Chessy: A Chess Engine

Group Members:
Ali Rahbar
Wilton James Miller
Christian Fisla
Daniel Rafailov

Project Overview

Chessy is a fully implemented software created for users to play chess with friends or with bots. This project is being coded for our CSC111 class to demonstrate our understanding and mastery of the course content.

Objective

To develop a chess engine that utilizes a tree structure to map out possible moves ahead (e.g., depth 4 for four moves ahead) and uses an algorithm to evaluate each move based on board positioning and piece values.

Scope

1. Graphics User Interface: This is what you see on the screen when you play. It's where the chessboard and pieces are displayed, and where you make your moves.
2. Chess Logic Folder: Inside this folder, we wrote the code that controls how the game works. It's like the brain behind the game, making sure the pieces move correctly and following all the rules.
3. Bot Chess Player: We also made a computer player you can play against. It's like having a virtual opponent when you want to practice on your own.

Resources Used

Knowledge-based resources:

- Chess engine article from Cornell University: `https://www.cs.cornell.edu/boom/2004sp/ProjectArch/Chess/algorithms.html`

- Chess Engine in Python by Eddie Sharick: `https://youtu.be/EnYui0e73Rs?si=PV5_Up-VEutjJb9Q`
- Graphics (Open source): `https://opengameart.org/content/madwares-chess-set`

Note: There were no datasets used in this project. The data that the tree works with is dynamically generated as the game progresses to save memory and make the program more efficient.

Developer tools:

- OpenAI's Chat GPT (Helped with writing comments and reports)
- Github Copilot (Developers coding assistant)
- Github (Teams version control system)
- Jetbrains Qodana (Analyze software vulnerabilities)

Timeline (3 Weeks)

Week 1: Setup and Initial Development

- Setup project environment using the python chess library (temporarily)
- Start developing the tree structure for mapping moves.

Week 2: Setup software architecture and create user interface

- Develop and test the evaluation algorithm.
- Created the User interface
- Integrated the UI with basic tree and fix compatibility issues

Week 3: Develop chess logic and minimax algorithms

- Develop the logic for each piece in the game
- Implement the minimax algorithm so the bots can play instead of players

How to run the program

1. Install the required packages in the requirements.txt file:
   `pip install -r requirements.txt`
2. Choose the mode you like to play in the `game/game.py` file:

- If you wish to play player vs player:

```
# define player states
is_player_white_human = True
is_player_black_human = True
```

- If you wish to see the bots play against each other:

```
# define player states
is_player_white_human = False
is_player_black_human = False
```

3. Run the `main.py` file in the directory and the code will start working.

About Software:

In this project we explored many different areas and had to work on so many different things. Here is a list of the most advanced stuff we implemented:

Algorithms:

In this project we were required to develop and implement many advanced algorithms. Here are a list of the main ones:

1. The Negamax Algorithm: This algorithm is the advanced version of minimax that we have used in the tree to go through all the possibilities, minimize the opponent's score and maximize its own score. The Negamax algorithm plays the core part of the decision-making process of the chess bots.

2. The king is check algorithm: This algorithm checks the king's position and whether or not the king is in check. The problem with this algorithm was its efficiency as it was being called multiple times in each turn.

3. Pieces get move algorithms: Each move on the chess board also had a get moves method which would generate the possible moves that algorithm could make. All pieces in the code inherit properties from the `chess_piece` class which has a not implemented method called `get_moves`. All the pieces on the board implement this method with their own special algorithm which generates how they can move. Coding these was also a major challenge of the project.

Graphics:

The graphics of the game were also implemented using a layered structure where smaller methods each generate a layer of content on top of each other to generate

the user-friendly easy-to-use interface. The layers are as follows from the top to bottom:

1. Displayed text
2. Chess pieces
3. Movement highlights
4. Board graphics

This structure allowed us to create a smooth and dynamic user experience as every time the page would update only the related section would change making it more efficient. Note that the order the layers were generated were from number 4 to 1.

Working with data in the prediction tree:

- The `move_finder_tree` is a tree structure that records all the possible moves for all the possible players so the Negamax algorithm can recursively go through it and find the best move. The original goal of this project was to load a dataset of 3.5 million games into the tree to evaluate the moves from there. Due to the limited computing power we had we decided to generate the next moves dynamically after each move.
- The tree has this setting value called `DEPTH`. If depth is 4, the tree will generate 4 moves ahead of the current board position. This will make the minimax make a decision based on 4 moves ahead. When a move is made, the unnecessary branches will get pruned and another layer is generated making the algorithm always run and see 4 moves ahead.
- An issue this process has is that it is very slow and the algorithm has to loop through all the possible moves of the project. To speed this up, we have made a random sample of the next valid moves and we run the minimax in that small sample. However, the better way is to use the alpha-beta pruning which is a bit complex and is out of the scope of our project.

Changes we made:

- There were many changes that we made to the project to make it more feasible. Firstly we were using a chess library but due to the efficiency problems, we had to code the engine ourselves.
- Secondly, we changed the idea to use a neural network and just implemented the Negamax algorithm.

Project reflection:

Overall, we're pretty happy with what we've achieved. However, our code isn't flawless. We've encountered a few bugs, especially when it comes to generating moves for the pieces. These bugs sometimes cause the program to crash. Despite these issues, we've successfully met the main goal of our project, which was to create a chess-playing bot. Along the way, we've learned a great deal.

We faced several challenges and limitations throughout the project. Firstly, not all team members were experienced with working on complex systems, so the learning curve was steep. Secondly, we had to rewrite our code multiple times due to problems with the software architecture, which made certain computations either impossible or very computationally intensive. Additionally, the language we used, Python, isn't ideal for high-performance applications. Given more time, we would probably reimplement the algorithms and improve the software architecture for better performance.