Deliverable 03: Adversarial Search

AI - 4007: Applied Artificial Intelligence

Submitted by: Sara Qayyum | 20I-0556

Submitted on: April 01, 2023

Student Signatures: Sara

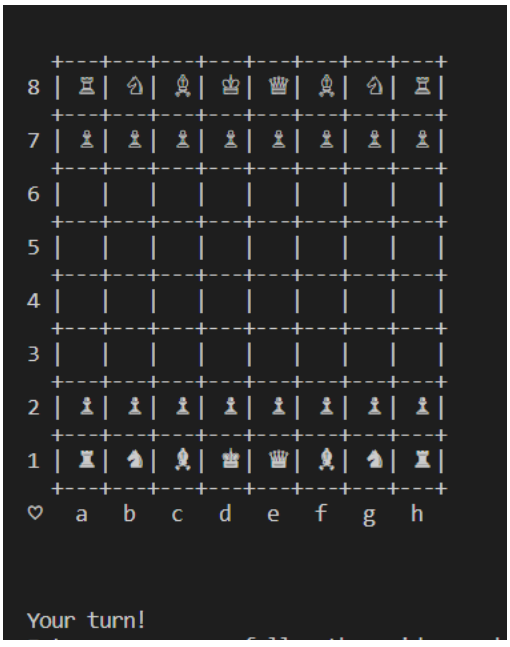
Introduction

For the purpose of this deliverable, we were tasked with visualizing a console-based chess game that implemented Mix-Max Algorithm complimented with Alpha Beta Pruning. To elaborate on the theoretical portion, the chess game followed a user vs computer program - wherein, the computer implemented the aforementioned algorithms to serve as the maximizing player. The standard Min-Max approach was used to gauge the best game state for future moves - pruning to a depth of four. An evaluative function introduced used applicable chess weights - for the purpose of my program, a standard set of pieces against their weights were utilized as follows:

```
# Setting up visual representations for the chess board and their
corresponding weights
_white_representations = ["♖", "♘", "♗", "♔", "♕", "♙"]
_black_representations = ["♖", "♘", "♗", "♔", "♕", "♙"]
_associated_weights = [1, 3, 3, 5, 9, 100]
```

To annotate on the weights, Bishop and Knight were assigned a board weight of 3, Rook as 5, Queen as 9, King with 100, and the Pawn was assigned a board weight of 1. The evaluative function will be delved in later on within the course of this deliverable.

Implementation



To annotate my approach theoretically for the purpose of this deliverable, the game's solution space was dictated by a while loop with arguments 'game_not_over' - upon stalemate and checkmate (functions elaborated later), this while loop would allow the game to end in a fair manner. A user-prompted input allowed the dictation of moves on the board, wherein, the response was dissected as string literals of 4 - (column where piece exists)(row where piece exists)(column where piece will be moved)(row where piece will be moved), which was dissected to ensure the

proper representation and existence of said-piece as well as legality of move.

This was a challenging feat - a separate class of ChessPieces was defined wherein, all possible pieces were represented and their possible legal moves - from with 500+ lines of code, I will elaborate on the Pawn's legalities.

```
class Pawn:
    representation = _whitePieces[5]

    def getLegalMoves(self, chess_board, original_row, original_column, human_turn):
        legal_moves = []

        if human_turn:
            if original_row <= 6:
                if chess_board[original_row + 1][original_column] == " ":
                    legal_moves.append([original_row + 1, original_column])

            # Implementing En Passante

            if original_row <= 5:
                if original_row == 1 and chess_board[original_row + 1][original_column] == " " and \
                        chess_board[original_row + 2][original_column] == " ":
                    legal_moves.append([original_row + 2, original_column])

            if original_row <= 6 and original_column >= 1:
                if chess_board[original_row + 1][original_column - 1] in _blackPieces and original_column >= 1:
                    legal_moves.append([original_row + 1, original_column - 1])

            if original_row <= 6 and original_column <= 6:
                if chess_board[original_row + 1][original_column + 1] in _blackPieces and original_column <= 6:
                    legal_moves.append([original_row + 1, original_column + 1])

            return legal_moves
        else:
```

```
a7a5


  +---+---+---+---+---+---+---+---+
8 | ♖ | ♘ | ♗ | ♔ | ♕ | ♗ | ♘ | ♖ |
  +---+---+---+---+---+---+---+---+
7 |   | ♙ | ♙ | ♙ | ♙ | ♙ | ♙ | ♙ |
  +---+---+---+---+---+---+---+---+
6 |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
5 | ♟ |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
4 |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
3 |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
2 | ♟ | ♟ | ♟ | ♟ | ♟ | ♟ | ♟ | ♟ |
  +---+---+---+---+---+---+---+---+
1 | ♜ | ♞ | ♝ | ♚ | ♕ | ♝ | ♞ | ♜ |
  +---+---+---+---+---+---+---+---+
♡   a   b   c   d   e   f   g   h
```

As the user's move was predetermined by all white-pieces, the initial bit of code allowed the sifting of moves between the computer and user. As explained and implemented with concepts from the assignment - the pawn piece could only move ahead, one board cell at a time - with the exception of 'En Passant', wherein, at its initial state, the pawn can move ahead two board cells - showcased by the console output at the right. On the other hand, provided that a capture was possible, the pawn moves diagonally to do so.

Therein, after the first user move, the computer begin's to generate a response by implementing adversarial search. Below is my approach for implementing the algorithms:

```
def assigningWeights(self, move, isComp):
    user_score = 0
    computer_score = 0

    # Perform move arbitrarily
    representation = chessBoard[move[0]][move[1]]
    chessBoard[move[0]][move[1]] = self._emptySpace
    temp = chessBoard[move[2]][move[3]]
    chessBoard[move[2]][move[3]] = representation
```

The heuristic evaluative function is deliberated upon first - a good evaluation of the board is its state at that point in time - the state is dictated by the remaining pieces on the board - a fantastic way of deliberating how captures by one side would decrease the performance measure of the opponent while increasing their own by a simplistic subtraction. Each mode was made on the board, later undone, to see its sequential impact on the outcome through a numerical value. However, one must also appreciate how some pieces serve a greater purpose than others, hence their importance was measured by their earlier mentioned weights that were added into the function to bring forth realism. Here's a snippet of weights being utilized.

```
    for row in range(self._chessRows):
        for col in range(self._chessColumns):
            piece = chessBoard[row][col]
            if piece == " ":
                continue
            if piece == blackPawn:
                computer_score += 1
            elif piece == blackRook:
                computer_score += 5
            elif piece == blackQueen:
```

Here I showcase the undo move and how the evaluative function provides negative values for poorer scores and positive values for strategic captures provided whose turn it is.

```
    # Un-do move
    chessBoard[move[0]][move[1]] = representation
    chessBoard[move[2]][move[3]] = temp

    if not isComp:
        return user_score - computer_score
    else:
        return computer_score - user_score
```

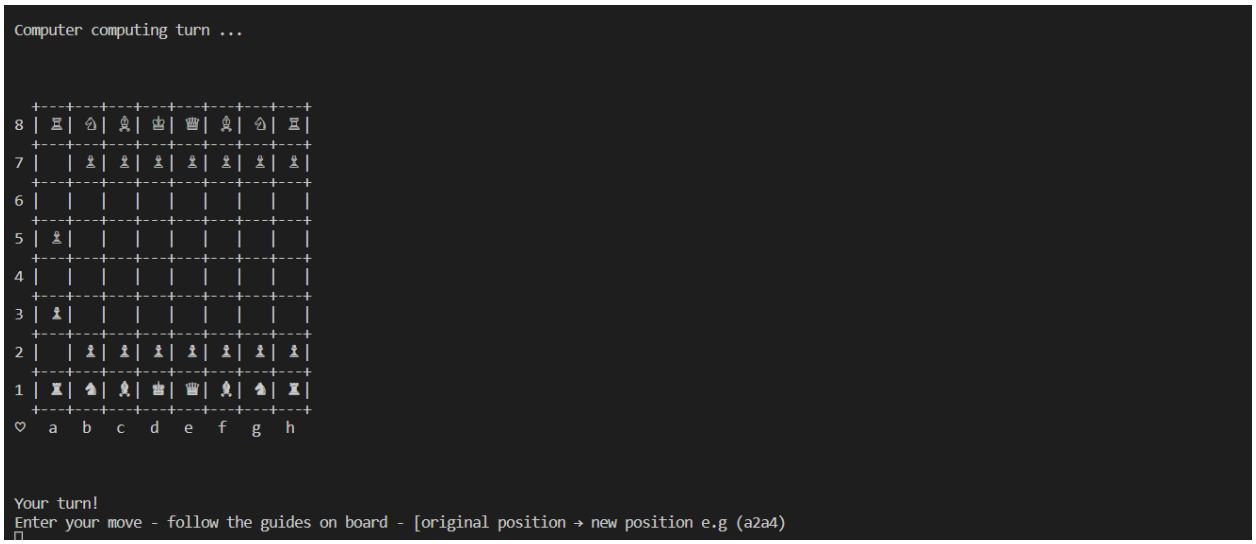
Perhaps the most crucial part of the deliverable is show-cased below. This function takes in arguments like depth that annotates whether the latest depth has been reached to stop traversing recursively, as well as, alpha - which annotates currently held best score for the maximizing player (the computer), beta - which annotates currently held best score for the minimizing player (the user), alongside some arbitrary boolean flags to keep track of player moves and states. Provided that the maximum depth is reached, the best solution is inherently returned to the caller. Once the computer calls the function at a pruning depth of 4, to foresee 4 future playable moves by the user, it assigns scores per moves generated - if the score supersedes the previously maximized score, this score is chosen and the move is saved and categorized as the 'best_move' - in parallel, it makes a recursive call to the user function of the min max algorithm to populate the beta values so that it intuitively plays against the best version or the best outputs the user could potentially generate. Pruning is held by maintaining alpha and beta values that prunes all those values that inherently perform worse than the alpha and betas assigned for the recursive iterations.

```
# Min-max Algorithm - Pruning through Alpha Beta Pruning

def minimax(self, current_depth, alpha, beta, is_computer_player, best_move):
    if current_depth == 0 or not self.game_not_over:
        return best_move

    if is_computer_player:
        max_eval = float('-inf')
        best_move = None

        for move in self.get_all_moves(True):
            score = self.assigningWeights(move, True)

            if score > max_eval:
                max_eval = score
                best_move = move

            alpha = max(alpha, score)
            if beta <= alpha:
                break

        return self.minimax(current_depth - 1, alpha, beta, True, best_move)
```

This snippet intuitively returns the best move that is utilized by the caller to display the move onto the chessboard as the return is in the form of (row where piece exists)(column where piece exists)(row where piece will be moved)(column where piece will be moved), as shown below.

```
        else:
            print("Computer computing turn ...")
            alpha = float('-inf')
            beta = float('inf')

            best_move = self.minimax(pruning_depth, alpha, beta, True, 0)
            orig_row, orig_col, new_r, new_c = best_move

            representation = chessBoard[orig_row][orig_col]
            chessBoard[orig_row][orig_col] = self._emptySpace
            chessBoard[new_r][new_c] = representation

            if can_shift:
                if self.checkmate(True):
                    print("Checkmate - You lose!")
                    self.game_not_over = False
                else:
                    your_turn = True
```

One such result is shown below - the computer moves a pawn piece from a2 to a3.

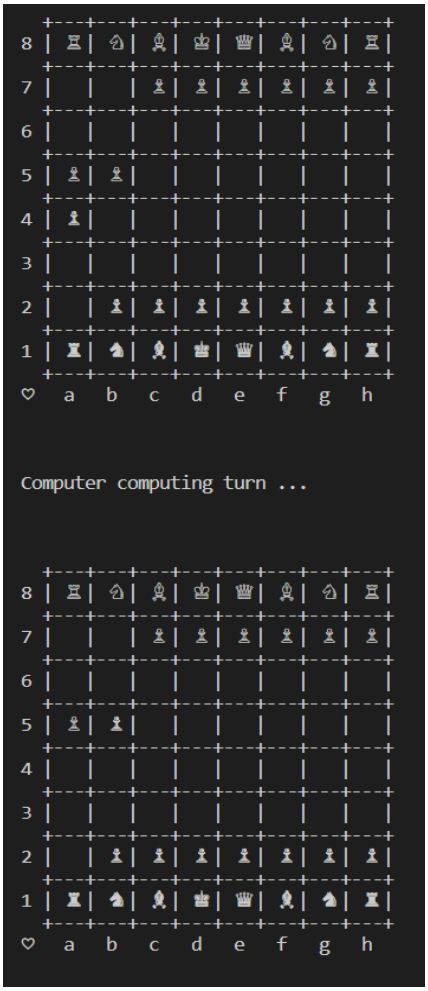


This also affirms how the chessboard is displayed after each iteration of the game. Moving onto the checkmate function - a standard function snippet as shown below - the basic of which annotate all the ranges of the king piece per player - it checks the existence of opponent pieces in its surrounding and then checks their legal moves - if the legal moves fall onto the king - a check is declared - the game does not conclude till the checkmate wherein, the King's legal moves are checked - if unescapable, then the game is drawn to its conclusion and the winner is declared.

```python
def checkmate(self, isComp):
    king_checked = True
    if isComp:
        for row in range(self._chessRows):
            for col in range(self._chessColumns):
                piece = chessBoard[row][col]
                if piece != "♔":
                    continue

                king_checked = False
                legal_moves = []

                for dx, dy in [(0, 1), (0, -1), (1, 0), (-1, 0), (1, 1), (1, -1), (-1, 1), (-1, -1)]:
                    new_row = row + dx
                    new_col = col + dy

                    if new_row < 1 or new_row > 6 or new_col < 1 or new_col > 6:
                        continue

                    piece = chessBoard[new_row][new_col]

                    if piece in ["♜", "♞", "♝", "♚", "♛", "♟"]:
                        if piece == blackRook:
                            moves = Rook().getLegalMoves(chessBoard, new_row, new_col, True)
```

The stalemate on the other hand is a draw per say, where no further legal moves can be annotated - this is drawn out by a simplistic function wherein, all legal moves of remaining players are checked - if none, the game is drawn to a timely conclusion.

Below are some implemented dynamics of an on-going game:



On the right showcases a capture by the computer upon the move b7b5 by the user. The black pawn intuitively captures it as its legal moves inculcate diagonal captures. Below showcases how legal moves are enforced regardless of player type - if the chosen move does not fall under standard move, a warning pops up and the turn does not change - till the player follows through with a legal move, the turn will not change as by the 'can_switch' flag in the solution code.

An entire game can be viewed here