BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

EXPLANATORY ANALYSIS OF THE CHESS GAME

VYSVĚTLUJÍCÍ ANALÝZA ŠACHOVÝCH PARTIÍ

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **Bc. VOJTĚCH HERTL**
AUTOR PRÁCE

SUPERVISOR **Ing. BOHUSLAV KŘENA, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2022

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav inteligentních systémů (UITS) Akademický rok 2021/2022

Zadání diplomové práce

Student: **Hertl Vojtěch, Bc.**
Program: Informační technologie a umělá inteligence
Specializace: Inteligentní systémy
Název: **Vysvětlující analýza šachových partií**
Explanatory Analysis of the Chess Game
Kategorie: Umělá inteligence

Zadání:

1. Prostudujte metody hraní hry šachy počítačem.
2. Porovnejte šachové programy dostupné pod open source licencemi jako například Stockfish, Igel nebo RubiChess.
3. Navrhněte rozšíření vybraného šachového programu, aby v něm využívaná analýza poskytovala vysvětlení kvality jednotlivých tahů šachové partie, a to včetně vhodné reprezentace výsledků.
4. Navržené rozšíření implementujte a otestujte.
5. Zhodnoťte dosažené výsledky a navrhněte možná vylepšení.

Literatura:

- Russel, S.J., Norvig, P. *Artificial Intelligence: A Modern Approach*. 2nd ed. New Jersey: Prentice Hall, 2003. 1081 s. ISBN 0-13-790395-2.
- Levy, D.N.L., Newborn, M. *How Computers Play Chess*. New York: Ishi Press, 2009. 260 s. ISBN 9784871878012.

Při obhajobě semestrální části projektu je požadováno:

- První tři body zadání.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Křena Bohuslav, Ing., Ph.D.**
Vedoucí ústavu: Hanáček Petr, doc. Dr. Ing.
Datum zadání: 1. listopadu 2021
Datum odevzdání: 18. května 2022
Datum schválení: 3. listopadu 2021

Abstract

The aim of this thesis is to design and implement an explanatory analysis of chess games. This analysis was created on the basis of a chess engine. As there are many great, freely available and open-source chess engines, this thesis does not focus on the implementation of a new one. Instead, existing chess engines were studied and compared. The most suitable one for this thesis was selected and an extension was designed for it. This extension provides an explanatory explanation of the quality of individual moves and entire chess games. This extension has been implemented, tested and evaluated.

Abstrakt

Cílem této práce je navrhnout a implementovat vysvětlující analýzu šachových partií. Tato analýza byla vytvořena na základě šachového programu (chess engine). Jelikož existuje mnoho kvalitních, volně dostupných a open-source šachových programů, tato práce se nesoustředí na implementaci nového programu. Namísto toho, již existující šachové programy byly prostudovány a porovnány. Nejvhodnější program pro tuto práci byl vybrán a pro něj bylo navrhnuto rozšíření, které poskytuje slovní vysvětlení kvality jednotlivých tahů a celých šachových partií. Toto rozšíření bylo implementováno, otestováno a ohodnoceno.

Keywords

Chess, chess engine, explanatory analysis, Stockfish

Klíčová slova

Šachy, šachový program, vysvětlující analýza, Stockfish

Reference

HERTL, Vojtěch. *Explanatory Analysis of the Chess Game.* Brno, 2022. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Bohuslav Křena, Ph.D.

Rozšířený abstrakt

Šachy jsou celosvětově jednou z nejpopulárnějších deskových her. Díky svým vlastnostem je tato hra zajímavá pro studium a výzkum. Mnoho let byly šachy studovány a rozliční lidí tuto hru hráli a mnohé turnaje byly uspořádány, aby se zjistilo, kdo je dobrý hráč.

V 60. letech minulého století se začaly objevovat šachové programy. Šachové programy jsou počítačové programy, které jsou schopné hrát šachy. V nedávné historii se vzestupem počítačů začaly být tyto programy velice silné a začaly běžně porážet lidské hráče. Od té doby nikdy neskončil další vývoj šachových programů a v dnešní době nemá i nejlepší šachový velmistr šanci uspět v partii s nimi. Existuje mnoho různých implementací šachových programů, které se liší zejména v prohledávání nebo v evaluační funkci. Pro evaluaci šachových pozic některé používají nejmodernější metody umělé inteligence a jiné zase ručně psanou funkci, která byla vytvořena s pomocí šachových mistrů.

Šachy jsou tvořeny taktikou a strategií. Taktiky jsou propočítány několik tahů dopředu a je analyzováno, zda některá sekvence vedou k výhodě či nevýhodě jednoho hráče. Strategie jsou zase propočty šachových pozic z dlouhodobého hlediska. Zejména v taktikách jsou počítače o mnoho lepší než lidé. Člověk se může naučit spoustu šachové teorie z různých knih nebo kurzů. Mimo to se může člověk zlepšovat, pokud hraje velké množství her. Nicméně momentálně neexistují dobré způsoby, jak by hry mohly být analyzovány a aby se zjistilo, z jakého důvodu prohrané hry dopadly špatně. Analýza, která by dokázala vysvětlit jednotlivé tahy a pomohla hráčům přijít na jejich nedostatky, by mohla sloužit jako dobrý výukový nástroj. Z tohoto důvodu se tato práce zabývá vysvětlující analýzou šachových partií.

Předtím, než se začne vyvíjet vysvětlující analýza, musí být vytvořen šachový program. Implementace takového šachového programu, který by byl alespoň tak dobrý, aby dokázal porazit průměrného hráče, by byla zbytečně náročná a zdlouhavá. Existuje velké mnoho silných šachových programů, které jsou dostupné jako open source. Pro tuto práci je dostačující, aby jeden z těchto programů byl vybrán a zmíněná analýza byla implementována jako rozšíření.

Ze začátku byly prohledány již existující programy, které se snaží dojít k podobnému cíli. Některé podobné programy existují, ale jejich funkcionalita je diskutabilní. Tato práce se vydává originálním směrem.
V této práci jsou nejprve popsány šachové programy a jejich funkcionalita. Tento teoretický popis se snaží zahrnout nejdůležitější metody a přístupy k implementaci šachových programů. Funkcionalita těchto programů se skládá zejména z reprezentace šachovnice, vyhledávací funkce, evaluační funkce a nakonec využití různých databází pro vylepšení náročných fází hry.

Dále jsou existující šachové programy důkladně porovnány na základě předem zvolených kritérií. Tato kritéria jsou zvolena po prezentování hlavních myšlenek k analýze. Z původních 1300 šachových programů, které jsou známy v komunitě šachových programátorů, bylo po postupném filtrování vybráno 10. Těchto 10 programů je důkladně porovnáno z různých hledisek.

Po výsledcích z porovnání je vybrán takový šachový program, který je určen jako nejvhodnější k rozšíření vysvětlující analýzou. Tímto programem byl zvolen nejpopulárnější Stockfish. Tento program je velmi silný, často aktualizovaný a jeho kód je kvalitně psaný. Program Stockfish byl prostudován a je popsána jeho funkcionalita a struktura kódu.

Následně byl vytvořen návrh vysvětlující analýzy a společně s její implementací je podrobně popsán. Návrh je strukturován do logických částí. Nejprve je vysvětleno vytváření analýzy tahů a pozic a poté je popsáno samotné vysvětlování. Vysvětlení spočívá v porovnání tahu uživatele a pokračující nejlepší sekvence tahů s úplně nejlepším tahem a pokračující sekvencí, kterou navrhuje šachový program. V obou těchto sekvencích je nalezena nejlepší stabilní pozice. Dále je zjištěn faktor z největším dopadem na rozdílné hodnocení těchto pozic. Hodnocení tohoto faktoru je poté důkladně popsáno a vysvětleno v obou pozicích.

Nakonec je popsáno několik experimentů, které byly provedeny v rámci testování. Testování proběhlo ručně, jelikož vysvětlující analýzu není možné hodnotit strojově. Na základě těchto testů a experimentů je rozšíření vyhodnoceno a některé možnosti pro vylepšení jsou navrhnuty.

Tato práce je zejména určena pro lidi, kteří mají alespoň základní znalosti šachu, jako jsou například základní pravidla a tahy figur. Tyto základy se tedy neobjevují v textu práce. Autorem této práce je mírně pokročilý šachový hráč a nejlepší čtenář je člověk na podobné úrovni v hraní šachů.

Explanatory Analysis of the Chess Game

Declaration

I hereby declare that this Master's thesis was prepared as an original work by the author under the supervision of Ing. Bohuslav Křena, Ph.D. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Vojtěch Hertl
May 16, 2022

Acknowledgements

First and foremost, I would like to thank my supervisor Bohuslav Křena for his assistance during this thesis. Thanks for his observations and suggestions as well as for the support he has given. Then, I would like to thank my family and girlfriend for all kinds of support. Last but not least, thanks also belong to the fellow colleagues that have made the whole study feasible.

Contents

Chapter 1

Introduction

Chess is one of the most popular board games worldwide. The features of this game make it great for study and research. For many years, chess has been studied and played by various people and many tournaments have been played to distinguish their skill and qualities.

In the 1960s, chess engines began to appear. Chess engines are computer programs that analyse chess positions and generate moves that it regards as strongest. In recent history, the rise of computers and online chess started. The playing strength of the chess engines increased and for the first time in 1997, a computer beat the human world champion in chess. Since then, the research and the continuous improvements of chess engines have never stopped. Currently, even the best human chess players are no match for computers. Many various chess engines with different implementations have been programmed. The implementations include the most sophisticated searching and evaluating methods. Some use the most modern artificial intelligence and machine learning methods, some rely on handcrafted functions.

Chess is made up of tactics and strategy. For tactics, moves and replies are calculated and analysed if some sequence of moves leads to an advantage or disadvantage. Strategy means analysing a position from a long-term point of view. Primarily in tactics, a computer is superior to humans. A human can learn chess theories from various chess books and courses. Another great way of learning chess is to play many games. However, there is currently no great way to analyse when and why the lost games went wrong. An analysis that could explain the moves and give players a better idea of their strengths and weaknesses would be a great learning tool. Therefore, an explanatory analysis of the game of chess could help human players improve their chess skills.

Before developing an explanatory chess analysis tool, a chess engine program must be created. Implementing such an engine that could play the game better than a human would be unnecessarily time-consuming and redundant. There is a great amount of strong chess engines already implemented and many of them are available as open-source. For this thesis, it is sufficient to select one of those and extend it with the analysis.

In this thesis, chess engines, and their functionality are described in detail including the most modern approaches. Further, several interesting engines are compared based on criteria chosen beforehand. Then, the best fitting chess engine is selected and described in detail and an extension to this engine is designed. The extension aims to provide explanatory

analyses of chess moves and whole games. Then, the design and implementation details of the extension are described. Finally, the extension is tested by multiple experiments, the final evaluation is given and some possible future improvements are proposed.

This thesis is primarily designed for people familiar with chess basics, such as moves and rules, the explanation of which is skipped. The author is an intermediate chess player and therefore the perfect recipient for this thesis is someone with similar game knowledge and skill but both better and worse players can find it beneficial.

Related Work

It is already possible to analyse games using bare chess engines. Usually, by loading a single position inside the chess program using either command line or graphical interface. A typical chess program provides some type of feedback:

- A single evaluation number which represents, how both players stand in the game.
- The best possible move (or multiple moves) in that position and the following sequence of best moves for each player.

This feedback is useful but very minimal. To analyse the whole game, every position would have to be loaded and analysed individually. This problem is already solved by many GUI programs which help with the analysis and provide the functionality to load the whole game using a specific chess game notation. These programs usually provide the option to select a chess engine with which they communicate via a protocol and can analyse the whole game and present results graphically.

Still, the main problem persists. The analysis is not good enough for an intermediate player. They are good on some occasions:

- Analysis by a chess master of his opponent games during a preparation for a tournament.
- Detection of very bad moves by checking the evaluation number before and after the move.
- The best possible move information can be sometimes great for an intermediate player to see the idea behind it.

These analyses lack the following concepts:

- Explanation of the “not so great” moves. These are the moves, that could have a big impact on the course of the game.
- Explanation of the whole suggested sequence after the following move.
- The analysis of a complete game. The accuracy of the moves overall, the strong moves in each of the phases of the game. Missed winning moves.
- Plans, tactics, and more concepts in a current position.

These concepts, or at least some, are exactly the ones that this thesis should solve. There are a few commercial programs that try to achieve the same goals as this thesis but none of them seem to be perfect. At the time of writing this thesis, the explanatory analysis seems to be a trend in the chess world. In the next paragraphs, two of these programs are introduced:

Chess.com Game Review Chess.com is a chess website where many people play online chess. Using the great number of games from its users, they started a project of reviewing games called Game Review [3] during the writing of this thesis. It aims to provide a better understanding of the games a user played. It implements a virtual coach that will explain interesting moments in the games and evaluates the moves. It gives analyses for both each move and the whole match. The platform itself is free but the explanatory analyses require a paid subscription. It has been tried by the author of this thesis. On one hand, it provided some inspiration but on the other hand, it confirmed, that the analyses are not great enough.

Decodechess Decodechess [22] is a web program that generates explanations that are similar to those of a human chess master. The authors use Stockfish as the engine and build on it. It provides explanations for positions and the whole game. Decodechess also requires a purchase after a few analyses. Decodechess is a new program and still has many imperfections. It will serve as an inspiration in this thesis. By feel, this application could be very good but the decodings take a very long time and the explanatory analysis does not seem to be the most straightforward. Some parts are unnecessary, some are lacking more explanation.

Structure of the Thesis

Chapter 2 introduces the methods of playing the game of chess by a computer The most important parts of the functionality of chess engines are described in detail. Primarily the representation of the chess board, the search function, the possibilities of evaluation methods, and the databases for opening and endgame phases are explored. Next in Chapter 3, a comparison of modern open-source chess engines is performed and a list of the most suitable engines is created. The comparison is based on criteria chosen beforehand. Then, the final choice of the most fitting chess engine is made in Chapter 4. The functionality and the code structure of the chosen engine are described in detail to be able to extend it. Using the chosen chess engine, an extension for explaining chess moves and games is designed and implemented and the process is described in Chapter 5. Various experiments are illustrated in Chapter 6. The extension is evaluated based on the performed experiments. This thesis is concluded in Chapter 7. In the conclusion, several problems are mentioned and their solution as future improvements are outlined.

Chapter 2

Chess Engine

In this chapter, computer programs, which can play the game of chess, are explained in detail. These computer programs are called chess engines. Nowadays, chess engines are capable of easily beating the best human chess players and as new improvements are invented, their playing strength is increasing. Most of the best chess players, known as Masters (the very best are Grandmasters), use engines during the preparation for their tournaments. Chess is originally a board game but has become a widely studied game from the viewpoints of computer science and artificial intelligence. According to the definitions in [25], chess is a game with **fully observable**, competitive **multiagent**, **deterministic**, **sequential**, **semi dynamic** (if played with a clock) and **discrete** environment. These characteristics make the game study-friendly and interesting for research. The fact that the game is competitive and multiagent, and many matches have been played makes it great for improving the engines. Chess could also be defined as a zero-sum game with perfect information.

Fully observable An environment is fully observable if an agent's sensors give access to the entire state of the environment at each point in time. For chess, both players have perfect information about the position of every piece on the board.

Multiagent Multiagent environment is the one that contains multiple agents. For chess, the agents are two – one for white and one for black. Also, the chess environment is competitive, meaning, that both agents try to maximize their gains which reduces the opponent's gains.

Deterministic This means, that the next state of the environment is determined by the current state and the action executed by the agent. In chess, after every move in every position, there is always a single next position.

Sequential Sequential means, that the performed action affects the future states. If a move is performed in chess, it has long-term consequences and therefore, the agent needs to think ahead.

Semi dynamic If the environment can change when the agent is deliberating, it is dynamic. Otherwise, it is static. Semi dynamic environment is in the middle, as it does not

change with the time but the agent's performance does. Chess is semi-dynamic, if played with a clock. If one side has less time, its performance decreases.

Discrete A discrete environment has a finite number of distinct states and a discrete set of percepts and actions.

Firstly, the programming of the chess engines is described. Although the programming has had a long evolution phase, in the past years, many standards have been set. Therefore, the description is about the functionality of a typical modern chess program. Generally, a chess engine consists of multiple parts. **Board representation** defines how the chess board, pieces, and more information are stored in the program. **Search** function does the search of a tree data structure of chess positions. In the **evaluation** part, chess positions are analysed and evaluated based either on human-defined fitness function or by artificial intelligence methods.

2.1 Representation

First, it is necessary to define how the state of the chess board will be represented. Not only the positions of chess pieces must be remembered, but also the castling rights, en passant square, the half move clock, and all the previous positions of the game. The half-move clock and previous positions because of the draw possibilities by chess rules in [6]. Nowadays, there are two main modern approaches – *arrays* and *bitboards*.

Second, the chess positions and the whole game must be deterministically represented for many various reasons. For example, to quickly manipulate data or to load and analyse a certain position by an engine. There are standard notations used to represent positions and games outside of the program.

Arrays

The simplest representation of the chess board in terms of programming is achieved using arrays. [11] Arrays were used in the first computer programs but also some recent ones have arrays in their implementation. The approaches may vary, one may use two-dimensional arrays, another one uses one-dimensional ones. The two-dimensional approach is more natural for a human to understand as it maps each square to the elements of the arrays one-to-one. However, at those times when primarily arrays were used, multiplication was considered an expansive operation on the hardware. By indexing and computing with a matrix, multiplication it is often inevitable. That is the main reason why the one-dimensional arrays are much more common.

One-dimensional arrays The first performance gain over the two-dimensional arrays is that the multiplication is eliminated and replaced with addition by a constant. For example, the squares can be indexed in an array within the program in the **8x8** representation in the following way:

- `array[0]` – **a1** square
- `array[7]` – **h1** square

- `array[56]` – **a8** square
- `array[63]` – **h8** square

Using this representation, it is simple to navigate around the chess board by adding or subtracting a constant, e.g., adding 9 moves the index diagonally to the upper right square and by adding 9 multiple times, the whole diagonal can be covered. In the previously mentioned approach, it would be necessary to use multiplication to reach the same goal. Nevertheless, there is still an inconvenience – testing of edges of the board. This reduces a lot of the performance gain. Often, border squares are used as the solution. Instead of the 8x8 board representation, **10x12** is used as illustrated in Figure 2.1. The two additional rows at the bottom and top are needed for the knight jumps, as they can move two squares. At the sides, one column is enough because while indexing, columns 1 and 10 are adjacent. The efficiency trick of the border squares is that if validating a move, only a simple check of numbers is required.

7	7	7	7	7	7	7	7	7	7
7	7	7	7	7	7	7	7	7	7
7	-4	-2	-3	-5	-6	-3	-2	-4	7
7	-1	-1	-1	-1	-1	-1	-1	-1	7
7	0	0	0	0	0	0	0	0	7
7	0	0	0	0	0	0	0	0	7
7	0	0	0	0	0	0	0	0	7
7	0	0	0	0	0	0	0	0	7
7	1	1	1	1	1	1	1	1	7
7	4	2	3	5	6	3	2	4	7
7	7	7	7	7	7	7	7	7	7
7	7	7	7	7	7	7	7	7	7

Figure 2.1: Representation of pieces, empty squares and border squares on the one-dimensional array at starting position. 0 – empty squares, 1-6 – squares with pieces, 7 – border squares. Positive and negative numbers for white and black pieces, respectively.

The **0x88** is the last widely used representation using a one-dimensional array. This method uses an array of 128 elements. In Figure 2.2, the left half represents the valid squares on the chess board while the right half is illegal squares. The important feature of this method is that 4 bits are used for a file and 4 bits for a rank. For the valid squares, the leftmost bit (highest value) of the 4bit value is always 0. In the computations, a mask with value $0x88$ (in hexadecimal) is used with any square index in `AND` operation and if the result is a non-zero value, the square is illegal and vice versa. For example, **c3** square is represented by the number `0x23`. If used in `AND` operation with `0x88`, the result is `0`, therefore the square is legal.

Bitboards

The earliest documentation about bitboards is in a journal about a bitboard-based chess program called DarkThought [9]. In the first versions, DarkThought did not use the bitboard representation but the authors had taken inspiration in at that time very fast and

```
70 71 72 73 74 75 76 77 | 78 79 7a 7b 7c 7d 7e 7f
60 61 62 63 64 65 66 67 | 68 69 6a 6b 6c 6d 6e 6f
50 51 52 53 54 55 56 57 | 58 59 5a 5b 5c 5d 5e 5f
40 41 42 43 44 45 46 47 | 48 49 4a 4b 4c 4d 4e 4f
30 31 32 33 34 35 36 37 | 38 39 3a 3b 3c 3d 3e 3f
20 21 22 23 24 25 26 27 | 28 29 2a 2b 2c 2d 2e 2f
10 11 12 13 14 15 16 17 | 18 19 1a 1b 1c 1d 1e 1f
 0  1  2  3  4  5  6  7 |  8  9  a  b  c  d  e  f
```

Figure 2.2: Representation by arrays using the 0x88 method displayed in hexadecimal base.

strong programs such as Cray Blitz [32]. This board representation style was convenient due to the triviality of bitwise operations and therefore a significant speedup in calculations. Also, the implementation was convenient with the increased growth of 64-bit architecture computers.

A bitboard [8] is essentially a finite set of up to 64 elements. All the squares of a chessboard fit into one bitboard, so each bit represents one square. The value of each bit indicates the presence or absence of some state on the corresponding square, such as if the square is attacked. A bitboard is usually stored in a 64-bit integer variable.

For piece representation [11], the simplest method is to use 12 bitboards for representation of the location of each different piece – king, queen, rook, bishop, knight, and pawn for each side. Also, it is useful to have two more bitboards for squares occupied by all the pieces for each colour. The biggest advantage of using bitboards is the speed and possible parallelism of bitwise operations between them. Bitboards can be used for example in the evaluation function to efficiently detect which piece attacks or defends another piece. An example of bitwise operations from position in Figure 2.3 is shown in Figure 2.4 . Usually, more bitboards are created and used for more complicated computations such as Rotated or Magical bitboards and different techniques with them have been invented.

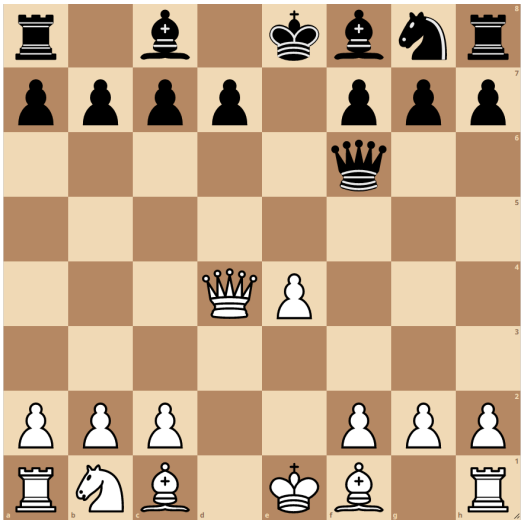

Figure 2.3: A chess position for demonstrating the bitwise operation efficiency.

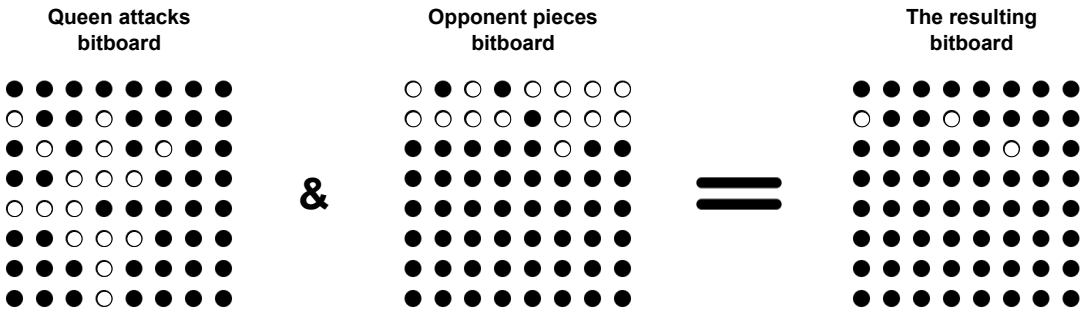

Figure 2.4: Demonstration of bitwise AND operation using bitboards representation from position in Figure 2.3. Showcasing calculation of a bitboard that contains all squares attacked by a white queen. The white and black circles are representing a bit value of 1 and 0, respectively.

Chess Position

It is also important to be able to describe how pieces are placed on the board outside of the program. There are a couple of notations exactly for this purpose [5]. They are used to quickly load a position without having to replay the whole game. To balance the simplicity, this type of representation is not complete and only consists of the essentials. It must contain information about piece placement, side to move, castling rights, en passant square, and half-move clock which covers the fifty-move rule and can skip information about the history for threefold repetition due to its volume. The most common notation is *Forsyth-Edwards Notation* and is used in this thesis. Another one worth mentioning is *Extended Position Description* which is more complex and expandable by new operations.

Forsyth-Edwards Notation Forsyth-Edwards Notation [5] or FEN is a standard for describing chess positions using ASCII. One record uses one line which has six data fields separated by a single space character. The data fields are the following:

1. **Piece placement data**

   The first field represents the placement of the pieces on the board. The content is represented starting from the *eight* rank down to rank *one*. Each rank is specified from file *a* to file *h*. The description of pieces is by corresponding uppercase and lowercase letters ("PNBRQK" and "pnbrqk") for white and black pieces, respectively. Blank squares are represented by a single digit where the digit is the number of consequent empty squares. To separate ranks, a slash ("/") symbol is used.

2. **Active colour** – The second field represents the colour which is on the move. Lowercase letter "w" or "b" is used to represent if its is white's or black's turn, respectively.

3. **Castling availability**

   The third field represents castling rights. According to the development of the game and current position, it might or might not be possible for each side to perform the castling move. The complete impossibility for both sides is represented by a dash ("-"). Then, for the availability, uppercase and lowercase letters are used for white and black pieces, respectively. The letters are "k" for kingside castling and "q" for queenside castling. They are ordered first by caseness (upper first) and second by the side (king first).

4. **En passant target square**

   The fourth field is for the en passant. If no en passant capture is available, the dash ("-") symbol is used. If there is one, coordinates of file and rank are used.

5. **Half-move number** – The fifth field is there for counting half-moves since the last pawn advance or capturing move. It is represented by a positive integer.

6. **Fullmove number** – The last field contains a positive integer representing the number of moves in a game. It increments after black's move.

Here is an example of a position in FEN after white's first move `1.e4`:

```
rnbqkbnr/pppppppp/8/8/4P3/8/PPPP1PPP/RNBQKBNR b KQkq e3 0 1
```

Move

Apart from the Standard Algebraic Notation (SAN) used by humans, the moves are often represented in Long Algebraic Notation (LAN). It is either 4 or 5 characters long. The first two characters specify the square of the piece that should be moved. The second two characters are for the square where to move the piece. The 5th character is only for pawn moves that promote and represent the promotion piece type.

SAN examples: `e4` (favourite opening move for white, pawn on e2 moves to e4), `0-0` (white short castling), `e8=Q` (promotion to a queen), `Nbxc4` (a knight on b file moves to square c4 while capturing a piece, multiple knights can move to square c4 and the knight on b file is specified).

LAN examples: `e2e4` (pawn on e2 moves to e4), `e1g1` (white short castling), `e7e8q` (promotion to a queen), `b2c4` (the same knight move as from SAN example).

Game

For the analysis of chess via computers, there must be some databases of completed games. The big amount of data is useful for many different studies such as machine learning or knowledge discovery from databases. Also, the representation can be stored and loaded into chess programs with the whole game history. Moreover, it is possible to add notes to moves with explanations or evaluations. To represent and store a whole game, there is one most widely used notation called Portable Game Notation (PGN).

Portable Game Notation Portable Game Notation [5] or PGN is a standard for describing chess game data using ASCII. It is structured to be both easily graspable by humans and for easy manipulation by computer programs. The purpose is to share game data among chess players, publishers, and computer chess researchers. PGN is a complicated notation and has a lot of options. In this thesis, it is sufficient to cover only the most basic rules. A PGN game is composed of two sections. The first one is **tag pair** section and the second one is **movetext** section. The tag pair section is used to provide metadata about the game. The movetext section is a simple list of moves with the possibility of additional annotations to the moves with a result at the end. The moves are saved in SAN (Standard Algebraic Notation) [6], which is the classical notation used by humans. For example, move **Nc6** represents a move of a Knight to square c6. In the following text, some useful and basic PGN tags can be used to describe a game:

- Event – the name of the tournament or match event
- Site – the location of the event
- Date – the starting date of the game
- White – the player with the white pieces
- Black – the player with the black pieces
- Result – the result of the game

Here is an example of a completed game in PGN:

```
[Event "F/S Return Match"]
[Site "Belgrade, Serbia JUG"]
[Date "1992.11.04"]
[Round "29"]
[White "Fischer, Robert J."]
[Black "Spassky, Boris V."]
[Result "1/2-1/2"]

1.e4 e5 2.Nf3 Nc6 3.Bb5 a6 {This opening is called the Ruy Lopez.}...
...  39.Kd2 Kb5 40.Rd6 Kc5 41.Ra6 Nf2 42.g4 Bd3 43.Re6 1/2-1/2
```

2.2 Search

Essentially, chess is a simple game. At every position, simply calculate every move and all the possible replies so deep until the game ends. Practically though, this solution is

neither feasible by human players nor by chess engines due to the extremely large number of calculations and memory requirements. For early computers, there has been created a distinction between two types of approaches by Claude Shannon in the years 1949 – 1950 [15]:

- **Type A** – brute force strategy looking at every combination of moves.
- **Type B** – usage of chess knowledge to examine only a subset of available moves.

Chess programs view the game as a tree data structure. In this tree, each node represents one position in the game. Each subtree of the node is a possible move from that position. The tree is made of layers where each subsequent layer is the opponent's turn. One full move in chess is a turn by both players. Turn by just one player is considered half-move, or ply. This tree is created by the engine and is called **search tree**. As mentioned previously, it is not possible for the tree to include every single ply and therefore a three-stage tree model was designed. The first stage uses a type A approach, the second a type B search, and the final one a strategy called **quiescence** search which is described later in this section.

The Basic Search Methods

Since chess has an average branching factor of about 35 and many games can have more than 50 moves, the search tree has about 10^{154} nodes (even though in reality, there are less unique nodes) [25]. This huge number of nodes cannot only be searched entirely but also the algorithm must have as lowest complexity possible. Most chess programs use an improved variation of the minimax algorithm from the adversarial search category.

Minimax The minimax algorithm [25] is widely used for searching a state space. For a game with two players, the one from whose perspective is the game played will be called MAX and the opponent MIN. Let's say MAX is the player with the white pieces and MIN with the black pieces. MAX moves first, and then they take turns moving until the game is over. It is assumed that both players play optimally from the start to the end of the game. Given a tree, the optimal strategy can be determined from the minimax value of each node – `MINIMAX(n)`. This value of a node is the utility (for MAX) of being in the corresponding state. The minimax value of a terminal state is just its utility. Furthermore, for moving to the next level assuming there are multiple children, MAX prefers the child with maximum value, whereas MIN prefers a state with minimum value. This can be seen in following equation:

$$MINIMAX(s) = \begin{cases} UTILITY(s) & if\ TERM-TEST(s) \\ max_{a \in Actions(s)}\ MINIMAX(RES(s,a)) & if\ PLAYER(s) = MAX \\ min_{a \in Actions(s)}\ MINIMAX(RES(s,a)) & if\ PLAYER(s) = MIN \end{cases}$$

The minimax algorithm computes the minimax value from the current state. It uses a recursive computation of the minimax values of each successor state, directly implementing the defining equations. The terminal condition for the recursion is at the terminal nodes of the tree, from where the minimax values are propagated backwards through the tree. In the example in Figure 2.5, the algorithm first nests down to the three left leaves and uses

the `UTILITY` function to get their values 3, 12, and 8, respectively. Then, it calculates the minimum of these values, 3, and returns it as the value of node B. This process is repeated for the other branches, node C gets 2, and node D gets also 2. Finally, the maximum of 3, 2, and 2 is calculated and the result is sent to the root node, which is the result, 3. This way, the minimax algorithm performs a complete depth-first exploration of the game tree. If the maximum depth of the tree is m and there are b legal moves at each position, then the time complexity of the minimax algorithm is $O(b^m)$. The space complexity is $O(bm)$ for an algorithm that generates all actions at once, or $O(m)$ for an algorithm that generates actions one at a time.

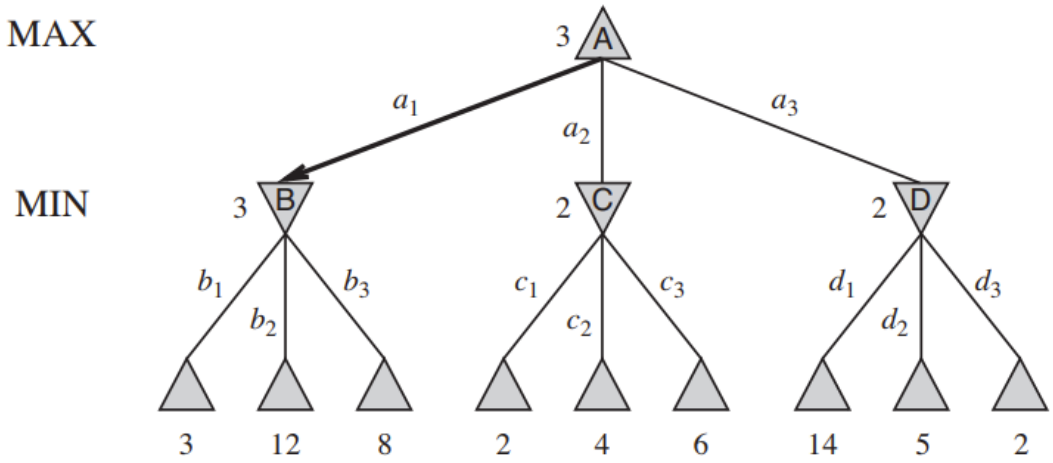

Figure 2.5: Minimax algorithm diagram taken from [25] – two-ply game tree. MAX and MIN labels mark which side's turn currently is. Terminal nodes show utility value for MAX, other nodes show their minimax values. Best move for MAX is *a1*, it leads to the highest minimax value. From MIN player perspective, the best response is *b1*, as it leads to the state with the lowest minimax value.

Although minimax is a fully applicable algorithm as a search for a chess engine, the number of states it has to explore is exponential with regards to the depth of the tree. This problem cannot be solved entirely but it is possible to reduce the number of nodes in many ways. The most known and used improvement of the minimax algorithm is called alpha-beta.

Alpha-Beta Alpha-beta algorithm [25] is a straight enhancement of the minimax algorithm. Considering the game tree from Figure 2.5, the same result can be achieved in fewer steps and some leaves can be skipped. Simply, by applying the in-order traversal to the tree, the *c2* and *c3* moves become redundant, since a smaller utility has been already found. Player MAX will not choose *a2* since *a1* guarantees score at least **3** while *a2* provides at most **2** regardless *c2* and *c3* evaluations. This algorithm follows the same procedure as the minimax, but adds two parameters that help to prune the tree:

- α = the value of the best choice that has been found so far at any choice point for MAX (with the highest value)
- β = the value of the best choice that has been found so far at any choice point for MIN (with the lowest value)

Alpha–beta search updates the values of α and β as it goes along and prunes the remaining branches at a node as soon as the value of the current node is known to be worse than the current α or β value for MAX or MIN, respectively. The complete alpha-beta search is illustrated in Algorithm 2.1.

The algorithm is split into three functions. The `alpha_beta_search` function expects an argument `state`, which represent current node on the tree. Then, it calls the `max_value` function with the current `state`, minimum and maximum vales for α and β, respectively. It returns the action (representing an edge on the tree) with the maximum value.

`max_value` is a function that expects the current `state` and the current α and β values. If the current `state` represents a terminal node, it returns its value. Otherwise, it loops all possible actions (moves) and calls the `min_value` function with the new state after the performed action and passes the current α and β values. The score of the searched subtree is stored to a variable `v`, which contains the maximal value of all searched subtrees from this `state` so far. Then, if this value exceeded or equalizes the current β value, it returns `v` and prunes the rest of the branches. Finally, if `v` is greater than α, its value replaces the current α and returns `v`.

`min_value` is the opposite function to the `max_value` one. It works in the same way, except it searches the minimal values and calls the `max_value` function. It skips the next branches if `v` is smaller than or equal to α. It updates β if `v` is smaller than current β at the end.

Alpha-beta Enhancements

Of course, in early computer engines, the base alpha-beta algorithm was sufficient enough to provide at least decent results. Then, the demand for better and faster engines made chess engine developers find new approaches. The trend is to make the computations faster and because the search is the bottleneck, a lot of attention is paid to the optimization. There were a lot of different enhancements over the years. For this thesis, only a small number is described [34].

Selectivity

The goal of selectivity is to make the search **type B** from **type A**, as explained previously in this section. Some branches can be skipped so the algorithm is faster and may be able to go deeper without having such a large number of nodes. On the other hand, some branches should be searched deeper because of their potential to contain better results.

Extensions Extensions is a discipline that extends the calculation in a node for some amount. There are many different approaches of extending the tree in chess such as *mate threat extensions*, *passed pawn extensions*, *capture extensions* or *check extensions*. Each of these are triggered by some other trait of a node.

Pruning Pruning are heuristics that completely remove certain branches of the search tree. Pruning can be divided into two categories, backwards and forwards pruning. *Backwards pruning* is sound and never affects the search result. Thus, alpha-beta may be considered as a pruning itself. Another example for chess is *Mate distance pruning*, which

Algorithm 2.1: Alpha-Beta Search Algorithm

```
function alpha_beta_search(state)
    v ← max_value(state, −∞, +∞)
    return action ∈ actions(state) with value v

function max_value(state, α, β)
    if terminal_test(state) then
        return utility(state)
    v ← −∞
    foreach a ∈ actions(state) do
        v ← max(v, min_value(result(state, a), α, β))
        if v ≥ β then
            return v
        α ← max(α, v)
    return v

function min_value(state, α, β)
    if terminal_test(state) then
        return utility(state)
    v ← +∞
    foreach a ∈ actions(state) do
        v ← min(v, max_value(result(state, a), α, β))
        if v ≤ α then
            return v
        β ← min(β, v)
    return v
```

cuts trees after a mate has been found and a branch can no longer lead to a shorter one. *Forward pruning* is the opposite, as it involves risks that the result will be influenced and differ from the bare method's result. Forward pruning is applied to reduce the search space by trying a move, then seeing if the score of the subtree search is still high enough to cause a beta cutoff. This type of pruning can also skip moves, which are very unlikely to exceed alpha.

Reductions Reductions are search heuristics that decrease the depth to which a certain branch of the tree is searched. As opposed to pruning, reductions do not cut the branches completely. First, *Late move reductions* [37] reduce moves that are ordered closer to the end of nodes which likely no move's score exceeds alpha. Second, *Fail-high reductions* search to a lower depth at positions that seem to be quiet and the side to move has established a great advantage according to evaluation.

Quiescence Search Quiescence search can be partly considered as a part of selectivity as it selects some parts of the search tree and performs operations on them [26, 27, 29]. This type of search is used for finding *quiescent* or *dead* positions. These are positions that can be assessed accurately without further search. In chess, they typically have no moves, whose outcome is unpredictable such as checks, promotions, or complex captures. Quiescence search is limited to dynamic moves to limit its size. The goal is to clarify the node so that a more accurate position evaluation is made than before. The methods which

perform quiescence search are simple, but in chess, they leave too much information that is needed. The not completely accurate definition could be that a quiescence search gives the rules for selecting the moves that make up the quiescence search tree [2]. So if the rules produced no moves at a position, the position would be a terminal node and alpha-beta would be used instead. The inaccuracy in the definition can be understood by looking at some positions where the captures would lose material but after some moves provide an advantage. The value of lost material by the sequence of captures is returned by the quiescence search according to the definition. That would be incorrect, and this method would not work effectively. It is important to add that at each node, the side to play is given the option of choosing the best capture *or* taking the static evaluation.

Parallel Search

With the continuous evolution of hardware, parallel searches have become very popular. The usage of parallelism and distribution of the computation amongst multiple processors speeds up the search enormously. Once again, there are many different parallel algorithms, but only a few of them are mentioned in this thesis.

Parallel Alpha-Beta Even though the alpha-beta algorithm seems to be inherently sequential as the tree must be traversed in a certain order, opportunities for parallelism still exist [23]. One way to obtain it is to simply search different subtrees in parallel. The computation in one node is split into several threads. Each thread searches a subtree and returns the final value and the updated alpha and beta parameters. The implementation requires more than starting additional processors though. Processors simply get the root position at the beginning of the search, and each searches the same tree while only communicating with the shared table. The gains come from the effect of nondeterminism, as each processor will finish after a different time and the trees diverge. The speedup is then measured on how many nodes the main processor can skip from transposition table entries.

Lazy SMP In the sequential case, the hash table and the data structures are always globally available to the search function [23]. In the parallel case, there is a problem with the access. Multiple simultaneous writes to the structures may have disastrous effects on the program. The shared hash table is the most common one in current chess engines. Lazy SMP [21] is one of the most popular algorithms among chess engines in 2022. It is a parallel algorithm discovered in 2013 where threads have minimal to no communication between them except a shared hash table. It was first described by Daniel Homan in a computer chess forum [10] after his experiments with parallelization. A very simple implementation gave him a significant speedup. That implementation has gone public and has been reworked and now is part of even the strongest chess engines. The main idea is that it uses a lockless hash table and therefore it avoids the overload of synchronization and communication problems and in exchange has a larger search overhead. Each thread searches the full game tree as fast as possible and the hash table ensures that the fully explored positions do not have to be re-explored. Additionally, there is a mechanism that leads the threads to search different subtrees.

2.3 Evaluation

Evaluation is the second crucial part of chess engines. The ways of evaluating a chess position differentiate the engines the most. Soft computing, machine learning and hand-crafted evaluation using chess knowledge are the most frequent techniques in this field. During this phase, a relative value of a position must be determined. This value corresponds to the chances of winning and if it was possible, every evaluation would decide if the game was always won, drawn, or lost. The evaluation is done at the terminal nodes of the search tree.

Hand-crafted evaluation

To manually create an evaluation function $f(P)$, that would be sufficient to beat the best players, not only an engineer but also a chess master's knowledge is needed during the process [28]. The chess master provides the necessary information for all the strategic decisions in the game which are afterwards implemented. The simplest evaluation function with only three results (0 for a draw, -1, and 1 for a certain win for either player) would not be enough as it would often evaluate many nodes with the same result and not necessarily choose the best move. Therefore, approximate evaluation functions were invented. Overall, the material values have the largest impact on the evaluation score. The evaluation function also considers long-term advantages and disadvantages of a position, which are effects that will persist over multiple moves such as positioning of pieces or pawn formations. Thus, the evaluation is primarily concerned with positional or strategic considerations rather than tactical ones. The tactics are projected in the static evaluation of the values of the pieces. All of the rules apply only to the middle game as the opening and end game require a completely different approach.

The material value of pieces is often interpreted a little bit differently, but the most common one is the following:

- **King** = 200 (more than the sum of all pieces)
- **Queen** = 9
- **Rook** = 5
- **Bishop**, **Knight** = 3
- **Pawn** = 1

Here is a list of some factors which can be included in the evaluation function:

- Material advantage (difference in total material).
- Pawn formation:
  - Backward, isolated and doubled pawns.
  - Relative control of centre (e.g., pawns on e4, d4 or, c4).
  - Weakness of pawns near king (e.g., advanced g pawn).
  - Pawns on opposite colour squares from bishop.

  - Passed pawns.

- Position of pieces:
  - Advanced knights (at e5, d5, c5, f5, e6, d6, c6, f6), especially if protected by pawn and free from pawn attack.
  - Knights near the edges.
  - Rook on open file or semi-open file.
  - Rook on seventh rank.
  - Doubled rooks.

- Commitments, attacks and options:
  - Pieces which are required for guarding functions and, therefore, committed and with limited mobility.
  - Attacks on pieces which give one player an option of exchanging.
  - Attacks on squares adjacent to king.
  - Pins to the pieces of high values.

- Mobility.

In the following equation, an example of a very simple evaluation function is shown:

$$f(p) = 200(K\text{-}K') + 9(Q\text{-}Q') + 5(R\text{-}R') + 3(B\text{-}B' + N\text{-}N') + 1(P\text{-}P') - 0.5(D\text{-}D' + S\text{-}S' + I\text{-}I') + 0.1(M\text{-}M')$$

Where characters "KQRBNP" are the number of kings, queens, rooks, bishops, knights, and pawns, D, S, I are doubled, blocked and isolated pawns and M is mobility, counted as the number of legal moves.

Machine Learning and Soft Computing

Many different machine learning and soft computing methods have been used in different engines to substitute the hand-crafted evaluation function. One of the most used models is the Effectively Updatable Neural Network. Some more examples – Knowledge-based, Problem solving, Statistical sampling, Genetic algorithms, CNN, Supervised models, GMM, and Deep learning.

Effectively Updatable Neural Networks NNUE [13] or Effectively Updatable Neural Networks (reversed abbreviation) were first proposed by Yu Nasu in 2018 in Japanese in [18]. This type of neural network started as an evaluation model for the game Shogi. The creator agreed to contribute to the chess community and published a paper with the idea. The big achievement was that this type of neural network could use alpha-beta algorithms for the search, as opposed to a Monte Carlo tree search that all previous methods needed. NNUE has only one output, which indicates the evaluation of the position in centipawns (pawn

has a value of 1, centipawn is one-hundredth of a pawn, so 0.01). The input is much more complicated. It is a binary encoding of a chess position. It starts with the player whose turn currently is and enlists all triples of all possibilities – (own king position, own piece type, the position of that piece) as well as (own king position, enemy piece type, position of that piece). The same information is then provided from the role of the other player. The architecture can be seen in Figure 2.6, it is a feed forward architecture, but is not completely fully connected. It is a network of three hidden layers, input bits are divided into halves due to the nature of the input. The first hidden layer is still halved, and each half consists of 256 neurons. The second and third hidden layers both have 32 neurons. From the second layer and forward, the neural network is fully connected up to the output. ReLU is used as an activation function throughout the network defined as follows:

$$ReLU(x) = max(0, x)$$

There is one oddity, the weights that connect the first half of the input layer to the corresponding half of the first layer are the same weights that are used to connect the second halves. They are shared in the sense that the mirrored piece-square relations share the same weight. With using NNUE, the alpha-beta search depth was significantly reduced due to the time spent computing evaluation scores but the strength increased. Nevertheless, the time penalty is a weakness. The current best engines use hybrid approaches where NNUE is only applied for quiet positions and otherwise, handcrafted evaluation is used. Regarding the name, the "Effectively Updatable" means, that the network can effectively update its weights by computing incremental differences with respect to a move that changes the position. The training of this network is performed beforehand by using the data of many played games. Its weights are saved to a file and loaded for evaluation.

2.4 Opening and Endgame Databases

During the opening and end game phases, different principles must be used for evaluation and often for the search too. For openings, there are many **opening theories** about what the best possible moves are up to a quite large number of moves. These moves are just considered best and there is not much more about them. Different moves often result in a disadvantage and the middle game starts there. Also, it is not good simple to calculate positions when all the pieces are on the board and there are not many reasons to do so when it can be calculated statically and not dynamically during the games. On the other hand, endgame databases are greatly needed for a computer, because the regular search cannot nest very deeply as the number of moves rises [7]. Human has an advantage here because they can eliminate a lot of move very quickly. But because there are not many pieces on the board, all the nodes can be searched and saved somewhere so the engines do not have to waste computational time during the games when it can be calculated beforehand. There are already databases (Syzygy) that contain all the possibilities for up to 7 pieces and every variation of their positions on the board. These databases are extremely large (hundreds of terabytes). Only parts of the databases can be downloaded. WDL – Win/Draw/Loss information can be used to decide which positions to aim for. DTZ – Distance To Zeroing (of the fifty-move counter by a capture or pawn move) can be used to reliably make progress in favourable positions and stall in unfavourable positions.

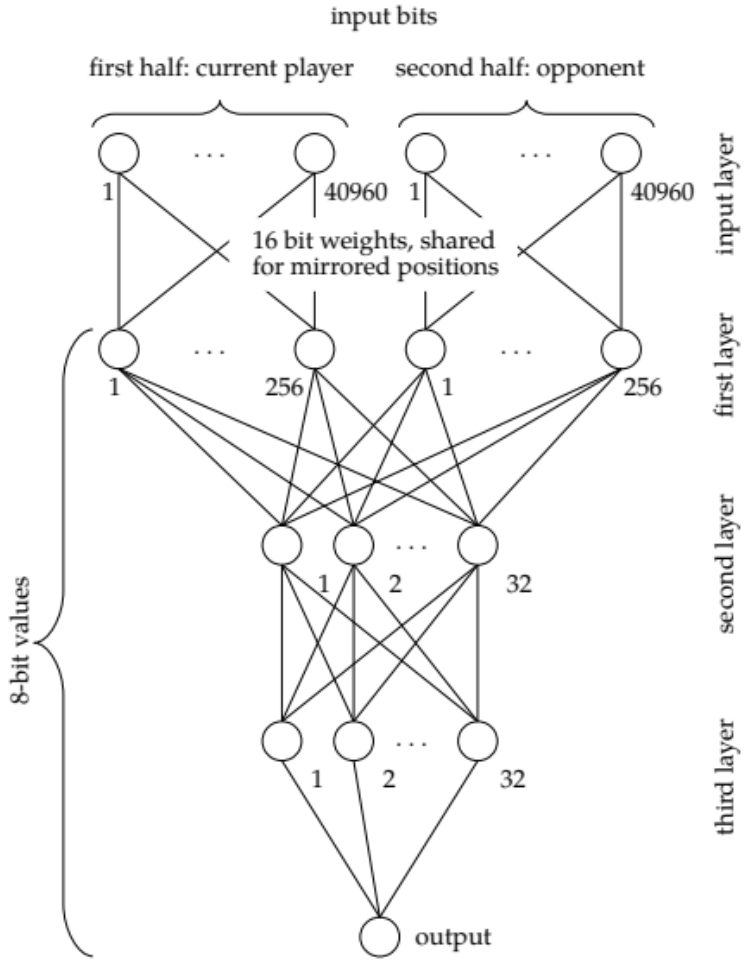

Figure 2.6: Architecture of NNUE. Taken from [13].

Chapter 3

Comparison of Chess Engines

In this chapter, a list of open source chess engines is assembled. From this list, the most suitable engine for this thesis will be chosen and be the subject of extension. The assembly is based on several criteria, which are specified beforehand. The selected engine must be easily expandable, the code should be properly documented and be suitable for the explanatory analysis. Most programs use the same underlying depth-first alpha-beta search algorithm [15]. What varies is, for example, the length of search assigned to each stage. Ultimately, the stage length is not fixed but varies by small amounts depending on the current sequence of moves being examined. For example, a search path may be locally lengthened because one side has attacked the king (given check), leaving the opponent with only a few alternatives to consider. There are so many options in search that even programs using the same basic model can achieve a radically different style and speed of play.

3.1 Choosing Criteria

Choosing the most suitable open-source chess engine is a complex process and requires a lot of consideration. The majority of the existing open-source chess programs follow the trends and are very similar in certain aspects but might have different features. This section focuses on choosing the criteria for the comparison.

Many different features can be taken into account during the comparison. Most of them are based on the source code, but some have a different nature:

- Board representation.
- Search features and enhancements.
- Adjustable depth of search.
- Type of evaluation.
- For hand-crafted evaluation, the aspects, and features.
- The optional settings such as enabling opening databases and endgame tablebases.
- Type of communication with the UI.
- The quality of code.

- Documentation or code commentary.
- Compilation time.
- Programming language.
- Playing strength (ELO).
- Maintenance and updates.
- Availability.
- Popularity.

It is unlikely that every point of this list might be useful in the comparison. Therefore, the most suitable one has to be chosen. The next part is very subjective, and the author of this thesis has made the decisions based on the theoretical research in chapter 2.

The first group of criteria is by far the most important one. Techniques, algorithms and methods used for the main functionality of the engines – **board representation**, **search** and **evaluation** – must be chosen wisely. Out of these three features, the evaluation is the most important part of this thesis. Board representation and search are often similar among chess engines.
The second group of features contains those, that can be helpful during the implementation but are not completely essential. The **type of communication** with the UI/GUI is quite an important feature. It should provide at least minimal possibility to communicate. Almost every modern chess engine uses UCI [35] – Universal Chess Interface which is an open communication protocol for chess engines designed in 2000. With that said, it would be great to use the most common protocol, but a different one would not be a great disadvantage. The **quality of code** and **documentation or commentary** would be a huge advantage as the code is often very complex. But despite the struggle without these features, the extension could still be implemented. Similarly, with **compilation** time. **Programming language** is primarily about the speed of the program and the personal choice of the programmer.

The last group contains some features that can be considered quite unnecessary. **Playing strength** is not very important to achieve the goals of this thesis. It is important for the chosen engine to be strong but does not have to be the strongest among them. All the candidates will be strong enough to provide a satisfactory analysis. **Availability** is extremely important but only open source engines can be considered. Otherwise, it would be impossible to use them for the extension. **Popularity, maintenance, and updates** the least important features but still must not be overlooked. These help in the regard that the extension could be based on a great program. Otherwise, the program could be outdated or even the evaluation function might be incorrectly implemented. Thus, these attributes should also play a role during the search but are not the most crucial ones.

3.2 Filtering Engines

After the importance of the criteria has been analysed, a suitable chess engine must be found. There are more than **1300** known chess engines based on the community of chess

programmers [33]. This number is enormous and it would be too difficult to go through all of them. Therefore, in this section, the process of elimination is described based on the criteria in the previous section.

First, the biggest reduction was done by the fact that for this thesis, an open-source engine must be used. This parameter alone shrunk the number to about **300** open source chess engines, either under GPL and MIT licences or for didactic purpose.

While there are so many chess engines, it is possible to choose by personal preferences. Generally, the C++ programming language is considered as a good language for this type of program. It is fast, which is convenient primarily for the search function. The author is also quite experienced with this language. Therefore, the engines written in C++ are the most interesting for this thesis. However, a few engines written in different languages should be also analysed. For example, engines written in other languages known by the author such as Python, Java, or Haskell can be also included. C++ is a dominant language regarding chess engines but still, some engines in languages such as Rust, Pascal, Javascript, and C were filtered out, and about **150** engines are left to be considered.

The next filtering is about the engines being modern and at least sometimes updated. The old approaches with arrays for the board representation and bare minimax algorithm are due. Also, the programs should be strong enough to beat a great chess player. Otherwise, the explanation would not be that useful. Some programs were not even complete, missing en passant capturing or castling moves. After this filtering, the number of engines is at this point about **100**.

As already mentioned, the board representation and search methods among the modern chess engines are quite similar with some exceptions. For the extension, the search method must be great but the algorithms that are used for its implementation do not matter that much.

The last part is about what remains to be compared. The concern is mostly about the **evaluation function**. Surely, for the extension to be able to explain the evaluation, this function must be sophisticated. Very likely, the evaluation function should be handcrafted to be able to easily be used for the extension. However, there might be some interesting ideas in the artificial intelligence methods, that could be used for the explanation.

There remain about **100** modern engines that are open source, written in a desired language, and can play chess well. Some of these are complicated, others are simple. Some use machine learning methods, others handcrafted evaluations, and so on. Many of them are similar and some are completely different. These engines were explored in more detail. As 100 engines are still too many to write a comparison about in detail, a selection of **10** engines with a diversity to cover all the differences was performed. For example, there are about 50 % engines written in C++ out of the remaining ones, so 5 of them are chosen for the comparison. The selection of similar ones was done based on their popularity. These engines are compared in detail in the next section.

3.3 Final List of Engines

The final 10 chess engines that met the requirements for the deeper comparison are briefly described and compared in this section. The remaining criteria to compare are mostly the functionality and the quality of the code with commentaries. The positives and negatives of each engine are mentioned. The list is sorted mostly by the popularity of the engine and its potential for this extension. Out of these engines, a single one is chosen that will be extended by the explanatory analysis.

Stockfish

Stockfish [19] is a powerful UCI chess engine. It is the strongest open-source engine, winning many different competitions. Its latest version is Stockfish 14, released on July 02, 2021. It is highly maintained, and the code is clean and well documented.
Programming Language: C++
Board Representation: 8x8 bitboards, magic bitboards
Search: Extremely sophisticated and tweaked search methods. Alpha-beta with iterative deepening, aspiration windows, transposition table, and move ordering. Parallel Lazy SMP. Many more enhancements.
Evaluation: NNUE or handcrafted
Positives: Opinion to evaluate either by neural networks or by handcrafted function. This function is sophisticated and includes the most features of all searched engines. Includes the optional usage of endgame tablebases. Popular, many people know only about Stockfish and want to work with it so this extension could reach someone.
Negatives: The complexity of this program might be a problem during the implementation. High traffic on the repository, frequent updates and commits, after the end of this thesis, already many new versions might be implemented. Its playing strength might be too much.

Leela Chess Zero

Leela Chess Zero [14] is a chess engine with UCI support. Its goal is to build a strong chess-playing entity following the same type of deep learning along with Monte Carlo tree search techniques of AlphaZero but using distributed training for the weights of the deep convolutional neural network. AlphaZero is a commercial chess engine by Google and evaluates positions using a non-linear function approximation based on a deep neural network, rather than the linear function approximation as used in classical chess programs. It is known as the best engine using deep neural networks. Uses reinforcement learning to train models.
Programming Language: C++
Board Representation: Five simple bitboards
Search: Monte Carlo tree search
Evaluation: Sophisticated deep neural network
Positives: Extremely strong deep neural network engine. Includes the optional usage of endgame tablebases. Uses the most modern machine learning techniques and is still being updated. Its approach might be the future.
Negatives: Not using handcrafted evaluation at all. Monte Carlo is based on randomness. Complicated code, which would be quite difficult to extend.

Igel

Igel [30] is a UCI chess engine developed by Volodymyr Shcherbyna. Having only one author, Igel is maintained, but not updated too frequently. It used handcrafted evaluation and switched to IGN, which Igel Generation Network, amended NNUE.
Programming Language: C++
Board Representation: Magic bitboards
Search: Lazy SMP, alpha-beta with many pruning enhancements
Evaluation: IGN or handcrafted
Positives: Strong engine with the choice of different evaluations. Includes the optional usage of endgame tablebases. Probably easily expandable code, well written. Not as frequently updated.
Negatives: Not the greatest commentary in the code. Not the most sophisticated evaluation function.

RubiChess

RubiChess [16] an UCI compliant chess engine by Andreas Matthies. Also has a single author but is updated monthly. Similarly to Igel, RubiChess switched to NNUE but still supports handcrafted evaluation.
Programming Language: C++
Board Representation: Bitboards
Search: Lazy SMP, alpha-beta with many pruning enhancements
Evaluation: NNUE or handcrafted
Positives: Strong engine with the choice of different evaluations. Includes the optional usage of endgame tablebases.
Negatives: Poorly written code and not well documented. It would be difficult to extend the code because it does not provide great modularity.

Bagatur

Bagatur [31] is a chess engine by Krasimir Topchiyski. Written in Java and is also available for Android OS Also has a single author but is updated monthly.
Programming Language: Java
Board Representation: Combination of bitboards and arrays
Search: Special algorithm based on the alpha-beta search
Evaluation: Single-layer perceptron, supervised learning is used to tune weights
Positives: The code is structured in multiple Java modules, which are great for the extension. Great opening database and generator. Could be used also on phone.
Negatives: Not using handcrafted evaluation. The codebase is large which might cause problems.

ConvChess

ConvChess [20] is a convolutional neural network (CNN) engine by Barak Oshri and Nishith Khandwala. It was the first engine using CNN. It is implemented in Python using its machine learning libraries such as NumPy. It used to be a didactic program. Contains 7 CNNs.
Programming Language: Python

Board Representation: Limited arrays
Search: First layer of CNN
Evaluation: CNN
Positives: Interesting idea. Whole chess engine is implemented in CNN. Quite short and simple code, easily understandable. Python libraries are great for machine learning purposes.
Negatives: Might be outdated and not strong enough. Not using handcrafted evaluation. UCI support is not implemented.

Sunfish

Sunfish [1] is a simple, but strong chess engine, written in Python, mostly for teaching purposes. It is compact with an even more compact version only written in 111 lines of code. It provides a platform for experimenting.
Programming Language: Python
Board Representation: Limited arrays
Search: MTD-bi, which is enhanced alpha-beta
Evaluation: Very simple handcrafted function
Positives: Not finished program, which could be a convenience in a way. Simple, easily extendable, and adaptive. Still maintained. Well documented.
Negatives: Might be not strong enough. Not complete, does not allow some promotions. A lot of coding would be needed.

Maia

Maia [17] is a chess engine featuring deep learning by multiple authors. It aims to have human-like behavior. It tries to predict the moves as close to the user's strength as possible. It does not learn from self-play but from human games. Maia models completely rely on training by supervised learning. Further, Maia does not use search at all, it just predicts moves.
Programming Language: Python
Board Representation: Arrays and convolutional network
Search: None
Evaluation: Prediction by deep learning
Positives: Maia is a completely original chess engine that brings a lot of new ideas. The possibility to set own strength and train using the appropriate level of a player can be great for learning. It can be great for the explanation of the moves. Greatly documented in a paper.
Negatives: Not using handcrafted evaluation. Not the strongest program.

Winter

Winter [24] is a chess engine by Jonathan Rosenthal. Winter is inspired by machine learning techniques, as applied in move ordering and evaluation. It has a unique evaluation method. It uses Gaussian Mixture Models (GMM) and Fuzzy C-Means algorithms. The newest versions also include neural networks.
Programming Language: C++
Board Representation: Bitboards, 8x8 board
Search: Lazy SMP, alpha-beta with some enhancements, move ordering based on logistic

regression classifier
Evaluation: Mixture model, fuzzy c-means, neural network, trained via reinforcement and supervised learning
Positives: Contains many different evaluation methods and it is possible to choose which one to use. Might be compiled on Android. Well-written code and maintained quite frequently.
Negatives: Not using handcrafted evaluation. Not very well documented code.

Barbarossa

Barbarossa [12] is a chess engine by Nicu Ionita. It uses the functional programming concept of monad transformers to control the search. Even with the disadvantage of the speed of the Haskell language, Barbarossa can compete with many engines.
Programming Language: Haskell
Board Representation: Bitboards
Search: Alpha-beta search with enhancements and improved by the monad transformers concept
Evaluation: Handcrafted evaluation function
Positives: Functional programming and Haskell are different in many aspects from procedural programming and could be great for some purposes. Great commentaries in the program and also Haskell itself provides some commentary by the type definitions. It is still maintained and updated.
Negatives: Haskell is for some a difficult programming language to understand. Functional programming might be great in some regards but also not suitable for different things. The handcrafted evaluation does not provide awesome results.

Now, the last step to find the most fitting chess engine must be performed. However, the main ideas for the extension must be introduced beforehand. This is described in the next chapter.

Chapter 4

The Chess Engine Selection

In this chapter, the main ideas for the extension are explained. The extension of the selected chess engine should be able to provide analyses of chess games. The analyses must have a humanly understandable explanations of the moves. Humans and computers have quite different approaches to solving chess positions. There are some positions that computer always solves perfectly, and it is not possible to provide an explanation for a human to understand. Therefore, the most appropriate parts of the game must be found to be then explained. The explanation is mostly intended for intermediate players.

4.1 Main Ideas for the Extension

In this section, the main ideas for the extension are explained. Based on these ideas, the most fitting engine out of the compared ones is chosen. The engine is then explained in the next section.

The explanation of the chess game is a very complex task. Its goal is that it should help a player to **understand** their games better and to **improve** their game in general. Explanation of the game should be implemented as an explanation of the moves that a player made in that game. Therefore, the main task will be an explanation of a single move. Then, each move should be evaluated separately and an explanation of the reason for the evaluation given. This can then be repeatedly used for every move of a given game and some summary of the most important moves provided. This way, the whole game can be explained.

Now, the question is what the ideal evaluation should look like. The evaluation must be **detailed** to be able to provide a lot of information to give to the user. Also, it is necessary to work with the **classical evaluation function** instead of using artificial intelligence algorithms. This function has to be reverse-engineered and split into many parts which would be then used for the explanation. The artificial intelligence evaluation functions would not be suitable because the reason for the evaluation would not be clear. If an engine with such an evaluation function would be used, the author of this thesis would have to implement the evaluation from scratch. The goal of this thesis is not to propose the best evaluation function as the author is not qualified enough. Therefore, the chosen engine must be as sophisticated as possible and its evaluation function should be implemented with the help

of the best chess players, grandmasters.

The explanation of a move should be done by comparing the user's move with the best possible move in the position by the engine on the chess board. Because of that, the engine with the best playing strength should be chosen. Then, the difference should be explained. The engine provides the possibility to perform the search and find the best possible sequence of moves and the estimated evaluation after the moves are made. Thus, the explanation of a move should be done in the following way:

1. Load the current position to the engine.
2. Perform a detailed evaluation of the position.
3. Perform a search to find the best possible move from this position according to the engine and get the estimated evaluation.
4. Perform a search after the user's move and find the best possible continuation and get the estimated evaluation.
5. Compare the positions based on the evaluation.
6. Grade the move based on the values, if a lot of the score has been lost, tag it as a bad move and vice versa.
7. Show both positions (and optionally also the starting one) so the user can compare them.
8. Find the most impactful factor of the evaluation difference and provide the explanation for each position.

Based on this sequence of actions, the user should get the necessary information to understand why the move they made was either good or bad. It might also be helpful to provide a complete comparison of the positions but it would be almost impossible to implement it. Also, the explanation could be much more detailed than explaining only the most important factor but the user could be easily confused by the amount of information. The goal is to provide the most important information so the user can understand it immediately. However, for a player that is very new to chess, this is not great as they probably cannot understand the terminology that is used in the explanation.

Another idea is that often the evaluation of the position is 0, which means that the game is in a dead draw with perfect play but the moves are very difficult. The reason is that the engine searches very deep and knows the perfect response for each move. This must be done differently in the extension. The search depth must be shortened, and the most important threats analysed. Moreover, the whole match will be analysed in the sense of accuracy and best moves. The opening and endgame phases of the game are already solved using databases. These databases must be either turned off during the explanation or no explanation will be provided for these parts. The representation of the results should be provided by the program at least in proper text output.

The code in the extension should be in modules so the extension could be separated from the other code. The commentary of the evaluation function is also handy as its parts will

be used for the explanation. The implementation should serve as an optional module in the existing chess engine so it could be either used on-demand or ignored completely.

From the comparison in Chapter 3, it is clear that to reach the expected functionality, a chess engine with a handcrafted evaluation function must be used. This function should be as complex as possible to be able to explore and explain as many of its features as possible. The programming language should be C++ as it is the most common, fastest out of the listed ones and the author is familiar with it. After the whole analysis, the main ideas and the comparison of the engines, the most suitable one for this thesis is the **Stockfish** chess engine.

4.2 Stockfish

After the comparison in Chapter 3 and the exploration of the engines, the Stockfish chess engine overcame the rest. Stockfish is also able to provide the important features that are needed for the implementation of the ideas in the previous section. It is a very strong engine with a proper classical evaluation function implemented with the help of grandmasters. It has a decently clean and well-documented code, the programming language is C++, and the modularity is also great. It was chosen as the most suitable one for the extension as it fulfills all the requirements the best. Its code will be used as a base for the extension implemented for this thesis. Therefore, this engine must be studied thoroughly. The version of the program is **Stockfish 14**. In this section, the Stockfish engine is described in detail.

Stockfish [19] is a free UCI chess engine. It is not a complete program and requires a UCI-compatible GUI for a proper usage. Many open source or free chess GUI programs capable of communicating using UCI commands exist, e.g., XBoard or Scid. Nonetheless, Stockfish provides a command line user interface which can be used with the knowledge of the instructions. Through this interface, the whole potential of the program can be utilized. Stockfish features two evaluation functions for chess. One of which is a NNUE base evaluation which gives currently the best results of all open source chess engines. The other is a classical handcrafted evaluation which is also very strong. It can switch between the evaluation functions. The classical evaluation is used during this thesis.

4.2.1 Functionality

As already mentioned, the program can be controlled by commands from the command line. Via the UCI, settings for the engine can be adjusted, such as the playing strength or the option to use NNUE evaluation. The program provides additional commands for the user to manipulate with the engine itself, such as running the evaluation or showing the current position on the chess board. The most important commands are listed further.
The commands that set UCI options have the following syntax:

```
setoption name <id> [value <x>]
```

Where `<id>` is an identification string of an option and `<x>` is the value which is assigned to the specified option. The `<id>` and `<x>` are case insensitive.

This is a list of available UCI options, which are important for this thesis with their types and possible values (Name – possible values):

- **Ponder – true|false**
  This option lets the engine calculate the response while the opponent is thinking about their move.
- **MultiPV – 1 to 100**
  An option that sets how many best moves should be searched for in a position.
- **Use NNUE – true|false**
  This option toggles between the NNUE and the classical evaluations. For the NNUE, the path to a file with the neural network model has to be specified.
- **UCI_LimitStrength – true|false**
  Must be set to true to be able to use other options which adjust the strength of the engine.
- **UCI_Elo – 1350 to 2850**
  The value of this option affects the playing strength of the engine.
- **SyzygyPath – string**
  Specifies the path for the Syzygy tablebase files. If no file is set, the tablebase evaluations are not used. Multiple paths can be input, separated by a semicolon.
- **Slow Mover – 10 to 1000**
  The value affects the time spent searching for the best moves.

Every option also has the default values. Once one of these commands is input, it is processed, and the values are saved and used later during the program run such as search and evaluation.

Aside from the UCI commands, Stockfish also accepts additional ones which are used for communication between either the GUI or the user with the engine. These are listed and explained in the following list:

- `uci`
  Lists all available UCI options with their types and default values.
- `ponderhit`
  Tells the program that the user has played the move that was expected. That means the best possible move in the current position calculated by the engine has been played by the opponent. This is only useful if the program was in a pondering mode (expecting the best move and calculating during the opponent's turn). The program quits this mode afterwards. Ponder mode is used for possibly very fast calculation.
- `go`
  This command calls the function that sets parameters for the search function and then runs the search. Many parameters can be set for the search. The majority of the parameters must be followed by a value which is processed. Multiple parameters can be input separated by a space. Some of them are listed here:

- `searchmoves`
  Can be followed by multiple values. The values are moves written in syntax explained later. The search is restricted to these moves only.
- `depth`
  Its value regulates the search depth to the number of plies.
- `nodes`
  Its value regulates the search to the number of nodes.
- `mate`
  Makes the search look for positions from which there is a forced checkmate sequence in a number of moves specified by the value.
- `perft`
  This is a utility command which makes the search function count all generated nodes up to the depth specified by the value.
- `wtime|btime` and `winc|binc`
  These parameters inform the engine about the time rules and remaining time of either player. The value is assigned in milliseconds.
- `movestogo`
  Also a parameter regarding time control. It informs about how many moves are left for an increment or next time control.
- `movetime`
  The search time is restricted to the value in milliseconds.
- `ponder`
  This parameter does not require a value. Starts the searching in a pondering mode. This means that the engine is calculating the best response and the best continuation during the opponent's move. If the opponent moves as expected, the `ponderhit` command is sent to the engine and the pondering results can be used. Otherwise, the search must be run from scratch.
- `infinite`
  This parameter does not require a value. Search runs forever unless stopped by the `stop` command.
- Example of this command in the starting position:
  `go depth 10 searchmoves e2e4`
  The engine searches up to the depth of 10. The first move of the search is restricted to the move `1.e4`. After the search is done, it returns the best sequence starting with the move `e2e4`, the ponder move, evaluation, and more information about the search.

- `stop`
  This command is used to stop the search started by the `go` command.

- `position`
  By this command, the position of a chess game is set. There are two different parameters for how to specify the position. One of them is a FEN string as explained in 2.1. The second possible parameter is string `startpos` which represents the starting position. Both ways load the position with all the important information. Additionally, both parameters can be followed by any number of moves. The moves are simulated from the position. Their syntax is specified later in this section.

- `flip`
  Flips the internal representation of the position to be able to show the game from either color point of view.
- `d`
  Prints the position in ASCII representation.
- `eval`
  Prints the static evaluation for the current position. Prints both classical evaluation and NNUE piece values and score contribution.
- `quit`
  Exits the program.

The move format is in long algebraic notation (LAN) explained in section 2.1.

4.2.2 Code Structure

The theory about the implementation of a regular chess program is described in Chapter 2. In this section, the code of the Stockfish chess engine is analysed in detail. Only the most important parts of the thesis are explained due to space reasons. As there are many authors of Stockfish, the architecture of the code is not smooth. Some files contain classes, while others use structures and namespaces. Therefore, the code structure can not be simply represented by a single diagram such as the UML class diagram. The simplest way of understanding the structure is by checking the files. The original code can be found in the repository [19].

As previously mentioned, the program is written in C++. Therefore, the program starts executing the `main` function in the `main.cpp` file. Inside this function, all classes, structures, maps, and other variables are initialised. First, the **UCI** is initialized by assigning each UCI option its types and default values. Then **PSQT** (piece-squared tables) are initialized. These tables contain score values for every piece type on every square on the chess board. The tables, such as other score tables always contain `Score` type that is defined as a tuple with – usually different – scores for middle game and endgame. Next, **Bitboard** tables are initialized. Then, **Position** class is initialized. This class contains information about the current position and starts in the classical chess starting position. After that the special **Endgame** evaluation classes are initialized. Finally, the **Threads** for the **Search** are created and the cache is cleared. And lastly, **Evaluation** class is initialized. The parameters for the initialization are set before the build by arguments.

UCI After all of the initialization is done, the program sets up the communication with the user or the GUI. The files `uci.cpp/uci.h` handle the interactions and call methods from the rest of the program. It includes a function called `loop`. This function waits for a command from standard input. Once the input is received, it is parsed and the appropriate function is called. This method can be terminated using the `EOF` signal or by the `quit` command. The available important commands are listed and explained in section 4.2.1.

Bitboards Files `bitboard.cpp/bitboard.h` contain everything related to the bitboards. Many bitboards are defined. Some are predefined and represent the color of squares, each rank or file on the chess board, queen- and king-side squares or flanks. Other are bitboards that are only defined and filled only during the run, such as distances from a square to a different square or attack from pieces of a given color. Then, some magic bitboards are defined. There are defined many operations over bitboards, e.g., AND, OR, or XOR for easier manipulation and calculations. Also, some utility functions, e.g., **shift** which shift the pieces on the board in a selected direction for example to check for the pawns moves, or **popcount** which counts the number of non-zero bits in a bitboard and is used for counting occurrences in a bitboard.

Position The representation of position in the game is implemented in the class `Position` in files `position.cpp/position.h`. This class stores information regarding the board representation as pieces, side to move, castling info, etc. This representation is used by the search method a lot as every move is simulated by methods `do_move` or `undo_move`. The position is created from information stored in structure `StateInfo` which is also defined in these files. A list of these structures from the starting position to the current position is created and used to detect a possible draw by the repetition rule. The `Position` class contains methods to simulate moves, for setting the position by FEN or getting the FEN string, methods that populate attacking bitboards, methods that validate the moves, and those that check for stalemate.

Search The search is implemented mostly in `search.cpp/search.h` files and the multithreading in `thread.cpp/thread.h` files. The search in Stockfish is a multithreaded minimax algorithm with various enhancements such as reductions or pruning called Principal Variation Search and is very similar to the alpha-beta algorithm. There are two search types. The main search function is used in the search tree up to the specified depth. After the depth is reached, the quiescence search function is called. The search function has many different parameters, for depth, time management, or enhancement options. The search is run in multiple threads. These threads are initialized at the start of the program and wait until the search is called. As this is done asynchronously, many precautions are taken into consideration. The search is called by unlocking the mutexes of the threads. The search functions are called in the threads. After the search in one thread is completed, it is locked again. The result can be retrieved after all threads finish and usually the thread with the best search result is chosen and the result kept. The result contains mainly the best move sequence and the one search evaluation score.

Evaluation The last, but most important part of this thesis is the files where the evaluation function is implemented. In the files `evaluation.cpp/evaluation.h` and also in `evaluate_nnue.cpp` there are classical and NNUE evaluation implementations, respectively. The neural network has the structure as explained in section 2.3. In the program, the neural network architecture and logic are defined but the evaluation only requires loading the network from a file. The file contains the model and its weights and can be used for evaluation. The classical evaluations include many constants that represent scores for every feature that can appear in a position and is used for the evaluation if it is present. The score constants always have the middle game and endgame values as the features often have different values. There is a main evaluation function that only sums up the decomposed ones. The evaluation is always calculated for a specific position and color. The parts

which are evaluated are values of the material, mobility of pieces, king safety, and much more. For some, there are even extra files, such as for evaluating pawns, their structure, and other features specified in `pawns.cpp/pawns.h`. The final score is then interpolated from the middle game and endgame values based on the current position.

Special evaluation The evaluation function cannot be used always, though. In some scenarios, the function would not give good enough results and therefore a special evaluation must be used. These are implemented in `material.cpp/material.h`. Also, if the game has reached the endgame phase, a special evaluation must be used. Either tablebases are used which are pre-calculated moves for up to 7 remaining pieces on the chess board, these are explained in section 2.4. These tablebases must be loaded from a file, but if there is no file, the endgame is calculated in files `endgame.cpp/endgame.h`. Endgame evaluations are tricky as they can be very precise because not many pieces are left, and they can be calculated very deep. However, for a person to know the best move in the endgame, requires a lot of chess theory knowledge.

Additional files There are more files, but they contain code that is not that important for this thesis. Some work with input and output, others are helper files for search, such as generating moves or picking the best moves. There is also a file `types.h` that contains macros and constants for pieces, squares, evaluation values, colors, and functions which work with them.

Now, that the selected chess engine has been explored and explained, the extension can be designed be implemented. The extension should follow the same code structure as is in the program base. That means the implementation should be in one separate .cpp file and its header and the formatting should remain the same. Also, the extension should interfere with the other code as little as possible. Its function should be accessed only from the `uci.cpp/uci.h` files. It must be accessible by a command from the command line. In the next chapter, the design and implementation of the extension are described.

Chapter 5

Design and Implementation

In this chapter, the design and implementation of the extension for the chosen chess engine Stockfish is described. The designing and programming part of this thesis is based on the main ideas in section 4.1 in previous chapter. The explanation of the chess game and the moves made by a player was designed for this engine and implemented as an extension which can be accessed by a command. Except this, it does not interfere with the rest of the program which still works as intended. As this extension is based on an existing chess engine, understanding of its code structure and functionality is essential. These are explained in section 4.2. Some of the code of the base program has been reused for this implementation. The description primarily focuses on the explanation of one move. The whole game explanation is described in the last section.

From the section 4.2.2, the following files were extended or new files were added:

- `uci.cpp/uci.h` files were extended by adding a new input parameters and input parsing
- `evaluation.cpp/evaluation.h` files were extended to get the detailed evaluations and the explanation
- `panws.cpp/pawns.h` files were also extended for a special evaluations for pawns
- `analysis.cpp/analysis.h` files were added and contain the move sequence analysis code
- `explanation.cpp/explanation.h` files were added with the code for the explanation part

The implementation starts in the loop in the `uci.cpp` file. This loop serves as the communication between input and output. It waits for a command from standard input which is parsed and the appropriate function is called based on the string tokens. The explanation of one move has a specific token which is recognised from the input and then the explanation algorithm starts by calling a function. This input token is "a" which is abbreviation of "analyse". The called function is `analyse` and is in the same file. This function has three arguments – `pos`, `is` and `states`. The `pos` argument represents the current position. The position can be set beforehand by the `position` command followed by a FEN string, explained in section 4.2. The second argument – `is` – is a string stream containing the input commands. The last argument, `states` contains the information about the previous

positions that led to the current one. In this extension, it is not possible to work with all previous positions, so argument `states` gets only a queue with one empty state which is then filled by function that sets the position. This must be done to perform a search from certain position.

5.1 Move Sequence Analysis

This section describes the implementation detail of the main idea. This part is implemented in the `analysis.cpp/analysis.h` files. To remind, the main idea is that the explanation should give the user a better understanding of the move they did. The idea to solve this problem is to give the explanation of the positions to both situations either if the user's move **was** performed or if it **was not** and the best move was played instead. Therefore, these two searches must be performed and the two positions for the explanation must be found.

Analysis First of all, a class `Analysis` is created. Its purpose is to store information about the analysis of the sequence of moves. It stores the following information:

- `move_num` – number of the last simulated move
- `moves` – vector of all simulated moves in the sequence up to this position
- `evaluations` – vector of detailed evaluation (structure `detailed_eval`), explained later in this section
- `pos_fens` – vector of positions in FEN
- `search_score_cp` – approximate score from the search in centipawn value (score * 100)
- `static_score` – score from the static evaluation of a position
- `move` – the first move
- `response` – the best response to the first move

All the moves are in the string format. One instance of this class is created for the best sequence and another one is dedicated to the sequence with the user's move.

Before the search is used, some of its options must be set. One of them is a UCI option to use NNUE evaluation. This option must be set to `false`. The second option is to set a limit for the search. This limit was set to the depth of **18** plies. The value was determined after a couple of tests and the main reason is the balance between the speed of a search and the quality of the found moves. Also, deeper sequences would be too difficult to explain for humans.

The search is performed a few times, always with the same parameters. It is run by waking up the thread designed for the search. These threads require four arguments – `position`, `states`, `limits` and `ponderMode`. Argument `position` is the position from where to start the search. `states` is again the state info which contains the information about the previous positions that led to the current one. `limits` contain the options for the search.

`ponderMode` is an option if the engine should be calculating the next move, expecting the opponent to play the best move. It is always set to false as it is not needed. Then, the threads start the search asynchronously and after all are finished, the best result is kept. Meanwhile, the main program is set to wait until the search is finished. The result of the search is originally printed to a standard output. There is no simple way without changing the code for the search to get the result. Therefore, the result is obtained by redirecting the standard input to a string before the search and after the search, the results can be accessed. The important parts of the search result are the list of the best moves and the evaluation. These are obtained by parsing the string variable.

The **first search** is performed at the beginning. The result contains the best possible sequence of moves from the starting position. From this search, only the first move and the opponent's best possible response are used. The moves are simulated and the `Analysis` instance for the best sequence is filled with the data. The simulation of a move is implemented in a specific way:

The simulation of a move

1. `states` variable is cleared and filled with a queue with one empty entry.
2. Starting position is set by the `Position::set` function, passing `states`, position in FEN and the main thread.
3. A move string is converted to a `Move`, which is executed by `Position::do_move` function.
4. An instance of `Analysis` class is filled with this information:
   - `moves_num` variable is incremented.
   - The detailed evaluation `detailed_eval` is added to the `evaluation` vector.
   - The FEN of the newly obtained position is added to the `pos_fens` vector.
   - The move in a string format is added to the `moves` vector.

The gained information is the search score in centipawns and mainly the sequence of the best moves. The score is saved to the `Analysis` instance for the best analysis.

Similarly, the information for the sequence with the user's move is gathered. First, the move from the input is simulated. From the new position, the **second search**. After this search, every move from the best sequence is simulated.

The main reason for these analyses of the best move sequences was to get a detailed evaluation of every position. For this detailed evaluation, a structure `detailed_eval` was implemented, described in the next paragraph. The purpose of this detailed evaluation is to decompose the standard evaluation which only is represented by a value telling which side on the chess board stands better. However, this evaluation must be detailed to get a better idea of the reasons why one side is better in that position. In a method called `detailed_value`, the decomposition of the evaluation is implemented. Before the implementation, the evaluation function of Stockfish called `value` had to be completely understood to know how to make to decomposition properly and which score-adjusting parts are important. The parts are for both white and black player.

Detailed evaluation The original static evaluation function provides only a score which represents which side is better in a position. This evaluation position had to be partitioned to be able to concentrate only on a certain part. If the whole evaluation was explained, the output text would be way too large. Therefore, the structure `detailed_eval` was created and contains the following parts:

- `mat` – simply the material scores without the regard to the location on the chess board
- `psq` (one value) – evaluation of material based on piece square tables, in which the score of every piece type on every square is defined
- `pawns` – score given by various factors regarding pawns, such as formations
- `knights` – score given by various factors regarding knights, such as outposts
- `bishops` – score given by various factors regarding bishops, such as pinning
- `rooks` – score given by various factors regarding rooks, such as control of open files
- `queens` – score given by various factors regarding queens, such as if they are in a danger
- `mobility` – score based on the number of possible moves of pieces
- `kings` – score of the king's safety
- `passed` – special evaluation for the pawns that are considered passed
- `threats` – score for the possible threats in future moves, such as pushing a pawn
- `space` – score primarily used in openings, it is based on the number of controlled central squares and safe squares for friendly pieces, not used in later stages of the game but frequently at the beginning
- `eval` (one value) – the summarized value of the evaluation

5.2 Finding the Ideal Positions

Now, that the sequence analyses are complete, one more problem must be solved before the explanation. It would make no sense to explain every position in both analyses due to two main reasons. The first reason is that the information output would be too large. The second reason is that the static evaluation of a position saved in `eval` field in `detailed_eval` structure does not have to correspond to the approximate score `search_score_cp` of the whole analysis (see section 5.1). The sequence of moves can contain some chess tactics, such as a sacrifice of a piece. After the move that sacrificed a piece, the static evaluation almost always shows a decrease in the score, because the player lost a more valuable piece than the opponent. However, a few moves later, the player can get some compensation or even checkmate the king. This is only an example but many alike situations can happen often during a game. The search counts in these situations. The best sequence of moves does not end in the middle of a tactic and tries to find a stable position. The search score is based on a quiet position at the end of the sequence. Thus, a position with a similar static evaluation score as the estimated search score must be found for the explanation to make sense.

For the explanation to be more user friendly, one more improvement has been implemented. Its purpose is to find balance between the precision of the score similarity and the length of the sequence. The precision is very important because if search and static scores were significantly different, the explanation would not be helpful. The length of the sequence is less important but still, if user should understand up to 8-move sequences, the information gain from the explanation would be less relevant. The improvement is programmed as presented further.

Precision-depth balance A threshold is created that must be passed by a static score of a position to be accepted. This threshold is set to be very low at the beginning and is gradually increased with every move. This threshold achieves that at a small depth, only minimal score deviations are accepted and at a bigger depth, a larger score difference is tolerated. The score difference of 1 generally means that one side has an advantage that could be converted to one pawn. With this knowledge, the threshold uses the following equation:

$$T_M = \frac{Dev_{max}}{Dep_{max} - M}$$

Where T_M is the threshold value at M-th move in the sequence, Dev_{max} is the maximum deviation, Dep_{max} is the maximum depth in the sequence that can be accepted and M is the current move number.

The values for the variables were assigned experimentally from the human point of view. The value of Dep_{max} is set to 10 (8 for the depth because more moves to simulate could be too many to understand for humans + 2 to get more strict threshold). The Dev_{max} value is 2 as a bigger value gives too imprecise results. With these values, at the first move the maximum deviation of 0.22 is allowed and 1.0 at the eighth move. If there were more moves in the sequence, this method ends on the eighth move to prevent division by zero by breaking the loop. On rare occasions, the deviation may never pass the threshold, so the position with the smallest deviation so far is used. Figure 5.1 showcases the threshold function in a graph.

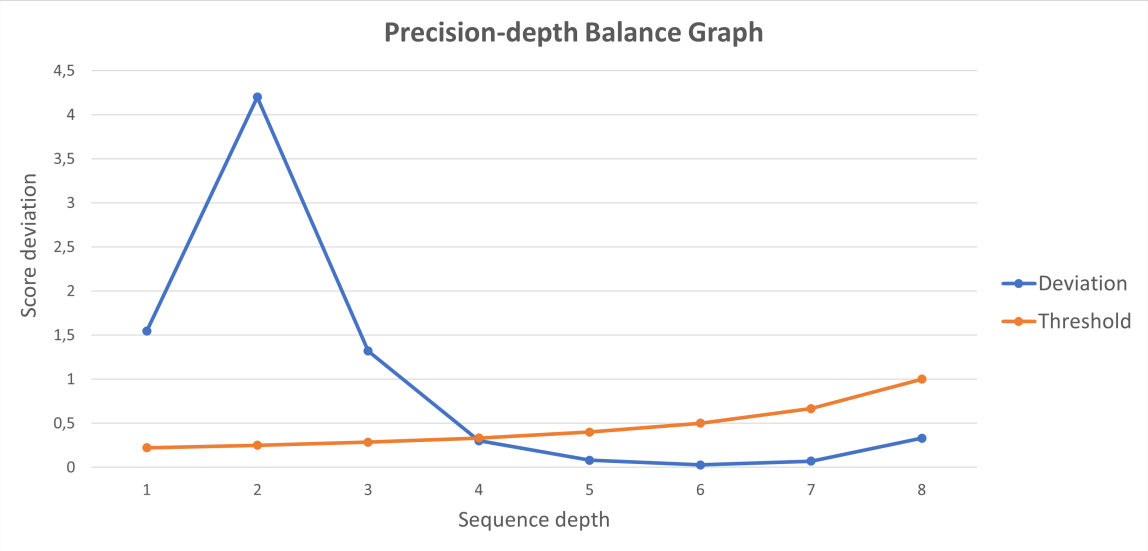

Figure 5.1: Graph showcasing the threshold to achieve balance between precision and depth of the move sequence. After 4 moves, the deviation barely passes the threshold so the corresponding position is used for the explanation.

This is implemented in a loop that calculates the deviation from the approximate search score and updates the threshold each iteration. This loop iterates over every move in the analysis up to the maximum number of moves, which is 8. The deviation is calculated as an absolute value from the difference between the search score and the static evaluation score from the current position. If the deviation is smaller than the threshold, the detailed evaluation of the position is saved with its index. If the loop is finished without passing the threshold, the position after the 8th move is saved. This is calculated for both analyses – for the best moves and user's moves continuation – and their combined deviation.

5.3 Explanation

This section describes the implementation of the most extensive part of this thesis and is implemented `explanation.cpp/explanation.h` files. So far, two analyses (see section 5.1) of move sequences were created, one for the best possible moves and one for the best possible moves after a move by user. Then, the most suitable positions from the analyses were found. These positions have as similar static evaluation scores to the estimated scores by the search function as possible. The positions are also as close as possible from the starting position in terms of simulated moves. The last remaining part is to explain the detailed evaluations (see section 5.1) from those positions.

Move rating First of all, the move is assigned a rating. The rating is a simple label, that summarizes the correctness of the user's move. It ranges from "BEST" – being the engine-selected move or very strong one – to "BLUNDER" – possibly a game losing move. The rating is assigned depending on a numerical value, which is calculated as the difference between the search score from the best moves analysis and the search score from the user's move analysis. If this value is small, it means that the user played a move which had

a similar evaluation after the best continuation and vice versa.

The explanation could be provided for every **factor** included in the detailed evaluation, but that would be too much for a human. Instead, only the **single most impactful factor** is explained. The most impactful means that its score changed the most from the best analysis to the user analysis. The implementation is in a loop that in each iteration, one factor score change is calculated. For example, for the pawns factor, it is calculated the following way:

$$change = BA.pawns[U] - UA.pawns[U]$$

Where BA is the best analysis and UA is the user's move analysis, U is the score for the color which the user is playing for.
In the equation, the scores for the opponent are not mentioned because the explanation concentrates on *what was done better* in the best analysis. This brings a potential problem, that the user's score could have not changed but the opponent could get better score. This problem is described in section 6.2.

Filtering There are some situations, that must be solved specifically. Sometimes, there is not much to explain as the reason is obvious from the analysis and is easily understandable by a human. If the best possible move led to a forced checkmate in the specified maximum number of moves, the explanation simply outputs this reality. If the user played a move which can still lead to a forced checkmate, this information is given. The other situation is if the user's move was the same as the one suggested by the engine. This is solved by simply outputting this information.

Now, the explanation of the most impactful factor can start. The implementation is in `evaluation.cpp` file as it contains the `Evaluation` class, which serves as the base for this part. A method `explain` is created and it requires one argument, which is an integer representing the factor. The output of this method is one string, which is created and edited by using `std::stringstream`. First, every bitboard and variable must be initialized. `Material` and `Pawns` classes contain specific hash tables, which are probed. Other initializations are for filling bitboards and other variables used later. Then, an explanation is done based on the factor. All these explanations are based on the existing code for the evaluation function from the engine and only those parts implemented in Stockfish are explained, nothing more, as they did not contribute to the evaluation score.

Factors

This part is devoted to **implementation of explanation** of the most impactful factor. These factors are implemented in the Stockfish engine. In this thesis, the code for factors is reverse engineered and every single part that affects the evaluation score is explained. The explanation is gradually built using C++ `stringstream`. When an appropriate part of a factor is found, a string is added to the stream. The explanation reasons are filtered as much as possible to provide only the most important features of the most impactful factor. The user should be able to understand the output simply by reading it and looking at the

corresponding position. Due to the code volume, this part is described in less detail and only the important and interesting parts are mentioned. Advanced chess terminology is used here and if needed, there is a list of most chess terms on Wikipedia in [36]. Also, a complete chess beginner might have difficulties understanding the described strategy.

Material Explaining material count is straightforward. A properly formatted list of each piece count for both colors is provided.

Piece square values The piece square table is used here. The score is assembled out of every piece value based on its position on the chess board. So, if the total score is positive, the user has the advantage. This score is disassembled to a value of every piece. If a piece has extremely high or low value, the information is added to the explanation.

Pawns Pawn structures and their evaluation is an extremely complex task. Its implementation is in different file `pawns.cpp`. A single pawn can have different features, which are important for its evaluation. Moreover, two or more pawns can form a structure. If this structure is good enough, it can increase the score significantly. First, for each pawn the features are determined, and the pawn is tagged accordingly. These features might be for example isolated, connected, or blocked, see Figure 5.2. For the implementation, bitboards and many bitwise operations among them were used. Also, if the pawn has some feature, a bitboard for keeping all the pawns is updated. The base program contains a table with scores for every feature. Finally, a feature that appears often and has a great impact on the score if found. An explanation is provided that some pawns on some squares have a feature that is important.

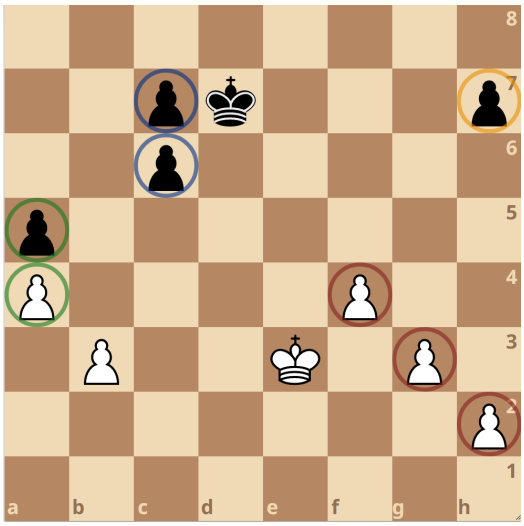

Figure 5.2: Example of a position with different pawns structures and features that can be explained. Pawns on **a4** and **a5** are blocked. Pawns on **c6** and **c7** are doubled. The pawn on **h7** is isolated. There is a strong pawn formation for white composed of three connected pawns – **f4**, **g3**, **h2**.

Knights The knight pieces in their specific evaluation have only a few score-affecting features. One of them is if they are providing a defence for the king. That means, their

possible moves can reach squares that are close to their king. Knight's specialty is standing on an outpost square which improves their score a lot. The knight evaluation features are illustrated in Figure 5.3. The outpost squares are detected by bitboards of own and enemy pawn attacks and the rank of these squares. The information if a knight is on an outpost or can reach it soon is added to the explanation. Knights also get a bonus for being shielded by a pawn.

Bishops Bishops are more difficult to evaluate. They share some features with knights such as providing defence or being on an outpost square. Other than that, their long range is evaluated. On one hand, if the diagonal of possible moves is full of own pawns, they get a penalty. On the other hand, if it attacks the center diagonal or the squares next to the enemy king, it gets a bonus. If a player has two Bishops, it is considered a strength. These features are provided in the explanation and demonstrated in Figure 5.3.

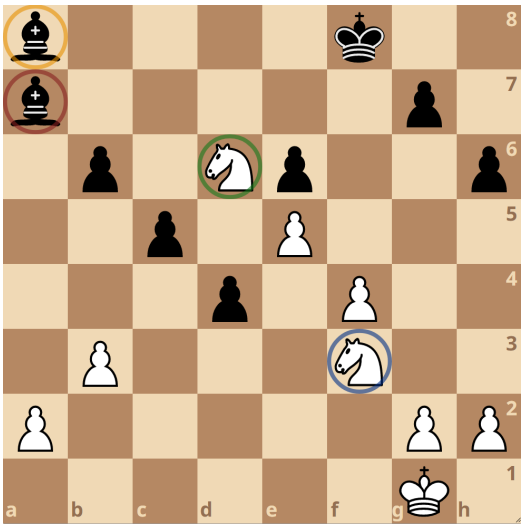

Figure 5.3: Example of a position with different bishops and knights evaluation features that can be explained. The knight on **d6** is on an outpost and the knight on **f3** is a great defender of the king. The bishop on **a8** is strong on a long diagonal. The bishop on **a7** is on a diagonal that is blocked by many friendly pawns and does not contribute as well. Black also has a bishop pair.

Rooks Rooks are specific for their long range and the possible domination of a file or rank. Therefore, an explanation is provided if a rook is on an open or semi-open file. A big problem occurs if the rook is blocked and cannot escape. The blockage can happen on a closed file by a friendly pawn blocked, by an enemy pawn, or on a friendly front rank blocked by the king that no longer has castling rights. These features are shown in Figure 5.4.

Queens Queens have a single evaluated feature in their specific function. This feature is that the queen piece can become weak if a bishop or a rook pins the piece protecting it. This is demonstrated in Figure 5.4. The calculation is done by using rook and bishop attacking bitboards of enemy pieces and checked if any friendly piece blocks the attack by occupying a square between.

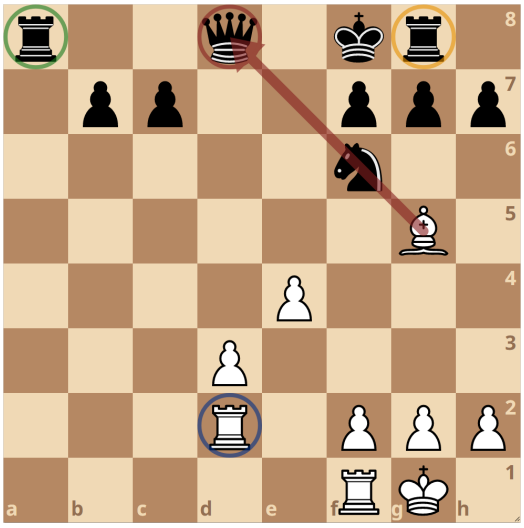

Figure 5.4: Example of a position with different rook and queen evaluation features that can be explained. The rook on **a8** controls the whole open **a** file. The rook on **d2** is on a semi-open file. The rook on **g8** is blocked behind the black king and can not properly join the game. The queen on **d8** is weak due to the pinned knight on **f6** by the white's bishop on **g5**.

Threats In addition to the piece-specific evaluations, there are more features that any piece can have. Threats are possible plans that carry an intention to damage the opponent's position. A threat can be simply a move that attacks one of the opponent's pieces with either an undefended piece or a higher valued piece. An example is if an opponent's weak piece can be attacked by a fork simultaneously with another, higher valued piece. Another example is a possible safe pawn push can be played to create more space or some enemy piece might become trapped. The example is illustrated in Figure 5.5. There are many more different threats that are explained. The threats are found by searching for the features on the squares using bitboards. For example, the feature of being a weak piece means that there is a piece on a square and that square is not defended by a pawn, and it is not attacked by more than one piece but the opponent attacks it at least once.

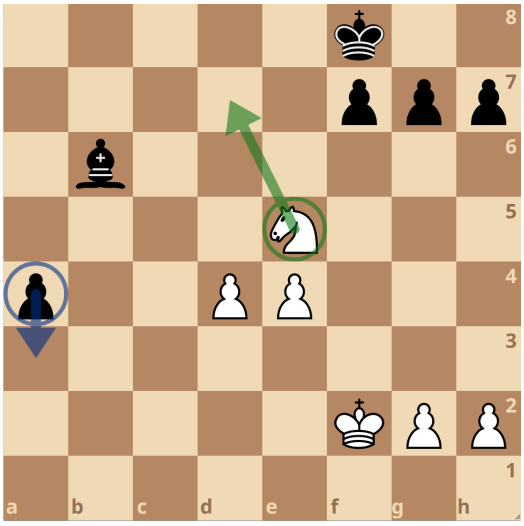

Figure 5.5: Example of a position with different threat evaluation features that can be explained. White threatens by moving the knight from **e5** to **d7** and attacking both the black's bishop and checking the black king. Black threatens by pushing the pawn from **a4** safely to **a3** and getting closer to the promotion rank.

King's safety Another extensive evaluation factor. The position and safety of the king is often very important for the evaluation. If the king is in danger and can be checked from every direction or an enemy pawn is approaching it, it is usually worse than having an extra pawn. The evaluation can be very different in the middle game and endgame. The explained features are for example the distance to friendly pawns, which is very important in the endgame. Another feature is about the strength of the king's shelter and the danger of approaching enemy pawns. This is calculated using predefined tables. The file of the king and both neighbor files are searched, and it is detected if any friendly pawns still exist in these files and if they did not move forward too much. For the approaching pawns, it is detected if they are too close or if they are possibly blocked by a friendly pawn. See Figure 5.6 for illustration. On some rare occasions, the approaching pawns can increase the safety of the king, if it blocks them and cannot be attacked. Then, the general safety is calculated by summing and multiplying various values, such as the attacker's count, possible checks, or subtracting a huge value if the opponent has no queen. The explanation is given based on this value.

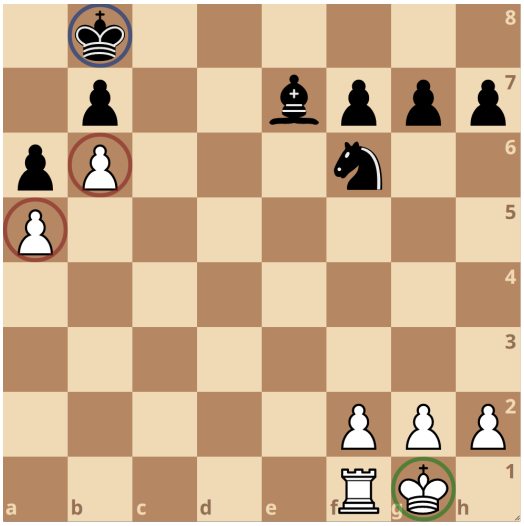

Figure 5.6: Example of a position with different king's safety evaluation features that can be explained. White king on **g1** has a very strong pawn shelter. Black king on **b8** has not so great shelter. The pawns on **a5** and **b6** are a blocked pawn storm that is approaching and endangering the black king.

Mobility Explaining mobility means simply stating that some piece types have many possible moves and therefore are more useful than those that are blocked. However, for each piece type, the number of possible moves for them to be good enough is different. There are tables for each piece type that contain scores for the number of possible moves that can the piece make. The explanation is not given for every single piece but the piece type instead. For example, if one knight can move 7 squares, it has a big score bonus and the second is completely blocked and has a penalty instead, the final explanation says nothing special about their mobility. The mobility is shown in Figure 5.7.

Passed pawns A special case for pawns that are not included in the pawns' evaluation is if a player has a passed pawn. This type of pawn is very strong and can often lead to bigger advantages. First, it is detected if a pawn is a passed pawn. Then, based on the safety of the remaining squares to reach the promotion square, an explanation is provided. The path safety is evaluated by checking the squares that are on the ranks between the promotion rank and the pawn's current rank. Also, the neighbor files are taken into consideration. The passed pawn is illustrated in Figure 5.7.

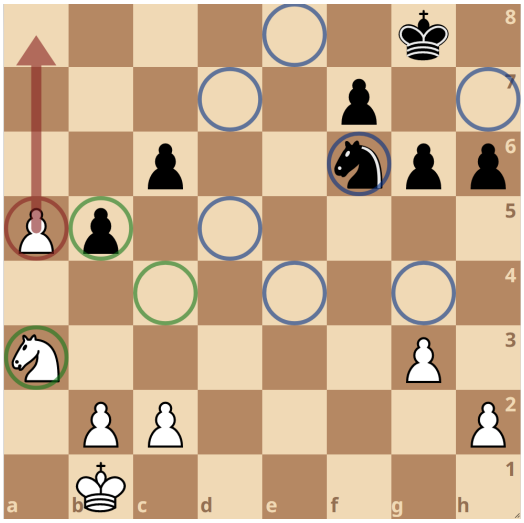

Figure 5.7: Example of a position illustrating the mobility of pieces and a passed pawn. The white knight on **a3** has only two squares to move to and is not positioned well. The black knight on **f6** has 7 move possibilities, which is good. The pawn on **a5** is a strong passed pawn with safe path to the promotion rank.

Space The space factor is used only in the opening and has the purpose to help the engine to take more space on the chess board. The explanation says how many central squares are under control or how many pawns are blocked.

A single factor is explained. One explanation is given for the best-found position after the best move sequence and one for the position after the user's move. The user now gets the following information.

5.4 Whole Game Explanation

For better user experience, it might be useful to know which moves in a game were influential on the game development and explained the reason behind them. The explanation for the whole game is implemented the following way. First, parsing of PGN was implemented. Second, every move is rated by the method described in the previous section. Last, the most important moves are explained as single moves, also explained in the previous section.

PGN Parsing

To implement the explanation for a whole game, all the moves must be loaded. The most common – and arguably the best – way to store information about a game is by using PGN as described in section 2.1. Mainly, the **movetext** section, which contains all moves, is important. First, the input is loaded, and then the moves in SAN (Standard Algebraic Notation) are parsed. The moves are then converted to LAN (Long Algebraic Notation).

Input Parsing The input parsing for this explanation is different. If the argument after the command `a` is not a move but a file name, then the mode for the whole game starts. The input file should contain a chess game in PGN. The file must have Unix-like CRLF line

endings. The **tag pair** rows are filtered, as they serve no purpose for this analysis. Then, the **movetext** part remains and must be parsed. For guaranteed correctness, the movetext should be without commentaries and notes. In a loop, every ply is extracted and stored to a vector. In the end, this vector contains all plies.

Simulating the Game

Now, the whole game must be simulated to rate every move. The moves in SAN must be converted to LAN (see section 2.1). For moves with pieces – knights, bishops, rooks, or queens – the conversion is very similar. For the king and the pawns, the process is slightly different. The first character in SAN determines which piece to move. Then, the last two characters determine the square where to move the piece. These two facts are parsed from the move in SAN.

Conversion for piece moves For piece moves, in method `gen_moves`, all possible moves of the corresponding piece type in the current position are generated in LAN. Out of these moves, only those which have the same ending square are saved to a vector. If the move represents capturing a piece, SAN contains "x" but nothing in this method changes, as still, the first character represents the piece and the last two represent the final square. There is one oddity that can be problematic. If the same piece type can move to the same square, it is differentiated in SAN. Then, the single correct move is determined in method `get_move`. This method loops all the found moves and returns the move with the largest number of matching characters between LAN and SAN of the move. An example:

If there is the first knight on `c3` and second on `d4` and the first one moves to `b5`, in SAN it is noted as `Ncb5`. The engine finds both knight moves which end on square `b5` and must be differentiated. In LAN the moves are noted `c3b5` and `d4b5` for the first and second knight, respectively. The first one has three character matches, whereas the second only two. Therefore, the first one is chosen as the correct one.

Conversion for king moves This conversion works very similarly to the piece moves. There is an exception with the **castling** moves. It is solved by statistically assigning the corresponding ending squares of the king after both the king-side castle (`0-0`) and queen-side castle (`0-0-0`).

Conversion for pawn moves Pawn moves are a little bit different. The moves are generated in the same way. The move with the same file as the pawn is chosen, as at most single pawn from a file can move to the specific square. The specific situation is if the pawn should promote in the next move. In that case, the last character from SAN represents the piece to promote to. The generated moves are always at least 4, for each possible promotion. The move with the last character same as the last character (in lowercase) is chosen. If the move is taking "en passant", this method still works as intended.

Rating all moves Then, all moves are rated by the same method described in the previous section. Every move is internally simulated and from the corresponding position, the rating of the player's move is calculated. The list of moves and their ratings is output.

Explaining the Most Important Moves

The five moves that had the biggest impact on the game are explained. The explanation of the moves is the same as for the single move explanation. The whole output is redirected from the standard output to a file. The file has the same name as the input file with added `.e` extension.

5.5 Other Implementation

Here, the rest and less interesting implementation parts are described. These parts are as important as the previously mentioned ones but they are not the core of this thesis.

Input parsing In the `analyse` function in the `uci.cpp` file, first, there is the completion of the input parsing. The `is` parameter contains the rest of the input string. The input token has to be followed by a string which represents a move in the long algebraic notation. The correctness of the move is then checked. The string must be either 4 or 5 characters long. The first two characters must specify a square that is occupied by a piece that is the same color as the color of the side to move. The second pair of characters then represents the square where the selected piece should move to. The last character is only accepted if the input string represents a pawn promotion move and the character represents the piece to which the pawn should be promoted. There are two possible ways to implement this, both of which are usable, but the latter one was finally chosen. One way could check if the input string for the move is correct. The characters are converted to a bitboard with one non-zero value which represents the square position. The piece, its color, and type are then recognised from the square bitboard and the bitboard with all pieces using operation AND. The move correctness depends on the piece type and its move possibilities. The possible moves can be checked using function `attack_bb` from `bitboard.h` file and passing the square, piece type, and a bitboard with all pieces. This was the first way how it was implemented. However, the second used version is more convenient and uses a reversed approach. All the legal moves in that position are generated using a structure `MoveList` from `movegen.h` file. Then in a loop, every legal move is transformed to its string representation in the same long algebraic notation and compared with the input string. If the input string matched any of the legal moves, it is considered correct and stored into a variable, if the input was a filename, the whole game analysis starts, otherwise, this function prints an error message and returns.

Perfect or great moves If the user's move is the same as the best found by the engine, the user only gets the information and no other explanation as it is not needed. If the user player a move that has close to equal value as the best one, the alternative is proposed. But often, the explanation does not give any special evaluation or it is the same for both compared positions. The string "There is no special explanation for this factor." is printed.

Checkmate moves If the search for the best moves found a checkmate sequence, the explanation is different. There are three possible different explanations. First, if the user played a move which was different but can result in a checkmate in the same number of moves, this information is provided. Second, if the move resulted in a sequence that is a few moves longer but also results in a checkmate, the explanation gives this information. Third, if the move missed the possible checkmate, the user is given a shortened explanation

without the comparison of the positions while the best move resulted in a checkmate. The checkmate detection is a little bit different, the score does not return a value in centipawns but a number in how many moves the player can force a checkmate.

Chapter 6

Experiments and Evaluation

The goal of this chapter is to describe performed experiments with the implemented extension. By the nature of the program, the experiments have been performed by a human. Based on the results of the experiments, evaluation of the whole program is provided.

6.1 Experiments

In this section, the performed experiments are described. These experiments and their results are explained. Unfortunately, there is no way to create automated tests as a human must evaluate the results. Therefore, only a limited number of experiments have been performed. In the real output, the positions are printed in formatted text. In this section, they are presented on a chess board for better representation.

6.1.1 Single Move Experiments

The most important part is the single move explanation. It is the core result of the whole thesis and must be tested the most. Many single-move experiments have been done and a few of them are illustrated in this part. Only the first experiment is described in detail with all steps.

The typical process of a user for the single move analysis is the following:

1. A user plays a game of chess.
2. During the game, the user calculates one move for a long time and then plays it.
3. After the game is finished, the user would like to know if the played move was good or bad.
4. The user runs the Stockfish engine with the extension implemented in this thesis to analyse the move.
5. The user loads the position and runs the command `a` with the move as an argument.
6. The program outputs the explanation.
7. The user reads the output and gets a better idea of the move they played.

Experiment 1

The first experiment shows the explanation in the opening phase of a game. The game started with a common opening, but already at the 5th move, the black player made the first mistake. The starting position of this experiment is in Figure 6.1.

Figure 6.1: The starting position of the first experiment. The evaluation is slightly better for white.

From this position, black played the move `5. ... c5`, which in LAN is `c7c5`. So first, the position by the fen was set in the Stockfish engine using command:

```
position fen rnbqkb1r/ppp2ppp/3p1n2/8/3NP3/2N5/PPP2PPP/R1BQKB1R b KQkq
- 2 5
```

The FEN string was gained by setting up the position in a free online chess GUI Lichess [4]. The position could also be loaded by using the `position fen moves M`, where `M` would be the list of the moves so far in this game. Then, the command to get the explanatory analysis was input:

```
a c7c5
```

After the analysis is complete, it gives the following output:

```
Best possible move:  f8e7
Best possible response:  c1f4
Best sequence of moves:  e8g8 d1d2 b8c6 e1c1 c6d4 d2d4 a7a6 f2f3 b7b5...
User's move:  c7c5
The best response to user's move:  f1b5
Best sequence of moves:  b8d7 d4f5 a7a6 b5e2 d7e5 c1g5 c8f5 g5f6 d8f6...
```

These are the analyses of the complete sequences of moves. Then the explanation starts:

```
Move rating:  MISTAKE
Reasoning:  King safety
After user's move and the continuation of the best moves:
c7c5 f1b5 b8d7 d4f5 a7a6 b5e2
The evaluation is:  -1.38
After the best move and the continuation:
f8e7 c1f4 e8g8 d1d2
The evaluation would be:  -0.37
The score loss is 1.01 points.
Where one point is equivalent to one pawn.
```

First, the **move is rated**. In this case, the move was not the best but neither very good. This move was a **MISTAKE** because it caused the evaluation score to drop a bit. The **most impactful factor** is the **King's safety**. Then, both sequences up to the ideal position are shown, the same as the whole static evaluation score of the positions. It is obvious, that the move by the user was a mistake because the evaluation is nearly one whole pawn worse than the evaluation after the best moves. Therefore, if the black player played the best move, they would keep the position quite equal. However, when they played the chosen move, they are in a worse position. Then, the first explained position is after the user's move and sequence illustrated in Figure 6.2. The explanation is the following:

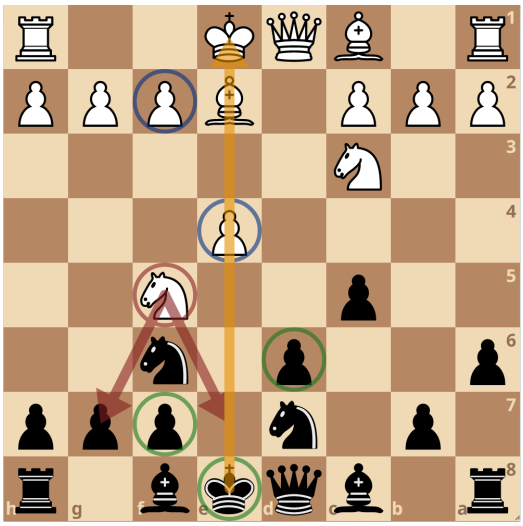

Figure 6.2: The position after the user's move and the continuation of the best moves. The black king still has a decently strong shelter (green). It is being approached by pawn formation (blue). The king is on a semi-open file (orange). There are a few strong attacks on the king's flank (red). Overall, the king is in danger.

```
Explaining factor King safety in the position:
Our king is close to our pawns.
Our king has a decent shelter.
Our king is being approached by an unblocked pawn formation,
which is dangerous.
Our king is on a semi-open file.
```

```
Our king is in danger.
Our king flank is under attack.
```

Whereas, the position after the best moves is in Figure 6.3 and the explanation looks like this:

Figure 6.3: The position after the best moves. The black king has a very strong shelter (green). It is far from the approaching pawns (blue). Overall, the king is not in danger.

```
Explaining factor King safety in the position:
Our king is close to our pawns.
Our king has a strong very strong shelter.
An unblocked pawn formation is approaching but is too far or not dangerous.
Our king is not in big danger.
```

By reading the explanation, the user can understand the positions better and compare the important factor between them. It is clear, that in the second position, the black king is in a better state. It has already castled, whereas from the first position the king can not even castle yet. Additionally, some pieces endanger the king in the first position. The user can conclude, that after the move they made, it could result in a position in which the king is not very safe and it could be if the very best move was played instead.

Experiment 2

The second experiment shows a typical big mistake in the middle game. The white player gained a great positional advantage in the opening and could even get a material advantage. However, the white player decided to play a move that looks very strong, threatening a checkmate in one move. Though, the checkmate could have been easily prevented by a black's great move with tempo and the opponent decreased the advantage a lot. The starting position of this experiment is in Figure 6.4.

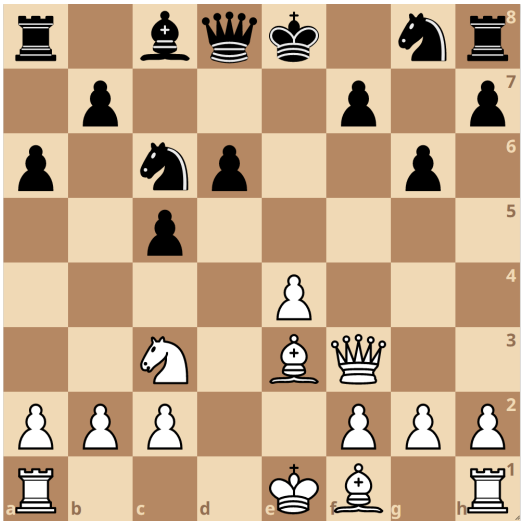

Figure 6.4: The starting position of the second experiment. The evaluation is a lot better for white. The FEN is `r1bqk1nr/1p3p1p/p1np2p1/2p5/4P3/2N1BQ2/PPP2PPP/R3KB1R w KQkq - 2 10`.

From this position, white played the move `10. Bc4` which in LAN is `f1c4`. This move threatens a checkmate in one with `11. Qf7`. The analysis gives the following output:

```
Best possible move: e1c1
Best possible response: c8e6
Best sequence of moves: e3c5 d8f6 d1d6 f6f3 g2f3 a8d8 d6d8 e8d8 c3d5...
User's move: f1c4
Best response to user's move: c6e5
Best sequence of moves: f3e2 g8f6 e1c1 b7b5 e3g5 h7h6 c4d5 a8a7 f2f4...
```

The explanation:

```
Move rating: BLUNDER
Reasoning: Material
After user's move and the continuation of the best moves:
f1c4 c6e5 f3e2 g8f6 e1c1 b7b5 e3g5 h7h6 c4d5 a8a7 f2f4 h6g5 f4e5 c8g4
The evaluation is: 0.92
After the best move and the continuation:
e1c1 c8e6 e3c5 d8f6 d1d6 f6f3 g2f3 a8d8 d6d8 e8d8 c3d5 g8e7
The evaluation would be: 3.96
The score loss is 3.04 points.
Where one point is equivalent to one pawn.
```

This move was a **BLUNDER** because the evaluation score dropped by three points which is as big as a bishop piece. The **most impactful factor** is the **Material**. The first explained position is after the user's move and sequence illustrated in Figure 6.5. The explanation is the following:

Figure 6.5: The position after user's move and the continuation of the best moves. The material is completely equal. White has no advantages in terms of the material. There is still an advantage for white in this position but black has some counter play (such as the threat from bishop on **g4**).

```
Explaining factor Material in the position:
Material (W - B):
Pawns:    7 - 7
Knights:  1 - 1
Bishops:  1 - 1
Rooks:    2 - 2
Queens:   1 - 1
```

Whereas, the position after the best moves is in Figure 6.6 and the explanation looks like this:

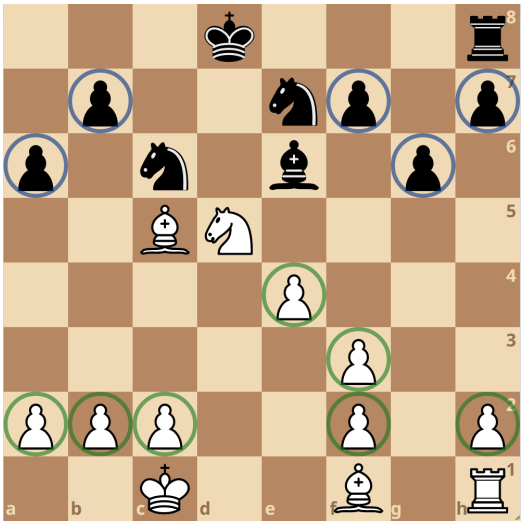

Figure 6.6: The position after the best moves. White has 2 more pawns which is a great material advantage. White also has a positional advantage, double bishop advantage and also a safer king. These factors are secondary and therefore not explained.

```
Explaining factor Material in the position:
Material (W - B):
Pawns:    7 - 5
Knights:  1 - 2
Bishops:  2 - 1
Rooks:    1 - 1
Queens:   0 - 0
```

By reading the explanation, the user can clearly understand the biggest difference between the two positions. After the user's move and the best continuation, the material is the same for both sides. However, after the best move and the sequence, white would gain an advantage of two whole pawns and also kept his two bishops. Even after the blunder, the white player still has some advantages in other factors but those are secondary and not explained. The user can conclude that a threatening move that looks great does not have to be that great if the opponent can easily block the threat. Instead of threatening checkmate, the white player should instead focus on gaining a small advantages gradually which would eventually lead to a win.

Experiment 3

The third experiment shows an explanation of a move in a later middle game. In the position, the white player is at a small disadvantage by having one pawn less and a worse mobility than the black player. The white player played a move that threatened the black queen. This move was a good one but still, it lead to a bad positioning of the white pieces. The starting position of this experiment is in Figure 6.7.

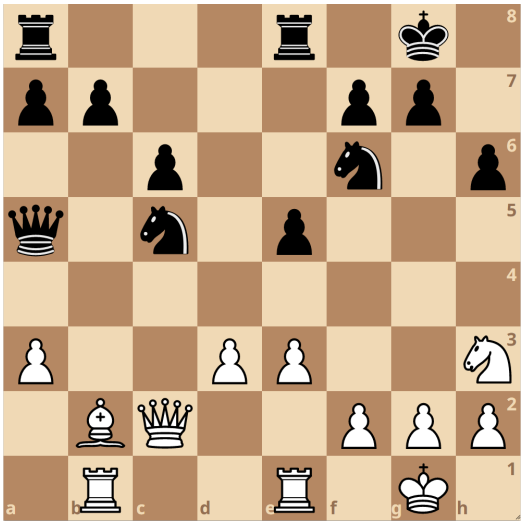

Figure 6.7: The starting position of the third experiment. The evaluation is slightly better for black. The FEN is `r3r1k1/pp3pp1/2p2n1p/q1n1p3/8/P2PP2N/1BQ2PPP/1R2R1K1 w - - 7 22`.

From this position, white played the move `22. Bc3` which in LAN is `b2c3`. This move threatens the black queen. The analysis gives the following output:

```
Best possible move:  d3d4
Best possible response:  e5d4
Best sequence of moves:  b2d4 c5d7 e1d1 a5c7 h3f4 b7b6 d4b2 c6c5 f4e2...
User's move:  b2c3
Best response to user's move:  a5a4
Best sequence of moves:  e1c1 a4c2 c1c2 a8d8 d3d4 c5a4 c3a1 f6g4 c2c1...
```

The explanation:

```
Move rating:  GREAT MOVE
Reasoning:  Psq
After user's move and the continuation of the best moves:
b2c3 a5a4 e1c1 a4c2 c1c2 a8d8 d3d4 c5a4 c3a1 f6g4
The evaluation is:  -1.29
After the best move and the continuation:
d3d4 e5d4 b2d4 c5d7 e1d1 a5c7 h3f4 b7b6
The evaluation would be:  -0.69
The score loss is 0.60 points.
Where one point is equivalent to one pawn.
```

This move was a **GREAT MOVE** because the evaluation score was only slightly. The **most impactful factor** is the **Psq** which are the positional bonuses for pieces. The first explained position is after the user's move and sequence illustrated in Figure 6.8. The explanation is the following:

Figure 6.8: The position after the user's move and the continuation of the best moves. The bishop on square **a1** is considered to be weak. It watches only one diagonal which can be often blocked (as it is here). The knight on square **h3** is very passive. Generally, a knight on the edges is considered bad (additionally, it can not move anywhere). The other three pieces in red circles are rated by the evaluation either positively or negatively. However, they have the same position as in the second position and therefore no impact on the score difference.

```
Explaining factor Psq in the position:
Bishop stands on a bad square a1.
Rook stands on a bad square b1.
Our king stands on a great square g1.
Pawn stands on a great square e3.
Knight stands on a very bad square h3.
```

Whereas, the position after the best moves is in Figure 6.9 and the explanation looks like this:

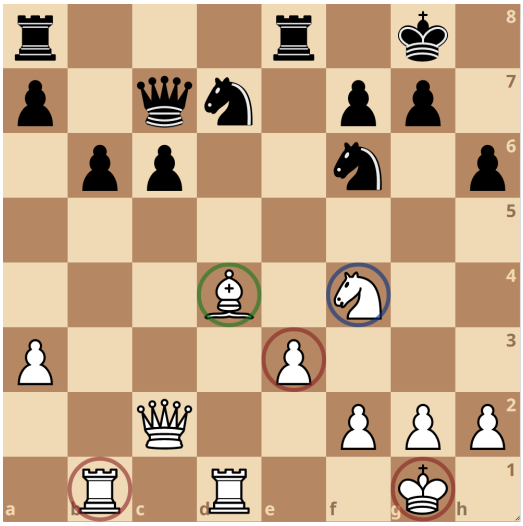

Figure 6.9: The position after the best moves. The bishop and knight are now on squares **d4** and **f4**, respectively. Both are considered great. The bishop attacks many squares and the knight is in the middle of board which is great.

```
Explaining factor Psq in the position:
Rook stands on a bad square b1.
Our king stands on a great square g1.
Pawn stands on a great square e3.
Bishop stands on a great square d4.
Knight stands on a great square f4.
```

From the explanation, it is clear that the positions differ in the positioning of the pieces for white. After the user's move and the best continuation, some pieces stand on good squares but some on very bad squares which results in a score penalty. However, after the best move and the sequence, white pieces stand on better squares. Mainly, the bishop and the knight are on very good squares and have a large impact on the game. The user can see that if the pieces are not on great squares, first, it is better to improve their positioning so they attack better squares. If the user tries to make threats while opponents have more active pieces, it can backfire.

Experiment 4

This is the 4th and final experiment for single move analysis. In this example, a move in position in the early endgame is analysed. In this position, there are only kings, pawns, and two other pieces for each player. The evaluation is almost equal with minimal advantage for black. The black player has one pawn more after a capture from the previous move. The white player is on the move and can re-capture the pawn back two ways. White made a wrong decision and captured the piece the wrong way. This move lead to a bigger advantage for the black player as the white pawns are in a worse spot. The starting position of this experiment is in Figure 6.10.

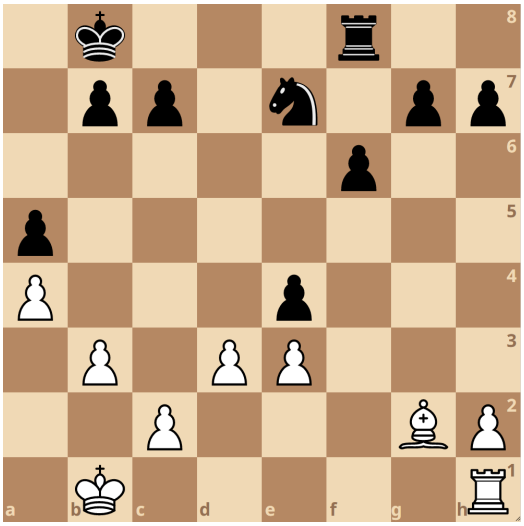

Figure 6.10: The starting position of the fourth experiment. The evaluation is almost equal. The FEN is `1k3r2/1pp1n1pp/5p2/p7/P3p3/1P1PP3/2P3BP/1K5R w - - 0 34`.

From this position, white played the move `34. dxe4` which in LAN is `d3e4`. This move captures a pawn. The analysis gives the following output:

```
Best possible move: g2e4
Best possible response: f6f5
Best sequence of moves: e4f3 g7g5 h1f1 c7c6 b1c1 b8c7 f1f2 e7d5 c1d2...
User's move: d3e4
Best response to user's move: e7c6
Best sequence of moves: g2f3 c6e5 f3e2 b8c8 h1d1 f8d8 d1d4 c7c5 d4d5...
```

The explanation:

```
Move rating: INACCURACY
Reasoning: Pawns
After user's move and the continuation of the best moves:
d3e4 e7c6 g2f3 c6e5 f3e2 b8c8
The evaluation is: -1.35
After the best move and the continuation:
g2e4 f6f5
The evaluation would be: -0.31
The score loss is 1.04 points.
Where one point is equivalent to one pawn.
```

This move was an **INACCURACY** because the evaluation score dropped by one pawn. The **most impactful factor** is the **Pawns** which represents the pawn structures and features. The first explained position is after the user's move and sequence illustrated in Figure 6.11. The explanation is the following:

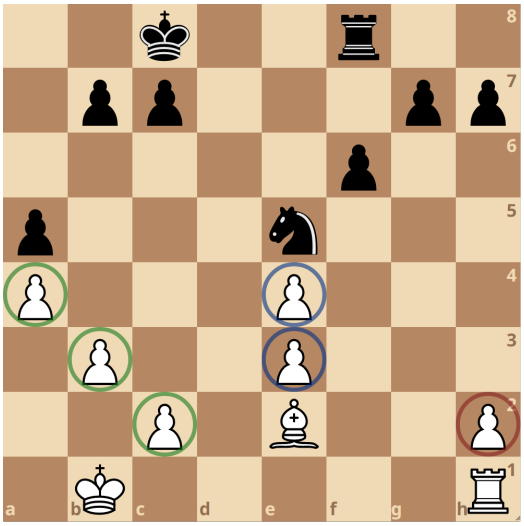

Figure 6.11: The position after the user's move and the continuation of the best moves. The pawn structure is not great. The main negatives are the doubled pawns on squares **e3** and **e4**. These pawns are also isolated (no friendly pawns on neighbor files), same as the pawn on **h2**. There is still a nice pawn formation from three connected pawns (green circle). The pawn on **a4** is blocked by a black's pawn.

```
Explaining factor Pawns in the position:
Pawns on squares c2 b3 a4 are connected.
Pawns on squares e3 e4 are doubled.
Pawns on squares h2 e3 e4 are isolated.
Pawn on square a4 is blocked.
```

Whereas, the position after the best moves is in Figure 6.12 and the explanation looks like this:

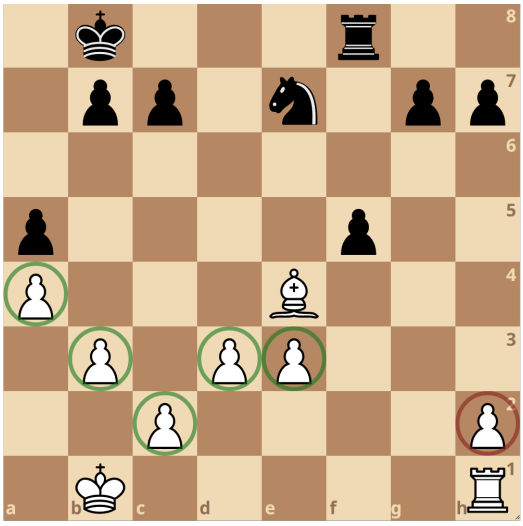

Figure 6.12: The position after the best moves. The pawn structure of 5 pawns is very strong (green circles). There is only one weakness at the isolated pawn on **h2**.

```
Explaining factor Pawns in the position:
Pawns on squares c2 b3 d3 e3 a4 are connected.
Pawn on square h2 is isolated.
Pawn on square a4 is blocked.
```

This explanation teaches the user that it is better to keep good pawn formations and if it is possible, avoid doubling pawns. Doubled pawns are quite bad and give a score penalty. After the best capture and the next moves, white keeps a strong pawn structure with a single weakness.

6.1.2 Whole Game Experiments

The whole game analysis was a secondary goal of this thesis and it is based on the single move analysis. It was tested briefly by analysing some games. Only a single experiment is described.

Experiment 1

The analysed game is the example game from PGN (see section 2.1). This game is saved in a file `game.pgn` in PGN. The user inputs this command:

```
a game.pgn
```

The analysis creates a new file called `game.pgn.e` and writes all the output into the file. The first part of the output is the evaluation of every move from the game which looks the following way (shortened):

```
Analysis of game provided in file game.pgn.
1| w (e2e4) - BEST MOVE | b (e7e5) - BEST MOVE
2| w (g1f3) - BEST MOVE | b (b8c6) - BEST MOVE
3| w (f1b5) - BEST MOVE | b (a7a6) - BEST MOVE
...
11| w (c3c4) - GREAT MOVE | b (c7c6) - BEST MOVE
12| w (c4b5) - INACCURACY | b (a6b5) - BEST MOVE
13| w (b1c3) - BEST MOVE | b (c8b7) - BEST MOVE
14| w (c1g5) - INACCURACY | b (b5b4) - BEST MOVE
15| w (c3b1) - BEST MOVE | b (h7h6) - BEST MOVE
...
18| w (h4e7) - BEST MOVE | b (d8e7) - BEST MOVE
19| w (e5d6) - INACCURACY | b (e7f6) - BEST MOVE
20| w (b1d2) - BEST MOVE | b (e4d6) - BEST MOVE
21| w (d2c4) - INACCURACY | b (d6c4) - BEST MOVE
22| w (b3c4) - BEST MOVE | b (d7b6) - GREAT MOVE
...
26| w (d1e1) - BEST MOVE | b (g8f7) - GREAT MOVE
27| w (e1e3) - GREAT MOVE | b (f6g5) - INACCURACY
28| w (e3g5) - BEST MOVE | b (h6g5) - BEST MOVE
...
```

```
41| w (d6a6) - GREAT MOVE | b (h3f2) - BEST MOVE
42| w (g3g4) - BEST MOVE | b (f5d3) - BEST MOVE
43| w (a6e6) - GREAT MOVE
```

The user now gets the overview of the game. It seems that the game was played by very good players. No large mistakes were done and only a few inaccurate moves were played. That is true, this game was played by two grandmasters. The file then continues and prints the explanations of the 5 moves that had the biggest impact on the evaluation. In this example, the worst moves were **12th**, **14th**, **19th** and **21st** for white. Only one move is explained for black – **27th**.

6.2 Evaluation

In this section, the extension is evaluated based on the experiments. The author is only an intermediate chess player, so the explanation is made in a way that is understandable for the author. Thus, the evaluation is subjective, based on the author's experience. For complete beginners, the detailed explanations are probably not simple to understand but even they can see some information about the positions and move sequences. For advanced players, the explanation is probably trivial but they still can see the two positions after the move sequences and see the differences themselves.

Overall, the extension works as intended. The user gets an explanation either of one move from a certain position or of the whole game. This explanation provides important information about the move or game. The information gives the user an idea of how good the move was in comparison with the best possible move. Additionally, detailed explanations of the two positions are provided. The first one is the position at the end of the continuation after the user's move. The second one is the position at the end of the best possible move sequence. These position explanations give feedback based on which the user can understand the score difference.

Only **one factor** explanation is provided, but for two positions. The user gets the idea of how good or bad the move was by a rating. Sees what the best possible move is and how the game should continue if the best moves were played. Also sees how the game would continue after their move. Then first, the position after the best moves is displayed and the explanation of the most impactful factor is given. And finally, the position after their move is displayed and an explanation of the same factor is shown. The user can now simply compare explanations and understand why the move of the engine was better and their move was not that good.

The focus of this explanation was the middle game. This part of the game is the most difficult as there usually are the most possible candidate moves to choose from. In the opening phase, the explanation is still solid but it would be much better for the user to rather learn the opening theories from books. For the endgame phase, the explanation is often not that great. These positions often are evaluated in special functions which were not explained in this thesis. For example, if there is a king versus king and pawn, there again many theories which should be studied to understand them. It would be close to impossible to explain all of these in detail.

The best explanations On one hand, if the user's move was very bad, the most impactful factor was very often the material. That means, that after the user's move, if the opponent responded with the best move, the user would lose some material, e.g., a bishop. In this case, the explanation helps only by suggesting the optimal move and that after the user's move, the player will get to a material disadvantage. On the other hand, if the move was close to perfect but not the best, the explanation could help but primarily the strong players. It only suggests the very best move and no further explanation is needed. Therefore, the most interesting explanations are provided if the move was not terrible but not the best one.

Simplicity The explanation is straightforward. It gives only the most important reason for the score difference. For example, it could explain that one position has nothing spectacular and that in the second position, the rooks have a good mobility. It is then up to the user if it is enough to understand the position better.

Diversity of results The search is non-deterministic. That means, that multiple searches with identical parameters can return slightly different scores, same as different move sequences. This deviation in the move sequence happens rarely in the first few moves, but the chance increases in the later moves. The reason is that there can be positions with more great moves. This is not necessarily a problem, but it might cause inconvenience while trying to replicate the same moves and not getting the same results.

Further explanation If the sequences suggested by the explanation contain moves, that the user does not understand, they can be explained further. If the user thinks that a different move would be better, they simply run the explanation again in the corresponding position and get a new explanation.

Games by masters One of the main problems is that if the games are played by masters almost perfectly, sometimes, the lower analysis depth might rate the moves wrong. For example, the PGN example game in section 2.1, is played almost perfectly. Some of the moves played by the players are even better than moves that are found by the engine, which has limited search depth. Therefore, this explanation is not very useful for the best chess players.

Explanation notes Some rare situations can occur in the explanation. These situations are marked by a `NOTE`. For example, positions, where the best move leads to a forced stalemate, are denoted by this fact. Another example, the explanation of the most impactful factor might not be always great. Sometimes, the user can lose score not by getting a worse evaluation but by letting the opponent get better. Then, the explained factor for both positions might not be that helpful and so this is also denoted. The next example is that sometimes the dynamic search evaluation might include long-term bonuses and the static evaluation does not see them. There are more similar notes for similar situations and often the explanation is not great in them.

Whole game explanation The explanation of the whole game can be either very beneficial or quite useless. If the analysed game is full of completely bad moves, the useful part of the explanation is only the rating of the moves. The explanation of a few moves with the

largest influence on the score will almost always be that the user could have lost material. However, if the game was a decent one with some bigger mistakes, the extension detects them and explains the reasons. The solution to this problem would be simple but this was a secondary task of the extension and the main focus was on the single move one.

Chapter 7

Conclusion

The aim of this thesis was to design and implement an explanatory chess analysis. As there is a large number of existing open-source chess engines and creating a new one would be redundant, this analysis was implemented as an extension for one of them. The thesis started by describing the theory and principles of a chess engine. Then, the most fitting chess engine for this purpose was selected after a thorough comparison. The selected one was the Stockfish chess engine. The functionality and code structure of Stockfish was described and the extension was designed and implemented. Lastly, multiple experiments with the extension were performed and the results were evaluated.

Generally, the implemented extension provides a great explanation of the moves and also of whole chess games. In short, the analysis provides an explanation of reasons that affect the evaluation of chess positions. The explanation might be useful for beginners, intermediate, and also for great chess players. Everything depends on the user's interpretation of the results. The explanation could be improved in the future to get even better results.

The first improvement suggestion is that the explanation could provide a comparison. The extension now just explains the positions and leaves the comparison up to the user. It might be convenient and help to understand the explanation. For example, if the material was different in both positions, it could compare them and output only the differences instead of a whole list of pieces for both.

Another possible future work could improve the explanation in the endgame phase. This thesis primarily focuses on explaining the middle game as explaining the endgame often is close to impossible. Usually, a theory must be known by the player to perform well in that phase. Stockfish usually has predefined best moves or the tablebases are used for calculating the best moves. However, the evaluation scores of the factors differ based on the game phase and that could be utilized to get better results.

Bibliography

[1] Ahle, T. D. *Sunfish* [online]. GitHub, 2021 [cit. 2021-12-29]. Available at: https://github.com/thomasahle/sunfish.

[2] Beal, D. F. A generalised quiescence search algorithm. *Artificial Intelligence*. 1990, vol. 43, no. 1, p. 85–98. DOI: 10.1016/0004-3702(90)90072-8.

[3] Chess.com. *Game Review now available for all Chess.com members* [online]. Chess.com, Nov 2021 [cit. 2022-05-15]. Available at: https://www.chess.com/news/view/chesscom-releases-new-game-review.

[4] Duplessis, T. *The best free, adless chess server* [online]. 2010 [cit. 2022-05-15]. Available at: https://lichess.org/.

[5] Edwards, S. *Portable Game Notation Specification and Implementation Guide* [online]. 1994 [cit. 2021-12-18]. Available at: https://www.thechessdrum.net/PGN_Reference.txt.

[6] FIDE. *FIDE laws of chess* [online]. Jul 2009 [cit. 2022-05-15]. Available at: https://www.fide.com/FIDE/handbook/LawsOfChess.pdf.

[7] Fiekas, N. *Syzygy Endgame tablebases* [online]. [cit. 2021-12-26]. Available at: https://syzygy-tables.info/.

[8] Frey, P. W. *Chess skill in man and machine*. 2nd ed. Springer, 1984. ISBN 978-0-387-90815-1.

[9] Heinz, E. A. How DarkThought Plays Chess. *Scalable Search in Computer Chess*. 2000, p. 185–198. DOI: 10.1007/978-3-322-90178-1_13.

[10] Homan, D. *Lazy SMP, part 2* [online]. Jan 2012 [cit. 2022-05-15]. Available at: http://talkchess.com/forum/viewtopic.php?t=46858.

[11] Hyatt, R. *Chess program board representations* [online]. 2013 [cit. 2021-12-16]. Available at: https://web.archive.org/web/20130212063528/http://www.cis.uab.edu/hyatt/boardrep.html.

[12] Ionita, N. *Barbarossa* [online]. GitHub, 2021 [cit. 2021-12-29]. Available at: https://github.com/nionita/Barbarossa.

[13] Klein, D. *Neural Networks for Chess* [online]. 2021 [cit. 2021-12-21]. Available at: https://github.com/asdfjkl/neural_network_chess/.

[14] LeelaChessZero. *Lc0* [online]. GitHub, 2021 [cit. 2021-12-29]. Available at: https://github.com/LeelaChessZero/lc0.

[15] Marsland, T. A. The Anatomy of Chess Programs. In: *Proceedings of the 4th AAAI Conference on Deep Blue Versus Kasparov: The Significance for Artificial Intelligence*. AAAI Press, 1997, p. 24–26. AAAIWS'97-04. DOI: 10.5555/2908791.2908797.

[16] Matthies, A. *RubiChess* [online]. GitHub, 2022 [cit. 2022-01-04]. Available at: https://github.com/Matthies/RubiChess.

[17] McIlroy Young, R., Sen, S., Kleinberg, J. and Anderson, A. Aligning superhuman AI with human behavior. *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2020. DOI: 10.1145/3394486.3403219.

[18] Nasu, Y. *Efficiently Updatable Neural-Network-based Evaluation Function for computer Shogi* [online]. 2018 [cit. 2022-05-15]. Available at: https://github.com/ynasu87/nnue/blob/master/docs/nnue.pdf.

[19] Official Stockfish. *Stockfish* [online]. GitHub, 2022 [cit. 2022-01-04]. Available at: https://github.com/official-stockfish/Stockfish.

[20] Oshri, B. and Khandwala, N. *ConvChess* [online]. GitHub, 2015 [cit. 2021-12-29]. Available at: https://github.com/BarakOshri/ConvChess.

[21] Østensen, E. F. *A Complete Chess Engine Parallelized Using Lazy SMP*. 2016. Master's thesis. University of Oslo.

[22] Radaelli, P. *Smarter Chess Analysis - start decoding for free* [online]. Jan 2022 [cit. 2022-05-15]. Available at: https://decodechess.com/.

[23] Rasmussen, D. R. *Parallel Chess Searching and Bitboards*. 2004. Master's thesis. Technical University of Denmark.

[24] Rosenthal, J. *Winter* [online]. GitHub, 2021 [cit. 2021-12-29]. Available at: https://github.com/rosenthj/Winter.

[25] Russell, S. J. and Norvig, P. *Artificial Intelligence: A modern approach*. 3rd ed. Prentice-Hall, 2010. ISBN 978-0136042594.

[26] Schadd, M. and Winands, M. Quiescence Search for Stratego. In:. October 2009.

[27] Schrüfer, G. A strategic quiescence search. *ICGA Journal*. 1989, vol. 12, no. 1, p. 3–9. DOI: 10.3233/icg-1989-12102.

[28] Shannon, C. E. Programming a computer for playing chess. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*. Taylor Francis. 1950, vol. 41, no. 314, p. 256–275. DOI: 10.1080/14786445008521796.

[29] Shapiro, S. E. *Encyclopedia of Artificial Intelligence*. John Wiley amp; Sons, 1987.

[30] Shcherbyna, V. *Igel* [online]. GitHub, 2021 [cit. 2021-12-29]. Available at: https://github.com/vshcherbyna/igel.

[31] Topchiyski, K. *Bagatur* [online]. GitHub, 2022 [cit. 2022-01-04]. Available at: https://github.com/bagaturchess/Bagatur.

[32] Wiki. *Cray Blitz* [online]. Chess Programming Wiki, 2020 [cit. 2021-12-21]. Available at: https://www.chessprogramming.org/Search.

[33] Wiki. *Engines* [online]. Chess Programming Wiki, 2021 [cit. 2021-12-21]. Available at: https://www.chessprogramming.org/Engines.

[34] Wiki. *Search* [online]. Chess Programming Wiki, 2021 [cit. 2021-12-21]. Available at: https://www.chessprogramming.org/Search.

[35] Wiki. *UCI* [online]. Chess Programming Wiki, 2021 [cit. 2021-12-21]. Available at: https://www.chessprogramming.org/UCI.

[36] Wikipedia. *Glossary of chess* [online]. Wikimedia Foundation, May 2022 [cit. 2021-04-10]. Available at: https://en.wikipedia.org/wiki/Glossary_of_chess.

[37] Winands, M. H., Werf, E. C. van der, Herik, H. J. van den and Uiterwijk, J. W. The relative history heuristic. *Computers and Games*. 2006, p. 262–272. DOI: 10.1007/11674399_18.

Appendix A

Contents of the Memory Media

The attached media contains the following items:

- `DT-xhertl04.pdf`
  - This document in PDF format.
- `src/`
  - The source code of the Stockfish chess engine and the extension implemented for this thesis.
- `text/`
  - Folder with LaTeX source files.
- `compiled/`
  - Folder with the compiled program for 64bit Windows OS.
- `README.txt`
  - File containing an installation and user manual.