Final Year Project Report

Full Unit – Final Report

Capitalising on Human Fallibility in Chess

Callum Moss

A report submitted in part fulfilment of the degree of

BSc Computer Science (Software Engineering)

Supervisor: Eduard Eiben

Department of Computer Science
Royal Holloway, University of London

24th April 2025

Declaration

This report has been prepared on the basis of my own work. Where other published and unpublished source materials have been used, these have been acknowledged.

Word Count:

13,767

Student Name:

Callum Moss

Date of Submission:

24/04/2025

Signature:

cmoss

Table of Contents

Abstract 4

Chapter 1: Introduction 5

1.1 Motivation: 5

1.2 Aims and Goals: 8

1.3 Project Value: 9

Chapter 2: Completed Work 10

2.1 Board Representation 10

2.2 Move Generation 10

2.3 Search 12

2.4 Evaluation 14

2.5 Zobrist Hashing and Transposition Tables 15

2.6 Universal Chess Interface (UCI) 15

2.7 Sequential Probability Ratio Test (SPRTs) 16

2.8 Testing 17

2.9 Quiescence Search 18

2.10 Reverse Futility Pruning 18

2.11 Opponent Modelling and Speculative Play 19

2.12 Gambit Performance Review 21

Chapter 3: Planning and Timeline of Development 24

3.1 Initial Specification and Planning 24

3.2 Revised Specification and Planning 25

3.3 Interim Specification and Planning 26

Chapter 4: Assessment 27

4.1 Professional Considerations for this Project 27

4.2 Self-Evaluation 28

4.3 Future Work 29

Chapter 5: Conclusion 33

Chapter 6: Project Diary and Demonstration Video 34

6.1 Project Diary 34

6.2 Demonstration Video 34

Chapter 7: Bibliography 35

Chapter 8: Glossary 37

Abstract

Chess engines often use an alpha beta search algorithm which is very effective for chess as it is a zero sum game with perfect information. However, what if your opponent is fallible? Especially when considering engine versus human games, it is not a fair assumption that the human will always find the optimal move. Therefore, I propose a new algorithm which uses opponent modelling and speculative play to win chess games in fewer moves than the standard alpha beta algorithm. My algorithm tracks the best and worst possible outcome for our opponents move and determines our evaluation of a position based on a weight between these outcomes and the likelihood of our opponent reaching the best outcome, which we determine by modelling their skill throughout the game in comparison to us.

After tests involving 5 players of varying skill levels playing 50 games against both algorithms, I found that my new search algorithm utilising opponent modelling and speculative play saw chess games finishing 23.8% faster than when using alpha beta pruning. In addition, there was no significance change to the win rate between algorithms.

This suggests that there is promise to opponent modelling and speculative play for beating opponents in fewer moves than the current standard algorithms. Due to this promise, I call for more research in this field and hope to see some work on my suggestions for improvements to my approaches.

Chapter 1: **Introduction**

1.1 Motivation:

Chess engines have long dominated human players, ever since 1997 when IBM's Deep Blue famously defeated the world chess champion Garry Kasparov [1], officially marking the end of human superiority in chess. Since then, engines have become increasingly powerful, vastly surpassing human capabilities. However, despite their overwhelming strength, engines do not defeat humans as quickly as one might expect. If an engine is so strong, why doesn't it deliver a checkmate in just a few moves? This is because of how an engine decides on moves. Most chess engines rely on variations of the **minimax algorithm** [2], which searches for optimal **lines** by assuming that the opponent will respond with their best possible move. The engine would never choose to play a move that could be exploited, even if it could also lead to a quick checkmate based on inaccuracies by its opponent.

While the minimax algorithm is logical in a two-player, zero-sum game with perfect information, it overlooks the fact that human players are fallible [3] and have a very different thought process than engines. For humans, the problem of which move is best is too complex to be solved even in analysis, therefore, they will have certain predictive hunches about risks and possible advantages. This means that it is predictable as to how a human may pick a move [4]. A means of taking advantage of this is producing an **opponent model**, such that we can begin to predict which moves a player may play, and which they could miss [5].

We propose that chess engines should account for human fallibility, and their superior capabilities. Engines should play "**risky moves**" that assume their human opponents will fail to counter effectively. This strategy would enable engines to achieve stronger positions and win more quickly and dominantly. These types of moves could be closely compared to gambits, a clever action in a game or other situation that is intended to achieve an advantage and usually involves taking a risk [6]. Hence, naming this engine "Gambit". These gambits, if not properly diffused, result in a much stronger position than if the engine had played the "safest" move. This project aims to explore whether engines can adopt this strategy to defeat human players faster and more dominantly than by following traditional **alpha-beta** lines.

Furthermore, we aim to discover and implement some factors at play to determine the likelihood of an opponent missing the counter moves and factors that should affect the engine's decision-making. Such factors include: evaluation of the current position (whether the engine is winning or losing and by how much), strength of the opponent (which will limit the risk factor), complexity of the foreseen tactic / line. We assume that if the engine is in a winning position, it is able to take more risks as humans are fallible and often can be deceived by complex tactics, bluffs and gambits. Therefore, we suggest that an engine should not make modest moves that are objectively the best, but should instead make more bold and ambitious moves when playing versus a human. It would also be interesting to see if this engine emulates human behaviour at all, and whether there is a noticeable behavioural difference between this engine and a human.

It is important to note that this approach will only work if our engine is stronger than its human opponents. This should be the case. This assumption is based on the mean average **Elo** rating of the 41 independently created chess engines, from a chess engine development discord server, being higher than the average Elo of the top 10 chess players. The average Elo of independently created chess engines was ~3223 **CCRL rating** in Blitz [7]. Using the formula: FIDE rating = 0.70*CCRL rating + 840 ELO points [8], we can determine that the average **FIDE rating** is ~3103. In comparison, the average FIDE rating of the top 10 blitz chess players in the world is ~2813 [9].

Some closely related concepts are opponent model search [10] and the **contempt** factor [11]. Explained briefly, in the opponent model search, the engine creates a profile of its opponent, and

uses this to predict what moves their opponent may respond with in various positions. This can be helpful to us as this can help us predict whether or not the opponent will spot the counter to our risky move. The contempt factor measures the level of respect an engine has for its opponent. If it has low respect, it will temporarily accept a worse position to avoid a draw with the assumption that if the game carries on, it will be able to regain its advantage and win. These aspects will be important when implementing **speculative play**.

A useful metric to calculate how quickly our engine can beat humans would be using an **EAS-Score** [12]. This can help us determine the average number of moves a game finishes in when we change any part of the decision process for making moves. Furthermore, it can tell us the number of short games it wins, the number of sacrifices made, and the number of **bad draws**, which gives us a greater idea of how the engine thinks throughout the game. Not only can we use EAS to gauge the level of success each of our changes has in reaching our goal of quick, dominant games vs humans, but we can also use **SPRT**s [13]. This allows us to see whether our engine is outperforming previous iterations of itself. This will be helpful for big changes like random mover to alpha beta, but it will be less helpful near the end of the project, as our engine is designed to play against a human rather than engines.

Typical chess engine programming standards would suggest that if an engine is stronger than its opponent, it should make use of the large contempt factor. The aim of this would be to give weight to moves that avoid draws, as a draw would be seen as a negative outcome for the better player. In other words, the engine will accept a worse position in the short term to avoid a draw if it believes in the long run, it can get a win. However, contempt has shown limited impact on effecting the outcome of the game. It does however on average increase the number of short wins.

An engine using the contempt factor will accept a slightly worse position if the engine thinks it will win given a further drawn out game by avoiding the draw. But it would never make what effectively is a blunder, in hopes that if the opponent doesn't find the line, which results in a position that would be better than if the engine made the technically best move originally. Therefore, we propose using a risk factor. If an engine is in a winning position, its confidence to take risks can increase as it has some advantage to fall back on. Furthermore, if an engine is in a losing position, it knows that it should take some risks to try and regain a winning position. Therefore, it could be argued that there is always reason to go for risky moves. The question is, what level of risk is acceptable and what factors should impact this threshold?

Here I propose three principles necessary for successful speculative play inspired by various publications for opponent modelling and speculative play, namely Andrew Reibman and Peter Jansen:

1. The fallible opponent makes mistakes with a probability close to 50%

2. The size of mistakes made by the opponent is (much) larger than the size of the mistakes made by the program (risk)

3. The program can predict with reasonable accuracy whether the opponent will make a mistake

There are two approaches I have considered to determine whether a move should be played.

Approach 1:

It is relatively simple to identify where conditions one and two apply to a position. We can use the following formula I created to calculate the necessary likelihood of a better position than alpha beta required for a line to be worth playing:

$$NSL = \frac{AB - F}{S - AB}$$

$$NecessarySuccessLikelihood = \frac{AlphaBetaLineEvaluation - FailedLineEvaluation}{SuccessLineEvaluation - AlphaBetaLineEvaluation}$$

We then multiply the necessary success likelihood by its evaluation, then store it in a 2d array. We pick the line with the highest likelihood to evaluation ratio to play.

Approach 2:

We will calculate two sets of moves, which are either successful or a failure. We will store these in two 2d arrays, where we will store the successful or failed line (whether it is positive or negative) and its corresponding ratio, calculated by the following:

$$R = L * (E - BLE)$$

$$LineLikelihoodRatio = LikelihoodOfOccurance * (BestLineEvaluation - ThisLineEvaluation)$$

We then add the two ratios of the entire sets. If above 0, then we should take a risk (it is more likely that a risky move will turn out well than not). We then play the line with the highest ratio. If below 0, it is not worth the risk. We then play the alpha-beta line.

Using either approach could easily affirm conditions 1 and 2, considering chess engines' superiority, which suggests that from the engine's perspective, it is taking low risk and its opponent is making many mistakes. The main challenge comes when addressing the third condition: to make a program to predict with reasonable accuracy whether the opponent will make a mistake. Therefore, to make Gambit effective, we must design a sophisticated opponent modelling system that makes use of various factors, such as opponent skill, without using a neural net, which would be out of the scope of this project. A neural net would not be as helpful here, as we would not be able to effectively assess what factors are helpful when developing an opponent model. For example, Leela Chess Zero uses a similar concept, contempt, to evaluate whether a move is worth playing based on whether Lc0 would be happy with a draw, given the strength of their opponent. Lc0 uses a sophisticated method of determining the size of the contempt factor, which makes use of WDL (win/draw/loss likelihoods). As Lc0 was trained on a large database of real games, it is more than likely that Lc0 incorporates some version of opponent modelling when determining WDL [14]. However, the factors and patterns that Leela takes into account when deciding the WDL are unknown, as it is a large, complex neural network. Therefore, to determine what conditions should be met when deciding on a risky move, we must make use of HCE.

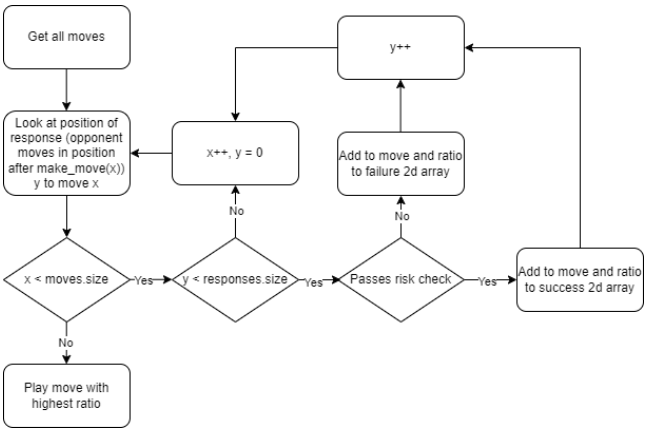

Figure 1. Search of up to depth 2

Some things to consider during the calculation of success likelihood by the opponent model are:

- Bad players are hard to predict, but lower risk. Good players are easier to predict, but higher risk
- Opponent skill will be determined throughout the game, but could have an initial value set by a puzzle quiz or Elo input. Otherwise, start with the average skill of a chess player and adjust from there.
- Complexity of position (if more move nodes, they are less likely to make the desired mistake, although they are more likely to make a mistake in general)
- Complexity of line (patterns of tactic spotting like knight forks, pieces of higher value sacrifices, etc)
- The number of missed best moves or the number of successful risky moves would indicate a higher likelihood of error.
- Stats about the player like gender, age, strength compared to engine, and self-described style. Can use this to look at stats for chess psychology.

Some things to consider during the evaluation of a position are:

- Standard principles like centipawn advantage, piece value sum, activity of pieces (how many attacks can be generated via counting the nodes of attacks), piece-square tables, etc.
- Engine vs Human strengths and weaknesses (for example, engines favour complexity, can do this by spotting of complex tactics like forks, and mid-end games whereas humans tend to excel at early game, or the solving of chess for 8 or less pieces on the board (only 5 would be viable as they exponentially get massive in size)). Humans prefer to build positions up over a long time, so subtly that the search doesn't even notice until it is too late. By being aggressive, this prevents this from happening.

1.2 Aims and Goals:

To summarize, our goal is to develop a chess engine that integrates standard chess programming principles and practices. It should also adhere to the Google C++ programming standard and follow a **TDD** workflow. We aim to design a new move decision algorithm to both enhance the engine versus human experience, but also to see if, by taking risks, engines can beat humans quicker and more dominantly than they already do, whilst maintaining at least an 80% win rate. The new move

decision algorithm should account for various factors, such as its current position (whether it is winning or losing) and whether the opponent is strong or weak. Not only will this make a more interesting experience for its opponent, but it will also attempt to capitalize on human fallibility, therefore winning quicker than other engines would by taking risks. It is important to note that this engine is not intended to compete with other chess engines (although it will do so to gain a rough evaluation of engine Elo), but is intended to make for a much more interesting experience for humans. In the process, we will understand standard AI techniques and algorithms and how they can be applied or adapted to play chess. We will also experiment with and contribute unexplored ideas for chess engine development.

We aim to determine whether we can use HCE to beat humans quicker than the average chess engine.

1.3 Project Value:

Since the publication of the idea of speculative play and opponent modelling in 1983, there has been too little research on these ideas. I believe that by applying modern knowledge of chess programming and human psychology, we can find a better approach. Some core concepts of human chess psychology have been discovered by Adriaan de Groot, who suggests, for example, that humans tend to scan the board and find the best move best on intuition, then spend time proving to themselves that it is the best [15]. This is a rather different approach than engines, as via the alpha-beta algorithm, the engine brute forces all possibilities and cuts off unpromising lines early, continuing to look at a higher depth on promising lines for a given amount of time. Although the results were not promising, this niche of chess engine programming is under-represented and researched. I believe with some expansion and development, there is a lot of potential in modelling an opponent. Finally, opponents can find satisfaction in knowing that Gambit is not playing the most optimum moves and that there is room to find vulnerabilities in a position. This could lead to a more fulfilling and engaging experience compared to other chess engines. The gameplay could be described as a constant sequence of puzzles, specifically designed off your own gameplay. This could have great benefits like helping players find weaknesses in their playstyles and improving on their own mistakes live during the game by solving these puzzles every move presented by Gambit.

Chapter 2: **Completed Work**

2.1 Board Representation

A chessboard is an 8 by 8 square with 6 types of pieces and 2 colours for each piece. We can store a single chessboard using a **bitboard**, a 64-bit unsigned integer, where a 1 bit represents an occupied square and a 0 bit represents an empty square.

We use the little-endian file-rank mapping (LERFT) to map our bits to chessboard squares. The index of the bits corresponds to squares in the following diagram, where the 0^{th} index represents A1, and 63 represents H8:

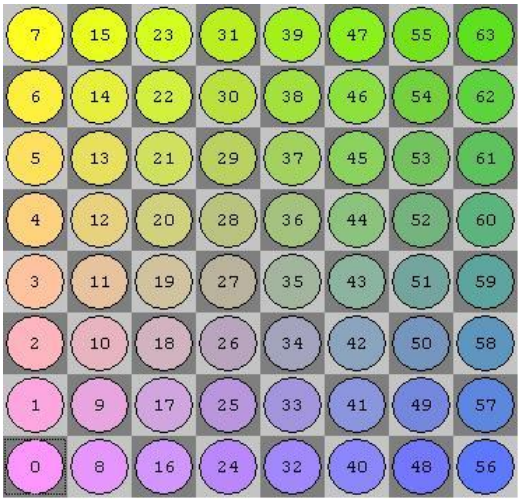

Figure 2. LERFT Bitboard Representation [16]

We have 2 colour bitboards, representing the location of white and black pieces, and we have 6 piece bitboards, representing the location of pawns, knights, bishops, rooks, queens, and kings.

By using bitboards, we are able to utilise very quick bitwise operations to get information about the current position and to update it. For example, we can get the board representation of all pieces by performing: white_pieces | black_pieces. This gives us the location of all pieces on the board. Furthermore, to get the location of specific pieces, we can perform: white_pieces & pawns. This will give us the locations of white pawns on the board [17].

2.2 Move Generation

We have three methods for generating moves for all piece types.

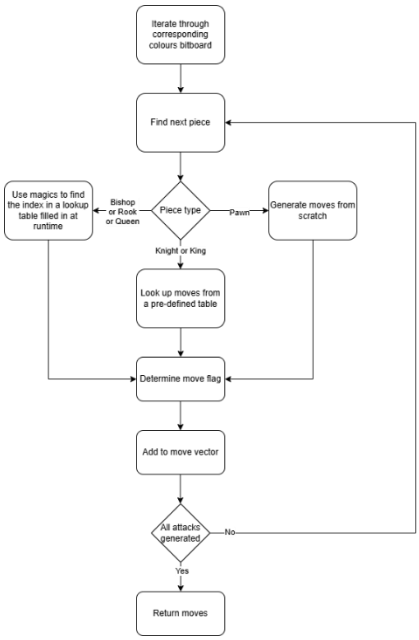

Figure 3. Move Generation

2.2.1 Pawn Moves:

Currently, we calculate pawn moves individually. We consider their location to determine whether it can move forward, forward twice, and diagonally based on a regular attack or en passant. We must also consider whether a pawn is promoting on this move, and if so, we generate 4 separate moves, one for each piece it can promote to.

2.2.2 Knight and King Moves:

To get these moves, it is rather simple. We perform a lookup in a pre-defined attacks table with the pieces square as the index. This is the most efficient way, as a lookup would just require an O(1) operation rather than calculating the move every time.

2.2.3 Bishop, Rook, and Queen Moves:

For sliding pieces, we use a technique called Magics. First, we find the blocker mask (where a mask is a bitboard) for a given piece type on a given square. Next, we must & this number with our pre-defined magic number and shift it such that only the first n bits of the mask are used. We can then use this number as an index to our attacks table, which is calculated upon initialisation once by using a slower move generation technique, which involves simply brute forcing, calculating attacks based on blockers and piece type [18].

Here is a snippet of code used to get the magic index, used to retrieve the data from the predefined magic table:

```
size_t Magics::get_magic_index(MagicEntry magic, u64 blockers)
{
// ands this blocker combination with the block attack mask
    blockers &= magic.mask;
    // Perform the magic multiplication (wrapping multiply equivalent)
    uint64_t hash = blockers * magic.magic_number;
    // Calculate the index by shifting the hash
    size_t index = static_cast<size_t>(hash >> (64 - magic.index_bits));
    return index;
 }
```

However, this requires knowing what magic numbers would not result in collisions within our table. To find magics, we simply use a brute force method of randomly generating an unsigned 64-bit integer and testing it to see if there are any collisions when used. We do this for every square for every piece type, resulting in 64 magic numbers for each sliding piece type. Once done, we can pre-define these in a table, and therefore, this magic number generation only has to occur once. Therefore, the code used to generate these numbers has been removed from the codebase.

Magic Generation:

We call find_magic(Piece piece_type, int square) to find magic numbers for a given sliding piece on a given square. For example, a bishop on a1 will have a different magic number to a rook on a1, and a bishop on a1 will have a different magic number to a bishop on a2.

```
Final_Magic Magics::find_magic(Piece piece_type, int square) {
    MagicEntry magic;
    // Bitboard of spaces where blockers would be an issue if occupied
    magic.mask = get_relevant_blocker_squares(piece_type, square);
    magic.index_bits = Utils::count_number_of_1bs(magic.mask);
    Final_Magic successful_magic;
    bb_vector magic_table;
    // Create a random device
    std::random_device rd;
    // Initialize a Mersenne Twister pseudo-random number generator
    std::mt19937_64 gen(rd());
    // Define the range of random numbers (inclusive)
    std::uniform_int_distribution<uint64_t> dis(0, UINT64_MAX);
    while(true) {
        // Magics require a low number of active bits, so we AND
        // by two more random values to cut down on the bits set.
        // Creates a random u64 magic number
        magic.magic_number = dis(gen) & dis(gen) & dis(gen);
        magic_table =
           create_magic_table(piece_type, magic.magic_number, square);
        // if full table (meaning no collisions or missed computations)
        if (magic_table.size()
           == ((std::size_t)(1 << magic.index_bits))) {
            successful_magic.magic = magic;
            successful_magic.table = magic_table;
            return successful_magic;
        }
    }
}
```

We then attempts to create a table of attacks using a randomly generated u64. If there are no collisions, this u64 works as a magic and we save the magic.

2.3 Search

Now that we are able to generate moves, we must traverse all nodes to determine which move is best. There are many search algorithms, however, the ones implemented here are known as negamax, alpha-beta beta and iterative deepening.

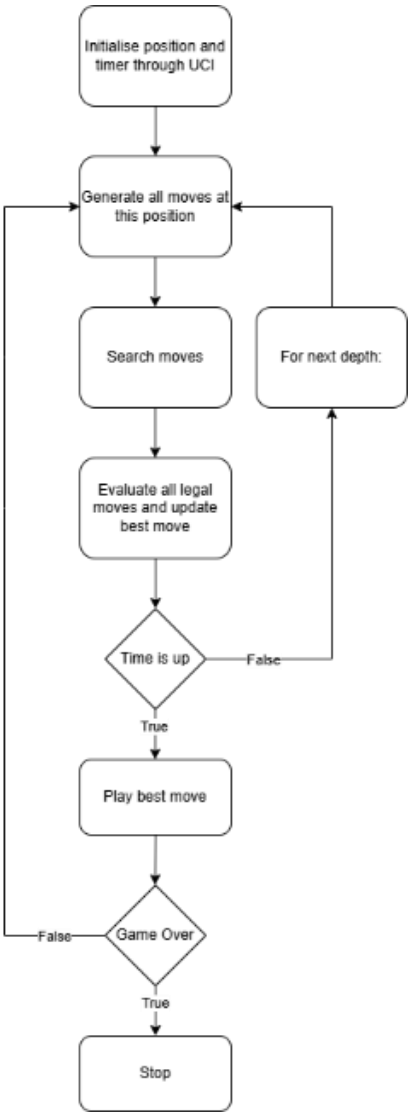

Figure 4. Basic overview of iterative deepening

2.3.1 Negamax:

This is a simpler implementation of minimax, where instead of using a minimising and maximising function, we use one function and make it slightly more general, and negate the result of the search each time we flip the turn. Hence the name negamax. This is an effective method for searching all moves, in which we perform static evaluation to get a move score, which would be used to determine the best move.

2.3.2 Alpha Beta:

Negamax is an effective algorithm, however, it is too slow. Alpha beta uses pruning to minimise the number of nodes needed to search, effectively reducing the search time. It uses the negamax algorithm but checks whether a move's score is less than another's. If said move is deemed unpromising, it assumes that the player would not play this move when there is a better alternative and therefore prunes this branch of the search tree and continues searching elsewhere. It is important to note that this pruning method is entirely sound and will always come to the same conclusion as pure negamax, just in less time. There are two approaches to this: fail soft and fail hard. We will be using fail soft as it is generally regarded as better, as it retains more information during search [19].

2.3.3 Iterative Deepening:

As chess is a time-based game, we must ensure we use some form of time management. At the start of each turn, the engine assigns an amount of time to search for before returning the best move.

By incrementing the depth of search from 1, this enhances performance as the engine will not spend too long searching for the best move each turn, but it also will not look too shallowly and potentially miss crucial moves.

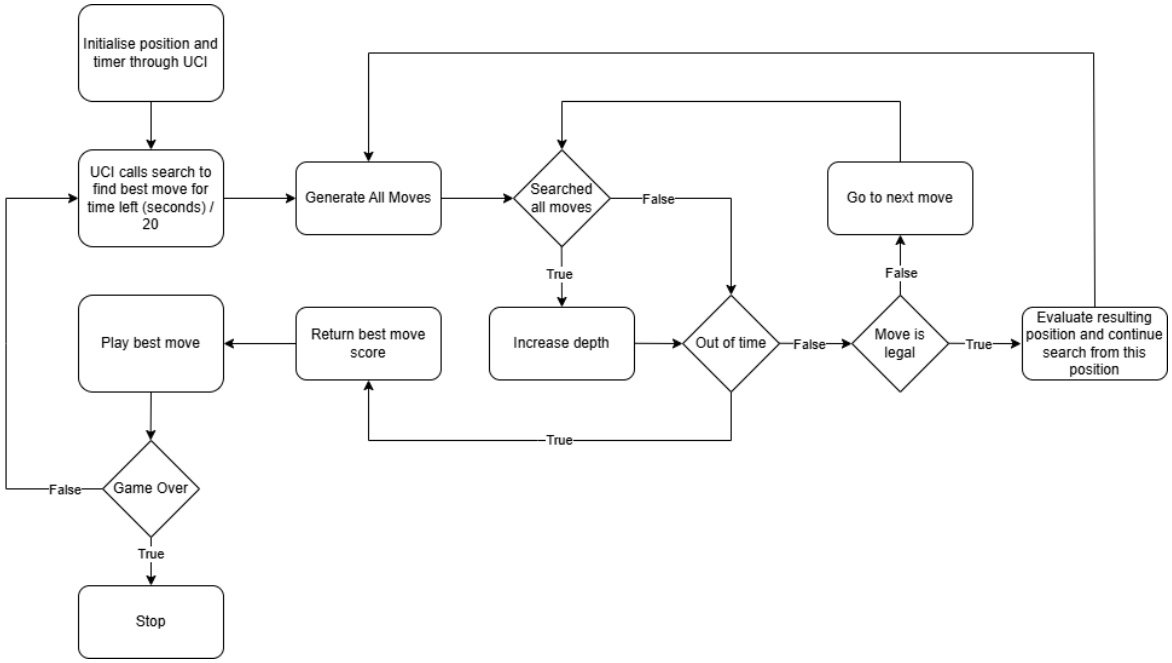

Figure 5. A deeper look into iterative deepening

I have also implemented Most Valuable Victim - Least Valuable Aggressor (MVV-LVA), a simple but effective move ordering heuristic for search. This sorts moves in a way such that the most promising moves are explored first in the search, therefore, when the timer runs out, the unexplored moves should be the least promising moves [20]. For example, if there is a pawn that can capture a queen, it will prioritise searching this before a queen captures a pawn. This has the effect of decreasing the time it takes for alpha and beta to converge, therefore reducing search time.

2.4 Evaluation

Currently, we statically evaluate a position based on either player's piece count. We sum the value of pieces for both players and determine who is winning by who has the highest sum of material. We measure piece value in centipawns. For example, a pawn is worth 100, a knight and a bishop 300, a rook 500, and a queen 900. This is an effective method for ensuring no clear blunders are made, like trading a queen for a pawn. Furthermore, we make use of piece square tables [21]. A piece should have a relative value based on the number of squares the piece can attack. This is known as a piece-square table. Intuitively, a piece on a square that can see more of the board is likely to be more useful than a piece in the corner. We also use different piece square tables depending on the phase of the game, such as early and end game. This is because a close to the end is much more valuable in the end game than early game for example.

```
int Evaluation::evaluate(const Position& pos) {
    int white_material = count_material(pos, Turn::WHITE);
    int black_material = count_material(pos, Turn::BLACK);
    bool is_endgame = false;
```

```
    if(white_material + black_material < 2200) { // if either side has a
total worth of a queen and two pawns each, endgame starts
        is_endgame = true;
    }
    int eval = white_material - black_material;
         eval   +=   calculate_piece_square_advantage(pos,   Turn::WHITE,
is_endgame); // check values from piece square tables
         eval   -=   calculate_piece_square_advantage(pos,   Turn::BLACK,
is_endgame);
    if(pos.get_turn() == Turn::WHITE) { return eval; } // turn is flipped
after make move
    else { return -eval; }
}
```

2.5 Zobrist Hashing and Transposition Tables

This is a method of representing a position as a single unsigned 64-bit number [22]. We hash our position to be able to store previous positions and compare the current position to previous ones. The Zobrist key for a position will represent the piece locations, including their colours, the castling rights, the side to move, and if there is a possible en passant. Intuitively, if all these properties of a position are the same, then it can be concluded that two positions with the same Zobrist key are also the same.

We store all the zobrist keys into a transposition table which we probe during search to check for equivalence with the current position. If the current position is already stored within the transposition table, we proceed to check the depth at which it was evaluated previously. If the depth is greater than or equal to the current search depth, we can simply return the corresponding evaluation score. If the previous search has not explored this position in sufficient depth, or is not in the transposition table, we carry on searching as normal. Whenever we encounter a unique, relevant position we add it to the transposition table along with the depth at which we found the evaluation, and the evaluation score itself. Furthermore, we can also use transposition tables to check for 3 fold repetition.

This technique greatly reduces search time as there are many ways to reach the same position, a transposition, in any give game of chess. Therefore, we can skip searching branches of the search tree that have already been explored.

2.6 Universal Chess Interface (UCI)

This allows us to communicate with various chess game hosting software and GUIs, such as Lichess and BanksiaGUI. By doing this, we can test our engine against other engines and also real people. This is crucial for evaluating the success of Gambit. It involves receiving commands from the terminal to perform various actions such as search and time management [23]. For example, we can choose whether to use Gambit features or run Gambit like a standard engine by using the Gambit toggle command during UCI communication.

A typical UCI interaction is as follows:

```
<Gambit Latest(0): id name Gambit v1.19.2
<Gambit Latest(0): id author Callum Moss
<Gambit Latest(0): uciok
>Gambit Latest(0): isready
<Gambit Latest(0): readyok
>Gambit Latest(0): ucinewgame
>Gambit Latest(0): position startpos
```

```
>Gambit Latest(0): isready
<Gambit Latest(0): readyok
>Gambit Latest(0): go wtime 30000 btime 30000
<Gambit Latest(0): depth 1 info score cp 0
<Gambit Latest(0): depth 2 info score cp 0
<Gambit Latest(0): depth 3 info score cp 50
<Gambit Latest(0): depth 4 info score cp 0
<Gambit Latest(0): depth 5 info score cp 40
<Gambit Latest(0): depth 6 info score cp 0
<Gambit Latest(0): bestmove e2e4
>Gambit Latest(0): position startpos moves e2e4 e7e5
```

Each line has value. It consists of checks for readiness, search results, desired move to play and a position to play from. Using this approach, engines can successfully communicate their intentions between each other, as well as providing useful information for other tools like evaluation graphs.

2.7 Sequential Probability Ratio Test (SPRTs)

SPRTs are a method of testing whether changes to your engine improve its performance. Typically, it involves simulating hundreds to thousands of games between the current version of your engine versus the best previous version of your engine.

Here is a typical output after an SPRT:

```
----------------------------------------------------
Results of new vs old (8+0.08, NULL, NULL, 8moves_v3.epd):
Elo: 49.67 +/- 21.98, nElo: 60.70 +/- 26.47
LOS: 100.00 %, DrawRatio: 40.48 %, PairsRatio: 1.74
Games: 662, Wins: 299, Losses: 205, Draws: 158, Points: 378.0 (57.10 %)
Ptnml(0-2): [24, 48, 134, 60, 65], WL/DD Ratio: 4.36
LLR: 2.90 (-2.25, 2.89) [0.00, 10.00]
----------------------------------------------------
```

The main figure to consider here is the log likelihood ratio (LLR). If the ratio is greater than the upper bound (2.89), we can conclude that the new version of our engine is an improvement. If the LLR is lower than the lower bound (-2.25), we can conclude that the new version of our engine has become worse. Finally, if it is somewhere in the middle, we can conclude that there has been no significant difference between the new and old versions in terms of performance. To conduct SPRTs, we use a software called FastChess [24].

It is worth noting that an SPRT test to test for Elo gain against speculative play heuristics will not be helpful, as speculative play is designed for fallible opponents, and therefore will not work on itself. This means all these feature implementations have been tested using the traditional alpha beta method. Here are the results so far (from least to most recent).

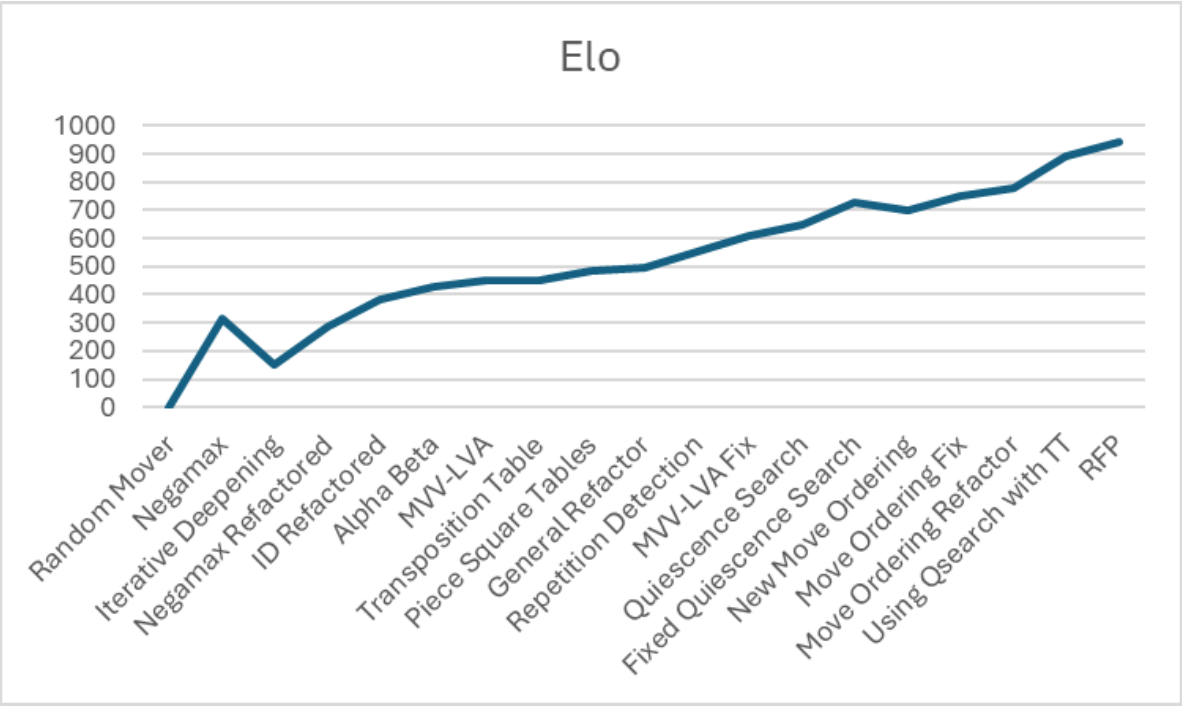

Figure 6. Elo after each feature implementation

To ensure that we get useful results, engines pick random openings from an opening book [25].

2.8 Testing

Along with SPRTs to test engine performance, we can also make use of performance testing (PERFT). This technique tests whether the number of generated move nodes by your engine is the same as the true number of possible moves for a given position, at a given depth. PERFT walks through a test suite of various positions intended to trip up an engine. At each depth, up to usually 5 – 7 depending on the complexity of the position, we check that the number of moves generated from this depth matches some pre-determined number generated from a sound engine. Working through these positions allowed me to debug many issues with my position representation and move generation. A useful technique I used during debugging was split PERFT which output the number of moves found at each depth. This allowed me to narrow down exactly where the engine deviates from the true values.

Furthermore, it also tells us how many nodes are searched per second, which can be used to test the speed of move generation in our engine. If an engine passes PEFT, in theory, the engine should be almost certainly sound in its move generation, as the likelihood of the engine getting the same number of nodes as the PERFT, when in the hundreds of thousands to millions, without being sound, is incredibly unlikely. It is important to note that I wrote a plethora of tests for move generation before implementing PERFT, but since they are incompatible with the code base and are no longer necessary, they have been removed.

Here is an example for how PERFT works:

```
std::vector<Move> moves = generate_all_moves(false);
for(Move& move : moves) {
    // Reset position
    // Create a copy of the current starting position at each depth
    Position new_position = pos;
    new_position.make_move(move); // apply move to current pos
```

```
         if(!new_position.is_legal(move)) { // if move isnt legal, don't
count
            continue;
        }
        nodes = split_perft(current_depth - 1,
            desired_depth, output_split,
            move, new_position, using_noisy);
        sum += nodes;

        if(output_split) {
            if(current_depth == desired_depth) { // if finished recursion
                 std::cout << Utils::move_to_board_notation(move) << ": "
<< nodes << std::endl;
            }
        }
    }
    return sum;
```

Using this, we can verify that the number of generated moves for various positions are equal to the true number of possible moves.

Finally, I have implemented a plethora of tests for various edge cases and used these to both design and test new features. These make use of Google Tests for C++.

2.9 Quiescence Search

At the end of each search, we conduct an extra quiescence search. This involves exploring the entire sequence of captures until the next move is not a capture. This ensures that there are no oversights during piece exchanges that perhaps would be missed if our standard alpha-beta missed it by 1 ply or more. Moves to explore are known as **noisy moves**. Depending on the implementation, the definition of a noisy move can vary. In my implementation, noisy moves simply are captures, although other implementations include checks and promotions. We can use the alpha-beta weights from our initial search to speed up convergence. Furthermore, we can also use MVV-LVA ordering, which is crucial for performing quiescence search in a timely fashion [26].

2.10 Reverse Futility Pruning

Also known as **RFP**, reverse futility pruning can be simply explained in the following instance: if we have a very significant advantage and we're not in check, we want to encourage the engine to explore potential threats from the opponent in other lines [27]. So, we cut off the node with an estimated score based on the static evaluation.

The code for this is as follows:

```
    if (depth <= 6 && !pos.in_check() && ply > 0)
    {
        int static_eval = Evaluation::evaluate(pos);
        if(static_eval - 80 * depth >= beta) return static_eval;
    }
```

This increases the rate of pruning, therefore reducing the size of the tree to traverse. This is a simple but notable gain in the speed of search, due to skipping “unnecessary” positions during search.

2.11 Opponent Modelling and Speculative Play

2.11.1 Promise Scores

Initially I attempted to implement promise scores for each line, where the promise score is the average the static evaluation of every node for each line. Intuitively, this means we can get a score which directly relates to the "promise" of each line. This therefore would favour choosing a move which has many promising outcomes rather than one critical line to play. However, this assumes that our opponent is average, rather than perfect like assumed in minimax. Whilst this sounds reasonable, we instead should choose moves with a likelihood for success against this particular opponent, not just an average player. This should correspond to taking less risks versus good players, and more risks versus bad players.

My first approach for promise scores consisted of storing the static evaluation of every node in a 2D vector for the entire search tree until our turn time limit is complete. Once search is complete, I took the average evaluation for each line and picked the move corresponding to the highest average. There are many reasons this did not work. Firstly, this approach meant average play from us as well as our opponent. This could simply be avoided by only adding the optimal move into the vector whenever it is our turn. Secondly, our first approach meant searching the entire tree and did not allow for any pruning, as doing so would skew the average. This resulted in a very inefficient search, which meant we could not search in nearly as deep as we can with alpha beta pruning. The result was searching at a depth of 3-4 in positions we normally could search at a depth of 6-7. This of course is far from ideal as we may miss crucial tactics that we may have seen at later depths.

Finally, and most importantly, the space complexity of this solution is very large. At a space complexity of $O(b^d)$, where b is the branch size (often roughly 35 on average from any given chess position) and d is the depth. This meant that the size of the vector would grow at an extremely fast rate, only at depth of 6 the size of the vector would be roughly 1.8 billion, which therefore would limit our depth of search even with unlimited time. Given these results for promise scores, I decided to use some other methods of evaluation and choosing the best line.

2.11.2 Expectimax

Here we track the best score and the worst score, where score is a static evaluation of a position. We then perform a calculation which takes into account the likelihood of playing the best and worst move in this position which returns an expected score. This approach follows very closely to what is known as expectimax [28], an implementation of minimax which accounts for probabilities. Here is the formula used to determine the expected score:

```
int Search::calc_expecti_score(int best_score, int worst_score)
{

  // Returning weighted average:
  return static_cast<int>(
    (best_score * opponent.get_probability_of_optimal())
      +
      (worst_score * (1 - opponent.get_probability_of_optimal())));
}
```

A key implementation detail here is that we only perform this calculation if it is our opponents turn. Otherwise, we know that Gambit will play what it thinks of as optimal play, and will therefore play the standard minimax-determined move. It is important to note that although this approach can use various techniques to speed up search, such as transposition tables, we currently do not use alpha beta pruning for our opponents turns as we need to find both the best and worst move for each position.

Rather intuitively, if we are playing versus a perfect opponent, this calculation simply returns the best score. As well as this, if we are playing versus an opponent that is sabotaging their game, this

calculation returns the worst score. With a position being scored lower, this would mean that Gambit should favour this line over others.

This is great as it offers a flexible approach, where Gambit can play sharper lines against weaker opponents, which may typically be punished by an engine, resulting in more favourable positions and in theory quicker games. This could lead to some tactical lines where we hope the opponent does not find the refutation, and mimics human like chess moves. Furthermore, Gambit can fall back on optimal play against strong opponents, reducing the likelihood of speculative play backfiring.

In this approach, Gambit will continue to play what it sees as the optimal move, but in this case the evaluation for positions is different. Another approach we could take is keeping the evaluation the same, and picking not the optimal move, but a move based on other factors such as risk to reward ratio. I hope to implement and test this in future versions of Gambit.

2.11.3 Updating Opponent Skill

Updating the internal skill of our opponent is vital to allow for safe speculative play. During search we add to an opponent responses table which stores the move and static score of our opponents' responses to the current best move for every depth. After we have chosen our best move, we sort this table and rank them based on their evaluation. We then wait for our opponent to make their move and update their skill corresponding to the rank of their chosen move. The calculation for this is as follows:

```
void Opponent::update_skill(int move_ranking, int move_list_size)
{
    assert(move_ranking < move_list_size);
    assert(move_ranking >= 0);
    int skill_adjustment_weight = 100;
    int middle = move_list_size / 2;
    skill += skill_adjustment_weight * (middle - move_ranking);
}
```

An example of this use-case is as follows:

- move_list_size = 40
- if best move, where index is 0, then skill += skill_adjustment_weight * (20 - 0), which if 100 then increased by 2000.
- if worst move, index is 39, then skill += skill_adjustment_weight * (20 - 39), which if 100 then decreased by 1900.
- if somewhere near the middle, then minor increase or decrease to score.

The issue with this is that the scores used to rank the opponent's responses are based off the expecti-score calculated during search. This is problematic as we rank the moves according to the opponent's perspective rather than an objective perspective which considers truly how good a move is based on alpha beta. One solution for this would be spending time pondering our opponents best moves using alpha beta and ranking them during their own turn time. This way our turn time can just be spent on our own move.

Whilst this approach works well, there are some bugs where the skill is not always being updated after our opponent has responded. This is something I must look into.

2.12 Gambit Performance Review

In this section, we discuss the experiment results of the Gambit search versus the alpha beta search. Firstly, it is important to note that configuring a robust test for this case is rather difficult, due to the nature of human involvement such as flagging (running out of time), draws and most importantly having a much smaller pool of data to analyse. Usually, a developer could run an SPRT to compare performance increases between changes, but as we are testing specifically against humans, we must conduct the test via Lichess.

I asked 5 participants of various skills to play 5 games versus both approaches, totally 50 games, and tracked the result, skill of the player and the number of moves the games ended in. One consideration was that which approach the participant started against was random to prevent player adjustment against the engine during the games from affecting the data.

2.12.1 Experiment Results

Participant Number	Elo	Average Moves Vs Alpha Beta Algorithm (Ply) (Wins Only)	Average Moves Vs Gambit Algorithm (Ply) (Wins Only)
1	2000	108.25	62.17
2	2138	93	99.75
3	1000	49	48.4
4	989	58.6	50
5	1000	66.4	54.4

Figure 7. Table of Results Comparing Alpha Beta and Expectimax

The average moves versus the Gambit search algorithm across all 5 participants was 63, whereas versus the Alpha Beta algorithm was 78.

The original data can be found here, along with the game links:


Data.pdf

2.12.2 Statistical Analysis of this Experiment

Looking at the data, and other data gathered, we can draw the following statistical conclusions:

- The Gambit search algorithm wins a game 23.8% faster than Alpha Beta.
- The Gambit search saw more improvement against lower skilled opponents than higher skilled opponents.
- The win rate for Gambit was 22/25 = 88%.
- The win rate for Alpha Beta was 21/25 = 84%.

- The win rate did not significantly change between using either algorithm.

- Participant 1 could be an outlier considering the other results. If we do not include participant 1, the average game lasted 66.75 moves against alpha beta and 63.14 against Gambit. This resulted in Gambit finishing games 5.4% faster, which is still above a significance level of 5%, suggesting there may be promise to these approaches.

2.12.3 Critical Analysis of the Quality of the Experiment

I took into consideration several factors for inconsistencies and unreliability in the data, such as player fatigue and player adjustment. As each participant had to play 10 games back to back, we can expect that their gameplay and gameplan will change throughout the games. For example, a player may begin to play worse as they get demoralised after many losses, or they may learn to play anti computer tactics, such as playing for long term positions. To counteract this, I changed which algorithm each participant would play first and second. This should mean that any significance in the change of gameplay throughout the games should be limited as it would affect the results for both algorithms equally. In addition to switching the first algorithm per participant, I also switched colours to ensure a rough split of 50/50 playing white and black. This is important as this will affect the gameplay through white's inherent advantage of moving first and it will adjust for aggressive and defensive playstyles. As well as these considerations, drawn games were redone to make the reliability of the data better as otherwise it would limit the number of examples we have for various calculations, such as the average moves per won game. Another consideration was games ending as a result to flagging. In these cases, we redid that game as it would drag down the average number of moves otherwise. Furthermore, to prevent this, I used the time control 3+1, which means every move our opponent gets 1 more second added to their time. This prevents flagging as our opponent can always play very quickly to regain time.

However, there are some limitations to the quality of this experiment. One limitation is that as the skill of participants gets higher, the Gambit search algorithm plays closer to the alpha beta chosen moves, which would limit statistical significance of the results from the lower skilled players. Another limitation is that it is difficult to model an opponent from just one game of chess. Therefore, the size of the impact opponent modelling will have on the search algorithms decision making, whilst certainly present, is not as strong as it could be if we use other methods for determining opponent skill prior to the game. Finally, a limiting factor as discussed earlier is a limited number of results to draw from. This is unfortunately inevitable in engine versus human gameplay.

Overall, whilst not perfect, this experiment did provide crucial information regarding the success of our new search algorithm, showing that there was a statistical significance in the reduction of the number of moves for the average game.

2.12.4 Further Experiments

In addition to this test, I would like to conduct a Turing test involving a human, Gambit and another engine such as Stockfish, or a more middle-of-the-range engine. The results would be interesting as we can see whether modelling our opponent and using speculative play closer resembles moves of a human. Another metric to look at besides the number of moves per game would be EAS-score as previously discussed, which can tell us a lot more useful information such as how many bad draws occurred. Finally, we should also compare Gambit's performance to other engines to get an idea of how it performs against, in theory, a very strong player. All of these tests should give us an idea as to how Gambit ranks in terms of human players, engines, whether Gambit's playstyle is truly unique, and whether our project goals have been met.

2.12.5 Experiment Conclusions

The rough Elo estimate based on roughly our experiment is about 2300 Lichess Elo, which is very respectable and higher than most players.

We should use a multi-pronged approach for reducing the length of the average game, such as other heuristics that favour aggression and keeping pieces on the board to increase the chance for an early checkmate.

Although we included high skilled opponents, our focus is not on them. As they are skilled, we played very closely to alpha beta versus these opponents, and therefore we would not expect to see a significant difference between the two algorithms versus these opponents.

A likely burden is that we cannot accurately model our opponent throughout the course of a single game. Therefore, an approach which can gain an accurate initial skill estimate would be beneficial.

Chapter 3: **Planning and Timeline of Development**

3.1 Initial Specification and Planning

My initial plans for this project were set out in a specification written during the summer as part of submitting a plan for a custom final year project.

3.1.1 Early Deliverables

- Be able to represent a board state given any FEN
- Be able to print a board to the terminal
- Be able to generate all possible moves for any given position with efficient legality checks
- Be able to test these move generations using perft
- Produce a report detailing methods used and include the key concepts used to develop the engine to its current state, such as explaining perft, the concept of bitboards and bit shifting, and move generation algorithms and optimisations such as lookup and transposition tables.

3.1.2 Late Deliverables

- Be able to search through generated moves and decide which one to play
- Be able to update board representation after a move has been played
- Be able to interact with a UI, whether a pre-existing one like cutechess-gui [29] or a custom UI
- Enable hosting on Lichess.com
- Produce a report including SPRTs for each new feature and compare it to other engines to see if these ideas have promise. Can compare to other engines using Cute Chess. In the report, highlight example moves where the new approach has taken place. For example, the engine knows move x is best, but using the new algorithm, it decides on move y, and why it has decided to choose that move, for example, it calculated that it is losing, and it has met the threshold to play a “bold” move.

3.1.3 Extensions:

- Most chess players have an Elo assigned by sites like Chess.com. However, should they not have that to hand, develop a means of evaluating their Elo
- Provide an opening book. Perhaps with well-known gambit lines and/or regular openings.
- Custom UI for interacting with Gambit
- Two modes, standard, which uses the Gambit approach, and the other uses the standard alpha-beta approach.

3.2 Revised Specification and Planning

After a meeting with my supervisor and some thought over the summer, we came up with a more comprehensive and useful step-by-step plan of development:

3.2.1 Board Representation

- 6-piece and 2 colour bitboards
- Parse FENs – consider UCI throughout the design process
- Print board

3.2.2 Move Generation

- Generate pseudo-legal moves, then add legality checks
- Pseudo-legal move gen using magic bitboards
- Knights: Precompute a lookup table for possible moves a knight could make for all 64 squares.
- Kings: Precompute a lookup table for possible moves a king could make for all 64 squares.
- Pawns: Shift them all set-wise and use masks?
- Bishops: Magics
- Rooks: Magics
- Queens: Magics (Rook glued to a bishop)
- Pawn promotions
- En Passant
- Castling
- Add legality checks
- "Make move" (function for searching, only changes its own boards whilst searching. To play a move is where UCI comes in)
- Implement make move as a random mover (which would be considered a V1)

3.2.3 UCI and Performance Testing

- Make UCI Compliant. This will enable us to communicate with GUIs and host games online.
- Performance testing (PERFT) – For various positions, generates the expected number of move nodes in a reasonable amount of time. (UCI software generally has a perft command)

3.2.4 Search Function

- Negamax
- Alpha Beta
- Consider the legality of the move just after you make it. Check for legality. If it's legal, then proceed with the search; otherwise, go and take it back. Keep a flag to have at least one legal move played. If not, then either it's a checkmate (if the king is in a check) or a stalemate.

3.2.5 Sequential Probability Ratio Test

SPRTs (Determine whether all expected move gen is accurate, and SPRT helps give a baseline of performance compared to future iterations of the project to see if additions help improve it), SPRT may have to be done after completion of search and eval as it needs to compare version 1 to version 2, although could compare it to other random movers. SPRT random mover vs various searches and evals. You SPRT to ascertain if the change improves the engine, even if the change should 'just work' it can help catch bugs in your implementation.

3.2.6 Evaluation Function

- Count material first, can make use of piece square tables and core chess principles, like castle before it is too late, develop, etc
- Incorporate the idea of Gambit (look at contempt, similar vibes, but not quite. May also want to make positions more complex when losing or against a worse opponent.

3.2.7 Testing

- Test with Lichess
- Test with the chess society people
- Conduct “Turing Test”

3.3 Interim Specification and Planning

After an interim review of the project timeline, the revised late deliverables were as follows:

- Quiescence Search
- Principal Variation
- Implement opponent modelling and risk calculation
- Evaluate the effectiveness of speculative play against humans, other engines, and determine this using an EAS-score.
- Extension: Perform a Turing test to see whether the new features will cause the engine to make more human moves.

Chapter 4: **Assessment**

4.1 Professional Considerations for this Project

Plagiarism is a vital ethical and professional concern for any academic or professional. In computer science, plagiarism can become particularly complex, as code can easily be copied or replicated, sometimes even unintentionally. It is not always easy to distinguish between legitimate inspiration and unethical copying, especially when widely known algorithms or standard solutions are involved. This is especially true in the domain of chess engine development, which has seen numerous open-source projects over the last few decades. The ease of accessing high-quality, publicly available implementations has contributed significantly to progress in this field, but it also makes upholding integrity and the standard ethical code of conduct essential. This section reflects on the ethical and professional issues surrounding plagiarism, citation, and licensing in relation to my final year project.

Oxford University defines plagiarism as: “Presenting work or ideas from another source as your own, with or without consent of the original author, by incorporating it into your work without full acknowledgement.” [30]. Chess engine design has been developed for a long time. As such, many individuals have contributed to open-source projects, and numerous techniques have emerged and matured over time. Therefore, it is our ethical duty as modern developers to correctly cite the work of others and use code and concepts with appropriate acknowledgement. In doing so, we can protect open-source ideologies that are key to accelerating and expanding our collective knowledge in this field.

A high-profile case of plagiarism in the public domain is the case of the Rybka chess engine [31]. In 2011, the then-strongest engine that dominated tournaments in the late 2000s, having won multiple world championships, was determined to have copied substantial parts of two chess engines, Fruit and Crafty, without proper attribution or compliance with their licenses. As a result, Rybka’s titles were revoked, and the author was banned from all future competitions. Fruit and Crafty were open-sourced, and had the author simply credited the two engines whilst adhering to their licenses, this issue could have been avoided. This scandal sparked debates about software licensing and discussions about the line between honest use and inspiration, and intentional copying, especially without proper credit. During development, I ensured to keep instances like this in mind. Fair and open use is vital for accelerating our field, and as such, it is my responsibility to uphold the integrity that any computer scientist should hold.

Throughout my project, I encountered several instances where I needed to walk the line between learning from others and maintaining originality. One such example is my use of the negamax algorithm. While this is a well-documented variant of minimax used in almost every chess engine, specific implementations vary widely. I consulted several online examples to understand their use of recursion, static evaluation, and flipping the signs of evaluations, but I then re-implemented the logic in my own words and structure. In such cases, even though the concept is general knowledge in the field, I ensured to cite my sources both within the code base and within this report. At the same time, this project gave me the opportunity to develop habits that I will carry into professional life, such as documenting sources carefully, citing appropriately, and reflecting critically on my development decisions.

Another example of a situation where plagiarism was a particular concern was my use of bitboards, an efficient method for representing a chessboard using 64-bit integers. The idea itself is common in modern engines, but certain tricks or optimisations (such as magic bitboards) are the result of extensive research and experimentation. When I incorporated any such idea or drew influence from specific resources, I ensured to acknowledge them, both in my report and in my source code. Even when adapting code that I modified, I cited the original authors to avoid any ambiguity.

The potential for accidental plagiarism also posed a concern. For example, it's easy to assume that something is "common knowledge" simply because it appears in multiple places, such as minimax and alpha-beta pruning. But this can result in the unintentional reuse of a specific implementation without credit. To address this, I maintained a log noting all resources I used. This not only kept my workflow transparent but also helped me avoid ethical grey areas when documenting the project.

As a computer science student at Royal Holloway, my course is accredited by the British Computing Society. As both a student and a future computer scientist, it is my duty to adhere to their code of conduct. This code of conduct emphasizes integrity, honesty, and respect for intellectual property. An example of this is that BSC requires members to "have due regard for the legitimate rights of third parties". This is not only a legal principle, but an ethical principle too. Such values ensure fair practice and make open-source collaboration viable, which plays a key role in mutual improvement and learning amongst computer scientists.

Working on this chess engine has shown me how easy it can be to blur ethical lines, especially in domains with a rich culture of code sharing, reuse, and community-driven development. At the same time, it has helped me better appreciate the importance of transparency, reflection, and honesty in the development process. By carefully documenting sources, crediting ideas appropriately, and making a conscious effort to understand and respect licensing terms, I believe I have upheld the ethical standards expected of a computer scientist. This experience has deepened my understanding of professional responsibility and has reinforced my commitment to maintaining high ethical standards in any future development work.

4.2 Self-Evaluation

One of my prouder elements of this project are developing very efficient move generation through using magics. However, currently I store generated moves at each node during search within a vector. This means during move generation; the vector is having to resize very often. Instead, I should use the following implementation which would be much more efficient [32]:

```
struct MoveList {
    array<Move, 256> moves;
    usize            length;

    constexpr MoveList() { length = 0; }

    void add(Move m) {
        assert(length < 256);
        moves[length++] = m;
    }
};
```

This will allow for more efficient move generation.

Also, I successfully implemented an efficient alpha-beta search algorithm with methods for sorting move search order to speed up convergence between alpha and beta, as well as designing methods for opponent modelling and speculative play.

Whilst I have implemented some features towards developing opponent modelling features, to influence our moves chosen, the accuracy of this is lower than I would like. If I were to undertake this project again, I would consider using a neural network to model the opponent, rather than relying on heuristics and manually crafted evaluation functions. A neural network, particularly one leveraging deep learning, could significantly improve accuracy in predicting opponent behaviour by capturing complex patterns that traditional methods might miss. This may entail training on a large dataset of games from Lichess, whilst filtering for engine vs human games, whilst rewarding

shorter engine wins. We could then fine-tune the neural network on our opponent's game history. We could adjust parameters during gameplay, but this could lead to volatility. A safer option would be to use a model trained on games from an Elo range that encompasses our opponent's Elo. We could then proceed to fine-tune on our opponent's game history if we wish. To gain access to their game history, we can make use of the Lichess API player's games command [33]. This gives us a JSON of relevant games, filtered by our current game mode, time control, piece colour, etc.

However, this approach comes with a trade-off. Neural networks often lack interpretability, meaning we would have less insight into why specific moves were chosen. This would reduce our ability to understand the engine's reasoning about human behaviour, which is a valuable and interesting aspect of this research. Furthermore, we would have to ensure that the fine-tuning process on opponents' games, should we decide to take this route, will be fast, say no longer than 30 seconds. Otherwise, opponents will have less satisfaction. During this process, we can let the user know the progress of the fine-tuning steps. This is important, as it has been shown that people are willing to wait longer if they are kept up to date with the time remaining to wait [34]. We could either create and train the neural network ourselves, or incorporate a human move predictor neural network like Maia chess to aid us during our evaluations of various positions based on the static evaluation and the probability of reaching there. The advantage of using Maia would also be to reduce the size of the tree to traverse by pruning branches that are very unlikely to occur. Maia claims to have a 50% accuracy at guessing a human's next move, which would be adequate for passing the 3rd condition for speculative play to work; the program can predict with reasonable accuracy whether the opponent will make a mistake.

I would make some adjustments to pawn attack generation. The current method for doing this involves individually generating attacks. However, we can do this much more efficiently by using setwise, bitwise operations. This is optimal due to the simple moves that pawns make, as well as the large number of them that exist. To generate moves, we should take the bitboard of our pawns and bit-shift them forward one square (>> 8 or << 8, depending on side to move), and make sure that it isn't blocked with a bitwise and. After that, we can simply loop through it to get individual moves.

Furthermore, I would like to add a feature to count the number of nodes searched. This would allow us to perform quicker operations every search. For example, during search we check whether our turn time has run out. Although, it does not make sense to check this as often as we do. Instead, we should perform a quicker operation, modulo, to check whether the number of nodes searched is equal to some number such as 1024. Then if this is true, we can check how much time we have remaining. Although this is a simple change, it would reduce the overhead of the search function.

4.3 Future Work

I would like to carry on working on this project. Here are some of my future goals for Gambit:

4.3.1 Human Behaviour in Chess

I would like to consider more human chess heuristics. There are many papers and articles relating to how humans perceive chess and how they determine their moves. For example, I should consider some of Adrian de Groot's earlier discoveries, as discussed in the introduction, of human behaviour in chess when determining whether a move is likely to be played. This should help increase the accuracy of move evaluations based on their static score and their likelihood of being played. It would also allow us to test some of Groot's observations by leveraging them against our opponents.

Consider EvH concepts:

- The opponent is less likely to catch complex lines

- They will see lines at a lower depth than the engine. (human may pick a line that looks good at depth 5, but engine realizes at depth 6 that line is bad, so they allow the human to take that line)

- Humans are great at concepts and long term planning, but will not see move by move their ideas through. Whereas an engine sees all combinations up to a given depth. This suggests that engines should prefer short and complex positions or long and simple positions.

- Humans have memory and experience, meaning there are calculations it can simply skip, allowing more time to be spent thinking about specific moves in a higher depth. This is especially prevalent in the opening. Thankfully to counter this we provide the engine with this knowledge by using opening books.

- Humans are weaker than engines

- Humans are worse than engines at figuring out complex positions (to determine complexity, a baseline for this would be number of possible depth 1 nodes, and number of pieces on the board)

- Humans are better at seeing broadly and having a general better gameplan, which they leverage by playing unprovoking moves and aim to slowly build up a dominant position before the engine can figure out it is losing. (This should be countered if an engine is actively trying to pick up the pace of the game).

4.3.2 Analysis of Opponent Modelling and Speculative Play

I would like to conduct further and deeper analysis of opponent modelling for speculative play and how successful this is against human players. I should consider the EAS-score of our various approaches compared to other more popular approaches, like standard alpha-beta, to determine the success of these approaches.

In addition, I aim to implement Bayesian Inference to get a probability distribution of our opponent's skill rather than just one number. This should allow for more accurate opponent modelling, which prevents us from taking risks against strong opponents and also prevents us from being too humble versus weak opponents [35].

Finally, I would like to get an accurate estimate for the strength of Gambit in relation to other engines. This can be achieved by playing versus various versions of the Stash engine. Stash has a list of known Elos for every version [36], and therefore we can play Gambit versus each version of Stash to get a good estimate [37].

4.3.3 Initial Elo / Skill Evaluation

I would like to develop a method for varying the initial Elo estimate of our opponent based on information we can gather before the game. This could take one of two forms:

Option 1:

We use the Lichess API to gain information about our opponent before / during the match. One option of data we could get would be the opponents' Elo. This would be very helpful in estimating our initial skill of our opponent, allowing a quicker convergence to an accurate internal skill estimate. This has the added benefit of not needing user interaction, as this happens automatically. Naturally, this could present some data privacy concerns depending on the type of information Gambit retrieves. However, as we would go through the Lichess API, we can safely assume that

the professional concerns surrounding this are addressed by Lichess, and that we cannot possibly retrieve information deemed sensitive.

Option 2:

Chess puzzles, initial skill approximation from the user, both an Elo input, and words like intermediate. It would also open more avenues to test many factors in human decision making for chess such as age. If we were to follow through with this option, there are some professional concerns that come with it. It is of the utmost importance to handle potentially sensitive data with care and with complete transparency. Of course, there is the consideration of dishonest input from our opponent, but we assume some integrity from out opponent.

It is likely that a mixture of the two approaches would yield the best results, as it can account for players who may know or not know their Elo, but also facilitate an audience based on their method of interaction. For example, if you are playing versus Gambit online, you may opt to dive straight into the game, in which case option 1 is more appropriate. However, if you are running Gambit locally, you may consider taking the time to help with option 2. This is likely, as we would be able to have a user-friendly GUI for our opponent to interact with, presenting some engaging but also helpful means of gathering information about the opponent. Whereas the alternative for this on a website like Lichess would be communication through the chat feature, which would be a lot less engaging and present fewer options for ways of gathering information. Naturally, approach 1 does rely on

4.3.4 Enhancing the User Experience

Through the Lichess API, we can implement several features for improving the user experience and potentially to assist in passing a Turing test. For example, we can set sleep periods to emulate an opponent's thinking. This can be especially helpful during the opening, where an engine is playing from an opening book. Based on personal experience, many players complain about engines playing the first 6 or so moves instantly, as it feels very unnatural and is not a satisfying experience for the player. In addition, a player actively considers that the engine is using an opening book at such a time, which could contribute to a player's feeling of unfairness or cheating, as the engine is not actually having to think. Artificial sleeping periods should not come at a big performance dip, as we can implement a feature known as pondering, or similar methods, to make this artificial sleeping time useful for deeper exploration rather than sleeping the engine for a period.

We can also chat to the user, saying various messages such as “I am impressed. I did not think you would find this move.”, or “Haha! You fell for my trap.”. This can also be used for live coaching, such as: “Unlucky. You missed this move […] to find the weakness in my position”.

Finally, I would like to host Gambit on Lichess so that anyone can play versus Gambit without having to install Gambit themselves. This could be achieved by using Oracle’s free tier.

4.3.5 Openings and Playstyle

I would also like to put together an opening book comprising of popular, aggressive gambit and trap openings, such as the Dutch Gambit, or the Traxler Counterattack. This on average should reduce the number of moves per game and force our opponent into positions where they are more likely to make a mistake.

Furthermore, I should apply weights to complex tactics, such as forcing a **zugzwang** [37]. There are many ways to detect many tactics. For example, to detect zugzwang, we can check whether any opponent move results in a static evaluation higher than the evaluation of the position before their turn. If this is the case, we have reached zugzwang. Another example could be checking for forks, where we check if a knight attack hits two pieces at once. This should result in positions that our opponents are less likely to figure out.

Finally, by getting a better idea of our opponent's playstyle, we can prune branches we think are very unlikely to be played. This should allow for a greater depth of search.

4.3.6 Principal Variation

A good idea would be to record the principal variation during search to keep track of what the engine thinks is the best sequence of moves at various depths. This will be very helpful in debugging during the implementation of various search and evaluation changes, especially in new territory where we are experimenting with ideas that are yet to be tested. It is worth noting that the implementation should be relatively trivial if we were to utilise our transposition tables.

4.3.7 Further Progression:

I would also like to spend some more time working on important steps for improving search. A few notable steps that are simple to implement and give high Elo gains would be: Late Move Reductions (LMR), Late Move Pruning (LMP), Futility Pruning (FP) and Internal Iterative Reductions (IIR) [38]. By implementing these search features, Gambit will be able to be more efficient during search, freeing up computational power to either explore at a deeper depth, or perform more sophisticated opponent modelling calculations.

4.3.8 Code Cleanup

I would like to make more use of templates. An example of where this would be useful would be alpha beta search with and without Gambit features, as well as quiescence search. This would greatly increase the legibility of the code, as well as reducing duplication, therefore aligning better with core software engineering principles such as reducing code smells. In addition, I should go ahead and try to make my code more robust by making more use of const within my code. This will prevent any unintended alteration to variables. Finally, I will ensure to properly comment my code within my HPP files, following an industry standard such as Google's standard, and use doxygen to generate helpful documentation.

Once I am satisfied with Gambit, I look forward to sharing my findings on the Chess Programming Wikipedia to help future engine developers pick up where I left off.

Chapter 5: **Conclusion**

In this dissertation, we aimed to reduce the average number of moves per chess game through the use of opponent modelling and speculative play, while maintaining a win rate of over 80%. Our approach demonstrated statistical significance in reducing the average number of moves needed to defeat an opponent, achieving a win rate of 88%. While these results are promising, there are limitations to our approach, particularly the relatively small size of our dataset. Nonetheless, the success from our experiment suggests potential, and given the abundance of possible avenues for further exploration, I encourage future research in this area. In particular, I propose next steps such as exploring neural networks or developing a more robust method for estimating an opponent's initial skill level.

Chapter 6: **Project Diary and Demonstration Video**

To run Gambit on your own system, you can follow the guide in the README.md.

6.1 Project Diary

Here you can download a copy of my work diary:


diary.txt

6.2 Demonstration Video

https://youtu.be/z3qG1FhMf5w

Chapter 7: **Bibliography**

REFERENCES

IEEE Format (using https://www.mybib.com/tools/ieee-citation-generator)

[1] Chess.com Team, "Kasparov vs. Deep Blue | The Match That Changed History," *Chess.com*, Oct. 12, 2018. https://www.chess.com/article/view/deep-blue-kasparov-chess (accessed Oct. 05, 2024).

[2] "Search - Chessprogramming wiki," *Chessprogramming.org*, 2010. https://www.chessprogramming.org/Search (accessed Oct. 05, 2024).

[3] "The Game Theory of Chess | Tidings Media," *www.tidingsmedia.org*. https://www.tidingsmedia.org/blog/the-game-theory-of-chess (accessed Oct. 06, 2024).

[4] D. de and Fernand Gobet, "Perception and memory in chess: Heuristics of the professional eye," *ResearchGate*, 1996, Chapter 9, pp.18. https://www.researchgate.net/publication/247130036_Perception_and_memory_in_chess_Heuristics_of_the_professional_eye (accessed Oct. 06, 2024).

[5] A. L. Reibman, "Non-Minimax Search Strategies for Use Against Fallible Opponents - AAAI," *AAAI*, Oct. 16, 2023. https://aaai.org/papers/00338-AAAI83-084-non-minimax-search-strategies-for-use-against-fallible-opponents/ (accessed Oct. 06, 2024).

[6] Cambridge Dictionary, "gambit," *@CambridgeWords*, Oct. 02, 2024. https://dictionary.cambridge.org/dictionary/english/gambit#google_vignette. (accessed Oct. 07, 2024)

[7] EngineProgramming, "GitHub - EngineProgramming/engine-list," *GitHub*, 2023. https://github.com/EngineProgramming/engine-list (accessed Oct. 07, 2024).

[8] "a direct comparison of FIDE and CCRL rating systems - Page 4 - TalkChess.com," *Talkchess.com*, Feb. 24, 2016. https://talkchess.com/forum3/viewtopic.php?p=661573#p661573 (accessed Oct. 07, 2024).

[9] "FIDE Ratings," *ratings.fide.com*. https://ratings.fide.com/top_lists.phtml. (accessed Oct. 08, 2024).

[10] "Opponent Model Search - Chessprogramming wiki," *Chessprogramming.org*, 2015. https://www.chessprogramming.org/Opponent_Model_Search (accessed Oct. 08, 2024).

[11] "Contempt Factor - Chessprogramming wiki," *Chessprogramming.org*, 2015. https://www.chessprogramming.org/Contempt_Factor (accessed Apr. 23, 2025)

[12] *Sp-cc.de*, 2024. https://www.sp-cc.de/files/eas_scoring_explanation.txt (accessed Oct. 08, 2024).

[13] "Sequential Probability Ratio Test - Chessprogramming wiki," *Chessprogramming.org*, 2024. https://www.chessprogramming.org/Sequential_Probability_Ratio_Test (accessed Oct. 09, 2024).

[14] "Win-Draw-Loss evaluation - Leela Chess Zero," *Lczero.org*, 2020. https://lczero.org/blog/2020/04/wdl-head/ (accessed Oct. 11, 2024).

[15] Thought and Choice in Chess. DE GRUYTER, 1978. doi: https://doi.org/10.1515/9783110800647 (accessed Oct. 11, 2024).

[16] "Square Mapping Considerations - Chessprogramming wiki," *Chessprogramming.org*, 2020. https://www.chessprogramming.org/Square_Mapping_Considerations#Little-Endian_File-Rank_Mapping (accessed Apr. 23, 2025).

[17] "Bitboard Board-Definition - Chessprogramming wiki," *Chessprogramming.org*, 2020. https://www.chessprogramming.org/Bitboard_Board-Definition#Denser_Board (accessed Apr. 23, 2025).

[18] "Magical Bitboards and How to Find Them: Sliding move generation in chess," *Analog Hors*, 2022. https://analog-hors.github.io/site/magic-bitboards/ (accessed Dec. 13, 2024).

[19] "Alpha-Beta - Chessprogramming wiki," *Chessprogramming.org*, 2015. https://www.chessprogramming.org/Alpha-Beta#Savings (accessed Dec. 13, 2024).

[20] "MVV_LVA - Creating the Rustic chess engine," *rustic-chess.org*. https://rustic-chess.org/search/ordering/mvv_lva.html (accessed Dec. 13, 2024).

[21] “Simplified Evaluation Function - Chessprogramming wiki,” *www.chessprogramming.org*. https://www.chessprogramming.org/Simplified_Evaluation_Function (accessed Dec. 13, 2024).

[22] “Zobrist Hashing - Chessprogramming wiki,” *www.chessprogramming.org*. https://www.chessprogramming.org/Zobrist_Hashing (accessed Apr. 23, 2025).

[23] “UCI Protocol,” *backscattering.de*. https://backscattering.de/chess/uci/ (accessed Oct. 09, 2024).

[24] Disservin, “Release v1.1.0-alpha · Disservin/fastchess,” *GitHub*, Oct. 2024. https://github.com/Disservin/fastchess/releases/tag/v1.1.0-alpha (accessed Apr. 23, 2025).

[25] AndyGrant, “openbench-books/8moves_v3.epd.zip at master · AndyGrant/openbench-books,” *GitHub*, 2023. https://github.com/AndyGrant/openbench-books/blob/master/8moves_v3.epd.zip (accessed Apr. 23, 2025).

[26] “MVV-LVA - Chessprogramming wiki,” *Chessprogramming.org*, 2015. https://www.chessprogramming.org/MVV-LVA (accessed Apr. 23, 2025).

[27] “Reverse Futility Pruning - Chessprogramming wiki,” *Chessprogramming.org*, 2015. https://www.chessprogramming.org/Reverse_Futility_Pruning (accessed Apr. 23, 2025).

[28] M. Simic, “Expectimax Search Algorithm | Baeldung on Computer Science,” *www.baeldung.com*, Oct. 31, 2021. https://www.baeldung.com/cs/expectimax-search (accessed Apr. 23, 2025).

[29] “cutechess/cutechess,” *GitHub*, Jan. 14, 2024. https://github.com/cutechess/cutechess (accessed Apr. 23, 2025).

[30] University of Oxford, “Plagiarism,” *University of Oxford*, 2025. https://www.ox.ac.uk/students/academic/guidance/skills/plagiarism (accessed Apr. 23, 2025)

[31] “Rybka - Chessprogramming wiki,” *Chessprogramming.org*, 2021. https://www.chessprogramming.org/Rybka (accessed Apr. 23, 2025).

[32] *Nocturn9x.space*, 2025. https://git.nocturn9x.space/Quinniboi10/Prelude/src/branch/main/src/move.h (accessed Apr. 23, 2025).

[33] “Lichess.org API reference,” *Lichess.org*, 2025. https://lichess.org/api#tag/Opening-Explorer/operation/openingExplorerPlayer (accessed Apr. 23, 2025).

[34] D. Maister, “The Psychology of Waiting Lines,” *Davidmaister.com*, 1985. https://davidmaister.com/articles/the-psychology-of-waiting-lines/ (accessed Apr. 23, 2025).

[35] Giuseppe Di Fatta, G. Haworth, and K. W. Regan, “Skill rating by Bayesian inference,” Mar. 2009, doi: https://doi.org/10.1109/cidm.2009.4938634. (accessed Apr. 23, 2025).

[36] “Releases · Morgan Houppin / stash-bot · GitLab,” *GitLab*, 2025. https://gitlab.com/mhouppin/stash-bot/-/releases (accessed Apr. 24, 2025).

[37] “CCRL,” *Computerchess.org.uk*, 2025. https://www.computerchess.org.uk/ccrl/404/cgi/compare_engines.cgi?family=Stash&print=Rating+list&print=Results+table&print=LOS+table&print=Ponder+hit+table&print=Eval+difference+table&print=Comopp+gamenum+table&print=Overlap+table&print=Score+with+common+opponents (accessed Apr. 24, 2025).

[38] *Chess.com*, 2025. https://www.chess.com/forum/view/for-beginners/why-zugzwang-and-maintaining-the-opposition-are-important (accessed Apr. 23, 2025).

[39] “Search Progression - Chessprogramming wiki,” *Chessprogramming.org*, 2024. https://www.chessprogramming.org/Search_Progression (accessed Apr. 23, 2025).

Chapter 8: **Glossary**

Bitboard: a bitstring of length 64 used to represent a chess board state.

Minimax: A search algorithm where player A attempts to maximize their position and player B attempts to minimize player A's position.

Line: A sequence of chess moves which lead to a given position.

Opponent Model: The engine creates a profile of its opponent, and uses this to predict what moves their opponent may respond with in various positions.

Risky Moves: Moves that make use of speculative play and an opponent model to choose moves more likely to succeed than the alpha beta line.

Alpha-Beta: An enhancement of minimax that prunes branches where there is a possible response to move B that results in a worse position than any response for move A.

Elo: A rating given to a chess player that represents their skill level.

CCRL Rating: The standard for an Elo assigned to engines to represent their skill level.

FIDE Rating: The standard for an Elo assigned to humans to represent their skill level.

Contempt: The contempt factor measures the level of respect an engine has for its opponent. With high respect it will accept moves which lead closer to a draw. Whereas with low respect, it will temporarily accept a worse position to avoid a draw, with the assumption it will win later in the game.

Speculative Play: Choosing to play non minimax moves, based on the assumption that your opponent will miss the correct counter move, with the goal to gain a better position than the alpha beta sequence.

EAS-Score: A score given to engines to gauge its aggression and success.

Bad Draws: Draws that should not have occurred, whether it happened too early in the game or off the back of an earlier advantage.

SPRTs: Compares the last version of your engine to your current version by playing many games against each other, to determine whether your recent modifications are impactful.

HCE: Hand-crafted evaluation method of determining which player is winning in a given position and by how much, typically in value of centipawns.

TDD: Test-driven development

Turing Test: A test conducted to determine whether a person cannot accurately determine whether they are interacting with a human or a machine.

Magics: a technique where you hash the pieces that can block a slider's attacks and use that hash as an index to look up a precomputed bitboard of attacks.

Bitboard: a 64 bit representation of a chess board state.

UCI: Universal chess interface, used for engine and user interface communication

CuteChess: An interface to conduct engine matches and also has a GUI.

BanksiaGUI: An interface to conduct engine matches and also has a GUI.

Perft: Tests move generation, time taken to generate moves and tests makemove.

Negamax: Simplified version of minimax that produces the same output.

Lichess: a popular chess website where we can host the engine to play real people.

FEN: Forsyth-Edwards Notation used as the standard for describing chess positions.

Pseudo legal: All legal moves that do not consider whether you are in check, or it results in you being in check

Mask: Board which shows the status of each square in a specific region. A mask could be for one file, or it could be for attacks of a piece. A knight attack mask would have 1s in the L positions and 0 elsewhere.

Piece-square Tables: Used to determine the rough value of a given piece type on a given square. For example, a pawn on e4 is stronger than a pawn on a2.

EvH: Engine versus human

EvE: Engine versus engine

Noisy moves: A move involving the capture of a piece

RFP: (Reverse Futility Pruning) Simple heuristic designed to increase the rate of pruning during search

Zugzwang: a position where every move a player makes will worsen their current position