BABEŞ-BOLYAI UNIVERSITY CLUJ-NAPOCA
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
SPECIALIZATION COMPUTER SCIENCE

DIPLOMA THESIS

Using artificial intelligence to assist chess players

Supervisor
Asist. univ. dr. Florentin Bota

Author
Cadar Eduard

2023

BABEŞ-BOLYAI UNIVERSITY CLUJ-NAPOCA
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
SPECIALIZATION INFORMATICĂ

DIPLOMA THESIS

Utilizarea inteligenței artificiale în asistarea jucătorilor de șah

Supervisor
Asist. univ. dr. Florentin Bota

Author
Cadar Eduard

2023

ABSTRACT

The usual way of searching for a move from a chess position is through tactics or strategy. Tactics are series of moves that would bring an immediate advantage, while strategy refers to a general 'sense' of advantage on the board, without too much calculation (having better developed pieces, better pawn structure etc.) [7].

Chess engines are usually built using a minimax algorithm, along with an evaluation function that takes into consideration the positions of the pieces on the board. This approach is able to find the moves that would bring material gain (capturing pieces) in a given depth, but cannot find moves that would slowly improve the position. Another approach would be training a neural network on professional chess games, which would be able to find good positional moves, but may be weak at finding tactical moves.

An algorithm combining these two approaches should yield better results than each of them on their own.

Contents

List of Figures

Chapter 1

Introduction

Despite being a centuries-old game, chess still remains unsolved, and is one of the most extensively researched areas of artificial intelligence. Due to the complexity of the game and the large number of possible positions, conventional methods of computing and searching for the best move are ineffective and produce unsatisfactory results, necessitating the use of heuristics in search, evaluation, and move selection.

Chess needs so much creativity and profound reasoning that it was originally considered that computers would never be able to perform it. And it stayed that way for a long time, until IBM's Deep Blue [1] defeated reigning World Chess Champion Garry Kasparov in 1997, winning 2 matches, drawing 3, and losing 1. Since then, the best chess engines have improved to the point that no human is able to win a single match against them.

There are several properties of the chess game that make it ideal for computers:

- Deterministic - all potential games result in either a win for white, a win for black, or a draw
- Finite - games cannot continue indefinitely (there are two restrictions to avoid this: a draw occurs after three repetitions of the same position, and 50 moves without a capture or a pawn move is also a draw)
- Complete information - there is no concealed information or ambiguity as there is in a card game, and the game is played sequentially; both players have access to the same information
- Zero-sum - the goals of the competitors are opposite: a win for one player is a loss for his opponent, and a draw results in a game sum of zero. If a position is worth +10 to one player, then the opponent's score is -10

//

In the first chapter I will give an introduction to the general history and directions in chess engines development. In the second chapter I will go over some of the techniques and algorithms used in programming chess engines,

as well as some of the programming methods that modern chess engines utilize. In the third chapter I will describe the methods and algorithms I used in building my chess engine, and in the fourth chapter I will describe the tools and technologies with which I built the game and the engine.

Chapter 2

Techniques and algorithms used

In his 1950 paper, "Programming a Computer for Playing Chess" [12], Claude E. Shannon describes how a chess engine might be implemented. Although not clearly stated, from his work we can deduce the three main parts a machine would need to do to be able to play chess, which are still valid and used in modern engines:

- Generating the legal moves of a position
- Searching for legally reachable positions from a position
- Evaluating the positions

2.1 Move generation

2.1.1 Legality

Regarding legality of the moves, move generation can be pseudo-legal or legal. In pseudo-legal move generation, the moves generated follow the normal rules of each piece's movement, but they are not checked to see if the king is left or moved into check. In legal move generation only legal moves are generated, checking beforehand if the king would be left in an illegal position. Pins, along with en passant moves, are particularly difficult to check. That is why the first approach is generally used: generating the pseudo-legal moves first, checking each move to see if it leaves the board in a valid state (the king of the side that moved is not in check), and removing the invalid moves.

2.1.2 Bitboards

2.1.3 Threat maps

2.2 Search

2.2.1 Minimax algorithm

The minimax algorithm represents the search as a tree, where the root node is the current position, edges to its children are the possible moves and the children are the positions reached by making each respective move. Then, for each child node (position) the process continues: edges for the legal moves, child nodes for reached positions. If enough computing power was available, generating the nodes would stop when a position which can be given a perfect evaluation would be reached - that is, an end position, which would be evaluated with win for white, draw, or win for black. Since from most positions the tree would be far too big to generate, the tree is usually generated to some depth (the evaluations would become more accurate the deeper the tree is generated), and the positions reached are then evaluated using a numerical heuristic function. The evaluations are then propagated back to the root node in the following way: for each node that has all of its children evaluated, if it is white's turn, the node is given the maximum value of its child nodes, and if it is black's turn, the node is given the minimum value of its child nodes instead.[7]

The complexity of the minimax algorithm in chess is dependent on the depth of the search tree and the number of legal moves on each level.

Assuming an average branching factor of around 35 moves per position, the number of positions to be evaluated by the minimax algorithm can grow exponentially with the depth of the search tree. For example, at a depth of 4 ply (i.e., 4 moves ahead), there are roughly $35^4 = 1,500,625$ positions to be evaluated.

In general, the time complexity of the minimax algorithm is $O(b^d)$, where b is the branching factor and d is the depth of the search tree. This can make it computationally infeasible to search deeply in the game tree, since there are a large number of possible moves and the branching factor is high.

To address this issue, techniques like alpha-beta pruning, transposition tables, and iterative deepening have been developed to reduce the number of positions that need to be evaluated by the minimax algorithm. Additionally, more advanced algorithms like Monte Carlo Tree Search and neural networks can be used to guide the search and reduce the number of positions that need to be evaluated, further improving the efficiency of the algorithm.

2.2.2 Alpha-beta pruning

Alpha-beta pruning is an optimization technique used in the minimax algorithm that can reduce the number of nodes that need to be evaluated in the game tree, making the search more efficient. Minimax searches through all leaf nodes to find the minimax value, while alpha-beta prunes leaves that have no influence on the outcome.

The algorithm works by maintaining two values: alpha and beta. Alpha represents the best score that the maximizing player has found so far, while beta represents the best score that the minimizing player has found so far.

As the algorithm searches through the game tree, it compares the scores of each possible move to alpha and beta. If a score is found that is worse than alpha (for the maximizing player) or better than beta (for the minimizing player), then that branch of the game tree can be pruned, since it will not lead to a better outcome [2].

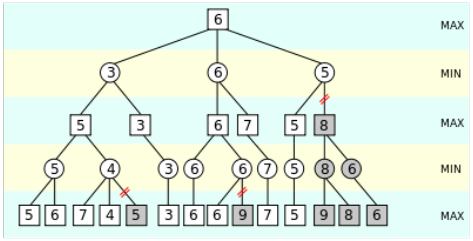

Figure 2.1: Alpha-beta pruning

The time complexity of alpha-beta pruning in chess is also dependent on the depth of the search tree and the branching factor of the game. In the worst case scenario, where alpha-beta pruning has no effect, the time complexity of alpha-beta pruning is still $O(b^d)$, the same as for the minimax algorithm. But alpha-beta pruning, in the best case, has a time complexity of $O(b^{\frac{d}{2}})$, which is significantly less than that of the classic minimax algorithm.

The effectiveness of alpha-beta pruning depends on the ordering of the moves in the search tree. If the moves are ordered in such a way that the best moves are considered first, then alpha-beta pruning can quickly identify the best move and avoid searching deeper than necessary.

2.2.3 Quiescence search

When searching for positions, if the search stops after a fixed depth, some of the positions might be in a situation where on the next move the material evaluation could be totally reverted (a big piece about to be captured, a pawn about to be promoted, etc.). In those cases, the evaluation would not be so reliable, and the situation might be reversed in the next move.

To deal with these situations, quiescence search is an improvement that continues the search in non-quiescent positions until a quiescent position is found. A position is considered quiescent if there are no forcing moves (checks, threats) or moves that have a big impact on material (captures, promotions).

2.2.4 Iterative deepening

When limiting the search by time, performing the minimax to a fixed depth has some disadvantages: if the time has elapsed and the algorithm didn't finish, the results are probably bad, but, on the other hand, if the algorithm finished there might still be some time left, which could be used to improve the result [2].

Adrian De Groot, dutch psychologist and chess master, first mentioned the idea of iterative deepening in "Thought and Choice in Chess" [6]. With this improvement, the search starts at depth one, and then the depth is incremented and the search is done again, until the given time elapses.

Along with the ability of the algorithm to work well with time limits, iterative deepening introduces an additional advantage: the moves can be ordered at each level using the results given by the previous search, and this has shown to bring improvements to the number of branches cut by alpha-beta pruning.

2.3 Evaluation

As mentioned earlier, if unlimited computing power was available, any position could be labeled as winning for white, draw, or winning for black. Because this is not the case, heuristics are used to assign an approximated evaluation to a position.

2.3.1 Centipawns

The main part of an evaluation function is the material advantage, that is, the number and types of pieces on each side. Pieces are usually assigned a base value, as can be seen in fig. 2.2. The king is either assigned no value, because the objective of chess is to checkmate the opponent's king, not capture it, therefore its value is considered irrelevant, or is assigned a very high value (bigger than all the other pieces combined), so the engine prioritizes the safety of the king above all else. In this case, the engine would attempt to avoid any moves that might put its own king in danger, even if it means giving up material or positional advantage.

But there are other aspects of a position that need to be taken into consideration by an evaluation function, for example development of the pieces (a piece that is not developed and doesn't attack/defend any squares is not of so much use). Some

pawn	knight	bishop	rook	queen
1	3	3	5	9

Figure 2.2: Basic piece values

features of the position are not as important as to add a full point to the evaluation, so usually centipawns are used to evaluate a position.

For example, some positional advantage a player has might be worth about three quarters of a pawn. A centipawn is a score unit corresponding to a hundredth of a pawn, so three quarters of a pawn would be 75 centipawns. That means a pawn will be worth 100 centipawns, a knight will be worth 300 centipawns, and so on.

2.3.2 Handcrafted functions

Most engines use a handcrafted function to evaluate positions, measuring features of the board that are considered to give an advantage. Some basic evaluation features are:

- Material - the number and types of pieces
- Pieces positions - where the pieces are placed (in the center, at the edge, in the corner of the board)
- Pawn structure - connected pawns (allied pawns on adjacent files) are considered good, isolated pawns (with no allied pawns on adjacent files) are considered weak, passed pawns (pawns that do not have any opponent pawns on the same file or adjacent files in front of them) are strong, because they cannot be traded by opponent pawns, etc.
- Mobility - the number of legal moves a player has in the position
- Trapped pieces - pieces that don't have any safe squares to move to are considered weak
- King safety - existence of allied pawns in front of the king, distance of opponent pieces to the king, etc.

Most of the times, handcrafted evaluation functions also take into consideration the phase of the game - the development of the pieces is the main concern in the opening, followed by attacking and defending in the middle game, and pawn promotion in the endgame.

2.3.3 Neural network

Another way of building an evaluation function is through training a neural network.

One approach to training the neural network is to use supervised learning, by giving it the position, sometimes along hand-crafted features as input (the ones previously mentioned: material, mobility etc.), and the corresponding evaluations, such as the scores given by human experts or strong chess engines.

Another approach is to use reinforcement learning, where the network is trained by playing against itself and learning from the outcomes of the games.

Chapter 3

State of the art chess engines

3.1 Stockfish

The strongest chess engine is currently Stockfish, an open-source project. It uses a minimax algorithm to search for the most promising positions, along with alpha-beta pruning, iterative deepening, and other improvements to reduce the search-space [9]. For evaluating the positions, Stockfish uses a hand-crafted evaluation function that consists of multiple features: material, position of the pieces on the board, pawn structure, mobility of pieces, king safety etc. [8]

Recent versions of Stockfish have been improved with NNUE (Efficiently Updatable Neural Networks). NNUE is a neural network-based evaluation function. Unlike traditional evaluation functions that are based on handcrafted features and heuristics, NNUE evaluates a chess position by analyzing it directly with a neural network. Because the neural network can learn and improve its evaluation of positions over time through training, the evaluation function becomes more flexible and adaptable.

Stockfish uses a multi-layer neural network architecture that takes as input a representation of the current board state and outputs a score that reflects the expected outcome of the game. The neural network is trained using a dataset of high-quality chess games, where the inputs are positions from the games and the outputs are the eventual outcomes of the games.

NNUE has been a significant development in computer chess, as it has led to significant improvements in the playing strength of the Stockfish engine. In particular, NNUE has improved Stockfish's ability to evaluate endgame positions, which were previously challenging for traditional evaluation functions to handle.

3.2 AlphaZero

AlphaZero, another strong chess engine that was first published in 2017, uses Monte Carlo Tree Search instead of minimax. This algorithm does not need an evaluation function, since it can compute meaningful evaluations based on random playouts that reach terminal game states, where the win/draw/loss outcome can be applied. Still, combining it with a strong evaluation function helps improve it.

DeepMind, the team behind the engine, built an evaluation function through deep reinforcement learning, training it with the matches the engine played itself. This brings the advantage that there is no need for hand-crafted feature selection and no data, since it generates the data by itself, and, with enough training, the model can learn its own useful features.

Another advantage of the Monte Carlo Tree Search algorithm is that the performance is expected to gradually grow with the computation time and the number of iterations. Iterative deepening in Minimax can provide some resemblance to this behavior, but it is typically less ”smooth” because the method takes far longer each time the depth is raised than it did for the previous depth. If it runs out of time, it must abandon the current search at the current depth limit; the last search will be useless, and the results from the previous search will be used instead.

3.3 Leela Chess Zero

Leela Chess Zero (LCZero) is a computer chess engine based on the principles of DeepMind's AlphaZero. Since AlphaZero is not open-source, and there was no way to test the results stated in DeepMind's paper, LCZero was created by a group of developers led by Gary Linscott, who started the project in 2018. The engine was largely driven by the community of chess enthusiasts and programmers, who contributed to the project by providing computing resources.

LCZero developers also brought some key improvements to the AlphaZero approach:

- Efficient implementation: LCZero is designed to run efficiently on standard hardware, such as CPUs and GPUs commonly available to consumers. This makes it more accessible than AlphaZero, which was trained on specialized hardware

- Improved neural network architecture: LCZero's neural network architecture is optimized for the game of chess, with modifications that allow it to better represent the game state and learn from self-play. This has led to better performance in practice. The developers also added Squeeze and Excitation

(SE) layers to the neural network, which improve the representation of CNNs (Convolutional Neural Networks), by enabling them to selectively attend to the most important features of the input data

- Dynamic evaluation: LCZero uses a dynamic evaluation function during self-play, which allows it to better evaluate the value of positions and guide its search. Unlike AlphaZero, which uses a fixed evaluation function throughout the self-play process, LCZero updates its evaluation function after each game played, based on the results and the positions encountered. This allows it to adapt to changes in the playing style and strategy of its opponents

Chapter 4

Methodology

In this chapter I will describe the implementation of the three main steps taken by the engine to find the best move: generating legal moves for a position, searching for legally reachable positions from the current position, and then evaluating each one and determining which one is the best [10].

4.1 Move generation

The engine computes the pseudo-legal moves of the pieces of the player that is moving, then it does each move, checks the state of the board, adds the move to the list of legal moves if the board is in a valid state, and then undoes the move.

To check the state of the board I used a threat map (fig. 4.1), which keeps track of the squares that are under attack or are defended by at least one piece (including empty squares, squares with allied pieces, and squares with opponent pieces). After each move, the threat map is updated in the following way: for each white piece on the board, it sets ”IsAttackedByWhite” to true for all the squares it attacks, and for each black piece on the board, it sets ”IsAttackedByBlack” to true for all the squares it attacks.

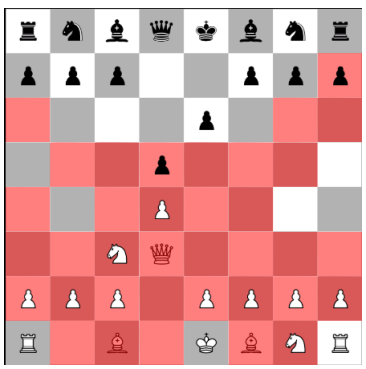

Figure 4.1: White threat map

4.2 Search

The simplest way of searching in the game tree is through a fixed depth Minimax algorithm, which would be mathematically accurate, but would not yield good results and would be very slow. However, in combination with some optimizations and heuristics, it can compute faster and find better moves.

4.2.1 Alpha-beta pruning

The Minimax algorithm was improved with Alpha-beta pruning, which preserves the accuracy of the algorithm and significantly reduces the number of nodes visited if combined with a good move ordering heuristic [13].

A simple pseudocode for Alpha-beta pruning is shown in fig. 4.2:

```
function alphabeta(node, depth, alpha, beta, maximizingPlayer)
    if depth = 0 or node is a terminal node
        return the heuristic value of node

    if maximizingPlayer
        v := -infinity
        for each child of node
            v := max(v, alphabeta(child, depth - 1, alpha, beta, FALSE))
            alpha := max(alpha, v)
            if beta ≤ alpha
                break (* beta cut-off *)
        return v
    else
        v := +infinity
        for each child of node
            v := min(v, alphabeta(child, depth - 1, alpha, beta, TRUE))
            beta := min(beta, v)
            if beta ≤ alpha
                break (* alpha cut-off *)
        return v
```

Figure 4.2: Alpha-beta pruning pseudocode

The heuristic I used in combination with Alpha-beta pruning is MVV-LVA (Most Valuable Victim - Least Valuable Agressor): the moves that capture pieces with high values are first, and if multiple moves capture a piece of the same value, the moves with the least valuable agressor are prioritized [11]. The heuristic is based on the observation that in chess, it is often advantageous to capture pieces of higher value than the capturing piece, while avoiding exchanges where the capturing piece is more valuable than the captured piece. It is also relatively easy to implement, and does not require much computational overhead. Other heuristics, use the evaluations of previous searches or the results from other branches, and require maintain-

ing and updating additional data structures to keep track of the move history and their performance.

4.2.2 Quiescence search

4.2.3 Iterative deepening

4.3 Evaluation

I used two methods of evaluating a chess board position. The first one is a simple evaluation function based on the position of the pieces on the board, and the second one is a neural network model, trained on matches played at grandmaster level.

4.3.1 Board pieces

This evaluation function assigns a value to the board configuration based on the position of the pieces and the state of the game (opening/middlegame/endgame). I considered the state to be the opening if there were less than 13 moves made, middlegame if there are at least a queen or 3 minor pieces (rook/bishop/knight) on the board, and the endgame otherwise.

Each piece has a base value (fig. 4.3), to which another value is added based on the square the piece occupies. Each piece has an 8x8 table, with a value for each square. The tables contain values as to encourage development of the pieces (example - fig. 4.4 shows the table for the white knights). The tables for black pieces are horizontally simetric to the ones for the white pieces. The king has an additional table for the endgame, because in the opening and middlegame it is encouraged to castle and discouraged to move far from the first rank (fig. 4.5a), while in the endgame it is encouraged to go towards the center and play an active part in the game (fig. 4.5b) [5].

```
PAWN   =   100
KNIGHT =   320
BISHOP =   330
ROOK   =   500
QUEEN  =   900
KING   = 20000
```

Figure 4.3: Base values for the pieces

```
-50,-40,-30,-30,-30,-30,-40,-50,
-40,-20,  0,  0,  0,  0,-20,-40,
-30,  0, 10, 15, 15, 10,  0,-30,
-30,  5, 15, 20, 20, 15,  5,-30,
-30,  0, 15, 20, 20, 15,  0,-30,
-30,  5, 10, 15, 15, 10,  5,-30,
-40,-20,  0,  5,  5,  0,-20,-40,
-50,-40,-30,-30,-30,-30,-40,-50,
```

Figure 4.4: Knight value table

Figure 4.5: King value tables

```
-30,-40,-40,-50,-50,-40,-40,-30,
-30,-40,-40,-50,-50,-40,-40,-30,
-30,-40,-40,-50,-50,-40,-40,-30,
-30,-40,-40,-50,-50,-40,-40,-30,
-20,-30,-30,-40,-40,-30,-30,-20,
-10,-20,-20,-20,-20,-20,-20,-10,
 20, 20,  0,  0,  0,  0, 20, 20,
 20, 30, 10,  0,  0, 10, 30, 20
```

(a) King middlegame value table

```
-50,-40,-30,-20,-20,-30,-40,-50,
-30,-20,-10,  0,  0,-10,-20,-30,
-30,-10, 20, 30, 30, 20,-10,-30,
-30,-10, 30, 40, 40, 30,-10,-30,
-30,-10, 30, 40, 40, 30,-10,-30,
-30,-10, 20, 30, 30, 20,-10,-30,
-30,-30,  0,  0,  0,  0,-30,-30,
-50,-30,-30,-30,-30,-30,-30,-50
```

(b) King endgame value table

4.3.2 Neural network

For the second evaluation function, I trained a neural network on master chess matches, which I will expand in a separate chapter.

Chapter 5

Training and technologies

5.1 Training

5.1.1 Dataset

The dataset consists of chess matches played at grandmaster level downloaded in PGN format from the website chessabc.com [3].

A PGN (portable game notation) file contains multiple chess matches. Each match has several headers (name of player with white pieces, name of player with black pieces, event it was played at, date it was played on, result, elo of players etc.), an empty line, and then the ”movetext section”. The matches in a PGN file are separated by two empty lines. Following is a sample PGN match:

```
[Event "Carlsen - Anand World Championship Match"]
[Site "Sochi RUS"]
[Date "2014.11.09"]
[Round "2"]
[Result "1-0"]
[White "Magnus Carlsen"]
[Black "Viswanathan Anand"]

1.e4 e5 2.Nf3 Nc6 3.Bb5 Nf6 4.d3 Bc5 5.O-O d6 6.Re1 O-O 7.Bxc6
bxc6 8.h3 Re8 9.Nbd2 Nd7 10.Nc4 Bb6 11.a4 a5 12.Nxb6 cxb6
13.d4 Qc7 14.Ra3 Nf8 15.dxe5 dxe5 16.Nh4 Rd8 17.Qh5 f6 18.Nf5
Be6 19.Rg3 Ng6 20.h4 Bxf5 21.exf5 Nf4 22.Bxf4 exf4 23.Rc3 c5
24.Re6 Rab8 25.Rc4 Qd7 26.Kh2 Rf8 27.Rce4 Rb7 28.Qe2 b5 29.b3
bxa4 30.bxa4 Rb4 31.Re7 Qd6 32.Qf3 Rxe4 33.Qxe4 f3+ 34.g3 h5
35.Qb7 1-0
```

The movetext section contains the chess moves, move numbers, optional annotations and a concluding game termination marker (the result of the match). The result can be ”1-0” (white won), ”0-1” (black won) or ”1/2-1/2” (draw). The moves

are written in SAN (standard algebraic notation). SAN identifies each square on the board as a combination of file (a-h) and rank (1-8) (see fig. 5.1).

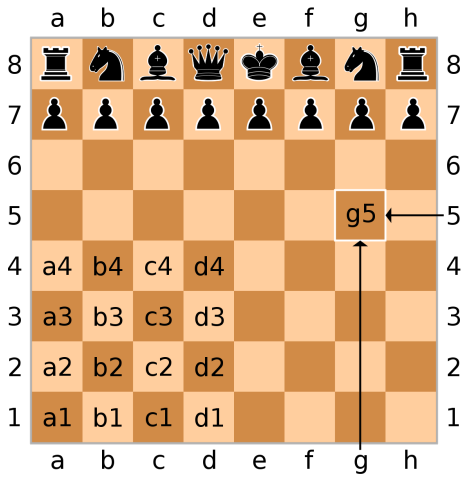

Figure 5.1: Board squares

Each piece can be identified by a letter (K - king, Q - queen, R - rook, B - bishop, N - knight, P - pawn). Simple SAN moves contain only the destination square, and the letter of piece that moves, if the piece is not a pawn (for example, pawn to c6 is written simply as c6, while knight to c6 is written as Nc6). There are additional rules for moves [4]:

- Moves that capture a piece contain the 'x' character right before the destination square (ex: Bxb4)
- Castling kingside is written as 'O-O', while queenside castling is written as 'O-O-O'
- Moves of pawns to the last rank (promotion moves) contain the equal sign ('=') right after the destination square, followed by the letter of the piece that the pawn promotes to (ex: e8=Q)
- If there are multiple pieces of the same type that can move to the destination square, the file of the originating square is added right after the piece (ex: Nbc6); if this does not solve the ambiguation, the rank of the originating square is added instead (ex: N8c6), and if neither of these works, both the file and the rank are added (ex: Nb8c6)
- If the move checks the opponent's king, a plus sign ('+') is added at the end of the move (ex: Qe7+); if the move checkmates the opponent's king, the octothorpe sign ('#') is added instead (ex: Qe7#)

I computed the positions that occured in the matches in the dataset, and the value I assigned to each position is the mean of the results of the matches in which

the position appeared. For example: if position A appeared in 10 matches, and 4 of the matches ended in a win for white, 2 in a win for black, and the other 4 were draws, the value assigned to the position is:

$$((4 * 1) + (4 * 0) + (2 * -1))/10 = 0.2$$

(a win for white is assigned 1, a win for black is assigned -1, and a draw is 0).

5.1.2 Layers

5.2 Technologies

5.2.1 Chess game

5.2.2 Chess engine

Chapter 6

Results and evaluation

Chapter 7

Conclusions

Bibliography

[1] Murray Campbell, A Joseph Hoane Jr, and Feng-hsiung Hsu. Deep blue. *Artificial intelligence*, 134(1-2):57–83, 2002.

[2] Jeroen WT Carolus. Alpha-beta with sibling prediction pruning in chess. *Amsterdam: University of Amsterdam*, 2006.

[3] Chess ABC, 2023. URL: `https://www.chessabc.com/en`.

[4] Steven J Edwards et al. Standard portable game notation specification and implementation guide, 1994. URL: `https://ia902908.us.archive.org/26/items/pgn-standard-1994-03-12/PGN_standard_1994-03-12.txt`.

[5] Simplified evaluation function, 2018. URL: `https://www.chessprogramming.org/index.php?title=Simplified_Evaluation_Function`.

[6] A.D. Groot. *Thought and choice in chess*. 07 2014.

[7] Dominik Klein. Neural networks for chess. *arXiv preprint arXiv:2209.01506*, 2022.

[8] Matthew Lai. Giraffe: Using deep reinforcement learning to play chess. *arXiv preprint arXiv:1509.01549*, 2015.

[9] Shiva Maharaj, Nick Polson, and Alex Turk. Chess ai: competing paradigms for machine intelligence. *Entropy*, 24(4):550, 2022.

[10] T Anthony Marsland. A review of game-tree pruning. *ICGA journal*, 9(1):3–19, 1986.

[11] Most Valuable Victim - Least Valuable Agressor, 2019. URL: `https://www.chessprogramming.org/index.php?title=MVV-LVA&oldid=15328`.

[12] Claude E Shannon. Xxii. programming a computer for playing chess. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 41(314):256–275, 1950.

[13] Eric Thé. *An Analysis of Move Ordering on the Efficiency of Alpha-beta Search*. PhD thesis, McGill University Libraries, 1992.