Pontifícia Universidade Católica
do Rio de Janeiro

Thomas Mergener Gouvêa Mendes

Neural Alpha-Beta Pruning Chess Engine Implementation

Final Project Report

Final project report presented to the Programa de Graduação em Ciência da Computação, do Departamento de Informática da PUC-Rio in partial fulfillment of the requirements for the degree of Bacharel em Ciência da Computação.

Advisor: Prof. Augusto Cesar Espindola Baffa

Rio de Janeiro
July 2025

Pontifícia Universidade Católica
do Rio de Janeiro

Thomas Mergener Gouvêa Mendes

Implementação de Motor de Xadrez Neural com Poda Alfa-Beta

Relatório de Projeto Final apresentado ao Programa de Graduação em Ciência da Computação da PUC-Rio, do Departamento de Informática da PUC-Rio como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação.

Prof. Augusto Cesar Espindola Baffa
Orientador
Departamento de Informática – PUC-Rio

Rio de Janeiro, July 8th, 2025

Thomas Mergener Gouvêa Mendes

Computer Science student at PUC-Rio.

Bibliographic data

MENDES,Thomas Mergener Gouvêa

Neural Alpha-Beta Pruning Chess Engine Implementation / Thomas Mergener Gouvêa Mendes; advisor: Augusto Cesar Espindola Baffa. – 2025.

45 f: il. color. ; 30 cm

Projeto Final (graduação) - Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática, 2025.

Inclui bibliografia

1. Ciência da Computação – Teses. 2. Inteligência Artificial. 3. Minimax. 4. Xadrez de Computadores. 5. Poda Alfa-Beta. 6. Algoritmos de Busca. I. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. II. Título.

CDD: 004

To my beloved parents Lilian and Jayme and girlfriend Giovanna for all the support throughout my life, academic pursuit and working on this project.

Acknowledgments

My sincere gratitude to Aron Petkovski and other members of Furybench for generously supporting Illumina's test with hardware, alongside providing invaluable advice throughout the project's development. I am also deeply thankful to my advisor, Prof. Augusto Baffa, for his guidance and insightful feedback.

Abstract

MENDES,Thomas Mergener Gouvêa; Prof. Augusto Cesar Espindola Baffa (Advisor). **Neural Alpha-Beta Pruning Chess Engine Implementation**. Rio de Janeiro, 2025. 45p. Projeto Final de Graduação – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Chess engines represent a critical benchmark in artificial intelligence due to the game's complexity and well-defined rules. This project aimed to develop a functional and competitive chess playing engine, Illumina, employing the Alpha-Beta pruning algorithm to search the game tree and an Efficiently Updatable Neural Network (NNUE) to perform static evaluation of positions. The development of the engine was iterative, with SPRT testing steps on each newly integrated feature to statistically ensure strength progression or non-regression. The neural network employed for static evaluation was trained on data generated by millions of Illumina's self-play games in very fast time control. As of the writing of this paper, Illumina has been tested against several other engines and ranks highly among many other superhuman chess engines.

Keywords

Artificial Intelligence; Minimax; Computer Chess; Alpha-Beta Pruning; Search Algorithms.

Resumo

MENDES,Thomas Mergener Gouvêa; Prof. Augusto Cesar Espindola Baffa (Orientador). **Implementação de Motor de Xadrez Neural com Poda Alfa-Beta**. Rio de Janeiro, 2025. 45p. Projeto Final de Graduação – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Motores de xadrez representam um marco importante na inteligência artificial devido à complexidade do jogo e às suas regras bem definidas. Este projeto teve como objetivo desenvolver um motor de xadrez funcional e competitivo, chamado Illumina, empregando o algoritmo de poda Alfa-Beta para realizar a busca na árvore de jogo e uma Rede Neural Eficientemente Atualizável (NNUE) para realizar a avaliação estática das posições. O desenvolvimento do motor foi iterativo, com etapas de testes utilizando SPRT a cada nova funcionalidade integrada, a fim de garantir estatisticamente a progressão de força ou evitar regressões. A rede neural utilizada para a avaliação estática foi treinada com dados gerados a partir de milhões de partidas autojogadas pelo Illumina em controles de tempo muito rápidos. Até o momento da redação deste trabalho, o Illumina foi testado contra diversos outros motores e apresenta desempenho competitivo em relação a outros motores de xadrez com nível super-humano.

Palavras-chave

Inteligência Artificial; Minimax; Xadrez de Computadores; Poda Alfa-Beta; Algoritmos de Busca.

Table of contents

List of figures

List of tables

List of algorithms

Definitions

File Column on the chess board. Named from 'a' to 'h'.

Rank Row on the chess board. Named from '1' to '8'.

North/South/East/West Absolute cardinal directions on the board.

King Side Portion of the board comprising files 'e' to 'h'.

Queen Side Portion of the board comprising files 'a' to 'd'.

Capture Any move that eliminates an opposing piece from the board.

Attack A condition where a piece or square is within capture range of an opposing piece.

Promotion Pawn moves that promote them to another piece.

Check When a king is being attacked.

Checkmate When, during a check, no legal move can prevent a king from being captured on the next move.

Stalemate When, not during a check, no legal move can be played.

Ply A single player's turn to move.

Legal position A position with one king per side where no king can be captured on the current ply.

Pseudo-legal move Any move that obeys the movement rules of the piece on a given board.

Legal move A pseudo-legal move that does not lead to an illegal position.

Material Quantity of pieces weighted by type (e.g., Queens > Rooks > Bishops/Knights > Pawns).

Position A specific state of a game of chess.

Endgame Stage of the game of chess where the number of remaining pieces is very low and/or there are no queens on the board.

1
Introduction

Chess is one the most popular games worldwide, being played by millions of individuals, either online or over the board. Since the 1997 match between IBM's Deep Blue and former world champion Garry Kasparov, it has become widely accepted that computers have surpassed humans in chess playing strength (CAMPBELL; HOANE; HSU, 2002). Since then, professional and amateur players make use of several different chess engine implementations to analyze and learn from their own games (GUSEV, 2021). This widespread use of computer chess has led to several experiments and discoveries in the field of Artificial Intelligence (HEATH; ALLUM, 1997)(RISI; PREUSS, 2020).

The Alpha Beta Pruning algorithm - a specialization of Minimax - was considered the de facto algorithm for implementing the search routine of a chess engine, and is still used today in most of the strongest modern chess engines (SILVER et al., 2017). Besides Alpha Beta Pruning, the Monte Carlo Tree Search algorithm - alongside Deep Convolutional Neural Networks - has also been successfully employed, initially by Google DeepMind's AlphaZero, in 2017, which managed to defeat Stockfish 8 - the strongest available engine at the time (MAHARAJ; POLSON; TURK, 2022).

Objectives and Purposes

For this project, the main goal was to review the Alpha Beta Pruning algorithm and present a fully-fledged chess engine implementation employing it. The presented engine will have each of its features tested - functionally and strength-wise. Furthermore, the final implementation would be tested against other existing engines in order to compare its strength against modern engines.

Hopefully, Illumina's implementation and this work can serve as a reference guide for further developments in chess engines or alpha-beta pruning engines for any other game.

2
Current Situation

As of April 2022, the two dominant search algorithms for chess engines were MCTS (Monte Carlo Tree Search) and Alpha Beta Pruning, with their most prominent names being Leela Chess Zero and Stockfish, respectively (MAHARAJ; POLSON; TURK, 2022).

The MCTS algorithm gained traction after Google DeepMind's AlphaZero managed to outperform the then strongest chess engine, Stockfish 8 (SILVER et al., 2017). Besides the breakthrough of MCTS, this match also demonstrated the power of Deep Learning and Neural Networks in computer chess (SILVER et al., 2017). Before this match, Neural Networks were deemed too computationally expensive to be used by chess programs - especially Alpha Beta pruning based software - since the performance cost of evaluating each chess position using a complex Neural Network was outweighed by the high throughput of using cheap hand crafted methods to evaluate an order of magnitude higher number of positions in the same time span (SILVER et al., 2017).

Although traditionally considered too slow for Alpha Beta pruning, Neural Networks also had their breakthrough in chess engines that employ this algorithm after the success of NNUE (stylised acronym for Efficiently Updatable Neural Networks). NNUE consists in a technique - first employed in Shogi engines by japanese programmer Yu Nasu - that allows neural networks to perform well during Alpha Beta searches by incrementally updating affine transformations on the first layer of the network, as well as reducing the number of necessary calculations by abusing the sparsity of input feature matrices (NASU, 2018). The advent of NNUE has turned previous hand crafted, human knowledge based methods of evaluation obsolete, with virtually every top Alpha Beta pruning based chess engine employing this technique and achieving far stronger results than older methods (MAHARAJ; POLSON; TURK, 2022).

Chess engines are constantly being tested by several different parties and often compete in computer chess tournaments, such as TCEC (Top Chess Engine Championship) (TCEC, 2023), CCC (Computer Chess Championship) or the World Computer Chess Championship (WCCC). Besides tournaments, chess engines are also tested and rated in online rating lists such as the CCRL (Computer Chess Rating List) (CCRL, 2025).

In order to measure the playing strength of competing engines, thousands to millions of games are simulated between two engines and an Elo difference is estimated (ALBERS P.C.H; VRIES, 2001). The simulation of chess games between two engines - from the standard chess starting position - would lead to many equal games in the same playing set. To circumvent this, tests between two engines are run with the assistance of opening books. These opening books consist in a set of moves and/or initial positions that define the starting point of a simulation game, instead of starting from the initial chess position. The positions that arise from such opening books might be considered imbalances, favoring one specific side (white or black). To ensure fairness in testing, every opening O in a test set played by two engines A and B, a pair of simulation

games using O is played, in which each engine gets to play white and black in one of the games.

3
Chess Implementation

3.1
Technologies

Illumina is a command-line application developed using C++17. C++ is a broadly used language for the implementation of chess engines due to its widespread popularity, low-level constructs and availability of highly-optimizing compilers capable of producing native code, providing the necessary performance required by engines to evaluate as many chess positions as possible in the minimum time. Notable examples of successful engines implemented in C++ are Stockfish and Leela Chess Zero, with Stockfish being the strongest chess engine implementation available as of May 31st, 2024.

Illumina is aimed at – although not necessarily limited to – and several instructions from this platform will be mentioned when presenting the engine's implementation and many of its optimizations. Many assumptions will be made based on the fact that this is the intended target platform, although most of them should still be applicable to many other popular architectures.

3.2
Chess rules

Before implementing the part of the engine capable of playing out or analyzing chess positions, it is necessary to have a functional implementation of all of the rules of the game of chess. Besides being fully functional, it's also necessary that the chess game implementation is very performant, so that it does not bottleneck the engine's search. There are numerous production-ready implementations available for this task, but for educational purposes, a homemade implementation was rolled out, highlighting each necessary step for proper testing and correctness insurance.

Chess is a turn-based game, in which each player controls an army of 16 pieces of 6 different types: 8 pawns, 2 knights, 2 bishops, 2 rooks, 1 queen and 1 king. Each player controls pieces of a different color: one controls white pieces, while their opponent controls the black pieces. The game is played in a square board with 8 rows and columns, commonly called *ranks* and *files*, respectively. By convention, ranks are numbered from 1 to 8 and files are named alphabetically from `a` to `h`. Squares are pairs of files and ranks (e.g., `e4`, `b3`, `a1`, `h8`). In the standard variant of chess, the game begins with the pieces disposed in the following formation:

- White pieces occupy ranks 1 and 2, while black pieces occupy ranks 7 and 8.
- Rank 2 is occupied by all of white's pawns, while rank 7 is occupied by all of black's pawns.

– Ranks 1 and 8 are occupied by the remaining pieces of each respective army, in the following order: Rook, Knight, Bishop, Queen, King, Bishop, Knight, Rook.

In every turn, a player can choose a piece to move from one square to another. Each chess piece has a different rule set of allowed movements. These rule sets define how each piece can move around the board and even capture other pieces. Once a piece is captured, it is completely eliminated from the game. The end goal of the game is *checkmate*: a situation in which one king cannot prevent being captured on the next move, even if it was able to stay in the same square without moving. Besides checkmate, different outcomes can occur, including multiple ways to end the game in a draw.

3.3 Piece movements

As mentioned, each piece on the board has a different rule set for moving. Before discussing each individual piece moving rules, some general rules apply to all of them: Pieces can capture other pieces on the squares they land on, eliminating them from the game. Kings cannot be captured. The game ends right before this situation would arrive. If a king is in a position that would allow it to be captured by the opponent in the next move (condition named as 'check'), the king's player is forced to make a move that prevents the opponent's capture. Pieces cannot move to squares occupied by other pieces of the same color.

Considering the aforementioned rules, the following are the specific rules for moving each piece type:

- **Bishops** can move any number of squares *diagonally*, as long as there are no pieces between their source and target squares.
- **Rooks** can move any number of squares *vertically* or *horizontally*, as long as there are no pieces between their source and target squares.
- **Queens** can move any number of squares—*vertically*, *horizontally*, or *diagonally*—as long as there are no pieces between their source and target squares.
- **Knights** can move in an L-shape: two squares in one direction (horizontal or vertical), followed by one square perpendicularly. Knights *ignore all intervening pieces* when moving.
- **Kings** can move *one square in any direction*. In addition to their regular movement, kings can perform a special move called *castling*, which comes in two forms:
  - **Kingside castling:** The king moves two squares towards the kingside rook, and the rook moves to the square immediately to the left of the king.
  - **Queenside castling:** The king moves two squares towards the queenside rook, and the rook moves to the square immediately to the right of the king.

Castling is only permitted if the following conditions are met:

1. The king is not currently in check.
2. The king has not moved earlier in the game.
3. The involved rook has not moved earlier in the game.
4. None of the squares between the king and its destination are under attack.
5. None of the squares between the king and the rook are occupied.

– **Pawns** follow more complex rules:
  – **Pushes:** Pawns move forward one square (north for white, south for black), provided the target square is empty. If on their starting rank, they may optionally move two squares forward (double push), also requiring both target squares to be empty.
  – **Simple captures:** Pawns can capture diagonally forward one square (north-east/north-west for white, south-east/south-west for black) if the target square contains an opponent piece.
  – **Promotions:** When a white pawn reaches the 8^{th} rank (or black pawn the 1^{st} rank), it must be promoted to one of the following: bishop, knight, rook, or queen.
  – **Promotion captures:** If the promotion occurs via a capture, the promotion rule still applies.
  – **En passant captures:** This special capture is possible immediately after an opposing pawn performs a double push and ends up adjacent (horizontally) to the player's pawn. The player's pawn may capture the opponent pawn by moving to the square that the opponent pawn passed over.

3.4 Game end conditions

Chess has three possible outcomes: win for white, win for black, or draw. Wins occur when a player manages to checkmate the opponent's king. Draws can occur in several scenarios:

– **Threefold repetition:** when the same piece configuration (placements) on the board occurs three times.
– **Fifty-move rule:** when 50 moves (or 100 plies) are played consecutively and none of them are captures or pawn moves.
– **Insufficient material:** when both sides don't have enough material to force checkmate upon an opponent, the game ends in a draw. For example, it is impossible to force checkmate with only a king and a knight, so if a game reaches a position where one side only has a king and a knight and the other only has their king, the game ends in a draw.
– **Stalemate:** if the game reaches a position in which the player to move is not in check but has no legal moves to make, the game ends in a draw.

Chess is commonly played with the use of a timer, limiting each player's move time. When playing with time constraints, additional rules apply to the game's end conditions:

- If a player's time ends and their opponent has sufficient mating material, the player whose time expired loses.
- If a player's time ends but their opponent has no sufficient mating material, the game ends in a draw.

It is important to note that some rules may differ depending on the chess organization, event, or tournament. For example, FIDE (Fédération Internationale des Échecs, International Chess Federation) proposes slightly different rules for insufficient material draws than the USCF (United States Chess Federation). The game end conditions specified here are in widespread use across many chess players, federations, and websites, as well as major chess engines — and particularly, Illumina.

3.5 Bitboards

Due to chess being played on an 8×8 board (totaling 64 squares), a common technique employed by most chess engines – including Illumina – is *bitboards*. Bitboards consist of 64-bit sets, used to represent sets of squares on the board.

There are numerous use cases for bitboards in chess programs. A single bitboard can, for example, represent the locations of all pieces of a specific type, all possible target squares for any given pieces, or all squares between two given positions. Figure 3.1 illustrates how we can use bitboards to encode all target destinations for a knight on the c6 square. Since most modern personal computers and servers use 64-bit architectures, bitboards can seamlessly fit into registers, thus offering very high performance.

$$\begin{bmatrix} 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} = 0xa1100110a000000$$

Figure 3.1: Bitboard representation of all squares attacked by a knight on c6.

As bitboards are essentially 64-bit integers, all bitwise operations can be applied to them. They are treated as sets, allowing for set-like operations: unions via bitwise OR (`|`), intersections via bitwise AND (`&`), and complements via bitwise NOT (`~`).

In many cases, it is necessary to extract individual squares from a bitboard. To do that, most modern CPU architectures provide instructions to extract the least or most significant set bit of a 64-bit integer. Assuming the existence of a function `lsb()`, Algorithm 1 demonstrates how to iterate through every set bit in a bitboard `bb`.

Algorithm 1: Bitboard Iteration

```
1
2 // Loop while the bitboard is different than zero
3 while (bb) {
4     // lsb() returns the least significant bit on bb
5     int square = lsb(bb);
6
7     // Do any desired processing with the square index...
8
9     // Finally, pop the least significant bit from the bitboard.
10    bb &= bb - 1;
11 }
```

It is often necessary to know the number of set bits in a bitboard, as this provides the size of a set being dealt with. Modern CPU architectures such as x86 and ARM also provide popcount instructions, which return the number of set bits in an integer.

3.6 Board representation

In chess, each position comprises the following information:

- Placement of every piece
- Color to move
- Number of moves (or plies) before the last capture and/or pawn move
- Castling rights for each color and each side of the board
- En passant square (or lack thereof)
- Previous positions, at least until the last capture and/or pawn move

With all this information, it is possible to reproduce a chess game taking into account every rule. Illumina utilizes a Board class that contains all aforementioned information, as well as some redundant and/or extra data for convenience or efficiency. To represent the placement of the pieces on the board, Illumina uses a redundant data structure that is composed by: An array of 64 8-bit integers in which each of them represent a piece, and each index matches a square on the board; An array of 12 bitboards, with each containing the set of squares occupied by a specific piece (ex. black knights) Two extra bitboards containing the set of squares occupied by all pieces of each specific color. An extra bitboard - called occupancy - containing the set of all squares occupied by a piece, of any color or type.

This redundant data structure for storing piece placements is very useful, since there are different demands over the data regarding the positioning of the pieces. It is trivial to get the piece that occupies given squares by simply

accessing the pieces array, as well as performing set operations over all pieces of type t and color c is easy due to the redundant bitboards. Color to move, number of moves before the last capture and/or pawn move, castling rights and en passant square are simple integers that are incrementally updated whenever they are required to change (after a move is made on the board, for example).

Finally, the information of "previous positions, at least until the last capture and/or pawn move" is a bit more tricky to be implemented. Naively recording all positions before the current one would be too expensive since comparing them would require iterating over either all bitboards or the entirety of the pieces array. In order to achieve this requirement, Illumina efficiently generates a 64-bit key for each position and stores it on the Board class, as well as the keys of all boards that came before the current one. This information is necessary for the threefold repetition draw rule, and whenever it needs to be retrieved, a lookup on the previous positions array is performed in order to count how many times the current position appeared before (by looking for the current key in the previous positions keys). In order to generate a 64 bit hash key for each position, Illumina utilizes a technique called Zobrist Hashing, which will be discussed in the next section.

3.7 Zobrist Hashing

As mentioned before, Illumina uses *Zobrist Hashing* to generate an almost unique 64-bit key for each chess position. This method allows 64-bit keys to be updated very efficiently and incrementally as moves are played on the board, using simple bitwise XOR operations.

The first step in Zobrist hashing is to precompute a global set of 64-bit keys:

- A key $K^{\text{piece}}_{p,c,s}$ for each piece type p, color c, and square s tuple (0 for unoccupied squares),
- A key $K^{\text{cast}}_{b,c}$ for every possible castling rights combination b and color c,
- A key K^{col}_{c} for the color to move,
- A key K^{ep}_{s} for each possible en passant square s,
- One constant key D for uninitialized boards.

All the listed keys are randomly generated at engine startup. A completely empty board is initialized with the key D. Then, every change to the board incrementally updates the hash key by XORing the current key with the corresponding precomputed keys.

For example, consider a position with hash key k. Suppose a black rook moves from square c_7 to c_2, capturing a white bishop previously located at c_2. The updated hash key k' is computed as:

$$\begin{aligned} k_{\text{change}} &= K^{\text{piece}}_{\text{rook,black},c_7} \oplus K^{\text{piece}}_{\text{rook,black},c_2} \oplus K^{\text{piece}}_{\text{bishop,white},c_2} \\ k' &= k \oplus k_{\text{change}} \end{aligned}$$

Due to XOR's involutory and commutative properties, the original key k can be recovered from k' as:

$$k = k' \oplus k_{\text{change}}$$

The Zobrist hashing method thus provides a semi-unique key for each position that can be efficiently incrementally updated. Assuming a random number generator with uniform distribution across the full span of existing 64-bit unsigned integers, it is possible to use the "birthday paradox" to approximate the chance of a key collision. The total number of possible keys n is equal to the number of distinct 64 bit integers, which is 2^{64}. According to the birthday paradox, the probability P of getting a Zobrist key collision can be approximated by:

$$P \approx \frac{\sqrt{n}}{n}$$

$$P \approx \frac{\sqrt{2^{64}}}{2^{64}}$$

$$P \approx \frac{1}{2^{32}}$$

$$P \approx 2.3283 \times 10^{-10}$$

Considering the very low chance of collision (approx. 1 every 4 billion positions), Illumina can reliably count on Zobrist keys to uniquely identify positions.

3.8 Move generation

Move generation is the process of computing all possible legal moves for a given chess position. The correctness and performance of the move generation algorithm are critical for traversing the game tree effectively, especially in Alpha-Beta pruning-based engines, which must evaluate millions of positions per second to achieve strong play. An incorrect move generator—one that produces illegal or invalid moves—can lead to a range of serious issues, including crashes, incorrect evaluations, and illegal play.

One of the most effective strategies to implement a performant move generator is to precompute most movement patterns at engine startup and store them in lookup tables. This allows for efficient and mostly branchless retrieval of bitboards containing all legal target squares for a piece on any given square.

For non-sliding pieces such as knights and kings, it is sufficient to precompute their movement patterns from each of the 64 squares on the board and store these in square-indexed arrays. At runtime, generating their legal moves reduces to retrieving the corresponding bitboard from the lookup table and masking it with the bitwise negation of the friendly piece occupancy bitboard. This prevents moves into squares already occupied by allied pieces.

Pawns are slightly more complex but follow a similar pattern. Two separate tables are used: one for single and double pushes, and another for captures. Capture bitboards must be masked against enemy pieces, as pawns

only capture diagonally, while push bitboards must be masked against all occupied squares, since pawns cannot move to any occupied square. Double pushes require an additional condition: the square between the source and target must also be unoccupied. Illumina handles this by first testing for the availability of a single push; if it's not possible, the double push is also ruled out.

The most complex pieces to implement efficiently are sliders: bishops, rooks, and queens. Their movement is blocked by pieces along their line of sight. Although one could precompute 64 bitboards (one per square) with all potential moves on an empty board, actual move generation still requires processing the current occupancy to determine blocked paths. To address this, Illumina implements a technique known as *PEXT Bitboards*.

The core idea behind PEXT Bitboards is to use perfect hashing to map relevant occupancy bits to precomputed move bitboards for each slider and square. Naively precomputing move sets for every possible occupancy pattern is infeasible due to the combinatorial explosion of occupancy states. However, many of these occupancy configurations are redundant: pieces outside a slider's attack ray—or those obscured behind blockers—have no effect on its movement.

PEXT Bitboards exploit this by identifying only the relevant occupancy bits that affect a given slider's path. These bits are extracted using the PEXT (Parallel Bits Extract) x86 instruction, which is available on processors supporting the BMI2 (Bit Manipulation Instruction Set 2). This instruction efficiently compresses the occupancy bits into a compact index used to access the precomputed attack table.

Algorithm 2 illustrates the core operation of the PEXT instruction, as described in the Intel Software Developer's Manual (INTEL CORPORATION, 2022).

Algorithm 2: PEXT instruction pseudo-code

```
1   procedure PEXT(src, mask):
2     initialize output to 0
3     initialize i to 0
4     for set bit m in mask
5       if bit m is set in src
6         set bit i in output
7       end if
8       increment i by 1
9     end for
10    return output
11  end procedure
```

After receiving a source integer `src` and a mask `mask`, `PEXT` will fill the bits of an output integer o by associating the least significant set bit b on `mask` to the least significant bit (of all) on o. If and only if b is set on `src`, the least significant bit of o will be set. Then, the same process is repeated for each of the n-th least significant set bits on `mask`, associating them to the n-th least significant bit of o and setting it if the associated bit on `src` is also set. The `mask` parameter for PEXT bitboards, for a given slider p at square s, is simply the bitboard of all squares that can potentially block the line of sight of p at s. Masks are precomputed for every possible square and slider piece type.

3.8.1 Validating Move Generation

Once the precomputed attack tables are done, one for every piece type, at every square and taking occupancy into consideration when needed, they can be used to generate moves. In order to generate all moves, a loop through all the pieces of the moving side is made, looking up their respective attack tables, iterating through their attack bitboards and adding every move that is legal in the position, always respecting the conditions imposed by the rules of chess. However, these rules are tricky to implement and can cause silent, hard to reproduce errors. In order to properly test them, Perft tests were implemented.

Perft (Performance Test) is the name given to the main test applied for move generation. Despite its name, the main purpose for this test in Illumina is not testing performance, albeit being a secondary goal. The test consists of – for a given chess position and a given depth – traversing all legal moves recursively until the depth reaches zero, while counting the number of leaf nodes.

Since move generation is a long solved problem, numerous datasets of expected Perft results for several position and depth pairs exist, as well as battle tested engines that generate accurate results. Given that this data is readily available, a test set with 756 different position+depth pairs was implemented to test whether Illumina's Perft results are correct and thus help ensure Illumina's move generation works properly. The test set includes many different positions, including ones containing rare edge cases.

4
Search

4.1
Minimax and Alpha-Beta Pruning

In order to analyze positions and find good moves, chess engines employ search algorithms that traverse the game tree and calculate lines, electing the ones they deem the most favorable to the side to move. The two most commonly used search algorithms for chess engines are Alpha-Beta pruning and Monte Carlo Tree Search. Both have been exhaustively tested—especially the former—and have been successfully employed in very strong chess engines.

Illumina is an Alpha-Beta pruning based chess engine. The output of search algorithms is a score that dictates or tries to predict the outcome of a given position. In the case of Alpha-Beta pruning for chess, the output is usually an integer in which values close to zero indicate balanced or drawish positions, positive values indicate an advantage for the side to move, whereas negative values indicate a disadvantage for them. Besides the score, the search routine usually also returns the move it considers best.

The Alpha-Beta pruning algorithm consists of an optimization of Minimax that, by itself, for any given chess position, returns the exact same output that would be produced by the latter. Minimax is a brute-force algorithm that recursively searches all available moves in positions up to a specified depth, and returns the best possible score—which can be decisive or heuristic—for a specified player. Minimax stops searching on two different occasions:

1. **Terminal nodes** (i.e., nodes with no children). In the case of chess, these consist of checkmates or stalemates, since no legal moves exist. In checkmates, the routine should return a constant that equates to the lowest value that can be returned, since checkmates are considered the worst possible outcome for the receiving end. In stalemates, typically 0 is returned, since they are a draw. Illumina returns a *contempt score*—which defaults to 0—that can be configured by the user. Higher contempt values make Illumina prefer slightly worse positions to draws, while lower contempt makes it prefer draws to even slightly better positions.

2. **Depth reached zero.** When the depth reaches zero—on non-terminal nodes—the search is halted. Since the game hasn't reached a final result, no decisive score can be returned. For this reason, a heuristic approximator—also known as an evaluation function—is implemented. The goal of the evaluation function is to return a score that tries to predict the likelihood of a possible game outcome. In the case of Illumina (and most other Alpha-Beta pruning chess engines), the evaluation function returns an integer in which values closer to 0 signal apparently drawn/balanced positions, higher positive values indicate a higher probability of the side to move being the winning side, and lower

negative values indicate a higher probability of the side to move being the losing side.

Algorithm 3 illustrates a pseudo-code implementation of the Minimax algorithm for chess:

Algorithm 3: Minimax Algorithm

```
procedure minimax(p, d, m):
    if d <= 0 then return static evaluation of p for m
    if p is stalemate:
        return 0
    end if

    if current player is m:
        if p is checkmate:
            return -INFINITY
        end if
        mvs := list of legal moves in p
        best_score := -INFINITY
        for each move mv in mvs:
            play move mv in p
            s := minimax(p, d - 1, m)
            takeback mv from p
            if s > best_score:
                best_score := s
            end if
        end for
        return best_score
    else:
        if p is checkmate:
            return INFINITY
        end if
        mvs := list of legal moves in p
        worst_score := INFINITY
        for each move mv in mvs:
            play move mv in p
            s := minimax(p, d - 1, m)
            takeback mv from p
            if s < worst_score:
                worst_score := s
            end if
        end for
        return worst_score
    end if
end procedure
```

4.2 Negamax and Alpha-Beta Pruning

A simplification of the Minimax algorithm that removes the necessity of the maximizer/minimizer branch is called *Negamax*. Negamax, assuming a zero-sum game, removes the maximizer/minimizer argument and branches, and flips the sign of the score of each recursive call. Algorithm 4 illustrates a pseudo-code implementation of Negamax:

Algorithm 4: Negamax Algorithm

```
procedure negamax(p, d):
```

```
    if d <= 0 then return static evaluation of p for the side to
       move
    if p is stalemate:
        return 0
    end if
    if p is checkmate:
        return -INFINITY
    end if
    mvs := list of legal moves in p
    best_score := -INFINITY
    for each move mv in mvs:
        play move mv in p
        s := -negamax(p, d - 1)
        takeback mv from p
        if s > best_score:
            best_score := s
        end if
    end for
    return best_score
end procedure
```

Minimax and Negamax are essentially the same algorithm, generating the exact same search tree, although Negamax is arguably a cleaner version of Minimax. Chess—and many other similar games—suffer from a very high branching factor. The exponential nature of chess and its high branching factor make Minimax a very slow algorithm that only allows for very shallow searches on the game tree.

Besides that, the Minimax algorithm overworks the search—it finds an exact score for each node it visits. When searching for the best score in a chess position, however, it is not necessary to assign an exact score for all visited nodes in the search tree.

Suppose we're searching a node n in which black is the color to move. One of the sibling nodes of n that was already searched was assigned a score a for white. Note that, if no other path leads to a score higher than a for white, it will be the final score for the parent node of n, which we will refer to as p. Thus, we can assume that a is a lower bound for the final score of p. With that assumption in mind, if we found a path after n in which black forces a score lower than a for white, we can safely assume that, regardless of what other children of n would lead to an even worse outcome for white, white would already not choose that path. Hence, it is unnecessary to search other sibling nodes.

Conversely, if there's a lower bound for white's scores, due to the search's recursive nature, there's also a lower bound for black's scores—which implies an upper bound for white's scores. The Alpha-Beta pruning takes those into consideration by adding two extra score parameters to the search: *alpha* and *beta*—the lower and upper bounds for the scores of the maximizing player, respectively.

Algorithm 5 illustrates the modified Minimax algorithm with alpha-beta pruning bounds (where a is alpha and b is beta).

Algorithm 5: Minimax Algorithm with Alpha-Beta Pruning

```
procedure minimax_ab(p, d, m, a, b):
    if d <= 0 then return static evaluation of p for m
    if p is stalemate:
```

```
4           return 0
5       end if
6
7       if current player is m:
8           if p is checkmate:
9               return -INFINITY
10          end if
11          mvs := list of legal moves in p
12          for each move mv in mvs:
13              play move mv in p
14              s := minimax_ab(p, d - 1, m, a, b)
15              takeback mv from p
16              if s >= b:
17                  return b
18              end if
19              if s > a:
20                  a := s
21              end if
22          end for
23          return a
24      else:
25          if p is checkmate:
26              return INFINITY
27          end if
28          mvs := list of legal moves in p
29          worst_score := INFINITY
30          for each move mv in mvs:
31              play move mv in p
32              s := minimax_ab(p, d - 1, m, a, b)
33              takeback mv from p
34              if s <= a:
35                  return a
36              end if
37              if s < b:
38                  b := s
39              end if
40          end for
41          return b
42      end if
43  end procedure
```

The Negamax version of Alpha-Beta pruning, as illustrated by Algorithm 6, adds the same two parameters and simply assumes that the lower bound for one player is the negative counterpart of the upper bound for the opposing player.

Algorithm 6: Negamax Algorithm with Alpha-Beta pruning

```
1  procedure negamax_ab(p, d, a, b):
2      if d <= 0 then return static evaluation of p for the side to
           move
3      if p is stalemate:
4          return 0
5      end if
6      if p is checkmate:
7          return -INFINITY
8      end if
9      mvs := list of legal moves in p
10     a := -INFINITY
11     for each move mv in mvs:
12         play move mv in p
13         s := -negamax(p, d - 1, -b, -a)
14         takeback mv from p
```

```
15          if s >= b:
16              return b
17          end if
18          if s > a:
19              a := s
20          end if
21      end for
22      return a
23  end procedure
```

4.2.1 Move Ordering

While Alpha-Beta pruning significantly reduces the number of nodes visited compared to basic Minimax, its efficiency is not fixed: it is highly sensitive to the order in which moves are explored. In the best case, Alpha-Beta pruning reduces the time complexity from $O(b^d)$ to $O(b^{d/2})$, where b is the branching factor and d is the depth of the search. However, reaching this optimal performance depends on searching the best moves first.

The reason for this is that pruning occurs when a branch cannot possibly produce a better outcome than one already considered. The sooner we find such strong moves, the more aggressive the pruning can be. For example, in a position where the first move examined leads to a beta cutoff (a scenario where a move that scores higher than the current upperbound), the remaining sibling moves do not need to be searched at all. However, if the best move is searched last, no pruning happens until the very end, meaning the algorithm will behave more like the original Minimax and explore most of the tree.

This makes move ordering one of the most impactful heuristics in real-world chess engines. Techniques like the following were used in Illumina – as well as most Alpha-Beta pruning chess engines – to improve move ordering:

4.2.1.1 History Heuristic

Although human knowledge based heuristics, such as searching captures first or evaluating whether a move hangs a piece, can be good indicators of potentially good moves, one of the most effective ways of achieving good move ordering is through the history heuristic. This heuristic allows the engine to learn about the current game state while it searches.

The way this heuristic works is by keeping track of beta cutoffs generated by each moves. Several different "histories" can be used, but the most basic and impactful method is to keep a 64x64 integer array that assigns a score for each from-to square pair. Once a move generates a beta-cutoff, we increment the array entry that corresponds to the move from-to squares using a quadratic function over the searched depth. It is important that depth is valued, since higher depth searches tend to be more accurate. Then, when ordering moves in other nodes, we use the history score for each move as a sorting criterion.

Illumina's current implementation uses a commonly used formula called History Gravity to update its move histories.

4.2.1.2 Transposition Table

Transposition Tables are a critical component of Illumina and any modern chess engine. Due to the nature of chess, the same position can often be reached through multiple different move sequences—a phenomenon known as a transposition. Rather than searching these positions repeatedly, Illumina stores previous search results on these positions in a hash table indexed by the position's Zobrist Key.

When the search encounters a position, it first checks the transposition table:

- If the position is found and the stored depth is sufficient, Illumina can reuse the result—either returning the exact score or using the stored best move for move ordering by searching it before any other move. This is one of, if not the most important move ordering heuristic in Illumina, since this heuristic by itself is responsible for the largest portion of beta cutoffs in the search, thus vastly reducing the number of searched nodes.
- If the position is not found, the search continues as usual, and the result is stored in the table for future lookups.

4.2.2 Selective Search and Heuristic Pruning

Alpha-Beta pruning still maintains Minimax's exact nature, since it only skips branches that cannot affect the final result. However, Illumina's search employs additional well-known pruning techniques that speculatively cut off parts of the tree that are unlikely to yield good results.

These techniques are not guaranteed to keep the exact value of the position, but work well in practice. Some speculative pruning methods employed by Illumina include:

- **Null Move Pruning:** Assumes that in most cases, doing nothing (passing the move) leads to a worse outcome. If a null move results in a cutoff, the position is likely not critical and can be pruned.
- **Late Move Pruning:** Based on the observation that weaker moves often appear later in the move list, Late Move Pruning prunes certain moves entirely if they come after a threshold number of better-ranked candidates.
- **Futility Pruning:** At shallow depths near the leaf nodes, if the static evaluation is already too low to affect the outcome, the move is not searched further.

These methods are *heuristic-driven* and may discard valuable lines in rare edge cases, but they significantly improve performance and allow engines to search deeper in practice.

4.2.3
Search Reductions and Extensions

In addition to pruning, Illumina and other engines make use of techniques that adjust the depth of the search dynamically:

- **Reductions:** Less promising moves may be searched with reduced depth. The main example is *Late Move Reductions (LMR)*, which reduces the depth of moves based on their ordering rank. The main premise is that moves ordered at the bottom of the move list are less likely to be promising, due to all applied move ordering heuristics.
- **Extensions:** In contrast, some situations require deeper searches. For instance, if a move gives check or triggers a singular reply, the engine may extend the search depth locally to gain a better understanding of tactical implications.

These dynamic modifications allow the engine to allocate its resources more intelligently: spending more time on complex, critical lines and less time on predictable or quiet ones.

4.3
Quiescence Search

One of the limitations of Alpha Beta pruning search is the so-called horizon problem. This occurs when the search depth ends just before a significant tactical event, such as a capture or a check. This phenomenon might lead to unstable or misleading evaluations.

For example, a static evaluation might consider a position favorable for one side, unaware that a strong capture will occur in the next ply. In order to mitigate this, Illumina and other chess engines employ a technique known as Quiescence Search.

Quiescence Search extends the search at the leaves of the tree, continuing to explore only captures and promotions. Other engines go as far as considering checks or check evasions in their quiescence search implementations. The end positions being evaluated become more stable, thus mitigating the Horizon Effect.

Instead of stopping the search at depth zero, the engine invokes a special search procedure `quiescence_search()` which explores all forcing moves until the position is "quiet". Once a stable position is reached (no more forcing moves), the static evaluation is returned.

Algorithm 7: Quiescence Search with Alpha-Beta Pruning

```
1  procedure quiescence_search(p, a, b):
2      stand_pat := static evaluation of p for the side to move
3      if stand_pat >= b:
4          return b
5      end if
6      if stand_pat > a:
7          a := stand_pat
8      end if
9
10     moves := list of capture and promotion moves in p
```

```
    for each move mv in moves:
        play move mv in p
        if p is checkmate:
            score := INFINITY
        else if p is stalemate:
            score := 0
        else:
            score := -quiescence_search(p, -b, -a)
        end if
        takeback mv from p

        if score >= b:
            return b
        end if
        if score > a:
            a := score
        end if
    end for
    return a
end procedure
```

This procedure was integrated into the main search loop by replacing the base case of the depth-limited search with a call to `quiesce` rather than returning a static evaluation directly.

5
Static Evaluation

As discussed previously, the branching factor of the game of chess makes it unfeasible to perform searches to terminal nodes in most positions. For that reason, instead of searching until terminal nodes are found, the quiescence search exhausts noisy moves and returns a static evaluation on quiet nodes.

To produce this static evaluation value, an evaluation function is used. This function tries to predict the outcome (or likelihood thereof) of a game without searching any moves ahead. Several different static evaluations are used by Alpha-Beta pruning chess engines. Illumina, throughout its development, used two distinct evaluation functions: PeSTO and NNUE.

The PeSTO evaluation function is based on the publicly available evaluation function of the PeSTO chess engine (FRIEDERICH, 2020). This function assigns, for each piece type and game phase—middlegame and endgame—a material score, and for each piece type, square, and game phase tuple, a positional score.

Material scores are assigned to pieces by their mere existence on the board, while positional scores are assigned to present piece-square tuples, such as a knight on f3 or a queen on c5. Assigning values to squares allows the engine to pursue these squares for their respective pieces, which in turn incentivizes known good chess habits—such as developing pieces toward the center and moving the king away from it to prevent danger.

The PeSTO evaluation function scores were tuned by the authors of the PeSTO engine on a data set of many games using a method called *Texel Tuning*.

5.1
Game Phase and PeSTO Evaluation

As a game of chess evolves, many strategies tend to change. Usually, as the game progresses, pieces start to be traded off the board. The reduced amount of material makes checkmating a king significantly harder, which is why it is well known that later stages of the game tend to focus less on attacks and more on promoting pawns to obtain a material edge.

In order to cater to this strategic shift, different positional and material scores are assigned: one for the middlegame and another for the endgame. These scores are interpolated using a game phase factor. The game phase factor is determined by the number of pieces on the board and a material weight associated with each piece type. The game phase is defined as

$$\text{Phase} = 1 \cdot P + 3 \cdot N + 3 \cdot B + 5 \cdot R + 9 \cdot Q$$

where

- P: number of pawns
- N: number of knights
- B: number of bishops

– R: number of rooks
– Q: number of queens

According to this formula, the phase factor of the game's starting position is 78. Hence, we assume that a position with factor greater than or equal to 78 is a "pure" middlegame phase, and only middlegame scores are taken into consideration.

In order to evaluate a position with the PeSTO evaluation function, the following steps are taken:

1. Compute the game phase factor p using the aforementioned formula.
2. For every white piece, calculate its middlegame and endgame material and positional values.
3. For every calculated value, interpolate the middlegame and endgame scores with an interpolant of $\min(p, 78)/78$.
4. Sum every value and repeat all steps above for the black pieces.
5. Subtract the white sum by the black sum. If the current color to move is white, return the subtraction result; otherwise, return its opposite value.

The PeSTO evaluation function is very straightforward to implement and already serves as a good starting point for engines. However, it still fails to grasp many strategic concepts associated with chess. Traditionally, chess engines expanded on the piece-square tables used by the PeSTO evaluation functions and included many hand-crafted evaluation features, such as:

– passed pawns,
– virtual king mobility,
– pawn structure evaluation (e.g., doubled pawns, isolated pawns),

Implementing these—with proper testing—would significantly increase Illumina's understanding of positions and hence its playing strength.

5.2 NNUE

Illumina and most of the strongest modern chess engines rely on even stronger evaluation techniques based on deep learning. Namely, the strongest Alpha-Beta pruning chess engines usually employ *NNUE*s (stylised acronym for Efficiently Updatable Neural Networks).

NNUEs, as a technique, were first implemented in Shogi engines by Yu Nasu, achieving great success in improving the strength of existing engines. Originally, Artificial Neural Networks were believed—and tested—to be too inefficient for the demands of Alpha-Beta pruning, since the algorithm relies on searching millions of positions in a span of a few seconds and evaluating a neural model was considered too slow.

NNUEs greatly improve the evaluation speed of Neural Network models by incrementally tracking updates to the board. A structure called the *Accumulator* is used to keep track of the results of the affine transformation between the input vector and L1 weights. A full affine transformation is computed when a chess position is initially defined. Subsequently, after each move is played during analysis, only the accumulator values that correspond to the squares affected by each move are recalculated.

Illumina's NNUE architecture is composed of:

- a 768-input input layer,
- a single fully connected hidden layer (L1) with 512 neurons,
- and a single output.

The input $\rightarrow$ L1 pass passes through a Squared Clipped-ReLU activation function (SCReLU). This final pass yields a scalar integer value. Higher positive numbers indicate that the current position is better for the side to move, lower negative numbers indicate positions that are worse for the side to move, whereas values closer to zero indicate more equal positions.

5.3 Training the Network

As mentioned, Illumina's first static evaluation method consisted of a simple hand-crafted linear function that scored positions based on weights assigned to piece-square pairs. Although simple, combining this evaluation method with the search algorithm already yields – by human standards – a decently high level of chess play. At this point, it was possible to generate a dataset by running millions of self-play games and saving results from these games, as well as scores derived from search. This data set would then be later fed into a trainer that would train our first network.

The self-play training games consisted of 5128 and 4096 fixed node searches for Gen 1-3 and Gen 4 onward, respectively. This means that Illumina's search, for each move of the game, was hard capped at the specified number of nodes and would be interrupted as soon as it hit the provided mark. The score returned from each search would be kept in memory until each game finished, as well as the position associated with the score. Then, after each game, a sample of 12-15 position-score tuples would be picked and

logged into a file, appended by the game result (win for white, win for black, draw). Each of these tuples is what is being defined as a data point. Each generated dataset aimed to contain roughly 1 million data point per hidden layer neuron.

Finally, for the training itself, Illumina employed supervised learning – being trained in a modified version of a PyTorch based trainer, Marlinflow (MARLINFLOW. . . , 2025).

6
Testing Methodology

Once the barebones of the engine is working and it's already capable of playing games, each newly implemented feature must be thoroughly tested to prevent the integration of code that impairs the engine's strength.

To perform this quality control, engine developers pair each version of an engine that contains a newly implemented feature against the last stable version. After a set of games is played, if the new version beats the previous version by a statistically significant margin, we can safely consider it the better version and, hence, it becomes the new stable version.

Determining the correct amount of games to be played for each test is an important measure: while playing too many games may be wasteful, playing too few might yield unsound results. Hence, chess engines – as well as Illumina – have adopted SPRT (Sequential Probability Ratio Testing) as a method to provide a stopping condition.

SPRT operates under two competing hypotheses: the null hypothesis h_0, and the alternative hypothesis h_1. For Illumina – as well as many other engines –, these hypotheses correspond to two Elo difference thresholds (WALD, 1945). For example:

$$\begin{aligned} h_0 &: \quad \Delta \leq \Delta_0 \quad \text{(the new version is no stronger)} \\ h_1 &: \quad \Delta \geq \Delta_1 \quad \text{(the new version is stronger)} \end{aligned}$$

where Δ represents the Elo difference between two engine versions.

Each result from a self-play game is modeled as an independent sample — scored as 1 for a win, 0.5 for a draw, and 0 for a loss from the perspective of the test engine. SPRT evaluates the *log-likelihood ratio (LLR)* after each game:

$$\text{LLR}_n = \sum_{i=1}^{n} \log \left(\frac{P(x_i \mid h_1)}{P(x_i \mid h_0)} \right)$$

where x_i is the result of the i-th game.

The test continues until the LLR crosses one of two bounds:

- **Upper bound**: accept h_1 (the new version is stronger).
- **Lower bound**: accept h_0 (no significant improvement).

These bounds are derived from the desired Type I and Type II error probabilities, α and β, respectively:

$$A = \log \left(\frac{1-\beta}{\alpha} \right), \quad B = \log \left(\frac{\beta}{1-\alpha} \right)$$

where A is the upper bound and B is the lower bound. During Illumina's testing steps, these parameters were $\alpha = 0.05$ and $\beta = 0.10$. The β value was chosen mainly due to hardware constraints, so that tests that were more likely to fail were terminated earlier.

Chess engine SPRT tests usually take thousands of games to finish. If all games were to be played from the starting position, the likelihood of multiple games with the exact same moves being played would be considerably high. In order to prevent that, Illumina – and most chess engines – don't start their test games from chess' initial position. Instead, each game starts from a preset position – usually 8 moves after the starting position –, generated from human datasets or by engines. The files that contain those preset positions are called Opening Books, and they contain hundreds of thousands or millions of positions.

For its testing sessions, Illumina used the opening books *8moves_v3.epd* (STOCKFISH, 2025), *Pohl.epd* (POHL, 2025) and *UHO_Lichess_4852_v1.epd* (POHL, 2025).

Distributed Computing

Dozens of the tests performed at early stages of development were ran in a personal computer, with each lasting from 30 minutes to 24 hours. At later stages, however, where tests tend to take more time to finish since elo progression becomes slower, Illumina offloaded tests to a distributed network of personal and dedicated computers connected using OpenBench. OpenBench is a Python based application that coordinates multiple machines to run chess engine tests and tuning sessions (GRANT, 2025).

7
Tuning

We consider search parameters to be every search constant that is used to determine conditions and/or the execution of each search heuristic. Examples include:

- the minimum depth for Null Move Pruning,
- the reduced depth amount with respect to current depth and move order,
- the threshold for a search tree branch to be "too bad" to explore given its history score.

One of the main difficulties in tuning search parameters in chess engines is that it is impractical to perform traditional supervised learning methods due to the impossibility of generating labeled data. A commonly employed algorithm to performing this kind of tuning is known as *Simultaneous Perturbation Stochastic Approximation* (SPSA).

SPSA allows search parameters to be tuned by estimating a loss gradient. We start by assigning an initial guess for each search parameter - for Illumina, these are all scalar parameters, integer or real (SPALL, 1987).

Then, for each iteration of the algorithm, we perturb each parameter in two opposite random directions, which creates two slightly distinct versions of the engine parameter set. These two versions are then paired against each other in game simulations—similarly to SPRT, but with far fewer games (for Illumina, each SPSA iteration consisted of 32 games).

Once the faceoff is complete, we can approximate the loss gradient based on the performance difference—in this case, the win/draw/loss ratio—between the two versions. With the approximated gradient, we tune the parameters in the direction of the gradient and repeat the same process.

8
Results

As of 27 June 2025, Illumina stands in 89-90th place at the CCRL (Computer Chess Rating Lists) blitz ranking out of 800 worldwide engines, with an elo rating of 3439 (+-16). CCRL conducts tests against several other chess engines by running matches and computing their elo ratings from the game results. Although no significant amount of testing was made against humans, Illumina's results against other engines – including results displayed at CCRL – strongly suggest notably superhuman playing strength, considering its superior performance against known superhuman engines, such as Stockfish 8.

Several changes – ranging from search heuristics, move ordering features, tuning sessions, training experiments and performance optimizations – were tested using SPRT during Illumina's development. In this chapter, we'll present a cherry-picked collection of test results including each of the aforementioned categories of implementations, alongside a brief explanation of the functioning and purposes for each implementation.

A important caveat of the results obtained is that the amount of elo gained by each feature is related to how strong the engine already was at the time the feature was implemented. Some features, such as Killer Moves, yield considerable strength results when implemented in early stages, but its effectiveness diminishes as the engine progresses. Conversely, some features such as Singular Extensions have been observed to become even more effective as the engine gets more developed.

Unless stated otherwise, all tests presented here were performed using a 8s+0.08s time control for games and a slightly unbalanced opening book.

8.1
Null Move Pruning

We define a null move as being a move that moves no piece. In other words, a turn skip. The Null Move Pruning heuristic derives from the assumption that, if we play a null move and we still fail high after a lower depth search, our position is good by a safe margin and no further search is required on this node (GOETSCH G.; CAMPBELL, 1990). Table 8.1 displays Null Move Pruning test results.

Table 8.1: Null Move Pruning SPRT Results

Games	**Wins**	**Losses**	**Draws**	**Elo**	**+/−**
1590	877	523	190	78.7	16.4

Although proven to be a very sound heuristic for Alpha-Beta pruning engines, the 78.7 elo figure is likely due to the fact that it was one of the first heuristics implemented in Illumina. Besides that, Illumina's first implementation of the heuristic contained a bug that, instead of reducing the search depth, it capped it to a certain value. For shorter time controls, this

change was less noticeable, but it caused severe scaling problems when playing in longer time controls. Table 8.2 displays further results on NMP, after this bug was fixed.

Table 8.2: Null Move Pruning Depth Problem Fix Results

Games	**Wins**	**Losses**	**Draws**	**Elo**	**+/−**
11088	2582	2358	6148	7.0	3.7

The Null Move Pruning, or Null Move Heuristic, has a well known pitfall when dealing with *Zugzwang* positions. These are typically defined as positions in which, had it been legal to do so, the best move would be to skip the current player's move. Consider the position in figure 8.1 with black to move.

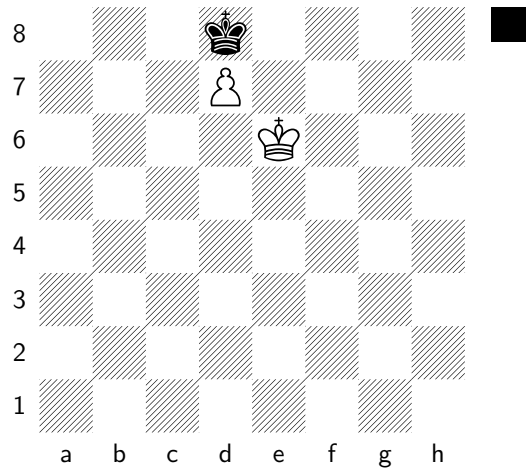

Figure 8.1: Pawn endgame *zugzwang* example

In figure 8.1's position, the player with the black pieces only has one legal move – Kf7 – which is followed by Ke7 by white, seizing control of the d8 square. When this happens, black is unable to prevent the promotion of the white's pawn to a king, leading to a forced checkmate position. If skipping a move was possible, black would be able to remain on the same square and draw the game. If not properly implemented, Null Move Pruning might cause the search to skip black's move and consider this position a draw. Initially, Illumina dealt with this by simply not applying the heuristic when the number of pieces on the board (including kings) was less than 4. However, pawn endgames are empirically known to have several *zugzwang* positions. To handle that, a condition – that has already proven successful in the past with most other engines – was added to Illumina that prevents the heuristic from being applied if the only pieces on the board are kings and pawns.

Table 8.3: Null Move Pruning with Pawn Endgame Condition

Games	**Wins**	**Losses**	**Draws**	**Elo**	**+/−**
6913	1201	1045	4667	7.8	4.7

8.2
Network Evolution

Illumina's evaluation network, as discussed in chapter 5, consists of a fully connected perceptron with a single hidden layer, using initially **CReLu** and then **SCReLu** as the activation functions.

Table 8.4 displays the test results from each of Illumina's network architecture upgrade.

Generation	Architecture	Data Points	Elo	+/−
1	768 → 64 → 1	~ 70M	60.6	+/- 14.2
2	768 → 96 → 1	~ 90M	96.1	+/- 18.2
3	768 → 128 → 1	~ 130M	122.5	+/- 21.0
4	768 → 256 → 1	~ 200M	56.4	+/- 11.8
5	768 → 512 → 1	~ 500M*	55.6	+/- 14.4

Table 8.4: Network generations and training information for Illumina's NNUE evaluation function.

* Generation 5 dataset reused 200M datapoints from Gen 4.

Table 8.5 displays the test results from switching the activation function from **CReLu** to **SCReLu**. The reason for the positive results probably stems from how **SCReLu** is better at modeling the non-linear nature of chess evaluations, and has also been observed in most major engines.

Table 8.5: CReLu → SCReLu

Games	Wins	Losses	Draws	Elo	+/−
8269	2418	2165	3686	10.6	5.6

8.3
Tuning Sessions

As discussed in chapter 7, the results from table 8.6 are from tests after applying SPSA tuning to the engine's internal parameters.

Table 8.6: SPSA Session Results

#	Games	Wins	Losses	Draws	Elo	+/−
1	1482	532	305	645	53.6	13.3
2	4411	1264	1039	2108	17.7	7.4
3	4988	935	779	3274	10.9	5.6
4	3264	751	635	1878	12.4	6.14

9
Conclusion

In this work, a superhuman chess playing engine was developed and successfully tested against other existing engines. This was made possible by a steady, iterative development process that ensured each implementation contributed to the engine's overall playing strength.

Illumina's features and test results can serve as a reference for the development of new chess engines or any other engines based on the Alpha-Beta pruning algorithm. Besides that, it also demonstrates how SPRT can be successfully employed for testing projects like this.

Future Work

The largest room for exploration lies in improving the static evaluation neural networks for chess engines. These can be reached by testing different network architectures, trying different data generation and filtering criteria and further hyperparameter testing.

Many modern chess engines employ the network buckets to switch between layers depending on different aspects of the current position – usually king positions and game phase. A possible idea is to train a model via unsupervised learning to classify chess positions and use this model to classify positions before search, switching network layers or the evaluation network entirely based on the returned classification.

As for Illumina itself, many other well-established features – such as network buckets themselves – can still be implemented in order to push its strength further.

10
Bibliography

ALBERS P.C.H; VRIES, H. d. Elo-rating as a tool in the sequential estimation of dominance strengths. 2001. Cited in page 14.

CAMPBELL, M.; HOANE, A. J.; HSU, F. hsiung. **Deep Blue**. [S.l.: s.n.], 2002. Cited in page 13.

CCRL. **CCRL Chess Website**. 2025. https://www.computerchess.org.uk/ccrl/. Accessed: June 27th, 2025, 16:19 UTC-3. Cited in page 14.

FRIEDERICH, R. **RofChade**. 2020. https://rofchade.nl/?p=307/. Accessed: June 2nd, 2025, 19:41 UTC-3. Cited in page 33.

GOETSCH G.; CAMPBELL, M. Experiments with the null-move heuristic. 1990. Cited in page 40.

GRANT, A. **OpenBench Repository**. 2025. https://github.com/AndyGrant/OpenBench. Accessed: June 10th, 2025, 20:33 UTC-3. Cited in page 38.

GUSEV, D. A. Using modern chess software for opening preparation. 2021. Cited in page 13.

HEATH, D.; ALLUM, D. The historical development of computer chess and its impact on artificial intelligence. 1997. Cited in page 13.

INTEL CORPORATION. **Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 2A: Instruction Set Reference, A–L**. [S.l.], 2022. Cited in page 23.

MAHARAJ, S.; POLSON, N.; TURK, A. Chess ai: Competing paradigms for machine intelligence. 2022. Cited 2 times in pages 13 and 14.

MARLINFLOW Repository. 2025. https://github.com/jnlt3/marlinflow. Accessed: June 1st, 2025, 16:02 UTC-3. Cited in page 36.

NASU, Y. Efficiently updatable neural-network-based evaluation functions for computer shogi. 2018. Cited in page 14.

POHL, S. **Stefan Pohl Computer Chess**. 2025. https://www.sp-cc.de/. Accessed: June 4th, 2025, 21:05 UTC-3. Cited in page 38.

RISI, S.; PREUSS, M. From chess and atari to starcraft and beyond: How game ai is driving the world of ai. 2020. Cited in page 13.

SILVER, D. et al. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. 2017. Cited 2 times in pages 13 and 14.

SPALL, J. C. A stochastic approximation technique for generating maximum likelihood parameter estimates. **Proceedings of the 1987 American Control Conference**, 1987. Cited in page 39.

STOCKFISH. **Stockfish Books Repository**. 2025. https://github.com/official-stockfish/books. Accessed: June 4th, 2025, 21:03 UTC-3. Cited in page 38.

TCEC. **TCEC Chess Website**. 2023. https://tcec-chess.com/. Accessed: October 26th, 2023, 19:19 UTC-3. Cited in page 14.

WALD, A. Sequential tests of statistical hypotheses. 1945. Cited in page 37.