Artificial Intelligence LAB 3 Game Search Problem

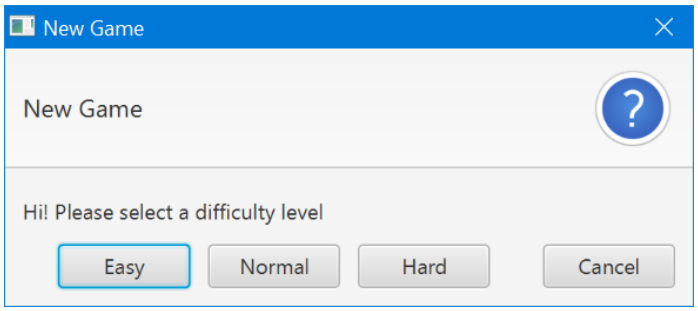
Draughts (International Checkers)

By Shubham Chawla
239571

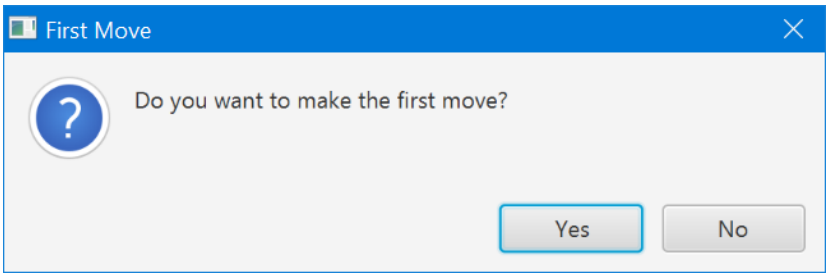
1. Description of the game and its rules.

This project is the implementation of a Checkers game in Java using AI concepts where a human can play against the computer. The game has a traditional board size of 8 x 8 squares .

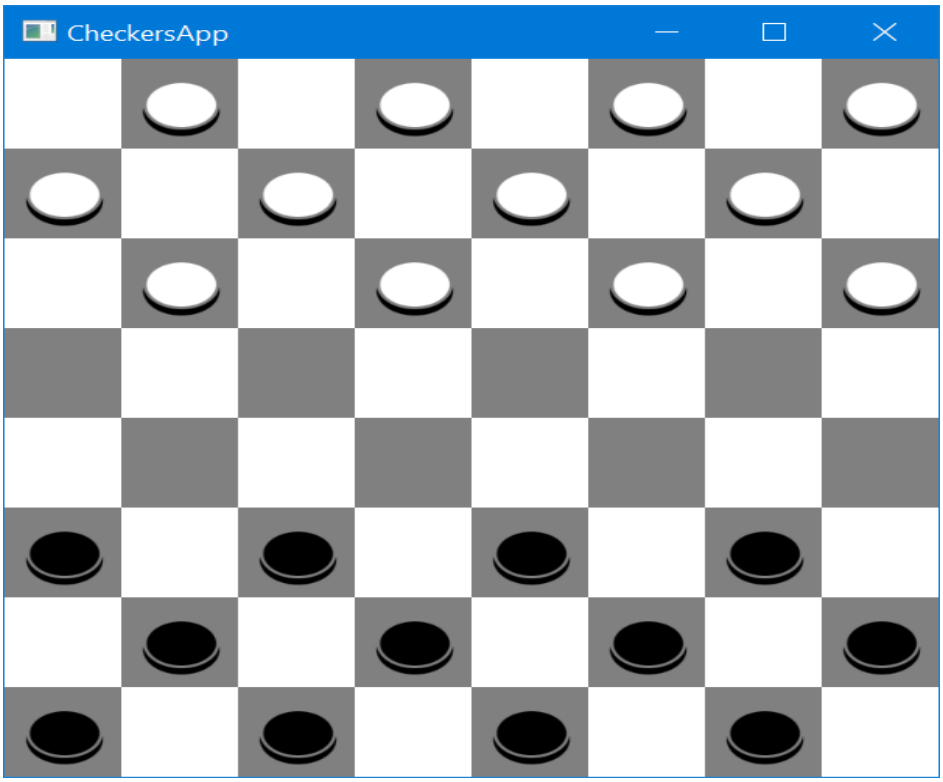
- A graphical user interface (GUI) developed using JavaFX which allows users to move pieces by mouse drag and drop.
- It involves a 8 x 8 board with 12 pieces of white and black each. Black pieces belong to the user and white to the computer. All pieces are placed on dark tiles.
- There are two types of moves possible, A regular move where a piece can move forward diagonally to an adjacent square that is empty. Or a capture move where a piece can jump over and capture opponent's piece in diagonal direction and land on an empty square.
- If there are no legal moves possible for the user, then their turn is skipped.
- The game ends when one player captures all pieces of the other player or if there are no more possible moves left for either of the players. In that case, the player with more number of pieces left wins the game. If both have same number of pieces left, then it will be a draw.
- The game supports three levels of difficulty: Easy, Normal and Hard. The user is prompted to select the difficulty level at the start of the game.



- The user can choose to move first or second at the start of the game.



- Then is shows the board.



2. Min – Max Algorithm Implementation.

This program uses mini-max search algorithm with evaluation function computing values of all valid moves in order to determine the best move among them. The program will search the generated game tree for certain levels and acquire values of current game state, then the program will pick the best values among them and execute the move accordingly. Considering more depth levels I have further used Alpha – Beta pruning .

```
private Move minimaxBegin(Board board, int depth, Side side, boolean maxPlayer)
{
    int alpha = -1000;
    int beta = 1000;
    List<Move> captureMoves = board.getAllValidCaptureMoves(side);
    //System.out.println("\ncapture moves not available?: " + captureMoves.isEmpty());

    List<Move> possibleMovesList;
    if(skip != null) {
        possibleMovesList = board.getValidCaptureMoves(skip.x, skip.y, side);
        skip = null;
    }
    else if(!captureMoves.isEmpty()) {
        possibleMovesList = board.getAllValidCaptureMoves(side);
    }
    else {
        possibleMovesList = board.getAllValidMoves(side);
    }

    List<Integer> heuristics = new ArrayList<>();
    if(possibleMovesList.isEmpty())
        return null;
    Board tempBoard = null;
    for(int i = 0; i < possibleMovesList.size(); i++)
    {
        tempBoard = board.clone();
        tempBoard.makeMove(possibleMovesList.get(i), side);
        heuristics.add(minimax(tempBoard, depth - 1, switchSide(side), !maxPlayer, alpha, beta));
    }

    int maxHeuristics = -1000;
```

3. Alpha-Beta Pruning Algorithm Implementation.

Alpha-Beta Pruning is also designed as part of the algorithm to achieve better search efficiency.

Terminal State	Utility Value
No black piece left (AI wins)	- 1000
No white piece left (User wins)	1000
Same number of white and black pieces left (Tie)	0

To determine the next best move, the Alpha Beta Search algorithm is used.

• Whenever the user's turn is over, the Alpha beta search (ABS) function is called, which based on the current board state considers all possible valid moves and determines the next best move.

• Based on the next best move using the ABS function, the board state changes and the user is allowed to make the next move.

• This cycle repeats till the game ends.

• There are three terminal states with assigned utility values:
Black wins (+1000), White wins (-1000), Tie (0)

• The ABS function uses an evaluation function which uses the difference between number of black or white pieces as a heuristic .

• It also uses the Cutoff function by setting the max depth limit. As soon as the search reaches the max depth limit, it cuts the search and returns the best move.

• The higher the max depth limit is set, the difficulty of the game increases.

• For implementing three different levels of difficulty, different max depth limits are used:
Easy (3), Normal (5), Hard (9).

4. Evaluation Functions used .

Function used by Minmax used to find all the possible moves in order to determine best move amongst them.

```
if(skip != null) {
    possibleMovesList = board.getValidCaptureMoves(skip.x, skip.y, side);
    skip = null;
}
else if(!captureMoves.isEmpty()) {
    possibleMovesList = board.getAllValidCaptureMoves(side);
}
else {
    possibleMovesList = board.getAllValidMoves(side);
}

if(possibleMovesList.isEmpty())
    return null;

Board tempBoard = null;
for(int i = 0; i < possibleMovesList.size(); i++)
{
    tempBoard = board.clone();
    tempBoard.makeMove(possibleMovesList.get(i), side);
    heuristics.add(minimax(tempBoard, depth - 1, switchSide(side), !maxPlayer, alpha, beta));
}

int maxHeuristics = -1000;

Random rand = new Random();         //using random to go through all possible moves
for(int i = heuristics.size() - 1; i >= 0; i--) {
```

The evaluation function compute additions of pieces' values for AI and Human Player separately. Then the entire function will return the value of the summation of AI minus the summation of Human Player. The higher the returned value, the better for the AI and worse for the human player.

```
/ Evaluation Function for using this as a heustric
   private int heuristic(Board b)
   {
       //using the difference between black or white pieces as heuristic
       if(getSide() == Side.BLACK) // for the user
           return b.getNumBlackPieces() - b.getNumWhitePieces();
       else // for AI
           return b.getNumWhitePieces() - b.getNumBlackPieces();
```

5. Move generation function used .

```
public List<Move> getAllValidMoves(Player.Side side) {

    Type normal = side == Player.Side.BLACK ? Type.BLACK : Type.WHITE;

    List<Move> possibleMoves = new ArrayList<>();
    for(int i = 0; i < SIZE; i++)
    {
        for(int j = 0; j < SIZE; j++)
        {
            Type t = getPiece(i, j);
            if(t == normal)
                possibleMoves.addAll(getValidMoves(i, j, side));
        }
    }

    return possibleMoves;
}
```

getAllValidMoves() function is for getting every possible valid move for each piece on board.

```
//gets all valid moves possible
public List<Move> getValidMoves(int row, int col, Player.Side side) {
    Type type = board[row][col];
    Point startPoint = new Point(row, col);
    if (type == Type.EMPTY)
        throw new IllegalArgumentException();

    List<Move> moves = new ArrayList<>();

    //2 possible moves
    if (type == Type.WHITE || type == Type.BLACK) {
        int rowChange = type == Type.WHITE ? 1 : -1;

        int newRow = row + rowChange;
        if (newRow >= 0 && newRow < SIZE) {
            int newCol = col + 1;
            if (newCol < SIZE && getPiece(newRow, newCol) == Type.EMPTY)
                moves.add(new Move(startPoint, new Point(newRow, newCol)));
            newCol = col - 1;
            if (newCol >= 0 && getPiece(newRow, newCol) == Type.EMPTY)
                moves.add(new Move(startPoint, new Point(newRow, newCol)));
        }

    }

    moves.addAll(getValidCaptureMoves(row, col, side));
    return moves;
}
```

This function gets the Valid Moves for each piece on board according to the rules of checkers.

6. Analysis

Max	Min ->	MinMaxAB with Evaluation 1	ABSearch with Evaluation 2
MinMax/alphabeta with Evaluation	Boards Evaluated	159968	30850
	Prunes	27323	5897
	Won	0	Tie
	Game Length	174	27

Game length showing the time calculated as average.
Boards evaluated are the nodes touched in minimax.