The Application of Minimax Algorithms to Solve Mate in X Chess Puzzles

1st Manuel 'Ricky' Puyana
Aerospace Engineering Sciences
Colorado Boulder University
manuel.puyana@colorado.edu

2nd Kellen Andrew
Aerospace Engineering Sciences
Colorado Boulder University
kellen.andrew@colorado.edu

3rd Julián Lema Bulliard
Aerospace Engineering Sciences
Colorado Boulder University
julian.lemabulliard@colorado.edu

***Abstract*—This paper presents a project that focuses on the implementation of the minimax algorithm to optimize chess endgame and solve chess on a 3x3 board. The project aims to develop an algorithm that can determine the optimal moves for the endgame of chess and the 3x3 board. The minimax algorithm is used to calculate the best move for each player by simulating all possible moves and selecting the one that maximizes the player's chances of winning or minimizes the opponent's chances of winning. The algorithm is designed to consider all possible moves, analyze the potential outcomes, and choose the move that leads to the best possible outcome. The algorithm is tested by implementing it in various endgame scenarios and 3x3 chess games. The results show that the algorithm is highly effective in solving the endgame of chess and finding optimal moves in 3x3 chess games.**

I. Introduction

The concept of using Machine Learning, Computer Vision, Artificial Intelligence, etc., and implementing it into playing Chess is not anything new. In fact, the first completely automated Chess AI was created in 1957 by an IBM engineer. Of course, over time the implementations have gotten more accurate and have gotten to a point that even the best Chess players can never beat the highest-rated Chess AI.

The implementation of an AI in Chess is a task that needs to be thought out very carefully as the amount of moves that must be considered at each point grows exponentially the earlier the AI plays in the game. Furthermore, the depth - or thinking ahead- of the search algorithms for Chess must be limited as it also becomes exponential with each searching node. Most algorithms such as DeepMind's *AlphaZero* were trained using Neural Networks, with no access to endgame or opening tables, for several hours, and then transition to the stored data, as well as a Monte-Carlo Tree Search algorithm to search through 80,000 positions in a second. Many other algorithms such as *Stockfish* use a combination of opening and endgame tables to play 'solved' positions, and then use a Minimax algorithm when in the middle of the game.

II. Previous Work

As mentioned previously, the application of AI in Chess is nothing new. However, the approaches, training schemes, and reward structures that have been developed have evolved over time to try and create more perfect AI through these processes. In this section, however, we will focus on the methods that match up with our work. Specifically on algorithms such as DeepBlue and AlphaZero that use Monte-Carlo Tree Search (MCSTS) Methods.

In general, there are several methodologies when it comes to caring for and training your AI algorithm. For example, programs such as *Stockfish* do not actually need to train to play chess, therefore they are very computationally efficient and can be run on almost any modern computer locally. While programs such as *DeepBlue* and *AlphaGo* were trained on hardware that was designed to train neural networks. Apart from these basic differences, *Stockfish* entirely depends on Alpha-Beta Pruning which is an extension of Minimax algorithms. Because of this design choice, *Stockfish* does not utilize MCTS in the early and late stages of the game due to the structural complexity that arises from these points in the game, instead it relies on pre-establish and known databases for openings and endgames.

AlphaZero on the contrary, was trained from scratch with nothing more than the basic rules of the game. It was trained entirely on self-play. To choose the moves *AlphaZero* used a Neural Network that would take the current state of the game as an input and produce a policy distribution as well as a value estimate as outputs. The policy distribution tells us the probability of choosing each possible move, and the value estimate tells us the value of how the game will be if we take that move. This process of self-play is done millions of times, and in the case of *AlphaZero* in a matter of less than a day of training, can play better than any current Chess Grandmasters.

Apart from this baseline knowledge, there is a never-ending supply of academic papers on AI implementation in Chess as it is the most popular application and test bed for new algorithms. We will survey some of these papers below.

A. Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm [6]

This paper outlines a very high-level overview of how the *AlphaZero* algorithm is structured. It explains how the algorithm takes in a board position and then returns a probability distribution for each move and a value of how the game will end if that move is chosen.

It further explains how the algorithm is entirely capable of achieving superhuman playing ability by playing against itself for millions of games. To do this, *AlphaZero* tries to optimally predict the best move for both players. As time goes on the algorithm learns to recognize patterns in the game and strategies against another optimal player. Of course, the algorithm still uses MCTS which can be the most useful in endgames as most endgames are solved.

The last and most impressive point presented in the paper is that *AlphaZero* was trained on games such as Shogi and Go and with only base knowledge of basic rules was able to play above the best players' levels.

B. Deep Blue [7]

"Deep Blue" is a 2002 paper that describes the development and implementation of the *Deep Blue* AI that famously beat Gary Kasparov in 1997. A general and simple overview of the hardware and software structure of the AI was laid out. The important details emerge when we realize that *Deep Blue* was completely built around being computationally efficient for Chess. Due to this, *Deep Blue* was able to evaluate up to 200 million positions per second, and use an MCTS up to 40 layers deep each second.

Due to the amount of positions that were evaluated, the computer was created in three layers. The first was a set of chips that were computed initial positions, these positions were then passed onto 'worker' computer chips that evaluated top-level moves, and lastly, these were passed onto more dedicated 'chess chips' which went more levels deep into the MCTS.

Lastly, the paper dives into the games between *Deep Blue* and Gary Kasparov and the tactical mistakes and advantages that each player had at several stages in the game.

C. Chess AI: Competing Paradigms for Machine Intelligence [8]

The following paper, written by the creators of *Deep Blue* dives into the more complex side of Chess, the endgame. Since most endgames, as stated previously, have an optimal solution, or a deterministic ending, being able to memorize or realize similar structures between positions becomes paramount.

The paper studies "Plaskett's Puzzle" one of the most famous endgames of all time, and studies how *LCZero* (a crowd-sourced version of *AlphaZero*) performs versus *StockFish*. The puzzle highlights the strengths and weaknesses of each algorithm due to the first move of the puzzle not providing the highest evaluation at a relatively low MCTS search depth. Due to the nature of each program, *LCZero* is unmatched in its evaluation algorithm but fails to find the optimal move due to its use of Alpha-Beta pruning which removes the optimal move after several layers of depth. While *Stockfish* is unmatched in its search algorithm as it explores more moves, even unfavorable ones, that proved to be the solution.

The paper then gives each of the algorithms the optimal first move in the puzzle. With this, *LCZero* was able to find the winning line while searching only 5.5 million nodes, while *Stockfish* had to search 500 million nodes to arrive at the same conclusion.

In conclusion, the paper highlights the biggest problem facing AI advancements: each algorithm has its strengths and weaknesses. Furthermore, it becomes increasingly more complex to allow algorithms to deviate from their nominal behavior for edge cases as they are an unfathomable amount to classify.

D. A0C: Alpha Zero in Continuous Action Space [10]

Decision making processes with large actions spaces, like chess, can be very challenging to approach. However, tackling problems with a continuous action space (robotics, navigation, self-driving cars, etc.) require a significantly different strategy. Historically, *AlphaZero* has been shown to excel in games with discrete action spaces but much less is known about its performance over continuous action spaces. "AOC: Alpha Zero in Continuous Action Space" presents the modifications necessary to implementing *AlphaZero* over a continuous action space and the challenges associated with this task.

The authors used a strategy that can be summarized by two main parts:

1) Obtaining a data set using MCTS
2) Training a neural network with this data set

A benefit to using MCTS is that it can be used on both discrete and continuous action spaces. However, MCTS requires a slight modification to work effectively over continuous action spaces. The authors implemented a strategy known as *progressive widening*. Since we cannot search over the entire action space at each state, *progressive widening* prioritizes revisiting actions that have good returns but progressively widens its search to actions "around" this action.

The data set produced by the MCTS is then transformed into a continuous target density and used to train the neural network. The neural network utilizes a standard network loss that compares the result to the target distribution. Additionally, the authors implement entropy regularization to prevent the search from collapsing on a single branch and avoiding any exploration.

The paper goes on to discuss the results of this *AlphaZero* implementation on the pendulum swing-up problem. The experiment found that using trees with 10 traces and training after each episode allowed the program to search over a broad enough space while still training at a suitable rate.

Overall, this study showed promising results for expanding the range of problem that can be solved with *AlphaZero*.

E. DeepChess: End-to-End Deep Neural Network for Automatic Learning in Chess [9]

The last paper reviewed outlines the process of creating the creation of *DeepChess* which is a Deep Neural Network (DNN) that learns to play chess from scratch. The authors divided the creation into three parts. First, train a neural network to judge a position and create an evaluator. Second, is giving the neural network a position and two possible moves and choosing the most favorable. And lastly, compressing the neural network to allow rapid calculation.

To train the network to evaluate a given position from scratch, not even knowing the rules of chess, the DNN was given a position in a database and then given the outcome of the game. The DNN would then find patterns in passed pawns, King position, etc. to learn over time.

Next, the DNN is given two positions, one in which White went on to win and one in which Black won. This approach gives a good approximation to arrive at similarly structured positions as the DNN has seen in the future. The approach also allows for a larger training set. The paper mentions if that the DNN was given 1 million positions in which White won, and a million in which White lost, they can create a 2×10^{12} training pair data set. The researchers also came to the conclusion that supplying the DNN with games that ended in a draw was not beneficial.

Lastly, the DNN was compressed to allow rapid computation. In testing *DeepChess* was evaluated against *Falcon* which is a chess AI that has competed in the World Computer Chess Championships and most notably won second prize. In the following matches between both AI, *DeepChess* was able to play on par, or better, than *Falcon* even with *Falcon* having a set of more than 100 hyperparameters that have been carefully hand-tuned, and recall the *DeepChess* was trained with no prior knowledge of Chess rules.

The paper, in the end, develops the first successful end-to-end application of machine learning to a computer chess algorithm. The resulting *DeepChess* algorithm plays very aggressively and sacrifices pieces for long-term tactical gains, because of this, the program results in a playing style that resembles grandmasters due to the training structure of the program.

III. Problem Formulation

Chess is a turn-taking game with two players, White and Black. The aim of the game is to take turns moving pieces until one player checkmates another. An overview of the rules of chess can be found on the ”Rules of chess” Wikipedia [5].

The state space of chess is the set of all possible board states (10^{40} different states). The action space is the set of all possible legal moves that a player currently has available. This makes the action space dynamic. Transitions in chess are deterministic, as a player may move any piece directly to the square they wish (so long as the move is legal). The reward function utilized in the implemented minimax tree is discussed in section VI.

Due to computational power, complexity, and time restrictions, the scope of this project will be initially limited to ’solved’ positions. These positions are known as ’Mate in X’ moves where even if both players play perfectly, there is a determined way where White or Black will win. Apart from this, 3x3 chess will initially be focused on as all positions are solved. In addition 3x3 chess generally has a smaller state and action space than conventional chess.

To tackle this a minimax algorithm with alpha-pruning was constructed and implemented. The theory and implementation of this algorithm is outlined in section VI.

IV. Chess Environment

The Python chess library was utilized to create the 3x3 chess environment [1]. The python chess library has functionality

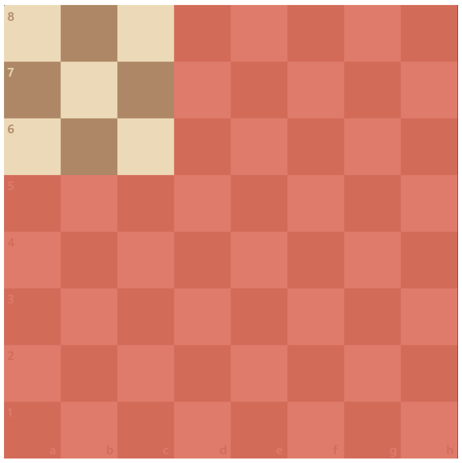
Fig. 1: Upper nine squares of chess board to use for 3x3 board

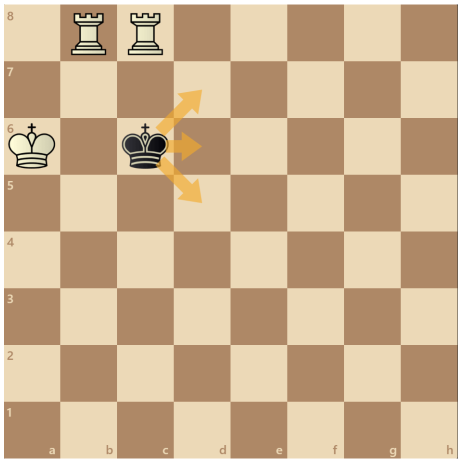
Fig. 2: Before 3x3 constraints, Checkmate Example

for setting up a full chess board, checking for legal moves, making moves, capturing pieces, promoting pieces, checking for checks and checkmates, and checking for draws; it contains all the functionality necessary to play out a game of chess. To create the 3x3 chess board. The following Python chess chess board methods were overloaded to abide by the rules of 3x3 chess in the upper-left nine squares of the board,

- board.legal_moves()
  - Give all the legal moves for the current player
- board.is_game_over()
  - Check if the current board position is checkmate or a draw
- board.is_checkmate()
  - Check is checkmate is present on the current board

For example, if Python chess is used without the additional 3x3 constraints imposed, the position in Fig. 2 (Black to move), the king has three legal moves, so the position is not checkmate. In Fig. 3, the 3x3 constraints are imposed. After constraints, the overloaded methods show that Black has no legal moves, the game is over, and Black is checkmated.

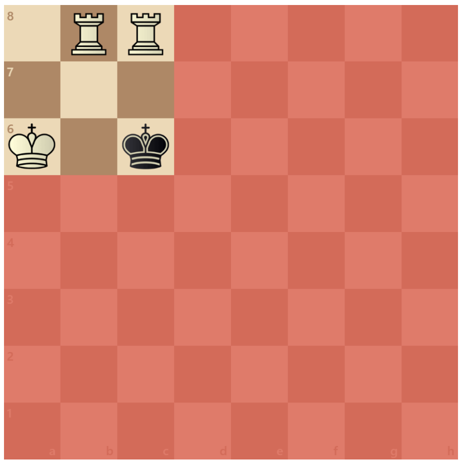
Fig. 3: After 3x3 constraints, Checkmate Example

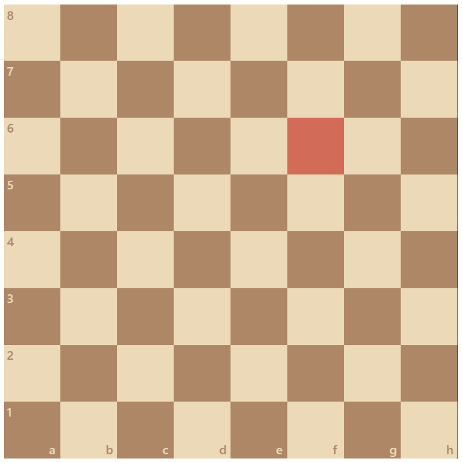
Fig. 4: f6 square

The Python chess library was also used as-is for the implementation of the minimax algorithm onto a full chess board.

V. Chess Notation

In chess, all squares are given a two-letter identifier. This identifier is generated based on a grid where the ranks of the board (rows) are given numbers 1-8 starting in the bottom left, and the files (columns) are given letters a-h starting in the bottom left. For example, the square highlighted in Fig. 4 is f6.

For the notating of chess moves, the from-to notation will be used. In this notation, a move is described by 4 characters, the first two signifying the starting square of the piece and the second two signifying the ending square of the piece. For example, in Fig. 5 White has 4 legal moves on the 3x3 board (see section IV for 3x3 board definition),

- Rook Moves: b6b7, b6b8, b6a6
- King Moves: c6c7

One exception to the to-from notation is when promoting Pawns. When a Pawn reaches the 8th rank, it must be promoted

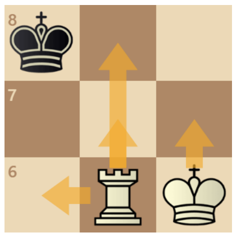
Fig. 5: White legal moves

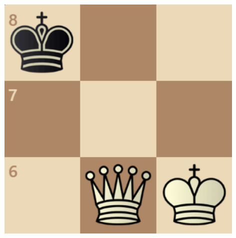
Fig. 6: Mate in 1, White to move

to a Knight, Bishop, Rook, or Queen. When this is done, a k, b, r, or q is added at the end of the 4 letter move identifier to signify which piece the pawn was promoted to (ex. a7a8q is a promotion of a Pawn to a Queen).

VI. Minimax Implementation to Mate in X Puzzles

To solve Mate in X puzzles with minimax and alpha-beta pruning, minimax trees must be expanded and backed up until a checkmate position is found at a leaf node. At this node, a reward of 1 is given if White wins, and -1 if Black wins. The sequence of moves from the current position that results in the checkmate is then taken as the solution to the puzzle. This assumes that the minimax tree is given enough depth to find the checkmate; if the puzzle is a Mate in 4 White to move, the minimax tree must be given a depth of at least 7 (4 moves for White and 3 moves for Black). When creating an AI to solve Mate in X puzzles, it is easiest to give the minimax tree exactly the required depth needed to solve the puzzle. This is acceptable as the puzzles are generally presented with the number of moves to checkmate displayed; however, when utilizing an AI to discern if the current position is a definitive ”Mate in X” moves or not, the number of moves to checkmate is not known. In this situation, depending how the tree is explored, the AI may mistakenly say that the position is a longer checkmate than it really is (ex. Mate in 2 when it is Mate in 1). For example, in Fig. 6 the move b6a6 results in a Mate in 2, where black is forced into a8b8, and mate is found with a6b7; however, if the minimax tree first finds b6b7 it will realize that the position is a Mate in 1.

To ensure that the minimax tree always find the quickest mate, the reward structure will be altered to abide by the following function,

$$r = (d - d_c + 1) \text{ if White checkmates Black} \quad (1)$$

$$r = -(d - d_c - 1) \text{ if Black checkmates White} \quad (2)$$

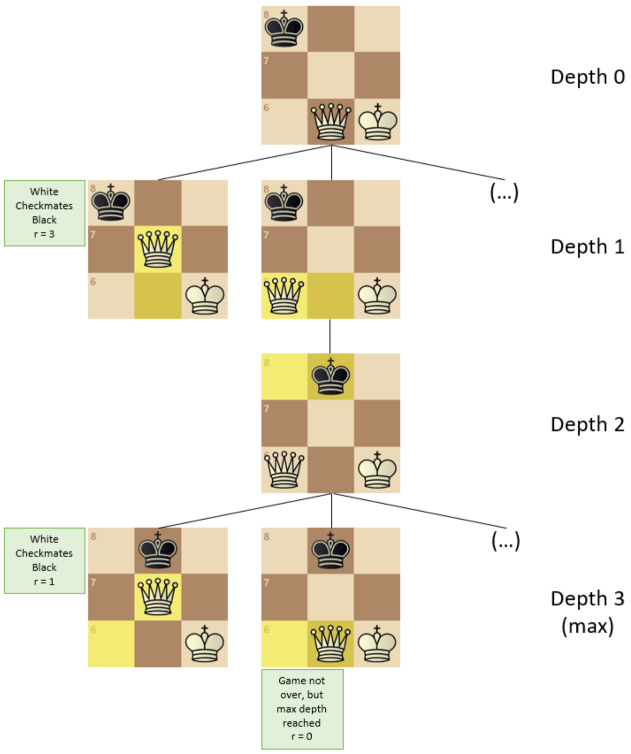
Depth 0
White Checkmates Black r = 3
(...)
Depth 1
Depth 2
White Checkmates Black r = 1
(...)
Depth 3 (max)
Game not over, but max depth reached r = 0

Fig. 7: Mate in 1 minimax tree

Where d is the maximum depth of the minimax tree, and d_c is the depth of the current node of the minimax tree. This will incentivize quicker checkmates as if the minimax tree has a max depth of 3, a Mate in 2 will receive a reward of 1, and a Mate in 1 will receive a reward of 3. This is illustrated in Fig. 7.

VII. Minimax Action Expansion in Chess

In chess, the moves available to a player (actions) change with each turn. When expanding the actions available to a player in the minimax algorithm, only legal moves must be expanded. This adds an extra layer of complexity to the algorithm, as the legal moves available to the player must be checked every time a node is expanded. For example, in Fig 5, the legal moves expanded for the current state and player (White) would be b6b7, b6b8, b6a6, c6c7.

VIII. Move Sorting in Chess For Smart Alpha-Beta Pruning

When using minimax with alpha-beta pruning, the faster the minimax tree finds good actions, the faster it can start pruning branches of the tree. This increases algorithm speed. When evaluating chess positions, actions can be expanded in an order such that the minimax tree heuristically searches certain actions before others. In the minimax algorithm built, moves that result in a check on the enemy king are considered first, then actions that result in the capture of an enemy piece, and then other moves.

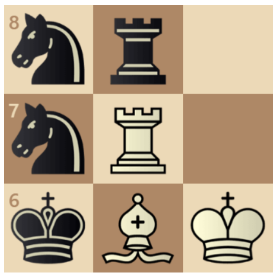

Fig. 8: Mate in 4, Initial Position

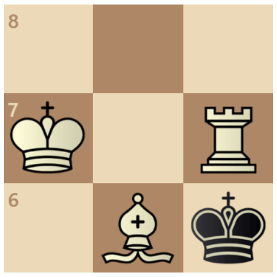

Fig. 9: Mate in 4, Final Position

IX. Evaluation

A. 3x3 Board Evaluation

To evaluate the performance of the AI, 3x3 positions with known Mate in X solutions were pulled from kirr.homeunix.org/3x3-chess/ [2] and tested. Fig. 8 and Fig. 10 show the starting positions of a 3x3 Mate in 4 and a 3x3 Mate in 10.

The AI output for the Mate in 4 is given in Algorithm 1. This solution matches the solution given in [2], and was generated almost instantly.

Algorithm 1 Fig. 9 Mate in 4 Solution

```
White Best move: b7b8, Score: Mate in 4
Black Best move: a8c7, Score: Mate in 3
White Best move: a6a7, Score: Mate in 3
Black Best move: c7a8, Score: Mate in 2
White Best move: b8c8, Score: Mate in 2
Black Best move: a8c7, Score: Mate in 1
White Best move: c8c7, Score: Mate in 1
```

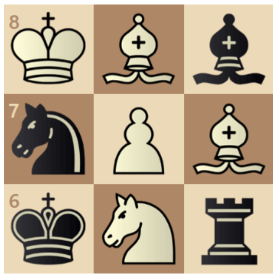

Fig. 10: Mate in 10, Initial position

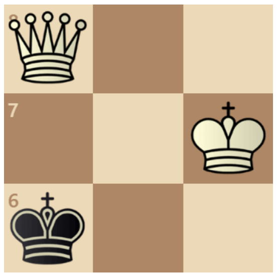

Fig. 11: Mate in 10, Final position

The AI output for the Mate in 10 is given in Algorithm 2. This solution matches a solution given in [2] (there are multiple), and was generated in 22 seconds.

Algorithm 2 Fig. 10 Mate in 10 Solution

```
White Best move: b6c8, Score: Mate in 10
Black Best move: c6b6, Score: Mate in 9
White Best move: c7b6, Score: Mate in 9
Black Best move: a7c6, Score: Mate in 8
White Best move: b6a7, Score: Mate in 8
Black Best move: c6b8, Score: Mate in 7
White Best move: a7b6, Score: Mate in 7
Black Best move: b8c6, Score: Mate in 6
White Best move: c8a7, Score: Mate in 6
Black Best move: c6b8, Score: Mate in 5
White Best move: a8b8, Score: Mate in 5
Black Best move: a6b6, Score: Mate in 4
White Best move: b8c8, Score: Mate in 4
Black Best move: b6a6, Score: Mate in 3
White Best move: c8c7, Score: Mate in 3
Black Best move: a6a7, Score: Mate in 2
White Best move: b7b8q, Score: Mate in 2
Black Best move: a7a6, Score: Mate in 1
White Best move: b8a8, Score: Mate in 1
```

B. Full Board Evaluation

The AI was also evaluated with full chessboard positions. The Mate in X puzzles on full boards were pulled from various sources, and the solutions for them were confirmed with Stockfish [3]. Fig. 12 and Fig. 14 show the starting positions of a Mate in 3 and a Mate in 5, and the final checkmate position for each.

The AI output for the Mate in 3 is given in Algorithm 3. This solution matches a solution given by Stockfish (there are multiple), and was generated almost instantly.

Algorithm 3 Fig. 12 Mate in 3 Solution

```
White Best move: h1a8, Score: Mate in 3
Black Best move: f7g8, Score: Mate in 2
White Best move: a8a1, Score: Mate in 2
Black Best move: g1g7, Score: Mate in 1
White Best move: a1g7, Score: Mate in 1
```

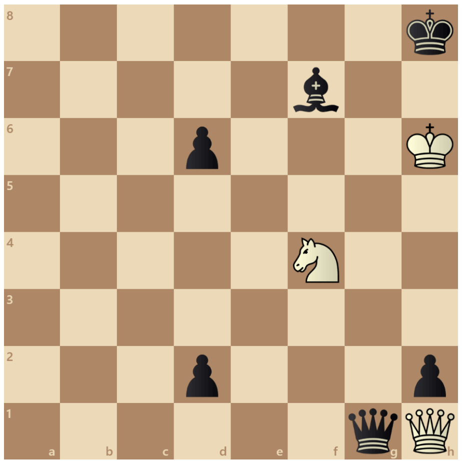

Fig. 12: Mate in 3, Initial position

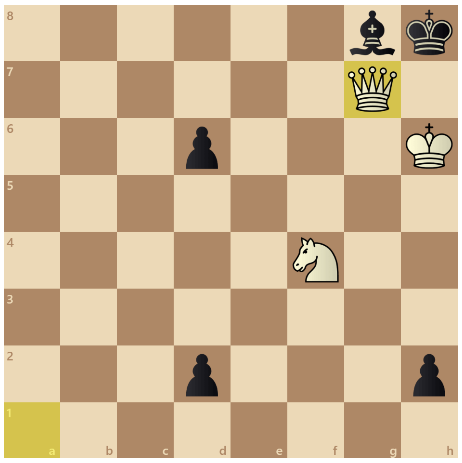

Fig. 13: Mate in 3, Final position

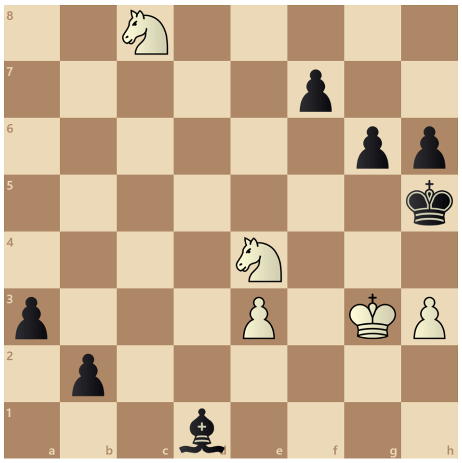

Fig. 14: Mate in 5, Initial position

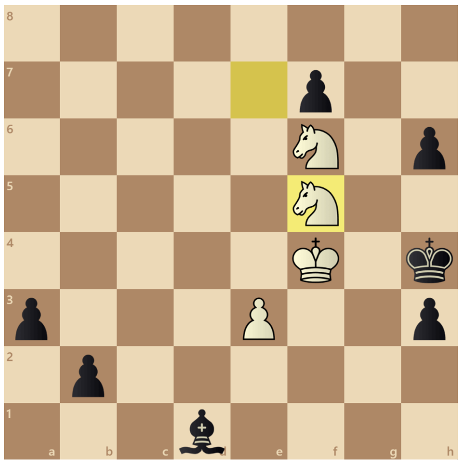
Fig. 15: Mate in 5, Final position

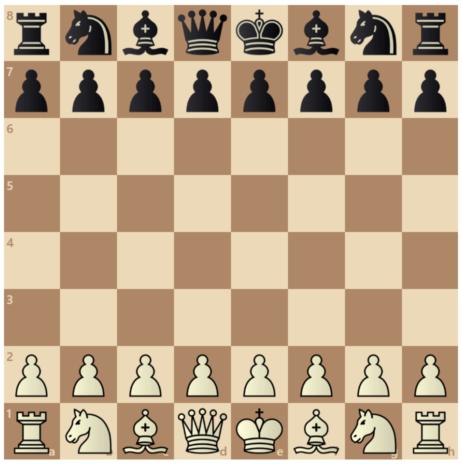
Fig. 16: Conventional starting position

The AI output for the Mate in 5 is given in Algorithm 4. This solution matches a solution given by Stockfish (there are multiple), and was generated in 96 seconds.

Algorithm 4 Fig. 14 Mate in 5 Solution

```
White Best move: h3h4, Score: Mate in 5
Black Best move: g6g5, Score: Mate in 4
White Best move: c8e7, Score: Mate in 4
Black Best move: g5h4, Score: Mate in 3
White Best move: g3f4, Score: Mate in 3
Black Best move: h4h3, Score: Mate in 2
White Best move: e4f6, Score: Mate in 2
Black Best move: h5h4, Score: Mate in 1
White Best move: e7f5, Score: Mate in 1
```

C. *Curse of Depth and Dimensionaltiy*

It has been shown the AI developed has the capacity to solve Mate in X puzzles on both 3x3 and full boards; however, the time to solve these puzzles increases exponentially with the length of the solution and the number of pieces on the board (especially for the full board). If a 3x3 Mate in >10 is given, or a full board Mate in >5 (or a mate with more than a handful of pieces for each side) is given, the solution time is prohibitively lengthy and the minimax tree can be prohibitively large to compute. This is a pitfall of minimax algorithms in that they cannot be sparsely sampled, all actions must be indexed. This is solved partially by alpha-beta pruning, but not completely. Larger computing power or the implementation of domain-specific pruning rules have the capacity to increase the minimax capability.

X. General Play

Though the AI developed in this paper was created to solve Mate in X puzzles, the algorithm can be employed with a small modification for general chess play. If an additional reward function is utilized to evaluate board positions at leaf nodes when checkmate is not found in the current depth, the AI can intelligently select moves. A natural choice for a board evaluation reward function is piece tables [4]. Piece tables assign values to pieces on specific squares. These tables can be learned through neural-network approximation, or heuristically built. For example, the heuristic piece tables given in [4] for White can be found in Appendix A. The tables for Black are flipped up-down and negated. To utilize these tables, all pieces are assigned the value corresponding to the square they occupy on their piece table, and the total value of all pieces is taken as the board evaluation (it is important to scale the rewards defined in Eq. 1 and Eq. 2 such that any checkmate will trump any other board evaluation in a minimax tree).

With the piece tables employed from [4], a conventional starting position, as given Fig. 16, develops as given in Algorithm 5 for 10 moves (5 for each player), to Fig. 17, as the AI plays itself with a depth of 5. Good and bad can be seen from the AI's development. The first couple of moves for each player activate the knights; this is generally a good practice. From there we see that the piece tables result in frequent, repetitive, knight movements and sacrificing of valuable pieces for pawns; this is generally a bad practice. An evaluation of the final position with Stockfish shows that the board is almost even; however, this makes sense as the AI is playing itself. An intermediate-level chess player could win with ease. Betterment of the board evaluation function used on non-checkmate board states is the next step in creating a competent chess AI for general play.

XI. Conclusion

This research project successfully developed a chess AI that utilized minimax and alpha-beta pruning to solve Mate in X chess puzzles on both 3x3 boards and full boards. Additionally, heuristic piece tables were employed to allow the AI to make decisions during general play.

Moving forward, the future of work in this area involves enhancing the AI's ability to solve large puzzles by creating heuristic pruning rules for the minimax tree and learning functions that enable early pruning in the minimax tree.

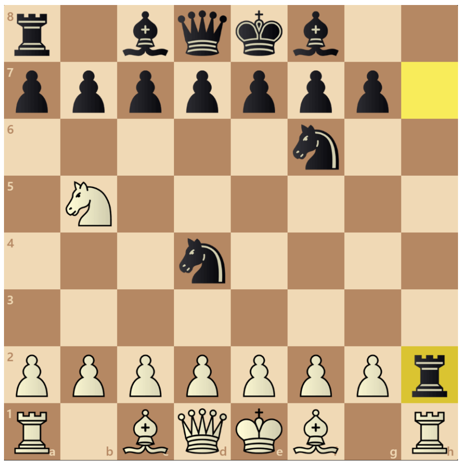

Fig. 17: General Play Position after 5 moves each

Algorithm 5 General Play, Depth=5

```
White Best move: g1f3, Score: 10
Black Best move: g8f6, Score: -5
White Best move: f3g5, Score: 30
Black Best move: b8c6, Score: -5
White Best move: g5h7, Score: 5
Black Best move: h8h7, Score: -15
White Best move: b1c3, Score: -5
Black Best move: c6d4, Score: -35
White Best move: c3b5, Score: -10
Black Best move: h7h2, Score: -30
```

In addition, neural network learning and approximation of dynamic piece tables, as well as incorporating other heuristic knowledge into the board evaluation, can improve overall performance for general play.

The work presented in this paper provides a strong backbone and template for continued advancements in the development of chess AIs capable of solving complex puzzles and making intelligent decisions during play.

XII. Housekeeping

The following contributions were made by each author,

- Ricky: Python Code, Chess Environment, Notation, and Conclusion
- Julián: Previous Work, Abstract, Problem Formulation, and Introduction
- Kellen: Previous Work, Abstract, Problem Formulation, and Introduction

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

References

[1] ”Python Chess,” https://python-chess.readthedocs.io/en/latest/
[2] “3x3 Chess,” kirr.homeunix.org/3x3-chess/
[3] “Stockfish,” https://stockfishchess.org/
[4] “Piece Tables,” https://www.chessprogramming.org/
[5] ”Rules of Chess,” https://en.wikipedia.org/wiki/Rules_of_chess
[6] ”Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm” Silver D., Hubert T., Schrittwieser J.
[7] ”Deep Blue” Campbell M., Joseph Hoane Jr. A., Hsu F.
[8] ”Chess AI: Competing Paradigms for Machine Intelligence” Maharaj S., Polson N., Turk A.
[9] ”DeepChess: End-to-End Deep Neural Network for Automatic Learning in Chess” David O., Netanyahu N., Wolf L.
[10] ”A0C: Alpha Zero in Continuous Action Space” Moerland T., Broekens J. , Plaat A., Jonker C.

Appendix

A. White Piece Tables

Pawn:

```
00, 00, 00, 00, 00, 00, 00, 00
50, 50, 50, 50, 50, 50, 50, 50
10, 10, 20, 30, 30, 20, 10, 10
05, 05, 10, 25, 25, 10, 05, 05
00, 00, 00, 20, 20, 00, 00, 00
05,-05,-10, 00, 00,-10,-05, 05
05, 10, 10,-20,-20, 10, 10, 05
00, 00, 00, 00, 00, 00, 00, 00
```

Knight:

```
-50,-40,-30,-30,-30,-30,-40,-50
-40,-20, 00, 00, 00, 00,-20,-40
-30, 00, 10, 15, 15, 10, 00,-30
-30, 05, 15, 20, 20, 15, 05,-30
-30, 00, 15, 20, 20, 15, 00,-30
-30, 05, 10, 15, 15, 10, 05,-30
-40,-20, 00, 05, 05, 00,-20,-40
-50,-40,-30,-30,-30,-30,-40,-50
```

Bishop:

```
-20,-10,-10,-10,-10,-10,-10,-20
-10, 00, 00, 00, 00, 00, 00,-10
-10, 00, 05, 10, 10, 05, 00,-10
-10, 05, 05, 10, 10, 05, 05,-10
-10, 00, 10, 10, 10, 10, 00,-10
-10, 10, 10, 10, 10, 10, 10,-10
-10, 05, 00, 00, 00, 00, 05,-10
-20,-10,-10,-10,-10,-10,-10,-20
```

Rook:

```
00, 00, 00, 00, 00, 00, 00, 00
05, 10, 10, 10, 10, 10, 10, 05
-05, 00, 00, 00, 00, 00, 00,-05
-05, 00, 00, 00, 00, 00, 00,-05
-05, 00, 00, 00, 00, 00, 00,-05
-05, 00, 00, 00, 00, 00, 00,-05
-05, 00, 00, 00, 00, 00, 00,-05
00, 00, 00, 05, 05, 00, 00, 00
```

Queen:

```
-20,-10,-10,-05,-05,-10,-10,-20
-10, 00, 00, 00, 00, 00, 00,-10
-10, 00, 05, 05, 05, 05, 00,-10
-05, 00, 05, 05, 05, 05, 00,-05
00, 00, 05, 05, 05, 05, 00,-05
-10, 05, 05, 05, 05, 05, 00,-10
```

```
-10, 00, 05, 00, 00, 00, 00,-10
-20,-10,-10,-05,-05,-10,-10,-20
```

King:

```
-50,-40,-30,-20,-20,-30,-40,-50
-30,-20,-10, 00, 00,-10,-20,-30
-30,-10, 20, 30, 30, 20,-10,-30
-30,-10, 30, 40, 40, 30,-10,-30
-30,-10, 30, 40, 40, 30,-10,-30
-30,-10, 20, 30, 30, 20,-10,-30
-30,-30, 00, 00, 00, 00,-30,-30
-50,-30,-30,-30,-30,-30,-30,-50
```