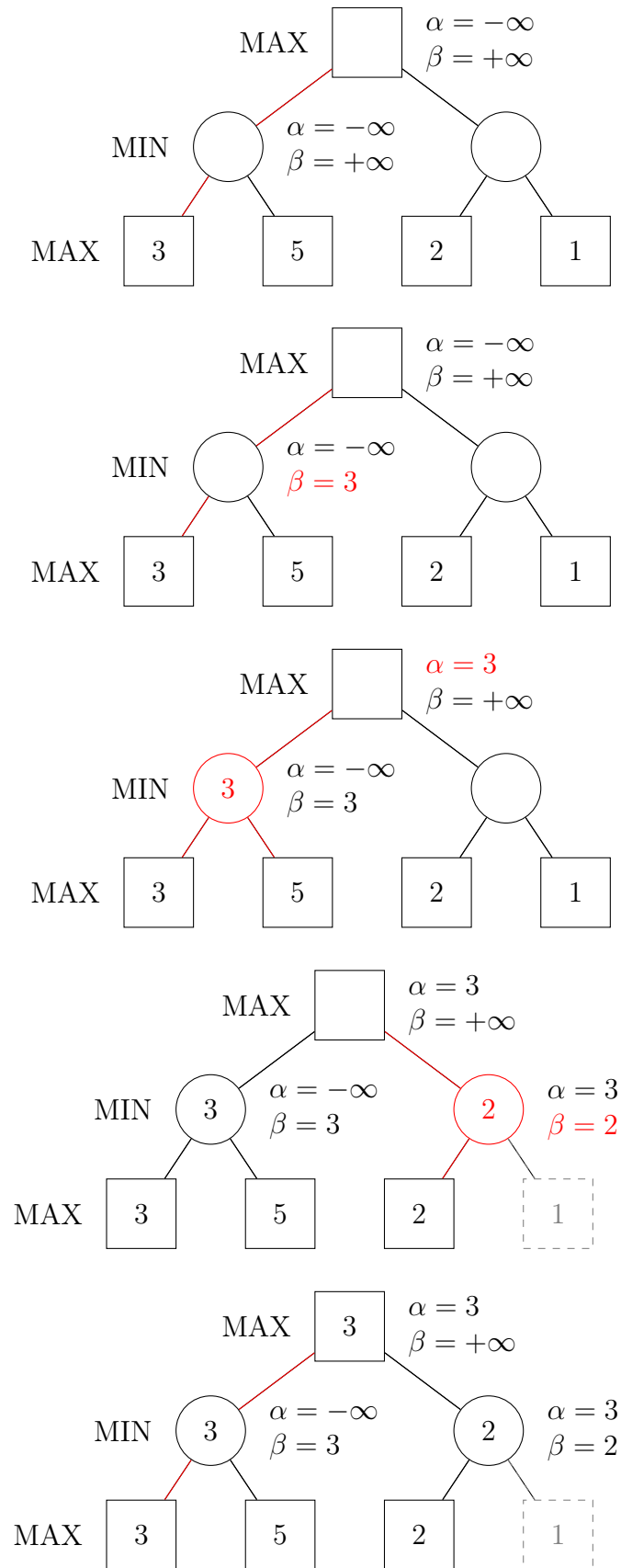
The complete implementation is available in the basic search file:

https://github.com/LauraWangQiu/AlphaDeepChess/blob/main/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 check if we can perform pruning.

Figure 3.3 demonstrates the alpha-beta search process. 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 3.3: Example of alpha-beta pruning with α and β values.

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. [Chess Programming Wiki] (2019) The engine will update the evaluation and the best move for the position in each iteration. The search can be halted at any point by issuing a *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. In subsequent sections, we will introduce techniques to tackle this inefficiency.

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. [David Eppstein] (1999b)

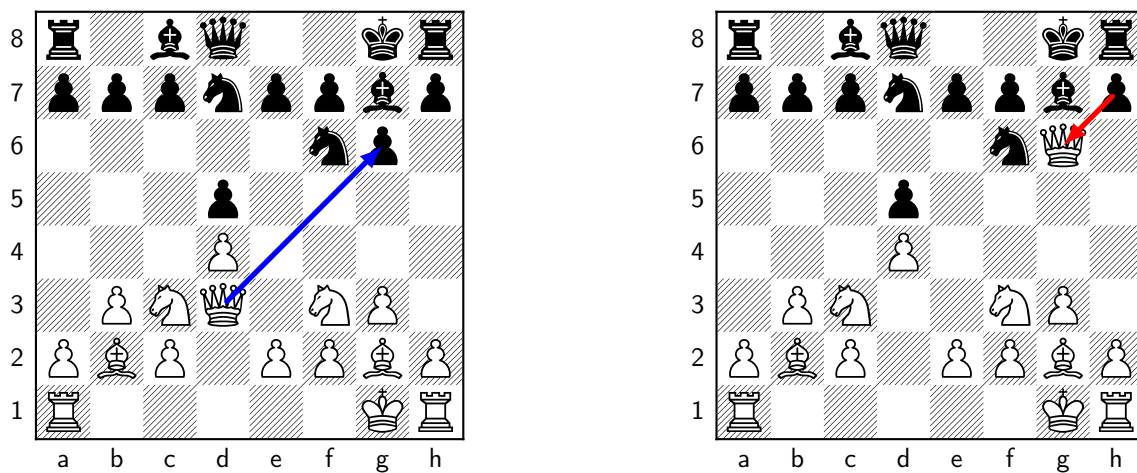


Figure 3.4: Horizon effect position example

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. [Chess Programming Wiki] (2024)

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). The sooner we explore the best move, the more branches of the tree will be pruned.
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.

Aspiration Window

Aspiration window's main objective 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 ($-\text{INF}$ and $+\text{INF}$) and a new search is performed to obtain an accurate evaluation. [David Eppstein] (1999a)

3.3. Evaluation

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. [Claude E. Shannon] (1950)

The full implementation can be found in the following source file:

https://github.com/LauraWangQiu/AlphaDeepChess/blob/main/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.

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 3.5.

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

Figure 3.5: Materialistic eval formula. Where $V(x)$ denotes the value of piece x .

Piece Square Tables (PST's)

The basic material evaluation described earlier has a significant limitation, it doesn't consider the fact that a piece could have more power in different squares of the board. For instance, as illustrated in Figure 3.6, a knight placed in the center can control up to eight squares, while a knight positioned in the corner can reach only two.

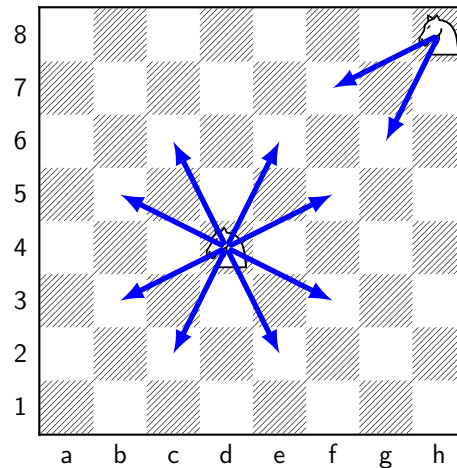


Figure 3.6: Knight's movement on corner vs in center.

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 [Chess Programming Wiki] (2022c).

An example PST for the bishop is shown in Figure 3.7, where the bishop receives a positional bonus for occupying central squares and a penalty for being placed in the edges of the board.

-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 3.7: Piece Square Table for the bishop.

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. [Chess Programming Wiki] (2021)

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 Equation 3.8.

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

Figure 3.8: Tapered evaluation formula, where α represents the proportion of middlegame.

We now need two PST's for each piece, in the following Figure 3.9 is the 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 3.9: Tapered Piece Square Tables for pawn.

3.4. Move Ordering

Having explained the evaluation function, in this chapter we detail the implementation of the move ordering heuristic.

The complete implementation can be found in the following file:

https://github.com/LauraWangQiu/AlphaDeepChess/blob/main/src/move_ordering/move_ordering_MVV_LVA.cpp.

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 [Chess Programming Wiki] (2022b).

Most Valuable Victim - Least Valuable Aggressor (MVV-LVA)

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 [Marcel Vanthoor] (2024b).

We implemented this heuristic using a look-up table indexed by the moving piece and the captured piece, as shown in Table 3.2. Capturing a queen with a pawn receives a score of 55, while doing the opposite receives 11 points.

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

Table 3.2: MVV-LVA heuristic table: Rows = Victims, Columns = Attackers.

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 branches of the tree at the same depth. [Marcel Vanthoor] (2024a)

Figure 3.10 shows an example of a killer move.

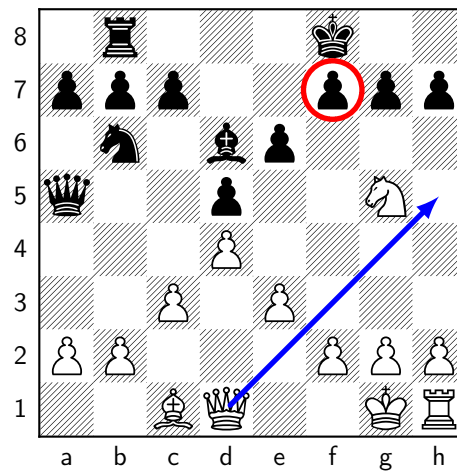


Figure 3.10: Killer move example. The queen moves to h5, threatening checkmate on f7. This quiet move prunes all other moves that do not respond to the threat.

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.

3.5. Improvements

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

At the end of each section, we present the results of the 100-game match between the engine with the implemented technique and a baseline version, which includes only the fundamental techniques described in the previous chapters. The objective of these tests are to evaluate the difference in playing strength introduced by the new implementation.

3.5.1. Transposition Table

As discussed in the previous chapter (see Section 3.2), 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 3.11 illustrates a position that can arise through multiple move orders. Where the white king could go to the g3 square from multiple paths.

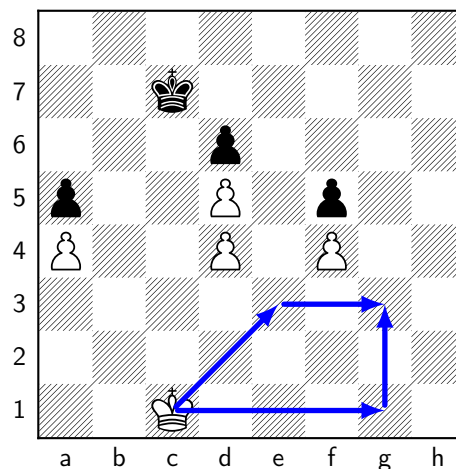


Figure 3.11: Lasker-Reichhelm Position, transposition example

Taking advantage of the concept 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. [Dennis Breuker, Jos Uiterwijk, Jaap van den Herik] (1997)

3.5.1.1. 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. ([Albert L. Zobrist] (1970))

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

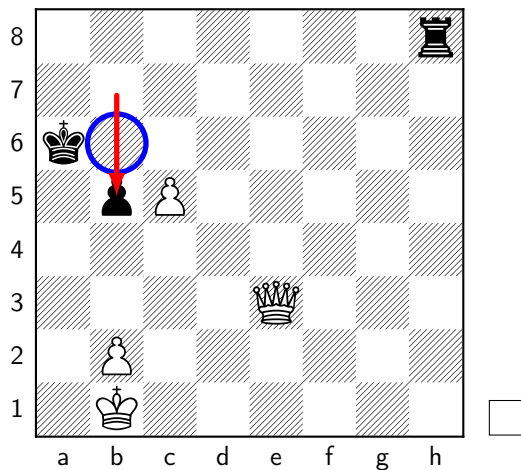
1. Pseudorandom 64-bit numbers are generated for each possible feature of a position:
 - a)* One number for each piece type on each square — 12 pieces x 64 squares = 768 numbers.
 - b)* One number to indicate the side to move is black.
 - c)* Four numbers to represent castling rights (kingside and queenside 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 3.12 illustrates an example of how the Zobrist hash is computed. The hash value is calculated by XORing the random values associated 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:

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



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

							3
12	555						
	10	67					
				909			
	69						
	111						

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

Figure 3.12: Zobrist hash calculation example

3.5.1.2. 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.
 - c) *LOWERBOUND* the evaluation is a lower bound, typically resulting from a beta cutoff.
 - d) *FAILED* entry is empty or with invalid information.

3.5.1.3. Collisions

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

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

3.5.1.4. Analysis

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



Figure 3.13: 64MB Transposition Table bot vs basic bot

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

3.5.2. Move generator with Magic Bitboards and PEXT instructions

To identify potential performance bottlenecks, we performed profiling on the engine, as shown in Figure 3.14.

Samples: 15K of event 'cycles:P', Event count (approx.): 15313528435

Overhead	Command	Shared Object	Symbol
+ 36.07%	AlphaDeepChess	AlphaDeepChess	[.] generate_legal_moves(MoveList&, Board const&, bool*, bool*)
+ 19.30%	AlphaDeepChess	AlphaDeepChess	[.] calculate_moves_in_direction(Square, Direction, MoveGeneratorInfo&) [clone .isra.0]
+ 16.63%	AlphaDeepChess	AlphaDeepChess	[.] evaluate_position(Board const&)
+ 16.23%	AlphaDeepChess	AlphaDeepChess	[.] update_danger_in_direction(Square, Direction, MoveGeneratorInfo&) [clone .isra.0]
+ 1.24%	AlphaDeepChess	AlphaDeepChess	[.] calculate_king_moves(Square, MoveGeneratorInfo&) [clone .isra.0]
0.96%	AlphaDeepChess	AlphaDeepChess	[.] quiescence_maximize_white(Board&, int, int, int)
0.81%	AlphaDeepChess	AlphaDeepChess	[.] Board::make_move(Move) [clone .isra.0]
0.74%	AlphaDeepChess	AlphaDeepChess	[.] order_moves(MoveList&, Board const&)
0.73%	AlphaDeepChess	AlphaDeepChess	[.] quiescence_minimize_black(Board&, int, int, int)
0.60%	AlphaDeepChess	AlphaDeepChess	[.] Board::put_piece(Piece, Square) [clone .isra.0]
0.59%	AlphaDeepChess	libc.so.6	[.] memset_avx2_unaligned_erms

Figure 3.14: Profiling results

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.

3.5.2.1. 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. [Pradyumna Kannan] (2007)

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

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

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

Rook $\rightarrow \rightarrow \rightarrow$ [Pawn1] [Pawn2]

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

- Compresses the blocker bitboard, pushing the important bits near the most significant bit.
- The final multiplication must produce a unique index for each possible blocker configuration. The way to ensure the uniqueness is by brute force testing.

As illustrated in Figure 3.15, we aim to compute the legal moves of the white rook in the given position. In practice, the only pieces that truly block the rook's path are those marked with a red circle.

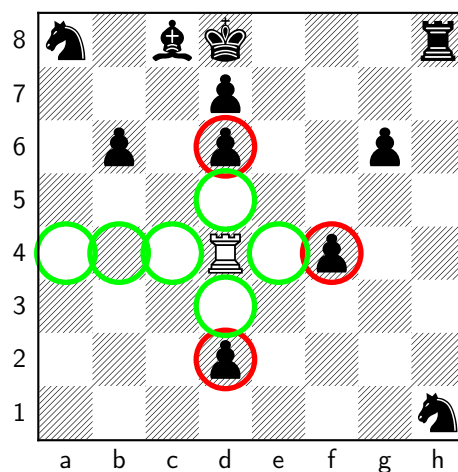


Figure 3.15: Initial chess position with white rook and blockers

First, we mask out all pieces outside the rook's attack pattern or on the board borders, as shown in Figure 3.16.

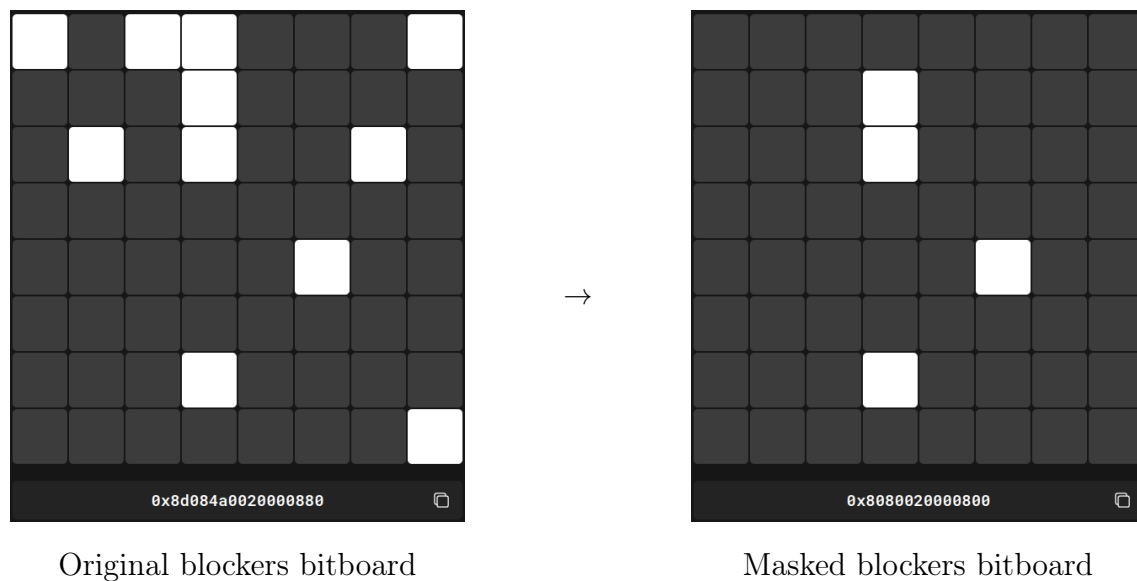


Figure 3.16: Pre-processing of the blockers bitboard

As illustrated in Figure 3.17, 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.

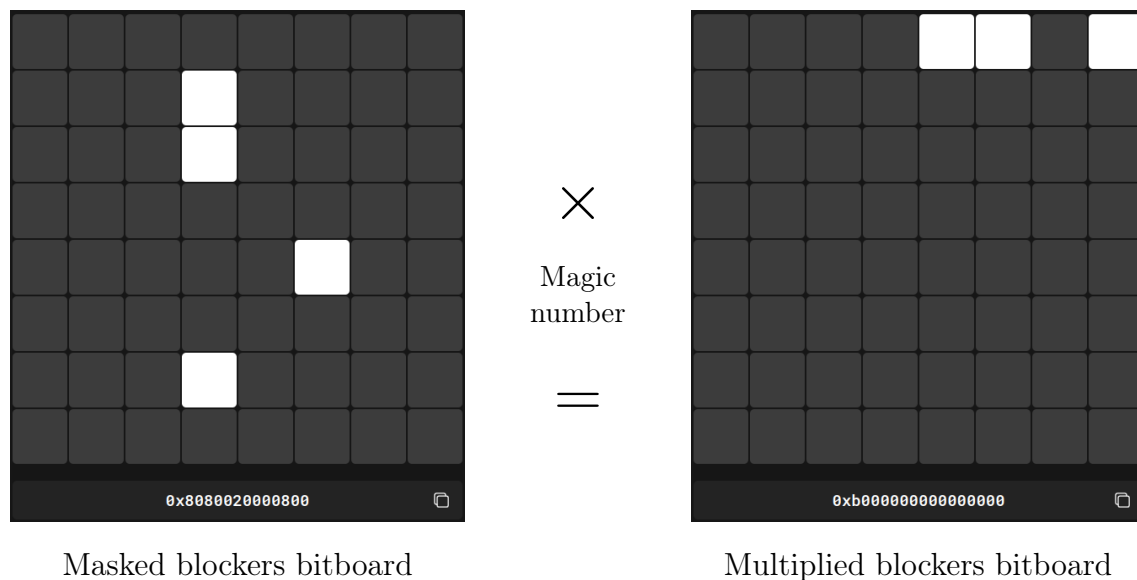


Figure 3.17: Multiplication by magic number to produce an index

Next, we compress the index toward the least significant bits by shifting right by 64−`relevant_squares`. The number of relevant squares varies per board square; Listing 3.18 shows this for the rook:

12	11	11	11	11	11	11	12
11	10	10	10	10	10	10	11
11	10	10	10	10	10	10	11
11	10	10	10	10	10	10	11
11	10	10	10	10	10	10	11
11	10	10	10	10	10	10	11
11	10	10	10	10	10	10	11
12	11	11	11	11	11	11	12

Figure 3.18: Relevant squares for rook piece.

The final index is thus computed as:

$$\text{index} = (\text{bitboard_of_blockers} \times \text{magic_number}) \gg (64 - \text{relevant_squares}).$$

3.5.2.2. 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. [Yedidya Hilewitz and Ruby B. Lee] (2006) It is ideally suited for computing our table index.

Figure 3.19 illustrates how PEXT works: it selects specific bits from register **r2**, as specified by the mask in **r3**, and packs the result into the lower bits of the destination register **r1**.

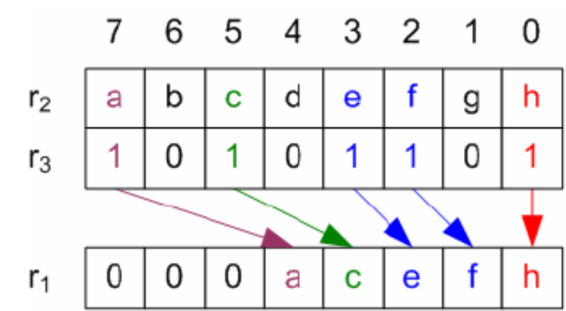


Figure 3.19: Example of the PEXT instruction: extracting bits from **r2** using **r3** as a mask, and storing the result in **r1**. [Yedidya Hilewitz and Ruby B. Lee] (2006)

For our previous example (see Figure 3.15), we only need the full bitboard of blockers and the rook’s attack pattern (excluding the borders to reduce space), as illustrated in Figure 3.20.

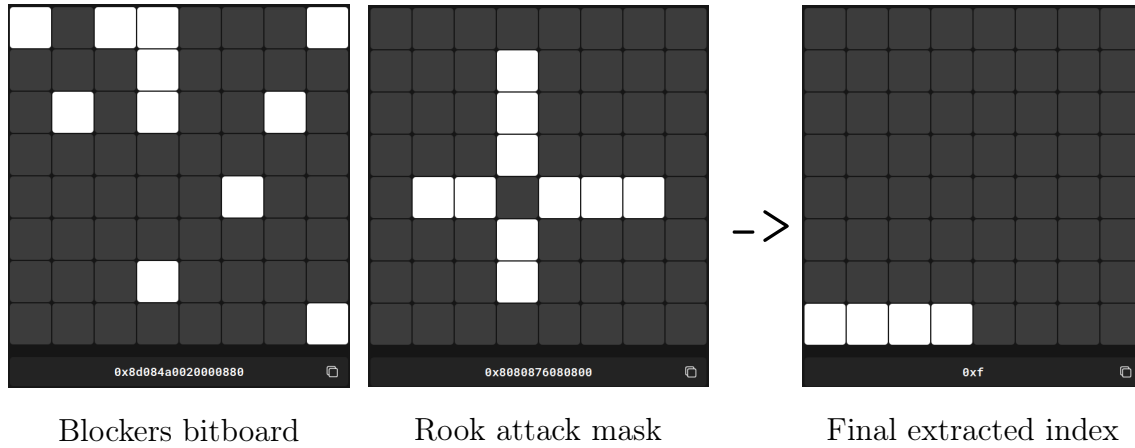


Figure 3.20: index extraction with Pext example

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

```
index = _pext_u64(blockers, attack_pattern);
```

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

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

3.5.2.3. Analysis

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



Figure 3.21: Move generator with PEXT instructions bot vs basic bot

Huge improvement with 64 wins versus 22 losses.

3.5.3. Evaluation with King Safety and piece mobility

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

1. King Shield Bonus: The king is typically safer when protected by friendly pawns in front of it. We assign a bonus in the evaluation score for each allied pawn positioned directly in front of the king.
2. King Safety Penalty: For each square within a 3×3 area surrounding the king that is attacked by enemy pieces, we apply a penalty to reflect increased vulnerability.
3. Piece Mobility: Greater piece mobility is generally indicative of a stronger position. Each piece receives a bonus for every available move to a square that is not attacked by enemy pawns.

3.5.3.1. Analysis

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



Figure 3.22: King Safety and Piece mobility evaluation bot vs basic bot

The results are slightly worse compared to the match using the material-only evaluation, 3.21 with 8 more losses than before. This may be due to the increased computational cost of evaluating these additional parameters. Furthermore, although these are abstract concepts commonly used by humans to assess positions, the engine may struggle to find a clear correlation between them and actual positional strength.

3.5.4. Search Multithread

This version of search follows YBWC mentioned in Section ???. It searches the first move sequentially after ordering the moves and launches the remaining moves in parallel using a thread. If it turns out that the first move was the best and pruning was applied, directly return the final node evaluation without the additional thread.

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

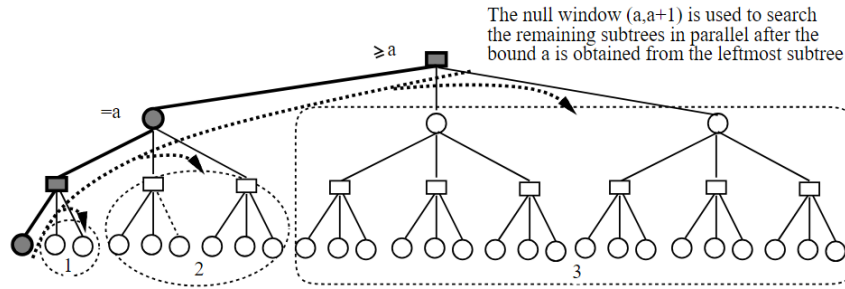


Figure 3.23: Principal variation splitting. [Yaoqing Gao and T. A. Marsland] (1996)

```
template<SearchType searchType>
static int alpha_beta_search(std::atomic<bool>& stop, int depth,
    int ply, int alpha, int beta, SearchContext& context, Board
    board)
{
    // ...
    // Check if it is in transposition table
    // If it is, return previously calculated evaluation

    generate_legal_moves<ALL_MOVES>(moves, board, &isCheck);
    // ...
    order_moves(moves, board, ply);
    // ...

    constexpr SearchType nextSearchType = MAXIMIZING_WHITE ?
        MINIMIZE_BLACK : MAXIMIZE_WHITE;

    // Search the first move sequentially
    board.make_move(moves[0]);
    int eval = alpha_beta_search<nextSearchType>(stop, depth - 1,
        ply + 1, alpha, beta, context, board);
    board.unmake_move(moves[0], game_state);
    // ...

    // Alpha-Beta pruning with first move
    if (/*pruned*/) goto end_search;
```



```

// Search remaining moves in parallel
thread = std::thread([&]() {
    for (int i = 1; i < moves.size(); i++) {
        // ...
        // Alpha-Beta pruning with the remaining moves
        if (/*pruned*/) break;
    }
});

if (thread.joinable()) {
    thread.join();
}

end_search:
// Store entry in tranposition table

return final_node_evaluation;
}

```

3.5.4.1. Analysis

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



Figure 3.24: Multithread Search bot vs basic bot

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

3.5.5. 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.

3.5.5.1. Analysis

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

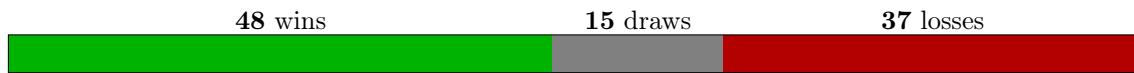


Figure 3.25: Reductions Search bot vs basic bot

The results are worse, with 15 more losses than the version without this aggressive pruning. 3.21 This could be because our move ordering is not that strong, and the best move in the position sometimes is in the last positions.

3.6. Additional tools and work

As it was mentioned in Section 2.3, some additional work apart from the engine implementation was done to ensure quality of the final product. This section includes some important specifications and features for the interactive board visualizer and testing workflow.

3.6.1. Interactive board visualizer using CustomTkinter

The board is implemented as a grid of squares, with each square capable of displaying a piece image. Piece images are loaded dynamically from the `assets` directory, which contains PNG files for each piece.

The visualizer employs an event-driven architecture, where user actions, such as clicks, trigger events handled by the `EventManager` class. When the Python script is executed, it ensures that an instance of the engine in release version executable exists, starts a new subprocess with the engine, and launches a window displaying the chessboard and interface.

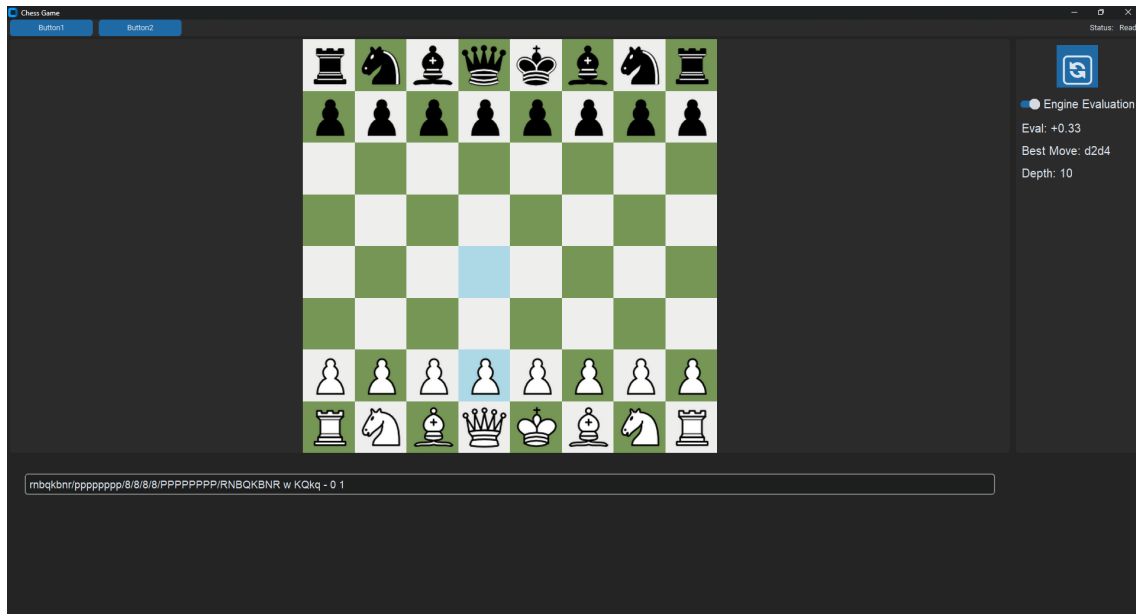


Figure 3.26: AlphaDeepChess GUI

On the right panel, there is a switch button to activate and deactivate the evaluation and suggested move in an specific depth. Also, current position's fen on the bottom panel can be modified by exchanging it to a new one.

3.6.2. Testing engine strength with Cutechess, Stockfish, and GitHub Actions

Cutechess and Stockfish can be downloaded from their respective webpages. Then, a JSON file must be provided to Cutechess to configure the available engines. An example of this `engines.json` file could be:

```
[
  {
    "name": "AlphaDeepChess",
    "command": "../build/release/AlphaDeepChess",
    "protocol": "uci",
    "options": {}
  },
  {
    "name": "Stockfish",
    "command": "./stockfish/stockfish/stockfish --windows-x86-64",
    "protocol": "uci",
    "options": {
      "UCI_LimitStrength": "true",
      "UCI_Elo": "1500"
    }
  }
]
```

]

By simply calling the Cutechess CLI with the names of the engines and additional specifications such as search time (`-st`), the number of games (`-games`), or depth (`-depth`), the matches can be executed.

Finally, automating this process of selecting the versions of the engine and Stockfish we created a workflow using a YAML file.

This workflow is triggered manually. It includes the following steps:

- Setup environment: Install dependencies such as Python packages, Cutechess and Stockfish.
- Build the engine: Compile the chess engine from source code.
- Run tests: Execute previously designed tests to validate the functionality of the engine using Cutechess CLI.
- Export generated data from Cutechess for later analysis.

This generated data are the results of the run games that includes:

- A `results.epd` with all the final positions in FEN format.
- A `results.log` with Cutechess additional detailed information like the beginning and end of a game, the summary of the game, and ELO stadistics.
- A `results.pgn` with all the played moves and games.

Conclusions and Future Work

AlphaDeepChess has proven to be an effective implementation of classical techniques focused on alpha-beta pruning. Despite the rise of neural network-based engines, this project demonstrates that well-optimized traditional approaches remain competitive.

The engine achieved an ELO score of 1900 on Lichess, running on a Raspberry Pi 5 with a 2TB transposition table, confirming its efficiency even on limited hardware. Link to the lichess engine profile: <https://lichess.org/@/AlphaDeepChess>

Incorporating techniques such as transposition tables, iterative deepening, move ordering heuristics, and magic bitboards contributed significantly to the engine's performance. The use of Cutechess testing and comparisons with Stockfish allowed this impact to be measured.

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

Personal contributions

Juan Girón Herranz

- 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. In addition to the integration 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 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.
- Research, 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.
- Research and developed the quiescence search enhancement to avoid the horizon effect in the alpha-beta pruning.
- Developed and test the search with 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.
- Creation of hundreds of unit tests, which have been a fundamental part of finding bugs and ensuring code quality. In addition to Perft testing of the move generator to ensure correctness of 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.
- Using Linux’s `perf` tool, analyze 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. This technique allows the principal variation to be explored first in a sequential way, deferring sibling node evaluations to parallel workers only after the most promising path has been examined. The principal variation refers to the first move after generated legal moves were ordered.
- 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. Bitboards were used extensively to implement both legal move generation and tactical evaluation routines, leading to a compact and performant engine state.
- 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 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 Cutchess 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

- [ALBERT L. ZOBRIST]. A new hashing method with application for game playing. The University of Wisconsin, 1970. Avail. at <https://research.cs.wisc.edu/techreports/1970/TR88.pdf> (last access, May, 2025).
- [CHESS PROGRAMMING WIKI]. Iterative deepening. Online, 2019. Avail. at https://www.chessprogramming.org/Iterative_Deepening (last access, May, 2025).
- [CHESS PROGRAMMING WIKI]. Tapered evaluation. Online, 2021. Avail. at https://www.chessprogramming.org/Tapered_Eval (last access, May, 2025).
- [CHESS PROGRAMMING WIKI]. Bitboards. Online, 2022a. Avail. at <https://www.chessprogramming.org/Bitboards> (last access, May, 2025).
- [CHESS PROGRAMMING WIKI]. Move ordering. Online, 2022b. Avail. at https://www.chessprogramming.org/Move_Ordering (last access, May, 2025).
- [CHESS PROGRAMMING WIKI]. Piece-square tables. Online, 2022c. Avail. at https://www.chessprogramming.org/Piece-Square_Tables (last access, May, 2025).
- [CHESS PROGRAMMING WIKI]. Quiescence search. Online, 2024. Avail. at https://www.chessprogramming.org/Quiescence_Search (last access, May, 2025).
- [CLAUDE E. SHANNON]. Programming a computer for playing chess. Computer History Museum Archive, 1950. Avail. at https://archive.computerhistory.org/projects/chess/related_materials/text/2-0%20and%202-1.Programming_a_computer_for_playing_chess.shannon/2-0%20and%202-1.Programming_a_computer_for_playing_chess.shannon.062303002.pdf (last access, November, 2024).
- [DAVID EPPSTEIN]. 1999 variants of alpha-beta search. UC Irvine, 1999a. Avail. at <https://ics.uci.edu/~eppstein/180a/990202b.html> (last access, May, 2025).
- [DAVID EPPSTEIN]. Which nodes to search? full-width vs. selective search. UC Irvine, 1999b. Avail. at <https://ics.uci.edu/~eppstein/180a/990204.html> (last access, May, 2025).

- [DENNIS BREUKER, JOS UITERWIJK, JAAP VAN DEN HERIK]. Information in transposition tables. University of Limburg, 1997. Avail. at https://www.researchgate.net/publication/2755801_Information_in_Transposition_Tables (last access, May, 2025).
- [FÉDÉRATION INTERNATIONALE DES ÉCHECS]. Fide laws of chess. Online, 2023. Avail. at <https://handbook.fide.com/chapter/E012023> (last access, May, 2025).
- [MARCEL VANTHOOR]. Killer move heuristic. Online, 2024a. Avail. at <https://rustic-chess.org/search/ordering/killers.html> (last access, May, 2025).
- [MARCEL VANTHOOR]. Mvv-lva. Online, 2024b. Avail. at https://rustic-chess.org/search/ordering/mvv_lva.html (last access, May, 2025).
- [PRADYUMNA KANNAN]. Magic move-bitboard generation in computer chess. Online, 2007. Avail. at http://pradu.us/old/Nov27_2008/Buzz/research/magic/Bitboards.pdf (last access, May, 2025).
- [STEFAN-MEYER KAHLEN]. Description of the universal chess interface (uci). Online, 2004. Avail. at <https://www.wbec-ridderkerk.nl/html/UCIProtocol.html> (last access, May, 2025).
- [YAOQING GAO AND T. A. MARSLAND]. Multithreaded pruned tree search in distributed systems. University of Alberta, 1996. Avail. at <https://webdocs.cs.ualberta.ca/~tony/RecentPapers/icci.pdf> (last access, March, 2025).
- [YEDIDYA HILEWITZ AND RUBY B. LEE]. Fast bit compression and expansion with parallel extract and parallel deposit instructions. Princeton University, 2006. Avail. at <http://palms.ee.princeton.edu/PALMSopen/hilewitz06FastBitCompression.pdf> (last access, May, 2025).