Artificial Intelligence: Chess Project

1st Justin Stickel (6718704)
Computing and Business Co-Op
Kitchener, Canada
js19ge@brocku.ca

2nd Haaris Yahya (7054984)
Computer Science
St. Catharines, Canada
hy20ao@brocku.ca

***Abstract*—This document serves to provide insight into the associated Java program, thus allowing for a deeper understanding of its components. Specifically, this document aims to assist in the understanding of key methods and algorithms used in the corresponding program.**

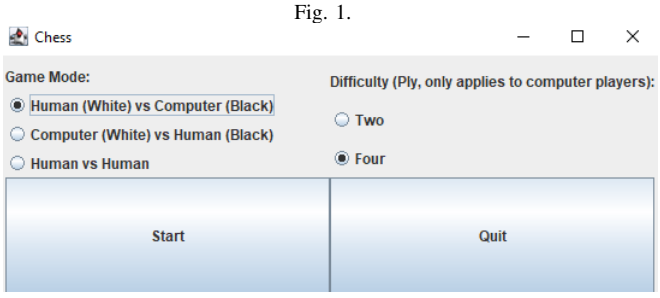
I. Introduction

The associated program was written completely in Java and aims to execute a fully-functional chess program that can be played either as two human players, or against an artificial intelligence. The program aims to implement an AI by utilising alpha-beta pruning which will determine the AI's moves. Since chess is a fairly complex game in that it allows for a multitude of moves from any given position, we have allowed the user to select either two ply or four ply for the AI. Whilst not too deep, these plies will allow for the AI to execute each move within a reasonable time frame. Naturally, selecting a higher ply will yield a more challenging AI as it looks deeper into the benefits and consequences of each move.

A. How to Run the Program

To execute the program, launch the associated program in any java-supporting IDE and run the GUI class. Please ensure that the following libraries are installed or supported by the IDE in use: *javax.swing.∗, java.swing,plaf.BoarderUIResource, java.awt∗, java.awt.event.ActionEvent, java.awt.event.ActionLisstener, java.util.∗, java.util.List*. When the class is run, this will then present the interface shown in **Fig 1**, which will allow for the game-mode and difficulty selection. Whilst the standard chess-setup is the default, if a player so chose to, they could alter the setup manually in the *chess_board* class, which would be supported.

Fig. 1.



II. Code

All code and files are contained within the associated zipped file. Classes were created for each chess piece, as well as each necessary chess and GUI aspect, as we opted to treat every element as an object, since this is a strength and natural attribute of the Java language. Whilst most classes contain minimal abstract information, or simply extend another class, the following methods and classes contain the backbone and core processes of the program.

A. Core Methods/Classes

1) Alpha-Beta Pruning: This is a search algorithm which seeks to improve upon the more simplistic minimax algorithm and locate the optimal node, or in this case move, based on an evaluation score. This algorithm is typically superior to minimax due to it decreasing the total amount of nodes that are present in the search tree. This in essence allows the search to be conducted faster, or conversely allows for a greater depth within the same time span. By including two parameters *Alpha* and *Beta* (hence the name), less desirable nodes within the search tree are eliminated and will not be searched any deeper. In general, the algorithm can be represented through the pseudocode in **Algorithm 1**.

2) Block & chessBoard: Once the algorithm returns the corresponding move (both the selected piece and the destination), the program then moves on to the *Block* and *chessBoard* class. These classes work together to update the board given a move. The *chessBoard* class displays moves that are allowed to the GUI, and additionally, for the Alpha-Beta pruning method, this class also provides the AI with a list of all possible moves for both the player's pieces and the AI's pieces. When a move is confirmed, this class utilises the *Block* class to determine the course of action; depending on if a piece is occupying the destination for the moved piece (in which case the piece is 'killed'), or alternatively if the piece simply moves to an unoccupied space.

3) GUI: Once a move finalized, the board is then updated in this class. Additionally, this class also contains the main method, hence why in **Section 1.A**, the user is instructed to run the program from inside this class. This class also utilises jpgs along with the aforementioned *java.swing* library to create a visual display of the board for the user.

Algorithm 1 Generic Alpha-Beta Pruning Algorithm

```
Set Alpha to -∞
Set Beta to ∞
while currentPly ≤ maxPly do
    if MaximizingPlayer then
maxEvaluation = -∞
        for Each Child of Current Node do
maxEvaluation = minimax(child,depth-1,alpha,beta,max)
Alpha = max(Alpha,maxEvaluation)          ▷ Updates Alpha
            if Beta≤ Alpha then Break
            end if
        end for
    else                                  ▷ Minimizing Player
minEvaluation = ∞
        for Each Child of Current Node do
minEvaluation = minimax(child,depth-1,alpha,beta,min)
Beta = min(Alpha,maxEvaluation)           ▷ Updates Beta
            if Beta≤ Alpha then Break
            end if
        end for
    end if
end while
return chosenMove
```

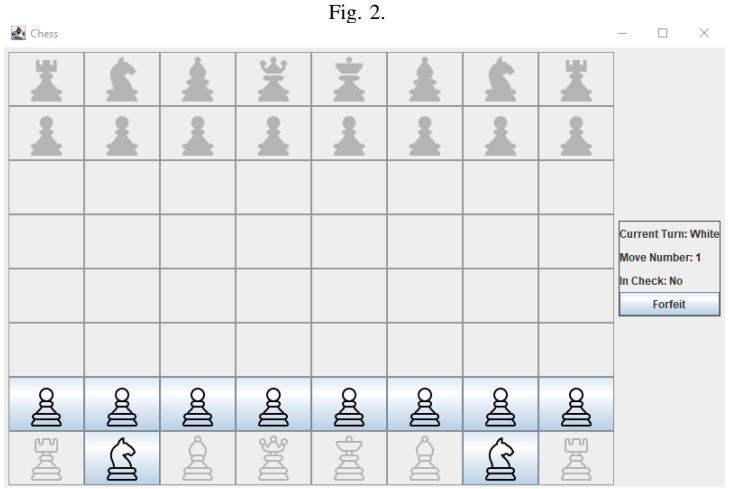
B. Heuristics

Following chess theory, we decided to keep use the standard scoring system for chess. This entails: queens being worth nine; rooks four; bishops/knights both three; and pawns one. Additionally, since the key objective is to attack the enemy king and defend your own king, we made kings worth an arbitrarily large number (two hundred), to ensure that when doing alpha-beta pruning, the AI would never be willing to let their king be taken (and would coincidental always pursue the enemy king). In addition to the basic scoring, another key element of chess theory is having an open position. This means that when your pieces are fully activated, and have a wide dominance over the board, you are typically in a better position than your opponent. To encapsulate this, we also took into account the total number of available moves, and integrated this into our heuristics. This encapsulates other chess rules such as not having your knights be on the 'rim' of the board, and having your bishops be on an open flank. With these heuristics combined, the AI will consider not just being ahead in piece value, but also evaluate the usefulness and openness of each piece. Meaning, in some cases, the AI may choose to give up a pawn for example, in order to obtain a better potion overall.

III. User Interface

Primarily utilising the Java libraries: *java.awt* and *java.swing*; we have chosen to provide a visual representation of the chess game to the user as shown in **Fig 1**. Once a user has chosen a game mode and difficulty, the interface changes to display the board along with the current turn and number of moves (shown in **Fig 2**). Note that in the representation of the board, it also highlights the piece that are able to move, and once clicked, it then highlights the places that piece can move to. This was done with the intention of preventing 'illegal moves', and to make the design as simple for the user as possible, even if they have never played chess before. Other functionality of the GUI (graphical user interface) includes option to forfeit (concede), as well as to return to the main menu where the game mode and difficulty can be re-selected. Once the game concludes wither by victory, loss, or stalemate, the outcome is displayed and the user is given the option to close the program, or begin a new game.

Fig. 2.



IV. Discussion

Our chess program encapsulates all core aspects of chess-theory and allows for full functionality through the GUI. Whilst our chess AI does not have knowledge of pre-established openings or gambits, it still performs admirably against novice chess players. Although alpha-beta pruning allows for a more optimal search tree to be created (compared to algorithms like minimax), due to the exponential growth in the size of the tree as ply is increased, the most the program can handle in a reasonable time-frame is a ply of four. Due to this ply/depth being relatively low, the AI does struggle against gambits that do not have immediate consequences or apparent benefits, such as the *Halloween Gambit*, which performs moves that do not become apparently strong until five moves later. Naturally, the ply could be increased further if a more optimized search algorithm than alpha-beta pruning were used, or if certain openings or board states were pre-programmed into it, which would allow it to skip searching al-together under the right circumstances. Additionally, faster and better hardware would naturally speed up the current search algorithm, which could potentially allow for a ply of six, if time constraints are not of importance. A possible extension onto this program would be to incorporate machine learning by having the AI play against itself and saving/learning from said games.

V. Comparability

In reference to more renowned Chess-AIs, such as Komodo or Stockfish, our AI pales in comparison in its lack of chess-knowledge. Stockfish for example is constantly undergoing minor tweaks with certain board states and counters being analyzed pre-programmed by chess masters. This makes the AI more of a database than an AI, as it is more often than not outputting the predetermined moves rather than calculated responses. It is also important to note that, whilst these 'lines' have been studied to a great extent, even these moves have not been proven to be the most optimal. Theoretically, given an infinite amount of time to analyze every move, and an unlimited amount of storage, our program ,when set to a ridiculously high ply, would find the undisputed optimum move; however as this is implausible, out program is unsurprisingly no match against any modern chess AIs.

References

[1] Beatrice Ombuki-Berman, In-class material (Artificial Intelligence.ppt), 2021