Senior Project - Chess AI

Gennady Sytnik
Foaad Khosmood, Ph.D.
December 12, 2022
Github: https://github.com/gsytnik/Cal_Poly_Senior_Project

Introduction

Thesis:

Research chess and chess AI and use the data gathered in order to create an AI that is an improvement upon an existing chess AI.

Overview

Chess tends to consist of 3 phases of the game: the opening, middlegame, and endgame. These phases of the game have approaches which differ from each other in both AI and in the real world. The opening tends to be widely studied in the human world, with theory for openings existing deep into most positions. The middlegame in the human world tends to be a place where players look for tactics and positional play which would in theory yield them an advantage or cause the other player to blunder. The endgame is widely studied by the best players in the world, but is difficult to perfect with only theory for some specific endgames being memorized (in example rook-and-pawn endgames).

With the vast improvements to AI in recent years, players began to use them to study games, play against, or even create AI to play chess against other AI. These AI also tend to have separate approaches to the opening, middle, and endgames.

Background and Tools

To complete this project the following background, tools, and methods are needed/used:

- An extensive knowledge of Python and how to utilize Python modules
    - Includes refactoring, python principles to an intermediate level
- Automation of refactored python AI written by others in order to play games against copies of the respective AI
- Data processing and File I/O knowledge for working with large datasets
    - Pandas
- Genetic Algorithm knowledge for evolution of Piece Square Tables using another original python project cloned from github
    - pyGAD
- Regex tools such as re, basic web scraping
- Python chess module
- Knowledge of the game of chess and basic principles behind chess, although not necessarily great skill or knowledge
- A very knowledgeable project advisor (Foaad Khosmood, Ph.D.) to guide the project.

General Chess AI Overview:

AI Opening:

Like in the human world, the openings are widely studied. Sometimes programs use Neural Nets to find the best heuristics for position evaluation and a deep minimax-like algorithm with pruning to find the best moves, but usually engines rely on databases of moves to make moves within the opening. These databases can either be AI generated, but often utilize play from various world-class-level games.

AI Middlegame:

This is the most tricky part of a chess game, where engines differ the most from one another. In recent years, neural networks (Specifically NNUE) have been the most popular tool to evaluate a position with a heuristic. Most engines however, even those utilizing these neural networks, use some form of piece square tables (PST) which are matrices denoting the value of a specific piece on a particular square of the chess board. Some engines rely almost entirely on this (usually paired with a minimax) in order to evaluate positions, such as the moderately-well-known sunfish (an engine recognized for its very short code that still plays at an elo of 2000)

AI Endgame:

The endgame of chess has been entirely solved for positions with 7 pieces or less. People extremely dedicated to the endgame have written programs to create ‘tablebases’ which are files or databases that contain every possible position from a certain amount of pieces and the winning moves from them. For example, the last largest tablebase created was the 7-piece tablebase, solving all positions with 7 pieces or less on the board. The downside is that tablebases can take up a very large amount of space, with the 7-piece taking up 140TB. Some programs also use programmed logic to recognize certain openings just like humans do, or perform a simple minimax with pruning for this phase of the game.

Solutions Used/Avoided:

Used:

- Opening Database via WebScraping - Online parsing of database found on lichess.com with games of players of average elo 2500+. Relatively easy to implement WebScraping and database already built.
- PST’s - Relatively easy to throw into a genetic algorithm provided the code used does not yield many strange issues (see roadblocks section)
- MiniMax - Most chess AI have this in their implementation, and so did the code from the repo used (see works cited/implementation sections)

- StockFish (endgame only) - Since endgames are theoretically solved to a certain amount of pieces (see results) and tablebases are too large to store on a personal computer, this project used the best possible alternative resource available for the endgame. Stockfish is the top engine in the world at the time of this project (2022).

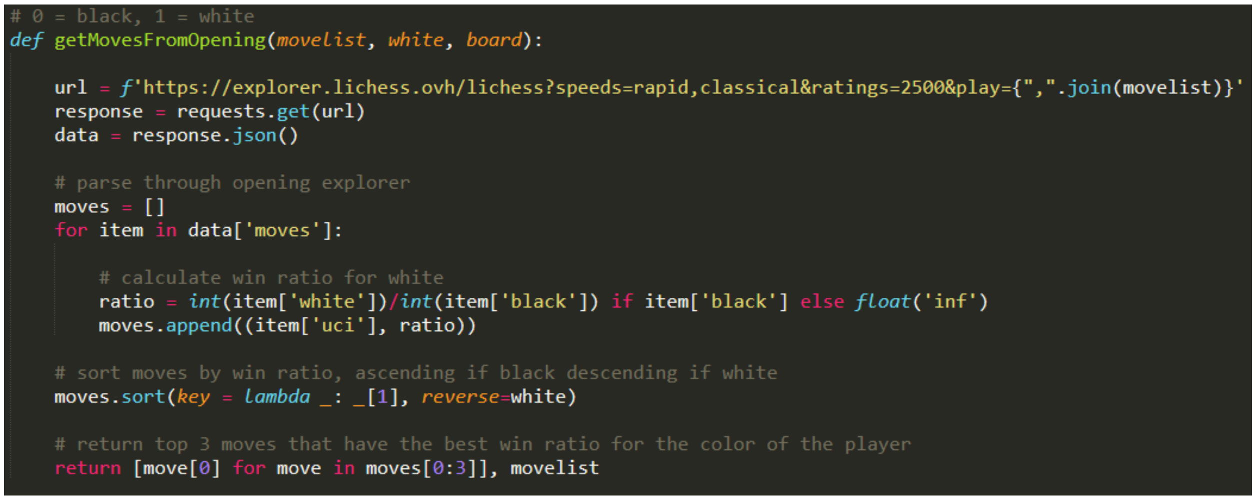
Avoided:

- Opening Books - Why parse strangely formatted book files and create a new database when it is much easier to use an existing one that is parsable via WebScraping with minimal effort?
- Neural Networks for position evaluation - Very difficult to implement/build properly, and requires extensive knowledge or experience using the networks publicly available. This method was attempted initially but did not seem to work properly
- Endgame Tablebases - Take up too much space on the computer for minimal return, alternative resource (stockfish) used for reasons explained in the results section.

Implementation

AI Opening:

This was probably the simplest part of the implementation of the project. It required no extensive research or manipulation, only simple data scraping. The code works by parsing a url (submitted as a specific formatted string containing all the moves of the game thus far) leading to a database of high level games on lichess.com, and obtaining the top 3 moves from that position (3 moves are used for variance in the resulting move) based on win ratio. The move to be used is then selected at random and played by the AI.



```python
# 0 = black, 1 = white
def getMovesFromOpening(movelist, white, board):

    url = f'https://explorer.lichess.ovh/lichess?speeds=rapid,classical&ratings=2500&play={",".join(movelist)}'
    response = requests.get(url)
    data = response.json()

    # parse through opening explorer
    moves = []
    for item in data['moves']:

        # calculate win ratio for white
        ratio = int(item['white'])/int(item['black']) if item['black'] else float('inf')
        moves.append((item['uci'], ratio))

    # sort moves by win ratio, ascending if black descending if white
    moves.sort(key = lambda _: _[1], reverse=white)

    # return top 3 moves that have the best win ratio for the color of the player
    return [move[0] for move in moves[0:3]], movelist
```

Figure 1: Opening Code

Code to get top 3 moves from a lichess database of high rated players using a list of moves played this game thus far.

AI Middlegame:

This was the most difficult and complex part of the implementation of the project. The idea behind the middlegame was to use code from an already-written engine in order to calculate the heuristics of a position. The method decided upon was PSTs for the reasons mentioned in the 'solutions used/avoided' section. Initially, sunfish was used due to its high rating of 2000. This proved highly problematic because the code takes many shortcuts in order to be high-performance while still remaining short (only 111 lines of code).

Each time the genetic algorithm using the pyGAD module was run on a new solution/child, the evaluation of sunfish from the previous solution/child would not properly reset. This resulted in only the first PSTs passed into the genetic algorithm being properly played out to calculate its fitness value, and all other games being read as a 'checkmate' after only a single move. Many hours were spent trying to fix this, but nothing seemed to work.

In order to solve this, sunfish was replaced with another chess engine called 'Andoma' also written in python. This had the added benefit of Andoma already utilizing the python chess module used in both the opening and endgame, but the drawback of Andoma's lower performance. In a test run against the original engine, I was able to best it despite my rating only being 1400-1700 depending on time control. However, the engine did not play horribly and the point of the project was to improve the PSTs used, so this was not a huge issue.

After modifying the code and passing the tables into pyGAD, a number of generations of PSTs ran through the genetic algorithm until a much better result was reached. Unfortunately, due to this engine utilization of minimax the response to a given move was always the same, so each PST was only pitted against the original once. However, this also means that once a superior PST was found, it would always beat the original in a game.

Minor Tweak

The one and only minor tweak made to the code during this run was to encourage the engine Andoma to capture pieces. Originally, the evaluation function was coded so that a higher value piece taking a lower value piece was bad, and vice versa was good. This seems alright on the surface, but in reality taking a lower value piece can still be good, it's just not as good. In no world is a queen capturing a free knight a 'bad move'. Thus, the code was tweaked from the original:

- Original: return valueCaptured - valueCapturing
    - As you can see the above would result in a negative (score reduction) for capturing a piece of lower value. This is not true.
- New: return (captured - capturing) if captured_more_valueable else (captured/capturing)
    - As you can see, an even piece value is 1 point, and anything where the piece captured is more valuable is still a high positive number.
    - Unlike the first implementation, when a lower value piece is captured, a small positive change between 0 and 1 occurs.
    - This encourages rather than discourages captures.

Fitness Function

The fitness function used was tweaked a number of times, and coming up with a good one was relatively difficult. The function used ended up relatively simple compared to the possibilities that existed, but seemed to work well. This is a possible area of expansion for the project if it were continued.

Function used:

- A PST would start off with total points equal to 0.
- The lower the points, the better the PST performed.
- Play a game until either draw, checkmate, or 80 cumulative moves are reached.
    - In the case of a checkmate, subtract 10 points
    - In the case of a stalemate or draw, do nothing

    - In the case of 80 move limit, increment based on stockfish evaluation:
        - eval 0-200: subtract 4 points if in favor of PST, or add if against
        - eval 200-500: subtract 7 points if in favor of PST, or add if against
        - eval 500+: subtract 9 points if in favor of PST, or add if against
- Add points to score equal to half the total number of moves. This is done because reaching checkmate after only 20 moves is better than reaching checkmate after 50 moves, since a faster checkmate can show strength up to a certain level.
- “Perfect play” was designated as a score of 5 points, assuming a checkmate in about 30 cumulative moves (-10 + 30/2 = 5)
- If the total number of points was somehow less than 5 (never happened), set the total score to 5 for “perfect play”
- The divisor was the abs(points - “perfect play”), so abs(points - 5). If there was a divisor smaller than 1 (perfect play), set the divisor to 1 for a perfect fitness value
- Fitness value = 1/divisor, with a value closer to 1 being the best.

After several tweaks to this algorithm and several separate runs, a solution was reached with a fitness value of .1666, which seems pretty bad, but is actually a checkmate in under 40 moves, so it is a great improvement over the original. These moves are cumulative moves, so the actual number of moves that PST made was under 20. The resulting PST can be found in the ‘Tables and Figures’ section.

Genetic Algorithm Parameters

- The fitness function described above to determine fitness
- 10 solutions/PSTs a generation
- 3 parents mating with a uniform crossover and a random mutation
- Keep 3 parents a generation
- Mutation values between -5 and 5, seeming appropriate for the values in the initial data.

AI Endgame:

This was the most interesting part of the project, with lots of data analysis and very entertaining games to watch, as well as problems to solve. Going into the endgame, research was done and it was determined that others have already solved all endgames with 7 pieces or less on the board. The question of what constitutes an endgame was still posed, because people are currently working on 8 piece tablebases, and they are most definitely generatable, albeit with a huge (possibly unrealistic) amount of memory required.

To answer this question, Stockfish was once again used. Stockfish was given a 100 move limit to lay out a position on a randomly generated, semi-equal position. The idea is that, if within 100 moves of perfect play, a position involving either checkmate, stalemate, or a known solved position (any position of 7 pieces or less) is reached, this must also be a ‘solved’ position.

This is very similar to how tablebases are generated, starting at a specific ending position and working backwards to find all possible preceding moves. Because Stockfish is the closest engine to ‘perfect play’ known at the time of the project, it is the engine used to play out the 100 moves on a board with a relatively equal evaluation.

The generation of legal, equal positions was very interesting to tackle. The following algorithm was used to generate the positions used to play these games:

Algorithm

Until a non-checkmate position of n pieces is found with an evaluation difference of no more than a minor piece (3):

- First assign both kings to the board.
- Then, for any given position, a random side was chosen to have its pieces ‘selected’ first.
- That side would then select a random piece out of possible chess pieces, and then the opposite side would receive a random ‘equivalent’ set of pieces.
- These pieces were randomly added to the board until there were n pieces.
- The position would then be checked for king-legality, where only 1 or 0 kings were in check, swapping the turn if the king currently in check was not set to move first.
- Finally, the position would be evaluated by stockfish to meet eval specs, and re-created if it did not pass.

Although this may not be the cleanest, most efficient algorithm, it was very fun and an interesting challenge to work on. In order to do this for varying piece counts, the following strategies were utilized:

Strategies

- Implementing a ‘balance dict’ containing a specific piece and all (reasonable) combinations of other pieces that are approximately equivalent in value.
    - For example, a queen is similar to a [bishop and two knights] or a [rook and knight and pawn]
- Implementing adjacency checking for kings.
- Returning a random board index from an initial number 0-63 due to potentially better variance.
- Converting this board into a fen for usage with the python chess module
- Creating a ui and an algorithm which would process user input for number of pieces and iterations, then write the resulting data into the correct csv file

The results showed that endgames seem to be solvable up to 12 pieces, but then the ratio of games reaching a known solved position seems to start to go down drastically.

Results

The instructions and full implementation of all the working parts can be found on https://github.com/gsytnik/Cal_Poly_Senior_Project and in the README.md file. ScreenShots of the program are provided in the 'Tables and Figures' section

Conclusion and Future Work

Modern chess engines are extremely complex and sophisticated pieces of software. To build a very simple chess engine of moderate strength that relies mostly on PSTs and Minimax, a lot of effort and research ws required. Two quarters of work went into the project despite large amounts of the code being existing code that was modified.

A lot of useful information about chess programming was also gained. Information such as endgames of up to 12 pieces being potentially solvable and ideal (or at least improved) PSTs can benefit chess programmers in creating better engines in the future. With more computational power, research, and time, The techniques used in this project can be expanded upon to create a very strong engine.

Many shortcomings also existed. Time was very constraining, preventing more challenging techniques such as opening books and neural nets from being used, which probably would have resulted in a stronger engine. Memory was also a constraint, preventing feasible usage of endgame tablebases. It also would have been beneficial to have more knowledge and experience with all of the techniques used in the project - I had no prior experience with genetic algorithms and limited knowledge of chess programming.

If this project was developed into a full product, many improvements could be made. The CLI can be turned into a full GUI, allowing players to see the board in a more user-friendly way. The engine itself can benefit from many additions and changes, such as finding the best possible fitness function for the PST genetic algorithm, more complex PSTs, position evaluation improvements, the addition of neural networks, tablebases, opening books, etc. These are all methods that are more time consuming and complex than was available within the scope of the project and my capabilities at the time.

References

General Chess Programming:

Chess Programming Wiki - https://www.chessprogramming.org/Main_Page
Syzygy - https://syzygy-tables.info/
TableBases - https://en.wikipedia.org/wiki/Endgame_tablebase
Why I avoided polyglot opening books (check top response) -
https://chess.stackexchange.com/questions/28874/how-to-use-polyglot-opening-book-bin-file

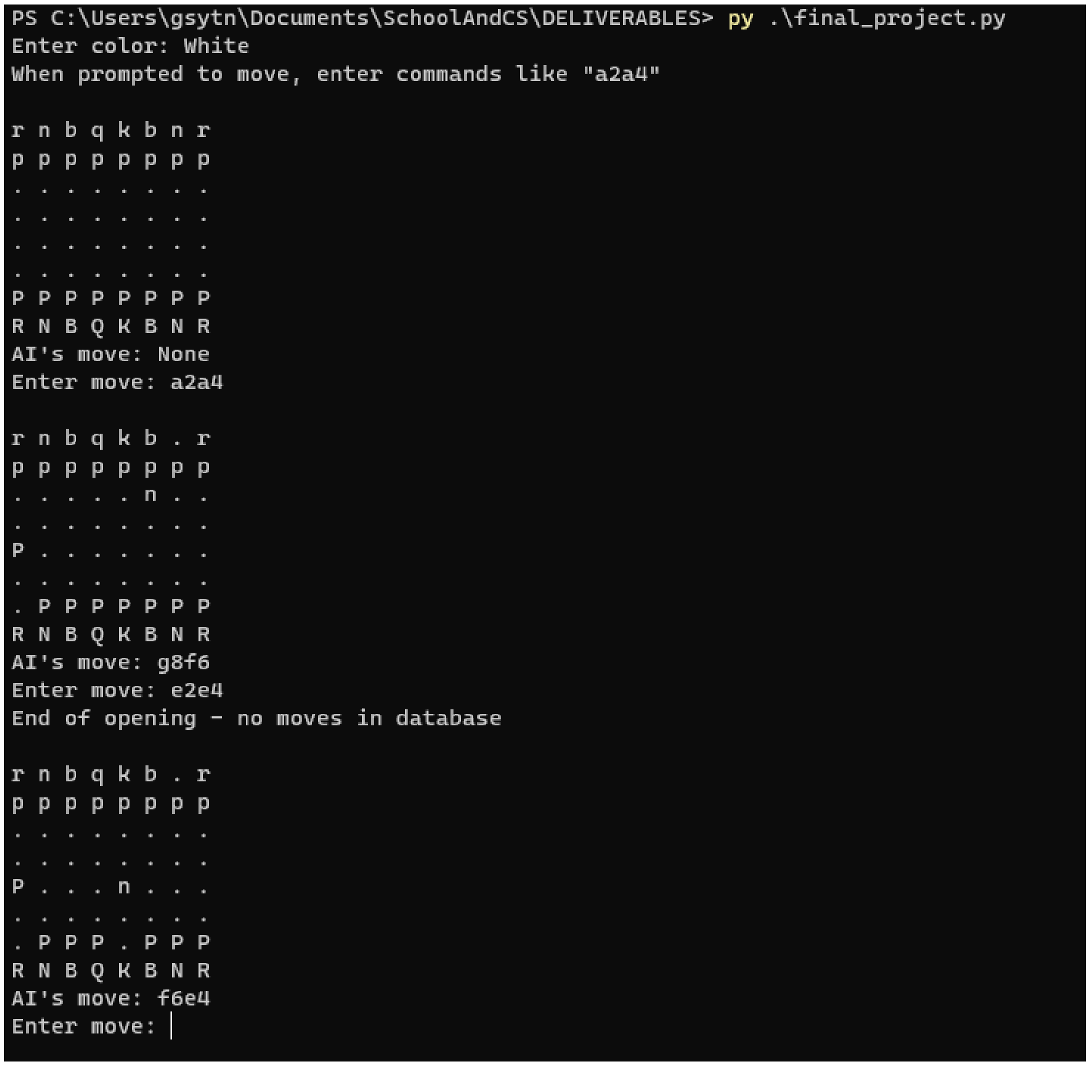
Documentation:

Py Stockfish - https://pypi.org/project/stockfish/
Py Chess - https://python-chess.readthedocs.io/en/latest/
PyGAD - https://pygad.readthedocs.io/en/latest/

Github Repos:

Sunfish - https://github.com/thomasahle/sunfish
Andoma - https://github.com/healeycodes/andoma

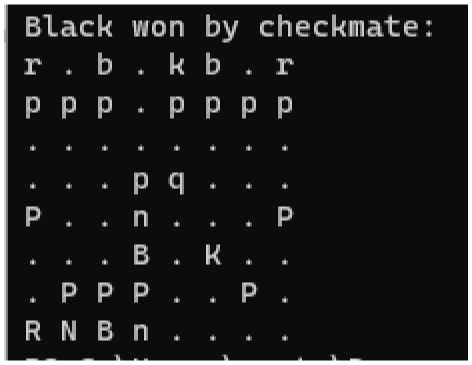
Tables and Figures



```
PS C:\Users\gsytn\Documents\SchoolAndCS\DELIVERABLES> py .\final_project.py
Enter color: White
When prompted to move, enter commands like "a2a4"

r n b q k b n r
p p p p p p p p
. . . . . . . .
. . . . . . . .
. . . . . . . .
. . . . . . . .
P P P P P P P P
R N B Q K B N R
AI's move: None
Enter move: a2a4

r n b q k b . r
p p p p p p p p
. . . . . n . .
. . . . . . . .
P . . . . . . .
. . . . . . . .
. P P P P P P P
R N B Q K B N R
AI's move: g8f6
Enter move: e2e4
End of opening - no moves in database

r n b q k b . r
p p p p p p p p
. . . . . . . .
. . . . . . . .
P . . . n . . .
. . . . . . . .
. P P P . P P P
R N B Q K B N R
AI's move: f6e4
Enter move:
```

Figure 2a: Example CLI

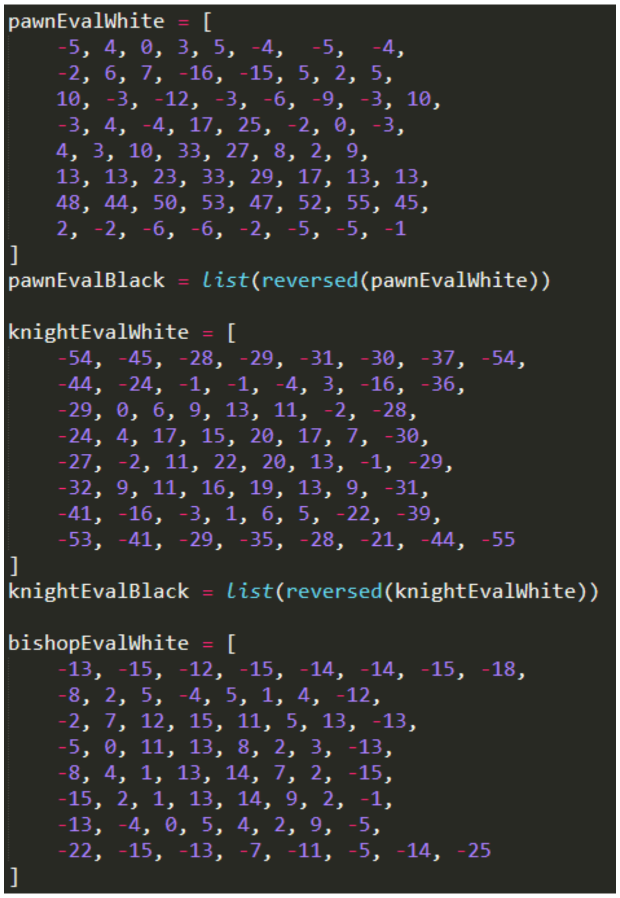
Example CLI of the early parts of the program



```
Black won by checkmate:
r . b . k b . r
p p p . p p p p
. . . . . . . .
. . . p q . . .
P . . n . . . P
. . . B . K . .
. P P P . . P .
R N B n . . . .
```

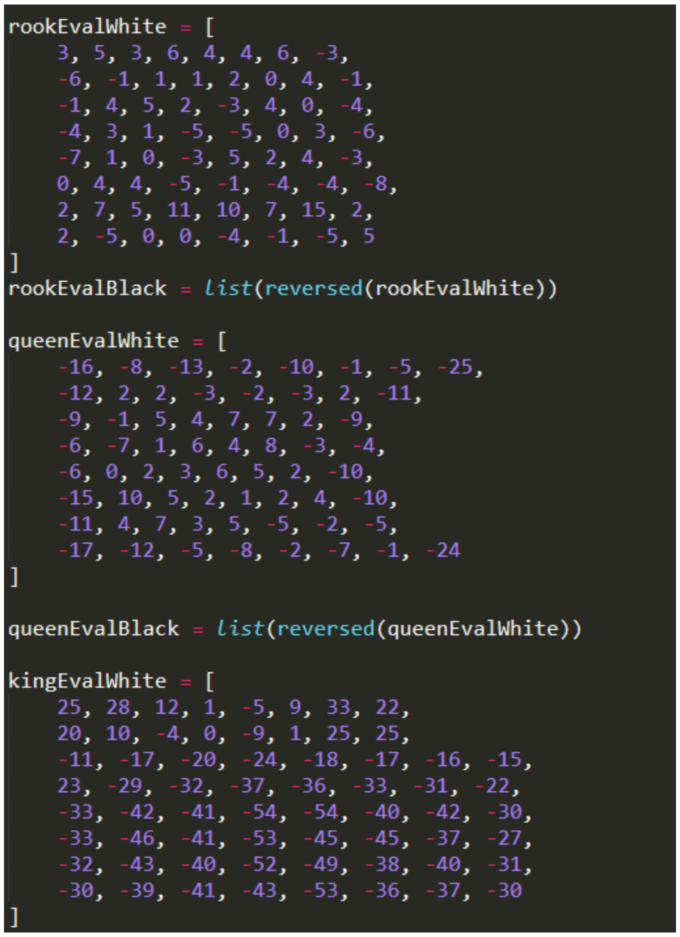
Figure 2b: Example CLI

Example CLI of the end of the program



```
pawnEvalWhite = [
    -5, 4, 0, 3, 5, -4,  -5,  -4,
    -2, 6, 7, -16, -15, 5, 2, 5,
    10, -3, -12, -3, -6, -9, -3, 10,
    -3, 4, -4, 17, 25, -2, 0, -3,
    4, 3, 10, 33, 27, 8, 2, 9,
    13, 13, 23, 33, 29, 17, 13, 13,
    48, 44, 50, 53, 47, 52, 55, 45,
    2, -2, -6, -6, -2, -5, -5, -1
]
pawnEvalBlack = list(reversed(pawnEvalWhite))

knightEvalWhite = [
    -54, -45, -28, -29, -31, -30, -37, -54,
    -44, -24, -1, -1, -4, 3, -16, -36,
    -29, 0, 6, 9, 13, 11, -2, -28,
    -24, 4, 17, 15, 20, 17, 7, -30,
    -27, -2, 11, 22, 20, 13, -1, -29,
    -32, 9, 11, 16, 19, 13, 9, -31,
    -41, -16, -3, 1, 6, 5, -22, -39,
    -53, -41, -29, -35, -28, -21, -44, -55
]
knightEvalBlack = list(reversed(knightEvalWhite))

bishopEvalWhite = [
    -13, -15, -12, -15, -14, -14, -15, -18,
    -8, 2, 5, -4, 5, 1, 4, -12,
    -2, 7, 12, 15, 11, 5, 13, -13,
    -5, 0, 11, 13, 8, 2, 3, -13,
    -8, 4, 1, 13, 14, 7, 2, -15,
    -15, 2, 1, 13, 14, 9, 2, -1,
    -13, -4, 0, 5, 4, 2, 9, -5,
    -22, -15, -13, -7, -11, -5, -14, -25
]
```

Figure 3a: Pawn, Knight, Bishop PSTs



```
rookEvalWhite = [
    3, 5, 3, 6, 4, 4, 6, -3,
    -6, -1, 1, 1, 2, 0, 4, -1,
    -1, 4, 5, 2, -3, 4, 0, -4,
    -4, 3, 1, -5, -5, 0, 3, -6,
    -7, 1, 0, -3, 5, 2, 4, -3,
    0, 4, 4, -5, -1, -4, -4, -8,
    2, 7, 5, 11, 10, 7, 15, 2,
    2, -5, 0, 0, -4, -1, -5, 5
]
rookEvalBlack = list(reversed(rookEvalWhite))

queenEvalWhite = [
    -16, -8, -13, -2, -10, -1, -5, -25,
    -12, 2, 2, -3, -2, -3, 2, -11,
    -9, -1, 5, 4, 7, 7, 2, -9,
    -6, -7, 1, 6, 4, 8, -3, -4,
    -6, 0, 2, 3, 6, 5, 2, -10,
    -15, 10, 5, 2, 1, 2, 4, -10,
    -11, 4, 7, 3, 5, -5, -2, -5,
    -17, -12, -5, -8, -2, -7, -1, -24
]

queenEvalBlack = list(reversed(queenEvalWhite))

kingEvalWhite = [
    25, 28, 12, 1, -5, 9, 33, 22,
    20, 10, -4, 0, -9, 1, 25, 25,
    -11, -17, -20, -24, -18, -17, -16, -15,
    23, -29, -32, -37, -36, -33, -31, -22,
    -33, -42, -41, -54, -54, -40, -42, -30,
    -33, -46, -41, -53, -45, -45, -37, -27,
    -32, -43, -40, -52, -49, -38, -40, -31,
    -30, -39, -41, -43, -53, -36, -37, -30
]
```

Figure 3b: Rook, Queen, King PSTs