



National University
of computer and emerging sciences

Chess Game Using Minimax & Alpha-Beta Pruning

CS-461 Artificial Intelligence

BS(CS) – D

Batch 2018

Submitted By:

Hassan Shahzad 18i-0441

Sana Ali Khan 18i-0439

Submitted to:

Sir Saad Salman

Date of Submission:

27-05-21

CONTENTS

Assumptions:3

Project Statement.....3

Chess Implementation.....3

 Classes:3

 Logic:.....3

 Minimax Algorithm.....4

 Alpha-Beta Pruning:.....5

 Piece Details5

Types of Implementations:.....6

 Non-GUI Version:.....6

 GUI Version:.....7

ASSUMPTIONS:

- There will only be one player and one AI.
- AI will always have the black piece and Player will always have the white piece
- Player will always have the first turn.
- Player will always enter the input in the following format (row-column). For example, if you want to move a pawn from location 2a to 3a, you will first input to starting location (i.e., 2a) followed by the destination location (i.e., 3a). *Note: This is only required for Non-GUI based Implementation.*
- White pieces are always on the lower side of the board facing the user.
- Row 1 is the top most (i.e., Index 1a = Black rook)

PROJECT STATEMENT

The goal of this project was to make a User vs AI chess game, such that user makes whichever (legal) move they like and in turn, the AI performs the best move it can find. The game ends when either the user or the AI enters checkmate or stalemate state - or if you quit forcibly :)

The algorithm selected for the AI was min-max with alpha-beta pruning, with customizable depth. Greater the depth, deeper the minimax recursion, which makes the AI more likely to find a better move. In our game, the user can select the depth/difficulty of the game. Game can be played either in the console or in the GUI.

No built-in libraries have been used for either the chess implementation or the minimax algorithm.

CHESS IMPLEMENTATION

Classes:

The project structure is OOP-based. We created classes for representing the pieces, the board, the AI and the graphical user interface.

The Piece class represents a basic piece on the board, which has its name (e.g., rook, king), its team, its position on the board (row and column), its points and the list of moves it can make. Other classes inherit from this one and go on to represent a specific piece i.e., King, Queen etcetera. Further elaboration on the pieces is provided later on in this report.

The ChessBoard class represents our game board. It has a two-dimensional list representing the board, with every cell being either empty – represented by a space character – or having an object of one of the classes representing the pieces. The board is manipulated and moves are made using the methods of this class.

The AI class represents the algorithm playing the game with you. It has its assigned team, and the maximum depth allowed for the minimax tree. The AI simply receives an object of the ChessBoard class, and uses minimax to generate the best move. This move is returned and then used by main/GUI to perform it on the board. The minimax algorithm itself is explained further in this report.

GUI class is used to make a graphical chess game playable with the mouse. User can select and drag a piece to wherever they like, and their move (and the corresponding move of the AI) call on the GUI class to update its board and the interface.

Logic:

The basic working of chess is that the user chooses to move a piece to a new position, the board is updated, the AI selects a move and the board is updated. From a programming perspective, the greatest amount of effort goes towards validating the move chosen by the user and verifying that it is possible, legal and allowed. These same checks are applied when the AI is selecting a move as well. Not that complicated, right? Wrong.

The complexity in programming chess arises due to not just the pieces all having a separate set of moves (unlike say, checkers), but also all the rules that chess has for what constitutes as a legal move. A large portion of our time was just spent in debugging and handling all the edge cases that arose.

When a move is being considered, there are extensive checks that it has to clear before it can be performed on the board:

- The input should be in the correct format (e.g., 7a to 6a)
 - The cell in question should not be empty
 - The piece selected should be of the user’s team (or the AI’s team, if it is the one considering the move)
 - The destination should be on the board
 - There should not be a friendly piece at the destination
 - There should be no friendly pieces on the path to the destination position
 - The move should actually be allowed for that piece (e.g., a rook cannot move diagonally)
- Once these checks are cleared, it is a simple matter to shift the piece to the new position on the board and empty out its previous position.
- Once a valid user move has been performed, it is checked that the AI is not in checkmate or stalemate. Then the AI move is performed. The AI gets a list of all the allowed moves for the current board state, and applies minimax on these moves to select the best one. This move is performed, it is checked that the user is not in checkmate or stalemate and then it is the user’s turn again. The game loop continues until either one of the players wins, or there is a draw.

Minimax Algorithm

Both players are trying to win the game, so they try to make the best possible move at each turn. Searching techniques such as BFS (breadth-first search) are not suitable for this as the branching factor is very high, and searching can take a lot of time. So, we needed a search procedure that would:

- Generate procedure such that only good moves are generated
- Test procedure so that the best move can be explored first

Subsequently, we opted to go with the Minimax Search algorithm, which is a depth-first depth-limited search procedure. In this algorithm, the recursive tree of all possible moves is explored to a given depth, and the position is evaluated at the ending “leaves” of the tree.

After that, we return either the smallest or the largest value of the child to the parent node, depending on whether it’s a white or black to move. (That is, we try to either minimize or maximize the outcome at each level.)

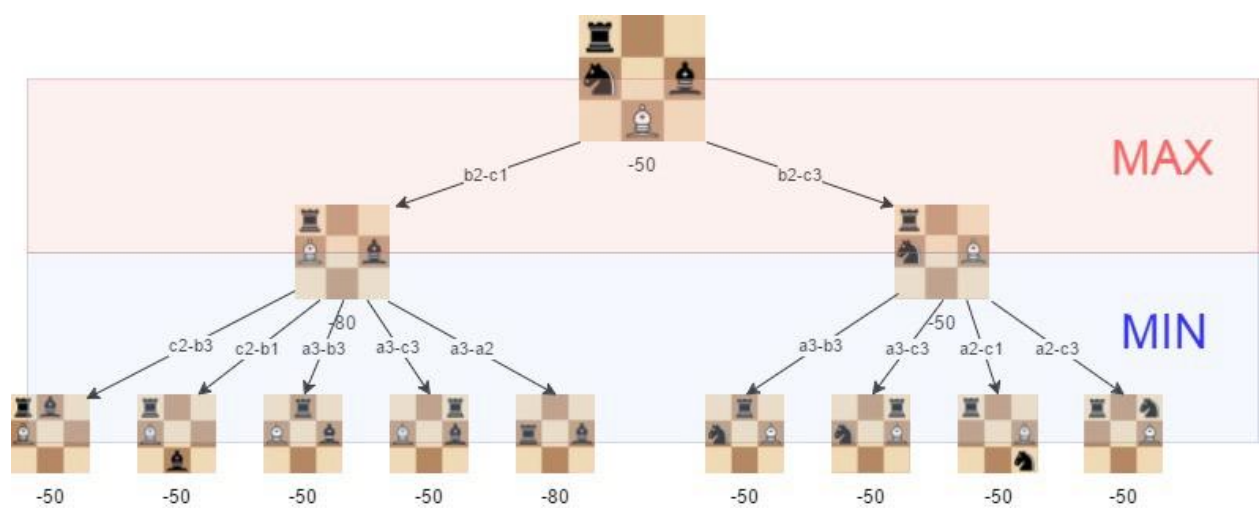


Figure 1: Minimax tree for chess, using example values

It uses a few utility functions:

- **get_all_allowed_moves:** This is a list of all the possible moves generated for the current board state. The function is called upon the ChessBoard object, which in turn calls on all its pieces to return their lists of allowed moves. An allowed move is not the same as a possible move – for example, it is possible for a Queen to move sideways but this move is not allowed if there is a friendly piece in its path. The minimax algorithm requires a list of allowed moves.
- **get_board_evaluation:** It returns a value for the current board state. This value is based upon the values of the pieces on the board, and on the values of their respective positions. The piece values themselves are fixed like so:

- Queen: 9 points
- Rook: 5 points
- Bishop: 3 points
- Knight: 3 points
- Pawn: 1 point
- King: 99999 points (aka infinity)

The evaluation depending on the piece position is done by using a slightly adjusted version of piece-square tables that are originally described in the [chess-programming-wiki](#). These tables are used as is for the team on the lower side of the board, and are reversed and negated for the team on the other side (as they face the other direction.)

Alpha-Beta Pruning:

This is an optimization modification to the minimax algorithm that allows us to disregard some branches in the search tree. This reduces the amount of processing the algorithm has to do, and allows further exploration of the tree.

When evaluating a part of the search tree, pruning is done when a move is found that leads to a worse evaluation/lower value than a previously discovered move. In this case, it makes no sense to explore that move further as it will always cost more than the other move. It is worth noting that alpha-beta pruning does not affect the outcome of minimax, only its speed.

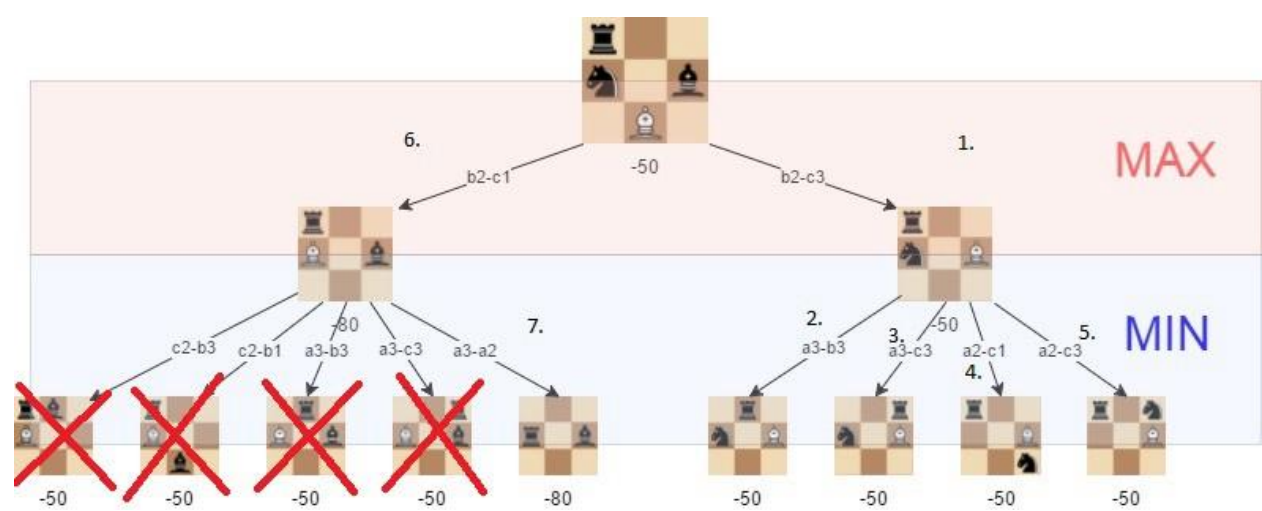


Figure 2: Pruning applied on the minimax tree

Piece Details

The inherited classes for every piece implement their moves. A piece’s moves are represented with a list, with every tuple in that list being two values – how much to move in the x-direction and how much to move in the y-direction. The list includes moves in any direction possible for that move.

As all pieces except pawn can move forwards or backwards on the board, it does not matter what direction we consider their team to be facing. Black could be up on the board or down, it won’t make a difference. The black King could move up or down whether its team was on the upper side of the board or the lower. However, the pawn can only move forwards, so it is important to update it’s moves according to which team it is in and which side of the board that team is on.

We have assumed that black is upper and white is lower. In any case, the default moves for every piece are in the case of them being on the team on the lower side. This is only significant for the pawn piece, which is not allowed to move backwards. So, the values for its moves are negated if it is on the upper side of the board (again, assumed to be black in our game.)

Here is an example of how we represent moves for the King:

```

self.moves = [
    [0, 1], [1, 0], [0, -1], [-1, 0], # right, down, left, up
    [-1, -1], [1, 1], [-1, 1], [-1, -1] # up-left, down-right, up-right, down-left diagonals
]

```

Figure 3: Code representation of the King’s moves

The [0, 1] indicates a move right by adding 0 to the row and 1 to the column. All the moves for all the other pieces are represented similarly.

TYPES OF IMPLEMENTATIONS:

We have implemented 2 different versions of Chess game:

- 1. Non-GUI Version
- 2. GUI Version

Non-GUI Version:

































































In the non-GUI version, we implemented the whole algorithms and here the game is running by taking inputs. The user needs to enter the moves in a specific format and then the piece is moved and the board is re-displayed with the updated pieces. Some features of this version are as follows:

- Only keyboard is used.
- User enters the input in the following format (row-column).
- Each move made by the AI will be displayed as Piece-Move (e.g., Rook 2a).
- List of rules and instructions are by-default displayed in the start of the game.
- Upon checkmate, stalemate etc., appropriate messages are displayed on the terminal and the program is terminated.
- Appropriate checks have been added for validation of inputs and moves made by the user.
- We have used colorama for this version in order to present it in a better way.



Fig 4.1: Rules and Instructions Displayed at the start

Select a difficulty level (1-5): 3

	a	b	c	d	e	f	g	h
1								
2								
3								
4								
5								
6								
7								
8								

Enter the piece to move (e.g. 8e): 7a
Where do you want to move it to: 5a

Moving pawn - 7a to 5a

































































	a	b	c	d	e	f	g	h
1								
2								
3								
4								
5								
6								
7								
8								

Fig 4.2: Chess Board Display (Non-GUI)

GUI Version:

In the GUI version, we are using the library Tkinter. We are totally converting to a GUI platform and using gifs as images. As we were new to Tkinter, hence we took a lot of help from the following link: [Github Link](#) (Note: Only a little help was taken from here and rest was done by ourselves)

User needs to drag and drop the pieces. Some features of this version are as follows:

- You will use mouse only.
- We are using “on click” and “on release” function. User will click on the piece desired to move, drag it to the destination location and simply release the button to place the board there.
- The board keeps refreshing every 100 milliseconds to show smooth movement.
- We have also used menu in this implementation.
- User can view “Rules and Instructions”, “Play Game” or simply “Exit” the game.
- Upon checkmate or stalemate a new popup screen will appear which will show who won and who lost.
- Appropriate checks and moves validations have been implemented here.
- We have used Tkinter for this implementation.

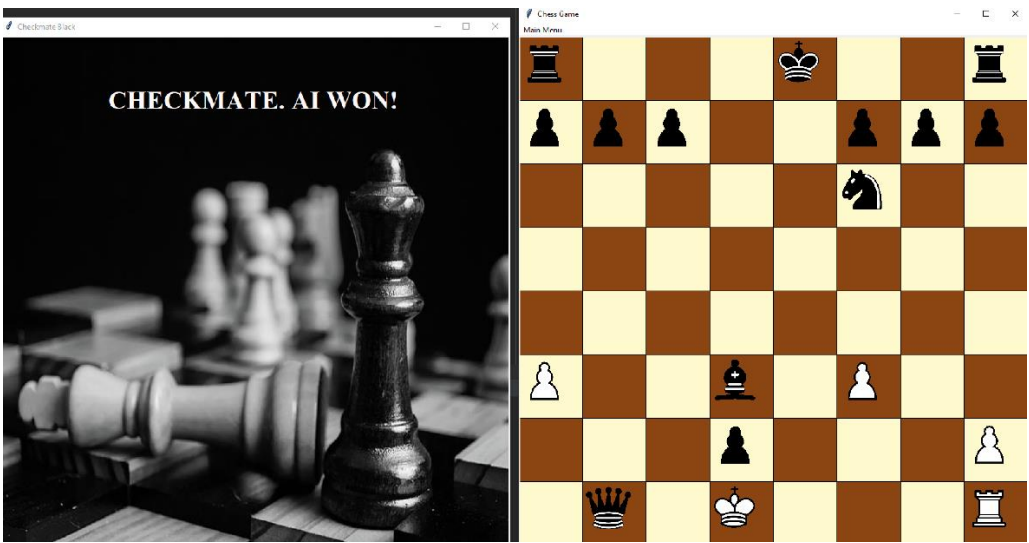


Fig 5.1: Checkmate (AI Wins) Popup Screen



Fig 5.2 Starting Screen of GUI

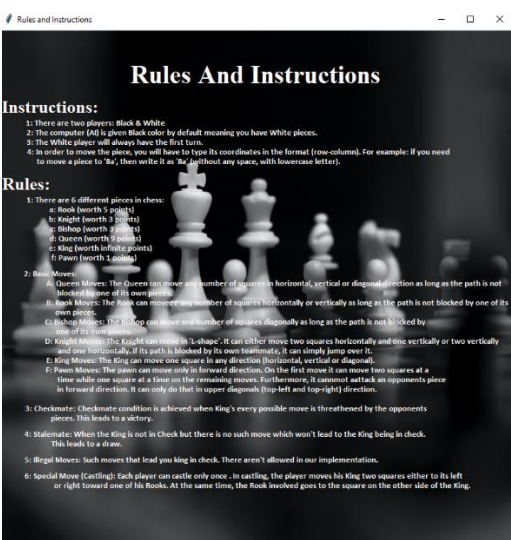


Fig 5.3: Rules and Instructions Screen



Fig 5.4: User Wins Popup Screen