ChessBot Battle

T Gerster
University of Lausanne
Lausanne, Switzerland

***Abstract*—This project aims to develop a Python-based chess program capable of grasping a sufficient understanding of chess dynamics to be able to beat a "random-move" player and potentially challenge low-rated opponents. Given the intricate nature of chess, the objective is to find ways for the program to comprehend, even remotely, the chessboard, its different pieces and their possible relationships. This paper proposes the development of evaluation methods to score chessboard positions. Two different approaches have been considered; a translation of the board state through static scoring (Advanced Programming) and through the training of a CNN(Advanced Data Analysis). The two resulting different ChessBots, respectively named Harold and Albert, are then tested through simulations.**

***Index Terms*—**

I. Introduction

"*When it comes to chess, computers have surpassed human grandmasters, but they will never outmatch the human creativity and intuition that make the game truly beautiful.*"
- Viswanathan Anand

Firstly, it is necessary to define a few key concepts in chess :

- Depth : When talking about the depth of search in chess programming, we refer to the number of moves ahead that we are analyzing, a depth of 2 would mean looking at all possible next moves and their subsequent responses.
- Positions : The second important concept it that we are going to consider a chess game as a sequence of static positions rather than a sequence of dynamic moves.
- Ply : A "ply" refers to a single move made by a player. It consists of one move by one player, such as moving a piece from one square to another on the chessboard.
- Elo : Chess Elo is a rating system used to measure the relative skill levels of players, based on their performance in chess games. To give a quick benchmark, an absolute debutant could be considered 400 elo and the world champion Magnus Carlsen is rated arround 2800 elo.

In the fascinating realm of chess programming, the journey towards creating intelligent and capable automated chess engines began in 1957 with a significant milestone. It was during this year that Alex Bernstein developed the very first automated chess engine after years of theoretical groundwork. This groundbreaking achievement marked the initial steps towards the development of chess engines.

At its inception, this early chess engine had limited capabilities compared to the modern behemoths we witness today. To put it into perspective, it took approximately eight minutes for the engine to perform a 4-ply search, meaning it could analyze four moves ahead in the game (commonly referred to as depth in today's engines). Despite its humble beginnings, this creation paved the way for future innovations in chess programming.

Fast forward to 1997, a defining moment arrived when IBM's Deep Blue became the first chess engine to triumph over a reigning world chess champion. Deep Blue's historic victory over Garry Kasparov, a grandmaster of unmatched brilliance, demonstrated the immense potential of chess engines and their ability to challenge even the most formidable human players. This groundbreaking event captured the world's attention and sparked a new era of chess programming.

Over the years, chess engines have made tremendous progress, propelled by a combination of algorithmic advancements and the exponential growth of computational power. Several popular algorithms have emerged throughout history to facilitate chess engine development. One notable algorithm is the minimax algorithm with alpha-beta pruning. This algorithm, introduced in the 1950s, helps engines evaluate different move sequences by recursively exploring potential moves and their resulting positions, maximizing their own gain while minimizing the opponent's advantage.

Another crucial aspect of chess engine development lies in board evaluation. To determine the value of a particular board position, engines employ various evaluation functions. Throughout history, different methods have been used, ranging from simple material counts to more complex techniques. Recent breakthroughs have seen the integration of Convolutional Neural Networks (CNNs) into evaluation systems. These deep learning models analyze the board state, leveraging their ability to identify patterns and strategic concepts, providing engines with an enhanced understanding of the game and enabling them to make more sophisticated decisions.

Among the modern chess engines that reign supreme in today's competitive landscape, Stockfish has emerged as a dominant force with an estimated elo of over 3500. Renowned for its exceptional strength and analytical prowess, CPP-based Stockfish engine has consistently outperformed its rivals in various chess competitions and matches. Its success can be

attributed to a combination of advanced algorithms, intricate board evaluation techniques, and the continuous efforts of a dedicated community of developers and contributors. As chess engines continue to evolve, we witness an ongoing pursuit of excellence in the realm of chess programming.

As for the chess programmation in python; while it is not a very popular language for high performance engines(the choice generally tends to be more in favor of compiled language), one amateur-designed bot, Sunfish, still managed to reach about 2000 elo rating.

II. Research Question

A. Problem

There are several key issues involved in designing a successful chess program. Firstly, it is essential to define how the program will behave and how it will be able to make decisions about which moves to play. Very much like what we would do as human, it is going to be necessary for the program to try and look into the possible future moves of the adversary. This central problem has led to the need to find methods that allow the program to iterate over the tree of possibilities and prioritise the results of these different moves.

When examining this question, it quickly became clear that the optimal solution would be for the program to be able to exhaustively explore the game tree and evaluate all the possible moves. Sadly, this realisation comes with the fact that exploring all possible options is exponentialy costly in terms of computational power; indeed in big-O notation the problem is O(b^d) in complexity where d is the depth at which the algorithm searches and b represent all the possible moves at each nodes. In terms of hard number, this means that if there is 30 possible moves on average at each ply, then at depth 3 there is already arround 27 000 possibilities. As for a comparaison, stockfish can go up to a depth of 28, exploring tens of millions of positions per seconds.

In my case, I realized that computional power and processing time were going to be the problem if I wanted to run an algorithm that could go to a meaningful depth. Since I have decided to work in python, I also need to account with the limitation that this choice of programming language will incure. Furthermore, looking into future positions requires the choice of an appropriate search algorithm to navigate through the tree of possibilities.

But while it is important that the program is able to search for future positions, it is also crucial to determine how to evaluate and prioritise these positions. Evaluating positions is a complex task and there is sadly no universal mathematical formula or algorithm that scores chess positions. Choosing an appropriate evaluation method is an essential challenge since every positions that are going to be explored by the program are going to be rated by it. Again, the question of the complexity of the evaluation function needs to be raised, if it needs too much time to return a score for each position, it's going to further slow down search process of the program. In other chess engines, a lot of different approaches have been considered; ranging from static evaluation functions based on prior human knowledge to more advanced techniques using deep convolutional neural networks and machine learning.

B. Objective

The two mains dimensions of the problem have been identified and we now need to implement two methods :

- a **Search method** (with depth as main parameter)
- an **Evaluation method** (with the different pieces locations on the board as main parameter)

After considering different options to answer both problems, I have decided to focus my research on the way my program would evaluate the different chessboard positions rather than the optimization of the search method that would, even at low depth, be limitated by python and available computational power.

Our main objective is then to be able to score chess positions in a way that translates who currently has the advantage and by how much (developing the Evaluation method). I have considered two different approaches :

- A static Evaluation method (Advanced Programming) based on the observation of the pieces on the board at a given turn, which requires a number of strategic factors to be taken into account. These factors help to determine the relative advantage between the players and influence the programme's decisions.
- A CNN (Convolutional Neural Network) method (Advanced Data Analysis) based on the training of specifically designed model to look at chess positions through 7 channels and try to predict Stockfish evaluations of board positions.

Our secondary objective will be to implement a search method; meaning choosing an algorithm that iterates through the possible moves at a chosen depth while being able to compare those multiple possibilities. The most interesting candidate is the minimax algorithm with alpha-beta pruning, it is often used in chess programming and can be easily implemented in python.

Our last objective will be to propose one or several Python-based programs that allow the player test to test the bot and to play against it.

III. Methodology

The first step in the project was to find a way to play chess in Python without having to implement the entire dynamics of the game. For this purpose, the Python-Chess library was selected due to its various features and functionalities.

The Python-Chess library provides convenient methods to handle chess-related tasks such as handling moves and boards,

reading and writing PGN (Portable Game Notation) files, and interacting with chess engines like Stockfish. It offers a wide range of useful functions and utilities that simplify working with chess games and positions.

A. The evaluate-board function

It is necessary to find a way to translate a static chess position into a numerical value representing the state of the game. In order to do so, I have come up with two solutions (respectively designed to translate an approach of the problem from the point of view of the two courses).

Static evaluate-board function (Advanced Programming) : I have decided for the evaluation function to employ a set of seven factors to try and accurately assess the chessboard position. Each factor contributes to determining the overall score of the position, indicating a positive score if the board is in favor of White and a negative one if it is in Black's favor. Those variables are (for each player); the material on the board, the mobility of the pieces, the relative threats facing the king, the relative control of the center of the board (D4,D5,E4,E5), the use of the piece-square tables(that assign a value to specific pieces on specific squares of the board), as well as bonuses for pawn structure and promotions. It is important to note that I have allocated different weights to those variables, in order to try and adapt the scoring more efficently through testing.

CNN evaluate-board function (Advanced Data Analysis) : I have designed a CNN with a specific architecture designe to read seven channels (1-6 : each type of pieces, pawns, knights, bishops, rooks, queens and kings, 7th : whose turn it is to play). The model training takes a nx8x8x7 array (representing positions and channels), which all have been associated with a Stockfish evaluation of the board (array containing 1 value).

B. The minimax function

The minimax algorithm is a decision-making algorithm used in two-player games, such as chess. It aims to determine the optimal move for a player by considering all possible moves and their outcomes. The algorithm evaluates each move by recursively exploring the game tree, considering both the player's moves (maximizing player) and the opponent's moves (minimizing player). Alpha-beta pruning is an optimization technique used in conjunction with the minimax algorithm to reduce the number of nodes explored. It involves maintaining two values, alpha and beta, which represent the best known scores for the maximizing and minimizing players, respectively. During the search, if a node's score exceeds the beta value for the minimizing player or falls below the alpha value for the maximizing player, further exploration of that node can be stopped because it will not affect the final decision. This pruning allows the algorithm to disregard branches of the game tree that are unlikely to lead to an optimal move, significantly improving its efficiency. It is also important to choose a structure that allows the user to choose the depth for the method in order to reduce complexity and processing time easily.

C. Putting the methods together

When both methods are implemented, there is the matter of creating a python class for each evaluation method combined with the minimax algorithm (classes are named Harold for the advanced Programming bot and Albert for the Advanced Data Analysis bot) that can return the chosen move. One code will handle the implementation of both bots into an interactive program that allows the human player to play versus Harold and Albert. The programm will implement a modest graphic display of the board at each moves.

Additionally, I will design a program that allows the user to test the bot against the loaded stockfish engine (with the python-chess library) or play games against simulated "random-move" player.

IV. Implementation

A. Bot : Harold class (Advanced Programming)

```
class Harold:

    ...
    all helper functions
    ...

    function evaluate_board:
        if the game is over, return corresponding score
        call evaluate_material
        call evaluate_pst_and_material
        call evaluate_mobility
        call evaluate_king_safety
        call evaluate_center_control
        call evaluate_pawn_structure
        call evaluate_pawn_advance
        combine all scores to compute final board score
        return board score

    function alphabeta:
        if depth is 0 or game is over, call evaluate_board and return score
        if it's bot's turn, maximize score
        if it's opponent's turn, minimize score
        return best score

    function best_move:
        get move from opening book if possible
        if not, call alphabeta for each legal move
        return best move
```

Fig. 1. Pseudo-Code of the class

Firsty, lets explain how the class works as a whole. When the best-move() method is called in the Harold class, the following steps are performed. The function attempts to retrieve a move from the opening book using a book reader. If a move is found, it is printed as the "Book move" and returned as the best move. If no book move is available or an exception occurs, the function proceeds with the main move selection process.

Next, the function initializes variables to track the maximum evaluation score (max-eval) and the best move (best-move). It iterates over all legal moves for the current board position. For each legal move, it applies the move to the board and

evaluates the resulting position using the alphabeta() method. This method performs the minimax algorithm with alpha-beta pruning to evaluate different board positions and returns the evaluation score.

After evaluating each move, the function compares the evaluation score with the current maximum evaluation (max-eval). If the score is greater than max-eval, it updates max-eval with the new score and updates best-move with the current move. Once all legal moves have been evaluated, the function has determined the best move based on the highest evaluation score. It prints the maximum evaluation score and the calculation time before returning the best move found.

In summary, the best-move() method goes through each legal move, applies the move to the board, evaluates the resulting position using the alphabeta() method, keeps track of the move with the highest evaluation score, and returns the best move found. It utilizes the minimax algorithm with alpha-beta pruning to efficiently search the game tree and determine the move with the highest evaluated score.

Let's look closer at the evaluation-board() function [1] [5], which calculates the overall score for the current board position. It combines several evaluation factors, assigns weights to each factor, and computes a weighted sum to determine the final score for the fed position. For each variable a positive value means advantage for White and a negative score represents an advantage for Black :

1) *mobility* : This variable measures the mobility of the pieces. It counts the number of legal moves available for each side (white and black) and calculates the difference between them.
2) *material* : This variable calculates the material score by summing up the values of all the pieces on the board. It iterates over the piece map and uses the predefined piece values to assign a numerical value to each piece.
3) *pst-scores* : This variable represents the contribution of the piece-square tables (PST) to the score. It iterates over all squares on the board, checks if a piece is present on a square, and adds the corresponding PST value for that piece-square combination. The PST values are predefined and vary based on the piece type and square position.
4) *king-safety* : This variable evaluates the safety of the kings. It counts the number of attacking pieces near the opponent's king and calculates the value of those attacks based on the piece values. The score is weighted by the number of attacking pieces and the predefined attack-weights.
5) *center-control* : This variable measures the control of the central squares on the board (E4, E5, D4, D5). It assigns a positive value for white if a white piece controls a central square and a negative value for black if a black piece controls a central square. The magnitude of the value depends on the type of piece controlling the square.
6) *pawn-strcture* : This variable evaluates the pawn structure on the board. It considers factors such as isolated pawns, doubled pawns, and pawn chains. It iterates over all pawns on the board, checks for each pawn if it meets the criteria for these pawn structure elements, and assigns a positive or negative value accordingly.
7) *pawn-advancements* : This variable checks if the board in in endgame(less than 13 pieces on the board), and if it is the case, returns a better value each time a pawn goes forward on the board. This is meant as a bonus to valuate promotions of pawns when possible.

Once all these variables are calculated, the final score is obtained by combining them using predefined weights (c1, c2, c3, c4, c5, c6) and the pawn-advancement bonus. The resulting score represents the overall evaluation of the current board position.

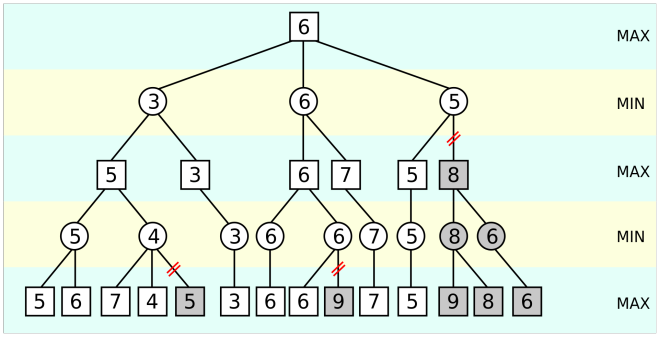

Fig. 2. Alpha-beta pruning minimax algorithm (*source: wikipedia*)

As for the alphabeta() algorithm [2] [3] [4], it is implemented to perform a minimax search with alpha-beta pruning. It takes in the current alpha and beta bounds, as well as the depth parameter to control the search depth in the game tree. The algorithm starts by checking the base cases: if the depth reaches 0 or the game is over, it returns the evaluation score of the current board position. The evaluation score is calculated using the evaluate-board() method, as explained before.

If it's the bot's turn (maximizing-player), the algorithm initializes max-eval to a very low value and iterates through all legal moves. For each move, it applies the move to the board, calls alphabeta() recursively with updated alpha and beta bounds and decreased depth, and stores the returned evaluation score in eval. It then updates max-eval with the maximum value between max-eval and eval. Additionally, it updates the alpha bound with the maximum value of alpha and eval.
If it's the opponent's turn (minimizing-player), the algorithm follows a similar process, but it initializes min-eval to a very high value and updates min-eval and the beta bound with the

minimum value between min-eval and eval.

Throughout the search, the algorithm prunes branches of the game tree by comparing alpha and beta. If at any point beta becomes less than or equal to alpha, it breaks the loop and returns the current evaluation score. Finally, the algorithm returns max-eval if it's the bot's turn and min-eval if it's the opponent's turn. The returned value represents the best evaluation score that can be achieved from the current position within the specified search depth.

B. Bot : Albert class (Advanced Data Analysis)

This section is meant as informative since I have developed this class for the Advanced Data Analysis project; I will then just briefly explain its training and design since an example of the class will be imported in the ChessBot Battle program. The class uses the same architecture as the Harold class with finding the best-move and using the alphabeta() algorithm, but deviates in its evaluation function. Indeed I have trained a convolutional neural network on about 10.5 millions chess positions with their associated stockfish evaluations (at depth = 7, approximately 2000 elo) that I then import in the ChessBot Battle program to initate the Albert class. This class is not optimized at this date and its minimax algorithm has a fixed depth of 1 to limitate processing time.

C. ChessBot Battle program

The "ChessBot Battle" program is an interactive chess game implemented in Python. It allows users to play against different chess bots named "Harold" and "Albert." The program starts by loading a pre-trained model for the "Albert" bot. Users can then choose to activate the random mode, where he will be able to watch to bot play versus a random-move generator. Then they will be asked to choose their preferred color (White or Black) and the opponent they will face. The game proceeds with alternating turns between the player and the bot.

For the player's turn, the current state of the chessboard is displayed, and the player can input their moves. If it's the bot's turn, the corresponding bot selects the best move using either a specified depth for "Harold" or a default depth of 2 for "Albert." The game continues until the chessboard indicates that the game is over due to checkmate, stalemate, insufficient material, or other conditions. At the end of the game, the result is displayed along with the last board representation.

This is accomplished by the use of just a few functions:

1) *display-board* : This function displays the current state of the chessboard using SVG format.
2) *get-user-move* : This function allows the user to input their move and returns the corresponding chess move object.
3) *check-game-status* : This function checks the current game status and returns True if the game is over due to checkmate, stalemate, or other draw conditions.
4) *main* : Implements the main logic of the program, including initializing the game, handling user inputs, and controlling the game flow.

D. Bot Testing program

The program consists of two main functions: test-random-player and test-stockfish, along with a supporting function for-elo.

The test-random-player function simulates chess games between a bot (Harold) and a random player. It plays a specified number of games (nb-sim) and records the results. The function alternates between the bot playing as White and Black. During each game, the bot generates moves using its best-move function, while the random player selects moves randomly from the list of legal moves. The games are saved in a PGN file, and the results are stored in a results list.

The test-stockfish function simulates chess games between the bot (Harold) and Stockfish, a strong chess engine. It follows a similar structure to test-random-player, but instead of a random player, Stockfish is used as the opponent. The Stockfish engine is initialized with a specified Elo rating, and it generates moves using the UCI protocol. The games are saved in a PGN file, and the results are stored in a results list.

Both test functions create a .pgn file (portable game notation) and use it to store the results of the simulations. Those .pgn games can then be copied in most chess website to visualise the game.

The for-elo function provides a higher-level interface to conduct simulations over a range of Elo ratings. It takes inputs such as the depth of the bot's search, the number of simulations per Elo rating, the starting and ending Elo ratings, the bot's name, the file paths for Stockfish and the model, the book open, and the time limit for Stockfish's moves. It utilizes the test-random-player or test-stockfish functions based on the user's choice (random player or Stockfish) and runs simulations for each Elo rating in the specified range. The results for each Elo rating are stored in a dictionary, where the Elo rating serves as the key.

E. Notes on parrallel programming

While I have explored the possibility to parallelize the minimax algorithm, it presents several difficulties in Python. The Global Interpreter Lock (GIL) limits the potential parallelism by allowing only one thread to execute Python bytecode at a time. Additionally, not all Python libraries used in the algorithm may be thread-safe, which can lead to issues when running in a multi-threaded environment. Synchronization and managing shared resources, such as the game state and transposition table, require

careful consideration to avoid race conditions and data inconsistencies. Moreover, the granularity of parallelism in the recursive and sequential nature of the minimax algorithm poses challenges in evenly distributing the workload among threads. Furthermore, the overhead and scalability of parallel execution may outweigh the performance benefits for smaller or less complex game trees.

Considering those difficulties, I did not implement parrallel programming in the Harold class.

V. Maintenance and Update

I have not used a Git repository to handle this project, I have instead used the following principles to try and maintain my codebase locally :

- Code Organization: Keep your codebase organized and modular. Divide your code into logical modules, packages, or classes to improve readability and maintainability.
- Naming Conventions and Documentation: Follow consistent naming conventions for variables, functions, and classes. Use meaningful names that accurately describe their purpose. Add comments and docstrings to explain the functionality, input, and output of your code.
- Testing: Implement notebooks to test for critical components or functions. Write test cases to verify the expected behavior of your code and ensure that it functions as intended.
- Backups: Regularly back up your codebase to prevent data loss. Maintain copies of your project directory in secure locations or use backup tools to ensure you have a backup in case of any accidental data loss.
- Environment: The use of google colab to set up a work environment while having access to better RAM disponibilities and even GPU to accelerate the processus.
- A README.txt file : Instructions for the users.

VI. Results

A. Harold versus a "random-move" player

I conducted several tests at different depths of evaluation against the random player to assess the performance of my chess bot. The processing time increased as the depth of evaluation increased, as shown in fig.3

Harold's Depth :	*Processing time of best_move() :*
1	0.01 seconds
2	0.5 - 3 seconds
3	5 - 40 seconds
4	60 - 360 seconds
5	60 seconds - 20 minutes

Fig. 3. Processing time at different depths

Despite the depth limitation due to the processing time of our bot, the results were promising. In fact, my bot consistently outperformed the random player, achieving a high win rate, as depicted in fig.4. These findings indicate that my bot has the capability to make some strategic moves and exhibit at least some understanding of chess positions.

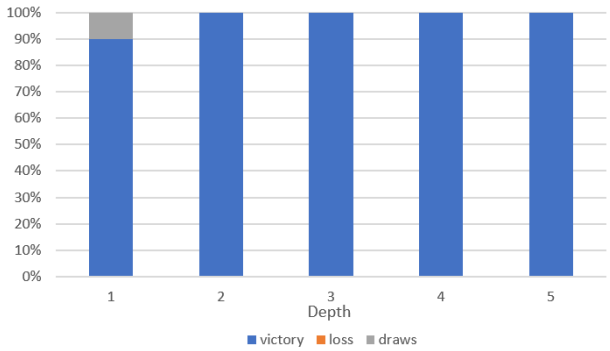

Fig. 4. Wins/Loss/Draws at different depth versus the random player

As we can see, the bot is not able to win every times against the random player when the depth is only set a 1. Over the 50 simulations I made, the bot drew ten percent of the time. However it wins everytime at higher depth which indicates at least some performances from my evaluation function.

B. Harold versus Stockfish (elo = 1350)

The objective was to find out if Harold was able to beat low elo players (around 400), but sadly I did not find a way to incorporate an engine that could simulate such a rating. Instead I used the stockfish engine with the minimum elo I could set: 1350. This is already the elo of someone who has played a lot of chess in his life and it indeed proved to be too much of a challenge for Harold. At any given depth (3 and 4), my bot was consistently beaten by stockfish; over more than 100 simulations over different depth, the bot successfully drew one time, losing every other game.

It however took a substential amout of moves from stockfish to achieve the wins (more that 45 moves on average and sometime almost 70) and this shows that it puts up a good fight. Another notable result is that (like shown in fig.5), the bot was able to predict the checkmates that he would take in a few turns (defined by depth) in advance. This shows clearly the capacity of the bot to anticipate.
When analyzing each game individually, it becomes evident that Harold demonstrates a correct understanding of the early game, making strategic moves and developing a favorable position. However, as the game progresses and the number of pieces on the board decreases, Harold encounters significant challenges in navigating the late game. It appears to struggle with accurately assessing the value of remaining pieces, planning ahead, and making optimal decisions in complex and limited material situations.

C. Estimate Harold's performances

Since it is not possible to precisely rate our bot in term of elo (it would need to compete against human beings),

```
h2g3
h3h2
Score : -2616.9
Calculation Time: 5.684162855148315 seconds
g3g4
h8h4
Score : -2611.9
Calculation Time: 3.2598776817321777 seconds
g4f5
h4e4
Score : -9999
Calculation Time: 1.5701522827148438 seconds
None
e4b4
Score : -9999
Calculation Time: 1.5416979789733887 seconds
None
h2h1q
Score : -9999
Calculation Time: 2.2886767387390137 seconds
None
c3c2
Score : -9999
Calculation Time: 1.5070767402648926 seconds
None
b4f4
Score : -9999
Calculation Time: 0.9553887844085693 seconds
None
c2c1q
```

Fig. 5. Checkmates prediction versus Stockfish

we will try to estimate its performance using the analyzing tools on chess.com. I have fed .pgn games generated by my testing program to chess.com and observed the results. On some games the bot yields move precision almost as good as stockfish (elo=1350)(fig.6) and don't seem to make major mistakes. We can see on fig.7 that the bot performs better

Fig. 6. Move precision rated by chess.com (White is stockfish and black is Harold)(*source: chess.com*)

during the early game, managing to even take an advantage on stockfish during that phase, but for the rest of the game its performance decreases.

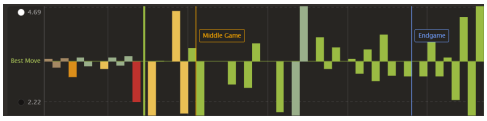

Fig. 7. Best move per side relative to game stage (White is stockfish and black is Harold)(*source: chess.com*)

Fig. 8. Interesting position from a game versus stockfish(*source: .pgn read on chess.com*)

D. ChessBot Battle program

As a last note, I would like to point out that the result of the ChessBot Battle program is satisfying, it upholds its objective and proposes an easily readable environment for the human player to interact with(fig.9 - 10).

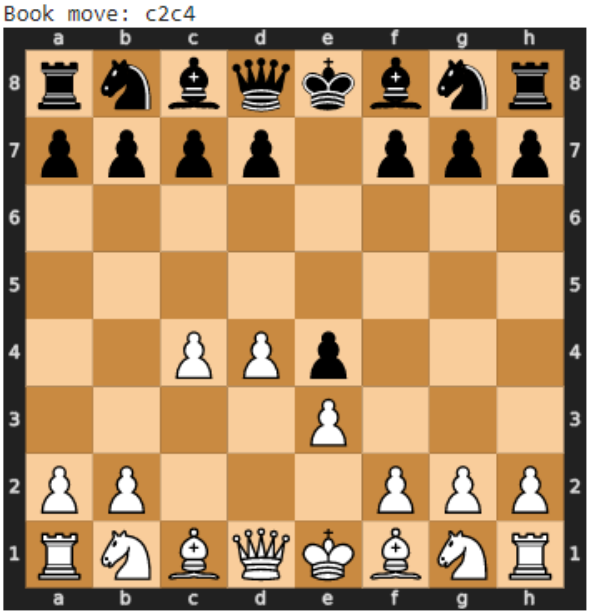

Fig. 9. ChessBot Battle prints

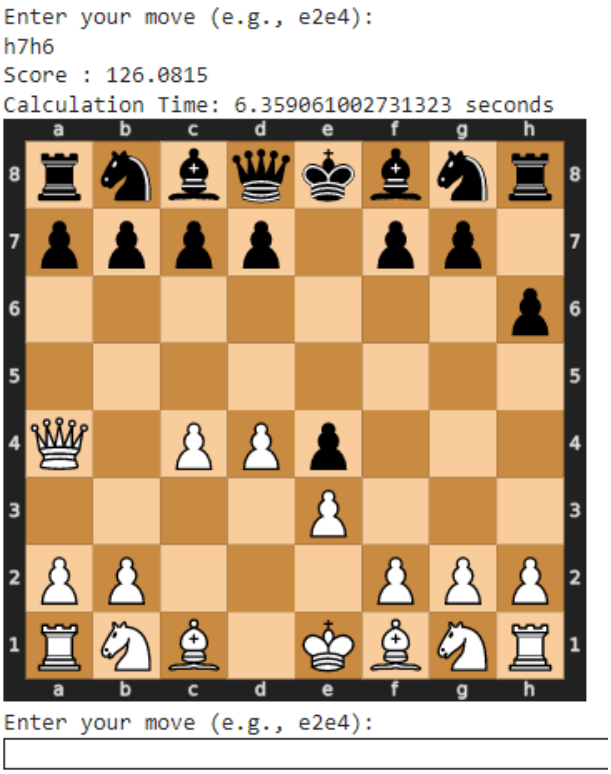

Fig. 10. ChessBot Battle prints

VII. Conclusion

In conclusion, the project aimed to develop a Python-based chess program capable of challenging opponents and exhibiting an understanding of chess positions. The ChessBot Harold was tested through simulations. Harold consistently outperformed the random player and showed potential in strategic moves during the early game. However, challenges arose in the late game, affecting Harold's decision-making. In the matchup against Stockfish with an Elo rating of 1350, Harold struggled to secure victories but put up a good fight, often taking substantial moves from Stockfish to achieve wins.

Acknowledging the complexity of chess, it is important to consider the limitations faced in achieving higher depths of evaluation and competing against higher-rated opponents like Stockfish. Professional chess engines employ advanced algorithms and optimized evaluation functions to efficiently explore the vast search space. The static evaluation function of stockfish has been in development for years using the cumulated knowledge of countless profesionnal players and software engineer and is evidently able to evaluate the board with much more accuracy.

I have played myself several games against the bot and it definitely demanded me some thinking time in order to defeat it. While he was sometimes sacrifying pieces, he was able to punish my mistakes if they were too big of a blunder.

Enhancements to the program could focus on improving late-game decision-making, refining the evaluation function to account for those different stages of the game. One of the most pertinent improvement would probably be the transformation of the minimax algorithmic search into a cpp program, hopefuly leading to better performances (in terms of speed).

Regarding the inherent complexity of chess and the challenges involved in developing competitive chess programs, it is essential to recognize the computational power and sophisticated algorithms employed by professional chess engines. These engines leverage advanced search techniques and optimization methods to efficiently analyze the vast number of possible positions and moves.

While Harold exhibited potential against lower-rated opponents, achieving competitive strength at higher levels requires further improvements. These may include the incorporation of advanced algorithms, refinement of the evaluation function, and the exploration of parallel processing or distributed computing techniques to enhance computational capabilities.

Developing a chess program that can consistently challenge strong opponents, both engines, and human players, is an ongoing endeavor. Continuous learning, research, and refinement are necessary to narrow the gap between the capabilities of developed ChessBots and state-of-the-art chess engines employed by professionals. And as this program may be considered simple in regard to the complexity of modern engines; it is, in my opinion, a successfull attempt to create a program that can comprehend the basics of chess.

I gladly invite you to try it for yourself !

References

[1] https://www.chessprogramming.org/Evaluation
[2] https://www.chessprogramming.org/Minimax
[3] https://healeycodes.com/building-my-own-chess-engine271–350.
[4] Diogo R. Ferreira, "THE IMPACT OF SEARCH DEPTH ON CHESS PLAYING STRENGTH", published, http://web.ist.utl.pt/diogo.ferreira/papers/ferreira13impact.pdf
[5] https://chessfox.com/example-of-the-complete-evaluation-process-of-chess-a-chess-position/
[6] https://openai.com/product/chatgpt