Chess AI Final Report

DS 3500 Final Project

By: Adrian Monaghan, Maxwell Arnold, Nicholas Gjuraj, and Brandon Onyejekwe

1. **Introduction**:

The goal of our project was to build an artificial intelligence that can play chess against a user. In order to achieve this objective, we built a playable chess game that understands almost all facets of chess and includes a graphical user interface. We then had to create an AI that could make optimal moves based on a search into the future states of the chess board, and our board evaluation scoring criteria. We decided to pursue this project because it is a challenging problem that will allow us to learn new skills in python as well as apply everything we learned in class this semester, such as object oriented design and functional programming. The project is a great opportunity to practice problem solving skills and apply relevant algorithms. In addition, the project of creating a chess AI will impress recruiters and graduate admissions committees alike. Although there are not many applications of this program outside of chess, the AI can be used to help one practice and hone their chess skills.

2. **Technical Approach**:

The chess game is built upon a heavy reliance on object-oriented programming. Each type of chess piece has a class inheriting from a general piece class. The piece objects know their position, color, worth score and other data. In addition, pieces can also move to spaces that are in the piece's list of legal moves depending on that piece's specified criteria. The board class keeps track of a two dimensional array of piece objects on the board. In addition, the board knows whose turn it is, when the game is over, and how many turns have occurred among other data. The board class is able to start a game by placing the pieces in their correct location on the chess board. The board class is also able to move a piece, check if any pieces are in the way of another, verify if a king is in check, and check many other states and update and retrieve different aspects of the board. The game class controls starting and playing a game against an AI. This class displays the chess board and allows the player to move pieces around. The implementation is done using pygame to draw the board and allow for user interaction.

We were able to determine the "skill" of the ai by determining how often it won games when playing a random moving opponent. If the AI often loses against a random moving opponent, that is really not good. If the AI draws against a random opponent, that is still not good. If the AI wins often or most of the time, if not all of the time against a random moving opponent, that is what we are looking for. However, we also need to make the AI good against people who even sort of know how to play chess. Therefore, while we play the AI we can evaluate the moves it makes and determine if it is a good or bad move, or if it generally makes good or bad moves, and then evaluate the performance of the AI based on that. We can then spend time altering our metrics for scoring as well as ensuring there are no bugs with the algorithm.

3. **Methodology**:

Pieces. There is a generic Piece class, and a specific class for each of the piece types (Pawn, Knight, Bishop, Rook, Queen, King) which all inherit from the generic Piece class. When created, a piece is initialized with generic piece attributes, such as its position, its team/color, and an all_legal_moves class variable. This variable is used for a piece to be able to easily access the list of its possible legal moves. Each piece overrides important attributes specific to that piece type. This allows each piece to have its own worth, moving criteria, string representation, and name identifier used to access its all_legal_moves. Pawns specifically have a method that allows it to make a move up two instead of one during its first move, when it hasn't moved before. All pieces also have a method can_move_to that allows you to determine whether a piece can move to a certain position given its current position and moving criteria.

The board class stores a 2d array of all of the pieces on the chess board. Each piece also stores its own position as well. The board is able to start a game with the correct starting positions. It sets the correct starting color, represented as 1 for white or -1 for black. The board also stores a dictionary of the white and black pieces, with the piece object as the key and the position as a value, there is a method that is able to update this dictionary as the game goes on. We have a method that is responsible for handling the movement of pieces. It takes as input two positions and checks the legality of the move. It checks if there are any pieces in the way, if the piece is put in check, whether or not the piece is capturing the right team and so forth. It is also responsible for moving the piece object on the board and the piece's internal storage of its location. By moving the piece, the piece object is able to check whether a piece moved in a legal direction, for example: diagonal for bishops and in straight lines for rooks.

We have individual methods to check the legality of moves as well as states of the board. The first of which is a method to determine if there are pieces in the way between two positions on a board. We have a method to determine whether a given team is in check or not for a board state. With all of this information we are able to determine all of the legal moves for the team who's turn it is. So if it is white's turn, it returns all of the legal moves for white pieces. Then it finally determines the end game, if there were 200 moves total the game ends, if there were 50 moves since a capture the game ends, and the game is able to detect and end with checkmate and stalemate.

- AI -

Our AI relies on a minimax algorithm to determine the best move it has available. On any AI turn, the AI evaluates a move by calling minimax and passing in the current state of the board, and whether the AI is maximizing (is white) or minimizing (is black). Minimax checks the list of available moves the AI can make, and for each move, copies the board, makes the move, then calls itself recursively with the opposite objective (if minimizing, maximize, if maximizing, minimize) and checks the player's legal moves in the first recursive call, and repeats the process to a depth of 0 (the depth parameter is subtracted by 1 for each ply we move forward. Our AI is ply 3, so we step down 3 times). At a depth of 0, a board score is evaluated instead of minimax being called recursively. This score is determined by a scoring function that calculates a board score on the board at the time (4 moves forward) based on our scoring criteria (material, pawn progression, knight development, negative king development, general piece development), and returns a score based on the board passed to it. For scoring purposes, a higher score implies a better board position for white, a lower score implies a better board position for black. The score is then passed up to the first depth level from the bottom, where the score is stored as the current eval, and then depending on the objective, m_eval (max or min eval) is updated if the score is higher (better board position for white) or lower (better board position for black). This best value after iterating through the moveset if returned upwards along with the move that made the board have that state as the evaluation for the move in the set, and the process repeats, selecting either the minimum evaluation or a maximum evaluation, until we arrive back at the first depth layer we entered at (3 in our case), where the same process is applied, finally returning the best move.

The logic behind this is that we should assume the opposing player (the player in this case) will be making the best possible choice given a set of moves. For this example, assume the player is black and the AI is white, and our depth is 3. The minimax algorithm would recurse down to a depth of 0, then return the score of the board after choosing the moves it chose to reach that node to depth 1. The AI would choose the move that maximized their score at this level, and it would be returned as the score for the node above as the resulting score that would follow the player move at a depth of 2. This repeats for every move at depth 2 until all of the moves have a score, and, since it is the player's turn, we can assume the player will choose the lowest possible score (best for black) and pass that value up to depth 3 as the result of the move from depth 2. This repeats for every move in depth 3 until we choose the highest possible score at this level (maximizing for AI), and return it as the best move for the AI.

To elaborate on what actually is a good board for white versus a good board for black, we use a scoring metric composed of many different criteria in chess (generally positive influencing criteria) that was listed earlier; material, pawn progression, knight development, negative king development, and general piece development. Material is the value of the piece itself and the value of the pieces of the other team. A score of a board totals the values of the remaining pieces of white, then subtracts the total of the values of the pieces remaining for black. This creates a weighting balanced around zero, where positive is a board where white has a material advantage and negative values are where black has a material advantage. The rest of the scoring criteria are totaled into a "white score" and "black score" total where black score is eventually subtracted from white score and added to the total score--the same principle of negative implies a black favored board and positive a white favored board is the same. The pawn progression adds points as the pawns of a given team move forward, promoting pawn movement down the board for both teams. Knight development follows the same logic, except with knights rather than pawns. Negative king development increases score as the king remains further back on the board, as it stays near the bottom edge for white and the top edge for black. General piece development increases the score if a piece has more options to move than it had in the previous board.

After implementing all of the features we spent time testing different scoring metrics to see if we could optimize the AI. We also spent time optimizing the most time consuming parts of our code in an effort to make the AI faster.

4. **Results and Analysis**:

Through the course of developing this AI, we had many different stages of results depending on the progress we made on the AI. Our first couple attempts at making an AI would only have an AI win half the time, and often get into stalemates. As we rewrote our AI, changed our scoring metrics, and shifted our paradigm (started working on minimax), our results gradually got much better.

The result of chess is simply the winner.

Results for 5 games of depth 3 AI (white) vs random movement (black). Board progress is displayed for each move in every game: https://github.ccs.neu.edu/adrianmonaghan/DS3500-Chess-Ai/blob/main/game_results.txt

Our depth 3 AI can beat a random movement AI every time, rather efficiently (5v0).

Results for x games depth 3 AI (white) vs depth 2 AI (black). Board progress is displayed for each move in every game: https://github.ccs.neu.edu/adrianmonaghan/DS3500-Chess-Ai/blob/main/depth_3_vs_2_results.txt

This implies that an AI that is looking 3 moves ahead can outperform an AI that is only looking 2 moves ahead, which is true (5v0).

Profiling: https://github.ccs.neu.edu/adrianmonaghan/DS3500-Chess-Ai/issues/42

Github repo: https://github.ccs.neu.edu/adrianmonaghan/DS3500-Chess-Ai

5. **Sources**:

- https://www.youtube.com/watch?v=U4ogK0MIzqk&ab_channel=SebastianLague
- https://en.wikipedia.org/wiki/Minimax
- https://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning
- https://commons.wikimedia.org/wiki/Template:SVG_chess_pieces

6. **Appendix**:

Because of the nature of our project, we do not have data visualizations nor diagrams. All of our results are linked in section 4.