Stochastic Exploration in Minimax Search by Using a Policy Predictor Network

Bachelor thesis by Jannik Holmer
Date of submission: May 1, 2022

1. Review: Prof. Dr. Kristian Kersting
2. Review: M.Sc. Johannes Czech
Darmstadt

TECHNISCHE UNIVERSITÄT DARMSTADT

Computer Science Department

Artificial Intelligence and Machine Learning Lab

Erklärung zur Abschlussarbeit gemäß §22 Abs. 7 und §23 Abs. 7 APB der TU Darmstadt

Hiermit versichere ich, Jannik Holmer, die vorliegende Bachelorarbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mir ist bekannt, dass im Fall eines Plagiats (§38 Abs. 2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden.

Bei der abgegebenen Thesis stimmen die schriftliche und die zur Archivierung eingereichte elektronische Fassung gemäß §23 Abs. 7 APB überein.

Bei einer Thesis des Fachbereichs Architektur entspricht die eingereichte elektronische Fassung dem vorgestellten Modell und den vorgelegten Plänen.

Darmstadt, 1. Mai 2022

J. Holmer

Abstract

In this thesis, we evaluate the combination of traditional depth first Minimax Search with Convolutional Neural Networks as Policy and Value predictor in chess and crazyhouse chess. Inspired by the success of Convolutional Neural Networks in combination with Monte Carlo Tree Search, we create a Minimax engine using the network architectures and parameters of the open-source engine CrazyAra.
We discuss and choose the appropriate implementations and improvements of Minimax search for our case. In addition, we propose the use of Policy Quantiles, a method to prune the search tree based on the Neural Network's Policy predictions. In the evaluation, we prove the efficacy of Policy Quantiles in the variant of crazyhouse chess, which has a higher branching factor than its classical counterpart. Lastly, we compare the created Minimax engines to their MCTS counterparts on their search efficiency. In a series of matches with both engines restrained to about the same numbers of Neural Network evaluations, Monte Carlo Tree Search outperformed Minimax Search quite comfortably.
Keywords: Minimax Search, Monte Carlo Tree Search, Deep Learning, Chess, Crazyhouse Chess

Contents

1 Introduction

1.1 Motivation

The world of computer engines has made leaps of progress in recent years with works like AlphaGo, AlphaZero and MuZero [14, 13, 12]. These algorithms which make use of convolutional neural networks in combination with Monte Carlo tree search have elevated computer engine performance in the games of Go, Starcraft2 and many Atari games to superhuman level, as well as breached the skill ceiling of state-of-the-art engines in Chess, a game extensively studied throughout human history and tightly interwoven in the history of computer game playing.
Before this breakthrough with Monte Carlo tree search, most high performing engines in chess used the Minimax approach to tree search, as made popular by Deep Blue [2]. It works well in highly tactical environments like chess, where there are many traps and precision is very important, but tends to drop off in performance when used in bigger environments with higher branching factors.

1.2 Problem Formulation

Monte Carlo Tree Search (MCTS), when used in combination with neural networks, seems to significantly outperform traditional Minimax Search in most environments with high branching factors. This work tries to explore the possibility of using Minimax Search in combination with neural networks as a policy predictor in domains of different sizes. In addition, we propose the use of Policy Quantiles, to leverage the neural network policy for pruning in environments with high branching factor. We try to answer these research questions.
First, can Policy Quantiles be used to improve Minimax performance in environments with high branching factors?

Second, how do Minimax Search and MCTS compare to each other in highly tactical environments with high branching factor?
To approach these questions a prototype engine for classical chess [19] and Crazyhouse chess[20], a variant with significantly higher branching factor, will be created, optimized and matched up against the multi-variant MCTS engine CrazyAra.

1.3 Outline

This work is structured in the following way: The next chapter will cover the basics and some of the more prominent parts of Minimax search, many of which will be used in the creation of the prototype in chapter 4, and will give a quick glance over MCTS and Convolutional Neural Networks.
After that, chapter 3 covers some related works on combining Neural Networks with Minimax search and introduces the CrazyAra project. Chapter 4 gives a detailed description of the decisions that went into the construction of the prototypes and an explanation of the functionality of the final version that was then evaluated in chapter 5. The Evaluation consists of parameter optimizations and a match against the CrazyAra MCTS engines.
Chapter 6 summarizes the results of the optimizations and evaluations and gives some outlooks on possible future research into the application of Policy Quantiles.

2 Background

This chapter will give an overview over the field of game tree search, as well as the algorithms and concepts used in creating the engine used in the evaluation.

2.1 Zero-sum Games

Zero-sum games are a category of games where the gain of one player is equivalent to the loss for the other players. If you were to add up all the gains and losses of all players, they will always add up to 0. Popular examples of zero-sum games are poker or the focus of this work, chess.

2.2 Minimax Search

The Minimax algorithm is a tree search method, invented in the 1920s, to calculate the score of a two player zero-sum game, like chess. Each board position in the game is represented by a node in the tree, the branches at each node are the legal moves playable in the position. Depending on whose turn it is in the position, the decision at the node is either to maximize or to minimize the score. This simulates playing optimally against a perfect opponent.
A move for a single player is called a ply while the combination of two plies, so a move for each player, is called a full move. In this way, the tree is traversed in a depth-first manner until terminal positions are reached.
As terminal positions do not have any legal moves, these positions are the leaf nodes of the game tree. These leaf nodes are then evaluated to be either the maximum score, the minimum score or a drawn position.
Unfortunately, evaluating every possible move to a terminal position is not practically

```
int Maxi(int depth)
        if (depth == 0) or is terminal:
                # return absolute score for terminal positions
                # return heuristic evaluation for non-terminal
                return evaluate_position()
        int score = -∞
        # search through all moves for the highest score
        for (all_moves):
                score = max(score, Mini(depth-1))
        return max

int Mini(int depth)
        if (depth == 0) or is terminal:
                return evaluate_position()
        int score = ∞
        # search through all moves for the lowest score
        for (all_moves):
                score = min(score, Maxi(depth-1))
        return min
```

Figure 2.1: Pseudocode for a Minimax implementation. Depending on the active player, either Max or Min is called first. Both functions call each other until the required search depth is reached.

feasible, as even the game tree of the comparatively small game of chess was estimated to have around 10^{120} leaf nodes, the Shannon Number [22].
In 1948 Norbert Wiener proposed the use of heuristics to evaluate positions only a few moves deep into the tree, which made the algorithm usable in a reasonable time frame and feasible for use in computer game playing. Figure 2.1 shows pseudocode for Minimax with the use of heuristics and a small exemplary search tree is shown in Figure 2.2.

2.2.1 Position Evaluation Heuristics

To evaluate the positions represented by the leaf nodes, different heuristics have been developed. A simple one, for instance, is to count the Material for each player, giving

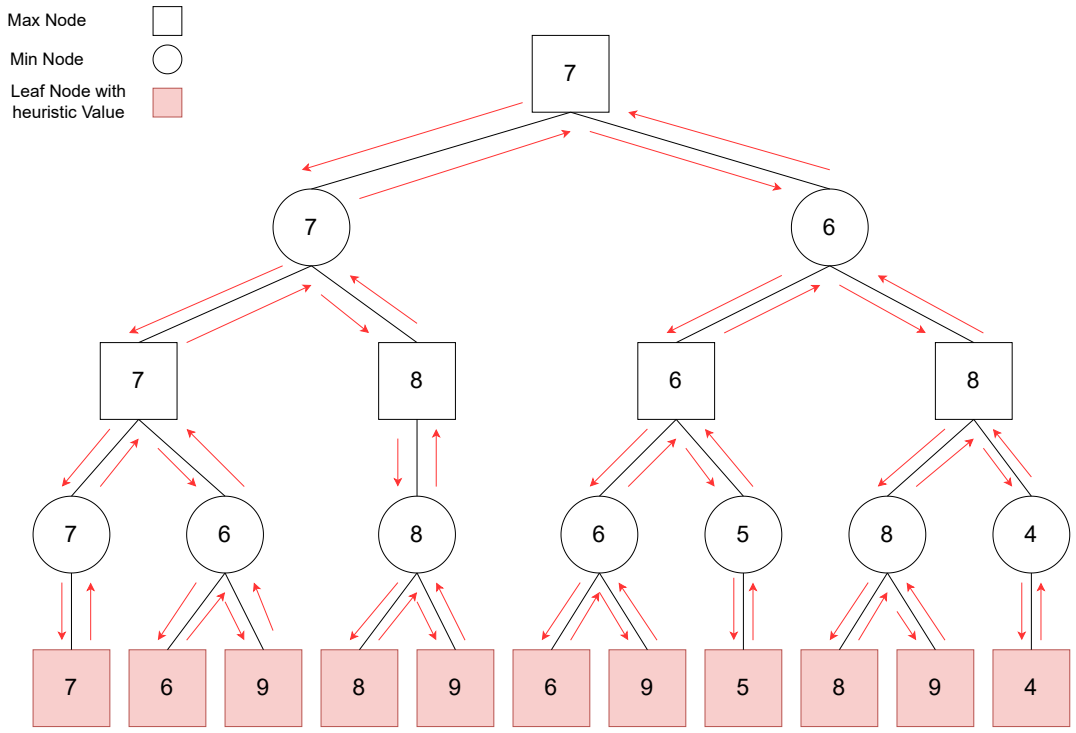

Figure 2.2: A classic Minimax search tree. The tree is traversed depth first (red arrows) and leaf nodes are heuristically evaluated. All non leaf nodes generate their value from their children by either taking the maximum or the minimum, depending on the node.

each piece-type a certain value and adding up the values of all pieces on the board. More sophisticated heuristics like Piece-Square-Tables also take the exact square of each piece into the evaluation or add and subtract points for observable advantages like still having castling rights or disadvantages like doubled pawns. For the case of crazyhouse chess, the potential of pocketed pieces would also have to be taken into account somehow.

2.3 Alpha-Beta-Pruning

In the 1950s, different experts around the world independently developed the most natural and arguably most significant improvement over classical Minimax, Alpha Beta pruning. It takes advantage of the repeated interchanging of maximizing and minimizing the score. Two variables, alpha and beta, are introduced and represent the minimal score the maximizing player is assured of so far and the maximum score the minimizing player is assured of so far.
If a branch has already been searched where the maximizing player can reach at least a certain score Alpha, a single refuting move in a minimizing node searched later on is enough to decide that the node's subtree will perform worse than the already searched branch and doesn't need to be searched any further. This is called an Alpha-Cutoff.
The same is true for the minimizing player, if an already searched branch guarantees the score won't exceed a certain value Beta, every maximizing node searched further down the tree can cause a Beta-cutoff if any of the options would result in a better score for the maximizing player. Figure 2.3 shows the pseudocode of an Alpha-Beta Search implementation, and Figure 2.4 shows the concept used on the Minimax search tree from earlier in Figure 2.2.
Alpha-Beta-Pruning has one caveat though, the amount of nodes that can be cut off depends on the ordering of the moves searched. When the best moves are searched first, more cutoffs occur and the searched game tree can be reduced to close to the square root of its original node count. On the other hand, if the moves are searched from worst to best every time, not a single cutoff will occur. Due to this fact, it is no surprise that many of the further Minimax improvements developed later have focused on improving move ordering in Alpha-Beta search via the use of heuristics.

```
int Maxi(int depth, α, β)
        if (depth == 0) or is terminal:
                # return absolute score for terminal positions
                # return heuristic evaluation for non-terminal
                return evaluate_position()
        int score = -∞
        # search through all moves for the highest score
        for (all_moves):
                score = max(score, Mini(depth-1, α, β))
                if (score >= β):
                        break
                α = max(α, score)
        return score

int Mini(int depth, α, β)
        if (depth == 0) or is terminal:
                return evaluate_position()
        int score = ∞
        # search through all moves for the lowest score
        for (all_moves):
                score = min(score, Maxi(depth-1, α, β))
                if (score<=α):
                        break
                β = min(β, score)
        return min
```

Figure 2.3: Pseudo-Code for Alpha-Beta Pruning. The variable Alpha is initialized with the minimum score and Beta with the maximum score and they are updated throughout the search. If a score exceeds Beta at a Max Node, or lies under Alpha at a Min Node, the search can be stopped at these nodes without changing the outcome of the search.

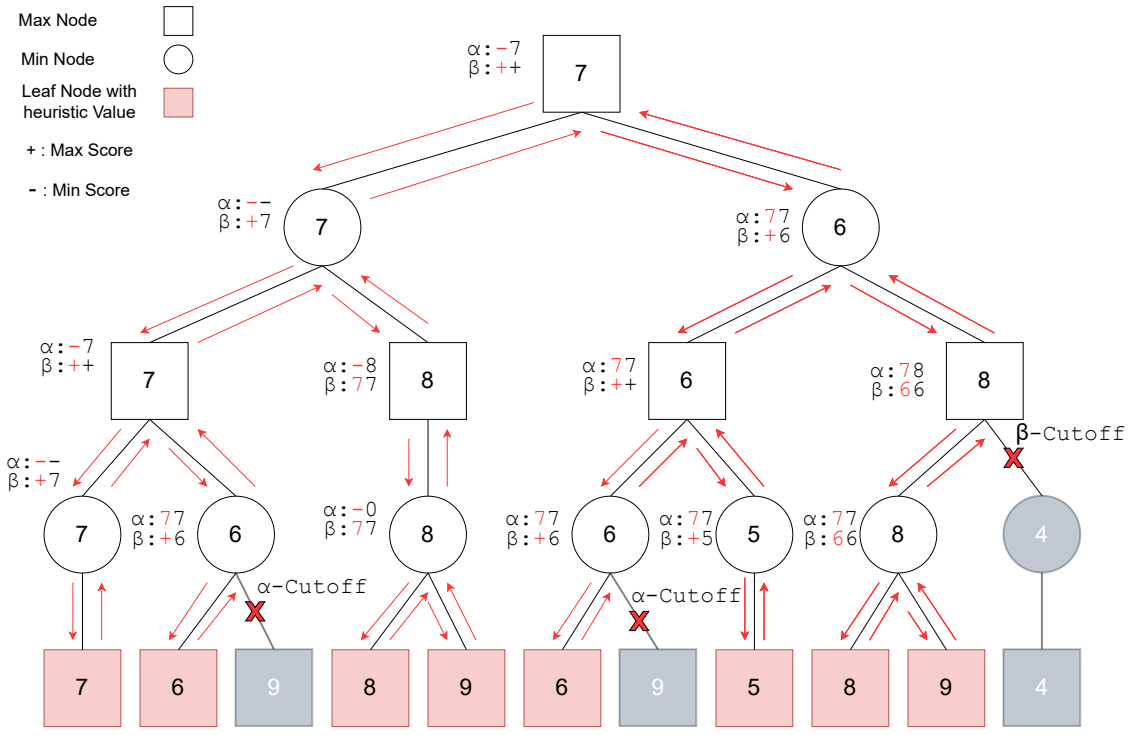

Figure 2.4: The Minimax search tree from Figure 2.2 enhanced with Alpha-Beta Pruning. The red values of Alpha and Beta are upon entry into the node and the black ones after the node has finished its search. The grey nodes have been cut without affecting the end result of the search.

2.3.1 Fail-Hard vs Fail-Soft

One important decision can be made while implementing Alpha-Beta-Pruning, if the returned score is updated before or after checking for cutoffs. When the value is updated first, the implementation is Fail-Soft and can return scores that lie outside the Alpha-Beta range, while Fail-Hard implementations return only scores within the given bounds of Alpha and Beta.

2.3.2 Null-Window searches

Null-Window searches are a type of Alpha Beta search, where Alpha and Beta are initialized with a tiny threshold between them for real-valued scores or as neighboring integers for integer scores. This initialization will result in a lot more cutoffs, but also in the search always failing either high or low. Failing high implying a better move than the first guess was found and the returned score is a lower bound for the true score. Failing low implies no better move was found and the returned score is a new upper bound of the true score. Repeated Null-Window searches can be used to bound the true score and converge towards it by adjusting the initial values of the searches.

2.4 Negamax

Negamax is an improvement of Minimax on the implementation side. Making use of the zero-sum nature of the games searched with Minimax, the score can be centered around zero in such a way that the maximizing player's score equals the negation of the minimizing player's score. This score centering allows it to reduce the Minimax implementation into a single recursive function which always maximizes over the negation of all its children's results. For simplicity, all the pseudocode shown from here on out will be based on a Negamax implementation of Minimax, as was used in the Engine evaluated in chapter 4.

2.5 Transposition Tables

First used in the 1960s, Transposition Tables are an easy way of improving the relatively simple, inefficient game tree search shown so far. As these algorithms don't keep track of

the positions already searched they are bound to search and re-search many of the same positions, which can be reached through different move combinations, again and again. Transposition Tables are hash-tables in which each searched position is saved together with the score and the depth to which the position was searched. If the search encounters the same position again later on in the search-tree it can look at the saved results and if the depth of the saved search was greater or equal to the current search depth, it can just return the saved results.
For use in Alpha-Beta Pruning one can not only save exact results but also the scores when cutoffs occur, e.g. upper or lower bounds on the true score, these can also be used when the position is revisited.
In addition to the score, the move leading to that score can also be saved. This can then be used in subsequent searches, even if the saved depth is lower than the current, as the best move from a previous search is more likely to be close to the true score and if searched first might lead to more cutoffs later on.

2.5.1 Transposition Keys

Different approaches to generate hash keys from positional information have been devised, but the game of chess has an upper bound of 10^{46} different possible positions, this is way too many to save them all in a hash table.
This means the hashing approaches have to break the positions down into a manageable number of keys, making collisions unavoidable.

Zobrist Hashing

Zobrist Hashing is a method invented to encode board positions of arbitrary size into a number of certain length. A set of pseudo-random numbers is generated for each square on the board with one number for each piece type, a few more are generated for information like the active player, castling rights or en passant captures. The corresponding numbers of each square and all currently applying info-numbers are combined via XOR to generate the key. Depending on the size of the pseudo-random numbers and the amount of positions evaluated during a search, collisions are more or less likely.

2.6 Improvements for Move Ordering

This subsection gives a quick overview over the different kind of heuristics and enhancements developed over the years to improve the move ordering in order to increase the cutoffs generated by Alpha-Beta search.
As the Engine developed in this work relies on a Convolutional Neural Network as a Policy Predictor, most of the methods that are going to be mentioned here have been replaced by the network except for the Best/Refutation-Move Heuristic.

2.6.1 Iterative Deepening (Best/Refutation-Moves)

The case of Iterative Deepening is a curious one. It was first proposed as a solution for Time Management in computer chess. As it is hard to predict how deep you can search a position in a given time frame, the position is evaluated to depth one and if there is still time left, the depth is increased and the position searched again. This is repeated until there is no more time left. This might sound inefficient at first, but surprisingly, if the information from previous depths is used correctly, a series of iterative deepening searches up to a certain depth can be faster than directly searching to said depth.
Through the use of Transposition Tables, the best moves as well as the strongest refuting moves for the other player can be saved for each position searched and then be tried first in the following searches, generating a lot more cutoffs than the direct Alpha-Beta Search.

2.6.2 Improved Checks and Captures

In most positions in chess, if there is a clear winning move, it is often a capture or a checking move. So it seems only natural to search these moves earlier than others when looking for the best. There are different heuristics to evaluate which of the captures are the most promising, for instance captures that win material, in the sense that the piece captured is worth more than the capturing piece, are searched before equal captures or captures losing material.

2.6.3 Killer Heuristic

The Killer Heuristic assumes that many Positions are similar to each other, meaning that positions that differ by a single or a couple of moves still share many of the same tactics. If there is a move found to refute one of these tactics, it is likely to also refute the tactics in similar positions. A Transposition table sorted by search depth is kept with the best ”killer” moves for each depth. The replacement strategy needs to make sure that the moves for each depth all differ from each other for the best results. Killer moves are usually searched after the best/refuting-moves, but before or in between the different enhanced captures.

2.7 Quiescence Search

Searching a game tree using Minimax has a certain flaw when it comes to the use of heuristics to estimate the value of positions in leaf nodes. Most heuristics either use a static evaluation of the pieces on the board and their position or keep track of the material captured in some way to adjust the scores throughout the game. Both of these approaches suffer from the *Horizon-Effect*, when the search ends on a position where a capture has just happened, both believe the capturing player to have gained something. If now the search goes one ply deeper and the capturing piece, let’s say a queen, gets recaptured by the other player, the result would be drastically worse. So when is the search deep enough? Quiescence Search tries to avoid this problem by only stopping the search in *quiet* positions, meaning those where no significant captures or checks can happen. This is done in the leaf nodes of the search tree. The Quiescence Search starts from the leaf and plays out the available captures and checks until only quiet positions have been reached. This can lead to a drastic increase in tree size, so measures to limit the Quiescence Search have to be taken.
This addition to Minimax stabilizes the search results drastically, especially when switching between even and uneven search depths.

2.8 NegaC*

NegaC* search (Figure 2.5) takes advantage of Null-Window Alpha-Beta searches to bound the true score of a position. The algorithm keeps two bounds on the score. The initial value of the searches is the midpoint between the current score bounds. The bounds

```
int negaCStar(int min, int max, int depth):
        int score = min
        while (min!=max):
                alpha = (min + max) / 2
                score = failSoftAlphaBeta(alpha, alpha + 1, depth)
                if (score > alpha):
                        min = score
                else:
                        max = score
        return score
```

Figure 2.5: Pseudocode for NegaC*. This tree search algorithm for integer search spaces keeps bounds on the true score of the position and updates them through repeated Null-Window Alpha-Beta searches until they collapse to the true score. The initial guess for the Null-Window searches is the midpoint between the bounds.

get updated after each search call. Taking the midpoint as start for the search at least bisects the remaining range for the true score with each Alpha-Beta search, allowing for faster convergence.

2.9 MTD(f)

The Memory-Enhanced Test Driver with value f (**MTD(f)**) is a game tree search algorithm framework based on Alpha-Beta search proposed in 1994 by Aske Plaat et al.[10]. The SSS* Algorithm which was temporarily researched as a potential successor of Alpha-Beta turns out to be a special case of the MTD(f) framework (Figure 2.6). The value f is a first guess at the true score of the position. The algorithm does repeated Null-Window Fail-Soft Alpha-Beta searches enhanced with a Transposition Table, starting with guess f. As the searches fail either low or high, the results are kept as new upper and lower bounds respectively. The next search is started with the result of the previous search as initial guess. This is repeated until the two bounds have converged towards a single value. As the convergence can take a while depending on the move ordering during the search and the specifics of the position, MTD(f) can trigger a lot of re-searches where the same nodes are searched over and over again. This fact is countered with the use of Transposition

```
int MTDF(root, f, depth):
        g = f
        upperbound = +inf
        lowerbound = -inf
        while (lowerbound < upperbound):
                if (g == lowerbound):
                        beta = g + 1
                else:
                        beta = g
                g = AlphaBetaWithMemory(root, beta-1, beta, depth)
                if (g < beta):
                        upperbound = g
                else:
                        lowerbound = g
        return g
```

Figure 2.6: Pseudocode for the Memory-Enhanced Test Driver with value f. The better the first guess f is, the better the algorithm performs. It is usually used within an iterative deepening framework, where f is the result from the previous depth.

Tables. The Transposition Table saves the best score for each position and the information, if the last search was a fail-high, fail-low or an exact score. This caching of information speeds up consequent searches immensely, making MTD(f) more effective than classical Full-Window Alpha Beta Search.

2.9.1 MTD-bi

The Memory-Enhanced Test Driver Bisection algorithm is a version of MTD(f) closely related to the NegaC* algorithm, but adapted for real valued scores instead of integers. The initial score f is replaced by the midpoint between the upper and the lower bound, in this way the possible range for the true score of the position is at least bisected e.g. halved with every call to the Null Window Alpha-Beta search. [9] Mańdziuk et al. propose the use of a small acceptance threshold for the bound convergence loop and the alpha-beta window, as the space of real numbers has no Null-Window between ”neighboring” numbers. This makes sense in a continuous score space but due to the finite amount of possible moves in each position there is still a discrete score space and such an acceptance

threshold is not needed as both bounds will converge to the true score in a finite number of calls and the threshold might only lead to us choosing a close to best move instead of the best.

2.10 Parallelization of Alpha-Beta Search

The task of running Alpha-Beta search in parallel turns out to be a pretty hard one because of the inherently serial nature of using the information found in the tree searched so far to discard more branches in the tree later on. To evaluate how well different approaches parallelize the search different metrics are looked at, the scalability, how well the search speed increases with growing number of processors, the speedup, how much faster a certain search runs with growing processor count and the scaling, referring to the increase in nodes searched per second. There are two underlying ways to approach this task, parallelizing as much as possible while running the risk of doing unnecessary work, so maximizing scaling while not necessarily maximizing the speedup and scalability, or spending more management overhead to minimize unnecessary work and having a lower scaling.

2.10.1 Lazy SMP

Lazy Symmetric-Multi-Processing is a relatively simple concept for running Alpha-Beta search in parallel. Multiple processes are started on the same search, but with a shared Transposition table. Through the differences in CPU timings, the searches diverge over time and the different processes benefit more and more of each other's searches and the overall search speeds up. This approach was later refined by measures to diverge the trees searched. Starting the different processes on different search depths, changing the move ordering at the root or tweaking the different move ordering improvements for each process turned out to improve the performance of Lazy SMP.
Early approaches were known to have a bad scalability beyond two processors, but improved versions in modern times were shown to scale well beyond eight processors[1].

2.10.2 Young Siblings Wait Concept (YSWC)

When an Alpha-Beta Search is started on a search tree, the leftmost branches will be expanded first until a leaf is reached. When it and all its siblings have been searched, the parent node will return its result to its parent.This result is then the first bound that can generate a cutoff later in the tree. From this point on in the search, every newly searched node will have a set of bounds.
When searching the children of a node with alpha and beta bounds two things can happen when the first child is searched, a beta cutoff happens and the further search of the node stops or the child might return a value within the bounds and tighten the search window for the search of its siblings.
Through the advances in move ordering heuristics in combination with iterative deepening, if a node is to generate a cutoff it will happen on the first childnode searched with a probability of >90% according to Robert Hyatt and Tord Romstad, two creators of prominent Minimax engines [5]. Looking at these stats, it is not surprising that the Young Sibling's Wait Concept is widely employed in parallel Alpha-Beta search, to reliably avoid unnecessary search overhead.
It dictates that at every node the leftmost child has to be fully explored before the rest of the children might be searched in parallel.

2.10.3 Dynamic Tree Splitting (DTS)

Early parallel tree search algorithms like Principal Variation Splitting (PVS) employed the YSWC to split the search tree into many threads, searching all children in parallel when the leftmost sibling had finished, first only on the leftmost branch of the tree up to the root and in later versions also recursively inside the subtrees. This simple approach had some obvious drawbacks, as the workload of sibling subtrees isn't always the same, if one move complicates the position and another simplifies it, their respective subtrees will vary enormously in size and shape. This meant that many of the threads lay idle while waiting for the others to finish.
Dynamic Tree Splitting as well as Enhanced Principal Variation Splitting (EPVS) address this problem by splitting the tree from the processors. When a processor is done with its subtree and there are no more nodes to search, the processor writes to a shared memory position, broadcasting to the other processors that it is idle. The still busy processors collect the state of their search tree and copy it into shared memory, from there the idle processor searches the tree for a good splitting point to join the search. This approach tries to maximize scaling as well as reduce search overhead as much as possible. As these

approaches were developed in the early days of multi-threading and parallel processing, they had some constraints regarding the capabilities of programming languages as well some unwanted behavior. In some cases nearing the end of a search, DTS would get stuck in a situation where 15 of the 16 processes would constantly ask the last working processor for its state, while not finding a suitable splitting point, as the search was nearly over [17].

The approaches mentioned here focus on parallelization with multiprocessing and were created in the 1990s, there are some more modern approaches making use of mass-multi-threading using GPUs, but those are not relevant for the scope of this work as the neural network based board evaluation will be running on the GPU.

2.11 Monte-Carlo Tree Search (MCTS)

MCTS is a best-first approach to tree search. The algorithm generates a game tree in memory by applying four phases repeatedly until the time runs out.
1. Selection: The saved game tree is traversed from the root to a leaf node, where a new move is chosen.
2. Expansion: The node resulting from the move selected is added to the game tree.
3. Simulation: The algorithm plays against itself from the newly added position to a terminal position.
4. Backpropagation: The result of the self play is propagated back up the tree along the path chosen during Selection back to the root. An average over all results is kept in each node.
When the time has run out, the most promising move is chosen. This approach works well strategically, but sometimes favors losing moves if the one or two refuting moves just haven't been found and explored sufficiently yet. This was later improved upon by Upper Confidence bounds applied to Trees (UCT) which improved node selection in the first phase, invented by Kocsis et al. in 2006 [6].
In recent years, computer game playing agents using MCTS in combination with Convolutional Neural Networks as Policy and Value predictors have been successful in reaching superhuman performance in Go, Starcraft II and many of the Atari games. MCTS has thus shown great potential in applications with large state spaces and high branching factors. Minimax on the other hand is known for its worsening performance with growing state

space, as the depths that can be reached in reasonable time shrink with growing branching factors.

2.12 Convolutional Neural Networks (CNN)

Convolutional Neural Networks [7] are a type of neural network where a series of pixel filters (convolution matrices) is applied sequentially to a single or multiple input planes. The weights of the filters are trained through supervised or reinforcement learning. These types of networks are usually used in computer vision and pattern recognition and have found success in computer game playing in recent years.

2.12.1 CNNs in Computer Chess

When applied to a set of input planes generated from the information in a chess position, two headed CNNs can be used to generate a Policy and a Value for the position. The Policy being a normalized probability distribution over all the legal moves in the position favoring the most promising moves, and the Value being a score from -1 to 1, estimating the probability of winning the game for the players.

3 Related Work

This chapter will talk about some recent works on combining neural networks with Minimax search and will give a quick introduction to CrazyAra, the open-source project this work is based upon.

3.1 Minimax vs MCTS

After the breakthrough of AlphaGo[14] with MCTS at the end of 2015 many researchers focused on MCTS, but in recent years there have been some works successfully combining Neural Networks with Minimax search for the games of Hex, Breakthrough, Othello and Chess. Most notably for Chess being Stockfish NNUE, using the Efficiently Updatable Neural Networks, which have, since their experimental development for the game Shogi in 2018, become the default evaluation for Stockfish in September 2020. The Stockfish Project[18] is with *Stockfish dev15_20220401* the world's strongest chess engine at the time of this writing [23].
The work on Hex, Othello and Breakthrough [3] has managed to create Minimax engines, using neural networks trained through reinforcement learning, that significantly outperformed established MCTS engines. In their experiments, Minimax search performed better than MCTS when both used the same neural network.
These works show that there is still great potential in Minimax Search. In this work, we will try to confirm these results by using traditional depth-first Minimax search methods in combination with the Convolutional Neural Networks of the CrazyAra Project and our proposed Policy Quantiles in the environments of chess and crazyhouse chess.

3.2 CrazyAra

CrazyAra is an open source Monte-Carlo Tree Search engine with a Convolutional Neural Network as Policy and Value predictor. It was first developed by Johannes Czech, Moritz Willig and Alena Beyer as a semester project for the course *Deep Learning: Methods and Architectures* as a crazyhouse engine with the CNN trained in a supervised learning setting on human games from the open source chess server Lichess [8].
Today it is a Multi-Variant engine supporting all chess variants featured on Lichess as well as the Chinese variant Xiangqi. Different Network Parameters were trained for the different variants in supervised as well as reinforcement learning settings.
The engine created in this work will be using two of CrazyAra's networks, for one the setup for crazyhouse chess, which based on the earlier supervised learning parameters, further trained with reinforcement learning, achieved superhuman performance and beat the human world champion 4:1 [cite master thesis johannes], as well as the supervised learning trained parameters for classical chess.

4 Engine Composition

4.1 General Observations

The aim of this chapter is to create a working Minimax engine for Crazyhouse as well as Classical Chess that bases its Evaluation and Policy on a Convolutional Neural Network. This sets a couple of boundaries and demands investigation into the choice of algorithm as well as the improvements used.

4.1.1 Convolutional Neural Network as Policy and Value Predictor

Using the Neural Network to evaluate nodes and suggest a move ordering is way more computationally expensive than evaluation and move generation for other Minimax approaches, this means we want to use an algorithm that searches the least amount of individual nodes. The algorithm should be enhanced with a Transposition Table to further reduce the amount of nodes that actually need to be evaluated. We also want to enhance it with an additional Transposition Table solely for the Neural Network evaluations to save the predicted Values and Policies independently of the other information, as they don't change throughout the different search depths.
As the positional evaluation of the Neural Network falls in the range [-1,1], a Minimax algorithm for real valued scores is necessary, MTD-bi seems to be the best suited and most node efficient when it comes to individual nodes searched.
Since the move ordering will be suggested by the Neural Network Policy, no other measures will be taken to improve the search order except for the best/refutation moves saved in the Transposition Table.
The Engine will also not feature Quiescence Search, as its purpose is to stabilize search and combat the horizon effect[2.7], both of which should already be addressed by the more complex and intuition-like positional evaluation of the Neural Network.

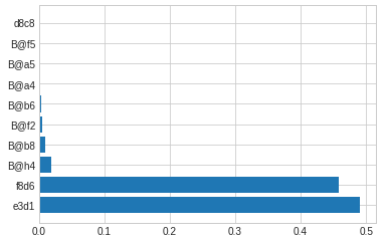

Figure 4.1: Top ten moves taken from the sorted Policy of a crazyhouse position. The X-Axis shows the normalized probability of being the best move according to the Neural Network evaluation.

4.1.2 Branching Reduction via Policy Quantiles (Iterative Widening)

As a consequence of the nature of Minimax the number of nodes searched grows exponentially with the depth of the search, this leads to obvious problems if the branching factor between depths becomes large and makes the use of Minimax intractable in many domains.
To counter this problem, we propose the use of Policy Quantiles, taking advantage of the representation of the Policy Output of the Neural Network.
During the search the child nodes are generated from the Policy Output for each position, traversing the subtrees from the most promising move, with the highest probability, to the least promising, with the lowest probability.
Having faith in the ability of our Network outputs, we expect the best move to be under those with the highest probabilities, meaning that during the search these subtrees will generate Alpha-Beta bounds leading to Cutoffs in the less promising subtrees, who are not contributing to tightening the bounds anyway. Based on this assumption we justify the use of Policy Quantiles, meaning that depending on the depth of the search we only expand the top X-% of child nodes, while still maintaining a high probability of having the best move in our search tree. Figure 4.1 shows the top ten moves from a crazyhouse position and even though the chosen position is not very sharp the Neural Network has very clear favorites. Taking a Quantile of the top 95% of moves or even the top 99% can still reduce the number of moves expanded from multiple hundreds to ten or even less in some positions.
This of course can lead to us not expanding the best move if it is not in the Neural Network's top moves. Because of this, a set of Quantiles of increasing size will be used, eventually capping out at 100%. The choice of Quantile will be coupled to the search depth and will

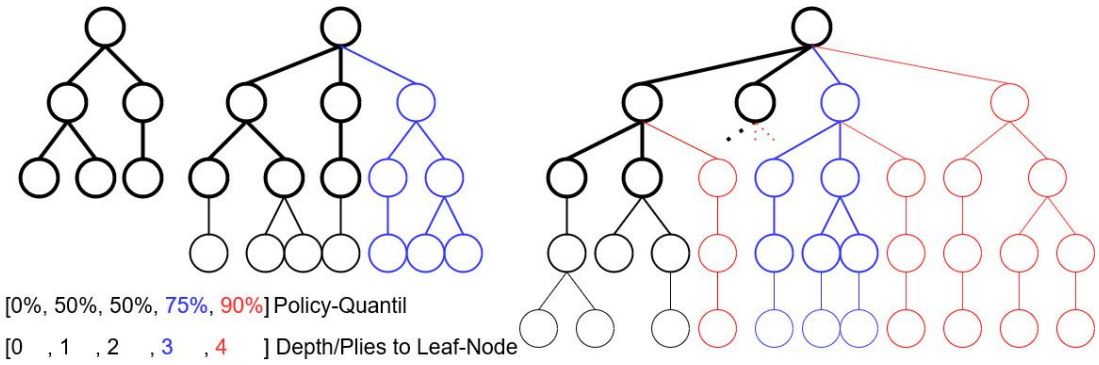

Figure 4.2: Iterative Widening via Policy Quantiles. The three trees are of the same search to depth 2, 3 and 4. The Quantile applied to the Neural Network's Policy changes depending on the remaining plies to the leaf nodes. The blue nodes are additionally searched when switching from depth 2 to 3. The red nodes are searched when switching from depth 3 to 4. The Quantile for depth 0 is just padding, as we don't expand the children of leaf nodes.

be used in an iterative deepening framework to also increase the breadth of the tree with increasing search depth. An example on how this would look in a search tree is given in Figure 4.2.

4.2 Prototyping

To allow for fast development, the prototypes developed in the work are all written in Python.
As the underlying Alpha-Beta implementation will be in the Negamax style and the engine will be using the neural networks of the CrazyAra project, it will be named AraMax from here on out.

4.2.1 Single-Threaded Prototype

The first AraMax prototype is based on the Open Source MTD-Bi Engine for classical chess, Sunfish[16]. As sunfish is single-threaded and the first prototype is going to be used as a proof of concept, the engine will stay single threaded for now.

Sunfish itself implements the MTD-Bisection algorithm in combination with a Transposition Table, the best/refutation-move heuristic and bases its heuristic state evaluation on Piece-Square-Tables. As the state evaluation is going to be based on the neural network, the Piece-Square-Tables and all evaluation code have been removed and replaced by the CNN evaluation. In addition to its own board representation, sunfish has its own move generation written for classical chess. As AraMax is supposed to be able to play classical and crazyhouse chess though, they were both removed and replaced by the *Python-chess* library which supports board representations and move generation for classical, crazyhouse and many other chess variants. As planned, the Move-ordering is based on the CNN's Policy Prediction.
As evaluating the neural network takes several ten-thousands of operations, it is way more expensive than traditional evaluation heuristics. To reduce this disadvantage, we take advantage of the inherent parallelism in Neural Networks and base the CNN evaluation on a GPU, where each layer of the network can be evaluated in one step in parallel. This is contrary to other uses of GPUs in tree search, like in LazySMP or other parallelization-schemes, where the search itself is parallelized and not only the board evaluation.

4.2.2 Multi-Threaded Prototype

Evaluating a single position is not even close to fully utilizing most modern GPUs. To take full advantage of the GPU in its evaluation role, batches of positions have to be evaluated at the same time. For this task, a multithreaded prototype is needed.
Once again, the choice of algorithm is heavily influenced by the increased cost of the Neural Network evaluation, as unnecessarily searched nodes are more costly. Because of this, any communication or locking overheads between threads become less important if it means we can avoid searching parts of the tree unnecessarily. For these reasons, the choice falls on an approach close to Dynamic Tree Splitting[17]. The general structure of DTS is kept but expanded by an additional inference thread solely for the purpose of running the GPU based network inference, Figure [4.3] shows a simplified overview of the thread structure.

Pseudo Code

The Pseudo Code shown in this chapter implements the MTD-Bisection algorithm in an iterative deepening framework using multithreaded Alpha Beta Search with Dynamic

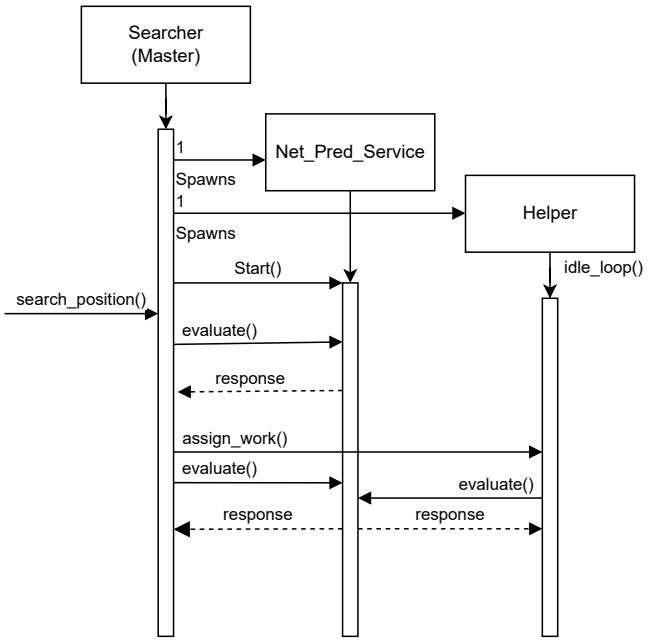

Figure 4.3: A simplified sequence-diagram of the engine's searcher class. The Searcher creates a new thread for the Net_pred_service, which will provide GPU based inference of the Convolutional Neural Network evaluation for all working threads. The Searcher then spawns in the specified amount of helper-threads, 1 for this example, and starts the network inference thread. When told to search a position, the searcher waits for an eligible splitpoint and assigns work to idle helper threads. With enough threads working at the same time, full batches of positions can be evaluated by the Net_pred_service.

```
searchPosition(fen, movelist) # startposition + move history
        board = createBoard(fen)
        for move in movelist:
                board.push(move)
        depth = 0
        while time left and !abortSignal():
                depth = depth + 1
                upper = 1
                lower = -1
                while (lower < upper):
                        mid = (upper+lower)/2
                        score = search(board, depth, mid, mid,0)
                        if score>=mid:
                                lower = score
                        else:
                                upper = score
                move = TT_lookup(getTT_key(board))
                return depth, score, move, time_needed
```

Figure 4.4: This is the entry point into the search. Given a time constraint or some other search condition like *depth* or *number of nodes*, an iterative deepening loop with an MTD-bi search inside is repeated until the condition is met, or the time has run out.

Tree Splitting and a Convolutional Neural Network as Policy and Value Predictor. The DTS Code is inspired by Pseudo Code presented by Tord Romstad, the original founder and first developer of the Stockfish Engine, at Big Tech Day 8 in 2015 [11], presenting the inner workings of the Stockfish Engine before version 7 switched to Lazy SMP in early 2016.

The entry point into the search is the Iterative Deepening loop in Figure 4.4, from inside which the repeated null window searches are started until the bounds have converged, then the depth is increased, and the search started anew until the given time runs out. The heart of the search happens inside the search(...) function in Figure 4.5.

First, if the position is terminal, meaning won, lost or drawn, the board's static evaluation is used to return an exact score. After that, the Transposition Table for Network Evaluations is checked and the Evaluation requested and saved in case none was found. If the depth is

at 0 the estimated value can be returned immediately.
Next the Transposition Table is searched for earlier search results and the alpha beta bounds are updated depending on the depth and result of the cached search. Independently of the depth of the cached search, the best move found there is switched to the front of the policy to be searched first. Now the Policy-Quantile is chosen depending on the depth and is applied to the Policy to generate the list of moves to search. The search loop iterates through all the moves and searches all child nodes in the order given by the Convolutional Neural Net.
After a child node has been completely searched, the algorithm checks if other threads are idle and if so, generates a Splitpoint from the current position to share with the other threads. In this way, the Young Sibling's Wait Concept is satisfied before any parallel search is started.
Here some kind of guard statements can be added to add restrictions like *certain depth remaining* or *certain amount of moves left to search* which have to be satisfied to justify the splitting overhead.
After the search loop concludes, either normally or through split-search, the results are saved in the Transposition Table and returned.
During the search, positions can be encoded as Splitpoints (Figure 4.6) for sharing with other threads. These Splitpoints are comprised of all the information needed for the search, the board, the list of moves and the alpha and beta bounds, best and bestmove needed to track the moves for the Transposition Table, as well as the IDs of its master thread and of all helper threads working on the Splitpoint, as they are needed when a cutoff occurs and work on the node can be stopped.
The Splitpoints master thread calls the split(Splitpoint) function from within the search(...) function (Figure 4.7).
Inside the split function only the IDs of the viable idle helper threads are aggregated for work assignment, as not all idle threads are always allowed to help everywhere. During the search it can happen that the master of a Splitpoint ends up having no work as the helpers are already searching all remaining child nodes. In this case the master can also offer his work to other threads, but only to its own helpers, because if it starts helping some thread somewhere else in the search tree and its helpers finish the original search it will still be busy instead of returning the original search result. If the master is only allowed to help its own helpers, it will always be ready to immediately return the result when the search finishes.
After work has been assigned to all viable threads, the master assigns the work to himself as well and heads into the idle_loop(...) function (Figure 4.8), that all idle threads return to when waiting for new work.
Inside the idle_loop(...) function, the threads wait until work is available and then

```
search(board, depth, alpha, beta, threadID)
        if terminal position:
                return board.exact_eval() # +1, -1, 0

        #NN_eval Transposition Table Lookup
        if lookup successful:
                value, policy = NN_TT_Entry
        else:
                value, policy = request_eval() # send request to net_pred_service
                save_NN_transposition(value, policy)
        if depth == 0:
                return value

        a = alpha #keep original alpha

        # Regular Transposition Table Lookup
        if lookup successful:
                score, move, flag, tt_depth = Score_TT_Entry
                if tt_depth >= depth
                        if flag == EXACT: return score
                        elif flag == LOWER: a = max(a, score)
                        elif flag == UPPER: beta = min(beta, score)
                if a > beta: return score #TT lookup might cause early cutoff
                switch_to_first(policy, move) # best/refutation move heuristic

        qIDX = min(depth, len(quantils)-1) # select Policy Quantil according to depth
        quantils = get_quantils(qIDX)
        moveList = apply_quantil(policy, quantils)
        best = -2 #anything smaller than the lowest score of -1
        while moveList not empty: # search begins
                move = pick_move(moveList)
                board.apply_move(move)
                score = -search(board, depth-1, -beta, -a, quantils, threadID)
                board.pop_move()
                if score > best:
                        best = score
                        bestmove = move
                        a = max(a,best)
                if score > beta: break
                if idle_threads_available(): # check for helpers after YSWC
                        S = Splitpoint(board, moveList, depth, a, beta, threadID, bestmove, best)
                        split(S)
                        best = S.best
                        bestmove = S.bestmove
                        break
        # save Regular Transposition
        if best <= alpha: #the original alpha
                save_score_transposition(best, bestmove, UPPER, depth)
        elif best >= beta:
                save_score_transposition(best, bestmove, LOWER, depth)
        elif:
                save_score_transposition(best, bestmove, EXACT, depth)
        return best
```

Figure 4.5: Pseudo-Code for fail-soft Alpha-Beta search enhanced with a Transposition Table and Neural Net evaluation as Value and Policy Predictor, enhanced for Multi-Threaded search in a Dynamic Tree Splitting fashion.

```
Splitpoint(board, moveList, alpha, beta, masterID, bestmove, best)
        self.board = board
        self.moveList = moveList
        self.alpha = alpha
        self.beta = beta
        self.masterID = masterID
        self.slaveIDs = []
        self.activeThreadCount = 0
        self.bestmove = bestmove
        self.best = best
```

Figure 4.6: A Splitpoint encapsulates all information about the current search, so it can be shared with other threads and the search can be picked up right where it was stopped before the split. In addition, a list of all the threads working on the Splitpoint is kept for messaging purposes when cutoffs occur.

```
split(splitpoint)
        IDs = get_idle_thread_ids()
        for ID in IDs:
                assign_work(ID, splitpoint)
        assign_work(splitpoint.masterID, splitpoint)
        idle_loop(splitpoint.masterID, splitpoint)
```

Figure 4.7: The current thread assigns the Splitpoint to all viable idle threads to be searched in parallel before assigning it to itself as well and heading into the idle_loop(...) which it will immediately leave again to start searching.

```
idle_loop(threadID, splitpoint)
        while True:
                if splitpoint is not None:
                        if work_available(threadID):
                                splitpoint_search(get_work(threadID), threadID)
                        if splitpoint.activeThreadCount == 0:
                                break
                else:
                        if work_available(threadID)
                                splitpoint_search(get_work(threadID), threadID)
```

Figure 4.8: All idle threads return to this loop when they are done working. Inside they wait for new work to be assigned to them. If the waiting thread is a master of a Splitpoint it can break out of the loop when it has to return its Splitpoints search result.

start a splitpoint_search(...) on the assigned Splitpoint. This is done via python queues with blocking retrieve functions, so the threads waiting for work don't needlessly use up cpu-time. If the thread is currently the master of a Splitpoint, the retrieve is non-blocking, but with a short timeout. This is done, so the master can repeatedly check if the activeThreadCount of its Splitpoint is 0, meaning the helpers are done searching and the master can leave the loop and return the search result up the tree.
Figure 4.9 shows the splitpoint_search(...) function. It is built like the search(...) function, but all variables and the moveList are shared and need basic locking mechanisms to work properly. At the start, the working thread copies the Splitpoints board state and then starts a loop of retrieving a new move and starting a new search as the master thread on the resulting position. This is repeated by all threads working on the Splitpoint until either all child nodes have been searched or a cutoff occurs either from the node itself or somewhere further up in the tree.

This concludes the overview of the inner workings of AraMax, which will be evaluated in the next chapter.

```
splitpoint_search(splitpoint, threadID)
        s = splitpoint # rename for space reasons
        s.activeThreadCount += 1
        board = s.board.copy() # every thread needs its own board object
        while s.moveList not empty and not told_to_stop(threadID):
                move = pick_move(s.moveList)
                board.apply_move(move)
                score = -search(board, s.depth-1, -s.beta, -s.alpha, threadID)
                board.pop_move()
                if score > s.best:
                        s.best = score
                        s.alpha = max(s.alpha, s.best)
                        s.bestmove = move
                if score >= s.beta:
                        tell_all_threads_stop(s) # cutoff, stop all workers
                        break
        # The master might be a slave itself and told to stop from further up in the search
        if threadID == s.masterID and told_to_stop(threadID):
                tell_all_threads_stop(s)

        s.activeThreadCount -= 1
```

Figure 4.9: The search function used by threads when working on a subtree in parallel. The list of moves and variables relevant for the search need to be secured by locking mechanisms to avoid undesired behaviour.

5 Evaluation

In this chapter, AraMax, the engine created in chapter 4, will be evaluated for classical as well as crazyhouse chess. The evaluation will consist of two phases for each variant. A first phase of ablation studies to find the optimal settings for the Policy Quantiles and a second phase of direct comparison to MCTS. In the second phase AraMax, equipped with the best Quantile settings, found in the first phase, will be matched against the MCTS engines CrazyAra and ClassicAra, which will be using the same Neural Network configurations and parameters as AraMax. The results of the matches will be measured in the ELO rating system [21].
For the case of classical chess, a lot of benchmarks like test positions have been created and accumulated over the years of research into computer game playing. Because of this the ablation studies will be done on the Eigenmann Rapid Engine Test-suite, a compact set of positions containing many different kinds of chess concepts, and in the form of a round-robin tournament. For classical chess, AraMax and ClassicAra will be matched on the performance on the test suite as well as in playing strength.
In the case of Crazyhouse chess, no such test suites are readily available online. Because of this, the ablation studies will be held only in the form of a round-robin tournament, ranking each configuration on playing strength. The victorious configuration will then be matched against CrazyAra.
To get a good comparison on which of the search types is more efficient and to balance out the inherent speed advantage of CrazyAra and ClassicAra being written in C++ instead of Python, the MCTS engine will be restricted to the same number of Neural Network Evaluations as the Minimax search.

5.1 Classical Chess

In this section AraMax will be compared to ClassicAra, the classical chess variant of CrazyAra, on the performance on the Eigenmann Rapid Engine Test as well as in a series

of matches of 300 games each, where AraMax is allowed to search to a certain depth and ClassicAra searches a previously determined average amount of nodes for each depth. This is done to accommodate for the Minimax engines need to fully explore a given search depth, in the hopes to make the comparison of these approaches as fair as possible.

5.1.1 The Eigenmann Rapid Engine Test(ERET)

The Eigenmann Rapid Engine Test is a suite of 111 chess positions, encapsulating a multitude of different tactical and strategic concepts, collected by Walter Eigenmann to roughly compare engine strengths without the need for long tournaments. The engine is given 15 seconds to search each position and the percentage of correctly solved positions is used as the metric. Solving a position means either finding the best move (confirmed by days of search in conventional engines like Stockfish) or in some cases not playing the at first glance tempting but ”poisonous”, wrong move.
The Eigenmann Rapid Engine Test is on the harder side of Engine Test suites, with LeelaChessZero solving 86/111 and Stockfish solving 85/111 in the given time on a 4 CPU setup in a ranking from July 2019[4].

5.1.2 Ablation Studies and Results on ERET | Phase 1

During the Ablation Studies, 9 different set of 3 Quantiles each have been tested on the ERET in comparison to a set of always 1, so using the full policy all the time. Out of those 9 sets, 7 are closed-ended sets, meaning that the last Quantile in the set is 1 and the search will eventually expand all the same nodes as a full policy Minimax search. The other 2 sets are open-ended, meaning the last Quantile in the set is smaller than 1 and moves deemed too bad by the Neural Network are never expanded. They are displayed in Table 5.1 and were chosen to cover a diverse set of Quantile ranges and step sizes between Quantiles.

The 10 different sets were tested on 2 different batch-sizes for the Neural Network evaluation, batch-size 1 with 2 Threads searching in parallel and batch-size 8 with 16 threads searching in parallel. As can be seen in Table 5.2 contrary to expectation the bigger batch size performs worse under the given conditions, so future matches will be held with a Batch_size of 1 and 2 Threads.
In addition to the performance testing on the ERET the different Q-Sets have been matched against each other in a Round-Robin Tournament. Each Set is matched against all other sets over the course of 9 rounds of 5 simultaneous encounters. Every Encounter will consist

Q-Set1	$[0.5, 0.75, 1]$
Q-Set2	$[0.6, 0.8, 1]$
Q-Set3	$[0.75, 0.9, 1]$
Q-Set4	$[0.75, 0.95, 1]$
Q-Set5	$[0.95, 0.99, 1]$
Q-Set6	$[0.99, 0.995, 1]$
Q-Set7	$[0.99, 0.999, 1]$
Q-Set8	$[0.5, 0.75, 0.99]$
Q-Set9	$[0.9, 0.99, 0.999]$
Q-Set10	$[1]$

Table 5.1: The 10 different sets of Policy-Quantiles for AraMax used in the ablation studies. 7 being closed-ended, 2 open-ended and Nr.10 as control with no restrictions on the policy.

	B_size: 1 Threads: 2	B_size: 8 Threads: 16
Q-Set1	36/111	28/111
Q-Set2	35/111	37/111
Q-Set3	35/111	31/111
Q-Set4	37/111	31/111
Q-Set5	38/111	34/111
Q-Set6	41/111	31/111
Q-Set7	41/111	28/111
Q-Set8	36/111	32/111
Q-Set9	40/111	34/111
Q-Set10	40/111	29/111

Table 5.2: Results of 10 different Quantile-Sets for AraMax with 15 seconds to search on the Eigenmann Rapid Engine Test. Tested with batch-sizes 1 and 8 and twice that amount of threads.

of 300 games played with 1 minute for each engine on 150 different starting positions, with each engine playing both sides once.
As a draw is the most likely outcome in high level classical chess, the opening suite will be chosen from the collection of Anti-Draw openings provided by Stefan Pohl[15]. They are a collection of either specially designed positions or positions curated from human games, that give a slight advantage to one player or change the starting layout to reduce the likelihood of a draw occurring.

As we can see from table 5.3 there was a joint first place between Q-Set Nr.10, the

	vs1	vs2	vs3	vs4	vs5	vs6	vs7	vs8	vs9	vs10	W-L-D	Place
Q-Set1		137-17	149-13	143-21	138-31	143-30	142-22	140-15	135-28	129-20	1256-1247-197	4th(1354.5)
Q-Set2	146-17		139-18	135-20	150-21	147-23	138-20	154-15	135-29	136-20	1280-1237-183	3rd(1371.5)
Q-Set3	138-13	143-18		144-21	140-25	134-15	132-29	137-30	134-25	137-27	1246-1245-209	5th(1350.5)
Q-Set4	136-21	145-20	129-27		144-21	138-31	143-18	134-22	133-22	137-28	1239-1251-210	6th(1344)
Q-Set5	131-31	129-21	135-25	135-21		137-17	142-21	149-25	131-26	132-28	1221-1264-215	10th(1328.5)
Q-Set6	127-30	130-23	151-15	131-31	146-17		138-27	144-17	121-33	143-16	1231-1260-209	7th(1335.5)
Q-Set7	136-22	142-20	139-29	139-18	137-21	135-27		140-23	128-31	129-22	1225-1262-213	8th(1331.5)
Q-Set8	145-15	131-15	133-30	144-22	126-25	139-17	137-23		137-28	136-27	1228-1270-202	9th(1329)
Q-Set9	137-28	136-29	134-25	145-22	143-26	146-33	141-31	135-28		134-31	1251-1196-253	1st(1377.5)
Q-Set10	151-20	144-20	136-27	135-28	140-28	141-16	149-22	137-27	135-31		1268-1213-219	1st(1377.5)

Table 5.3: Results of the Round Robin Tournament held with the different Q-Sets for AraMax in classical chess.To safe space, the table only shows the wins and draws for each encounter. A win is valued at 1 point and a draw at $\frac{1}{2}$point.

control set, and Q-Set Nr.9, the open-ended set with higher quantile values, but we can also see, that the spread in the tournament is not very big, with the first places reaching 1377.5/2700 Points and the last place reaching 1328.5/2700 Points. This indicates that, for classical chess at least, Policy-Quantiles do not have a positive impact on playing strength despite the slightly better performances by some Q-Sets on the ERET.
As Q-Set Nr. 6 had the best performance on the ERET together with Q-Set Nr.7 but had a slightly better score in the tournament and a better performance in their 1vs1 match, it will be used in the match against ClassicAra in phase 2.

5.1.3 AraMax vs ClassicAra | Phase 2

In the first part of this second phase of the evaluation, ClassicAra is also tested on the ERET. For this, the number of Neural Network evaluations used by the Minimax engine equipped with Q-Set Nr.6 was recorded for every one of the 111 positions and ClassicAra was set to search each position for the corresponding number of evaluations.
As we remember, Quantile-Set Nr. 6 reached 41/111 correctly solved positions and ClassicAra in the previously described setup also solves 41/111 positions, indicating that

both approaches have a similar search efficiency. As a reference to the power of the Convolutional Neural Networks used, the ERET was performed with just one network evaluation and the best move of the Policy was chosen every time. The raw network's highest rated move was the correct solution 22/111 times.

For the second part of the second phase, the average amount of Neural Network evalua-

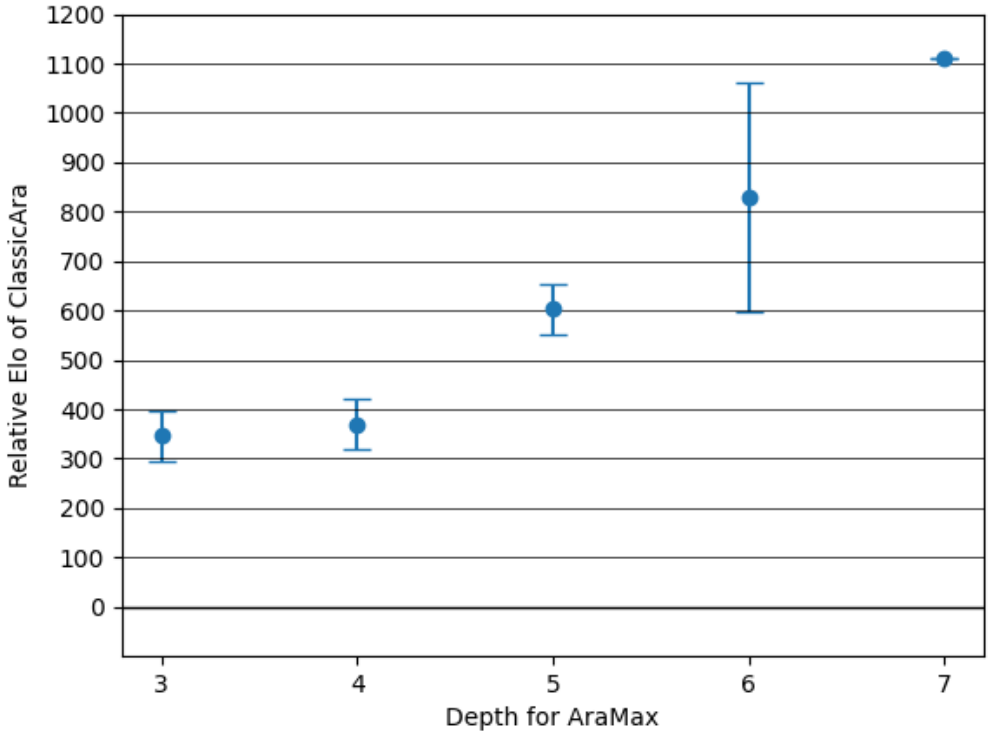

Figure 5.1: The relative ELO of ClassicAra against AraMax in a 300 game encounter for each depth. At depth 7 ClassicAra won 299-0-1, so a confidence interval could not be computed

tions for each search depth was recorded for Quantile-Set 6's performance on the ERET. Afterwards, a series of matches was held where ClassicAra is allowed to search as many nodes as the averages recorded for each depth. This is done to get an overview over the search efficiency of both approaches, independently of the implementation specifics.

	vs1	vs2	vs3	vs4	vs5	vs6	vs7	vs8	vs9	vs10	W-L-D	Place
Q-Set1		158-2	171-2	164-2	149-1	148-1	120-0	101-1	134-1	130-1	1275-1414-11	6th(1281.5)
Q-Set2	140-2		165-1	152-1	136-1	132-1	96-0	97-2	127-2	105-1	1150-1539-11	9th(1155.5)
Q-Set3	127-2	134-1		133-3	149-1	123-0	92-3	85-1	119-1	98-4	1060-1624-16	10th(1068)
Q-Set4	134-2	147-1	164-3		155-0	145-1	99-3	95-2	126-0	130-2	1195-1491-14	7th(1202)
Q-Set5	150-1	163-1	151-0	145-0		141-0	117-1	84-0	114-0	117-1	1182-1514-4	8th(1184)
Q-Set6	151-1	167-1	177-0	154-1	159-0		127-1	111-2	138-1	117-2	1301-1390-9	5th(1305.5)
Q-Set7	180-0	204-0	205-2	198-3	182-1	172-1		132-2	154-0	146-4	1573-1114-13	2nd(1579.5)
Q-Set8	198-1	201-2	214-1	203-2	216-0	187-2	166-2		168-0	170-1	1723-966-11	1st(1728.5)
Q-Set9	165-1	171-2	180-1	174-0	185-0	161-1	146-0	132-0		148-4	1463-1228-9	4th(1467.5)
Q-Set10	169-1	194-1	198-4	168-2	182-1	181-2	150-4	129-1	148-4		1519-1161-20	3rd(1529)

Table 5.4: Results of the Round Robin Tournament held with the different Q-Sets for AraMax in crazyhouse chess.To safe space, the table only shows the wins and draws for each encounter. A win is worth 1 point and a draw $\frac{1}{2}$point

As can be seen in Figure 5.1 the Monte Carlo Tree Search outperforms the Minimax Search for every depth with its relative performance increasing with increasing depth as well.

5.2 Crazyhouse Chess

In this section, AraMax will be matched up against CrazyAra, the original variant of the Multi-variant engine, whose Convolutional Neural Network parameters power the Value and Policy prediction of AraMax. The evaluation will consist of two phases again, a phase of Ablation studies for the optimal Quantile-Set followed by a series of matches between the two engines.

5.2.1 Ablation Studies | Phase 1

Sadly, the amount of material regarding the crazyhouse variant is not as vast as for classical chess and no established widely available test suites exist to my knowledge. For this reason, the Ablation Studies will be held in the form of a round-robin tournament between the different Quantile-Sets.
In Each Round of the Tournament, 300 games with 1 minute per side will be played between each engine pair on 150 different starting positions, switching sides on each position once. The Tournament will consist of 9 rounds, after which every pairing has happened once. The overall best Quantile-Set will then be used in the match against CrazyAra like before in the match against ClassicAra. Contrary to the results in classical

chess, the results of the crazyhouse tournament are more spread out, with Q-Set Nr.8, one of the open-ended ones, claiming first place with 1728.5/2700 Points, while the last place only has 1068/2700 Points. In second place is Q-Set Nr.7 with 1579.5/2700 Points and in third place is the control, Q-Set Nr.10 with 1529/2700 Points. This shows that in crazyhouse chess, with its higher branching factor, Policy Quantiles can be used to effectively increase the playing strength of AraMax.
As Q-Set Nr.8 was the strongest, it will be used in the match in the second phase.

5.2.2 AraMax vs CrazyAra | Phase 2

To measure the average amount of NN evaluations for each depth, AraMax was set to play a match of 100 games against itself, equipped with Q-Set Nr.8. Just like for classical chess a match of 300 games was held for each combination of depth to NN evaluations, the results can be seen in Figure 5.2. While AraMax was slightly stronger in shallower depths, the same trend as for classical chess occurred and the Monte Carlo Tree Search performed better with increasing depth/nodes.

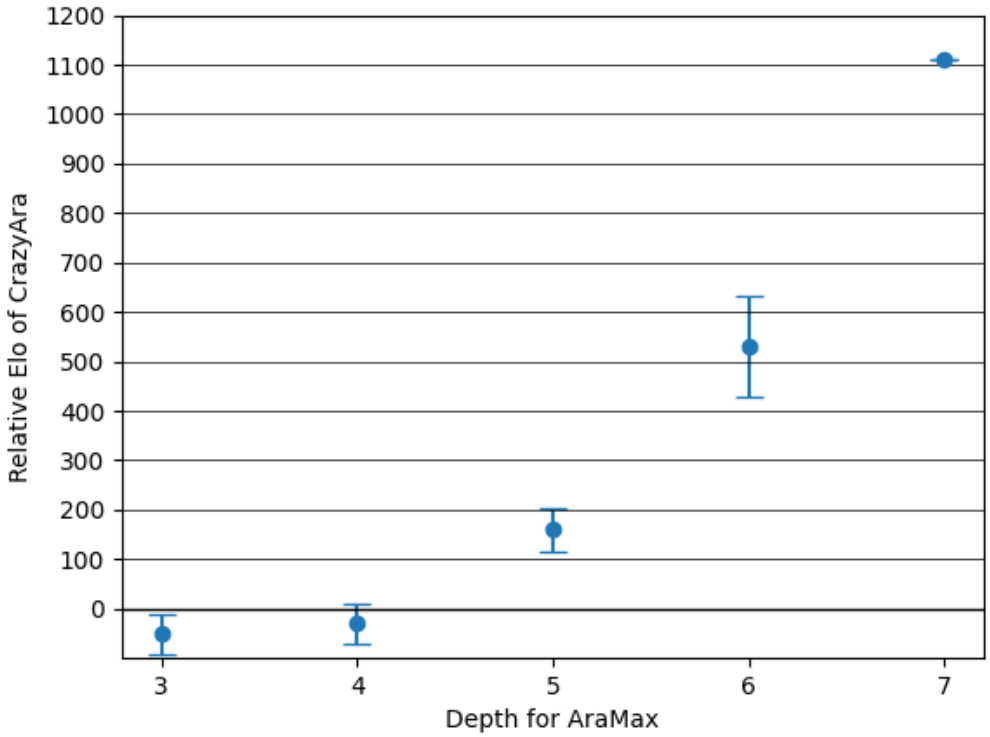

Figure 5.2: The relative ELO CrazyAra against AraMax in a 300 game encounter for each depth. At depth 7 CrazyAra won 300-0-0, so no confidence interval could be computed

6 Conclusion

6.1 Summary

In this work a multi-variant Minimax engine for chess and crazyhouse, AraMax, was created based on the convolutional neural networks of the CrazyAra project. In addition, the use of Policy Quantiles, a method to reduce branching throughout the search based on the Neural Network's Policy, was proposed and successfully implemented in AraMax. The Evaluation has shown, that while Policy Quantiles were not, or only slightly effective in classical chess, they were able to effectively increase the playing strength of AraMax in crazyhouse chess. This shows, that Policy Quantiles might be used to prune the search tree and make Minimax approaches more viable in environments with higher branching factors.
By directly comparing MCTS and Minimax approaches using the same network parameters, we have shown that, while Minimax can be used in combination with neural networks, MCTS seems to provide the more efficient search in terms of network evaluations to playing strength. CrazyAra and ClassicAra continuously improved their performances with increasing depth/Number of evaluations in comparison to AraMax and eventually dominated AraMax in terms of playing strength.

6.2 Future Work

As Policy Quantiles have been shown to be effective, an investigation into different sizes of Quantile-Sets would be interesting to find out if further improvements can be achieved. For this, a faster AraMax implementation in C++ would be useful to make ablation studies to higher depths with larger sets of quantiles more viable.
Additionally, an investigation into the combination of Policy Quantiles with larger step

sizes for iterative deepening could be of interest. Iterative deepening while being useful for time management is most notably contributing information about promising moves to the search. Combining this with Policy Quantiles to get cheaper but not perfect information about the next few new depths and then increasing the depth by 2 or more at a time could lead to faster depth exploration while still gaining some of the benefits of searching the more promising moves first and generating more cutoffs.
Finally, investigating if Policy Quantiles can be incorporated into a best-first search approach somehow, might yield some promising results.

Bibliography

[1] *A new chess engine : m8 (comming not so soon)*. URL: http://www.talkchess.com/forum3/viewtopic.php?t=55170&start=11 (visited on 05/01/2022).

[2] Murray Campbell, A. Joseph Hoane, and Feng-hsiung Hsu. "Deep Blue". In: *Artif. Intell.* 134.1–2 (Jan. 2002), pp. 57–83. ISSN: 0004-3702. DOI: 10.1016/S0004-3702(01)00129-1. URL: https://doi.org/10.1016/S0004-3702(01)00129-1.

[3] Quentin Cohen-Solal and Tristan Cazenave. "Minimax Strikes Back". In: *CoRR* abs/2012.10700 (2020). arXiv: 2012.10700. URL: https://arxiv.org/abs/2012.10700.

[4] *ERET test results: Leela-Chess the new Number One...* URL: http://talkchess.com/forum3/viewtopic.php?f=2&t=71166 (visited on 05/01/2022).

[5] Robert Hyatt. *"CUT"vs"ALL" nodes*. CCC. URL: http://www.talkchess.com/forum3/viewtopic.php?topic_view=threads&p=398061&t=38317 (visited on 04/30/2022).

[6] Levente Kocsis and Csaba Szepesvári. "Bandit Based Monte-Carlo Planning". In: *Machine Learning: ECML 2006*. Ed. by Johannes Fürnkranz, Tobias Scheffer, and Myra Spiliopoulou. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 282–293. ISBN: 978-3-540-46056-5.

[7] Yann Lecun and Yoshua Bengio. "Convolutional networks for images, speech, and time-series". English (US). In: *The handbook of brain theory and neural networks*. Ed. by M.A. Arbib. MIT Press, 1995.

[8] *Lichess.org*. Lichess.ORG. URL: https://lichess.org/ (visited on 05/01/2022).

[9] Jacek Mańdziuk and Daniel Osman. "ALPHA-BETA SEARCH ENHANCEMENTS WITH A REAL-VALUE GAME-STATE EVALUATION FUNCTION". In: *ICGA Journal* 27 (2004). 1, pp. 38–43. ISSN: 2468-2438. DOI: 10.3233/ICG-2004-27104. URL: https://doi.org/10.3233/ICG-2004-27104.

[10] Aske Plaat et al. “A New Paradigm for Minimax Search”. In: (2014). DOI: 10.48550/ARXIV.1404.1515. URL: https://arxiv.org/abs/1404.1515.

[11] Tord Romstad. *Parallelism and Selectivity in Game Tree Search*. [Online; accessed 1-May-2022]. 2015. URL: https://www.youtube.com/watch?v=R0L3AuJUkk0.

[12] Julian Schrittwieser et al. “Mastering Atari, Go, chess and shogi by planning with a learned model”. In: *Nature* 588.7839 (Dec. 2020), pp. 604–609. DOI: 10.1038/s41586-020-03051-4. URL: https://doi.org/10.1038%2Fs41586-020-03051-4.

[13] David Silver et al. “A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play”. In: *Science* 362.6419 (2018), pp. 1140–1144. DOI: 10.1126/science.aar6404. eprint: https://www.science.org/doi/pdf/10.1126/science.aar6404. URL: https://www.science.org/doi/abs/10.1126/science.aar6404.

[14] David Silver et al. “Mastering the game of Go with deep neural networks and tree search”. In: *Nature* 529 (Jan. 2016), pp. 484–489. DOI: 10.1038/nature16961.

[15] *Stefan Pohl Computer Chess*. URL: https://www.sp-cc.de/ (visited on 05/01/2022).

[16] *Sunfish*. URL: https://github.com/thomasahle/sunfish (visited on 05/01/2022).

[17] *The DTS high-performance parallel tree search algorithm*. Robert Hyatt. URL: https://craftychess.com/hyatt/search.html (visited on 05/01/2022).

[18] *The Stockfish GitHub repository*. Stockfish open-source Community. URL: https://github.com/official-stockfish/Stockfish (visited on 05/01/2022).

[19] Wikipedia contributors. *Chess — Wikipedia, The Free Encyclopedia*. [Online; accessed 1-May-2022]. 2022. URL: https://en.wikipedia.org/w/index.php?title=Chess&oldid=1084878623.

[20] Wikipedia contributors. *Crazyhouse — Wikipedia, The Free Encyclopedia*. [Online; accessed 1-May-2022]. 2022. URL: https://en.wikipedia.org/w/index.php?title=Crazyhouse&oldid=1081194151.

[21] Wikipedia contributors. *Elo rating system — Wikipedia, The Free Encyclopedia*. [Online; accessed 30-April-2022]. 2022. URL: https://en.wikipedia.org/w/index.php?title=Elo_rating_system&oldid=1085105679.

[22] Wikipedia contributors. *Shannon number — Wikipedia, The Free Encyclopedia*. [Online; accessed 1-May-2022]. 2022. URL: https://en.wikipedia.org/w/index.php?title=Shannon_number&oldid=1084877187.

[23] Wikipedia contributors. *Top Chess Engine Championship — Wikipedia, The Free Encyclopedia*. [Online; accessed 1-May-2022]. 2022. URL: https://en.wikipedia.org/w/index.php?title=Top_Chess_Engine_Championship&oldid=1084543610.