Chess AI: ScuffedFish

Authors: Yiu Tran, Rohit Khurana, Jovian Wang

A. Introduction

Chess is a popular board game that has captivated players around the world due to its strategy and seemingly infinite number of board configurations. In chess, two players - one black and one white - take turns moving pieces on an eight-by-eight checkered square board. The overall objective of the game is to checkmate the opponent's king using an arsenal of pieces such as knights, bishops, and queens. Each piece is constrained in its movement; for example, rooks can only move vertically and horizontally whereas bishops can only move diagonally. As a result, different weights are given to pieces on the board, with queens typically having the highest point value due to their versatility. The game is fully observable, deterministic, sequential, multi-agent, static, and discrete.

Solving chess has been subject to intense research by the AI community. A major milestone in AI research came through the development of Deep Blue, an intricate chess AI designed by IBM that beat Garry Kasparov in 1997. Kasparov was considered to be one of the best chess players at the time. Since then, many other chess engines have sprung to life, with each consistently ranked hundreds of points higher in rating compared to the best human grandmasters. Nevertheless, though diverse in their abilities, chess engines typically build off of and employ common AI techniques and methods like the minimax algorithm. For example, the popular open-source chess engine Stockfish was released in 2008 and uses a minimax base algorithm behind its neural networks.

The minimax algorithm was originally formulated for multi-agent zero-sum games. Zero-sum games have a limited utility that is split among each agent, with each agent attempting to maximize their score. Therefore, each player effectively minimizes the utility of their opponents during their turn. Minimax reflects this by assuming one player maximizes utility and one player minimizes the first player’s utility. In the context of chess, a standard method of zero-sum utility evaluation is by scoring material. Each player gains utility corresponding to the sum of the values of their remaining pieces on the board and loses utility corresponding to sum of the values of their opponent’s remaining pieces. The standard piece values, proposed by Claude Shannon in his 1949 paper *Programming a Computer for Playing Chess*, are 1 for pawns, 3 for knights and bishops, 5 for rooks, and 9 for queens. In this way, players can only gain utility by capturing their opponent’s pieces, causing the opponent to lose the same utility. While evaluation of the piece values remaining produces a decent chess agent, such an agent generally does not consider other factors that human players keep in mind such as positional advantage, factors neural networks typically pick up. Thus, this report explores the effectiveness of minimax with different methods of board evaluation. We seek to compare minimax-based chess agents that evaluate boards based on only piece values, minimax-based chess agents that evaluate boards based on only positional values, and minimax-based chess agents that evaluate boards based on both.

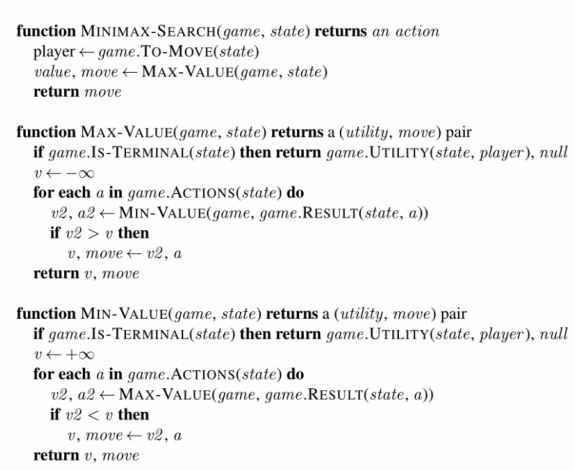
To do so, we made and analyzed a more rudimentary version of Stockfish, without the use of neural networks. We sought to produce a platform in which users could explore the minimax

algorithm with different evaluation methods and the playstyles produced from a combination of parameters.

B. Solution

Using the minimax algorithm with alpha-beta pruning, we explored different methods of board utility evaluation.

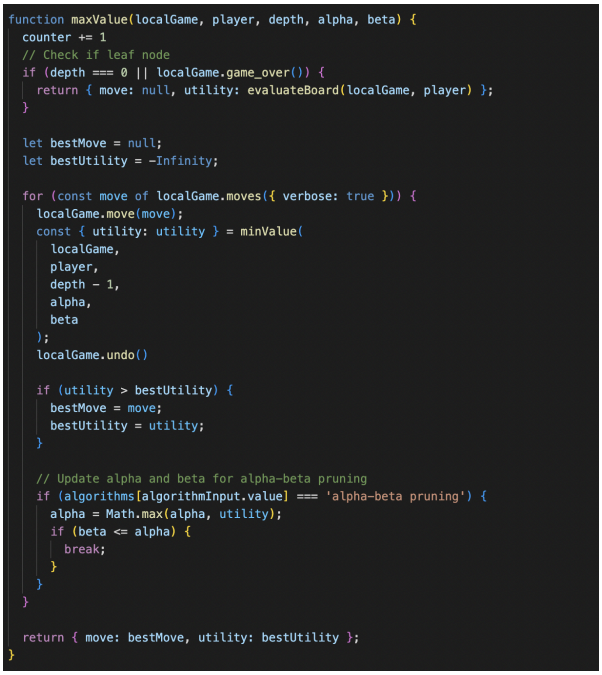
The minimax algorithm involves finding the maximum value for the player's possible moves and the minimum value for the opponent's possible moves. Programatically, the algorithm works by recursively generating all possible configurations for both players to a certain depth, evaluating each possible resulting game state to determine utility, and selecting the action that maximizes the current player's advantage.



function MINIMAX-SEARCH(*game*, *state*) **returns** *an action*
  player ← *game*.TO-MOVE(*state*)
  value, *move* ← MAX-VALUE(*game*, *state*)
  return *move*

function MAX-VALUE(*game*, *state*) **returns** a (*utility*, *move*) pair
  if *game*.IS-TERMINAL(*state*) **then return** *game*.UTILITY(*state*, *player*), *null*
  v ← −∞
  for each *a* **in** *game*.ACTIONS(*state*) **do**
    v2, *a2* ← MIN-VALUE(*game*, *game*.RESULT(*state*, *a*))
    if *v2* > *v* **then**
      v, *move* ← *v2*, *a*
  return *v*, *move*

function MIN-VALUE(*game*, *state*) **returns** a (*utility*, *move*) pair
  if *game*.IS-TERMINAL(*state*) **then return** *game*.UTILITY(*state*, *player*), *null*
  v ← +∞
  for each *a* **in** *game*.ACTIONS(*state*) **do**
    v2, *a2* ← MAX-VALUE(*game*, *game*.RESULT(*state*, *a*))
    if *v2* < *v* **then**
      v, *move* ← *v2*, *a*
  return *v*, *move*



```
function maxValue(localGame, player, depth, alpha, beta) {
  counter += 1
  // Check if leaf node
  if (depth === 0 || localGame.game_over()) {
    return { move: null, utility: evaluateBoard(localGame, player) };
  }

  let bestMove = null;
  let bestUtility = -Infinity;

  for (const move of localGame.moves({ verbose: true })) {
    localGame.move(move);
    const { utility: utility } = minValue(
      localGame,
      player,
      depth - 1,
      alpha,
      beta
    );
    localGame.undo()

    if (utility > bestUtility) {
      bestMove = move;
      bestUtility = utility;
    }

    // Update alpha and beta for alpha-beta pruning
    if (algorithms[algorithmInput.value] === 'alpha-beta pruning') {
      alpha = Math.max(alpha, utility);
      if (beta <= alpha) {
        break;
      }
    }
  }

  return { move: bestMove, utility: bestUtility };
}
```

Figures 1 and 2: Minimax-search algorithm

We implemented this algorithm in JavaScript. The function corresponding to *MAX-VALUE* in Figure 1 is shown in Figure 2. We also implemented alpha-beta pruning to help speed up move generation. This technique involves keeping track of two values, known as the "alpha" and "beta" values, which represent the minimum and maximum values for the player and opponent, respectively. As the algorithm generates and evaluates possible moves, it can prune and discard any moves that are known to be worse than the current alpha or beta values. This significantly reduces the number of moves that the algorithm needs to consider, making it faster and playable in a standard browser.

The function *evaluateBoard* reflects the player's utility given the current board state. A typical evaluation method is to sum piece values according to the guidelines presented earlier. However,

to simulate factors that humans and neural networks consider, but a piece-value-based minimax agent does not, we added additional values for each piece by position. We hypothesized that, by taking piece position into consideration, the minimax agent is able to gain utility for correctly developing the board, similar to how humans play common chess openings, and ultimately play better than a minimax agent that does not.

A matrix is created for each piece where each element corresponds to the additional value awarded by the agent if the piece is in that position on the board. For example, the positional matrix of a knight is shown in Figure 3.

-5.0	-4.0	-3.0	-3.0	-3.0	-3.0	-4.0	-5.0
-4.0	-2.0	0.0	0.0	0.0	0.0	-2.0	-4.0
-3.0	0.0	1.0	1.5	1.5	1.0	0.0	-3.0
-3.0	0.5	1.5	2.0	2.0	1.5	0.5	-3.0
-3.0	0.0	1.5	2.0	2.0	1.5	0.5	-3.0
-3.0	0.0	1.0	1.5	1.5	1.0	0.0	-3.0
-4.0	-2.0	0.0	0.0	0.0	0.0	-2.0	-4.0
-5.0	-4.0	-3.0	-3.0	-3.0	-3.0	-4.0	-5.0

Figure 3: Knight position-value matrix

The numerical elements of the matrices were adapted from piece square tables, as explained at https://www.chessprogramming.org/Simplified_Evaluation_Function. Each numerical value was created based on basic chess techniques. For example, it is common to de-value board edge positions for knights, as they have less mobility in those positions; thus, position values on the edges of the board are negative, while position values in the center are positive. For each of the agent's pieces, our evaluation adds the piece value with the corresponding positional value and sums the result from all pieces together. It then does a similar sum of sums for the opponent's pieces and subtracts that result from the agent's pieces to get the total utility of the agent. The result is an agent that considers both piece and positional values, one that we affectionately dub *Scuffedfish*.

To evaluate and let users explore our algorithm and metrics, we made a website (found here: https://jovian.wang/chessbot). Users can play against the selected algorithm and customize the agent. In addition, we added functionality to allow two AIs play against each other, each with their own customizable parameters.

C. Evaluation

In our online platform, the user has three options for selecting how to evaluate the board: material only, position only, and a blend of both material and position.

Our first area of analysis revolved around comparing the computational efficiency of the standard minimax algorithm with alpha-beta pruning; these results are highlighted in Fig. 4. It was found that standard minimax without any optimization shows an exponential increase in the number of board configurations explored as search depth increases. Alpha-beta pruning, though tending towards the same pattern, shows a marked improvement in the time needed to complete its search at depth 4, only considering 45190 nodes (~6.76 seconds) compared to minimax, which visited 374806 nodes (~43.745 seconds)

Our next area of analysis sought to observe, generally speaking, how *ScuffedFish* was evaluating and responding to positions on the board. As demonstrated in Fig. 5, *ScuffedFish* thinks the position is fairly neutral for most of the game (1-30 moves); however, due to some unfortunate blunders, *ScuffedFish* was unable to keep this neutrality for long, ultimately succumbing to checkmate by losing its pieces. This resulted in a final game evaluation of around -110, in favor of white.

Lastly, we compared different evaluation metrics and search depths in an effort to record their associated win rate and benefit when utilized in an AI vs. AI match (Tables 1-3). Our research indicates that, all other conditions kept equal, a purely material board evaluation trumps a purely positional evaluation regardless of side (Table 1). Moreover, when one AI took into account both material and positional aspects of the board, that side won more games on average against an AI that only took into account material, supporting our human intuition regarding the game (Table 1). Similarly, the side with larger search depth, all other conditions kept equal, won more games due to its ability to look further ahead in the future (Table 2). Other AI vs. AI modalities were explored (Table 3) to compare different settings when changed together.

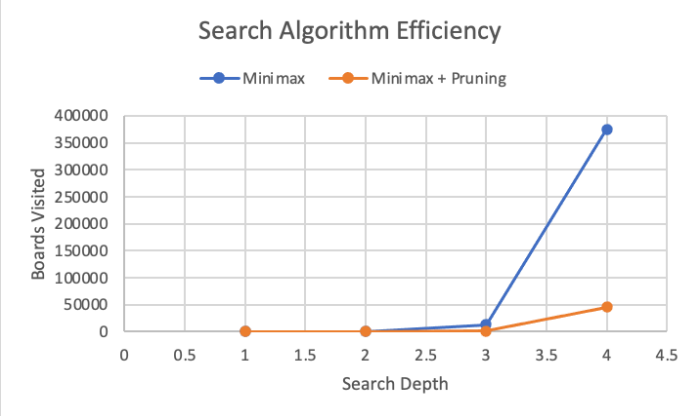


Figure 4: Search algorithm efficiency comparison.

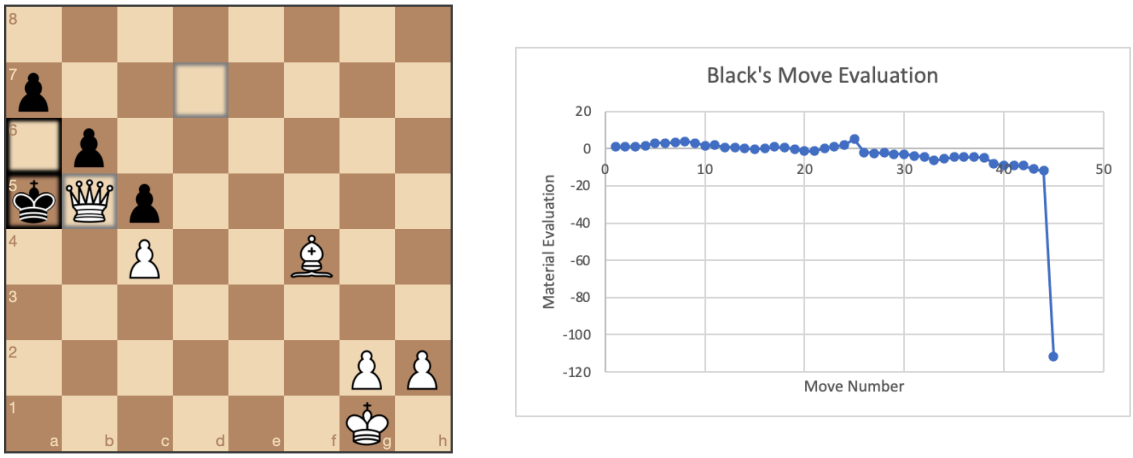


Figure 5: Game evaluation history. *ScuffedFish* plays black (algorithm: alpha-beta pruning, evaluation: material, depth: 3).

Table 1: Comparison of different evaluation metrics. *ScuffedFish* (regardless of being white or black) utilized alpha-beta pruning with a search depth of 3.

	White Material			White Positional			White Material & Positional		
White | Black | Draw	W	B	D	W	B	D	W	B	D
Black Material	4	4	2	0	10	0	6	4	0
Black Positional	10	0	0	0	0	10	10	0	0
Black Material & Positional	4	6	0	0	10	0	3	7	0

Table 2: Comparison of different search depths. *ScuffedFish* (regardless of being white or black) utilized alpha-beta pruning while evaluating both material and position.

	White Depth 1			White Depth 2			White Depth 3			White Depth 4		
White | Black | Draw	W	B	D	W	B	D	W	B	D	W	B	D
Black Depth 1	2	0	8	7	0	3	10	0	0	9	0	1
Black Depth 2	0	10	0	0	2	8	10	0	0	9	0	1
Black Depth 3	0	10	0	1	9	0	2	2	6	9	0	1
Black Depth 4	0	10	0	0	7	3	0	7	3	0	2	8

Table 3: Other AI vs. AI modalities. *ScuffedFish* (regardless of being white or black) utilized alpha-beta pruning.

Black	White	Result
Evaluation: Material & Positional, Depth: 2	Evaluation: Material, Depth: 3	White Wins
Evaluation: Material, Depth: 2	Evaluation: Positional, Depth: 4	Black Wins

D. Conclusion

In general, our AI, *ScuffedFish*, successfully played like a typical chess player, aware of pins, checks, attacks, and any positional advantage the board may offer for their side. We note that *ScuffedFish* even played common openings just as a human would; for example, a common opening pattern *ScuffedFish* played as black in single-player mode was the Four Knights Game. In fact, it won more games than an agent that only considers piece values, reflecting the advantages of board development. The limitation of *ScuffedFish* is that it was only playable on the online platform with a search depth of up to four. At depth four, moves, on average, took up to five to ten seconds. We attempted to implement Monte Carlo tree search, but due to limitations with time, we were unable to fully deploy the algorithm. Our immediate future step for this project is to do so.

E. Presentation Q/A

Which of the state-eval metrics performed best and to what degree?

- The state evaluation metric that was the best was positional & material. According to Table 1, position & material won a majority of times against other metrics.

Any ideas on further improvement?

- In the future, we hope to further compare other utility evaluation functions. We also hope to set up a server so that we can compare heavier algorithms without having to run in the user's browser. We would also like to develop a Monte Carlo tree search.

How would you further decrease the response time of each move?

- We could decrease the response time by increasing hardware, possibly by setting up a website backend server. Given more time, we can also implement Monte Carlo tree search, as well as cache board evaluations to avoid recalculations.

If the depth of the search tree increased dramatically and alpha-beta pruning may not be the best approach to speed up the process, do you have any other options to speed up your chess completion time?

- See above.

How is your solution compared with AlphaZero's solution?

- The solution is, for all intents and purposes, not comparable due to the nature of the platforms.

What is the expected result if you have two AI play each other with the exact same depth?

- The expected solution is draw or even amounts of wins and losses. Refer to Table 2 for results.

What skill level is the AI compared to a human?

- We cannot evaluate ScuffedFish according to chess ratings at this time. However, it is definitely competitive against the typical player.

Amazing demo. It looked professionally made. I wonder what techniques stockfish uses to decreases the runtime as it looks down the decision tree at a greater depth.

- Stockfish uses a lot of hardware. Though relying on the CPU primarily, Stockfish is able to use many threads to speed up evaluation. In addition, its memory reserves are gigantic, allowing it to be able to look many moves into the future.

Could there be an application of reinforcement learning for this problem?

- Yes, reinforcement learning can be implemented in a chess AI. However, it is not within the scope of our problem, which uses minimax and alpha-beta pruning.

How does the opening move vary with differing search depths and algorithms?

- Refer to the conclusion section. In summary, yes. Agents that considered both positional and piece values displayed more common openings.

The AI seems to like to move the queen a lot. Perhaps this is due to there being not enough depth of search?

- We can attribute this to the vast number of moves the queen has, thereby increasing the chance that a queen move should be selected.

F. Author Contributions

Yiu Tran: Positional tables, positional utility functions, material utility function, material + position utility function, tables and figures, and solution analysis.

Rohit Khurana: Solution evaluations, data collection, tables and figures, solution analysis, contributed to AI vs AI functionality and front-end design.

Jovian Wang: Minimax and alpha-beta pruning, chessboard visualization and functionality, website front-end and hosting, AI vs. AI functionality.