Name: Olatunji Honour Oluwakorede UID:3035553484

COMP3270 Mini-Project Report

Introduction

This project creates an AI that can play a very decent amount of chess. This is done by using the minimax algorithm with alpha beta pruning to search for the most “cost-effective” move to make. This project was done using the python language and external libraries such as the chess library were used. The chess library implements all the rules of a chess game, including generating legal moves, making a move, creating the chess board, checking for checkmates and so on. Hence, this project majorly focused on the implementation of the minimax algorithm, the alpha-beta pruning and the heuristic functions used for the static evaluation for the configuration of a board in the state space of possible moves.

In order to play the game, the code must be run in a python environment. At the start of the game, the player must choose which colour he will play as. White always goes first. Each move the player needs to make must be inputted in the UCI format. That is, b3b4 means piece on b3 moves from b3 to b4. The board configuration is such that the rank runs from “a” to “h” from left to right, and the files run from 1 to 8 from bottom to top.

Methodology

The AI starts by asking which colour the player would like to play. If the player chooses black colour, the AI plays as a maximizing player and plays first move in every round and vice versa.

When its AI’s turn, it calls the minimaxStart function passing on the current board state, a search depth of 4 and a Boolean value of True (signifying it is a maximizing player) to the minimaxStart function.

The minimaxStart function is responsible for getting the most “cost-effective” move the AI can make given the present board configuration passed to it. It does so by calling the minimax function which returns a static value for each possible legal move in the state space of legal moves for the present board configuration.

In order to give the static evaluation of the chess board for each possible legal move, the minimax function makes the move, evaluates the new board configuration due to that move (by calling the “evaluateBoard” function) and undo the move. The minimax function uses two heuristic functions to give the static evaluation which are “piecesEval” to evaluate the value of the piece and “positionEval” to evaluate the value of the position occupied by a certain piece on the board. The values obtained from these functions are added together to give the total static value of the board configuration.

Based on these values returned from the minimax function, the minimaxStart function returns the best move (if AI is playing as a maximizing player), or the worst move (if AI is playing as a minimizing player). The AI makes the move, prints the move (in UCI format) and the board configuration so the player can know what move the AI made.

When it’s the player’s turn, the AI generates all possible moves the player can make and prints it out. After the player inputs a move, the AI checks the move against the list of legal moves to ensure the move is a legal move. If it isn’t, the AI repeatedly tells the user to try again until he enters a legal move. After making the move, the AI prints the board configuration and repeats its turn.

Implementation

- **piecesEval function**:
  This function takes in the object which is a piece in the chess game and returns a value for the piece based on its type. That is, a pawn gets a value of 10, a bishop gets a value of 30, a knight gets a value of 40, a rook gets a value of 50, a queen gets a value of 90, and a king who is the most important piece in a chess game gets a value of 900.
  In general, chess AIs give the same value to the knight and the bishop. I personally think the knight is more dangerous due to its movement and the fact that it can fork three opponent pieces at once. Therefore, I gave the knight a higher value than the bishop.
  If the piece is a white piece, this function returns a positive value and if the piece is a black piece, this function returns a negative value.

- **positionEval function**:
  This function takes in the object which is a piece in the chess game and the square's name which the piece is found on. This function first translates the square's name to the indexes of the 2D array as seen in lines 106 and 107 and then uses the 2D arrays to get the value of a position based on what piece it is and its colour. For example, a black king would have the following position values:

```
[[ -3.0, -4.0, -4.0, -5.0, -5.0, -4.0, -4.0, -3.0],
 [ -3.0, -4.0, -4.0, -5.0, -5.0, -4.0, -4.0, -3.0],
 [ -3.0, -4.0, -4.0, -5.0, -5.0, -4.0, -4.0, -3.0],
 [ -3.0, -4.0, -4.0, -5.0, -5.0, -4.0, -4.0, -3.0],
 [ -2.0, -3.0, -3.0, -4.0, -4.0, -3.0, -3.0, -2.0],
 [ -1.0, -2.0, -2.0, -2.0, -2.0, -2.0, -2.0, -1.0],
 [  2.0,  2.0,  0.0,  0.0,  0.0,  0.0,  2.0,  2.0 ],
 [  2.0,  3.0,  1.0,  0.0,  0.0,  1.0,  3.0,  2.0 ]]
```

  Therefore, a black king on g8 square would be translated to x = 8, y = 7 (where x is the ranks (row) and y is the file (column)) and would have a positional value of 3.0. If you consider the above array as a chess board where the 0th element represents the 1st rank and the 7th element represents the 8th rank, then it is easy to understand the translation. At the beginning of the game, black pieces stay on the 7th and 8th rank, corresponding to the 6th and 7th element in the array. Hence, for a black king, it is much safer to be found in the 6th and 7th element of the array, than it is to be found in the 0th and 1st element of the array.

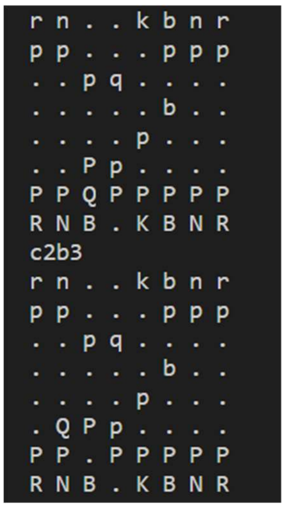
- **evaluateBoard function**
  This function takes in the board as an object. It runs through every square of the board calculating and summing up all the static value of every piece and their position by calling both the piecesEval function and positionEval function. It returns the total static value of a board configuration.

- **minimaxStart function**
  This function takes in the board configuration, the search depth and the Boolean value of maximizingPlayer. Based on the given board configuration, this function first generates a list of possible moves, iterates through the list to find either the worst move (for minimizing player) or the best move (for the maximizing player) with the help of the minimax function. This function returns the most cost-effective move for that player.

- **minimax function**
  This recursive function uses the minimax algorithm and the alpha-beta pruning technique to find either the maximum evaluation or the minimum evaluation down a path of the search tree. It takes in the board configuration, the search depth, value of alpha, value of beta and the Boolean value of maximizingPlayer. It first generates all the possible moves given that configuration. Then it iterates through the list of possible moves and for each move, it plays the move to get the resulting board configuration, evaluates the board by calling itself recursively but passing in the new board configuration, depth minus one, value of alpha, value of beta and the opposite Boolean value of maximising player. Assuming the values for all the new board configurations are gotten, it compares all the values to get either the maximum (if it's a maximising player) or the minimum (if it's a minimising player) and stores as either maxEval or minEval. Then it unplays the move to restore the board to the state it was before the move was made (because technically, it is still thinking and hasn't made a move just yet).
  The alpha beta is used to prune the tree. This is implemented by keeping track of the greater value between the evaluated value of a maximizing move and alpha and the lesser between that of a minimizing move and beta as the tree climbs back up to its root. At any node of the recursive tree, if beta is less than alpha, then the unsearched subtree of that node is pruned, and this is implemented by breaking the for loops that iterates through the possible moves of the present board configuration.
  For a maximising player, this function returns the maxEval value, and for a minimizing player, this function returns the minEval value.
  The base of this recursive function occurs when the leaf of the search tree is reached, that is, depth = 0. The board configuration at this depth is evaluated for its static value by calling the evaluateBoard function and passing the board configuration object to it.

Sample game and overall comment on result

I tested my code with another AI from chess.com mobile app. The AI had a rating of 1600, so I expected it to make smart decisions. My code played as white (capital letters), and the AI from chess.com mobile app played as black (small letters). Here is an instance in the game.



```
r n . . k b n r
p p . . . p p p
. . p q . . . .
. . . . . b . .
. . . . p . . .
. . P p . . . .
P P Q P P P P P
R N B . K B N R
c2b3
r n . . k b n r
p p . . . p p p
. . p q . . . .
. . . . . b . .
. . . . p . . .
. Q P p . . . .
P P . P P P P P
R N B . K B N R
```

Given the board configuration at the top, white's queen was under threat from black's pawn on d3. White could move its queen to a safer spot (d1, b3 or a4), or could take black's pawn on d3. The former would mean white keeping its queen and still having a static value of +90 (value of the queen), and the latter would mean that black would be forced to take the queen on d3 leading to a static value of -90. The code knows to try to maximize its values, so it would not go for the latter.

Comparing the options for the safer spots, staying on b3 could lead to a further capture of the pawn on b7 and possibly the rook on a8 or the capture of pawn on f7 which could led to a capture of the king. This would give an overall static value of +10+50 or +10+900 respectively. Compared to taking the pawn on b7 and the knight on b8 which gives the overall static value of +10+40. Staying on d1 doesn't lead to any attacking of black pieces, so static value is 0 not considering the positional values and given that same amount of both coloured pieces are still alive. Staying on a4 provides same static value as staying on b3 except that on a4, the queen cannot attack the pawn on f7. There is no piece to attack diagonally, hence its static value options doesn't include the +10+900 gotten from attacking the f7 square. Hence, the code deems it fit that its maximized option is to go to b3 which is the move it made. I personally would have move back to d1 to keep my queen safe, but I applaud the code for making a smart decision not to take any pieces just yet and move to a more dangerous square.

Conclusion

This code plays a decent level of chess. Next steps include implementing how the code reacts to a checkmate within a round. It only checks for if the game is over after each round. Also, if I have better computational resources, I would increase the search depth to about 7.