Chess AI

In this assignment, I'm required to build a chess-playing program using Python's chess library. I'll start by implementing a depth-limited Minimax algorithm with a cutoff function, and then optimize it using Alpha-Beta pruning and Iterative Deepening. I'll also create a heuristic evaluation function for board states that aren't game-ending. I aslo made extensions for extra points, such as implementing Zobrist hashing, moves ordering, and advanced evaluation.

Minimax and Cutoff Test

- `choose_move()`
  - choose_move method iterates through all `legal moves` and uses `minimax` to evaluate them. It keeps track of the best move and its evaluation value (best_ev). If it's White's turn, it looks for the move that maximizes the evaluation, and if it's Black's turn, it looks for the move that minimizes it. Once the best move is found, it's returned and I also print the number of visited nodes for debugging.
- `cutoff_test()`
  - My cutoff method checks if the `game is over`, if it's a `stalemate`, or if the `fifty-move` rule can be claimed. If any of these conditions are met, or if the maximum depth is reached, the method returns True for further board evaluation.
- `evalute_board()`
  - For board evaluation in evaluate_board, I assign values to each type of chess piece and sum them up. I also count the number of opponent's pieces and subtract it from the total evaluation. If the game is over, I return a very high or low value depending on the winner. I add a small random factor to the total to ensure diversity in moves.
- `Minimax()`
  - My minimax uses recursion to explore the game tree up to a certain depth. If the `cutoff_test` returns `True`, it evaluates the board using evaluate_board. Otherwise, it iterates through all legal moves, pushing and popping each move on the board to simulate the game states. It keeps track of the best evaluation value (`maxEV` for `maximizing`, `minEV` for `minimizing`) and returns it.
- `Observation`
  - While the Minimax algorithm is a powerful tool for decision-making in chess, it has its limitations. Specifically, it can be slow and may result in repetitive moves during the endgame. This is because the board evaluation is based solely on the sum of material values, without considering the positional value of pieces like the opponent's king. As a result, the AI may end up chasing the king in circles, leading to a draw after 50 moves without any capture or pawn movement.
  - Minimax - depth 3 After implementing Minimax with a depth of 3, the AI's behavior improved significantly, showing a clear strategy for targeting the opponent's high-value pieces like Queen or Rook. However, when the depth is shallow, the AI tends to rely heavily on a single knight and three pawns throughout the game. Additionally, I observed that the Minimax AI

struggles to deliver a checkmate when the opponent is left with only a king, a situation that should be relatively straightforward to resolve.

Evaluation function

- I have a dictionary values that holds the evaluation values for each type of chess piece, with positive values for white pieces and negative values for black pieces.
- First, I check if the game is over using `board.is_game_over()`. If it is, I look at the result. If `white wins`, I return `5000`; if `black wins`, I return `-5000`; and if it's a `draw`, I return `0`.
- If the game is not over, I loop through all the squares on the board using chess.SQUARES. For each square, I check what piece is present using `board.piece_at(square)`. I then look up the value of that piece in the values dictionary and add it to total.I also count the number of opponent's pieces with nagative value. This implicitly evaluation the actions of attack and deffense.
- Finally, I count the num of enemy's pieces and subtract `opponent_piece_count` from total to get the final evaluation. I also add a small `random value` (multiplied by 0.01) to total to introduce some variability in the AI's choices to reprevent repetition. Then, I return the final total as the board's evaluation value.

Iterative deepening

- In my code, I've implemented an iterative deepening version of the Minimax algorithm, which is a bit different from the standard Minimax. Instead of diving straight to the maximum depth to evaluate the board, I start shallow. I first look one move ahead, find the best move at that depth, and then go two moves ahead, and so on, until I reach the maximum depth I've set.
- I initialize two variables, `best_move_global` and `best_ev_global`, to keep track of the best move and its evaluation across all depths.
- I then loop through each depth from 1 to the maximum. For each depth:
    - I initialize `best_move` and `best_ev` for that specific depth.
    - I run my standard Minimax algorithm up to that depth to find the best move and its evaluation.
    - I compare this `best_ev` with `best_ev_global`. If it's better, I update `best_move_global` and `best_ev_global`. At the end of the loop, `best_move_global` will contain the best move I've found across all depths.
- This way, I can update my "best move" more frequently, and if I run out of time, I can just use the best move I've found so far. It's like getting snapshots of the best moves at different depths, and it helps me make more robust decisions.

Alpha-Beta Pruning

- I've added a few optimizations to make it more efficient. Here's how it works:
  - Sets the maximum search depth.
  - Initializes an empty dictionary for the transposition table.
  - Initializes Zobrist hashing table with random 64-bit integers for each piece type and square.
- `minimax()`
  - Checks if the board state has been evaluated before (using Zobrist hashing) and returns the stored value if found.
  - Calls `cutoff_test()` to see if the search should be terminated.
  - `Sorts` the legal moves based on their evaluations to improve Alpha-Beta pruning.
  - Recursively explores the game tree, updating `alpha` and `beta` and pruning branches where possible.
  - Stores the evaluation of the board state in the transposition table.
- `choose_move()`
  - Resets the visited nodes counter.
  - Sets initial alpha and beta values based on whose turn it is.
  - Iterates through all legal moves, evaluates them using minimax(), and updates alpha and beta.
  - Chooses the move with the best evaluation.
  - Prints out performance metrics like the number of visited nodes and time taken.
  - I make this function turn dependat so that I can use the same class to play against each other

Transposition table

- In the implementation of the Alpha-Beta Pruning algorithm for chess, a key optimization technique employed is the use of `Zobrist hashing` and a `transposition table`.
- During the initialization phase, a Zobrist table is created, which assigns a unique `64-bit random` number to each possible piece for each square on the board. This table is then used to generate a unique hash for any given board state through the zobrist_hash method.
- The hash is generated by `XORing` the Zobrist numbers corresponding to the pieces on their respective squares. Alongside this, a transposition table is initialized as an empty dictionary. This table serves as a cache to store the evaluation of previously encountered board states.
- Before diving into the evaluation of a new board state within the minimax function, the algorithm first checks if the board's Zobrist hash already exists in the transposition table. If it does, the stored evaluation is returned immediately, thereby avoiding redundant calculations and speeding up the algorithm. After the evaluation of a new board state, its value is stored in the transposition table for future reference.

Extensions

- Evaluation functions
  - I created three evalution in this assignment:
    - basic material value evaluation for `MinimaxAI`
    - basic material value evaluation for `Alpha_Beta_Pruning_basic`
    - advanced evaluation and move ordering for `Alpha_Beta_Pruning`
  - In my chess AI, I've incorporated a specialized evaluation strategy that adapts to the `endgame` and aims for a more `aggressive` playing style by restricting the opponent's king's mobility.

  - I trigger the endgame strategy when the `number of opponent pieces` falls below a certain `threshold`, which I've defined as endgame_threshold. During the endgame, my evaluation function doesn't just look at the material balance; it also considers the number of `legal moves` available to the `opponent's king`. To do this, I use chess.`Move.null()` to temporarily pass the turn to the opponent and count their legal moves. I then subtract this count from the total evaluation score, effectively penalizing board states where the opponent has more freedom of movement.
  - Additionally, I've added a spatial component to my evaluation by considering the `distance` to the `opponent's king`. The idea is to reward board states where my pieces are closer to the opponent's king, thereby restricting its mobility. I scale this reward by the inverse of the number of opponent pieces, making it more significant as I get closer to a checkmate situation.
  - For move ordering, I've implemented a sorting method called `evaluate_sort`. This method assigns a score to each legal move based on multiple factors. First, it rewards moves that control the center of the board, specifically squares `D4`, `D5`, `E4`, and `E5`. Then, it adds the evaluation score of the resulting board state after making the move. Finally, it adds a term proportional to the square root of the number of legal moves available after the move. This encourages positions that offer more tactical possibilities. I then sort the moves based on these composite scores, allowing my Alpha-Beta Pruning algorithm to examine more promising moves first, thereby increasing its efficiency.
- Complexity comparision (num of nodes visited)

```
random.seed(1)
depth = 3

making move, white turn True
Alpha-Beta Pruning AI recommending move g1h3
(minimax) visited nodes = 206603
(alpha-beta) visited nodes = 9774
--------------------------------
making move, white turn False
Random AI recommending move h7h6
---------------
making move, white turn True
Alpha-Beta Pruning AI recommending move h3g5
(minimax) visited nodes = 398038
(alpha-beta) visited nodes = 19090
--------------------------------
making move, white turn False
Random AI recommending move a7a5
---------------
making move, white turn True
Alpha-Beta Pruning AI recommending move g5f7
(minimax) visited nodes = 697510
(alpha-beta) visited nodes = 32209
--------------------------------
making move, white turn False
Random AI recommending move e8f7
---------------
making move, white turn True
Alpha-Beta Pruning AI recommending move h1g1
```

```
(minimax) visited nodes = 936145
(alpha-beta) visited nodes = 44094
--------------------------------
making move, white turn False
Random AI recommending move f7g6
---------------
making move, white turn True
Alpha-Beta Pruning AI recommending move g1h1
(minimax) visited nodes = 1160113
(alpha-beta) visited nodes = 55607
--------------------------------
making move, white turn False
Random AI recommending move b8c6
---------------
making move, white turn True
Alpha-Beta Pruning AI recommending move h1g1
(minimax) visited nodes = 1417635
(alpha-beta) visited nodes = 68841
--------------------------------
making move, white turn False
Random AI recommending move c6d4
---------------
making move, white turn True
Alpha-Beta Pruning AI recommending move g1h1
(minimax) visited nodes = 1715452
(alpha-beta) visited nodes = 85251
--------------------------------```
```