– Independent Work Report Fall 2023 –

AlphaOne: A Monte-Carlo Tree Search Solver for Chess

Jiacheng (Joseph) Xu
Adviser: Zachary Kincaid

Abstract

Recently, the AlphaZero chess engine has found great success using Monte Carlo Tree Search (MCTS) combined with deep neural networks and reinforcement learning to play chess at a high level, but its neural networks are very computationally expensive. This project explores the effectiveness of using AlphaZero's core algorithm of MCTS combined with the usage of static evaluations (as used in Alpha-Beta-based engines) as a computationally cheaper substitute for AlphaZero's value network. As a comparison, the same static evaluation function was used to build an Alpha-Beta engine. This project found that MCTS's key flaw of tactical shortsightedness limits its ability to perform well even with the usage of static evaluations; in comparison, the Alpha-Beta engine performed well at approximately 2000 ELO, even on a relatively low depth by modern standards of 4.

1. Introduction

People have been developing and playing chess for well over a millennium now, with the earliest versions dating back to the 6th century CE, when people were playing an early form called Chaturanga in India. In more recent history, people have been using computation to study chess by writing programs called chess engines to explore and play the game. The goal of a chess engine is to find the best move given a legal position while accounting for possible opponent responses.

This problem is especially difficult because of the vast search space that chess presents. Computationally, chess is generally modeled in a search tree structure, where each node is a board position or state. Children nodes are generated by enumerating the next legal moves from the parent node's position. Leaf nodes are terminal positions where the game has ended either by one side winning or by a draw. Various estimates exist for the size of this search space, such as Shannon's number, which posits that the total number of chess games is on the order of 10^{120} [10].

One dominant approach in the realm of chess engine development relies on Minimax search and its related optimizations or improvements. In a truncated version of Minimax search, the search tree is constructed for a certain depth (e.g. search 14 ply[1] into the future), and the 2 players attempt to find the best option given what the other player can do. A common and powerful optimization upon Minimax which improves performance while maintaining correctness is called Alpha-Beta search; this search prunes out parts of the search tree by tracking the best current options for each player using 2 variables, α for player 1 and β for player 2.

While the Alpha-Beta search is fairly thorough and can perform reasonably well even at a relatively lower depth (e.g. search 4 ply into the future), it relies heavily on domain knowledge. Beyond the basic game rules, it requires a static evaluation function and a heuristic move ordering function to be informative and efficient. The role of the static evaluation function is to give an estimate of how "good" a given position is. In chess, this requires understanding domain-specific concepts such as pawn structures or king safety. Once constructed, the static evaluation function is then used at the terminal points of the Alpha-Beta search to guide the 2 players' decisions. The role of the heuristic move ordering function is to estimate how "good" a given move is. This is then used to guide the order in which certain branches of the search tree are traversed. When "good" branches are searched first, this is ideal, as the Alpha-Beta optimization allows us to bypass looking at "bad" branches. Various miscellaneous optimizations can then be made in addition to further improve performance.

Alpha-Beta search has led to the successful development of many very strong engines, such as Stockfish, which has been rated at 3500+ ELO [7]. ELO is a relative rating of player strength, where players with higher ELO are estimated to have higher winning probabilities than players with lower ELO. In the ELO system, as the gap between player ELOs increases, the estimated winning probability of the higher-rated player also increases. A rule of thumb is that a 100-point gap corresponds to a 64% estimated winning probability for the higher-rated player, and a 200-point gap corresponds to a 75% estimated winning probability [4]. For reference, the peak human chess

[1] A ply is a half-move, where only one player takes a turn.

ELO of 2882 was achieved by Magnus Carlsen in 2014[5]. A comprehensive list of hundreds of chess engines and their ELOs (including both commercial and freely available engines) can be found on the Computer Chess Rating List (CCRL) website, including Dragon, Beserk, and more[7].

While the classical chess engines have mostly relied on Alpha-Beta search and its related algorithms, some other engines have started to make use of Monte-Carlo Tree Search (MCTS). Whereas Alpha-Beta search enumerates every sequence of moves up to a certain depth and pruning when it is certain pruning is permissible, MCTS foregoes this precise approach for a stochastic approach by searching probabilistically and evaluating potential candidate positions by running a large number of random simulations [2]. The intuition, then, is that "good" positions should lead to winning games more often than not in enough random games.

Notably, the DeepMind team has used MCTS with deep neural networks and reinforcement learning techniques to develop AlphaZero, which has even been shown to rival even Stockfish 8[12]. Other examples include LC0 (Leela Chess Zero), which was inspired by AlphaZero, and Dragon. AlphaZero receives the Zero in its name as it requires no prior human domain knowledge of the game other than the rules of the game. Using reinforcement learning and deep neural networks, it learns to derive salient features of the game to provide estimated evaluations of board positions, which it then uses to guide its search algorithm. While AlphaZero searches fewer positions per second compared to Alpha-Beta based algorithms like Stockfish (80k positions per second vs 70M positions per second), it makes up for this difference by using its deep neural networks to focus selectively on what it deems to be promising branches of the search tree[12].

While AlphaZero's performance is remarkable, one issue is that it requires immense computational resources. It used 5000 1st-generation Tensor Processing Units (TPUs) to generate self-play games to learn and 64 2nd-generation TPUs to train its neural networks over 44 million training games[12]. These resources were critical in building the reinforcement learning networks that AlphaZero used to learn how to evaluate boards.

This project aims to combine ideas from AlphaZero and Alpha-Beta based engines by taking AlphaZero's core algorithm of MCTS, but making its evaluations computationally cheaper by

using prior domain knowledge in the form of static evaluations (as done in Alpha-Beta engines), thereby eliminating the need to train computationally expensive deep neural networks. Instead of generating self-play games to learn how to evaluate the features of the board from scratch, this project will instead use handcrafted feature creation and supervised learning techniques to create a static evaluation function to perform similar functions as AlphaZero's deep neural networks. This ultimately results in a computationally cheaper version of AlphaZero which I've named AlphaOne, as it does not use Zero prior domain knowledge. As a comparison, the same static evaluation function also will be used to create an Alpha-Beta engine.

2. Background and Related Work

2.1. Static Evaluations

We can define the *static evaluation* as a heuristic function that estimates the goodness of any given board[2] in a way such that a higher value means that the board position is better for the White player, and a lower value means that the board position is better for the Black player. In other words, we create a heuristic evaluation of the board that White wants to maximize and that Black wants to minimize.

As a basic example, let us assign each piece type a value (pawn = 1, bishops/knights = 3, rook = 5, queen = 9). The *material* for each player is defined as the sum of the corresponding piece values across all of the player's remaining pieces. Generally, having more material in chess is advantageous, for example by allowing a strong attack that is hard to defend against. Therefore, we can define our static evaluation function as the difference between White's material and Black's material. White would prefer to maximize this value, as it indicates White has more material; for a similar reason, Black would prefer to minimize this value.

While this is a useful estimation, it is important to derive more features about the board, because the material count can be misleading. For instance, it is not uncommon for a player to be winning in terms of material, but losing because they are on the verge of being checkmated.

[2]The terms board, position, and state are used interchangeably in this project's context.

2.2. Truncated Minimax with Static Evaluations

Untruncated Minimax gives the optimal strategy assuming that the opponent also chooses optimal moves. If the opponent deviates from the optimal moves, then Minimax gives better payoffs than it would have otherwise. Here, payoffs refer to the values given to terminal states (states where the game has been won or drawn) from the perspective of the White player (e.g. win = 1, draw = $\frac{1}{2}$, loss = 0).

While untruncated Minimax searches all possible sequences of states up until the termination of a game, in practice this is not feasible for chess as the search space is too large. Instead, it is more practical to cut off the search after a set depth to limit the size of the search tree and return a heuristic estimate by using the static evaluation function. This gives us a *truncated Minimax*.

Given a position s, define $moves(s)$ as the set of positions that arise from s after playing the possible legal moves from s, $eval(s)$ as the static evaluation value given s, and $turn(s)$ as the current side to move in s.

Then, we can define the truncated Minimax value of s, given a specified search depth d as:

$$minimax_d(s) = \begin{cases} eval(s) & \text{if } d = 0 \\ \max_{c \in moves(s)} minimax_{d-1}(s) & \text{if } turn(s) = \text{White} \\ \min_{c \in moves(s)} minimax_{d-1}(s) & \text{if } turn(s) = \text{Black} \end{cases}$$

If the branching factor is b and the depth of this search is d, then the time complexity is $O(b^d)$. In chess, b is given by the number of legal moves in a given position, a value that can vary, but suppose on average it is approximately 30. Depth is a parameter that we can control and generally, the greater depth an engine can search, the greater its playing strength. If b or d could be decreased, then the time required for this algorithm can be improved.

2.3. Alpha-Beta

Alpha-Beta pruning, an improvement on Minimax, is defined as follows for $\alpha \leq \beta$:

$$(\alpha, \beta) - minimax(s) = max(\alpha, min(\beta, minimax(s)) \tag{1}$$

We can observe that by definition, Alpha-Beta pruning will always return a value between $[\alpha, \beta]$. If $minimax(s) < \alpha$, it returns α instead, and if $minimax(s) > \beta$, it returns β instead. In practice, in the course of executing Alpha-Beta pruning in code, $\beta <= \alpha$ is used as a cutoff condition when pruning occurs since values must be within $[\alpha, \beta]$.

With best performance, the time complexity of Alpha-Beta improves to $O(b^{d/2})$. In practice, such performance gains are realizable.

2.4. Deep Blue

Deep Blue is one of the early major successes in the chess engine world, known for beating Garry Kasparov, and it relied on NegaScout, which is a variation on Alpha-Beta search[3]. Deep Blue's search is non-uniform and therefore prioritizes what it deems to be promising branches of the search while having "insurance" against short-term blunders by searching at least a minimum depth.

Another of the main features behind its success was its complex evaluation function, which is able to account for roughly 8000 features. Moreover, feature values could be either static or dynamic; static means they remain constant through the search, and dynamic means they are scaled throughout the search.

Other techniques employed were using a massively parallel system as opposed to a single-process search, a large database of Grandmaster[3] level games and openings, and dedicated hardware to perform functions such as evaluation. Altogether, in practice, Deep Blue performed at an average depth of around 12.2 for a 3-minute search.

2.5. MCTS

Monte-Carlo Tree Search uses 4 steps: Selection, Expansion, Simulation (also known as Rollout), and Backpropagation.

[3] Grandmaster is the highest chess title.

MCTS initializes as a tree with just a root node, representing the start of the game. It creates children nodes to represent positions that can be legally reached from parent positions. In each node, it stores the following information:

1. Q: the total payoff[4] of simulations from this node.
2. N: the total number of simulations from this node
3. parent: the parent node of this node

In *Selection*, MCTS greedily continues selecting the best child node in its tree until it hits one of its leaf nodes. To quantify the goodness of children nodes, MCTS uses the UCT (Upper Confidence bound for Trees) metric.

For a given node n, its UCT value is defined as:

$$\frac{n.Q}{n.N} + c\sqrt{\frac{log(n.parent.N)}{n.N}} \tag{2}$$

This quantifies the idea of exploitation versus exploration. The left summand is directly proportional to the expected payoff, and nodes with higher expected payoffs are generally worth exploring further to develop promising winning ideas. The right summand is directly proportional to the parent's N while being inversely proportional to the node's N, meaning that if a node is relatively unexplored compared to its parent, then it's worth investing some exploration into. The right summand is also proportional to c, which is called the exploration factor. If this value is higher, then the right summand is weighted more strongly, thereby encouraging more exploration. It is important to balance exploitation versus exploration because exploitation allows MCTS to continue developing winning ideas, whereas exploration ensures MCTS does not overlook potentially strong ideas that manifest in the future.

In *Expansion*, MCTS adds a new node to the tree from the selected node, thereby expanding the search depth. The vanilla MCTS algorithm chooses such nodes uniformly randomly, but this can be

[4]As an implementation detail, here, the payoff is defined from the perspective of the current node's associated player, rather than solely from the perspective of White.

improved by estimating the goodness of moves and expanding with a higher probability the nodes that arise from likely good moves.

In *Simulation/Rollout*, a random game is run from the expanded node. The vanilla MCTS algorithm plays random games by selecting moves uniformly randomly until the game ends.

Lastly, in *Backpropagation*, the results of the simulation are used to update the tree from the leaf back up to the root by increasing the Q values of the side that won the simulation and the N values of the nodes encountered in traversing back up.

2.6. AlphaZero

The key insights of AlphaZero are that instead of expanding randomly and simulating out random games, it is more useful to use more informative methods; namely, it uses reinforcement learning and deep neural networks to create a *value network* and a *policy network*[12]. The goal is to produce the following function:

$$(p, v) = f_\theta(s)$$

Given an input position s with a parameterization of θ, the policy network outputs a probability distribution p where $p_a = Pr(a|s)$ is the probability of picking move a from position s, and the value network outputs a scalar v that estimates the expected payoff z from s, giving $v \approx E[z|s]$. This function is learned from self-play and then is used to guide Selection and Expansion. The Selection stage now relies on UCT values that are derived from the value network's estimates of expected payoff, rather than those derived from random simulations as in vanilla MCTS. In the expansion stage, instead of picking nodes at uniform random, AlphaZero uses the probability distribution p to pick nodes in a more informed manner; this allows it to prioritize branches that are likely promising.

2.7. Alpha-Beta vs MCTS Pros and Cons

While Alpha-Beta is a very thorough, systematic search, one potential downside is that it requires a well-crafted static evaluation function, which generally requires great domain knowledge about

chess to quantify what aspects of a position make it good or bad.

On the other hand, by design, MCTS can be good at generalizing to many games because the simulation step does not necessarily require any domain-specific knowledge (besides knowing the rules). Additionally, it can be selective about which branches of the game search tree it wants to dedicate time to, whereas Alpha-Beta must explore all branches up to a certain depth (with the exception of known prunable branches). This results in what has been described as human-like play, as humans often engage in a similar thought process of identifying promising branches and dedicating their time to exploring those variations more in-depth[12]. More importantly, this can result in very well-developed, deeply explored ideas, if the correct branches of the search tree are chosen.

However, depending on the game, MCTS may require a high amount of computational power because random game simulations are not necessarily good at proving the goodness of a position. For instance, in chess, often a side is winning only if it finds a very specific sequence of moves, but this may not be sampled frequently enough or at all during the random simulations. As such, engines such as AlphaZero have opted to estimate the expected payoff using more sophisticated methods such as by using deep neural networks and reinforcement learning, as opposed to random simulations.

3. Approach

3.1. Static Evaluation

Instead of using AlphaZero's value network, this project will substitute a static evaluation to estimate the goodness of a position. This same static evaluation will also be used in an Alpha-Beta engine as a comparison.

The preliminary version of the static evaluation used in this project relies on a method called tapered evaluation, which accounts for material and positional advantages or disadvantages based on the phase of the game. Intuitively, the material component of tapered evaluation accounts for tactics while the positional bonuses or penalties assigned account for strategy; Tactics refer to a series of

moves that create immediate threats such as winning material or checkmating, whereas strategy refers to the long-term plan to gain an advantage, usually by careful maneuvering and positioning.

To build upon this function, it is useful to incorporate other features. Broadly, it is informative to include information that accounts for material, king safety[5], piece play[6], and pawn play[7].

It is difficult to estimate what the weight of these features should be manually, and it is difficult to verify whether these weights are optimal. This project uses supervised learning in order to learn these weights based on positions with previously known evaluations given by Stockfish. Although this project tested various methods (e.g. Random Forest Regression, Support Vector Machine Regression, etc.), the most practical one ultimately was Linear Regression, as it achieved comparable results to other methods while having the advantage of a significantly faster runtime.

3.2. Truncated MCTS

Similar to how Alpha-Beta search can be truncated, so can MCTS. Instead of doing a full simulation, we can simulate up to a specified depth, and then use a heuristic from there. AlphaZero essentially performs a truncated search with depth 0 and uses its value network as a heuristic; this project uses a similar approach, but the difference is that this project uses a static evaluation rather than AlphaZero's value network. Previous MCTS algorithms such as AlphaGo did use rollouts alongside the value network, taking a weighted average of the rollout result and the value network result[11], but the disadvantage of using rollouts is that they can often be uninformative. If performed poorly, they may even be very misleading, such as by missing the existence of a hard-to-find winning idea and thereby estimating good positions to be poor. Additionally, the exploration factor in UCT will be determined experimentally.

[5] King safety can often be just as, if not more, informative than material, as the end goal of the game is not to have the most pieces but to checkmate.

[6] Here, piece refers to bishops, knights, rooks, and queens. Players often consider the mobility of these pieces, trying to maximize their own while restricting the movement of the opponent.

[7] Players often consider the position of their pawns to inform their plans, gauge the structural integrity of both sides and estimate who is better in an endgame.

3.3. Alpha-Beta Engine

On top of Alpha-Beta search, this engine also uses move ordering and quiescence search. *Move ordering* estimates the goodness of each legal move from a given position, where higher values indicate that a move is likely better. For instance, lower-valued pieces capturing higher-valued pieces are given bonuses because this generally indicates potential material gain.

Quiescence search extends out the Alpha-Beta search for more depth but only looks at tactical moves, which in this engine, are defined as just the available capture moves. Because it only looks at capture moves, the branching factor of the quiescence search is relatively low. The importance of using a quiescence search is to find "quiet" positions to evaluate, as static evaluations often fail when positions have too many tactics involved. For instance, the end of Alpha-Beta search might end with a queen taking a pawn, but if the next move was a pawn recapturing the queen, then it would have significantly blundered. Another way to consider the significance of quiescence search is that Alpha-Beta by itself is useful for finding strategies, while the addition of Quiescence ensures that the engine is also good at spotting tactics and not making tactical blunders.

3.4. Criteria for Success

To evaluate performance, the primary metric used is ELO, as ELO helps quantify the relative playing strength. To estimate ELO, we can compare this project's engines to ELO-adjusted versions of Stockfish and have them compete. Stockfish supports adjustable strength levels, where it intentionally progressively introduces noise and error into its evaluations and move selections the lower its playing strength is set. It is also useful to examine qualitatively the playing styles of the engines by examining some of its games in detail.

4. Implementation

4.1. Project Link

This project's source code can be found using the following link: https://github.com/josephxu1234/AlphaOne

$$\begin{bmatrix} -65 & 23 & 16 & -15 & -56 & -34 & 2 & 13 \\ 29 & -1 & -20 & -7 & -8 & -4 & -38 & -29 \\ -9 & 24 & 2 & -16 & -20 & 6 & 22 & -22 \\ -17 & -20 & -12 & -27 & -30 & -25 & -14 & -36 \\ -49 & -1 & -27 & -39 & -46 & -44 & -33 & -51 \\ -14 & -14 & -22 & -46 & -44 & -30 & -15 & -27 \\ 1 & 7 & -8 & -64 & -43 & -16 & 9 & 8 \\ -15 & 36 & 12 & -54 & 8 & -28 & 24 & 14 \end{bmatrix} \begin{bmatrix} -74 & -35 & -18 & -18 & -11 & 15 & 4 & -17 \\ -12 & 17 & 14 & 17 & 17 & 38 & 23 & 11 \\ 10 & 17 & 23 & 15 & 20 & 45 & 44 & 13 \\ -8 & 22 & 24 & 27 & 26 & 33 & 26 & 3 \\ -18 & -4 & 21 & 24 & 27 & 23 & 9 & -11 \\ -19 & -3 & 11 & 21 & 23 & 16 & 7 & -9 \\ -27 & -11 & 4 & 13 & 14 & 4 & -5 & -17 \\ -53 & -34 & -21 & -11 & -28 & -14 & -24 & -43 \end{bmatrix}$$

Figure 1: Early/middle-game vs endgame piece square tables for the king.

4.2. Python and Python-Chess

This project was written in Python, primarily because it has support for various useful libraries including Python-Chess. Python-Chess supports all of the required functions to represent a game of chess, such as legal move generation, evaluating the outcome of a game, and keeping track of the rules of the game. Moreover, it uses an efficient representation of boards known as bitboards, in which each piece type of each color player is represented using a single 64-bit integer, where each bit corresponds to 1 square on the board.

4.3. Tapered Evaluation

Material can be calculated using a lookup table, where each piece is assigned a value. Positional advantages and disadvantages are encapsulated using *piece-square tables (PSTs)*[8]. Based on a piece's type and location on the board, the piece-square table can give a bonus or a penalty. Each piece type has 2 associated piece-square tables: 1 for the early/middle stages of the game, and 1 for the end stages of the game. This is necessary, as different pieces have different responsibilities depending on the phase of the game. For instance, Figure 1 represents the difference between the king's responsibilities in the early/middle-game vs the endgame. In the first table, most of the negative values that represent penalties are concentrated in the center, signifying that a king should not be in the center in the early/middle-game, as this is generally unsafe. In the second table, the center contains positive values that represent bonuses instead, signifying that a king should be more

[8]Note: this project used pre-existing PST values from another chess engine, RofChade[6].

actively involved in the endgame, as the king is often crucial in roles such as supporting pawn promotions.

Phase is calculated as follows. Assign each piece type a phase value (excluding the king), where its phase value represents how great of a contribution its loss from the board makes towards approaching an endgame. Create a constant that represents the value when all of the pieces are on the board, and therefore represents the earliest stage of the game.

This project uses the following assignments:

1. (*P*: Pawn) *P_PHASE* = 1/8
2. (*N*: Knight, *B*: Bishop) *N_PHASE* = *B_PHASE* = 1
3. (*R*: Rook) *R_PHASE* = 2
4. (*Q*: Queen) *Q_PHASE* = 4
5. TOTAL_PHASE = 16 * P_PHASE + 4 * N_PHASE + 4 * B_PHASE + 4 * R_PHASE + 2 * Q_PHASE = 26

Initialize a variable $PHASE = TOTAL\_PHASE$. Whenever a piece comes off the board then, the *PHASE* value is lowered by the corresponding amount (e.g. losing a Queen from the board lowers *PHASE* by 4). As endgames are characterized by having a lower number of pieces on the board, then a smaller value for *PHASE* indicates that the game is progressing towards its end. Lastly, in using the *PHASE* to weight the contributions of the piece-square tables, *PHASE* is scaled to between $[0,1]$ by taking $PHASE/TOTAL\_PHASE$.

Altogether, the tapered evaluation for each player is calculated as:

$$\Sigma(\text{Material}) + (\frac{PHASE}{TOTAL\_PHASE})\Sigma(\text{Early/middle-game PST values})$$
$$+ (1 - \frac{PHASE}{TOTAL\_PHASE})\Sigma(\text{Endgame PST values})$$

Therefore, using tapered evaluation as a static evaluation function, we can take White's tapered evaluation − Black's tapered evaluation.

4.4. Introducing More Features Into the Evaluation

To potentially improve the static evaluation's ability to understand the game, the following additional features were explored based on the goals of quantifying king safety, piece play, and pawn play:

1. Mobility[9]: The number of legal moves.
2. King Attack[10]: Define a *king zone* to be the area a king can reach, with 3 more forward squares facing the enemy position. For each piece that can attack a square in the enemy king zone, increment the king attack bonus (+2 for knights/bishops, +3 for rooks, and +5 for queens).
3. Pawn Shield[11]: Increase the pawn shield bonus (+1) for each pawn directly in front of a king and add a smaller bonus (+0.5) for each pawn 2 rows in front of the king.
4. Passed Pawns[12]: Count the number of passed pawns that are not blocked on the way to promotion by enemy pawns.
5. Doubled Pawns[13]: For each pair of pawns on the same file[14], increment a counter.
6. Pawn Islands[15]: Count the number of groups of pawns that are on adjacent files. Breaks in these groups, marked by a file that doesn't have a pawn, lead to the creation of pawn islands.

For each feature, calculate White's value and Black's value, and take the difference to derive a final feature value. In addition to the features listed above, the tapered evaluation was also used. To weight these features, linear regression was used.

The dataset used consisted of 16 million positions of boards represented using FEN[16] notation, and their associated evaluations produced by Stockfish 11[13]. To represent the dataset, the Pandas library in Python was used.

From the original dataset, a large sample of 200000 randomly selected games was used. The

[9] Having more mobility enables more attacking or defending options.

[10] Having more pieces that can target an enemy king leads to a greater chance of checkmating or tactics involving checkmate threats.

[11] Pawn shields keep the king safe, and generally not having a pawn shield exposes the king to attacks and checks, leading to losing tactical skirmishes.

[12] These are often key in endgames, where the goal is generally to promote before your opponent.

[13] Doubled pawns are generally "bad" because they are harder to mobilize, and pawns cannot support each other on the same file. Therefore, they often end up as targets or liabilities.

[14] Files are the columns of the board, whereas ranks are the rows.

[15] Having too many pawn islands can be a weakness, as adjacent pawns are often strong in the endgame.

[16] FEN is a unique string representation of any chess board position.

evaluations were clipped to a range of $[-1500, 1500]$, where evaluation units were in centipawns[17]. Here, positive centipawn evaluations mean White has an advantage, whereas negative centipawn evaluations mean Black has an advantage. Clipping was necessary to remove large outliers and can be done since for practical purposes, an evaluation of ± 1500 is in practice completely dominant, and any further values outside the range do not represent any more useful information.

The dataset was divided with a $75:25$ split between training and test data. Then, the Scikit-Learn LinearRegression model was used with default parameters to learn the appropriate weights of the features.

4.5. MCTS implementation of AlphaOne

Algorithm 1 Selection Algorithm

```
1: function SELECT_NODE
2:     node ← root
3:     while node is not a leaf node do
4:         max_UCT_child ← get child node with highest UCT value
5:         node ← max_UCT_child
6:     end while
7:     return node
8: end function
```

In Algorithm 1, which performs the Selection stage, AlphaOne starts from the root node and greedily selects the highest UCT child until it reaches a leaf node.

Algorithm 2 Expansion Algorithm

```
1: function EXPAND(leaf_node)
2:     if is_game_over() then
3:         return leaf_node
4:     end if
5:     children ← generate children of leaf_node
6:     leaf_node.add_children(children)
7:     return random.choice(children)
8: end function
```

In Algorithm 2, which performs the Expansion stage, AlphaOne uses the leaf node selected in Algorithm 1 to generate its children nodes; it does so by generating the nodes that represent the

[17] In chess, a pawn is generally worth 1 point, so in centipawns, a pawn is worth 100 centipawns

states that occur when taking the legal possible moves from the state corresponding to the current leaf node. This stage also picks a node to simulate in the next stage.

Algorithm 3 Simulation Algorithm

```
1:  function SIMULATE(state, max_depth)
2:      depth ← 0
3:      while depth < max_depth and not state.is_game_over() do
4:          next_move ← random.choice(state.get_legal_moves())
5:          state.push(next_move)
6:          depth ← depth + 1
7:      end while
8:      static_eval ← state_copy.eval()
9:      if static_eval > 0 then
10:         return (chess.WHITE, static_eval)
11:     else if static_eval < 0 then
12:         return (chess.BLACK, static_eval)
13:     else
14:         return (None, 0)
15:     end if
16: end function
```

In Algorithm 3, which performs the Simulation/Rollout stage, AlphaOne pushes random moves onto the state associated with the node provided by the expansion stage. In this truncated version, it cuts off the simulation after max_depth moves, or if the game terminates early. While this max_depth parameter can be adjusted, this project opts to take the approach that AlphaZero takes, which is essentially $max\_depth = 0$, for the reasons mentioned previously. Then, at the end of the simulation, the predicted outcome and static evaluation are returned. This serves to perform a similar function that AlphaZero's value network does.

Lastly, in Algorithm 4, which performs the Backpropagation stage, AlphaOne updates the tree's nodes using the static evaluation. From the simulated node, it returns up to the root by following the parent node pointers. Positive rewards are given to children nodes whose parent nodes match the predicted winner to incentivize these parent nodes to pick those children nodes more in the future. Conversely, negative rewards are given to children nodes whose parent nodes do not match the predicted winner to disincentive these parent nodes to pick these children nodes in the future.

These 4 stages can be run as many times as desired to construct as large of a search tree as desired,

Algorithm 4 Backpropagation Algorithm

```
1:  function BACKPROP(node, turn, winner, static_eval)
2:      if winner = None then
3:          reward ← 0
4:      else
5:          reward ← −absolute_val(static_eval) if winner = turn else absolute_val(static_eval)
6:      end if
7:      while node is not None do
8:          node.N ← node.N + 1
9:          node.Q ← node.Q + reward
10:         node ← node.parent
11:         reward ← −reward
12:     end while
13: end function
```

and then the most explored move can be chosen. In practice, the 4 stages can run as many times as possible under a given time limit. Note that in selecting MCTS's best move, it can be dangerous to use the Selection stage's criteria of highest UCT, as this can be a node that merely appears to have a high UCT but only because the moves following it were simply not explored in depth. Instead, the most explored next move is safer, as the greater amounts of exploration indicate the moves following it are good as well.

4.6. Alpha-Beta Implementation

Algorithm 5 demonstrates the core Alpha-Beta algorithm. If the game terminates before $depth$ reaches 0, then a large value is returned in checkmate positions or 0 for drawn positions. In the base case, once depth reaches 0, then Alpha-Beta continues by returning the value from qsearch, short for quiescence search, in which it performs a similar search as Alpha-Beta, but only looks at capture moves in case there are tactics to consider in the position. It also uses a search depth variable ($qdepth$) which can in practice be set to a larger value than the $depth$ value that Alpha-Beta uses, as qsearch's branching factor is much smaller. This project opts to fix a maximum quiescence search depth of 6. At the end of its search, qsearch passes back a static evaluation for Alpha-Beta to use.

If it has not hit the base case yet, then Alpha-Beta recurses on the next legal states. It iterates through these states in order of which ones are estimated to be better, using a move-ordering function.

Algorithm 5 Alpha-Beta Algorithm

```
function ALPHABETA(depth, board, α, β, color)
    if board.is_checkmate() then
        return −MATE_VALUE if color = chess.WHITE else MATE_VALUE
    else if board.is_game_over() then
        return 0
    end if
    if depth ≤ 0 then
        return qsearch(qdepth, board, α, β, color)
    end if
    if color = chess.WHITE then
        best_val ← −∞
        moves ← get_ordered_moves(board)
        for move in moves do
            board.push(move)
            v ← alphabeta(depth − 1, board, α, β, chess.BLACK)
            best_val ← max(best_val, v)
            α ← max(α, best_val)
            board.pop()
            if β ≤ α then
                break
            end if
        end for
        return best_val
    else
        best_val ← +∞
        moves ← get_ordered_moves(board)
        for move in moves do
            board.push(move)
            v ← alphabeta(depth − 1, board, α, β, chess.WHITE)
            best_val ← min(best_val, v)
            β ← min(β, best_val)
            board.pop()
            if β ≤ α then
                break
            end if
        end for
        return best_val
    end if
end function
```

The move-ordering function incentivizes lower-value pieces to capture higher-value pieces, checks, and improvements in PST values. For each of these recursive calls, it keeps track of the best current value. White also keeps track of its best value using the α parameter, while Black keeps track of its best value using the β parameter. The cutoff point is $\beta \leq \alpha$, at which point pruning occurs.

This Alpha-Beta implementation can be called to find the move with the highest Alpha-Beta value, and this move can then be returned and used.

5. Evaluation

5.1. Static Evaluation

MSE (Mean Squared Error) and MAE (Mean Absolute Error) were used to analyze the static evaluation function, which was created using linear regression. Tapered evaluation by itself was used as a baseline, to assess to what degree the additional features were able to improve the evaluation.

Table 1: Evaluation Results

Method	MAE on Train	MAE on Test	MSE on Train	MSE on Test
Tapered Evaluation	224.507	225.253	140822.180	142318.623
Static Evaluation	224.671	225.935	130406.278	131985.780

As shown in Table 1, while MAE did not change greatly, MSE did improve. Since MSE generally penalizes larger errors to a greater extent, this indicates that the static evaluation with additional features doesn't make as significant of errors as the tapered evaluation by itself. In theory, this also should produce practical benefits for the algorithms themselves, as this improved static evaluation can account for more features of the game, thereby making more strategic considerations.

Table 2 shows results that can reasonably be expected. Pawn Islands and Doubled Pawns receive negative penalty weights as having too many of these compared to the opponent is generally disadvantageous. On the other hand, the other features all receive positive reward weights, as having greater values in these features compared to the opponent is generally advantageous. One

Table 2: Feature Weights

Feature	Weight
Pawn Islands	−33.993
Doubled Pawns	−9.669
Tapered Evaluation	0.822
Pawn Shield	1.461
Mobility	4.237
King Attack	4.947
Passed Pawns	108.254

concerning weight, however, was the Passed Pawns feature, as this value essentially argues that a passed pawn is worth twice as much as a normal pawn, which is usually only worth 100 centipawns. While it is understandable that passed pawns get a large bonus, as these are often what accounts for winning an endgame, it may pose a problem in practice to have such a large bonus value, such as by overvaluing pawns over pieces like rooks.

5.2. Time Complexity of the Static Evaluation

We can show that indeed, using the static evaluation is not computationally costly. To estimate the static evaluation's time complexity, we must analyze the complexities of the feature extractions that account for it. Once these features are derived, aggregating them using a linear combination is a $O(1)$ operation, for a constant number of features.

The time complexities of the features can be estimated as follows:

1. Tapered Evaluation - $O(n^2)$: Iterate through the board to find the pieces, which costs $O(n^2)$. Evaluate each individual piece, which costs $O(1)$ each time.
2. King Attack - $O(1)$: Iterate through the king zone, which is at most 11 squares
3. Mobility - $O(n^2)$: Assuming P pieces on the board, and M legal moves per piece, it costs $O(P \cdot M)$ to generate the legal moves. At worst, there are 4 rows worth of pieces, with each row being length n, so $P = O(n)$. The most mobile piece, the queen, can at best travel 1 rank, 1 file, and 2 diagonals, each of which is at most length n, giving $M = O(n)$.
4. Pawn Shield - $O(1)$: Iterate through the at most 6 available Pawn Shield squares in front of the king.

5. Passed Pawns - $O(n^2)$: Compare every pair of ranks, of which there are $O(n^2)$, to determine whether any pawns serve to block other pawns from promoting[18].
6. Doubled Pawns - $O(n^2)$: Compare every pair of ranks to determine whether any pawns are stacked on the same file.
7. Pawn Islands - $O(n)$: Iterate through all n ranks to determine which files contain no pawns at all.

Clearly, the bottleneck is $O(n^2)$ for this evaluation. In practice, however, n never grows, as chess boards have a constant size of 8×8. This gives an evaluation that is relatively cheap to compute.

5.3. ELO Estimation

The Stockfish library in Python has the option to adjust Stockfish's ELO. This is not a perfect representation as it does not exactly match human performance when adjusted, or even other engine performance necessarily, but generally, as its ELO is lowered, it plays with greater error. Nonetheless, it can serve as a way to help estimate performance, albeit not as precisely as more extensive methods such as full chess engine tournaments.

The following methodology was used. This project's engines would play against ELO-adjusted Stockfish from a pre-set list of common positions, derived from common GM (Grandmaster) openings. Specifically, the top 3 opening moves were chosen from Lichess.org's master's database, and from there, the top GM moves (those that GMs chose most frequently) were repeatedly chosen up until 8 ply were completed, resulting in a reasonable opening for the engines to play[9]. The reason it is important to vary the openings is that if 2 engines were to play deterministically, they would play the same game repeatedly, and repeated trials would therefore not present any new information. Specifically, the following openings were used:

1. Open Sicilian (1. e4 c5 2. Nf3 d6 3. d4 cxd4 4. Nxd4 Nf6)
2. Queen's Indian (1. d4 Nf6 2. c4 e6 3. Nf3 b6 4. g3 Ba6)
3. Reti transposed to English (1. Nf3 Nf6 2. c4 g6 3. Nc3 Bg7 4. e4 d6)

Winning gives a side $+1$ point, drawing gives $+1/2$ point, and losing gives $+0$ points. The engine is pitted against X-ELO Stockfish in 6 rounds (all 3 openings are used, and each side gets to

[18]These comparisons can be done using bitwise operations, using the bitboard representations.

play both Black and White for each opening). If the engine can beat or tie X-ELO stockfish, then X is incremented, and the competition restarts. For efficiency, this is implemented as a binary search, starting with a lower bound of 0 and an upper bound of 3200. Each time this competition is run, either the lower or upper bound is changed (if the engine beats/ties ELO-adjusted Stockfish, the lower bound is raised; otherwise, the upper bound is decreased). This is done a total of 5 times, cutting the range from 3200 to 100.

For AlphaOne, the parameters tested were max depth= 0, time limit= 60 seconds, and exploration factor=[1...10][19]. However, regardless of the exploration factor or static evaluation method, AlphaOne was never able to rival even 100-ELO stockfish. Observations of the games played showed that while AlphaOne was often able to maneuver pieces strategically and even arrive at a balanced middle-game[20], it generally would make at least one tactical blunder, and go on to lose the game.

Table 3: ELO Rating Ranges for Version 1 and Version 2

Method	ELO on Depth 1	ELO on Depth 2	ELO on Depth 3	ELO on Depth 4
Version 1	[0, 100]	[300, 400]	[1500, 1600]	[2000, 2100]
Version 2	[0, 100]	[300, 400]	[1400, 1500]	[1800, 1900]

The Alpha-Beta engine was able to perform with much greater success. In Table 3, Version 1 refers to using the Alpha-Beta engine with just a tapered evaluation, whereas Version 2 refers to using the engine with the static evaluation function that was derived using linear regression. Notably, both were able to achieve fairly high performance[21] on Depth 4[22]. For reference, only approximately 5% of adult tournament players are at 2000+ ELO[1].

[19]Unfortunately, my computer was having difficulty supporting any greater time limits, as the tree data structure was growing too large.

[20]This was observed from both personal experience and by using Chess.com's analysis to objectively analyze MCTS's games.

[21]As an additional sanity check, the Alpha-Beta engine was able to defeat one of my friends who has reached 2000 ELO previously.

[22]Depth 5+ was not explored due to time constraints.

5.4. Tactics Abilities

Additionally, the engines were specifically evaluated on their abilities to solve tactics puzzles. Specifically, a set of 219 mate-in-two puzzles was selected, in which checkmating the opponent in 2 moves was possible, but only with 1 specific sequence of moves[8].

A preliminary sample of 20 random puzzles was selected to test AlphaOne. However, at each exploration factor tested ([1...10]), AlphaOne was only ever able to solve 1 of 20 puzzles each time (60 seconds per move was allotted). As preliminary results showed poor results, no further tests were conducted.

The larger set was used to test Alpha-Beta, with results shown in Table 4.

Table 4: Mate in 2 Puzzles Solved by Alpha-Beta

Depth	Number Solved (% Solved)
1	19 (8.7%)
2	32 (14.6%)
3	219 (100%)
4	219 (100%)

Alpha-Beta showed much greater success and performed perfectly when the puzzles were captured within its depth, as expected.

6. Conclusions and Future Work

While AlphaOne was ultimately unsuccessful, and the Version 2 engine underperformed, this still gives interesting insights into the pros and cons of MCTS versus Alpha-Beta, and the merits and downsides of evaluation approaches.

Firstly, this project demonstrates the main weakness of MCTS, namely that it can underestimate the importance of certain subtrees, which often contain crucial tactics. As shown in the mate-in-two puzzles testing, AlphaOne's abilities were highly lacking compared to Alpha-Beta's. Chess players readily recognize the importance of seeing short-term tactics, as often a single tactical blunder is sufficient to decide the rest of the game; therefore, lower ELO games are frequently characterized

by greater amounts of short-term tactical blunders, whereas high-level games rarely have short-term blunders at all. Therefore, it is understandable that the 2 engines' ELOs parallel their tactical abilities.

Perhaps the most surprising result was that the Version 2 engine actually underperformed the Version 1, even though MSE values from Table 1 show that theoretically, the Version 2 engine should be outperforming. There may be various reasons for this, which may be confirmed with future work. It is possible that the weights were improperly calibrated (for instance, by overvaluing passed pawns, as discussed previously). Additionally, it may be possible that at higher depths, it would outperform Version 1. Observations of several games showed for example, that sometimes the Version 2 engine would attack aggressively, such as by placing its Queen nearer the enemy king (likely in an attempt to increase the King Attack parameter), only to get its material trapped in long sequences that were passed its search depth. Therefore, one hypothesis is that given more search depth, it may be possible for Version 2 to still capitalize on its more extensive strategic understanding, without falling into tactical blunders.

Overall, it appears that the primary weakness to be addressed ought to be MCTS's tactical nearsightedness. For future research, it may be possible to form some different type of hybrid between MCTS and Alpha-Beta such that we can leverage the advantages of both methods: MCTS's ability to search deeply down promising branches, and Alpha-Beta's ability to catch sharp tactics.

References

[1] T. Bardwick, "Csca informant – observations about chess rating distribution and progression – chess academy of denver," Chess Academy of Denver. Available: https://coloradomasterchess.com/informant-ratings-and-expectations/#:~:text=Approximately%2070%25%20of%20rated%20tournament

[2] C. B. Browne *et al.*, "A survey of monte carlo tree search methods," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 4, pp. 1–43, 03 2012.

[3] M. Campbell, A. Hoane, and F.-h. Hsu, "Deep blue," *Artificial Intelligence*, vol. 134, pp. 57–83, 01 2002.

[4] "Elo rating system - chess terms," Chess.com. Available: https://www.chess.com/terms/elo-rating-chess

[5] "Magnus carlsen - top chess players," Chess.com. Available: https://www.chess.com/players/magnus-carlsen#achieving2882

[6] "Pesto's evaluation function - chessprogramming wiki," Chessprogramming.org, 2018. Available: https://www.chessprogramming.org/PeSTO%27s_Evaluation_Function

[7] "Ccrl 40/15 - index," computerchess.org.uk. Available: https://computerchess.org.uk/ccrl/4040/

[8] B. Harvey, "Mate in 2," Wtharvey.com, 2024. Available: https://wtharvey.com/m8n2.txt

[9] "Chess analysis board," lichess.org. Available: https://lichess.org/analysis

[10] C. E. Shannon, "Xxii. programming a computer for playing chess," *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, vol. 41, pp. 256–275, 03 1950. Available: https://vision.unipv.it/IA1/aa2009-2010/ProgrammingaComputerforPlayingChess.pdf

[11] D. Silver *et al.*, "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, pp. 484–489, 01 2016.

[12] D. Silver *et al.*, "Mastering the game of go without human knowledge," *Nature*, vol. 550, pp. 354–359, 10 2017. Available: https://www.nature.com/articles/nature24270

[13] "Chess evaluations," www.kaggle.com. Available: https://www.kaggle.com/datasets/ronakbadhe/chess-evaluations