ARTIFICIAL INTELLIGENCE
ASSIGNMENT – 2
ADVERSIAL SEARCH IMPLEMENTATION FOR CHESS PLAYING

APPROACH:

The complexity of the 'chess' game required to take a step-by-step approach while development. The various factors were analyzed and considered during the development are explained in the order they were encountered:

a) Representation of states:

The chess 8x8 board is represented as a **one-dimensional array using square-centric model** where details of each square in the chess board in encoded in one index of the array. The other options of piece centric and 2D representation of the board.

The piece-centric approach would require more calculation while determining valid moves, mobility score during earlier phase of the game and would become easier as the game progresses and pieces are captured.

b) Rules of game:

The rules of valid moves for each piece in terms of direction and distance were formulated and then tested for each type of piece. The implementation required verifying that the pieces stayed inside the board after every possible action/move applied.

The board was initialized with the initial state of standard chess game.

c) Primitive UI implementation to visualize the progress:

The board was first displayed using Python terminal in a 8x8 square using beginning alphabets of each piece type to visualize the performance of the project.

d) Getting user input:

The game for **WHITE is played by the agent** and the game for the BLACK is designed to be keyed-in by the user. Several notations for the chess move exists like standard algebraic notation, figurine algebraic notation, long algebraic notation, etc... The user is required to input the move using long algebraic notation, this notation removes all ambiguities and is idle for programming and computing.

LONG ALGEBRAIC NOTATION:

[P]$f_r f_f$[-|x]$t_r t_f$

where,

- P — Piece type (k – king, q- queen, r – rook, n – knight, b-bishop), P is marked in square brackets to mark it as optional because for moves of pawn we don't mention the piece type as 'p'
- f_r — Rank of from location
- f_f — File of from location
- t_r — Rank of to location
- t_f — File of to location.

- Hyphen between from and to location refers move

x Alphabet 'x' between from and to location refers capture where the piece from 'from' location moved to 'to' location and captured the piece which was occupying it previously.

Rank – refers to the x co-ordinate of a chess board which is represented using alphabets from 'a' through to 'h', while 'a' refers to the left most column and 'h' refers to the righmost column in the chess board where WHITE occupies the bottom and BLACK occupies the top.

File – refers to the y co-ordinate of a chess board which is represented using numbers from 1 to 8, while 1 refers to the bottom row and 8 refers to the top row.

e) Terminal state check:

Process to identify the terminal state is done in two steps:
1) identify if 'king' of the 'player to move' is under checking
2) check if there are no moves for the 'player to move' which makes his king's position not attacked by any of the opponents piece. This is possible only by enumerating all of the 'player to move' moves and for each of those moves trying out the opponent moves, if no such move exists then the program exists and terminates.

'KING UNDER CHECK' and 'CHECKMATE' are printed in the terminal when the situation arises.

f) Utility functions:

Each state of the chess board is analyzed using material score and mobility score.
1) material score: material score identifies number of pieces of each type for 'player to move' and the opponent and calculates a linear score using the weightages assigned for each piece. The weightages for each piece is configurable.

Initial weightage:

- king – 200
- queen – 9
- rook – 5
- bishop – 3
- knight – 3
- pawn – 1

2) mobility score: mobility score depicts the number of valid moves for each player at a particular state when compared to that of the opponent. The moves which attack opponents pieces are gives score based on the opponent pieces which are captured.

Scoring for mobility is done as given below:

- each valid move – 1
- king capture – 9
- queen capture – 5
- rook capture – 4
- bishop capture – 3
- knight capture – 3
- pawn capture – 2

g) Improving UI:

The UI was then uplifted from the primitive version using termcolor package available for Python which allows color display. The pieces are displayed by printing the unicode characters available for the chess pieces i.e. \u2654 to \u265F.

ARCHITECTURE:

Programming Language Used: Python

Algorithm:

Alpha-beta pruning algorithm is used for implementing the adversarial search implementation of the chess game.

The skeleton of the adversarial search algorithm with alpha-beta pruning parameters and the game class was used from http://aima.cs.berkeley.edu/code.html.

An object for the game class should be created and then instantiated with the initial state of the chess game. The pieces were placed based on the chess game using game.__init__() method and the initial state was the passed to the adversarial search algorithm.

The adversarial search algorithm uses alph-beta pruning which uses the idea that if currently the play is at a state m and if there was a better choice of state n in the parent of m or earlier in the game play then we would not have reached state 'm' and so the tree below state 'm' could be ignored and that would not affect the game decision made at the root using the utility function.

Since the game for chess play cannot be analyzed reaching till the terminal nodes, utility function analyzes the state of a non-terminal node when it reached depth 'd', this 'd' can be configured using the UI input terminal, the initial depth is set as 4. As the depth increases the time taken for the move increases exponentially because the number of nodes in the tree increases depth to the power of the branching factor when complete search is done.

In addition to the alpha-beta pruning done by the algorithm custom pruning techniques was also employed as allowed by the game rules, a flag is provided to ignore the moves which are not capture moves above the depth of 3, this would make the game more aggressive and would allow the program to go to more depth taking comparatively lesser time.

A configurable parameter to give more weightage to capture move was also provided so that the agent would be playing a attacking game than a defensive one.

The game class has __init__(), actions(), result(), utility(), terminal_test() methods.

game.actions() - gets all possible moves for the player from a given state
game.result() - gets the resultant state after taking a particular action from a given state
game.utility() - analyzes the value of a state in terms of material value and mobility value as explained before
game.terminal_test() - checks if a state is terminal test

print_chess_board() function is used to print the chess state internal representation as human readable version for game playing.

EXPERIMENTS AND OBSERVATIONS:

1) Depth of the search:

As the search depth increases the agent would play more intelligent game, it would analyze all the game play utility values of the nodes at depth d and take the one which maximizes/minimizes the utility, so increased depth would mean more informed search rather than to avoid immediate pitfalls.

Increasing depth would also take more time to analyze all the nodes at depth d since number of nodes increases exponentially as the depth increases.

2) Branching factor:

Pruning branches of the tree during the search greatly reduces the search time. Though alpha-beta pruning algorithm provides in build pruning mechanism, we can add additional pruning techniques based on the rules of the game and how would we expect the agent to play e.g. attacking game or defensive game.

The custom pruning technique used in the project is that of to prune non-attacking moves after certain depth (d=3) in the program, the depth was fixed as 3 because at lesser depths there were instances there was no move without capture, at depth 3 there were at least some moves to capture opponent pieces. Pruning allowed to go to deeper search depths which took so long without pruning.

3) Changing weightage values for pieces:

The piece types dictate which piece to capture when options are present, what to lose when there are bad states. When weightage was increased for a piece, the piece was saved when there was a chance than to compromise for capture.

4) Move ordering:

The capture moves were analyzed first before analyzing the non-capture moves because analyzing the better moves would more likely result in pruning the remaining branches while doing alpha-beta pruning.