Kai Tanna-Shah

Creating an Interpretable Chess Engine

Computer Science Tripos - Part II Dissertation

Churchill College

University of Cambridge

2023-2024

Declaration of Originality

I, Kai Tanna-Shah of Churchill College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose. In preparation of this dissertation I did not use text from AI-assisted platforms generating natural language answers to user queries, including but not limited to ChatGPT. I am content for my dissertation to be made available to the students and staff of the University.

Signed: Kai Tanna-Shah

Date: May 9, 2024

Proforma

Candidate Number:	**2358G**
Project Title:	**Creating an Interpretable Chess Engine**
Examination:	**Computer Science Tripos - Part II, 2023-2024**
Word Count:	**10090**[1]
Code Line Count:	**2307**
Project Originator:	**The candidate**
Project Supervisor:	**David Khachaturov**

Original Aims of the Project

The project aimed to create a program which can evaluate chess positions and play chess against other programs (chess engines) or human opponents. I planned to experiment with which features of a chess position are most effective for determining the best move. I aimed to provide explanations for the best moves and evaluations that the engine generates. The engine should use standard search techniques used by other chess engines to reach an ELO of 1500 according to the CCRL ratings [1].

Work Completed

I successfully developed a chess engine which met the original ELO target. I implemented and evaluated standard search and optimisation techniques used by other chess engines. I designed a visualisation for the explanations that are provided to the user.

Special Difficulties

Due to time constraints, I was unable to conduct a user study to evaluate whether the explanations were useful to chess players. As such, the project focused more on evaluating the impact that using various features and techniques had on the performance of the engine.

[1] This word count was computed using texcount.

Contents

List of Figures

Chapter 1

Introduction

In this chapter, I outline my motivation for the project, including a summary of previous work in the area. I then define my initial objectives for the project as well as what was subsequently achieved.

1.1 Motivation

Chess engines have long been an area where new artificial intelligence techniques have been tried and tested against human opponents. It is a well-known game that is often regarded as a sign of intelligence and logical thinking, so when DeepBlue became the first chess engine to beat a world champion in 1997 it was a huge achievement in the field of AI [2]. Since then, the skill level of the top chess engines has rapidly outpaced the best human players in the world. Nowadays, elite chess players use chess engines as part of their training and preparation, treating the engine output as a source of truth [3].

In recent years, there has been a large influx of beginner and intermediate chess players to online chess websites such as lichess and chess.com [4]. These websites have built-in tools which allow users to analyze their games using chess engines. The tools can tell players where they made the mistakes according to the engine, as well as what the best sequences of moves are in a given position. Players can use these tools to identify weaknesses in their game and learn from their previous mistakes.

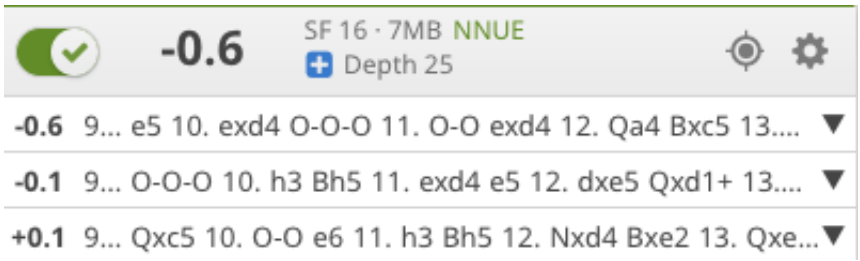


Figure 1.1: The lichess user interface to show the engine evaluation of the current position and the best sequences of moves from the current position

The problem is that **users are not able to see why the chess engine has evaluated a position a certain way**, and why the move it suggests is better than the others. This project investigates how engines evaluate a position and create an engine which provides reasoning as well as evaluations. The project will also be open-source, so that it has more transparency than the existing tools.

1.2 Previous Work

Traditionally, chess engines have used handcrafted evaluation functions based on heuristics from expert chess players. However, in recent years many chess engines have transitioned to using neural networks [5]. This has improved their performance but makes them even less interpretable.

During my research for this project, I found two sets of recently released tools which provide explanations alongside the engine's suggested moves. One of these is DecodeChess [6], a paid tool which explains the moves of the Stockfish chess engine. The other is the new Chess.com game review [7], which is available with a paid subscription. The previous work in this area is closed source, and they have not released a detailed description of how the explanations are generated. Therefore, this project will work from a fresh starting point, without trying to reproduce the methods used in these previous approaches.

1.3 Project Objectives

I break down the overall goal of helping players to see why the chess engine makes certain decisions into some more defined objectives. I outline both the core objectives of this project and the extensions I completed in the remaining time.

Core

1. The chess engine should be able to produce evaluations of all legal chess positions, as well as provide tactical and strategical aspects that contributed to that evaluation.

2. The engine should be able to correctly classify 80% of test positions as equal, winning for white or winning for black according to Stockfish.

3. I will test the chess engine by playing matches against other engines of a known strength. The engine should have an estimated Elo of above 1500.

Extensions

1. I implemented search techniques to improve the performance of the chess engine. This had a measurable increase in the estimated Elo rating.

2. I originally planned to use a large language model to generate natural language explanations. However, studies show that visualisations improve understanding compared to verbal explanations [8]. Therefore, I designed and implemented a visualisation for the explanations instead.

Chapter 2

Preparation

In this chapter, I discuss the standard structure of chess engines, as well as details about the algorithms that are commonly used. I then outline the requirements analysis and software development techniques that were used to implement the chess engine.

2.1 Starting Point

I started the project with no prior experience in programming a chess engine, or an engine for any other similar games. The minimax algorithm and the use of alpha-beta pruning to reduce the number of nodes searched were covered in the IB Artificial Intelligence course. I would consider myself an intermediate chess player because I have never played competitively, but I have played chess online for the past few years.

2.2 Research on Chess Engines

I used the Chess Programming Wiki [9] as the starting point for my research. It consists of articles describing many of the algorithms used by the most common chess engines.

2.2.1 Board representation

The internal board representation is the core data structure that the engine maintains, so it was one of the most important design decisions to make. The other board representations are also one of the primary ways of communicating with other programs.

The Forsyth–Edwards Notation (FEN) is the most widespread notation used to represent a chess position [10]. It encodes any chess position into a unique single line of text, consisting of six different fields. The first field represents the position of each of the pieces, with a different character for each type of piece, and numbers for empty spaces. The remaining fields are used for the active colour, castling rights, en passant availability, half-move clock and full-move number. The half-move clock (total number of moves from either player since the last capture or pawn move) is required for the 50-move rule which means the game is a draw if this clock reaches 100 [11].

The program will use FEN notation for loading chess positions as well as if it needs to output a position. However, representing a position in a string is not well suited to identifying which moves can be played or for making moves. Instead, I researched and decided on a different internal representation to use. This was important to consider because it impacted the implementation of many different aspects of the chess engine. A representation which allows efficient generation of moves, application of moves and evaluation of the position is crucial.

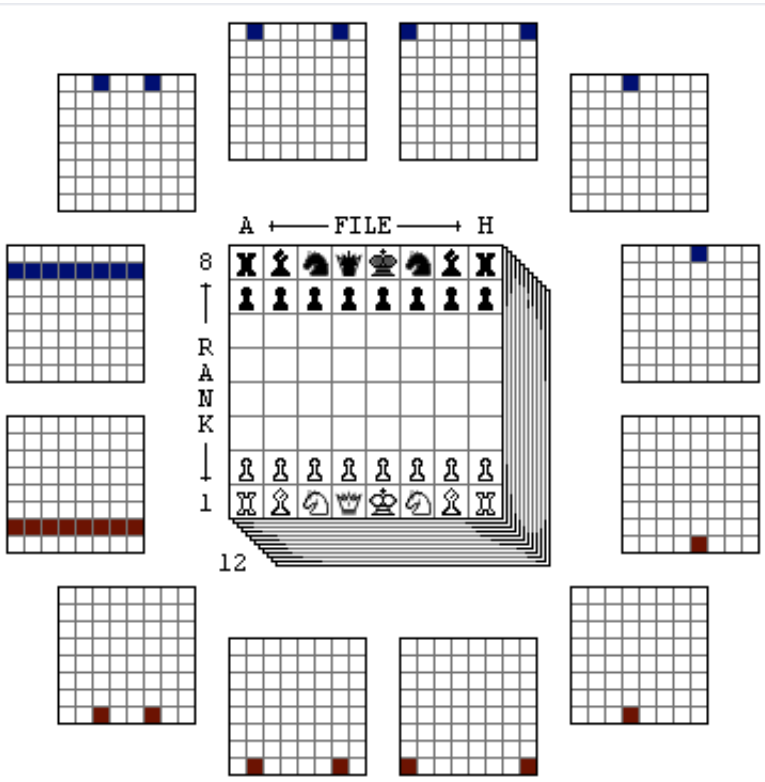


Figure 2.1: A diagram showing the bitboards corresponding to the starting position of a chess game

There are two common approaches to board representation, using either piece-centric *bitboards* or a square-centric array. The bitboard representation uses 64-bit arrays (bitboards) for each of the types of pieces. Each bit is a 1 if there is a piece on the corresponding square on the 8x8 board and 0 otherwise. Figure 2.1 shows what the bitboards would be initialised to at the start of the game.

On the other hand, a square-centric array to represent the board would consist of a single array with 64 elements. Each element would correspond to a square on the board and have a different code for either being empty or the piece that is on the square.

I decided to use the bitboard piece-centric approach since each piece was used differently in the move generation and application functions. The bitboard approach also allowed quick element-wise operations, for example, to generate bitboards for all of the pieces of a certain colour.

As well as storing the positions of all of the pieces on the board, a complete board representation was needed to maintain other information, implement certain special moves and draw conditions [11]. These include a record of the castling rights of each player, the number of half-moves since the last capture or pawn move and where en passant can be played.

2.2.2 Move generation

The move generation is completed in two stages, firstly a list of pseudo-legal moves is generated. In the second stage, the list is filtered to only the moves which do not leave the king in check. The pseudo-legal move generation considers the following FIDE rules [11]:

- Piece movement rules
- Captures

- En passant
- Castling rights

The move generator also needs to encode each move in a way that is uniquely identifiable for the starting position so that each move can be applied to the board representation. One approach to encoding moves is the algebraic notation [12], which is used in the Portable Game Notation (PGN). It is the most commonly used notation for recording games, however, it is optimised to be human readable. In contrast, from-to notation encodes a move with the index of the starting position of the piece and the index of the ending position. An additional code is also used to specify the type of move. Although this code is not required to uniquely identify each move, it reduces the time required to apply and unmake each move.

Code	Move type
0	Quiet move
1	Double pawn push
2	Castle king-side
3	Castle queen-side
4	Capture
5	En passant capture
8	Promote to knight
9	Promote to bishop
10	Promote to rook
11	Promote to queen
12	Promote to knight & capture
13	Promote to bishop & capture
14	Promote to rook & capture
15	Promote to queen & capture

Figure 2.2: A table showing the codes for each type of move [13]

These move encodings allow promotions and captures to be easily identified and applied. However, to unmake captures, the type of piece captured must either be included in the move encoding or a record must be kept in the board representation.

2.2.3 Communication protocol

The engine needs to play matches against other chess engines of known strength to evaluate its performance. There are two widespread chess communication interfaces which are implemented by a lot of other chess engines. One of these is the Universal Chess Interface (UCI) and the other is the Chess Engine Communication Protocol (CECP), used by XBoard. Implementing either of these allows the chess engine to play automated matches against other engines and provides a Graphical User Interface (GUI) to watch the games and play against the engine as a human.

The main difference between the two is that UCI takes away some of the decision-making regarding how long to search for the best move away from the engine. Instead, the GUI makes these decisions for the engine [14]. XBoard also uses long algebraic notation, which includes the algebraic starting position of the piece as well as the destination square. I used the XBoard communication protocol because I think that time management is an important aspect of playing chess that should be controlled by the engine. The XBoard GUI when two engines are playing against each other is shown in Figure 2.3.

Figure 2.3: A screenshot of the XBoard GUI

The XBoard GUI communicates with the engine using UNIX pipes, so the engine receives commands on its standard input, and output responses on its standard output. There are some initial commands and responses to tell the engine that a new game has started. While the chess game is being played, the GUI communicates the amount of time left for each player, as well as relaying the moves played by the opponent. The engine sends back the moves that it is playing to the GUI [15].

2.2.4 Timing

Classical chess games between humans are played over many hours, but games between engines are usually played in a shorter amount of time. There are many time controls used in both human and computer chess, with varying amounts of time to play the entire game or for a set number of moves for example. Time controls without any increment force the engines to estimate the number of moves remaining in the match to effectively manage time. The chess engine is able to manage its time effectively with any time control, so I arbitrarily chose a standard time control of 40/5 to test my chess engine against others with. This is a commonly used time control, which is also the default in the XBoard GUI. This time control means that each engine starts with 5 minutes on its clock, and each clock counts down while the respective engine is deciding its next move. Every 40 moves which each chess engine makes adds another 5 minutes to the clock. This means that the engine can use 7.5 seconds per move without running out of time before the increment.

2.2.5 Search

The goal of the chess engine is to find the best move of the possible options from a given position. The current position will be the root of the search tree, with the position after each move being a child node of the current position. Under the assumption that the opponent will play optimally, the goal of the engine is to find the path down the tree which corresponds to the best sequence of moves. Chess is a zero-sum game, so the best move for us is the worst move for our opponent and vice-versa.

If we assign each node in the tree with a value for how good the position is for us, then we are trying to maximise the value, while our opponent is aiming to minimise it. This approach describes the minimax algorithm, which is shown in Figure 2.4. With small games, such as tic-tac-toe, the leaf nodes can be assigned a value based on the result of the game, however, chess games last far too many moves, with many possible moves at each layer for this approach to be tractable. Instead, the minimax search must be performed up to a certain depth, with a heuristic evaluation of the position being used as the value for nodes of this depth.

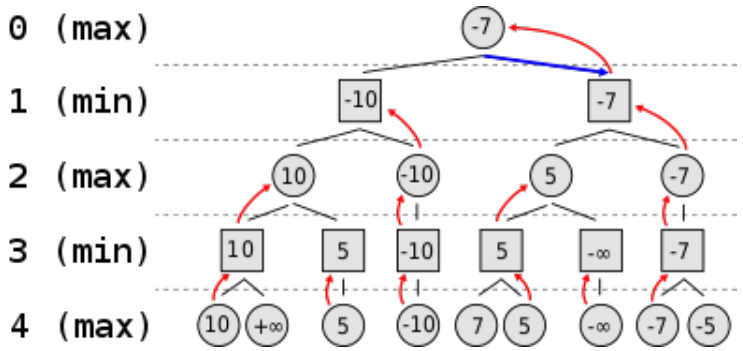


Figure 2.4: A diagram showing the minimax algorithm [16]

Quiescence

One limitation of using a heuristic evaluation for a certain depth is the *horizon effect*. Since the heuristic only evaluates a position statically, without considering captures, the minimax search is prone to making moves which seem better in the short term. For example, a heuristic evaluation of the two positions in figure 2.5 may determine that the first position is better for white since it has an extra pawn. However, the first position is worse since black can capture white's queen on the next move, unlike in the other position.

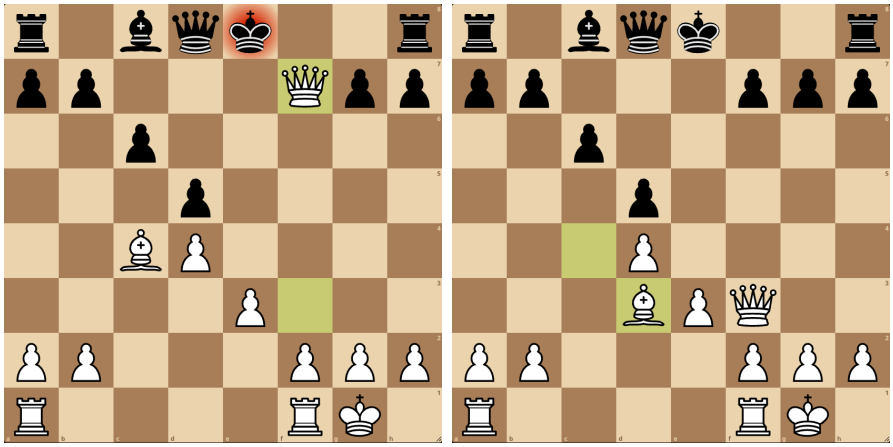


Figure 2.5: Two positions to demonstrate the *horizon effect*

The method I used to mitigate this problem is to perform an additional search, called the quiescent search, on the leaf nodes instead of the heuristic evaluation. In this stage "quiet" moves are ignored to significantly reduce the branching factor of the tree. Once a node with only quiet moves remaining is reached the heuristic evaluation is used as the value for the position. The decision of which moves should be considered as quiet is a trade-off between increasing the branching factor of the quiescent tree and ensuring that the search does not stop on the horizon of the heuristic evaluation decreasing dramatically.

Alpha-beta pruning

Alpha-beta pruning [17] is a technique which can double the search depth reached while exploring the same number of nodes in the best case. Subtrees can be pruned from the search if it is known that they cannot influence the final decision. For example, in figure 2.6, the final two subtrees below node C can be pruned. Since C is a minimising node, it has a value of 2 or lower. A is a maximising node, so it is guaranteed to have a value of 3 or greater.

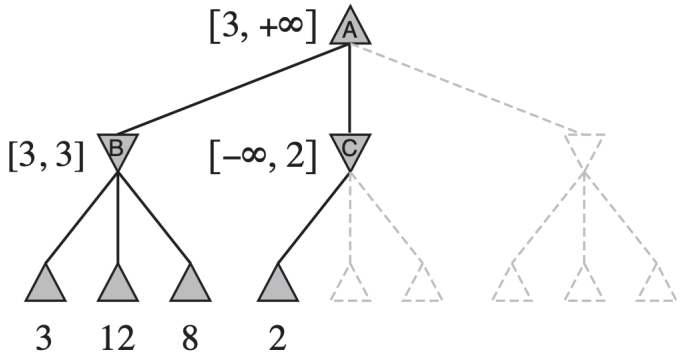


Figure 2.6: A diagram showing alpha-beta pruning [17]

Iterative Deepening

Iterative deepening is the process of first running the search at depth 1, and then iteratively increasing the depth of the search until either the maximum depth has been reached or another cutoff condition is satisfied. Alpha-beta pruning is the most effective when the nodes are considered from best to worst, so the closer to this order, the less time the engine spends searching to a given depth. Using iterative deepening allows the engine to use the best ordering of moves at search depth n as the order to consider moves at search depth $n + 1$. Furthermore, it allows the engine to return the best move at the highest depth it has reached at any point in time so it can manage its remaining time.

Transposition table

In chess, many different sequences of moves can reach the same position, but they are still different nodes in the search tree. A transposition table ensures that the same positions are not re-evaluated. It is a hash table, with the key being a hash of the position and the value being the evaluation of the position alongside the depth it was evaluated. Another advantage of keeping a transposition table is that it can be used to order moves when using iterative deepening.

2.2.6 Heuristic evaluation

When the game is over the engine can simply use the game result as the evaluation. However, it needs a heuristic for most positions to determine whether it is better for one side or the other and by how much. Since the final engine provides reasoning for its moves, I aimed to use features of the position which would be familiar to a chess player as the basis for the heuristic.

Material

The number of pieces that each side has is an obvious starting point, since the pieces can protect their king from being attacked, and can be used to checkmate the opponent. The different pieces should be weighted differently according to how powerful they are, a common scheme used is shown in figure 2.7. The sum of these values for each side is known as *material*.

Piece	Value
Pawn	100
Bishop	300
Knight	300
Rook	500
Queen	900

Figure 2.7: A table showing commonly used values for each piece [18]

The relative value of knights, bishops and rooks is widely debated and analysed by chess players [19], with bishops often being regarded as more powerful later in the game when there are fewer pieces. I used these values as a starting point and investigated how different values affect the performance of the chess engine.

Piece Positioning

The position of different pieces on the board is another commonly used heuristic in chess engines. For example, a king in the middle of the board can be more easily attacked by the opponent, and pawns that are further up the board are closer to promotion. I used piece-square tables, which assign a value to each piece for every position on the board. The sum of these values for all of the pieces is added to each side's heuristic score.

Pawn Structure

One aspect of the position that piece-square tables cannot account for is how the pawns interact with each other. Pawns which are adjacent or diagonal can defend each other, whereas pawns directly behind another pawn are much weaker.

Figure 2.8 shows (from left to right) connected pawns, isolated pawns and doubled pawns. Connected pawns are assigned a positive value, and isolated or doubled pawns are assigned a penalty. Passed pawns are another aspect of the pawn structure which could be considered in the heuristic. These are pawns which do not have an opposition pawn in front of them or any of the adjacent ranks.

Mobility

The mobility can be measured as the number of moves available to each side. Having greater mobility than the opponent is considered an advantage because it means that your pieces control more of the board [20]. It is also a sign of better development from the opening.

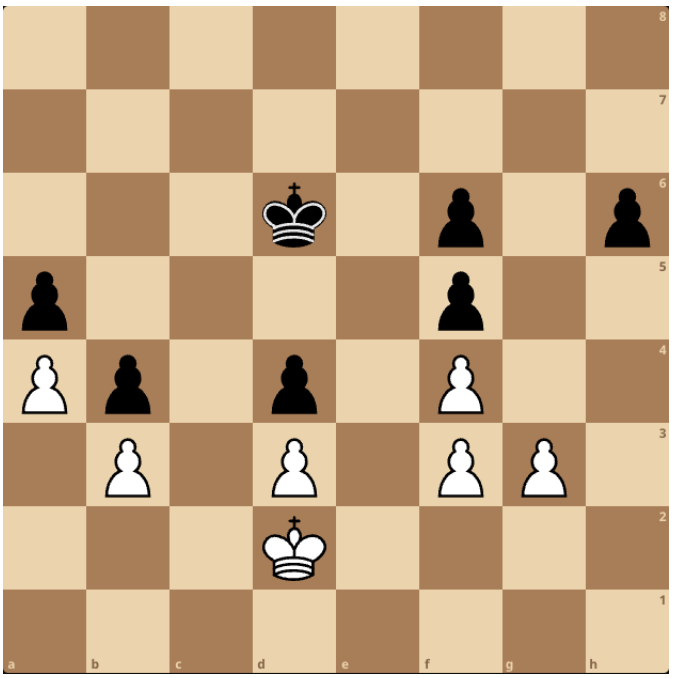


Figure 2.8: A diagram showing different pawn structures

Tapered Evaluation

The relative importance of these different factors varies from the start of the game to the end of the game. For example, rooks are much more powerful in the endgame when fewer pieces are remaining and they can control more of the board, so the engine assigns a greater relative value. By calculating a phase of the game, based on the number of pieces remaining, the engine can interpolate between two different heuristics, one for the start of the game and one for the endgame.

2.3 Research on Providing Interpretations

In this section, I outline the research I completed on existing approaches to adding interpretations to chess engines. I also describe how I added to these and how the tool provides the explanations to the user.

2.3.1 Principle Variation

The principle variation is the sequence of moves which the engine considers to be optimal for both sides. Following the principle variation down the search tree leads to the position with the heuristic evaluation that is assigned to the root position. The principle variation is provided alongside the evaluation by many engines.

Splitting the heuristic evaluation of a position into meaningful and interpretable components could allow them to be used for the reasoning of the engine. Instead of only returning the sum of the components, I modified the evaluation function to return the individual components as well. These could then be passed up the search tree so that the initial call can return the component values associated with its evaluation.

One of the positives of this approach is that it only required small adaptations to the search and evaluation functions. Furthermore, the reasoning always aligns with the overall evaluation, since it is simply a sum of the individual components.

The main disadvantage of this approach is that it does not provide any reasoning about the other positions which are not part of the principle variation. For example, the second-best starting move may have a very similar evaluation to the best, but with a very different component split, which is not shown to the user. Alternatively, the other moves may be significantly worse, but the reason for this difference is not provided using this approach.

2.3.2 Real-time Tutor

Studies have shown that providing learners with real-time feedback on their responses has a positive outcome on both engagement and understanding [21]. One of the motivations for this project was to improve the existing engine tools for helping beginner and intermediate chess players.

I used the engine-generated explanations to provide real-time feedback on each move that the user makes. This allows the player to reflect on whether they considered the same advantages and disadvantages as the engine. Furthermore, when the suggested move from the engine is different to the player's move, they can understand why it was better.

For the explanations to be most effective in real-time, they need to be quick to process and understand. Therefore, I designed a simple visualisation to convert the individual component values into a more digestible interface. I took inspiration from the visualisation techniques that existing tools use where appropriate to make the interface easier to understand for users who are already familiar with other tools. The visualisation replaced the natural language explanations which I had originally planned as a project extension.

2.4 Hardware

The hardware used needed to be consistent because of the real-time aspect of the time control discussed in section 2.2.4. More performant hardware would allow the same chess engine to perform better in the same time control. Therefore, all of the development and testing of the project was completed using my laptop, with the following specifications: **CPU:** Apple M1, 8 cores; **RAM:** 8GB; **OS:** macOS 14.3.1.

2.5 Data

As part of the evaluation of the success of the project, I compared the engine to Stockfish, the most commonly used open-source chess engine. I used a dataset of FEN strings and the corresponding precomputed Stockfish evaluation at depth 22 [22]. Computing the evaluations to a high depth can take a long time without powerful hardware, so using the dataset sped up the evaluation process. The evaluation of almost all positions converges by depth 22, so the author of the dataset chose this as the search depth. The positions in the dataset were collected from a database containing millions of games from chess masters. I compared the evaluations of positions in the dataset to those that my engine computes.

2.6 Requirements Analysis

In this section, I explain how I broke down the core objectives and extensions of the project into implementable milestones. I also analyse how these milestones depended on each other.

The first major milestone of the project was a working prototype of the chess engine. I first had to implement the rules of chess with the move generation and apply functions. I also had to implement a heuristic evaluation function and the minimax search algorithm.

To evaluate different versions of the engine, I needed to automate the playing of games between my engine and other engines. To do this I implemented the Chess Engine Communication Protocol and used the XBoard GUI.

The first prototype version of the engine was too weak to effectively evaluate, so I implemented the various techniques discussed in section 2.2.5 to improve its performance. Implementing this extension before the core of the project was not the original plan, however, it was required to ensure that the explanations were going to be useful.

The final core component of the project used the principle variation and the associated components of the heuristic evaluation as the basis of the explanations. I designed a visualisation for these explanations as a second extension for the time I had remaining.

Figure 2.9 shows the revised milestones and deliverables of the project after the research and preparation. I analysed which deliverables were required to begin working on the next deliverable. This allowed me to plan my time for implementing the project effectively.

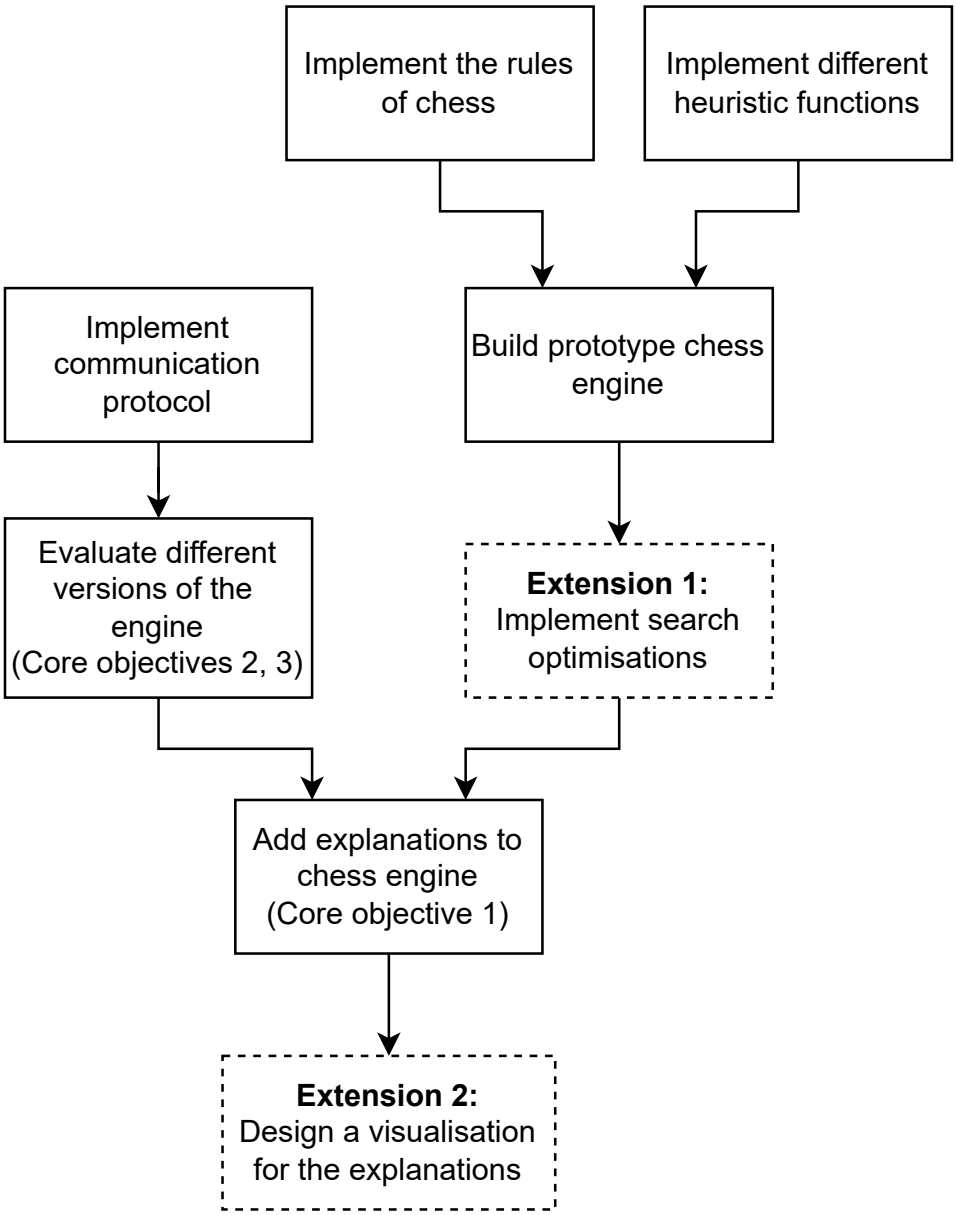


Figure 2.9: A dependency analysis of the project deliverables

2.7 Development Model

I used an Agile development model to develop the chess engine to ensure that I had a working prototype engine as soon as possible. I was then able to iteratively improve the engine, using unit tests to ensure that I was not breaking the existing engine. The short development cycles used in the Agile development model also worked well with alternating between project work and my other university assignments. I used regular meetings with my supervisor and profiling tools to decide which areas to expand on and improve at each iteration.

2.8 Backups and Version Control

I used GitHub for version control and as a primary backup, with commits after every milestone. I used Overleaf to write the dissertation, which also has version control and backups.

2.9 Development Environments

All development of the engine was done in Python, using Visual Studio as the IDE. I chose to use Python because most of the project was focused on rapidly prototyping and implementing new ideas. Python is the language I have the most familiarity with, so using another language would have required me to spend time learning which could have been used for development. I used cProfile [23], the profiler provided by the standard library, to test the performance and identify which functions to focus on optimizing. I also used XBoard to provide the GUI and to communicate with other engines. I used a library called bitarray to provide an efficient C implementation for the bitboards.

Chapter 3

Implementation

In this section, I outline the steps I took to implement the chess engine as well as the project extensions. I describe the algorithms discussed in the preparation section in more detail, alongside the impact that they had.

3.1 Board Representation and Move Generation

Implementing the board representation as described in the preparation section was quite straightforward. Similarly, implementing the pseudo legal move generator and apply functions with the move encodings described only required converting the different piece movements into the corresponding changes in the bitboard index.

The next function I implemented was the unmake function, which allowed the search to be able to apply and unmake moves when it searches the tree of positions. When implementing it, I realised that undoing captures would require the piece that was captured to be known, so that it could be placed back on the board. One approach would have been to modify the move encoding to include the type of piece captured. Instead, I decided to add a stack of the piece captures to the board representation with the function popping from the stack when unmaking captures. This has the advantage of only having to check which piece is being captured when applying the move, rather than on move generation.

Initially, I also encountered problems when writing the unmake function, where making a move and then unmaking it would prevent the opponent from making en passant moves which should be available. The ability to castle would also be removed when king or rook moves were made and unmade. Similarly to the previous problem of captures, I solved this by using a stack to record the history of castling and en passant availability.

3.1.1 Perft

Performance **t**est, or perft, is a standardised function used to test the correctness and performance of a move generator [24]. The move generation tree of all legal moves from a given position is traversed. The number of nodes at a certain depth is counted and compared to a set of known results. The time taken to traverse the tree is an indicator of the performance of the move generation, apply and unmake functions.

```
def perft(depth):
    moves = generateMoves()
    if depth == 1:
        return len(moves)

    nodes = 0
    for move in moves:
        applyMove(move)
        nodes += perft(depth-1)
        unmake(move)

    return nodes
```

Figure 3.1: The perft function in pseudocode

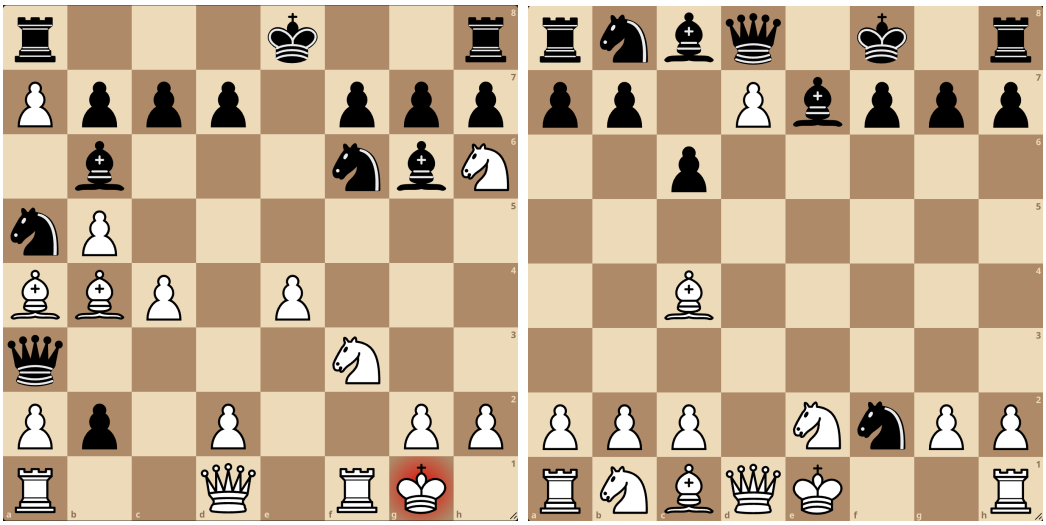


Figure 3.2: The two positions I used to run the perft test

To test for the correctness of my move generation function I ran the perft test on the starting position and the two positions in figure 3.2. I selected positions with checks, checkmates, promotions, en passant and castling so that every aspect of move generation is tested. The final version of my move generation function produced the correct results for all three positions up to a depth of 4. The number of nodes at each depth in the two test positions is shown in figure 3.3.

Depth	Nodes
1	6
2	264
3	9467
4	422333

Depth	Nodes
1	44
2	1486
3	62379
4	2103487

Figure 3.3: The results for the perft test when run on the two positions in figure 3.2

I also used the time taken to run the perft function at different depths to test different versions of my move generation, apply and unmake functions.

My first approach to filtering out moves which leave the king in check was to apply every move and generate the pseudo-legal moves in the subsequent position. If any of the moves were

captures of the king, then the original move must have left the king in check, and so it was not legal.

For the second approach, to determine whether a move leaves the king in check, the move generator first identified if the king was in check. If so, only king moves or captures of the piece giving the check could be valid. Only these moves are then tested using the first approach. Otherwise, only moves involving the king or a piece which could be pinned to the king (for example if an opponent bishop and the king are both on the same diagonal as the piece) are tested to see if a subsequent move can capture the king. These moves are filtered out, and the rest are returned as legal moves.

By testing with the perft function, I verified that both approaches correctly implemented move generation. I also compared the performance of the two approaches using the perft function at depth 3. The first approach took 3.2 seconds to count the 9467 nodes in the first position, whereas the second approach took only 0.8 seconds.

3.2 Communication Protocol

The full Chess Engine Communication Protocol (CECP) was not required for the project, only enough to allow the engine to automatically play against other engines [25]. I investigated which commands were required and implemented this subset of the commands in the protocol:

- `xboard`

  This is the first command that is sent to the engine by the GUI, to tell the engine to go into xboard mode if required.

- `protover N`

  This is sent immediately after the `xboard` command, to tell the engine which version of the protocol to use. The engine should respond with a list of which non-default features it implements. For example, it responds with the engine's name and the format it expects to receive moves in.

- `new`

  This command tells the engine to begin a new game, playing as black by default.

- `quit`

  This command tells the engine to exit, to ensure that the program exits cleanly.

- `time N`

  This command tells the engine how much time remains on its clock in centiseconds so that the engine can update its internal clock.

- `otim N`

  This command tells the engine how much time remains on the opponent clock in centiseconds.

- `ping N`

  The engine should respond to this command with `pong N`. Its purpose is to prevent certain race conditions, which existed in previous versions, from occurring.

- `usermove MOVE`

  This command tells the engine its opponent's next move. The engine applies this move, and then decides on its response with the command `move MOVE`. The moves are in long algebraic notation, which has the coordinates of the starting and ending positions of the piece being moved. If the game is over after the opponents move, then the engine should respond with the result of the game.

- `go`

  The engine should respond with its next move. This command is usually only sent at the beginning of the game when the engine is playing as white.

- `setboard FEN`

  This command is used to change the position on the board. The new position is sent in FEN notation [10], which defines the position in one line of text in terms of the positions of pieces, which side is next to play, the castling rights and en passant availability.

- `white`

  This command tells the engine that it is playing as white.

- `black`

  This command tells the engine that it is playing as black.

3.3 Search

I first implemented a minimax search with the heuristic and then modified the search to include the optimisation techniques described in the preparation section. As I implemented additional optimisations, I measured the improvement in terms of reduction in the number of nodes explored in the search tree. I used the first position in figure 3.2 as the test position. The initial minimax search was implemented as follows:

Using the search to a depth of 4 on the test position, the number of nodes searched was the same as in the perft test shown in figure 3.3, 432070 in total.

3.3.1 Alpha-beta Pruning

The first optimisation which I implemented was alpha-beta pruning. As described in the preparation section, the search keeps track of upper and lower bounds for the evaluation of the root position. This allows many sub-trees to be pruned from the search when it is known that they are not part of the principal variation.

The alpha-beta search reduced the number of nodes searched in the test position to 5292. For identically and independently distributed leaf values with no move-ordering, the alpha-beta search allows the search depth to be increased by a factor of $\approx \frac{4}{3}$ [26]. This would mean the number of nodes searched would be approximately equal to the nodes at depth 3, 9467. The actual number of nodes searched was fewer, which suggests that the moves were ordered slightly better than randomly. This can explained by captures and promotions being earlier in the move generation than quiet moves.

```
def eval(depth):
    if depth == 0:
        return heuristicEval()

    moves = generateMoves()

    if len(moves) == 0:
        return heuristicEval()

    if toPlay == WHITE:
        value = -inf
        for move in moves:
            applyMove(move)
            value = max(value, eval(depth-1))
            unmake(move)
        return value
    else:
        value = inf
        for move in moves:
            applyMove(move)
            value = min(value, eval(depth-1))
            unmake(move)
        return value
```

Figure 3.4: The minimax algorithm in pseudocode

3.3.2 Quiescence Search

I implemented the quiescence search after alpha-beta pruning because there would be too many nodes to explore without the optimisation. Figure 3.5 shows the number of nodes at each depth when the search was performed on the search position to a depth of 3. Note that most of the nodes are at a greater depth than 3 because they are part of the quiescence search. This initially surprised me, however it is expected due to the long sequences of captures which can occur.

In the quiescence search, the assumption that one of the moves available will improve the evaluation of the position is made [27]. This is commonly referred to as the *Null Move Observation*. Therefore, before searching any of the child nodes, the heuristic evaluation of the current position is used as a lower bound for the evaluation. This reduces the number of nodes which are searched in many cases.

When implementing the quiescence search had to decide which moves to filter out as quiet moves and which to keep as quiescent moves. Initially, I used all captures and promotions as the quiescent moves, however, this proved to be too expensive in terms of the number of nodes explored. Instead, I filtered out pawn captures from the list of quiescent moves, since they would have the smallest impact on the heuristic evaluation, which reduced the total number of nodes explored in the test position by 96%. Figure 3.6 shows that with alpha-beta pruning, the quiescent search can be performed with only slightly more nodes than the original minimax search.

I also chose not to include checks and check evasions in the quiescence search since they also added a lot of nodes to the quiescence search. The *Null Move Observation* does not hold for check evasions, since a move must be made to escape the check. Therefore, the branching factor for sequences of checks can be much greater than when only considering captures.

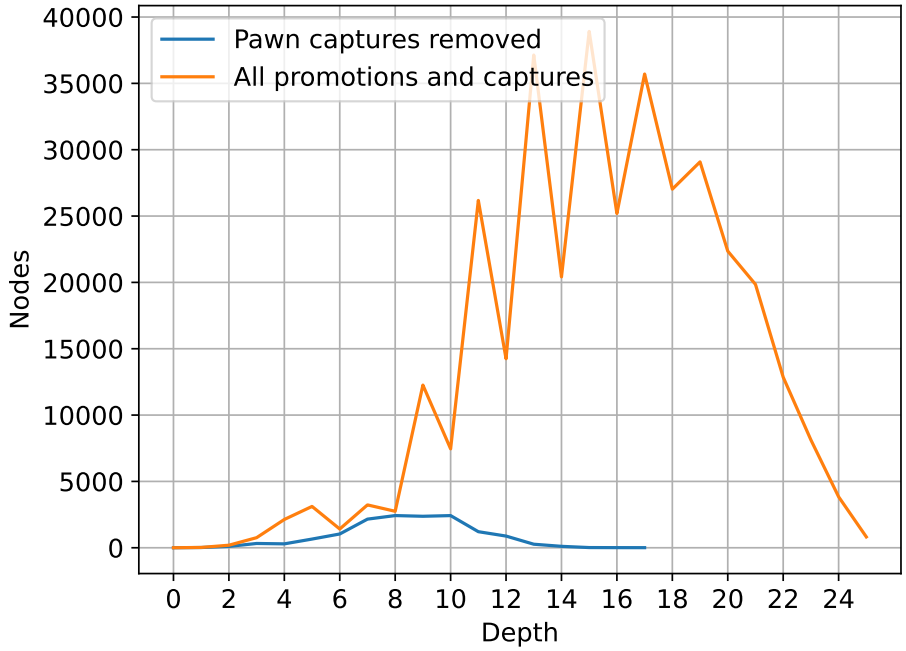


Figure 3.5: A graph showing the number of nodes at each depth

3.3.3 Iterative Deepening and Move Ordering

As explained in the preparation section, more nodes can be pruned when they are closer to the optimal ordering. One approach to improving the move order is to use heuristics, such as ranking captures of high-value pieces first. However, these can often be costly to compute. By iteratively deepening the search depth, the optimal ordering of moves from the previous search can be used to order the moves for the next depth. The increase in nodes pruned is enough to offset the additional nodes searched at lower depths, as shown in figure 3.6. Another benefit of iterative deepening is that the search can be stopped at any point in time, and the search to the previous depth can be returned.

3.3.4 Transposition Table

I used a Python dict for the transposition table, which implements a hash table. For the key of the hash table, I used the concatenation of the twelve bitboards used to represent the pieces on the board. I also appended binary encodings of the next player, castling rights and en passant availability.

Each entry in the transposition table consisted of four parts. Firstly, the evaluation of the position is calculated by the search or the heuristic. Each entry also contained the depth to which the position was evaluated, with quiescent nodes being depth 0 as well as positions evaluated with the heuristic. This is included to ensure that the search does not use an evaluation that was calculated to a lower depth than is required.

The age of the position in terms of the total number of half moves is also a field in each transposition table. After each move that is played, any positions in the transposition table with an age less than the current position are removed, to prevent it from continuing to use more memory. This can lead to fewer hits in the transposition table if the same position occurs again after some more moves have been played. However, the memory trade-off is worthwhile since most of the removed positions will not be reached again.

In an alpha-beta search, the exact evaluation is not always calculated, with many nodes assigned an upper or lower bound. The node type - exact, upper or lower bound - is the final

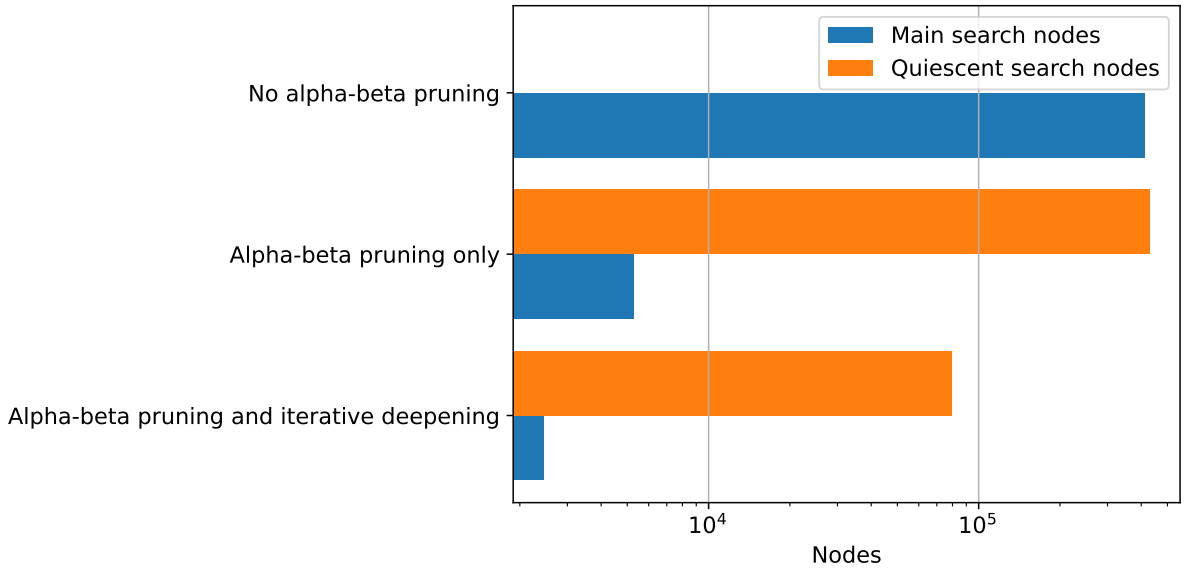


Figure 3.6: A graph showing the number of nodes searched in different versions of the search

field in each transposition table, since only exact nodes can cut the search short. The bounded nodes are still stored in the transposition table because they can be used for move ordering.

Implementing the transposition table reduced the total number of nodes searched in the original minimax search by 4.2%, down to 413880. In total, there were 149303 transposition table hits, however, most of these were in the leaf nodes so only the heuristic evaluation was prevented.

3.3.5 Testing the Search

Due to the scale and complexity of the search tree, it was difficult to test if all of the algorithms had been implemented correctly. I ran tests to measure the number of nodes in the search tree after every optimisation. I used a reduction in the number of nodes as an indication of a successful implementation. Furthermore, I ran the tests described in the evaluation section to ensure that the improvements to the search were being reflected in the performance of the engine.

The extensive number of games played also provided many chess positions to test the engine. The fact that there was no unexpected behaviour or crashes in the final evaluations is a strong indication of the correctness of the move generation and making functions.

3.4 Tuning Heuristic Evaluation

Tuning the parameters of a chess engine is a difficult problem since the impact of any small change is hard to measure. Although a change may improve the engine's performance in one phase of the game, it can be detrimental in others. To accurately assess whether a small change has a positive impact the two versions must play many chess games to obtain statistically significant results. For example, Stockfish allows volunteers to contribute computation time to perform these tests and improve the engine [28].

Due to computation and time limitations, that scale of testing is out of scope for this project. Therefore I focused on large improvements which show statistically significant increases in performance with a small number of test games. I also aimed to use parameters which have already been tuned where possible.

For the material and piece-square tables, I used the tables tuned for Ronald Friederich's RofChade engine [29]. They were tuned to maximise the performance of an engine which only used piece-square tables for the heuristic evaluation function. There are two sets of tables, one for the opening and middle game, and one for the endgame. The evaluation function smoothly transitions from one to the other with a tapered evaluation based on the amount of material left on the board.

I decided on the values for the pawn structure and mobility of a position based on my knowledge and understanding of chess and from reading articles and books on chess [30] [20]. I tested the performance of the engine with and without the addition of these features to verify that they had the intended impact of improving the engine. This is discussed further in the evaluation chapter.

3.5 Providing Explanations

To provide the explanations to the user, I first modified the chess engine to generate explanations alongside the evaluation and best move. I then designed the visualisation to show these explanations to the user.

3.5.1 Generating Explanations

It was quite straightforward to modify the alpha-beta search to return the principle variation by accumulating a list of the best moves at each level of the search. The heuristic evaluation also only required a small change to return the overall evaluation as well as a tuple of values for the different components. Both the principle variation and the tuples were then passed up to the root of the tree.

3.5.2 Implementing Tutor Module

The user plays against the engine when using the tutor, so there needs to be two instances of the engine. One to make moves against the player, and one to analyse the user's moves and provide explanations. The engine acting as the opponent communicates with the XBoard GUI, as well as the tutor instance.

When it is the player's turn to move, the tutor engine performs a search to find the best move to play, as well as the associated principle variation and each of the components. After each move the user makes, the engine performs another search to evaluate the new position. If the user's move matches the original move that the engine found, the reason why it was the best move is shown to the user. Otherwise, the user is shown why their move was not as good.

3.5.3 Visualisation

Since I used XBoard for the primary GUI were the user plays chess, I was constrained to using the command line to provide the visualisation. I used evaluation bars constructed from Unicode characters as the primary visualisation to show the user. Each evaluation bar represents either the overall evaluation of the position or the value of a component feature used in the heuristic. The evaluation bar is scaled from completely in white's favour on the left, to completely in black's favour on the right, with the middle indicating equality. Evaluation bars are commonly used by other tools so many users will already be familiar with what they represent. They are also intuitive to understand, because the position of the transition from white to black represents the evaluation, so a larger white bar means the evaluation is better for white.

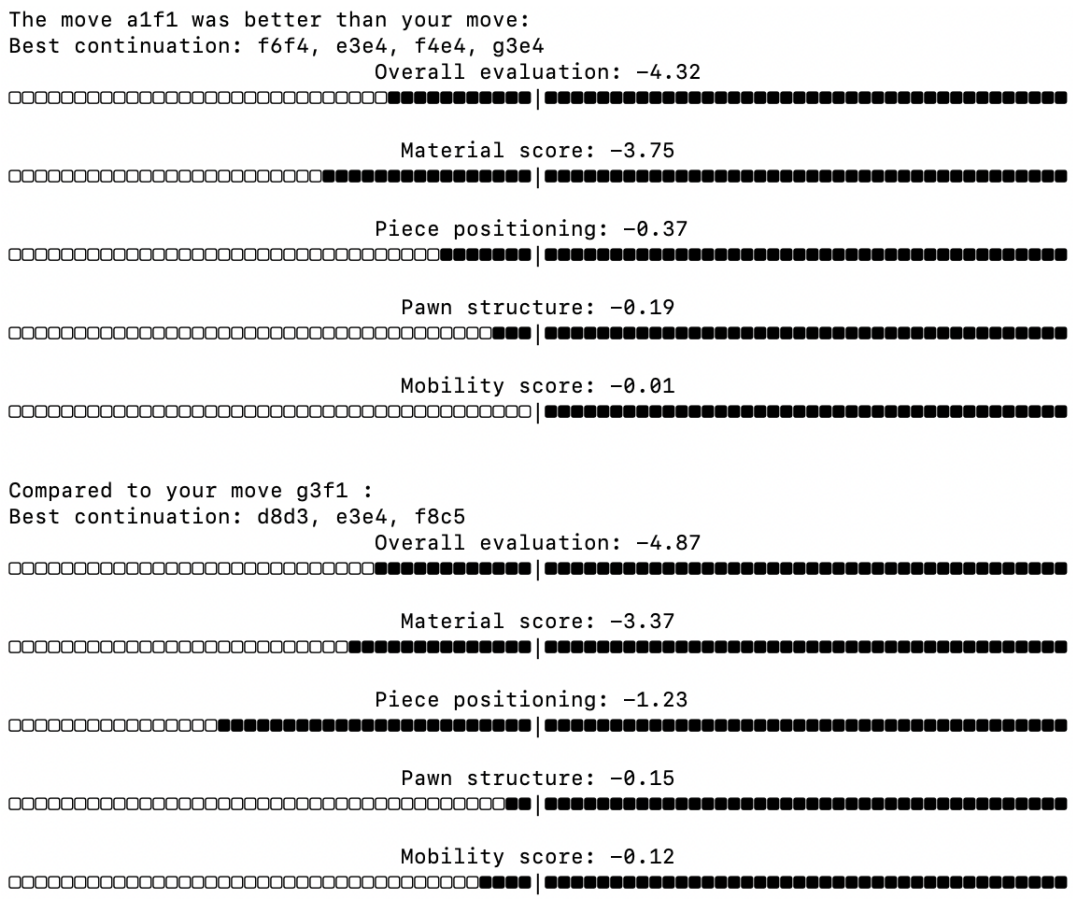


Figure 3.7: A screenshot of the visualisation for explanations

As shown in figure 3.7, each evaluation bar is accompanied by the corresponding value. The width of each bar represents the standard range of values that would be encountered during a game of chess so that the user has context for the scale of the values. The bars for different features are also ordered by the scale of their contribution to the overall evaluation so that the most significant features are more prominent in the visualisation.

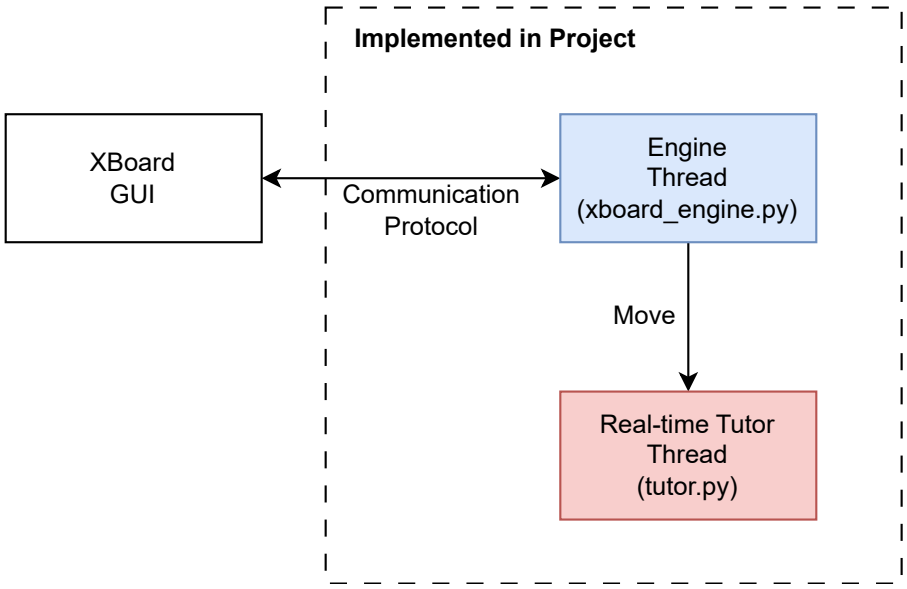
3.6 Repository Overview

The engine needs to run two threads when it is providing explanations to the user because the user's move needs to be evaluated at the same time as the engine calculates the best response. Only one thread communicates directly with the XBoard GUI, so this thread relays the moves to the second thread.

The first thread acts as the user's opponent and communicates with the XBoard GUI using the Chess Engine Communication Protocol (CECP). The user makes moves using the GUI, and these are communicated to the first engine thread. Each move that the user makes is relayed to the second thread by writing to a file. The first thread then computes the best response and sends it to XBoard.

The second thread acts as a tutor for the user by suggesting alternative moves after the user makes a mistake. It gives visual feedback to the user via the command line interface after every move that it receives from the first thread.

The file tree in figure 3.8 gives an overview of the different files in the project and a brief description of what each file implemented, as well as how they correspond to the different threads that run when the engine is in tutor mode.




```
Project
├── board.py
│   ├── Class definition for the board representation
│   ├── Implements the move generation function
│   └── Implements the make and unmake functions
├── engine.py
│   ├── Class definition for the engine
│   ├── Implements the heuristic evaluation function
│   ├── Implements the minimax search function
│   └── Implements the interpretable versions of these functions
├── xboard_engine.py
│   ├── Implements the Chess Engine Communication Protocol
│   └── Relays moves to the tutor thread
├── tutor.py
│   ├── Receives moves from the XBoard engine thread
│   ├── Gives move suggestions to the user
│   └── Provides explanations for why the move is better
├── bitarray_masks.py
│   └── Implements bitboard helper functions
├── colour.py
│   └── Enum for the two different sides
├── piece.py
│   └── Enum for the different piece types
├── ttentry.py
│   └── Class definition for transposition table entries
└── eval.py
    └── Evaluation script to calculate the accuracy of the engine
```

Figure 3.8: A diagram showing the engine pipeline and an overview of the different files in the project

Chapter 4

Evaluation

This chapter contains an evaluation of the engine's performance in terms of estimated Elo rating and accuracy using Stockfish as a ground truth. The chapter also contains an evaluation of the interpretable version of the engine.

4.1 Engine

I used two different methods to evaluate the engine, one testing its ability to play chess games against an opponent and one comparing how it evaluates different positions to Stockfish. I used XBoard to automatically play matches against different engines to test its ability to play chess games. I estimated the Elo rating of my engine from the match results. For the accuracy of the engine I used a dataset of chess positions and ran the engine's evaluation function. I compared this to the evaluations that Stockfish assigned to calculate the accuracy of the engine.

I used my laptop as described in the hardware preparation section to run all of the evaluations. To ensure consistent and fair results, I checked that no other high-demand applications were running simultaneously with any of the matches and that my laptop was connected to power.

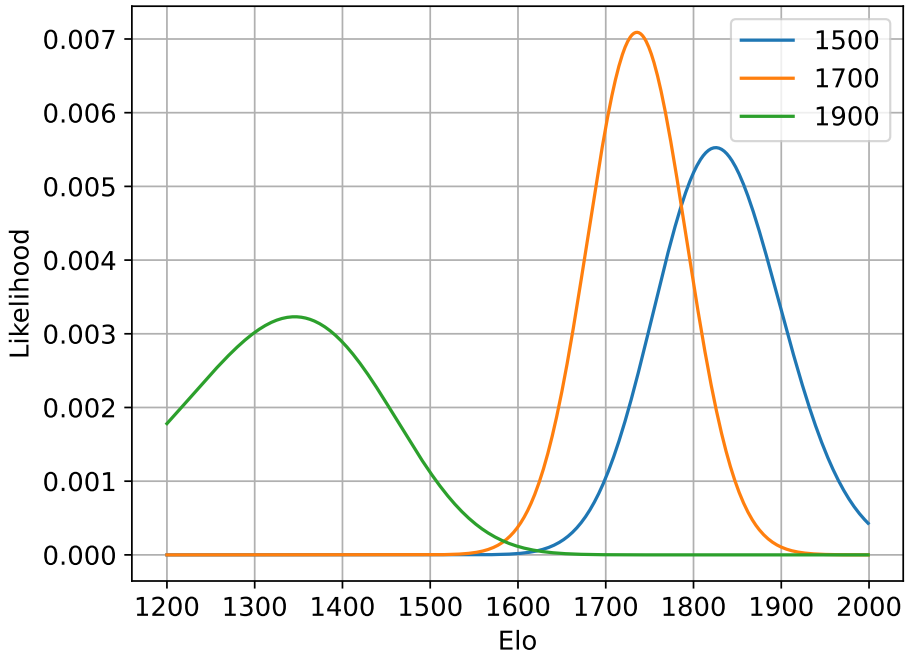
4.1.1 Elo

The Elo rating system was first proposed in 1967 by Arpad Elo [31] and has been the standard rating system used for chess and other games since then. The rating system is derived from the assumption that a player's performance level in a given game is normally distributed. It also assumes that the player with a higher performance level in a game always beats their opponent. The mean of a player's performance distribution is then defined as being their Elo rating. The standard deviation of an individual's performance in a match was defined as being 200 in Elo's proposed system. Since the standard deviation of each player's performance is 200, the standard deviation of the difference in performance is $200\sqrt{2}$. Therefore the expected score of one game against an opponent of rating r is the CDF of a normal distribution with mean r and standard deviation $200\sqrt{2}$.

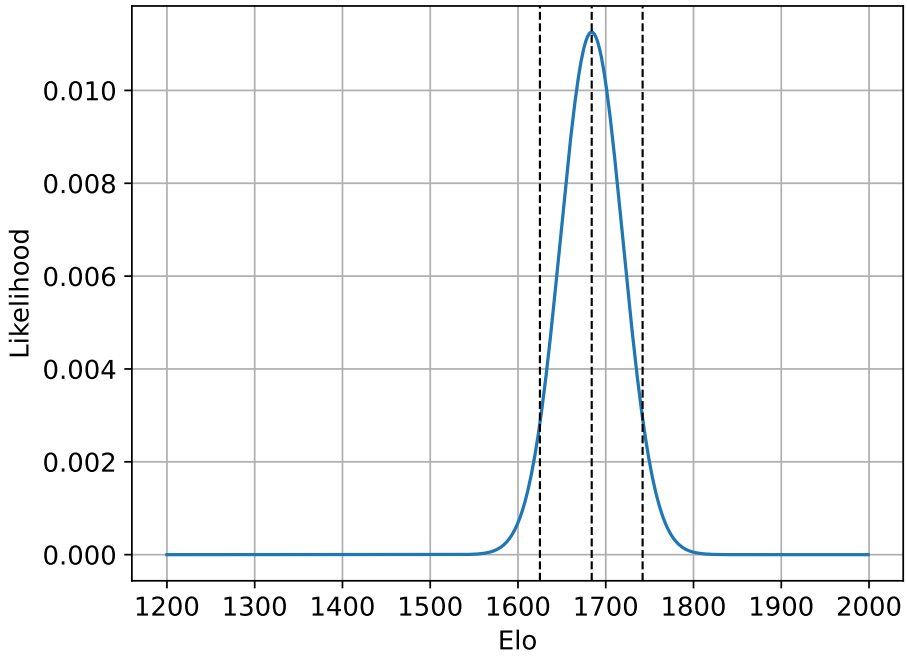
Human players can improve or decline between the games that they play, so a formula is used to update their rating. However, we can assume that the same version of a chess engine has a constant Elo rating. The Computer Chess Ratings List (CCRL) has run over one million games between thousands of different chess engines [1] to estimate their performance ratings. For evaluating different versions of my chess engine, I used ten-game matches between my engine and different engines with a known CCRL rating. I then calculated a maximum likelihood estimator and 90% confidence interval for the Elo rating of that version of the engine.

Opponent	Opponent Elo Rating	Score
Stockfish set to limit	1500	9/10
Stockfish set to limit	1700	5/10
Fairy-Max	1900	0/10

Figure 4.1: A table showing the results of the final version of the engine in ten-game matches against different opponents



(a) A graph showing the Elo probability distribution for the match against each opponent



(b) A graph showing the overall Elo probability distribution with the maximum likelihood estimator and 90% confidence interval

Figure 4.2: Graphs showing the Elo probability distributions calculated from the results in figure 4.1

From playing against the engine, I estimated that it had an Elo rating similar to mine, approximately 1700. Therefore I aimed to choose opponent engines of a similar rating so that the confidence intervals would be tighter. Fairy-Max is an open-source engine which is installed as part of XBoard. It has an Elo rating of 1900, so I used it as one of the opponents. I could not find open-source engines with ratings of 1500 and 1700, so I chose to use Stockfish instead. Stockfish implements strength limiting, so it can be set to perform at a certain rating. I used ten-game matches against each of these opponents to test each different version of my engine. The score is calculated as $wins + \frac{draws}{2}$. The results of the final version are shown in figure 4.1.

I modelled the result of each game in the match as two independent Bernoulli variables with a probability equal to the expected score. I used two variables to allow for the possibility of a draw, represented by one of the two random variables being 1. I also modelled each game as independent, so that each match was modelled as a binomial random variable, with $n = 20$ and p being the expected score of a single game. The probability distribution for each of the matches is shown in figure 4.2a.

Under the assumption that the results of each match are independent random variables, I computed the combined probability distribution of all three match results. This is shown in figure 4.2. I used this distribution to calculate a maximum likelihood estimate for the rating of the engine, 1692, and a 90% confidence interval, 1633 to 1749.

I repeated this analysis with different versions of the engine to identify which features and search techniques improved the rating of the engine the most. During this analysis, I also noticed that the further the estimated rating was from the rating range of the opponents the wider the confidence interval was. This is shown in figure 4.3.

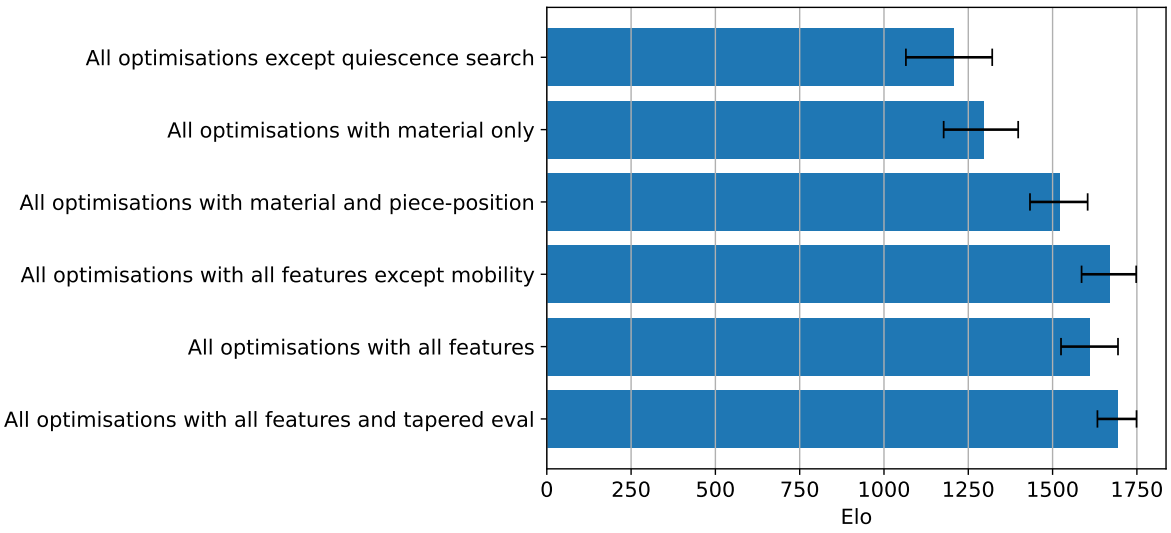


Figure 4.3: A graph showing the estimated Elo rating of different versions of the engine with 90% confidence intervals

4.1.2 Accuracy

In this section, I use accuracy as the percentage of positions correctly classified winning for white, winning for black or drawing. I used a dataset of precomputed Stockfish evaluations of positions taken from master games as the test data [22]. The evaluation that Stockfish outputs can either be a forced checkmate sequence or a continuous evaluation, with 0 being an equal position for both sides. The evaluations have traditionally been measured in pawns or centipawns (1 pawn = 100 centipawns), however, they are now normalized to a probability of winning with perfect play [32]. An evaluation between -1.0 and 1.0 means that the most

probable outcome is a draw, with lower and higher evaluations meaning a win for black or white is more probable respectively. When estimating the accuracy of the engine when evaluating a position I treated it as a classification problem with these three classes.

The heuristic evaluation assigns a value of 100 to pawns, so the evaluation function should be measured in centipawns. I estimated the accuracy of each engine version using the same random subset of 100 positions from the dataset. The results are shown in figure 4.4.

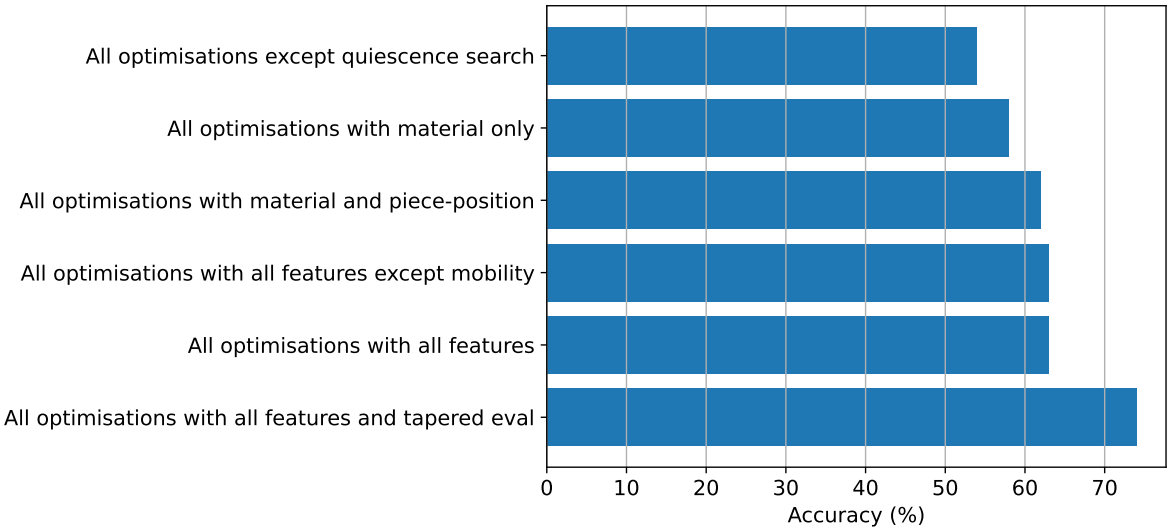


Figure 4.4: A graph showing the accuracy of different versions of the engine when compared with Stockfish

The tapered evaluation did not significantly improve the estimated rating of the engine, since there is a large overlap in the 90% confidence intervals of the final three versions of the engine. However, it was able to correctly classify 9 more positions than the version without tapered evaluation. The tapered evaluation improves the engine's evaluations in the endgame since it requires a slightly different heuristic than in the opening and middle game. One explanation for the difference in the two evaluations is that the games played against other engines were mostly won or lost before the endgame so the tapered evaluation did not impact many of the games. However, the dataset contained a mixture of opening, middle-game and endgame positions and the tapered evaluation would have had a better accuracy when classifying the endgame positions.

4.2 Interpretable Versions

Due to time frame limitations, I could not conduct a user study to evaluate the effectiveness of my project in explaining how to improve at chess. Instead, I used the **Cognitive Dimensions (CDs) of Notation** to evaluate the visualisation of the explanations. The CDs are a set of design principles created by Green [33] which can be used to analyze the quality of a design. I used the framework to evaluate the visualisation and identify how it could be improved.

Before analyzing the visualisation using each of the CDs, I must first consider the type of activity that the notation is being used for. A user of my visualisation is actively playing a game of chess at the same time as reflecting on their moves. Since the user is not directly interacting with the visualisation, the activity can be categorised as *incrementally understanding* why certain moves are better than others. Since the user is not modifying or extending any aspects of the visualisation while they play the game of chess, not all of the cognitive dimensions are relevant. I only considered the applicable dimensions.

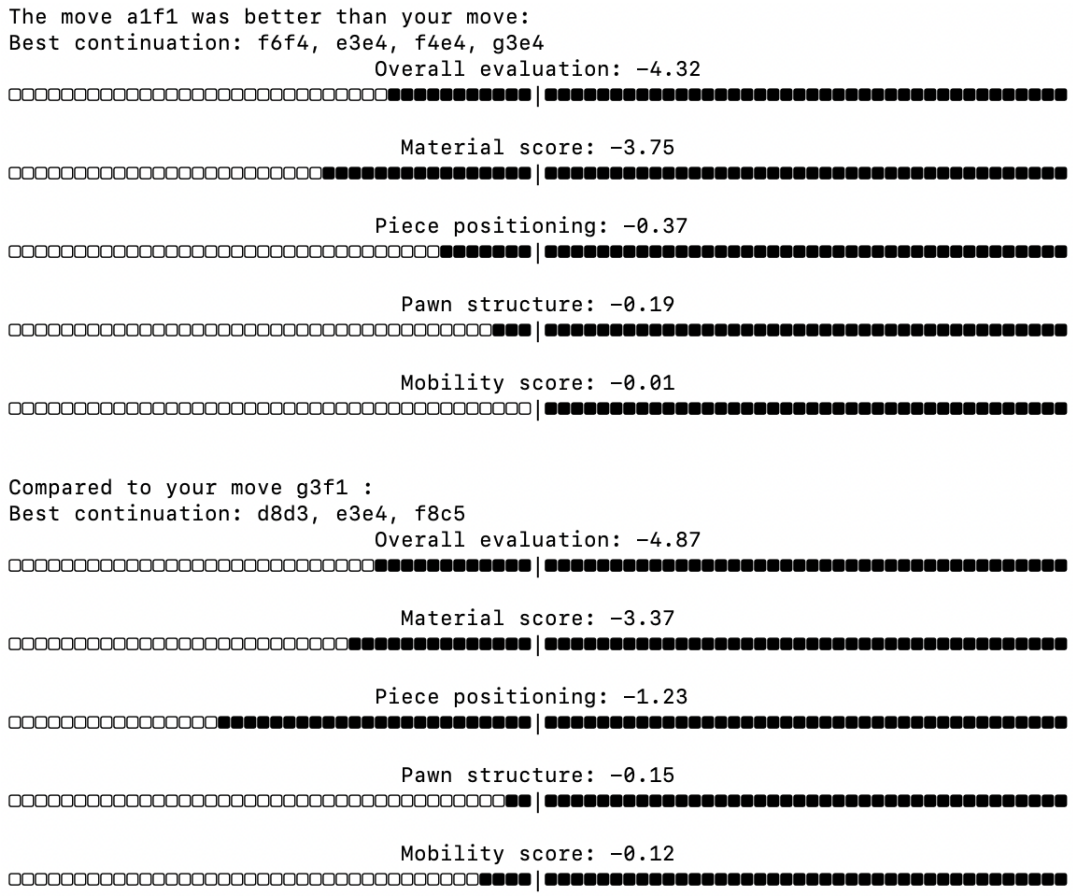


Figure 4.5: A screenshot of the visualisation for explanations (repeated from figure 3.7 for clarity)

Closeness of mapping: *How closely does the notation match the chess board?*

The notation uses several abstractions for how to think about a game of chess. Firstly, the moves are represented using algebraic from-to notation. This requires the user to think of the board in terms of coordinates rather than visually seeing a piece move from one square to another. Most experienced chess players can make this abstraction naturally, but newer players may not be used to thinking of moves in this way.

Another mapping that the visualisation uses is the evaluation bars to represent how far in one side's favour each metric is. This should be easier for a new user to interpret since the numerical values are located next to the evaluation bars. The colours of the evaluation bars also match the colours of the pieces in a game of chess.

Finally, a less obvious mapping is measuring the evaluation in terms of pawns. This mapping is not made clear by the notation, so new users are likely to struggle with interpreting the scale of the evaluations.

Visibility: *How easy is it to view and find elements?*

All of the elements are visible on the screen at the same time, so the visibility is relatively high. The different elements are on separate lines, so it is easy to see each of them separately. However, the user's eyes may be drawn to the evaluation bars, which could reduce the visibility of the move played and the best continuations. The visibility of the most relevant features in the position is prioritised by them being higher up in the visualisation.

Juxtaposition: *Can different parts of the notation be compared side by side at the same time?*

The notation is split into two sections, one for the move that the user made, and one for

the engine's suggested move. Both parts of the notation use the same structure, which makes them easy to compare. However, the individual components of the evaluation are quite far apart between the two sections, so the juxtaposition is not as strong. Some of the components may be quite close in value, so the user may struggle to identify which of the evaluation bars is showing a greater value.

Diffuseness: *How much space does the notation require to express a certain meaning?*

The use of algebraic notation to represent the moves is more concise than showing them on a chess board. However, the evaluation bars are more diffuse, because they contain redundant information. The evaluation is shown both numerically and with the bar. There are also two evaluation bars for each feature, where a more concise notation could include both the user's move and the suggested move together.

Hard Mental Operations: *When using the tool, are there difficult things to work out in your head?*

The notation requires the user to apply very hard mental operations if they want to think about the best continuation. Experienced chess players can visualize the chess board and how each move changes it, but most beginners and intermediates struggle. If the notation was more closely integrated with the GUI showing the chess board a possible improvement could be that hovering over the moves would apply them to the board. For many players, finding the algebraic move suggestion on the board is a hard mental operation in itself.

The existence of the overall evaluation bar and value reduces the number of hard mental operations that the user has to make. By having redundancy in the notation, they do not have to add the individual component values to get the overall evaluation.

Consistency: *Where aspects of the notation mean similar things, is the similarity clear in the way they appear?*

The evaluation bars are consistent in that they represent whether the feature is in one side's favour. Although the scale is not the same between the different bars, they consistently range between the greatest extremes that each value could be.

One way in which the notation is not consistent is that the features are not always in the same order. This was done to improve the visibility of these features. Furthermore, the visualisation is not consistent when the user plays the best move, because only one set of evaluation bars is shown in that case.

Error Proneness: *To what extent does the notation influence the likelihood of the user making a mistake?*

The mistake the user is most likely to make is misreading the algebraic notation and visualising the sequence of moves on the board because it is such a hard mental operation to make.

Chapter 5

Conclusion

I am pleased with what was accomplished by the project. In this chapter, I summarise the achievements of the project compared to the original objectives. I also reflect on the lessons I have learnt and the additional work I would add to the project if I had more time.

5.1 Achievements

In this section, I explain whether each of the initial objectives was successfully achieved, as well as any additional achievements of the project.

Core

- **The chess engine should be able to produce evaluations of all legal chess positions, as well as provide tactical and strategical aspects that contributed to that evaluation.**

  This objective was successful. The engine has been thoroughly tested to ensure that it can correctly generate the moves and apply them for all legal chess positions as demonstrated in section 3.1. The engine provides evaluations, the best move and the factors to which the heuristic evaluation components contributed to the evaluation as shown in figure 3.7.

- **The engine should be able to correctly classify 80% of test positions as equal, winning for white or winning for black according to Stockfish.**

  This objective was not completely successful. The final version of the engine was able to correctly classify 74% of the test positions as shown in figure 4.4. I have outlined a possible approach to improve the performance of the engine below.

- **I will test the chess engine by playing matches against other engines of known strength. The engine should have an estimated Elo of above 1500.**

  This objective was completely successful. The best version of the engine had an estimated Elo of 1692, with the 90% confidence interval being between 1633 and 1749 as shown in figure 4.3.

Extensions

- **I implemented search techniques to improve the performance of the chess engine. This had a measurable increase in the estimated Elo rating.**

  This extension was completely successful. I implemented all of the search optimisation techniques discussed in the preparation section. The technique which had the greatest impact on performance was the quiescence search, which increased the estimated Elo from 1207 to 1692.

- **I originally planned to use a large language model to generate natural language explanations. However, studies show that visualisations improve understanding compared to verbal explanations [8]. Therefore, I designed and implemented a visualisation for the explanations instead.**

  The visualisation was successful as evaluated using the Cognitive Dimensions of notation, but I would add more features with more time as discussed below.

5.2 Reflections

Throughout the project, I learnt about the challenges of managing a long project whilst balancing my workload with other assignments. The importance of planning became very apparent during the project and in hindsight, I would have more rigorously organised my time to prevent such a busy end to the project. For future large software projects, I plan on using tools such as Trello boards to keep track of the progress of subtasks.

I have gained a lot of experience and knowledge about how chess engines work, and the difficulties of building interpretable AI systems. In the past few years, computers have transitioned to being an invaluable training and preparation tool for the top players, but I believe they can become a useful learning tool for beginner players as well. On a personal note, I was proud of my progress when my chess engine first beat me in a game. Furthermore, it has been enlightening to see the explanation for why some of my moves were so bad.

Although I would have preferred to conduct a user study to evaluate the visualisation and the explanations, the CDs of Notation framework was an insightful way to identify areas for improvement. This project was one of the first times I have designed a user interface, so it was an interesting experience to try and apply what I have learnt from the human-computer interaction lectures.

5.3 Future Work

In this section, I discuss some of the ideas I had for further work on the project. I explain why they would be an improvement and how I would go about implementing them.

- I would conduct a user study to evaluate my visualisation and the explanations. This would provide useful insights into how intuitive the visualisations are to new users. Hopefully, this would validate my analysis using the Cognitive Dimensions to ensure the changes I make would be positive. It would also allow me to evaluate the effectiveness of the explanations in helping players understand their mistakes, unlike the CDs of Notation. The user study would ask beginner and intermediate players to use the tutor tool and give their feedback on the tool overall.

- I would design and build a GUI to play chess and interact with the engine. This would allow me to further visualise the explanations and move suggestions, reducing the number of hard mental operations. The biggest improvement that this would bring is the ability to click on the best continuation to see the moves applied to the board.
- I plan on porting the core of the engine to a lower-level and more performant language, such as C++. Using Python allowed me to debug and prototype new additions rapidly. However, rewriting the move generation and search functions in a lower-level language would increase the performance allowing the search to reach a greater depth. This would improve the accuracy of the move suggestions and explanations.

Bibliography

[1] CCRL. *CCRL - Index*. Mar. 2024. URL: https://computerchess.org.uk/ccrl/404/.

[2] Deborah Yao. *DeepBlue vs Kasparov Article on AI Business*. May 2022. URL: https://aibusiness.com/ml/25-years-ago-today-how-deep-blue-vs-kasparov-changed-ai-forever.

[3] Wall Street Journal. *Computers Revolutionized Chess*. Dec. 2021. URL: https://www.wsj.com/articles/magnus-carlsen-ian-nepomniachtchi-world-chess-championship-computer-analysis-11639003641.

[4] Chess.com. *Chess is Booming!* Jan. 2023. URL: https://www.chess.com/blog/CHESScom/chess-is-booming-and-our-servers-are-struggling.

[5] the$_r$*eal*$_g$*reco*. *Evolution of a Chess Fish: What is NNUE, anyway?* Dec. 2020. URL: https://www.chess.com/blog/the_real_greco/evolution-of-a-chess-fish-what-is-nnue-anyway.

[6] DecodeChess. *About DecodeChess*. Jan. 2022. URL: https://decodechess.com/about/.

[7] Chess.com. *Chess.com Game Review Features*. Mar. 2023. URL: https://www.chess.com/news/view/chesscom-launches-game-review-v2#speech.

[8] Eliza Bobek and Barbara Tversky. "Creating visual explanations improves learning". In: *Cognitive Research: Principles and Implications* 1.1 (Dec. 2016). ISSN: 2365-7464. DOI: 10.1186/s41235-016-0031-6. URL: http://dx.doi.org/10.1186/s41235-016-0031-6.

[9] *Chess Programming Wiki*. Nov. 2023. URL: https://www.chessprogramming.org/Main_Page.

[10] Sri Vathsan. *Forsyth-Edwards Notation*. July 2021. URL: https://vathzen.medium.com/forsyth-edwards-notation-in-chess-fen-d7c75397ff5a.

[11] *FIDE Laws of Chess*. Nov. 2023. URL: https://www.fide.com/FIDE/handbook/LawsOfChess.pdf.

[12] Chess.com. *Chess.com Notation Article*. Nov. 2023. URL: https://www.chess.com/terms/chess-notation.

[13] *Chess Programming Wiki Move Encodings*. Nov. 2023. URL: https://www.chessprogramming.org/Main_Page.

[14] Frank Quisinsky. *Interviews about engine protocols with Prof. Dr. Robert Hyatt, Tim Mann and Martin Blume*. May 2002. URL: https://web.archive.org/web/20020925204655fw_/http://www.playwitharena.com/directory/interviews/interviews.htm.

[15] Tim Mann H.G.Muller. *Chess Engine Communication Protocol*. Sept. 2009. URL: https://www.gnu.org/software/xboard/engine-intf.html#8.

[16] *Wikipedia Minimax Page*. URL: https://en.wikipedia.org/wiki/Minimax.

[17] Stuart J. Russell; Peter Norvig. *Artificial Intelligence: A Modern Approach*. 2021.

[18] J R Capablanca. *Chess Fundamentals*. McKay Chess Library. New York, NY: Random House, Apr. 2006.

[19] GM Noël Studer. *Bishop vs Knight in Chess*. Jan. 2024. URL: https://nextlevelchess.blog/bishop-vs-knight/#:~:text=Open%20positions,-The%20Bishop%20thrives&text=We%20can%20see%20its%20superiority,one%20at%20the%20same%20time..

[20] Eliot Slater. *Statistics for the Chess Computer and the Factor of Mobility*. 1950. URL: https://www.eliotslater.org/index.php/chess/147-statistics-for-the-chess-computer-and-the-factor-of-mobility.

[21] Abdulaziz AlShahrani, Scott Mann, and Mike Joy. "Immediate feedback : a new mechanism for real-time feedback on classroom teaching practice". In: *International Journal on Integrating Technology in Education (IJITE)* 6.2 (June 2017), pp. 17–32. URL: https://wrap.warwick.ac.uk/93603/.

[22] Ronak Badhe. *Chess Evaluations*. Mar. 2021. URL: https://www.kaggle.com/datasets/ronakbadhe/chess-evaluations/data?select=chessData.csv.

[23] Sadra Yahyapour. *Python Profiling Guide*. Jan. 2024. URL: https://www.turing.com/kb/python-code-with-cprofile#cprofile-visualization-in-gui.

[24] *Chess Programming Wiki Perft*. Nov. 2023. URL: https://www.chessprogramming.org/Perft.

[25] H.G.Muller Tim Mann. *Chess Engine Communication Protocol*. Sept. 2009. URL: https://www.gnu.org/software/xboard/engine-intf.html#8.

[26] Judea Pearl. "The Solution for the Branching Factor of the Alpha-Beta Pruning Algorithm and its Optimality". In: *Communications of the ACM* (1982).

[27] Murray Campbell Gordon Goetsch. "Experimenting with the Null Move Heuristic in Chess". In: *AAAI Spring Symposium Proceedings* (1988).

[28] *Fishtest Wiki*. URL: https://github.com/official-stockfish/fishtest/wiki.

[29] Ronald Friederich. *Rofchade Piece Square Tables*. 2018. URL: https://www.talkchess.com/forum3/viewtopic.php?f=2&t=68311&start=19.

[30] Jeremy Silman. *How to reassess your chess*. 4th ed. 2010.

[31] Arpad Elo. *The Proposed USCF Rating System*. Aug. 1967. URL: https://uscf1-nyc1.aodhosting.com/CL-AND-CR-ALL/CL-ALL/1967/1967_08.pdf#page=26.

[32] *Stockfish Terminology*. URL: https://disservin.github.io/stockfish-docs/stockfish-wiki/Stockfish-FAQ.html.

[33] T.R.G. Green and M. Petre. "Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework". In: *Journal of Visual Languages Computing* 7.2 (1996), pp. 131–174. ISSN: 1045-926X. DOI: https://doi.org/10.1006/jvlc.1996.0009. URL: https://www.sciencedirect.com/science/article/pii/S1045926X96900099.

Project Proposal

Background

Since Deep Blue first beat Kasparov in 1997 there have been many chess engines which can far outperform the best humans in both speed and chess ability, such as Stockfish which is currently regarded as the most advanced chess engine. However there are not many chess engines which also provide explanations, or a way of interpreting why the suggested move and evaluation is correct. There is a commercially sold piece of software which does claim to achieve this, however they have not published details of how they determine the explanations.

Terms

- **ELO** - A method used to calculate the relative strength of players in chess and other zero-sum games
- **Tactic** - A move or sequence of moves which gives a short term advantage to a player, often by forcing the opposing player to choose between two negative outcomes (such as by attacking two pieces at the same time)
- **Strategy** - Long term ideas in the chess game to try to gain an advantage over the opponent, such as the structure of the pawns
- **Evaluation** - In the context of chess, the evaluation function attempts to assign a value to the advantage that either player has, in terms of the number of pawns that the advantage is equivalent to (with an advantage for the white player being positive, and negative for white)

Introduction

Chess engines such as Stockfish are becoming increasingly more powerful, and are now able to easily outperform even the best humans with much less computation/thinking time per move. They achieve this by searching through the many possible positions that could be reached from the starting position, and evaluating them based on a number of features of the position. The best move can then be determined using the minimax algorithm, by selecting the move which results in the most favourable position (in terms of the evaluation) for the player. Many players use the evaluations and continuations of moves suggested by these engines to better understand positions and improve their ability. They often achieve this by looking through the various continuations of moves that the engine suggests in order to identify the tactical or strategical advantage that one move has over another.

However, less experienced players will have more difficulty identifying these differences, since they will have encountered the tactical and strategic ideas less frequently in the games they

have played. Furthermore, many beginners will not have been explicitly taught about some tactical ideas, and may not be able to understand why a position is favourable for one player even after going deep down a path of suggested moves. Understanding why a certain move was better than another is one of the most important aspects needed to learn from a mistake made in one game, and to make a better move in a similar position in future games. My project aims to provide both move suggestions and reasons for why the move is the best available. The reasoning will be in the form of which tactical and strategical features of the position had the biggest impact on determining which move is the best available in the position.

Starting Point

The project will start from scratch, so I will devise and build the interpretable evaluation model first. This work is likely to use a library such as NumPy for the implementation of a linear regression model and any other models I use, since it will be better tested and optimized than my implementation of the same mathematical model would be.

I have an understanding of the algorithms that I will be using for the engine, from the IB Artificial Intelligence course, the IB Data Science course, the IA Algorithms course and from my own interests, however I have no experience in building a chess engine. The Part II Natural Language Processing unit of assessment that I will take in Michaelmas 2023 will also add to my existing knowledge. I know how to play chess, and have a reasonable ability and understanding of the game.

Success Criteria

- The project should be considered a success if the chess engine is able to produce evaluations of all legal chess positions, as well as the tactical and strategical aspects that contributed most to that evaluation.
- The project should be considered a success if the evaluation matches Stockfish in 80% of test positions in assigning a position to approximately equal (-1.0 to 1.0 pawns), winning for white or winning for black.
- The project should be considered very successful if I produce a chess engine that has an estimated ELO of above 1500 (estimated by testing the engine against other engines of a known ELO).
- The project should be considered very successful if it is able to produce explanations for its evaluations and moves in natural language.
- The project should be considered very successful if the explanations match the expected explanation for 80% of the positions in a test suite (these positions will come from puzzles where the best move and explanation for this move is known, and then manually reviewed to check that the explanations match closely).

Work to be Undertaken

1. **Implementing the chess engine:**
   I will implement a chess engine using the minimax algorithm. The engine will be programmed in such a way that any function that takes in a position and outputs an evalu-

ation can be used. During this stage I will test that the move generation is correct, and that all legal positions are being explored.

2. **Extracting features from a position:**
   During this stage I will write functions to encode a chess position into a set of interpretable tactical and positional features, such as the material count of each side and whether a piece is attacking two other pieces at once. I will then create the initial evaluation function by manually deciding on weights for these features. In order to decide on which features to use, I will use educational chess material such as textbooks.

3. **Extending the chess engine to provide interpretations:**
   I will extend the chess engine to use a weighted sum of the extracted features in the children positions of each node in the search tree. These feature weights will be passed up the search tree to the root node, to give weights to the most important features in determining the evaluation of the position.

4. **Fitting the evaluation function model:**
   I will fit a linear regression model using a dataset of chess positions and the evaluation assigned to each position by Stockfish. The linear regression model will assign weights to each of the features extracted from the position, which will ensure the evaluations are still interpretable. I will also explore the possibility of other models that could be used here.

5. **Evaluating the chess engine:**
   There are two aspects of the chess engine which will need to be evaluated, the performance of the chess engine measured in ELO, and the explanations of its evaluations. The accuracy of the evaluations can be evaluated by testing its ability to beat chess engines of a variety of known ELO levels. The interpretations will be more difficult to evaluate, as they are often more subjective. One approach could be to find a dataset of puzzles which have a defined explanation for which is the best move.

Extensions

1. **LLM to provide natural language explanations:**
   In order to make the explanations for the evaluation easier to understand for beginners, I could use a Large Language Model (LLM) to combine the most important factors with explanations of how they create an advantage for one player to provide a more natural explanation. The LLM could be given the suggested move, the most important tactical and strategical features of the position, explanations of what these features are and how they create an advantage for one player as part of a prompt in order to provide an explanation in natural language.

2. **Using techniques to increase search depth:**
   In order to improve the search depth that can be reached I could use search techniques such as alpha-beta pruning and iterative deepening in my chess engine to reduce the number of positions that need to be searched and evaluated at each depth. This will allow a greater search depth to be reached in the same amount of time.

Timetable

1. Michaelmas weeks 2-3 (12th October - 25th October)

- Familiarize myself with the representation of chess positions and evaluations in the dataset
- Research existing chess engines and their approaches to generating evaluations and moves
- Set up the project and the repository, researching any libraries that I will need

Deliverables

- Final project proposal (Deadline 16th October)

Other work

- Unit of assessment assignment (Deadline 26th October)

2. Michaelmas weeks 4-5 (26th October - 8th November)

   - Write a framework to support representing chess positions, and applying moves to a position
   - Write a function to generate all of the legal moves in a position, and a function to check if a position is a valid chess board
   - Write unit tests to ensure that the framework is able to determine legal moves and positions

   Other work

   - Unit of assessment assignment (Deadline 9th November)

   Milestone

   - A correct move generation function and support for applying these moves to a position

3. Michaelmas weeks 6-7 (9th November - 22nd November)

   - Write a trivial evaluation function (material count and checkmate)
   - Implement the minimax search function

   Deliverables

   - A functional chess engine

4. Michaelmas weeks 8 & Christmas Vacation Week 1 (23rd November - 6th December)

   - Decide on which features to extract from the position
   - Write the functions which extract the features from a position
   - Write an evaluation function which uses manually decided weightings of these features

   Other work

   - Unit of assessment assignment (Deadline 1st December)

   Milestone

   - Chess engine using updated evaluation function

5. Christmas Vacation weeks 2-3 (7th December - 20th December)

    - Extend the minimax algorithm to pass weightings of how much each feature contributed to the evaluation up the search tree
    - Fit a linear regression model to get new weights for the features

    Deliverables

    - Chess engine that provides weightings of features as well as evaluations

6. Christmas Vacation weeks 4-5 (21st December - 3rd January)

    - Evaluate the chess engine against other chess engines in order to estimate its ELO
    - Evaluate the explanations by testing the engine on a puzzle with known tactical or strategic themes, to see if the engine is able to identify them correctly

    Milestone

    - Completed evaluation of the core chess engine

7. Christmas Vacation weeks 6-7 (4th January - 17th January)

    - Apply alpha-beta pruning to the engine
    - Write a first draft of the progress report using the evaluations to be reviewed by my supervisor

    Deliverables

    - First draft of progress report

8. Lent weeks 1-2 (18th January - 31st January)

    - Apply iterative deepening and other techniques I come across during research to improve the depth that the engine can search to
    - Finish the progress report with feedback from supervisor

    Deliverables

    - Progress report (Deadline 2nd February)

9. Lent weeks 3-4 (1st February - 14th February)

    - Set up framework for using an LLM to convert the feature weights into an explanation in natural language
    - Prepare and present progress report

    Deliverables

    - Progress report presentation (Deadline 7th February)

    Other work

    - Unit of assessment assignment (Deadline 16th February)

10. Lent weeks 5-6 (15th February - 28th February)

- Fine tune the prompt for the LLM
- Run the evaluations for the updated versions of the chess engine

Milestone

- Final version of LLM prompt working with the rest of the chess engine

11. Lent weeks 7-8 (29th February - 13th March)

    - Begin writing preparation section of the dissertation

    Deliverables

    - Completed project

    Other work

    - Unit of assessment assignment (Deadline 15th March)

12. Easter Vacation weeks 1-2 (14th March - 27th March)

    - Finish writing preparation section of dissertation
    - Write implementation section of dissertation

    Deliverables

    - First draft of preparation and implementation sections submitted to supervisor and director of studies

13. Easter Vacation weeks 3-4 (28th March - 10th April)

    - Write evaluation and conclusion sections of dissertation

    Deliverables

    - First draft of dissertation submitted to supervisor and director of studies

14. Easter Vacation weeks 5-6 (11th April - 24th April)

    - Review feedback and make changes to the dissertation

    Deliverables

    - Final draft of dissertation submitted to supervisor and director of studies

15. Easter weeks 1-2 (25th April - 8th May)

    - Make final changes to dissertation based on continued feedback

    Deliverables

    - Project source code and dissertation (deadline 10th May)

Resources

1. For the development and writing of the project and dissertation, I will primarily use my personal laptop (M1 MacBook, 256GB SSD, 8GB memory). In case of a laptop failure, I have access to college computers in the library, spare laptops from my family, or purchasing a replacement laptop. I will use GitHub for version control, as well as Google Drive for backing up my code and any written work. The Google Drive backups will be made automatically whenever changes are made to the files. I accept full responsibility for this machine and I have made contingency plans to protect myself against hardware and/or software failure.

2. I will use a dataset of chess positions which I will acquire from the internet to be used for training and testing data. The positions will be labelled with evaluations from the Stockfish chess engine, and will be used as a source of "ground-truth" evaluations for training and testing

3. A set of puzzles with solutions which I will acquire from the internet to be used in the evaluation of the interpretations that my engine produces.

4. Access to 4 RTX 8000 GPUs through my supervisor's group, which can be used to run an LLM to generate natural language explanations.