Checkers Report

File Structure

1. ***Checkers.py*** contains all the functionalities of the checkers game such as move generators, minimax algorithm and evaluation functions. The functions are well documented.
2. ***Assets*** contains the necessary images used in the GUI

How to Run

You must have python 3 installed.
just type in the terminal ***python Checkers.py*** to run the game.

it's cross-platform.

Configuration

1. You can change the size of the checkers instead of the default 8*8 to any even number greater than 3.
2. You can change the mode of the game, it can be either ***Mode.MULTIPLE_PLAYER*** or ***Mode.SINGLE_PLAYER***
3. You can change the algorithm used to play a computer move, it can be either ***Algorithm.MINIMAX*** or ***Algorithm.RANDOM*** (minimax is much harder).

Evaluation Function 1

$$Evaluation\ function\ 1 = (2 * \#maximizer\ kings + maximizer\ men - (2 * opponent\ kings + opponent\ men)) * 5$$

This was the first evaluation function I wrote, and it's very intuitive, you need to maximize the number of your pieces and minimize the number of the opponent pieces, giving the kings more weight

Evaluation function 2

I implemented the evaluation function suggested in this paper.

It depends on several parameters.

1. Number of men
2. Number of kings
3. Number of pieces in the middle box (the middle 2 rows and middle 4 columns). This is an advantageous position.
4. Number of pieces in the middle row but not in the middle box

5. Number of vulnerable pieces, that is in an attacked position.
6. Number of protected pieces, that is protected by other pieces or by a wall. I modified the weights suggested by the paper.

Game Internals:

Gameplay: **Satisfied**

- I have developed an interactive GUI based gameplay between a human user and the computer. The Checkerboard consist of squares with different colors and easy to play for human user by clicking on the piece and place it into its highlighted possible square.

- User can select different levels of cleverness of AI by select easy to hard. Easy level can easily be defeated with some move while hard is very hard for human to defeat the AI, most of he time AI won at hard level. Different levels is integrated with different evaluation functions in the minimax algorithm, evaluation function 1 is used for E*asy level* of AI while evaluation function 2 is used for *Hard Level of AI.*

Search algorithm: Satisfied

- Checkers employs an appropriate representation of the game state, encapsulated within the *Checkers* class. This representation includes the current positions of pieces on the board, as well as information about kingship and turn order.

- The successor function is generating all possible moves for the AI player given a particular game state. It iterates over each piece controlled by the AI, identifying legal moves and capturing sequences.

- The Minimax algorithm with alpha-beta pruning is implemented to traverse the game tree and select optimal moves for the AI. At each level of recursion, the algorithm evaluates potential board positions using a heuristic evaluation function and prunes branches that are determined to be suboptimal.

- The evaluation function used in conjunction with the Minimax algorithm provides a heuristic measure of board positions. This evaluation function considers factors such as piece count, king count, and piece positioning to determine the desirability of a given state for the AI player.

Validation of Moves: **Satisfied**

- The AI player is programmed to generate only valid moves according to the rules of checkers. The successor function, responsible for generating AI moves, ensures that each move adheres to the game's rules, including legal piece movements and mandatory captures when available.

- User moves are checked for validity before being executed. When the user selects a piece to move, the program verifies that the chosen move is legal within the current game state. If

the move is invalid (e.g., attempting to move to an occupied square or make a non-diagonal move), the program rejects the move and prompts the user to make a invalid selection.

- If a user attempts to make an invalid move, such as moving a piece in an illegal direction or failing to capture when required, the program rejects the move and provides clear feedback to the user. Like; “Invalid position: You must select one of you highlighted pieces” or “Invalid move: You must select in a highlighted choice”.

- The program handles forced capture scenarios, where the user must capture an opponent's piece if a capturing opportunity is available. If multiple capturing opportunities exist on the same turn, the user is given the choice to select which capture to execute, ensuring compliance with the game's rules and providing strategic decision-making opportunities.

Other Features: **Satisfied**

- The program supports multi-step capturing moves for the user, allowing consecutive captures to be made if additional capture opportunities arise after the initial capture. When the user makes a capture that opens further capture possibilities, the program prompts the user to continue capturing until no further captures are possible, ensuring to the rules of checkers.

- Similarly, the AI player can execute multi-step capturing moves, leveraging its ability to evaluate potential moves and select the sequence that maximizes capture opportunities. The AI evaluates multiple capture paths to identify the most advantageous sequence of moves, providing a challenging opponent for the user.

- The program implements the rule of king conversion when a normal piece reaches the opponent's back row. Upon reaching the baseline, a normal piece is automatically promoted to a king, including the ability to move and capture diagonally backward.

- The program incorporates a help facility to assist users in identifying available moves based on the current game state. This feature highlights squares containing movable pieces or potential move destinations, providing highlighted square cues to assist users in making informed decisions.

Display-Specific Features: **Satisfied**

- The program effectively displays the checkers board on the screen, providing a visually appealing and intuitive interface for players to interact with. The board consists of alternating red and dark squares, mimicking a traditional checkers board layout. Each square is appropriately sized and positioned within the window, ensuring clarity and ease of use during gameplay.

- Following each completed move, the interface promptly updates to reflect the current game state, including the positions of all pieces on the board. This real-time feedback enhances

the user experience by providing immediate visual confirmation of moves made by both the user and the AI opponent.

- The program incorporates helpful highlight squares to guide users on how to play the game. These instructions are readily accessible within the interface, providing clear and concise guidance on game rules, piece movements, and other essential gameplay mechanics.

- The program offers smooth game interaction mechanisms, by clicking and selection the position functionality, to enable users to intuitively move pieces on the board. Users can click the pieces and select to their desired destination squares, streamlining the gameplay process and enhancing overall user engagement.

- The program features a dedicated display of the game rules, accessible through a button that opens a pop-up window containing comprehensive information about the rules of checkers.

Challenges Encountered

While developing the checkers program, I faced several challenges and were encountered. Here's a detailed analysis of the five most significant ones:

1.Implementing Minimax with Alpha-Beta Pruning:

Implementing the Minimax algorithm with alpha-beta pruning was challenging due to its complexity. Understanding how to represent game states, generate successor moves, and evaluate board positions efficiently required thorough research and experimentation. By breaking down the algorithm into smaller components and testing each part individually, I was able to gradually build a working implementation.

2.Handling Multi-Step Capturing Moves:

Implementing multi-step capturing moves for both the user and AI was challenging. Ensuring that the program correctly identified all possible capturing sequences and allowed the user to choose between multiple options required careful consideration of game logic. Debugging was essential to identify and resolve issues with move validation and execution.

3.GUI Design and Interaction:

Designing a user-friendly GUI and implementing smooth interaction between the user and the game posed a significant challenge. Balancing aesthetics with functionality and ensuring that the interface properly updated after each move required careful attention to detail. Refactoring code and conducting user testing helped improve the overall usability of the interface.

4.Handling Edge Cases and Invalid Moves:

Handling edge cases and invalid moves, such as when a player attempts an illegal move or the game reaches a draw state, required thorough error checking and validation. Implementing robust error handling mechanisms and providing informative error messages helped improve the overall user experience and prevent unexpected behavior.

5.Designing an Efficient Successor Function:

Generating all valid moves for the AI within a reasonable time frame can be computationally expensive, especially for complex game states. The successor function was optimized by considering only diagonal moves within jumping distance for regular pieces and all eight directions for kings. Additionally, capturing moves were prioritized to reduce unnecessary explorations.