Checkers Game - User vs AI

Candidate No. 198732

Introduction

This report outlines how a player vs AI game of checkers was created, using a depth-first search and a minimax algorithm with alpha-beta pruning.

The user interface of the game was created in Java Swing. The player selects the piece that they would like to move and the available moves for that piece are shown in green. To achieve this, every black tile contains a *Piece*. This class stores the objects shown in the UI (the button on each tile), the location of the piece (as shown below), variables used by the UI, the info for this piece, and the list of nodes adjacent to it. The info, contained within another class that is instantiated when the piece is created, stores the variables isPlayer, isActive and isKing. To update the display for each move the UI method UpdateColour is called for any affected pieces. The information stored for that piece is analysed and the colour of the piece updated accordingly. As such, the objects the player manipulates do not move; the colour is changed instead and empty tiles changed to black.

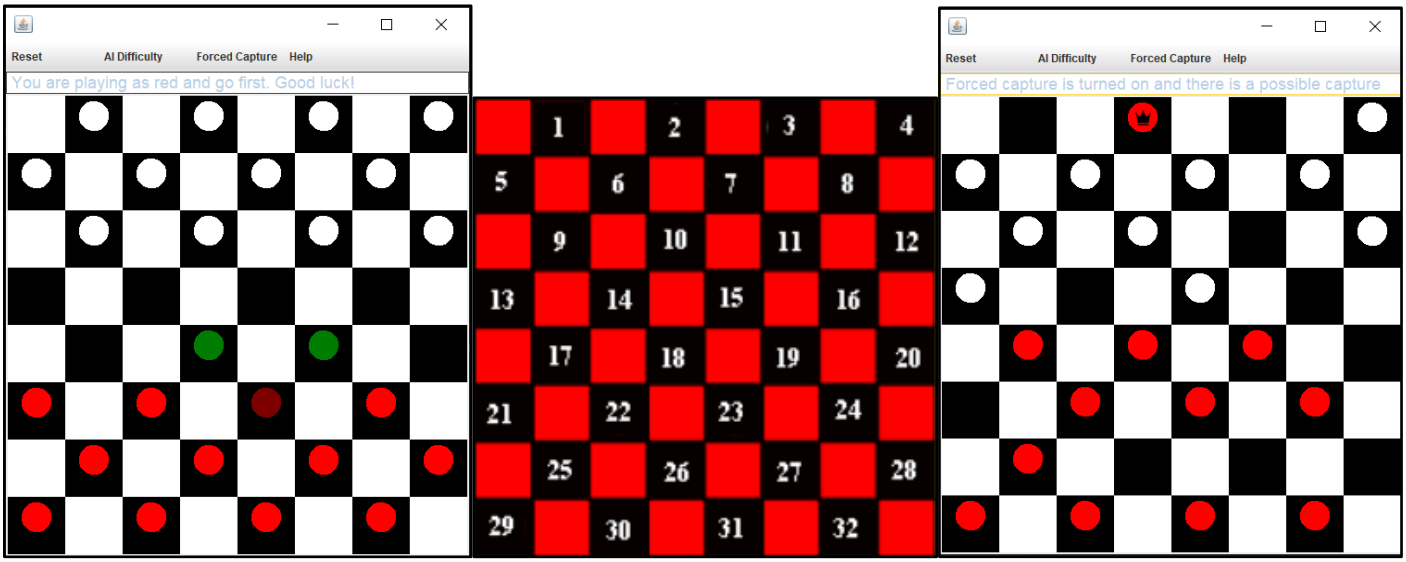


Figure 1: Tile locations. (Checkers Rules and Layout, n.d.)

Any messages for the user are shown in a message box above the board. The colour of the box reflects the urgency of the message: red for errors, orange for warnings, grey for neutral. If the user is attempting an invalid move, then they will be shown a specific warning as to why their selection is not allowed.

The user also has a series of options to choose from:

- Reset - reset the game or reset the game and play as the other colour. The red player starts first so this gives the user an opportunity to play as white and have the AI start.
- AI Difficulty - the user can change how difficult the AI is to play against. The options are Easy, Medium, Hard, Expert, and God Tier. The latter two options also generate a warning that the AI response time will be slower.
- Forced Capture - the player can turn forced capture on or off. If turned on, both players are forced to capture an opponent's piece if that option is available. If the user selects another piece, then a warning is shown to notify them that a capture is available.
- Help - this offers the option to view the rules of checkers. This launches the Checkers Rules and Layout page. (Checkers Rules and Layout, n.d.)

The controller class sets the game up and controls which player's turn it currently is. This is achieved by having one of two classes initialised: AITurn or PlayerTurn. Both of these extend TurnHelpers: a class dedicated to methods that both the AI and Player will need to complete their turn. Regardless of whose turn it is, each class is aiming to pass a Turn class to the CompleteTurn method. Each Turn class contains the piece of origin (which piece is being moved), the final piece (where the piece is moved to), and any pieces captured along the way. Each of these Piece objects will have an updated Info class that can be passed to the UI UpdateColour method to complete the turn.

AI Turn

The AI implements a Minimax algorithm. This works by searching the moves the AI can make, and the moves the player could respond with. The depth of the search (how many turns ahead the AI explores) is determined by the AI difficulty that the player has selected. The options correspond to depths between 2 and 6.

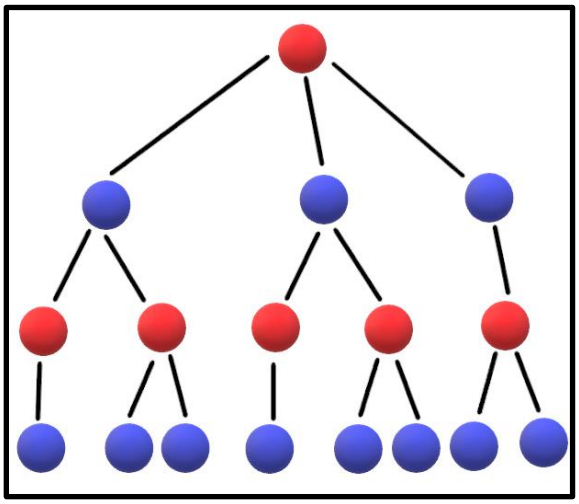
The base minimax algorithm being used is:

```
Minimax (turn, depth, isMaximising)
    IF depth = 0
        RETURN turn.score
        //will return 1 as a default to show that this turn has been explored
    IF isMaximising
        RETURN MAX(turn, depth)
    ELSE
        RETURN MIN(turn, depth)

MAX(turn, depth)
    //determines what the best move for the AI is
    value ← -∞
    DoMove()
    UpdateScore()
    FOR(nextTurn in possibleTurns)
        eval ← turn.score + Minimax(nextTurn, depth - 1, false)
        //switch to a minimising move for the next turn
        value ← max(value, eval)
    UndoMove()
    RETURN value

MIN(turn, depth)
    //determines the best move that the player will respond with
    value ← ∞
    DoMove()
    UpdateScore()
    FOR(nextTurn in possibleTurns)
        eval ← turn.score + Minimax(nextTurn, depth - 1, true)
        //switch to a maximising move for the next turn
        value ← min(value, eval)
    UndoMove()
    RETURN value
```

This essentially creates a tree of possible moves as shown below.



Maximising moves are shown in red whilst blue indicates a minimising iteration. The branch with the greatest overall score is the move that the AI will select. This may result in a piece being captured with the expectation that the player's likely response will make them vulnerable and produce a greater overall score.

DoMove() and UndoMove() are used to update the Info class of any piece being manipulated in a move. The former returns a list of Moves. The Move class stores the piece location and two Info classes for the given piece: before and after. When this list is passed to UndoMove it returns each piece to its 'Before' Info.

The successor function used is an integer score contained in the Turn class. This is updated each recursion of the Minimax function to produce an overall score for that branch. The scores are calculated as follows:

MAX		**MIN**	
Advance move	+5	Captured piece	-10 for each
Captured piece	+10 for each	Captured king	-5 for each
Captured king	+5 for each	Becomes a king	-5
Becomes a king	+5		

For example, with depth = 3, the best move may be advance piece 9 to 14 this turn, anticipate the player's king piece 18 capturing it by moving to 9, then moving piece 6 to 13 to capture their king. The initial move sacrificed a piece to open up a move where the player's king can be captured.
The downside to this scoring system is that these scores are biased, and a better scoring system may produce better results.

The Minimax algorithm can be optimised in two ways:

Proactive Search Validation

Only search moves that are actually possible. Each node has four adjacent nodes that it can explore (2 for edge pieces) but not all of these will be a legal direction for non-king pieces or may be blocked by another piece. Thus, the method GetPriorityPieces returns a list of pieces that show potential at being able to move this turn. This is a simple method to return a list of all nodes that have a neighbouring player piece that can be explored to see if a capture is possible, and all pieces with an empty tile adjacent to it that the piece could advance to.

From here, it is necessary to see which moves each of these pieces is actually capable of making. The Search method implements a Depth First Search to return such a list.

The algorithm for this is as follows:

```
Search(origin, piece, moves, existingTurn)
    //origin is the piece the search was initiated from
    //piece is where the search is currently exploring (e.g. for multi-leg jumps)
    possibleJumpMoves ← list of jump moves from piece
    possibleAdvanceMoves ← list of advance moves from piece
    FOR(nextNode in possibleJumpMoves)
        nextPiece ← getPieceFromNode(nextNode)
        //add this turn to the list of moves being collected
        newTurn ← existingTurn.Clone() or new empty turn
        newTurn.capturedPiece ← getCapturedPiece()
        nextPiece.isKing ← isKingNow(nextPiece) || piece.isKing || capturedPiece.isKing
        //update the isKing value for future iterations of search
        newTurn.origin ← origin
        newTurn.piece ← nextPiece
        moves.add(newTurn)
        //continue search to look for multi-leg jumps
        Search(origin, nextPiece, moves, newTurn)
    FOR(nextNode in possibleAdvanceMoves)
        nextPiece ← getPieceFromNode(nextNode)
        //add this turn to the list of moves being collected
        newTurn ← existingTurn.Clone() or new empty turn
        newTurn.origin ← origin
        newTurn.piece ← nextPiece
        moves.add(newTurn)
        //advance is a single move, no need to search further
    RETURN moves
```

This has now set up a list of all possible moves that this piece can make. So, we have a list of all possible moves for all pieces that can be moved this turn. If forced capture has been enabled by the player then the list of valid moves will cover this. It will use the ForcedCapture method to find the jump moves the AI must take, then run Minimax on those to determine the best / least damaging move to make out of the jumps it is forced to consider. The ForcedCapture method is covered in more detail in the Player Turn section.

With the list of valid moves, Minimax can be executed for each without wasting unnecessary iterations on nodes that will not provide a legal move. As an additional benefit, by limiting the search to the valid moves, there is no need for any validation after a move is scored by Minimax. Minimax will only return scores for valid moves!

Along the way, the search also updates the isKing value of the piece being searched. If a piece becomes a king, then this update allows the next iteration of the search to explore all four directions. A piece becomes a king if it reaches the opposite edge from the side it started on or captures one of the opponent's kings (regicide).

It also updates the active Turn to contain a list of all the captured pieces that jump moves are taking. This eliminates the need to re-iterate over a move after the search has been completed. All the information needed to conduct the move is provided in the list of Turns returned by the search.

Alpha-Beta Pruning

Alpha-Beta pruning can be used to eliminate branches that have already produced a worse result partway through the execution of Minimax. Alpha is the maximum score that a maximising iteration is able to achieve, whilst beta is the minimum score that a minimising iteration will return. Thus, MAX will only alter alpha, and prune any branches where alpha >= beta. MIN will adjust beta, pruning where beta <= alpha.

Adding these two optimisations to Minimax produces the following updated pseudocode:

```
AIMove()
    potentialTurns ← GetPriorityPieces()
    allTurns ← empty list of turns
    FOR(piece in potentialTurns)
        viableTurns ← SEARCH(piece, piece, null, null)
        viableTurns.FOREACH(t -> allTurns.add(t)
    bestTurn ← null
    FOR(turn in allTurns)
        //can now iterate through all possible turns that AI can currently make
        alpha ← -∞
        beta ← ∞
        turn.score ← Minimax(turn, depth, true, alpha, beta)
        //depth will already be established from the player's settings choice
        bestTurn ← bestScore(bestScore, turn)
    DoMove(bestTurn)

Minimax (turn, depth, isMaximising, alpha, beta)
    IF depth = 0
        RETURN turn.score
        //will return 1 as a default to show that this turn has been explored
    IF isMaximising
        RETURN MAX(turn, depth)
    ELSE
        RETURN MIN(turn, depth)

MAX(turn, depth, alpha, beta)
    //determines what the best move for the AI is
    value ← -∞
    DoMove()
    UpdateScore()
    FOR(nextTurn in possibleTurns)
        eval ← turn.score + Minimax(nextTurn, depth - 1, false, alpha, beta)
        //switch to a minimising move for the next turn
        value ← max(value, eval)
        alpha ← max(alpha, value)
        IF(alpha >= beta)
            BREAK
            //prune branch here, no point continuing exploring this turn
    UndoMove()
    RETURN value

MIN(turn, depth, alpha, beta)
    //determines the best move that the player will respond with
    value ← ∞
    DoMove()
    UpdateScore()
    FOR(nextTurn in possibleTurns)
        eval ← turn.score + Minimax(nextTurn, depth - 1, true, alpha, beta)
        //switch to a maximising move for the next turn
        value ← min(value, eval)
        beta ← min(beta, value)
        IF(beta <= alpha)
            BREAK//prune branch here, no point continuing exploring this turn
    UndoMove()
    RETURN value
```

Player Turn

The user is not given the option to attempt moves that are invalid. When they select one of their pieces the Search method generates a list of options for their selected piece. If Forced Capture is enabled, then the selected piece will be compared to a list of pieces that must be moved this turn (to capture the opponent's piece).

This list of pieces is generated by the ForcedCapture method. This searches each of the player's pieces for those that have a capture available. If there are two pieces capable of capturing a single piece, then both of these will be returned. If there are these two, plus another that can capture two of the opponent's pieces then only the latter would be returned.

```
ForcedCapture()
    potentialJumps ← GetPriorityPieces()
    forcePieces ← empty list of Pieces
    score ← 0
    FOR(piece in potentialJumps)
        filteredMoves ← FilterMoves(piece)
        //this is a list of the actual moves available to this piece
        IF(filteredMoves IS NOT EMPTY)
            possibleMoves ← Search(piece, piece, null, null)
            FOR(turn in possibleMoves)
                IF(turn.capturedPieces.size() > score)
                    //this move captures more pieces than those previously found
                    //replace the list
                    forcePieces ← empty list of Pieces
                    forcePieces.add(turn.origin)
                    score ← turn.capturedPieces.size()
                ELSE IF(turn.capturedPieces.size() = score)
                    //this move captures the same number of pieces as those previously found
                    forcePieces.add(turn.origin)
    RETURN forcePieces
```

Game Over Validation

Within TurnHelpers there is a method IsGameWon that is called after each move to determine if the game has been won. There are three situations in which a game might be over at the end of a player's turn: the current player's pieces are all trapped, the opponent's pieces are all trapped, and the opponent's pieces are all captured. If either of these three cases is found to be true, then GameOver is called. This displays a dialog box to notify the user of which player has won (and how) and offer them the opportunity to play again.

References

Checkers Rules and Layout. n.d. A4 Games.
Available at: https://a4games.company/checkers-rules-and-layout/

Code Sources:
(any applicable sections of the code are denoted in a comment)

Board Game Grid:
Cylab. 2007. "*Creating a GUI for a board game".* jvm-gaming.
Available at: https://jvm-gaming.org/t/creating-a-gui-for-a-board-game/30808

Closing a JFrame:
camickr. 2009. "*How to programmatically close a JFrame*". StackOverflow.
Available at: https://stackoverflow.com/questions/1234912/how-to-programmatically-close-a-jframe/1235994

Dialog Boxes:

How to Make Dialogs. n.d. Oracle: Java Documentation.
Available at: https://docs.oracle.com/javase/tutorial/uiswing/components/dialog.html

King Icon:
tkacchuk. n.d. "*Crown king icon on white background vector image*" VectorStock.
Available at: https://www.vectorstock.com/royalty-free-vector/crown-king-icon-on-white-background-vector-4424984

Launch a URL:
Ha Minh, Nam. 2019. "*How to create hyperlink with JLabel in Java Swing*". CodeJava.
Available at: https://www.codejava.net/java-se/swing/how-to-create-hyperlink-with-jlabel-in-java-swing

Resizing an Icon:
tim_yates. 2010. "*resizing a ImageIcon in a JButton*". StackOverflow.
Available at: https://stackoverflow.com/questions/2856480/resizing-a-imageicon-in-a-jbutton

RoundButton Setup:
How to make Round JButtons. n.d. javacodex.
Available at: https://www.javacodex.com/More-Examples/2/14

[All links were last accessed in December 2021]

Appendix

Controller

```
import Classes.*;
import javax.swing.*;
import java.awt.event.WindowEvent;

public class Controller {
    private UI ui;
    private GamePlay game;
    public static void main(String[] args) {
        new Controller();
    }
    private PieceColour defaultPlayerColour;

    public Controller(PieceColour _playerColour, boolean isForcedCapture, Difficulty
aiDifficulty) {
        defaultPlayerColour = _playerColour;
        //create board - setup pieces
        //give game controller to UI (to create bridge to invoke event methods)
        ui = new UI(defaultPlayerColour, this);
        //create tree for game controller to search
        Piece[] allPieces = ui.GetPieces();;
        CreateTree(allPieces);
        //create game
        game = new GamePlay(ui, allPieces, defaultPlayerColour, isForcedCapture,
aiDifficulty);
        ui.InitialiseDifficulty(aiDifficulty);
        ui.InitialiseCapture(isForcedCapture);
    }

    public Controller(){
        defaultPlayerColour = PieceColour.red;
        boolean defaultForcedCapture = true;
        Difficulty defaultDifficulty = Difficulty.Medium;
        //create board - setup pieces
        //give game controller to UI (to create bridge to invoke event methods)
        ui = new UI(defaultPlayerColour, this);
        //create tree for game controller to search
        Piece[] allPieces = ui.GetPieces();;
        CreateTree(allPieces);
        //create game
        game = new GamePlay(ui, allPieces, defaultPlayerColour, defaultForcedCapture,
defaultDifficulty);
        ui.InitialiseDifficulty(defaultDifficulty);
        ui.InitialiseCapture(defaultForcedCapture);
    }

    private void CreateTree(Piece[] pieces) {
        for(int i = 1; i < pieces.length; i++){
            Piece p = pieces[i];
            int loc = p.getLocation();
            int x = p.getCol(), y = p.getRow(), g = p.getGridSize();
            int[] indexes = y % 2 == 0 ? new int[]{-5, -4, 3, 4} : new int[]{-4, -3, 4,
5};
            if(loc + indexes[0] >= 1 && y > 1 && x > 1) p.possibleMoves.add(new Node(loc
+ indexes[0], Direction.UpLeft));
            if(loc + indexes[1] >= 1 && y > 1 && x < g) p.possibleMoves.add(new Node(loc
+ indexes[1], Direction.UpRight));
            if(loc + indexes[2] <= g * g / 2 && y < g && x > 1) p.possibleMoves.add(new
Node(loc + indexes[2], Direction.DownLeft));
            if(loc + indexes[3] <= g * g / 2 && y < g && x < g) p.possibleMoves.add(new
Node(loc + indexes[3], Direction.DownRight));
```

```
        }
    }

    protected void ClickPiece(Piece piece) {
        game.pieceClicked(piece);
    }

    public void UpdateDifficulty(Difficulty diff) {
        game.UpdateDifficulty(diff);
    }

    public void ResetGame(PieceColour playerColour) {
        new Controller(playerColour, game.isForcedCapture, game.aiDifficulty);
        JFrame currentFrame = ui.GetFrame();
        currentFrame.dispatchEvent(new WindowEvent(currentFrame,
WindowEvent.WINDOW_CLOSING));
    }

    public void ToggleForceCapture(boolean isForcedCapture) {
        game.isForcedCapture = isForcedCapture;
    }

    public void CloseGame() {
        /*
        The code for closing a JFrame is from
https://stackoverflow.com/questions/1234912/how-to-programmatically-close-a-
jframe/1235994
        */
        JFrame currentFrame = ui.GetFrame();
        currentFrame.dispatchEvent(new WindowEvent(currentFrame,
WindowEvent.WINDOW_CLOSING));
    }
}
```

UI

```
import Classes.Board;
import Classes.Difficulty;
import Classes.Piece;
import Classes.PieceColour;
import Components.RoundButton;
import javax.imageio.ImageIO;
import javax.swing.*;
import javax.swing.border.LineBorder;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.io.IOException;
import java.net.URI;
import java.net.URISyntaxException;

public class UI implements ActionListener {
    private JFrame frame;
    public JPanel rootPanel;
    private final int gridSize;
    private final Piece[] pieces;
    protected Controller controller;
    protected PieceColour playerColour;
    private JCheckBoxMenuItem[] aiDiffs = new JCheckBoxMenuItem[5];
    private JMenu reset;
    private JMenu help;
    private JCheckBoxMenuItem[] forceCaptures = new JCheckBoxMenuItem[2];
    private JTextField messageBox;
```

```
    public UI(PieceColour _playerColour, Controller _controller) {
        playerColour = _playerColour;
        controller = _controller;
        Board board = new Board();
        gridSize = board.getGridSize();
        pieces = new Piece[(gridSize * gridSize/ 2) + 1];
        CreateBoard();
    }

    private void CreateBoard() {
        //setup main layout
        frame = new JFrame();
        SetupOptions(frame);
        rootPanel = new JPanel();
        rootPanel.setLayout(new BoxLayout(rootPanel, BoxLayout.PAGE_AXIS));
        SetupMessageBox();
        rootPanel.add(messageBox);

        //setup board
        /*
        * The idea for this JPanel grid setup is from https://jvm-gaming.org/t/creating-
a-gui-for-a-board-game/30808/2
        */
        JPanel panel = new JPanel(new GridLayout(gridSize, gridSize));
        boolean colour = true;
        for (int index = 0; index < gridSize * gridSize; index++) {
            //setup panel for grid colour
            JPanel cellPanel = new JPanel();
            if (index % gridSize != 0) colour = !colour;
            if (colour) cellPanel.setBackground(new Color(255, 255, 255));
            else cellPanel.setBackground(new Color(0, 0, 0));

            if(!colour){
                Piece piece = new Piece();
                piece.setLocation((int) Math.ceil(((double)index + 1)/2));

                RoundButton pieceButton = new RoundButton();
                if(index < gridSize * 3)
                {
                    piece.info.isActive = true;
                    piece.info.isPlayer = playerColour == PieceColour.white;
                }
                else if(index + 1 > (gridSize * gridSize) - (3 * gridSize)){
                    piece.info.isActive = true;
                    piece.info.isPlayer = playerColour == PieceColour.red;
                }
                else{
                    piece.info.isActive = false;
                    piece.info.isPlayer = false;
                }
                //setup event handler
                pieceButton.addActionListener(e -> controller.ClickPiece(piece));

                piece.button = pieceButton;
                UpdateColour(piece);
                cellPanel.add(pieceButton);
                cellPanel.setAlignmentY(SwingConstants.CENTER);


                pieces[piece.getLocation()] = piece;

            }
            panel.add(cellPanel);
```

```
        }

        //piece it all together
        panel.setPreferredSize(new Dimension(500, 500));
        GridBagConstraints c = new GridBagConstraints();
        c.anchor = GridBagConstraints.CENTER;

        frame.setContentPane(rootPanel);
        rootPanel.add(panel);

        frame.pack();
        frame.setVisible(true);
    }

    private void SetupMessageBox() {
        messageBox = new JFormattedTextField();
        messageBox.setEnabled(false);
        messageBox.setFont(new Font("Arial", Font.PLAIN, 18));
        messageBox.setHorizontalAlignment(SwingConstants.LEFT);
        String message = playerColour == PieceColour.red ? "You are playing as red and go
first." : "You are playing as white. AI starts.";
        ShowMessage(message + " Good luck!", Color.darkGray);
    }

    //TODO: add rules
    private void SetupOptions(JFrame frame) {
        JMenuBar optionMenu = new JMenuBar();

        //AI difficulty options
        JMenu aiDifficulty = new JMenu("AI Difficulty");
        aiDifficulty.setPreferredSize(new Dimension(100, 30));
        JCheckBoxMenuItem easy = new JCheckBoxMenuItem("Easy");
        easy.addActionListener(this);
        JCheckBoxMenuItem med = new JCheckBoxMenuItem("Medium");
        med.addActionListener(this);
        JCheckBoxMenuItem hard = new JCheckBoxMenuItem("Hard");
        hard.addActionListener(this);
        JCheckBoxMenuItem expert = new JCheckBoxMenuItem("Expert");
        expert.addActionListener(this);
        JCheckBoxMenuItem god = new JCheckBoxMenuItem("God Tier");
        god.addActionListener(this);
        aiDiffs = new JCheckBoxMenuItem[]{easy, med, hard, expert, god};
        aiDifficulty.add(easy);
        aiDifficulty.add(med);
        aiDifficulty.add(hard);
        aiDifficulty.add(expert);
        aiDifficulty.add(god);

        //Reset game option
        reset = new JMenu("Reset");
        reset.setPreferredSize(new Dimension(100, 30));
        JMenuItem resetGame = new JMenuItem("Reset Game");
        resetGame.addActionListener(this);
        reset.add(resetGame);
        JMenuItem resetColour = new JMenuItem("Switch Colour");
        resetColour.addActionListener(this);
        reset.add(resetColour);

        //Forced capture options
        JMenu forcedCapture = new JMenu("Forced Capture");
        forcedCapture.setPreferredSize(new Dimension(100, 30));
        JCheckBoxMenuItem on = new JCheckBoxMenuItem("On");
        on.addActionListener(this);
```

```
        JCheckBoxMenuItem off = new JCheckBoxMenuItem("Off");
        off.addActionListener(this);
        forceCaptures = new JCheckBoxMenuItem[]{on, off};
        forcedCapture.add(on);
        forcedCapture.add(off);

        help = new JMenu("Help");
        help.setPreferredSize(new Dimension(100, 30));
        JMenuItem rules = new JMenuItem("Game Rules");
        rules.addActionListener(this);
        help.add(rules);

        optionMenu.add(reset);
        optionMenu.add(aiDifficulty);
        optionMenu.add(forcedCapture);
        optionMenu.add(help);
        frame.setJMenuBar(optionMenu);
    }

    public void ShowMessage(String message, Color boxColor){
        messageBox.setText(" " + message);
        messageBox.setBorder(new LineBorder(boxColor));
        new java.util.Timer().schedule(
                new java.util.TimerTask() {
                    @Override
                    public void run() {
                        messageBox.setText("");
                        messageBox.setBorder(new LineBorder(Color.darkGray));
                    }
                },
                2000
        );
    }


    public void UpdateColour(Piece piece) {
        DisplayKingIcon(piece);

        if(piece.isOption)
        {
            piece.button.SetColour(new Color(0, 125, 0));
        }
        else if(piece.info.isActive && piece.info.isPlayer && piece.isSelected){
            if(playerColour == PieceColour.red){
                piece.button.SetColour(new Color(125, 0, 0));
            }
            else{
                piece.button.SetColour(new Color(128, 128, 128));
            }
        }
        else if (piece.info.isActive){
            if((piece.info.isPlayer && playerColour == PieceColour.red) ||
(!piece.info.isPlayer && playerColour == PieceColour.white)){
                piece.button.SetColour(new Color(255, 0, 0));
            }
            else{
                piece.button.SetColour(new Color(255, 255, 255));
            }

        }
        else{
            piece.button.SetColour(new Color(0, 0, 0));
        }
```

```
    }

    private void DisplayKingIcon(Piece piece) {
        if(piece.info.isKing && !piece.isOption){
            try{
                /*
                * This icon was copied from https://www.vectorstock.com/royalty-free-
vector/crown-king-icon-on-white-background-vector-4424984
                */
                Image img = ImageIO.read(getClass().getResource("king.png"));
                /*
                * The code to resize an icon was taken from
https://stackoverflow.com/questions/2856480/resizing-a-imageicon-in-a-jbutton
                */
                Image newimg = img.getScaledInstance( piece.button.GetSize() / 2,
piece.button.GetSize() / 2,  java.awt.Image.SCALE_SMOOTH ) ;
                piece.button.setIcon(new ImageIcon(newimg));
            }
            catch(Exception ex){
                ShowMessage("There has been an error in UI/DisplayKingIcon. There was an
issue displaying the king icon in piece " + piece.getLocation(), Color.red);
            }
        }
        else{
            piece.button.setIcon(null);
        }
    }

    public Piece[] GetPieces(){
        return pieces;
    }

    //This can be refactored. V messy.
    @Override
    public void actionPerformed(ActionEvent e) {
        Object source = e.getSource();

        //AI difficulty options
        if(source == aiDiffs[0]){
            //easy AI
            aiDiffs[0].setState(true);
            aiDiffs[1].setState(false);
            aiDiffs[2].setState(false);
            aiDiffs[3].setState(false);
            aiDiffs[4].setState(false);
            controller.UpdateDifficulty(Difficulty.Easy);
        }
        else if(source == aiDiffs[1]){
            //medium AI
            aiDiffs[0].setState(false);
            aiDiffs[1].setState(true);
            aiDiffs[2].setState(false);
            aiDiffs[3].setState(false);
            aiDiffs[4].setState(false);
            controller.UpdateDifficulty(Difficulty.Medium);
        }
        else if(source == aiDiffs[2]){
            //hard AI
            aiDiffs[0].setState(false);
            aiDiffs[1].setState(false);
            aiDiffs[2].setState(true);
            aiDiffs[3].setState(false);
            aiDiffs[4].setState(false);
```

```
            controller.UpdateDifficulty(Difficulty.Hard);
        }
        else if(source == aiDiffs[3]){
            //expert AI
            int n = JOptionPane.showConfirmDialog(frame,
                    "This will make the AI respond much more slowly. Continue?", "Slow
Game Warning",
                    JOptionPane.YES_NO_OPTION);
            if(n == 0){
                aiDiffs[0].setState(false);
                aiDiffs[1].setState(false);
                aiDiffs[2].setState(false);
                aiDiffs[3].setState(true);
                aiDiffs[4].setState(false);
                controller.UpdateDifficulty(Difficulty.Expert);
            }
            else{
                aiDiffs[3].setState(false);
            }
        }
        else if(source == aiDiffs[4]){
            //god AI
            int n = JOptionPane.showConfirmDialog(frame,
                    "This will make the AI respond much more slowly. Continue?", "Slow
Game Warning",
                    JOptionPane.YES_NO_OPTION);
            if(n == 0){
                int m = JOptionPane.showConfirmDialog(frame,
                        "It is futile trying to beat this AI. Try anyway?", "Think you're
up to the challenge?",
                        JOptionPane.YES_NO_OPTION);
                if(m == 0){
                    aiDiffs[0].setState(false);
                    aiDiffs[1].setState(false);
                    aiDiffs[2].setState(false);
                    aiDiffs[3].setState(false);
                    aiDiffs[4].setState(true);
                    controller.UpdateDifficulty(Difficulty.God);
                }
                else{
                    aiDiffs[4].setState(false);
                }
            }
            else{
                aiDiffs[4].setState(false);
            }
        }
        //Reset options
        else if(source == reset.getItem(0)){
            //reset game
            int n = JOptionPane.showConfirmDialog(frame,
                    "You will lose any progress you have made.", "Reset Game?",
                    JOptionPane.YES_NO_OPTION);
            if(n == 0){
                controller.ResetGame(playerColour);
            }
        }
        else if(source == reset.getItem(1)){
            //reset game as other colour
            int n = JOptionPane.showConfirmDialog(frame,
                    "You will lose any progress you have made.", "Switch Colour?",
                    JOptionPane.YES_NO_OPTION);
            if(n == 0){
```

```
                //swap colour
                playerColour = playerColour == PieceColour.red ? PieceColour.white :
PieceColour.red;
                controller.ResetGame(playerColour);
            }
        }
        //Forced capture options
        else if(source == forceCaptures[0]){
            //on
            forceCaptures[0].setState(true);
            forceCaptures[1].setState(false);
            controller.ToggleForceCapture(true);
        }
        else if(source == forceCaptures[1]){
            //off
            forceCaptures[0].setState(false);
            forceCaptures[1].setState(true);
            controller.ToggleForceCapture(false);
        }
        //Help options
        else if(source == help.getItem(0)){
            //show rules
            /*
            * The code to launch a URl is from https://www.codejava.net/java-
se/swing/how-to-create-hyperlink-with-jlabel-in-java-swing
            */
            try {
                Desktop.getDesktop().browse(new URI("https://a4games.company/checkers-
rules-and-layout/"));
            } catch (IOException | URISyntaxException e1) {
                e1.printStackTrace();
            }
        }
    }

    protected JFrame GetFrame() {
        return frame;
    }

    protected void InitialiseDifficulty(Difficulty diff){
        if(diff == Difficulty.Easy){
            aiDiffs[0].doClick();
        }
        else if(diff == Difficulty.Medium){
            aiDiffs[1].doClick();
        }
        else{
            aiDiffs[2].doClick();
        }

    }

    protected void InitialiseCapture(boolean isForcedCapture){
        if(isForcedCapture){
            forceCaptures[0].doClick();
        }
        else{
            forceCaptures[1].doClick();
        }
    }

    protected void GameOverDialog(String message){
        /*
```

```
        * The code for dialog boxes is from
https://docs.oracle.com/javase/tutorial/uiswing/components/dialog.html
        */
        int n = JOptionPane.showConfirmDialog(frame,
                message + " Play again?", "Game Over",
                JOptionPane.YES_NO_OPTION);
        if(n == 0){
            controller.ResetGame(playerColour);
        }
        else{
            controller.CloseGame();
        }

    }
}
```

AITurn

```
import Classes.*;

import java.awt.*;
import java.util.ArrayList;
import java.util.List;
import java.util.stream.Collectors;

public class AITurn extends TurnHelpers{

    protected static int aiDepth;

    public AITurn(UI _ui, Piece[] _allPieces, PieceColour _playerColour, GamePlay _game)
{
        super(_ui, _allPieces, _game, _playerColour);
        isPlayerTurn = false;
    }

    public void MakeMove(){
        //get all possible turns
        List<Turn> allTurns = new ArrayList<>();
        //if forced capture is on then only look at the pieces that have a capture
available
        List<Piece> forcePieces = game.isForcedCapture ? ForcedCapture(isPlayerTurn) :
new ArrayList<>();
        //only look at pieces that have a potential move - less expensive
        List<Piece> potentialTurns = forcePieces.isEmpty() ?
GetPriorityPieces(Priority.Both, isPlayerTurn) : forcePieces;
        for(Piece p : potentialTurns){
            List<Turn> pTurns = new ArrayList<Turn>();
            pTurns = Search(p, p, MoveType.Both, pTurns, null, isPlayerTurn, false);
            pTurns.forEach(t -> allTurns.add(t));
        }

        //for each potential turn, run MINIMAX to explore its score. Run the best scoring
move
        Turn bestTurn = null;
        for(Turn turn : allTurns){
            turn.score = Minimax(turn, aiDepth, true,Double.NEGATIVE_INFINITY,
Double.POSITIVE_INFINITY);
            bestTurn = bestTurn == null || turn.score > bestTurn.score ? turn : bestTurn;
            System.out.println("Piece " + turn.origin.getLocation() + " has score " +
turn.score);
        }
        if(bestTurn == null || bestTurn.score == 0){
            isPlayerTurn = !isPlayerTurn;
            GameOver("All pieces are trapped");
```

```
        }
        else{
            System.out.println("MOVING piece " + bestTurn.origin.getLocation() + " to
piece " + bestTurn.piece.getLocation());
            Turn finalBestTurn = bestTurn;
            new java.util.Timer().schedule(
                    new java.util.TimerTask() {
                        @Override
                        public void run() {
                            CompleteTurn(finalBestTurn);
                            game.isPaused = false;
                        }
                    },
                    1500
            );

        }
    }


    private double Minimax(Turn turn, int depth, boolean isMaximising, double alpha,
double beta) {
        //TERMINAL NODE
        if (depth == 0) {
            //if score has max + min to 0 then return 1 - to show that it has found a
score
            return turn.score == 0 ? 1 : turn.score;
        }
        //MAX
        if (isMaximising) {
            return Max(turn, depth, alpha, beta);
        }
        //MIN
        else if (!isMaximising) {
            return Min(turn, depth, alpha, beta);
        }
        else{
            ui.ShowMessage("There has been an error in AITurn/MiniMax. The code has
failed to meet the conditions for either min or max.", Color.red);
            return 0;
        }
    }

    private double Min(Turn turn, int depth, double alpha, double beta) {
        double value = Double.POSITIVE_INFINITY;
        List<Move> moves = DoMove(turn, false);
        moves.forEach(m -> turn.changes.add(m));

        //update score
        //--10 for each captured piece
        turn.score -= turn.capturedPieces.size() * 10;
        //--5 for each captured king
        turn.score -= turn.capturedPieces.stream().filter(p ->
p.info.isKing).collect(Collectors.toList()).size() * 5;
        //--5 for becoming a king
        turn.score -= turn.origin.info.isKing != turn.piece.info.isKing &&
turn.piece.info.isKing ? 5 : 0;

        List<Turn> nextTurns = new ArrayList<>();
        //if forced capture is on then only look at the pieces that have a capture
available
        List<Piece> forcePieces = game.isForcedCapture ? ForcedCapture(!isPlayerTurn) :
new ArrayList<>();
```

```
        //only look at pieces that have a potential move - less expensive
        List<Piece> potentialTurns = forcePieces.isEmpty() ?
GetPriorityPieces(Priority.Both, !isPlayerTurn) : forcePieces;
        for(Piece p : potentialTurns){
            List<Turn> pTurns = new ArrayList<Turn>();
            pTurns = Search(p, p, MoveType.Both, pTurns, null, !isPlayerTurn, false);
            pTurns.forEach(t -> nextTurns.add(t));
        }
        for(Turn nextTurn : nextTurns){
            //get the score for this branch (depending on depth it may branch in the next
search too - will return best branch)
            double eval = turn.score + Minimax(nextTurn, depth - 1, true, alpha, beta);
            //return the move that the player would make - as it advantages them the most
            value = Math.min(value, eval);
            //alpha-beta pruning
            //if this branch is already known to disadvantage the player then don't
bother looking at the rest of it
            beta = Math.min(beta, value);
            if(beta <= alpha){
                break;
            }
        }

        //undo move otherwise it'd shuffle the board
        UndoMove(turn);
        return value;

    }

    private double Max(Turn turn, int depth, double alpha, double beta) {
        double value = Double.NEGATIVE_INFINITY;
        List<Move> moves = DoMove(turn, false);
        moves.forEach(m -> turn.changes.add(m));

        //update score
        //++5 for moving forward
        turn.score += turn.moveType == MoveType.Advance ? 5 : 0;
        //++10 for each captured piece
        turn.score += turn.capturedPieces.size() * 10;
        //++5 for each captured king
        turn.score += turn.capturedPieces.stream().filter(p ->
p.info.isKing).collect(Collectors.toList()).size() * 5;
        //++5 for becoming a king
        turn.score += turn.origin.info.isKing != turn.piece.info.isKing &&
turn.piece.info.isKing ? 5 : 0;

        List<Turn> nextTurns = new ArrayList<>();
        //if forced capture is on then only look at the pieces that have a capture
available
        List<Piece> forcePieces = game.isForcedCapture ? ForcedCapture(isPlayerTurn) :
new ArrayList<>();
        //only look at pieces that have a potential move - less expensive
        List<Piece> potentialTurns = forcePieces.isEmpty() ?
GetPriorityPieces(Priority.Both, isPlayerTurn) : forcePieces;
        for(Piece p : potentialTurns){
            List<Turn> pTurns = new ArrayList<Turn>();
            pTurns = Search(p, p, MoveType.Both, pTurns, null, isPlayerTurn, false);
            pTurns.forEach(t -> nextTurns.add(t));
        }
        for(Turn nextTurn : nextTurns){
            //get the score for this branch (depending on depth it may branch in the next
search too - will return best branch)
            double eval = turn.score + Minimax(nextTurn, depth - 1, false,alpha, beta);
```

```
            //return the move that the AI would make - as it advantages them the most
            value = Math.max(value, eval);
            //alpha-beta pruning
            //if this branch is already known to disadvantage the AI then don't bother
looking at the rest of it
            alpha = Math.max(alpha, value);
            if(alpha >= beta){
                break;
            }
        }
        //undo move otherwise it'd shuffle the board
        UndoMove(turn);
        return value;
    }


}
```

PlayerTurn

```
import Classes.*;
import java.awt.*;
import java.util.ArrayList;
import java.util.List;
import java.util.stream.Collectors;

public class PlayerTurn extends TurnHelpers {
    protected Turn turn;

    private List<Turn> possibleMoves;

    public PlayerTurn(UI _ui, Piece[] _allPieces, GamePlay _game, PieceColour
_playerColour, Piece _origin) {
        super(_ui, _allPieces, _game, _playerColour);
        turn = new Turn(_origin);
        isPlayerTurn = true;
    }

    protected void ShowOptions(){
        //if forced capture then check for any pieces that need to capture
        if(game.isForcedCapture){
            List<Piece> forcePieces = ForcedCapture(isPlayerTurn);
            if(!forcePieces.isEmpty() && !forcePieces.contains(turn.origin)){
                ui.ShowMessage("Forced capture is turned on and there is a possible
capture", Color.orange);
                game.RestartMove(turn.origin);
                return;
            }
        }

        possibleMoves = new ArrayList<Turn>();
        if(turn.origin.info.isPlayer && turn.origin.info.isActive) {
            turn.origin.isSelected = true;
            ui.UpdateColour(turn.origin);
            possibleMoves = Search(turn.origin, turn.origin, MoveType.Both,
possibleMoves, null, isPlayerTurn, true);
            for(Turn t : possibleMoves){
                t.piece.isOption = true;
            }
            if(possibleMoves.size() == 0){
                ui.ShowMessage("This piece is trapped - no moves possible",
Color.orange);
            }
        }
```

```
    }


    public void RemoveSelection(Piece piece) {
        //clear options
        if(possibleMoves != null){
            for(Turn t : possibleMoves){
                t.piece.isOption = false;
                ui.UpdateColour(t.piece);
            }
        }

        //clear selection
        ClearSelectedPiece(turn);
        ClearOptions(possibleMoves);
    }

    public void ChooseMove(Piece piece) {
        List<Turn> matchingTurns = possibleMoves.stream().filter(t ->
t.piece.getLocation() == piece.getLocation()).collect(Collectors.toList());
        //no matching turns
        if(matchingTurns.size() == 0){
            ui.ShowMessage("This is not a valid move", Color.orange);
            ClearSelectedPiece(turn);
        }
        else {
            turn = matchingTurns.get(0);
            //if there are multiple matching turns then use the most beneficial one for
the player
            for(Turn t : matchingTurns) {
                turn = t.capturedPieces.size() > turn.capturedPieces.size() ? t : turn;
            }
            CompleteTurn(turn);
        }
        ClearOptions(possibleMoves);
    }
}
```

TurnHelpers

```
import Classes.*;

import java.awt.*;
import java.util.ArrayList;
import java.util.List;
import java.util.Optional;
import java.util.stream.Collectors;

public class TurnHelpers {
    protected UI ui;

    protected Piece[] allPieces;

    protected GamePlay game;

    protected boolean isPlayerTurn;

    protected PieceColour playerColour;

    public TurnHelpers(UI _ui, Piece[] _allPieces, GamePlay _game, PieceColour
_playerColour){
        ui = _ui;
        allPieces = _allPieces;
        game = _game;
```

```
        playerColour = _playerColour;
    }

    protected List<Piece> ForcedCapture(boolean isPlayer) {
        List<Piece> possibleJumps = GetPriorityPieces(Priority.High, isPlayer);
        List<Piece> forcePieces = new ArrayList<Piece>();
        int score = 0;
        //go through each potential jump to see if it is a viable move
        for(Piece p : possibleJumps){
            if(!FilterMoves(p, p.possibleMoves, MoveType.Jump).isEmpty()){
                List<Turn> possibleMoves = new ArrayList<Turn>();
                possibleMoves = Search(p, p, MoveType.Jump, possibleMoves, null,
isPlayer, false);
                for(Turn t : possibleMoves){
                    if(t.capturedPieces.size() > score){
                        forcePieces = new ArrayList<Piece>(){};
                        forcePieces.add(t.origin);
                        score = t.capturedPieces.size();
                    }
                    else if(t.capturedPieces.size() == score){
                        forcePieces.add(t.origin);
                    }
                }
            }
        }
        return forcePieces;
    }

    protected List<Piece> GetPriorityPieces(Priority priority, boolean isPlayer) {
        List<Piece> priorityPieces = new ArrayList<>();
        for(Piece p : allPieces){
            if(p == null || p.info.isPlayer != isPlayer || !p.info.isActive){
                continue;
            }
            for(Node n : p.possibleMoves){
                Piece poss = allPieces[n.pieceLocation];
                //pieces with a player piece adjacent
                if(poss.info.isPlayer != isPlayer && poss.info.isActive &&
!priorityPieces.contains(p) && priority != Priority.Low){
                    priorityPieces.add(p);
                }
                //pieces with an empty space adjacent
                if(!poss.info.isActive && !priorityPieces.contains(p) && priority !=
Priority.High){
                    priorityPieces.add(p);
                }
            }
        }
        return priorityPieces;
    }

    protected List<Node> FilterMoves(Piece currentPiece, List<Node> possibleMoves,
MoveType moveType) {
        List<Node> filteredMoves = new ArrayList<Node>();
        for (Node possN : possibleMoves) {
            if (IsValidDirection(possN, currentPiece)) {
                Piece possP = allPieces[possN.pieceLocation];
                if ((moveType == MoveType.Advance || moveType == MoveType.Both) &&
!possP.info.isActive) {
                    filteredMoves.add(possN);
                }
                if ((moveType == MoveType.Jump || moveType == MoveType.Both) &&
possP.info.isPlayer != currentPiece.info.isPlayer && possP.info.isActive) {
```

```
                    Object[] nextNs = possP.possibleMoves.stream().filter(x ->
x.direction == possN.direction).toArray();
                    Node nextN = nextNs.length > 0 ? (Node) nextNs[0] : null;
                    if (nextN != null) {
                        Piece nextP = allPieces[nextN.pieceLocation];
                        if (!nextP.info.isActive) {
                            filteredMoves.add(nextN);
                        }
                    }
                }
            }
        }
        return filteredMoves;
    }

    public List<Turn> Search(Piece origin, Piece piece, MoveType legalMoveType,
List<Turn> moves, Turn existingTurn, boolean isPlayer, boolean updateColour) {
        if(legalMoveType == MoveType.Jump || legalMoveType == MoveType.Both){
            List<Node> jumpMoves = FilterMoves(piece, piece.possibleMoves,
MoveType.Jump);
            for(Node nextNode : jumpMoves){
                Turn newTurn = existingTurn == null ? new Turn(origin) :
existingTurn.Clone();
                Piece nextPiece = allPieces[nextNode.pieceLocation];
                Optional<Node> capturedNode = piece.possibleMoves.stream()
                        .filter(p -> p.direction == nextNode.direction).findFirst();
                Piece capturedPiece = allPieces[capturedNode.get().pieceLocation];
                if(!newTurn.capturedPieces.contains(capturedPiece)){
                    //update nextPiece IF this is not an exploratory exercise
                    if(updateColour){
                        nextPiece.isOption = true;
                        ui.UpdateColour(nextPiece);
                        nextPiece.info.isKing = isKingNow(nextPiece) || piece.info.isKing
|| capturedPiece.info.isKing;
                    }

                    //update list of possible turns
                    newTurn.moveType = MoveType.Jump;
                    moves.add(newTurn);
                    newTurn = moves.get(moves.size() - 1); //get the duplicate
                    newTurn.piece = nextPiece;
                    newTurn.capturedPieces.add(capturedPiece);

                    //continue search
                    //i don't remember why i wrap this in the isPLayer/!isPlayer
                    //but when I remove it something breaks
                    nextPiece.info.isPlayer = isPlayer;
                    Search(origin, nextPiece, MoveType.Jump, moves, newTurn, isPlayer,
updateColour);
                    nextPiece.info.isPlayer = !isPlayer;
                }
            }
        }
        //if we're looking for advance moves and there are no jump moves available -
forced capture
        if((legalMoveType == MoveType.Advance || legalMoveType == MoveType.Both) &&
moves.isEmpty()){
            List<Node> advanceMoves = FilterMoves(piece, piece.possibleMoves,
MoveType.Advance);
            for (Node nextNode : advanceMoves){
                Piece nextPiece = allPieces[nextNode.pieceLocation];
                if(nextPiece != origin){
                    Turn newTurn = new Turn(origin);
```

```
                    newTurn.piece = nextPiece;

                    nextPiece.info.isKing = isKingNow(nextPiece);

                    newTurn.moveType = MoveType.Advance;
                    moves.add(newTurn);
                    if(updateColour){
                        nextPiece.isOption = true;
                        ui.UpdateColour(nextPiece);
                    }
                }
            }
        }
        return moves;
    }

    private boolean IsValidDirection(Node n, Piece p){
        if((p.info.isPlayer && playerColour == PieceColour.red) || (!p.info.isPlayer &&
playerColour == PieceColour.white) ){
            //look up
            return ((n.direction == Direction.UpLeft || n.direction == Direction.UpRight)
|| p.info.isKing);
        }
        else if((p.info.isPlayer && playerColour == PieceColour.white) ||
(!p.info.isPlayer && playerColour == PieceColour.red)){
            //look down
            return ((n.direction == Direction.DownLeft || n.direction ==
Direction.DownRight) || p.info.isKing);
        }
        else return false;
    }

    protected List<Move> DoMove(Turn turn, boolean playerMove) {
        List<Move> changes = new ArrayList<Move>();
        if(turn.moveType == MoveType.Advance){
            //create move for fromPiece
            Move m = new Move(turn.origin);
            m.after = new Info(false, false, false);
            turn.origin.info = m.after;
            changes.add(m);
            //create move for toPiece
            Move n = new Move(turn.piece);
            n.after = new Info(playerMove, true, turn.origin.info.isKing);
            turn.piece.info = n.after;
            changes.add(n);
        }
        else if(turn.moveType == MoveType.Jump){
            //create move for fromPiece
            Move m = new Move(turn.origin);
            m.after = new Info(false, false, false);
            turn.origin.info = m.after;
            changes.add(m);
            //create move for toPiece
            Move n = new Move(turn.piece);
            n.after = new Info(playerMove, true, turn.piece.info.isKing);
            turn.piece.info = n.after;
            changes.add(n);
            //create move for captured piece
            for(Piece p : turn.capturedPieces){
                Move c = new Move(p);
                c.after = new Info(false, false, false);
                p.info = c.after;
                changes.add(c);
```

```
            }
        }
        else{
            ui.ShowMessage("There has been an error in TurnHelpers/DoMove. The MoveType
passed in was invalid.", Color.red);
        }
        return changes;
    }

    protected void UndoMove(Turn turn) {
        while(!turn.changes.isEmpty()){
            Move m = turn.changes.remove(0);
            Piece piece = allPieces[m.pieceLocation];
            piece.info = m.before;
        }

    }

    protected boolean isKingNow(Piece piece) {
        int gridSize = piece.getGridSize();
        if((isPlayerTurn && playerColour == PieceColour.white) || (!isPlayerTurn &&
playerColour == PieceColour.red)){
            //is the piece in the bottom row?
            return piece.getLocation() > ((gridSize * gridSize) / 2) - (gridSize / 2);
        }
        else {
            //is the piece in the top row?
            return piece.getLocation() <= gridSize / 2;
        }
    }

    protected void CompleteTurn(Turn turn) {
        //move player piece
        turn.piece.info.isPlayer = isPlayerTurn;
        turn.piece.info.isActive = true;
        turn.piece.info.isKing = turn.origin.info.isKing || turn.piece.info.isKing;
        ui.UpdateColour(turn.piece);

        turn.origin.info.isPlayer = false;
        turn.origin.info.isActive = false;
        turn.origin.info.isKing = false;

        //clear any captured pieces
        turn.capturedPieces.forEach(p -> {
            p.info.isActive = false;
            ui.UpdateColour(p);
        });

        //clear selection formatting
        ClearSelectedPiece(turn);
        IsGameWon();

        isPlayerTurn = !isPlayerTurn;
    }

    protected void ClearSelectedPiece(Turn turn){
        Piece needsClearing = turn.origin;
        needsClearing.isSelected = false;
        turn.origin = null;
        ui.UpdateColour(needsClearing);
    }

    protected void ClearOptions(List<Turn> possibleMoves){
```

```
        if(possibleMoves != null){
            for(Turn t : possibleMoves){
                t.piece.isOption = false;
                t.piece.info.isKing = t.piece.info.isActive ? t.piece.info.isKing :
false;
                ui.UpdateColour(t.piece);
            }
        }
    }

    private void IsGameWon() {
        //it's my turn
        //are all their pieces now captured?
        //if they have any pieces left, are all of them trapped?
        boolean thisPlayerTrapped = true, otherPlayerTrapped = true, otherPlayerCaptured
= true;

        for(Piece p : allPieces){
            if(p == null || !p.info.isActive){
                continue;
            }
            else if(p.info.isActive && p.info.isPlayer == isPlayerTurn){
                //are all my pieces trapped?
                //if so, game over and I've lost
                if(!FilterMoves(p, p.possibleMoves, MoveType.Both).isEmpty()){
                    thisPlayerTrapped = false;
                }
            }
            else if(p.info.isActive && p.info.isPlayer != isPlayerTurn){
                //not all their pieces are captured
                otherPlayerCaptured = false;
                //are all their pieces trapped?
                //if so, game over and I've won
                if(!FilterMoves(p, p.possibleMoves, MoveType.Both).isEmpty()){
                    otherPlayerTrapped = false;
                }
            }
        }
        if(thisPlayerTrapped){
            isPlayerTurn = !isPlayerTurn;
            GameOver("All pieces are trapped");
        }
        else if(otherPlayerCaptured){
            GameOver("All pieces are captured");
        }
        else if(otherPlayerTrapped){
            GameOver("All pieces are trapped");
        }
    }

    //legacy - returned whether a given piece was in danger
    //was fun to write so leaving in, in case I decide to add to the AI
    //probably not needed with how MIN is set up
    protected boolean InDanger(Piece piece) {
        for(Node adjacentNode : piece.possibleMoves){
            Piece playerPiece = allPieces[adjacentNode.pieceLocation];
            if(playerPiece.info.isPlayer != piece.info.isPlayer &&
playerPiece.info.isActive){
                //this piece is the opposite player and is active - possible threat
                //get the node that moves the player piece to the passed in piece
                List<Node> moveNodes = playerPiece.possibleMoves.stream().filter(n ->
n.pieceLocation == piece.getLocation()).collect(Collectors.toList());
                if(moveNodes.isEmpty()){
```

```
                    ui.ShowMessage("There has been an error in TurnHelpers/InDanger.
moveNodes is empty.", Color.red);
                    continue;
                }
                //get the piece that the player piece would move to if it takes the
passed in piece
                List<Node> nextNodes = piece.possibleMoves.stream().filter(n ->
n.direction == moveNodes.get(0).direction).collect(Collectors.toList());
                if(nextNodes.isEmpty()){
                    //piece is at the edge and the player piece can't take it
                    continue;
                }
                Piece oppositePiece = allPieces[nextNodes.get(0).pieceLocation];
                if(!oppositePiece.info.isActive){
                    return true;
                }
            }
        }
        return false;
    }

    protected void GameOver(String message){
        game.isPaused = true;
        message = isPlayerTurn ? "Congratulations! You won! "  + message: "The AI won! "
+ message;
        ui.GameOverDialog(message);
    }
}
```

GamePlay

```
import Classes.*;
import java.awt.*;

public class GamePlay {
    protected UI ui;

    protected Controller controller;

    protected Piece[] allPieces;

    protected boolean isForcedCapture;

    protected Difficulty aiDifficulty;

    private TurnHelpers turnHelpers;

    private PlayerTurn playerTurn;

    private AITurn aiTurn;

    private PieceColour playerColour;

    private boolean isPlayerTurn = false;

    protected boolean isPaused = false;

    public GamePlay(UI _ui, Piece[] _allPieces, PieceColour _playerColour, boolean
_isForcedCapture, Difficulty _aiDifficulty) {
        ui = _ui;
        allPieces = _allPieces;
        playerColour = _playerColour;
        isForcedCapture = _isForcedCapture;
        aiDifficulty = _aiDifficulty;
```

```
        if(playerColour == PieceColour.white){
            AI();
        }
        else{
            isPlayerTurn = true;
        }
    }

    public void pieceClicked(Piece piece) {
        //is it the player's turn?
        if((playerTurn != null && !isPlayerTurn) || isPaused) {
            ui.ShowMessage("The AI is thinking...", Color.darkGray);
            return;
        }



        //are they clicking the first button?
        if (playerTurn == null){
            //have they clicked one of their pieces?
            if(!piece.info.isActive || !piece.info.isPlayer){
                ui.ShowMessage("Select one of your pieces", Color.red);
            }
            else{
                playerTurn = new PlayerTurn(ui, allPieces, this, playerColour, piece);
                playerTurn.ShowOptions();
            }
        }
        //piece has been clicked again, deselect
        else if(playerTurn.turn.origin != null && piece == playerTurn.turn.origin){
            playerTurn.RemoveSelection(piece);
            playerTurn = null;
        }
        //are they clicking the second button?
        else if(playerTurn != null){
            //have they selected an empty square?
            if(piece.info.isActive){
                ui.ShowMessage("Select an empty square", Color.red);
            }
            //have they selected one of the valid moves?
            else if(!piece.isOption)
            {
                ui.ShowMessage("That move is not possible. Select a valid move",
Color.red);
            }
            //validation passed - complete the move!
            else{
                playerTurn.ChooseMove(piece);
                playerTurn = null;
                isPlayerTurn = !isPlayerTurn;
                AI();
            }
        }
    }


    private void AI(){
        if(!isPaused){
            isPaused = true;
            ui.ShowMessage("The AI is thinking...", Color.darkGray);
            aiTurn = new AITurn(ui, allPieces, playerColour, this);
            aiTurn.MakeMove();
            isPlayerTurn = !isPlayerTurn;
```

```
        }
    }



    public void UpdateDifficulty(Difficulty diff) {
        aiDifficulty = diff;
        switch(aiDifficulty){
            case Easy:
                AITurn.aiDepth = 2;
                break;
            case Medium:
                AITurn.aiDepth = 3;
                break;
            case Hard:
                AITurn.aiDepth = 4;
                break;
            case Expert:
                AITurn.aiDepth = 5;
                break;
            case God:
                AITurn.aiDepth = 6;
                break;
        }
    }

    //restart player's move
    protected void RestartMove(Piece piece) {
        playerTurn.RemoveSelection(piece);
        playerTurn = null;
    }
}
```

RoundButton

```
package Components;

import javax.imageio.ImageIO;
import javax.swing.*;
import java.awt.*;
import java.awt.geom.Ellipse2D;

/*
* The foundations of this class were copied from:
* https://www.javacodex.com/More-Examples/2/14
* */

public class RoundButton extends JButton {

    public RoundButton() {
        setFocusable(false);

    /*
     These statements enlarge the button so that it
     becomes a circle rather than an oval.
    */
        Dimension size = getPreferredSize();
        size.width = size.height = Math.max(size.width, size.height);
        setPreferredSize(size);



    /*
     This call causes the JButton not to paint the background.
     This allows us to paint a round background.
```

```
    */
        setContentAreaFilled(false);
    }

    public void SetColour(Color colour){
        setBackground(colour);
    }

    public int GetSize(){return preferredSize().width;}

    protected void paintComponent(Graphics g) {
        if (getModel().isArmed()) {
            g.setColor(Color.gray);
        } else {
            g.setColor(getBackground());
        }
        g.fillOval(0, 0, getSize().width - 1, getSize().height - 1);

        super.paintComponent(g);
    }

    protected void paintBorder(Graphics g) {
        g.setColor(Color.black);
        g.drawOval(0, 0, getSize().width - 1, getSize().height - 1);
    }

    // Hit detection.
    Shape shape;

    public boolean contains(int x, int y) {
        // If the button has changed size,  make a new shape object.
        if (shape == null || !shape.getBounds().equals(getBounds())) {
            shape = new Ellipse2D.Float(0, 0, getWidth(), getHeight());
        }
        return shape.contains(x, y);
    }
}
```

Board

```
package Classes;

public class Board {
    public int getGridSize() {
        return gridSize;
    }

    protected int gridSize = 8;
}
```

Difficulty

```
package Classes;

public enum Difficulty {
    Easy, Medium, Hard, Expert, God
}
```

Direction

```
package Classes;

public enum Direction {
    UpLeft,
    UpRight,
    DownLeft,
```

```
    DownRight
}
```

Info

```
package Classes;

public class Info {
    public boolean isPlayer;

    public boolean isActive;

    public boolean isKing;

    public Info(boolean _isPlayer, boolean _isActive, boolean _isKing) {
        isPlayer = _isPlayer;
        isActive = _isActive;
        isKing = _isKing;
    }
}
```

Move

```
package Classes;

public class Move {
    public int pieceLocation;

    public Info before;

    public Info after;

    public Move(Piece p) {
        before = new Info(p.info.isPlayer, p.info.isActive, p.info.isKing);
        pieceLocation = p.getLocation();
    }
}
```

MoveType

```
package Classes;

public enum MoveType {
    Advance, Jump, Both, Neither
}
```

Node

```
package Classes;

public class Node {
    public int pieceLocation;

    public Direction direction;

    public Node(int pieceLocation, Direction direction) {
        this.pieceLocation = pieceLocation;
        this.direction = direction;
    }
}
```

Piece

```
package Classes;

import Components.RoundButton;

import java.util.*;
```

```
public class Piece extends Tile{

    public Info info = new Info(false, false, false);

    public List<Node> possibleMoves = new LinkedList<Node>() {};;

    public boolean isSelected = false;

    public boolean isOption = false;

    public RoundButton button;
}
```

PieceColour

```
package Classes;

public enum PieceColour {
    white, red
}
```

Priority

```
package Classes;

public enum Priority {
    High, Low, Both
}
```

Tile

```
package Classes;

public class Tile extends Board{
    public int getLocation() {
        return location;
    }

    public void setLocation(int location) {
        this.location = location;
    }

    protected int location;

    public int getRow() {
        return (int)Math.ceil((double)location / ((double)gridSize / 2));
    }

    public int getCol() {
        double x = (double) location / ((double) gridSize / 2);
        int r = (int)(Math.ceil(x));
        double c = (10 * x - 10 * (r - 1))/10 * 8;
        c = r % 2 == 0 ? c - 1 : c;
        return (int)c;
    }
}
```

Turn

```
package Classes;

import java.awt.*;
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;
import java.util.stream.Collectors;
```

```
public class Turn {

    public Piece piece;

    public Piece origin;

    public List<Piece> capturedPieces = new LinkedList<Piece>();

    public List<Move> changes = new ArrayList<Move>();

    public List<Integer> explored = new ArrayList<>();

    public MoveType moveType = MoveType.Neither;

    public double score = 0;

    public Turn(Piece _origin) {
        origin = _origin;
    }


    public Turn Clone() {
        Turn clone = new Turn(this.origin);
        clone.piece = this.piece;
        clone.capturedPieces = new ArrayList<>();
        this.capturedPieces.forEach(p -> clone.capturedPieces.add(p));
        clone.explored = new ArrayList<>();
        this.explored.forEach(i -> clone.explored.add(i));
        clone.score = this.score;
        return clone;
    }
}
```