Building Base Algorithms to Compete with Stockfish*

1st Andrew Dellsite
Dept. of Aerospace Engineering
CU Boulder
SID: 106140453

2nd Adam Gormous
Dept. of Aerospace Engineering
CU Boulder
SID:10604987

3rd Corey Huffman
Dept. of Aerospace Engineering
CU Boulder
SID:106025439

I. Introduction

Chess is one of the worlds oldest and most complex games. Chess has anywhere between 10^{111} and 10^{123} possible moves. The amount of moves possible makes it impossible to develop a single strategy to win every game of chess. However there are many machine learning programs that seek to output the best possible move for any board configuration. Two of the most impressive chess engines that succeed at this are Stockfish and AlphaZero. This paper explores the base machine learning algorithm used for these two chess engines: Monte Carlo Tree Search (MCTS). Similarly, a minimax with alpha-beta pruning algorithm is developed and evaluated. These algorithms are used in a chess game against Stockfish in order to highlight the methodology behind these chess engine algorithms and how they compare to a complex algorithm.

This paper also seeks to accomplish 3 different objectives. The first objective is to simply beat a random move selection policy. This is to demonstrate that the MCTS and minimax algorithms that are implemented are functioning correctly. It is also important to note that the functionality of this function is tested in multiple ways that are elaborated on in the following sections. The second objective is to create an algorithm capable of calculating the best move using a depth of 10 in under 10 minutes. This would demonstrate that our algorithm is capable of making the estimated best move in a reasonable amount of time. The final objective of this paper is to develop a machine learning algorithm that can operate at the same maximum elo as Stock fish (i.e. beating Stockfish when its at its best).

The approach to completing these objectives first requires that a MCTS and minimax simulation are developed. This methodology follows a specific set of steps.For example, per iteration, the MCTS first goes through a selection phase, determining which child node to go explore from the current node. The next step is to expand that child node into every possible move from that node. Once expansion is complete, the next step is a rollout of the game until completion. The final step is a backup where each node accumulates the reward from the terminal state of the rollout. Minimax can be described in a similar manner. These methodologies are described more in depth in the following sections.

The overall scope of this project is to develop the base algorithm used in the best chess engines currently available. However, to beat these chess engines would be incredibly difficult due to their complexity. Therefore, beating Stockfish is an ambitious goal and will be attempted to show the difference between early iterations of chess engines and the more developed ones. However, this is not required to show the effectiveness of using MCTS to develop chess strategies.

II. Background and Related Work

Chess is one of the most widely studied games in the field of artificial intelligence. The most sophisticated chess engines use complex search algorithms, consider a variety of chess formations, and have well designed evaluation functions. Take for example the algorithm that has reached the highest chess elo amongst any human or machine; AlphaZero. As a note, elo describes the relative strength of a player against an average opponent. By this definition AlphaZero is the strongest possible player. AlphaZero is impressive because it only requires the rules of the game and no other domain knowledge. It is able to do this through its use of self-play and neural network reinforcement learning. This methodology is beyond the scope of this paper so it will not be elaborated on. AlphaZero is mentioned to act as a benchmark to Stockfish. Similar to AlphaZero, Stockfish also acts at a superhuman elo, just under AlphaZero. One major difference between the two is that Stockfish performs at a consistent elo across all iteration sizes while AlphaZero improves its elo as the iteration size increases [1].

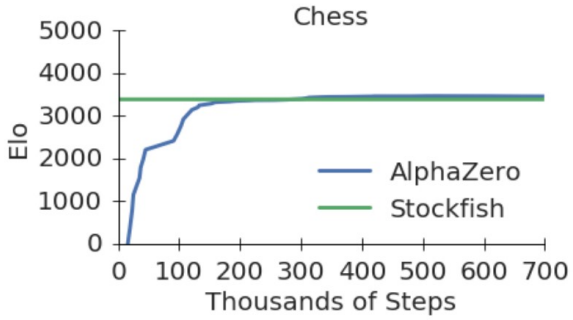

Fig. 1. Comparison between AlphaZero and Stockfish as steps size is increased. [1]

Where AlphaZero uses neural networks to improve its performance, Stockfish uses a database populated by matches from professional chess players. This data is used to evaluate the current configuration of the board and decide on the best move [2]. Stockfish also works in conjunction with MCTS to find the best move. A quick description of the tree format of MCTS can provide clarity as to how the two work together.

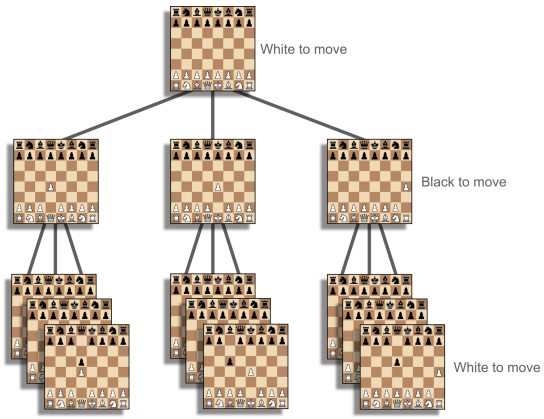

Fig. 2. A chess tree diagram of depth 2.

Figure 2 shows a decision tree with a depth of 2, meaning only after 2 turns that many decisions are possible. In other words, per player turn the number of moves possible increases at an exponential rate. To consider only the next best move would not benefit a player as chess is a game where it is beneficial to think as many moves ahead as possible. For an algorithm to search down all the branches and consider every possible move would require a massive amount of processing time and power. By using a database of professional-level moves, many of the branches that the algorithm would search can be eliminated, drastically reducing the time to converge on the best move. Hence, using the MCTS algorithm without a database can be inefficient and ineffective without some customization.

The minimax with alpha-beta pruning algorithm seeks to also remedy this ineffectiveness by eliminating certain branches that do not contribute further to the outcome. This methodology, as well as MCTS, are described in further detail in the Solution Approach section.

III. Problem Formulation

In order to accurately simulate chess, it must first be formulated as a Markov game. Markov games can be thought of as an extension of game theory to environments that act like a Markov decision process (MDP). This means chess can be modeled similarly to an MDP. For MDPs, an optimal policy is one that maximizes the expected sum of the discounted reward. However, it is difficult to develop an optimal policy for Markov games as the adversary's decisions must also be considered [3]. Hence, chess can be defined similarly to an MDP, but with some caveats. Chess can be defined by its states, actions, transition probabilities, rewards, and discount factor. However, unlike an MDP this space applies to more than one player as the adversary's actions also must be considered.

- $\mathcal{S}$: The current configuration of the board (this applies to both players).
- $\mathcal{A}_1$: All legal moves in the current board configuration for player 1 (also referred to as just player).
- $\mathcal{A}_2$: The set of legal moves for player 2 (also referred to as the opponent), given the board configuration after player 1's move.
- $\mathcal{T}(\mathcal{S}, \mathcal{A}_1, \mathcal{A}_2, \mathcal{S}')$: Non-deterministic transition values.
- $\mathcal{R}(\mathcal{S}, \mathcal{A}_1, \mathcal{A}_2)$: Non-deterministic reward values.
- γ: The discount factor is 1.

While the action and state spaces seem pretty straightforward, the transition and reward spaces need some elaboration. Although to some extent players' decisions in a game of chess can be deterministic, which is how Stockfish and AlphaZero consider their transition space, for the purpose of this paper the opponent's behavior is completely random as per the first objective of this paper. This assumption drastically decreases the effectiveness of the MCTS algorithm due to the size of the action space at each decision node. The reward space for chess is also considered deterministic. But similar to the transition space, by having a non-deterministic reward space the effectiveness of MCTS can vary. In order to improve the algorithm developed in this paper, the rewards space has been customized to reward various chess strategies such as specific starts, terminal state configurations, and chess board configurations. This is elaborated on in the next section when discussing the problem approach.

Now that the problem is formulated, it is critical to describe the goals that this paper aims to accomplish. The first goal is the simplest - the algorithm developed in this paper must be able to beat an opponent who is selecting its next chess move at random given a set of legal chess moves. Although this seems like an easy task, random move selection makes it difficult to find tactics that typically work. This baseline also demonstrates the functionality of our algorithm. Once the functionality of the code is confirmed, it can be used in conjunction with a maximum depth exploration of 10 and a processing time of 10 minutes to demonstrate the developed algorithm is able to play a competitive game of chess. A good side goal here would to be able to beat Stockfish at some discounted elo on a scale of 1-20 as this can assure the performance of the algorithm. The final objective is to beat Stockfish at a skill rating of 20 which is the highest elo achieved by Stockfish. More information about these options can be found in the documentation for the Julia based Stockfish package [4].

IV. Solution Approach

This section is broken up into three different subsections. The first one briefly outlines the packages and code used to analyze and simulate the game between the algorithm developed in this paper and Stockfish. The next subsection describes the methodology behind MCTS and the policy used for move selection during the rollout phase. The final

subsection discusses the use of the minimax algorithm along with alpha-beta pruning.

A. Code Development

The coding language used to create the simulation is Julia because of the readily available reinforcement learning, chess, and Stockfish packages. The first portion of the program that is developed is the setup for a game between a random move selection policy and the developed algorithm. We can also simultaneously develop the Stockfish opponent. Once the simulation is developed, the game is ran between the two different algorithms.

B. MCTS Algorithm

MCTS is simulated for a user defined amount of nodes each time a move needs to be selected by the player algorithm. MCTS goes through 4 different steps per iteration in order to select the best possible move given the current board configuration. These steps have been outlined in previous works [5] and are used in this paper's developed algorithm.

1) **Selection**: For this explanation it is assumed that it is player 1s turn. Also for this explanation, the current node represents the current state of the chessboard. The children nodes all represent any legal moves possible from player 1. The child node that is selected is determined by the upper confidence bound (UCB) which is given by equation 1.

$$UCB = \max_{a \epsilon A(s)} \left(\hat{Q}(s,a) + c\sqrt{\frac{2ln(\overline{n})}{N_a^s}} \right) \quad (1)$$

N_a^s represents the number of times this specific child node has been chosen for expansion. $\overline{n}$ is the number of times the children nodes have been called from the current node. For the first few iterations, the node visitation will be zero so the term inside of the square root will be $-\infty$ which is an imaginary solution. To account for this, the initial UCB for each child node is set to ∞ which encourages the algorithm to explore every node as the maximum UCB will always occur at a unvisited node. c represents the exploration factor. A larger exploration factor encourages the algorithm to explore new move combinations rather than exploiting nodes with high values. $\hat{Q}(s,a)$ is the state-action value function calculated in equation 2.

$$\hat{Q}(s,a) = R(s,a) + \frac{1}{N_a^s} \sum_{s' \epsilon s_a^s} \hat{V}(s') \quad (2)$$

The $\hat{V}(s')$ is the value function that is calculated during the rollout step.

2) **Expansion**: Once the child node is selected, the opponent's move is selected using a transition matrix. Because chess is modeled as non-deterministic in this paper, the opponent's move is decided based on a custom policy that scores the current gameboard. This is described in further detail later in this subsection.
3) **Simulation/Evaluation**: This step is also known as the rollout. The position from the expanded node is evaluated in a tuned heuristic policy, returning a simulated score. Then, each legal move from the previous position is evaluated using the same heuristic, and the move with the highest score for the given side is further explored. This process - Evaluating the board position, finding the move that returns the highest value, and then evaluating the board position following the best move - is repeated for a given depth or until the board reaches a terminal state. The reward for the initial expanded node that started the rollout is based on the immediate board score and the final board score at the end of the rollout.
4) **Backup**: The final step back propagates the value calculated in the previous step throughout all the ancestor nodes in the tree.

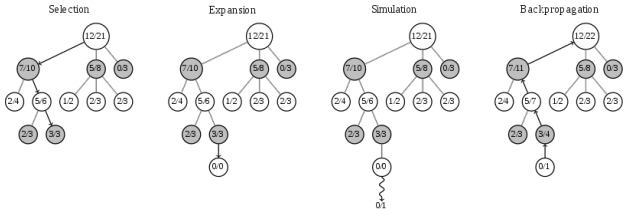

Fig. 3. Diagram demonstrating the process MCTS goes through for each iteration

MCTS is illustrated in figure 3. The process repeats for a given number of iterations or for a chosen amount of time and populates the Monte Carlo tree. At the end of the simulation, the children nodes of the root board position are assessed, and the node with the highest UCB is chosen - representing the optimal move for our algorithm to make.

The policy used in the rollout portion of the MCTS is the crux of optimizing the algorithm to accurately choose a good move, in a timely manner. It is critical to develop a good policy when it comes to implementing MCTS as a chess bot, because of the sheer size of the state space that must be considered. In this algorithm, a heuristic policy was developed to help influence what moves are good and what moves shouldn't be considered. The heuristic policy was developed over numerous simulation games and tuned to account for various board positions. Some of the aspects of a board position that are taken into account in this policy are: The score of piece material, king safety, pawn structure, piece mobility, end game tactics, among others.

C. Minimax: Alpha-Beta Pruning Algorithm

Minimax allows for a simple search through a decision tree by going all the way down to the leaves (defined by maximum depth or terminal states) and using an evaluation function to get a value and then recursively passes the values back up through the tree. As this value is passed up to the parents the layers alternate starting with a maximum node at the root and followed by a minimum layer of nodes and so on. The idea is to find the maximum value for an action a user should

take that also minimizes the value that the opposing user can achieve. This will be evaluated at every new state to get the best action at that given moment.

Due to the large number of possible states a chess board may see, this algorithm may fall into a time and memory issue as it explores every possible outcome. To mitigate this problem, the number of nodes needs to be reduced and only specific branches should be explored. For this decision to be made on which branches to prune, a value for the maximum possible value and a value for the lowest possible value can be tracked as the tree is explored. This is the basis for Alpha-Beta pruning. The alpha will keep track of the highest value within the maximum layers and beta will keep track for the minimum layers. If any leaf within the tree does not provide more value than either alpha or beta (for max and min layers, respectively), then it can be reasonably assumed that the path will not be explored. Fig. 4 shows a step by step process of a minimax tree creation using alpha-beta pruning.

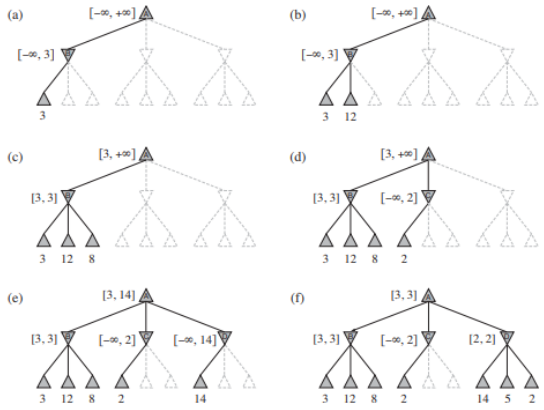

Fig. 4. Figure showing a Minimax tree utilizing Alpha-Beta Pruning [6]

Due to its general simplicity the important piece for a Alpha-Beta tree is the evaluation function used when a leaf node is hit. While Chess is a well understood game and there are many ways to evaluate a board, this paper seeks to create its own evaluation and use it as the basis to test against other players. The first, and most straight forward means of evaluation is simply counting the pieces on the board for either player and comparing the difference. For this implementation, a player receives a 100 for all pawns on the board, a 325 for all knights, a 340 for all bishops, a 500 for all rooks, and a 900 for all kings. The same player will then lose a negative (but equal in magnitude) score for all the opposing pieces on the board. So at the beginning the score for each player, in regards to the pieces, is 0.

While this is a good initial set up, it does not encourage any movement. So other considerations need to be put into place for a more realistic score to be calculated and encourage movement toward capturing the king. This algorithm takes into account four other considerations. One is the placement of the knights. The best position for a knight (in a general sense) is in the middle of the board allowing for more versatility in their movements, and the worst is in the corners. So, a knight provides 7 points to the scored if it is in a better position and -14 if it is not in a good position. The second consideration is taking account of positions that are under attack and unguarded. A count of all pieces that are at risk are count for both sides with a negative score being applied to the moving players score and a positive score for the opponents pieces. This allows the moves that are less risky for the moving player to take precedent over those that may put their piece at risk, while also encouraging movement to attack the opposing player.

The final two considerations center around the king. The first of these takes account for how many opposing pieces are in an attacking position of the king. This will apply a negative score for the number of pieces against the moving player and a positive for the pieces attacking the opposing king. The finally consideration is a terminal state. If the leaf is a terminal state, a score of 10000 with the subtraction of the depth is applied. Since the moving player alternates this is always a positive addition. These considerations are put in place in hopes to encourage protection of the king and to avoid reaching that state for a player.

In order to test the evaluation function with the alpha-beta algorithm, different depths will be used against a random operator to see how the depth affects the results and decision making time. The different depth being used are: 1, 100, 100,000, and 100,000,000. The algorithm will also be tested against different levels of the Stockfish bot [4], while maintaining the same depth for each round. This depth will be 100,000,000. All results will be tested against 200 simulations and averaged.

V. Results

In the MCTS algorithm, it is critical to adjust the exploration rate such that each of the legal-move children nodes are explored enough to conclude which move is the best one to make. The results in 5 show the number of times each child node branch was explored at the end of the MCTS simulation for a single move. The nodes that were explored more, are the ones with higher Q values and are effectively better moves to make.

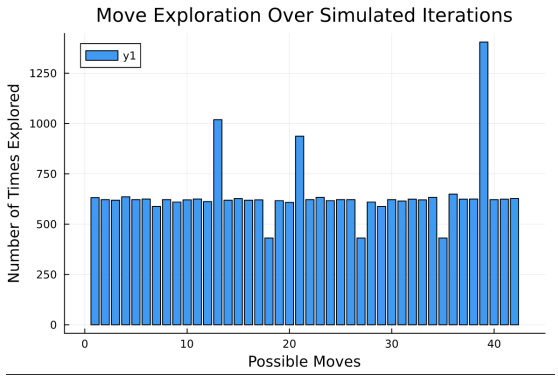

Fig. 5. MCTS level 1 child node exploration results

The success rates of the MCTS algorithm vs the various adversaries are shown in 6. The heuristic policy leveraged in the rollout was tuned and reworked a number of times to achieve a 100 percent success rate vs the random move adversary. The final version of rollout policy was only able to win 1 game, out of 8 full game simulations vs the level 1 Stockfish.

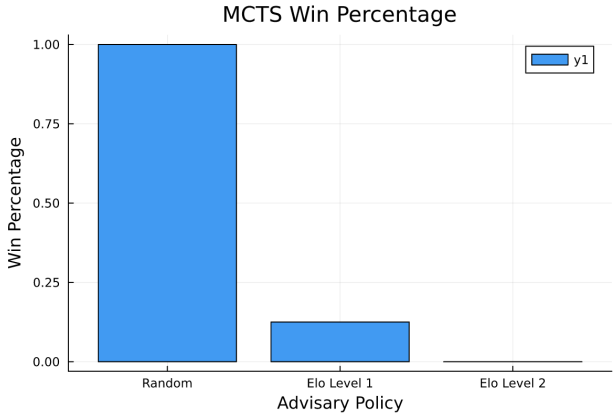

Fig. 6. MCTS win percentage vs adversaries

The Alpha-Beta algorithm did rely heavily on depth, and fine tuning of the evaluation function. With the final implementation, the results of win percentage and decision time versus the random operating bot, can be seen in Table I. Table II shows the win percentage and decision time version different levels of Stockfish.

depth	*Win Percentage*	*Move Decision Time*
1	3.5%	0.0004s
100	48.0%	0.0141s
1e5	57.6%	0.0207s
1e8	60.0%	0.0212s

TABLE I
WIN PERCENTAGE AND MOVE DECISION TIME AGAINST RANDOM PLAYER

Stockfish Level	*Win Percentage*	*Move Decision Time*
0	0%	0.0197s
1	0%	0.0188s
2	0%	0.0192s
3	0%	0.0212s

TABLE II
WIN PERCENTAGE AND MOVE DECISION TIME AGAINST STOCKFISH

VI. CONCLUSION

As presented in the results, the MCTS algorithm developed for this project was limited to accomplishing only the first goal - Beating a random-move adversary. While the MCTS was able to beat the lowest level of Stockfish once, it failed to prove that it could perform consistently at this level. The main limitation of the MCTS algorithm is that it has to explore every possible move combination in order to accurately pick the best move for a given board position. The sheer size of the chess state space makes this task impossible to achieve. The heuristics policy does a decent job of discouraging the exploration of branches representing bad moves, however, the development of a good chess heuristic takes significantly more time and resources than were at our disposal.

The MCTS algorithm could be improved by further developing the heuristic policy used in the rollout section. Additionally, a library could be used to store positions that have been reached before, helping to eliminate the need to explore possible positions that are losing positions. Both of these options help to eliminate the flaw of MCTS by reducing the number of branches that need to be explored to determine a good move.

The results for the Alpha-Beta algorithm show that, with a deep enough tree, it can beat a random player more than half the time. The main take away is the speed at which is was able to calculate a move, and that the difference between a depth of 100,000 and 100,000,000 is not significant. This is also seen in the speed of decision making against the Stockfish bot as well, with a roughly 0.02s decision making time. This algorithm did not beat Stockfish, which is expected as it could not beat a random player consistently. This is most likely a result of a weak evaluation function and not taking into consideration enough of what the board state really says about what the next move should be.

In order to make the algorithm better, a better evaluation function would have to be utilized. In most evaluation algorithms, a neural net is utilized to find the optimal weights for different situations to add to the score of a board. That along with deeper consideration of the state such as castling opportunities, different situations a pawn can end up in, and position of other pieces along with the knight can all be added in to give a more accurate score for making a decision on which move to make.

CONTRIBUTIONS

- Andrew - Wrote MCTS results and conclusion sections. Wrote the MCTS rollout policy, and helped with the MCTS algorithm.
- Adam - Helped write the code base for the MCTS algorithm. Wrote the introduction, background, problem formulation, and the MCTS description in the solution approach.
- Corey - Wrote code for Alpha-Beta Pruning. Wrote Alpha-Beta section, along with results and conclusion pertaining to algorithm.

NOTES

The authors grant permission for this report to be posted publicly.

REFERENCES

[1] Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T., Simonyan, K., and Hassabis, D., "Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm," , Dec. 2017. URL http://arxiv.org/abs/1712.01815, arXiv:1712.01815 [cs].

[2] J. Golz and R. Biesenbach, "Implementation of an autonomous chess playing industrial robot," 2015 16th International Conference on Research and Education in Mechatronics (REM), Bochum, Germany, 2015, pp. 53-56, doi: 10.1109/REM.2015.7380373.

[3] Littman, Michael L. "Markov games as a framework for multi-agent reinforcement learning." Machine learning proceedings 1994. Morgan Kaufmann, 1994. 157-163.

[4] Stockfish: Analyze Chess Games with the 'Stockfish' Engine. https://cran.r-project.org/web/packages/stockfish/stockfish.pdf.

[5] M. C. Fu, "A Tutorial Introduction to Monte Carlo Tree Search," 2020 Winter Simulation Conference (WSC), Orlando, FL, USA, 2020, pp. 1178-1193, doi: 10.1109/WSC48552.2020.9384090.

[6] S. J. Russell, P. Norvig, and E. Davis, "Alpha-Beta Pruning," in Artificial Intelligence: A modern approach, Third Edition., Upper Saddle River, NJ: Pearson Education, 2010, pp. 165–171.