A Chess Engine Using Evolutionary AI

Ethan Gu, Preston Reep, Chase Rooney, Daniel Xu

Northeastern University, Boston, MA, USA

Abstract

This project aims to create a chess engine using the Python programming language and evolutionary computing principles. To utilize our program, we developed a rudimentary front-end for use in the Python terminal. It allows a user to play a game of chess against our engine by inputting a chess board position a piece is currently occupying followed by the position the user desires to move the selected piece to, the computer opponent plays the game automatically. We used a variety of scores to help our engine evaluate the best moves given a board condition. These include passed and doubled pawns, center control, and raw material evaluations. Those are then paired with 3 different defined game states (opening, middle, and end game) to weight those evaluations depending on the phase of the game. We were very limited on computing power so we needed to make a trade-off between computation time and move-evaluation depth. We found that looking more than five moves in the future would break our computers, even with the removal of “bad” moves from each level. While we were unable to complete a fully functioning chess engine with an evaluation depth greater than one, the base methods as well as the evaluation methods could yield a decent AI with more work. With our current iteration, our engine tries every possible move, finds the one with the highest evaluation score, and then makes that move.

Introduction

Our goal coming into this project was to take a bespoke approach to creating a chess engine using Python and evolutionary computing. We wanted to see if our approach could yield better results than other traditional engines that rely on move-based evaluation. Our ambitious goal was to get our engine to a point where it will play better than an average chess player, and to at least consistently beat a player making random moves. To anyone familiar with the chess ELO system, we were hoping to get north of the low 1000s in rating. Our hypothesis was that if we’re able to create a move-based evaluation system with evolutionary computing, then our engine will be able to determine a better move than the average player could identify in the same time frame. While variations of chess engines have been around since the genesis of the

computer, we thought it'd be interesting to use a specific topic learned during our time in class to produce an efficient evaluation system. A quick Google search of our topic does not yield any results that mirror our approach. We also thought it'd be the best mix of complexity while not going beyond our limitations of computing power and time.

Methods

Pieces.py

We started off by constructing a super class, Piece, to represent a chess piece. Each piece has attributes of its color, name, its value in points, and position. The Piece class also keeps track of the total number of black and white pieces. The object of the Piece class also has the ability to return its position with get_posn, and can compute all possible vertical and diagonal moves the Piece can make. These possible moves are to make sure that the pieces would not move off of the board. The Piece also stores whether it is the piece's first move. Other limitations on possible moves will be made in later classes.

These attributes are then inherited into individual classes for each type of piece: King, Queen, Rook, Bishop, Knight and Pawn. Possessing the characteristics of the Piece, each of these specified piece objects now has the ability to store possible moves the piece can make into a single list. These moves are filtered by the legal moves each of these pieces can do as in accordance with the game of chess.

Chessboard.py

Now that we have each of the pieces as classes, we then need to place these pieces on a board. The chessboard class constructs an 8x8 chess board as a list of lists. The function then creates each piece as its class, or adds blank spaces as strings, populating the first two rows as black pieces and the last two rows with white pieces and the middle six rows consisting of strings to hold the space.

The chessboard object is then given functions to display the board, print_board, count the values of total pieces, piece_eval, count the number of pawns, pawn_count, count the number of knights and bishops, num minor, and the count of rooks and queens. Each of the evaluating functions separates black and white pieces. These tools will come in handy when constructing our engine.

Chess.py

With a chessboard containing pieces and an engine to select the best move given a board state, we can move on to actually making moves on the chess board itself. We do this through the Chess class. Our class stores the board after each move, as well as the move log, a list of all valid moves given the state of the board, whose turn it is, if the state of the board results in a stalemate, if it is a checkmate, if it is a draw, and what engine we are using to play against.

Within our class functions, we have four static methods. Coordinates that take chess notation and convert it into coordinates on the board. Uncoordinate takes coordinates on the board and converts them to chess notation. In_check checks if the king can be taken by another piece. locate_piece finds a given piece's location on the board. These functions will be implemented into our class functions.

Our functions on the Chess class can be separated into two categories: functions that modify the board and functions that allow the engine to operate.

Change_turn function changes who's turn it is.

Evaluating if an ending is reached, check_mates checks if the game is either in check or drake mate, draw_by_rep checks for instance where the player or engine repeats the same move three times in a row. If there is a change in the state of either of these variables, the init variable is updated. Check_game_over checks if either of those conditions is met.

Promotion replaces a pawn into a queen if it reaches the other end of the board.

Next, we look at functions integral to movement of pieces. try_move creates a deep copy of the board for a move to be placed on. This will allow us to evaluate a move without making it on the board itself, and returns both the board and all the possible moves the opponent can make. Make_move overrides the current board and replaces it with the board following the piece moving positions, and then makes the next turn and checks for check. get_possible_moves returns a dictionary of all pieces and the positions in which they can move to. get_valid_moves builds upon get_possible moves and filters out possible moves that could endanger the king.

With the ability to move pieces and be in accordance with the rules and conditions of chess, we now need to make evaluations of the game state for the engine to use to determine which move is best to make. The game_phase function checks to see what point of the game the engine is in, either the beginning, middle, or end. The phase of the game effects the wight in which we give certain attributes that will be calculated via the following means: material_eval brings the material_eval function from the chessboard to be used on the game, control_evalution generates a positional evaluation of board control. center_control evaluates the control only in spaces in the center of the board, get_pawn_locs updated the init with the current locations of all the pawns, passed_pawn counts the number of passed pawns, doubled_pawns counts for number of files (column) that contain two pawns.

The opening, middle_game, and end_game scale the matieral_eval are functions that allow for the board evaluation to be scaled based on the state of the game itself. Opening adds center_control*0.15 to the evaluation to emphasize center control, middle_game adds control*0.2 and doubled_pawns*0.1 to emphasize positional principles, and end_game multiplies the initial material_eval by 1.2 and then adds control*0.1 and passed_pawn*0.25 to value king activity and passed pawns more. Get_eval looks at the state of the game and uses it to make the proper evaluation given the aforementioned rates. This evaluation is what our

engine attempts to maximize with the moves that it makes. There are two versions of the chess class. In chess.py, out initial attempt at the engine is presentent. In chess_with_engine.py, the version of the engine that only looks at the present board state is implemented.

When the engine moves, it's first move is hardcoded to be a move of two space forward for a pawn in either file C,D, or E. In each engine move turn following this, the engine move function will take the best possible move calculated by our engine and update the board to make that move. Our engine works by checking all of the valid moves that can be made, trying each move, evaluating said move, and returning the move that maximizes the evaluation.

chessdriver.py

It is the chessdriver.py file that finally gives us a main function that allows the game to play! To allow the game to be played, functions are introduced to allow for player inputs. The player_move function allows the player to modify the board and checks to see if the inputs are valid. If the inputs are not valid, the user is given the opportunity to make another move without causing an error. check_game_over function checks if a game over function is reached, determined by previous functions implemented into Chess. Based on what turn this condition is met, it will output who had won the game. Lastly, the chess_input function allows the player to select what color they want to play and the depth of the bot they would like to go against it.

The game is run through the main function. An instance of the chess class is created to construct the board. The player is then asked to make their color and depth selection from the chess_input function. Based on the inputs, the proper eplayer and engine colors as well as the engine depth are set in the Chess object. The function then makes the player or engine move first based on which color is selected, and after each move is made the chessboard is printed out and check_game_over is run to see if the game is over. If it is not, the board is printed again and the game continues, flipping between player and engine turns until a winner is determined.

Analysis

```
Which color would you like to play?
White
What depth would you like to play against?
5
```

Game starts by having the player select what color they want, and the difficulty of the bot they are playing against.

A	B	C	D	E	F	G	H	
b Rook	b Knight	b Bishop	b Queen	b King	b Bishop	b Knight	b Rook	8
b Pawn	b Pawn	b Pawn	b Pawn	b Pawn	b Pawn	b Pawn	b Pawn	7
--	--	--	--	--	--	--	--	6
--	--	--	--	--	--	--	--	5
--	--	--	--	--	--	--	--	4
--	--	--	--	--	--	--	--	3
w Pawn	w Pawn	w Pawn	w Pawn	w Pawn	w Pawn	w Pawn	w Pawn	2
w Rook	w Knight	w Bishop	w Queen	w King	w Bishop	w Knight	w Rook	1

```
White to move
Enter what piece you would like to move:
```

Prints initial board and prompts the user to input a move

```
Enter what piece you would like to move:A1
---Canceled Move---
---Select New Piece---
White to move
Enter what piece you would like to move:A7
---Canceled Move---
---Select New Piece---
White to move
Enter what piece you would like to move:I2
---Canceled Move---
---Select New Piece---
White to move
Enter what piece you would like to move:
```

Move function accommodates for missed inputs and prompts user to select a new move

A	B	C	D	E	F	G	H	
b Rook	b Knight	b Bishop	--	b King	b Bishop	b Knight	b Rook	8
b Pawn	b Pawn	b Pawn	b Pawn	--	b Pawn	b Pawn	b Pawn	7
--	--	--	--	--	--	--	--	6
--	--	--	--	b Pawn	--	--	--	5
--	--	--	--	--	--	w Pawn	b Queen	4
--	--	--	--	--	w Pawn	--	--	3
w Pawn	w Pawn	w Pawn	w Pawn	w Pawn	--	--	w Pawn	2
w Rook	w Knight	w Bishop	w Queen	w King	w Bishop	w Knight	w Rook	1

```
Black wins by checkmate!
```

If a game_over condition is met by:

checkmate: the game ends and displays a winner

stalemate or three-move repetition: the game ends and displays a draw

Conclusions

At an initial test during our project conference, we had our engine play completely random moves as a proof of concept for our game-related methods. Even self-professed “mediocre” chess players were able to consistently beat this iteration of our program.

We initially planned to build upon the evolutionary engine that was given to us in glass, however we did not find it to be the optimal method in addressing the problem in hand. For our method of evolution, we were not creating a large population of instances and then culling on multiple parameters/fitness criteria. Instead, we compiled and weighted our parameters based on the game state and removed dominated solutions that resulted in a worse evaluation. One limitation we ran into was our weighting of the game states. Choosing those values proved to be difficult and had a profound effect on how the game was played out. Further work could help smooth those values out and potentially introduce an algorithm that will adjust them for us.

If we were to do further work on this project, we would want to spend more time on the recursive engine move function that looks ahead to the players best moves in order to find an outcome that produces the best outcome in the long run. Additionally, one thing we would be interested in implementing is a main function that makes the AI play against itself and compare the performance of bots of different depths.

Author Contributions

Ethan Gu

- Initialized piece objects, wrote functions to get possible diagonal and vertical moves

Preston Reep

- Wrote player input for driver, worked on static functions (coordinate, un coordinate), worked with Daniel on coding the possible moves for pieces as, made singular depth AI, presentation slides, methods for writeup

Chase Rooney

- First iteration of chess board, presentation slide, abstract, introduction, and part of conclusion of report

Daniel Xu

- Did substantial work on implementing the pieces. Updated chessboard class, Chess class and related functions, Chess Driver and related functions , Recursive engine