

Explanation:

- Board Representation: The board is represented as a 3x3 grid. Static Evaluation Function (`evaluate_board`): This function calculates the board's score based on piece values.

Static Evaluation:-

The static evaluation number in a chess engine (or in this case, the simplified chess game) is chosen based on a heuristic that assigns relative values to pieces and board positions to reflect their strategic importance. The goal is to estimate the strength of a particular board configuration for a given player. Here's how these numbers are typically chosen:

1. Piece Value:

Soldier/Pawn (S): The Pawn is the most basic and least powerful piece in chess, but it can still play a crucial role. In a simplified game like this, it's given a lower value (1) because it's weaker in terms of mobility and potential impact compared to other pieces. Horse/Knight (H): The Knight is more versatile due to its unique movement, allowing it to jump over other pieces and cover various positions on the board. It's more valuable than a Pawn, so it's assigned a higher value (3). Elephant/Rook (E): The Rook is generally the most powerful among these three pieces because it can control an entire row or column, giving it a significant strategic advantage. Thus, it's given the highest value (5).

2. Positional Value:

In more complex chess engines, additional points might be added or subtracted based on the position of the pieces on the board. For example, central control is usually more valuable. This is because controlling the center of the board generally provides more mobility and options for attack or defense. However, in this simplified game, the board is small, and the impact of positional value is reduced.

3. Material Balance:

The evaluation score is calculated by summing the values of all pieces for one player and subtracting the sum of the opponent's pieces. This gives a quick assessment of who has more material (pieces) on the board. The basic formula is: $\text{Score} = \sum(\text{My Pieces' Values}) - \sum(\text{Opponent Pieces' Values})$

4. Special Considerations:

Endgame vs. Midgame: In a real chess engine, the importance of certain pieces might change depending on the game phase. For instance, Rooks often become more valuable in the endgame. However, in this simple model, the values are kept constant throughout the game.

Mobility and Potential Attacks: More advanced engines consider the mobility of pieces (how many legal moves a piece has) and potential threats (pieces that can be captured on the next move).

- Move generation:

The actions function generates all possible legal moves for a given player in a game where the pieces on the board are represented by uppercase and lowercase letters. The function iterates over the

game state (the board) and computes the available moves for the player's pieces, considering different movement rules based on the type of piece.

Key Concepts:

- State: The game board, represented as a 2D array where each element is a piece or an empty space ('').
- Player: The player whose turn it is. 'white' or 'black' indicates the color of the player. Uppercase letters represent white pieces, and lowercase letters represent black pieces
- Legal Moves: The function returns a list of legal moves, where each move is represented as a tuple (from_position, to_position).

Pieces and Movement:

1. S' (Soldier):

- Soldiers ('S' for white, 's' for black) move one step forward, but they can capture diagonally.
- White soldiers move upwards (direction = 1), while black soldiers move downwards (direction = -1).
- They can move one square forward if it is empty, or they can capture a piece diagonally if it belongs to the opponent.

2. 'H' (Horse/Knight):

- Knights ('H' for white, 'h' for black) move in an L-shape (two squares in one direction and one square perpendicular).
- Knights can jump over other pieces, so all potential positions are checked.
- A knight can move to an empty space or capture an opponent's piece if it is on the target square.

3. 'E' (Elephant/Rook):

- Elephants ('E' for white, 'e' for black) move horizontally or vertically any number of spaces, like a rook in chess.
- They can move to any empty square along the row or column, but their movement is blocked by other pieces.
- If they encounter an opponent's piece, they can capture it, but then they stop moving.

Minimax Algorithm:

- The minimax function recursively evaluates the board and selects the best possible move for the AI.
- The find_best_move function is used to initiate the Minimax process and determine the best move for the AI.
- If it's the maximizer's turn, the algorithm chooses the move that maximizes the evaluation score. If it's the minimizer's turn, it chooses the move that minimizes the evaluation score.
- The algorithm alternates between the maximizer and the minimizer, and at the end, it returns the best possible outcome for the player whose turn it is.

Result function

The result function simulates the effect of making a move on the game state. It updates the game state by moving a piece from its current position to the desired position on the board. It handles the game logic for validating moves for different types of pieces and raises errors if the move is invalid.

Terminal state

The terminal_state function checks if the game has reached a terminal condition, meaning the game is over and there are no further moves to be made for the current player. A terminal state occurs in one of the following cases:

1. No White Pieces Left: If no white pieces ('S', 'H', 'E') are still standing on the board.
2. No Black Pieces Left: If no black pieces ('s', 'h', 'e') are still standing on the board.
3. No Legal Actions Left: If the current player cannot make any valid moves (i.e., no possible actions remain for the player).

Time Complexity:

- The time complexity of the Minimax algorithm depends on the branching factor (b) and the depth of the game tree (d).
- Branching Factor (b): The average number of legal moves available at each point in the game.
- Depth (d): The maximum depth of the game tree, which corresponds to the number of moves until the game ends.
- The time complexity of the Minimax algorithm is given by: [$O(b^d)$]

Space Complexity:

- The space complexity of the Minimax algorithm is determined by the maximum number of nodes stored in memory at any point during the execution. This is typically proportional to the depth of the game tree.
- Depth (d): The maximum depth of the game tree.
- The space complexity of the Minimax algorithm is given by: [$O(d)$]