351 AI Chess Project

By JT Ohlandt

Problem Space

The main objective is to design a chess board that can be played in 2 player mode and design an AI to play against the player. The chess board will have all the features of a game of chess. Including special moves, not allowing the player to move into check, and detecting checkmate and stalemate. The AI will be built using a minimax algorithm. The AI should be good enough to make reasonably smart moves and not move into check or checkmate.

I encountered multiple challenges building this project, both related to the AI and the board. Implementing the many rules required to castle, pawn promotion, and en passant introduced complexity to the boards structure. I had to redesign functions and take into account other complexities as these special rules often required multiple pieces to be moved per turn or the player to be prompted for input. The biggest challenge I had relating to designing and programming the AI was getting it to be efficient enough to run the game. To improve AI efficiency, I took a multipronged approach. I focused on redesigning the various functions required to perform a move. Implementing beta and alpha pruning reduced the amount of paths the tree had to explore. Finally, I introduced multiplexing allowing the algorithm to run on multiple processes.

Board and AI Techniques

As mentioned previously, for my AI I am using a minimax algorithm. The algorithm works just like the worksheets we did in class. It recursively goes through the tree focusing returning the move that will generate the best score. The score is calculated by the number of pieces and value of those pieces on the board, and putting the king in check or checkmate or stalemate as well as preventing the AI's king from moving into check checkmate or stalemate. It uses alpha-beta pruning and multiplexing to increase efficiency.

When designing my initial board I did not make it possible to undo a move. In addition my validate_move function would move pieces in special cases like pawn promotion, castling and en passant. Due to these factors when testing for check, checkmate, or stalemate the program had to create a copy of the board each time. In addition, it had to create a copy for the whole game for each node in the minimax function. This made each move incredibly inefficient. This wasn't a problem when it was in 2 player mode as only a few moves were being performed each turn. However, when running the AI it made the performance awful. To fix this issue I rewrote several functions and added the undo function. I moved any alterations in the validate_move function into the make_move function. This allowed the validate_move function to be called without altering the state of the game. I then coded an undo_move function. This function undid the move based on the previous move. It then restored the various game attributes from a stack that was updated whenever a move was made. Once I had undo and redo functionality I rewrote the is_checkmate_stalemate function and the minimax function. As there were no longer several game copies being made each turn this dramatically enhanced performance of each move and the minimax algorithm.

Alpha-beta pruning works by reducing the amount of branches the AI must explore before generating a best score. It does this by keeping track of a beta and alpha value. These beta and alpha values represent the best score each player can get at a certain node. Once it is clear that a player will never choose a move down that path as beta > alpha or beta < alpha then the AI will no longer explore that path.

Adding multiplexing was the final way I boosted performance. Rather than running the whole algorithm through one big minimax tree it breaks apart each legal move into its own tree in its own process. This allows each legal move to run its own tree in parallel enhancing performance substantially.

The minimax algorithm time complexity is $O(b^d)$ where b is the number of branches and d is the depth. This can be improved with alpha-beta pruning as it is much less likely to have to go through all the branches. This makes the runtime $O(b^{d/2})$ in best cases. As only one branch is being stored at a time as the tree is being searched with depth first search the space complexity is just O(d) where d is the depth. After introducing multiplexing this changed the algorithm to $O(b^{d/2})$ in the best case per process. The space complexity is still O(d) however it did increase in size as copies of the game were created for each legal move.

There are multiple limitations for my current algorithm. First, it assumes the player is trying to minimize their score based on the AIs score calculations. The other player is not perfect and may make mistakes or they may have other ideas that the AI is not accounting for. In addition, the AI would be more accurate taking into account multiple parameters and making further efficiencies that would allow for increased depth.

An alternative to using minimax would be using a neural networks instead of or in combination with minimax. The neural networks could be trained on chess data then generate whether the move is beneficial to white or black.

AI vs Human Thought Process

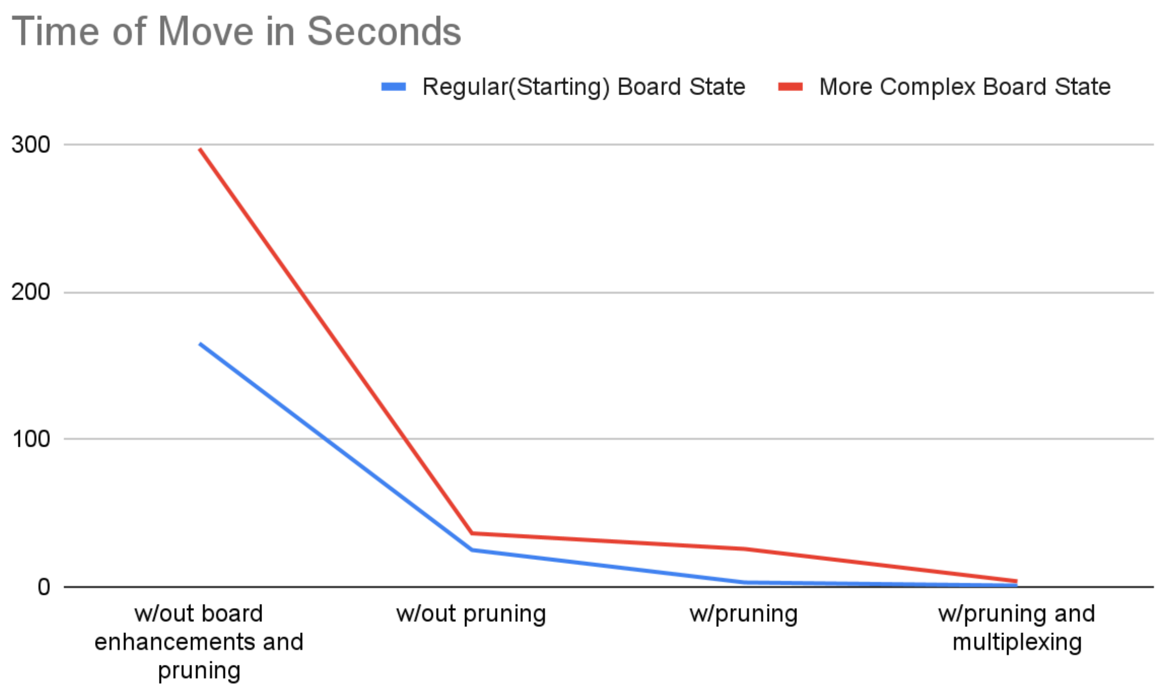
When playing chess an average player will often make the move that they believe is the best move. What makes a move the “best” is how the player believes it sets them up to take favorable trades or get out of unfavorable ones and to prevent checkmate of their own king or put the other player in checkmate. I designed my AI using the minimax algorithm. A score of whether or not it is a good move is decided based on various factors such as what pieces are gained or lost or if the move puts the opponent's king in check or checkmate. This is very similar to the factors weighed by humans when playing the game. Increasing the depth of the minimax tree is basically like the AI thinking multiple moves ahead. The power of having an AI is that it can think further ahead and consider more possibilities than a human could when designed efficiently.

In a game like chess, it’s difficult if not impossible for a player to consider all the possibilities. This is why we take shortcuts, memorizing common strategies, and consider whether moves are worth exploring. Introducing alpha-beta pruning does the same thing. The AI is able to figure out that assuming the player is trying to minimize their score they would never make that move and therefore it's not worth considering.

Multiplexing is another strength that the AI has that a human does not. Humans cannot run things in parallel and instead can only think about one thing at once. This allows AI and computers in general to solve algorithms and problems at much greater speeds.

Empirical Analysis

Upon running my algorithm for minimax the program ran incredibly slow to the point of being unusable. This was because my board was incredibly inefficient and I had not made any optimizations to minimax. To reach an acceptable level of performance I first improved the board which substantially improved efficiency per move. Next I introduced alpha and beta pruning, reducing the amount of exploration necessary to generate the best move. Finally, I introduced multiplexing which allowed the algorithm to run on multiple processes allowing branches to run in parallel. Below is a graph of the time in seconds for the AI to generate a move. In the blue line the AI has substantially less legal moves then the red line. The depth was set to 4.



The results are what I expected them to be. Each one of these improvements did and should have resulted in significant performance gains.

Lessons & Improvements for Next Time

I definitely learned a lot in this project both about AI and other skills. While I did take multiple object oriented programming classes and python classes. My senior year I did not focus on either of those so this project was a good way of refreshing those skills. I had never built this large of a python project from scratch. I feel much more confident in my ability to code other larger projects. It was surprising to see how inefficient my AI algorithm was before I made improvements and a reminder of how problematic a program with a high growth rate (A high O runtime) can be. It makes me have further appreciation for the ability of LLMs with billions or more parameters are able to function in a reasonable amount of time.

Originally when I implemented the special moves in chess I made it so that some portion of the move would happen when the piece was called rather than when the make_move function was called. This led to much more complexity down the line and made it difficult to reverse the moves as the move was taking place in 2 different functions in different parts of the move process. In hindsight when designing games or systems with AI in mind I would make it so my initial design could be easily reversed. In addition, encapsulating the function's purpose in the function is generally a good coding practice.

For my AI, I would start by implementing alpha beta pruning right from the start as it makes a huge difference in efficiency. In addition I would add multiplexing right away as well and possibly making the multiplexing not only apply to the first move but making it apple recursively. I would also add more parameters improving its abilities and potentially use other algorithms or techniques to further enhance efficiency or performance.

Chess Board

How to Play

Select Game Mode

When prompted enter “AI” to play against the AI and type “Player” to play against the player. The AI will always be using the black pieces.

Move Format

Moving is simple. When prompted to enter your move in long algebraic notation. For example, to move a pawn from “d7” to “d5” enter d7d5. Even when capturing it is the exact same (no need to add “x”).

Special Moves

How to castle

To castle a player can enter the initial and final spot of the king. The rook will move automatically. For example, to castle the king on the top left corner of the board type “e8c8.” Of course this will only work if the rules of castling are met.

Pawn Promotion

When the pawn reaches the end of the board the player will be prompted to select a piece to replace it with.

En Passant

If the requirements for en passant are met and the player may enter the long algebraic notation diagonal move for their pawn and the other pawn will be captured.

Other Commands

Quit

Input “q” to quit the current game.