Chess engine in pure JavaScript (Human vs AI)
Made by Benas Sarachovas

Introduction

The chess engine for a Human vs. AI experience was developed using algorithms and heuristic evaluation techniques, which are covered in this poster.

DeepBlue, a computer programme, became the first AI to defeat a human expert at a chess game in 1997. The algorithm returned the optimum move for the AI given the situation of the board after searching 6–8, and occasionally even 20–ply, deep.

In order to understand how the chess AI works and what makes it effective, as well as to see my own creation defeat me in a chess game, I came up with the concept of building a chess engine.
Also, those who want to learn more about chess AI will find this engine to be a useful example.

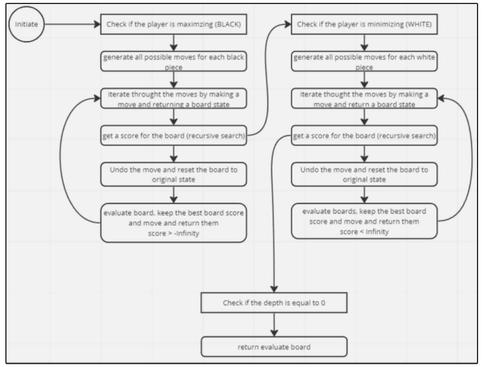
Algorythms used for AI:
- -Minimax
- -Alpha-Beta Prunning

Heuristic evaluation based on:
- -Piece value
- -Piece position

Methods

Here is a condensed version of the engine's minimax algorithm.

The algorithm first generates all moves that are available for black pieces and iterates through them. Once the first move for black pieces is made, the algorithm then generates all moves that are available for white pieces in that board state and iterates through the moves, returning the board's final score, which is used to heuristically evaluate the best board (highest score for black piece). The best score and move for AI are returned after iterating through all of the black piece's potential movements and subsequent white movements.



An artificial intelligence (AI) chess engine uses the following condensed version of the heuristic evaluation function.

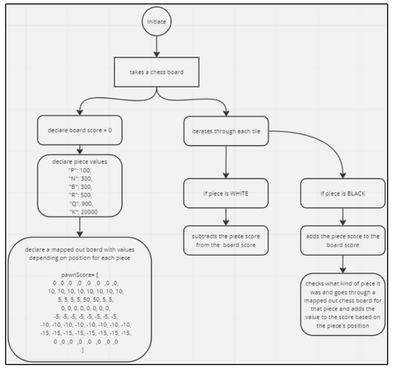
The evaluation function declares a beginning score of 0 after receiving the board state. Following that, it checks to see if a piece is on each tile as it iterates across them.

If the piece on the tile is White, the board score is reduced by the corresponding amount.

If the piece on the tile is BLACK, the algorithm finds a positional value for that piece and adds it to the board score along with the matching value.

The minimax algorithm cannot move forward without the positional values because the starting positions yield no piece taking of the other colour, which means the algorithm is only effective when a white piece threatens the black piece and vice versa.

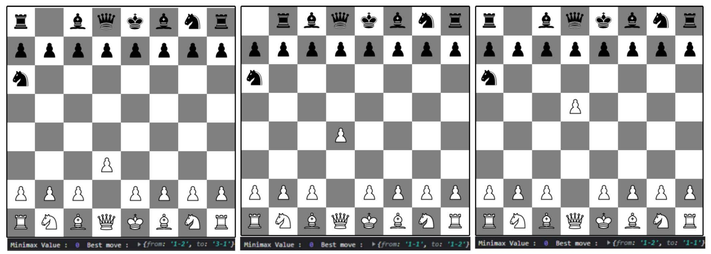
The mapped out positional values only favour the black because we want to encourage the movement of all the pieces rather than just one.



Data Analysis / Results

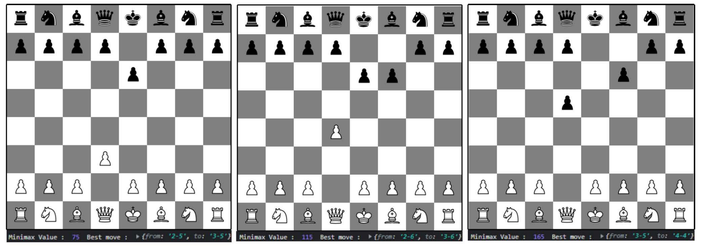
Comparing the difference in gameplay using the different settings of the heuristic evaluation function explained above these are the results.

Without mapped out positional values for pieces:



This essentially proves the idea that I mentioned before that without positional values the only way the pieces will progress is if they are threatening to take the opposite color or vice versa.

With mapped out positional value for pieces



https://github.falmouth.ac.uk/BS260655/BS260655-COMP250-AI

Even with the alpha beta prunning, the pace of the existing algorithm is relatively slow, indicating that this kind of algorithm is not the greatest fit for the task. Minimax (which is simpler to comprehend but much harder to implement and a little bit slower) or Negamax (which is simpler to construct but much harder to understand and more efficient) were the two algorithms I had to pick from when trying to create a chess AI.

I chose minimax because I didn't want to take a chance on being trapped with a broken AI.

A more efficient algorithm would require the addition of additional evaluation functions that would control when the minimax algorithm should be initiated. As it stands, the algorithm is inefficiently checking all of the game states, which takes too long and makes the game boring. This understanding came from analysing the results I obtained from experimenting with different depth (ply).

Ply(depth) 1 speed: 81.06 ms (self 0.55 ms) minimax

Ply(depth) 2 speed: 15.36 s (self 8.13 ms) minimax

Ply(depth) 3 speed: Endless

Conclusion

As a result of the statistics I'm observing, I may need to reconsider the algorithm I'm using because NegaMax would work better for this type of game since it requires less computer power than its counterpart, minimax. Also might consider other options as there are many : Monte Carlo Tree Search, Iterative Deepening, NegaMax etc.

It's also important to note that alpha beta prunning may have been implemented incorrectly or the minimax may not be functioning properly, which would explain why it is not reducing the search time of the tree.

The game is still playable and the player might enjoy a simplified form of chess even though it is not as efficient as I had initially anticipated. It still needs improvement in the areas of game states, game end states, a lot of optimisation, UI update, etc.

My goal is to finish this project in year three and turn it into a fully functional chess game with multiple options (Player vs Player, 3 AI difficulties, AI vs AI)

RESOURCES:
Campbell, Murray S., and T. Anthony Marsland. "A comparison of minimax tree search algorithms." Artificial Intelligence 20.4 (1983): 347-367.