AI ASSIGNMENT
AUTOMATED PLAYER FOR CHECKERS

Astha Tiwari
IIT2018052

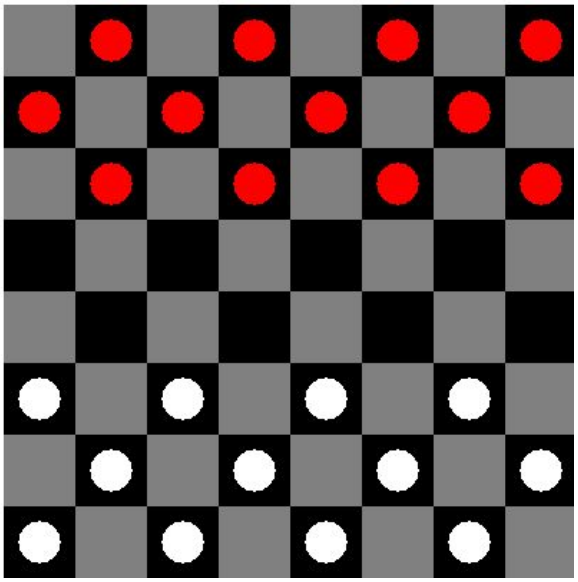
ABSTRACT- Checkers is a very popular game all over the world. In this report we are going to discuss our approach for implementing an AI agent that plays the game.

INTRODUCTION-

Checkers (also known as draught) is considered to be a complicated two player game, English draught has around 10^20 possible states. To design an automated player we are going to make use of AI. Our approach mainly relies on the famous minimax algorithm and some heuristics. Minimax is a decision rule used in decision theory, game theory, statistics and philosophy for minimizing the possible loss while maximizing the potential gain. There are two players involved, MAX and MIN. A game tree is generated, depth-first, starting with the current game position up to the end game position. However generating a whole game tree for a complicated game like checkers may take a lot of time, so we are going to limit the depth of the tree.

INITIAL SETUP-

The checkers board is an 8× 8 grid with a total of 64 squares with alternating colours .one player takes the dark pieces; the other takes the light pieces, each one has 12 pieces initially. Each player places his 12 pieces in the first 12 dark squares of his/her side.



RULES-

We have designed an automated player for the game of checkers with following rules-

1) The player with the black pieces moves first . Then each player takes a single turn. In fact, a player must move in turn. In other words a move cannot be skipped.

2) Ordinary checkers can move one step diagonally forward if no capturing move is possible.
3) **Crowning**: When an ordinary man reaches the, the farthest row forward, it becomes a king.
4) A King can move one step diagonally forward and backwards, if no capturing move is possible.
5) **Capture Move-** If a player's checker is located in the diagonal nearest to the opponent's checker Then the player can jump and capture that checker. To capture a checker the player's checker has to jump over the opponent's checker by moving two diagonal squares in the direction of the opponent's checker .For capturing a checker, the square on the other side of your opponent's checker must be empty.
6) **Mandatory Capture-** If a player has a possible capture move then he/she must make a capture. However if multiple capturing moves are possible then the player has the freedom to choose which capturing move he/she wants to make
7) **Multiple Captures-** If, after capturing an opponent's piece, a piece is in a position to make another capture, it must do so, all as part of the same turn.
8) A player wins the game if all of the opponent's pieces have been captured, or all of his pieces are blocked . We consider the game to be a draw if there have not been any capturing moves in the last 80 moves.

IMPLEMENTATION

GRAPHICAL USER INTERFACE-

Java swing is used in this project for creating a gui. We have used JFrame for drawing the game state and mouse listeners to take the user inputs in case of a human player.

GridBoard Class:

This class extends the JFrame class. This class is responsible for showing the game board state. After every move, changeBoardState and repaint functions are called to update the game state in the GUI. The attribute mat stores the current game state in the form of a 2-D matrix To draw different squares of the board we used drawRect function, and for drawing the pieces we used draw oval function.

Mouse Class:

This class implements the MouseListener class, it is used for taking user inputs, we have used mousePressed and mouseReleased functions. If the user clicks on any square of the board then the variables moveFromXcoordinate, moveFromYcoordinate, moveToXcoordinate, and moveToYcoordinate get updated.

GAME -

State Class:

This class is for storing the game states. It has the attributes board, and player. Board attribute stores the matrix that stores the

positions of different pieces in the board. Player variable is to determine the player that has a chance to move at that game state

JavaApplication1 Class:
This class contains all the game logic.

Function checkCapture:
This function checkCapture determines whether the player has any capturing move is possible or not.

Function isValidMoveForPositivePlayer:
Checks whether the input given by the red player is a valid move or not.

Function isValidMoveForNegativePlayer:
Checks whether the input given by the white player is a valid move or not.

Function getAllOrdinaryMoves:
This function returns all the non capturing possible from a given game state, for pawns it checks for the empty diagonal squares that are one step forward, the pawn is shifted to that empty diagonal squares, if the pawn reaches the crowing row then it becomes a king. For the kings, it checks for the empty diagonals that are one step forward or backward, the king is shifted to that empty diagonal. And then the updated board matrix is added to the possible moves list.

Function getAllCaptures:
This function returns all the capturing possible from a given game state, for pawns it checks for the diagonal squares containing opponents piece, that are one step forward, the square on the other side of the opponent's checker must be empty, the pawn is shifted to that empty diagonal square the opponent piece is removed , if the pawn reaches the crowing row then it becomes a king.it checks for the diagonal squares containing opponents piece, that are one step forward or backwards, the square on the other side of the opponent's checker must be empty, the pawn is shifted to that empty diagonal square the opponent , if multiple captures are possible then it updates the board matrix until no further capturing move is possible, and then that board matrix is added to the list of all possible moves.

Minimax Algorithm-

Function Minimax(board, player)
{ return the action for which the value of mini(result(board,a))is minimum}

Function mini(board, player, depth) {
if(depth>cutoff){
Return heuristic value of board

For all possible action{
mi=minimum(mi,maxi(result(board,a)),otherplayer, depth+1)
}
Return mi}
mi = infinite

Function maxi(board, player, depth) {
if(depth>cutoff){
Return heuristic value of board}

ma= -infinite

```
For all possible action{
    ma=maximum(mi,mini(result(board,a
)), otherplayer, depth+1)
     }
    Return ma}
```

We have set the value of cutoff to be equal to 5.

Heuristic Function-

For this we assign each piece in the board a specific value . Advanced pawns are much closer to becoming Kings, we give them extra value in our evaluation. And we give more value to the kings than pawns.

We split the board into halves.

Red Pawn in the opponent's half of the board value = 7

Red Pawn in the player's half of the board value = 5

Red King's value = 10

White Pawn in the opponent's half of the board value =- 7

White Pawn in the player's half of the board value =- 5

White King's value =- 10

The heuristic value of the game state is equal to product of +1 and sum of value of all the pieces for the red player and product of -1 and sum of value of all the pieces for the white player .

Function play -

This function takes the current state of the game and returns the next state decided by the minimax algorithm. If no further states are possible then it returns null.

Main Function:

In this function, we are initializing the board and the initial game state, and we are asking the user for the mode of the game computer vs computer, human vs computer or computer vs human.

If the player chooses computer vs computer mode, then play function is called after every two second after calling the play function, changeBoardState function is calling to update the state of the game in GUI. When the game states start repeating then a draw is declared.

When the state returned by the play function is null, then the player which had the turn in the previous not null state is declared the winner.

If the player chooses computer vs human mode,

For computers turn the play function is called, which returns the next state decided by minimax algorithm, then for taking the inputs from human player mouse listeners are used, we use isValidMoveForNegative for checking whether the input given by user is a valid move, if the user enters a valid move, then the game state is updated, if a pawn reaches the farthest possible row then it crowned if the move is a capturing move then our program continues taking input from user until no further capturing is possible, for that we are using the function capturePossibleForPiece, and isValidMoveForNegativePlayer, if the user is not able make any valid move in 30 seconds then timeout is declared, computer is declared as winner, if the output from play function is null then human player is declared as winner. If the game states start repeating themself then a draw is declared.

In human vs human mode, we use mouselisteners for taking input, we wait until the player makes a valid move or there is a time out, After the player inputs a valid move , the gamestate get updated, if the move was a capturing if a pawn reaches the farthest possible row then it crowned if the move is a capturing move then our program continues taking input from user until no further capturing is possible, then the inputs for the other player is taken, and so on. This process is repeated until there does not occur any timeout.

CONCLUSION-

Checkers is a complicated game with a large number of legal game states. We used the famous minimax algorithm for implementing an automated player. However due to such a large number of possible states it is not possible to generate the whole game tree, so we used depth limited approach with some good heuristic functions that are giving a satisfactory performance of the agent.