Intelligent Systems Techniques

1. Introduction

For the purpose of this assignment, I have implemented a fully playable Checkers program in Java, complete with an AI opponent of a variable difficulty level as well as an accompanying graphical user interface.

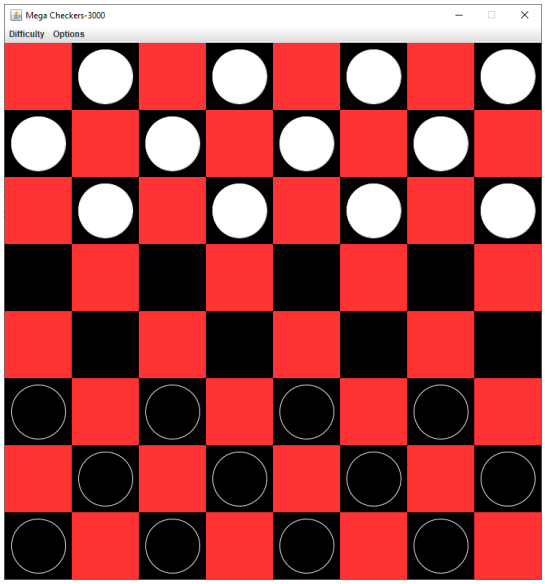

Figure 1 – GUI upon launch

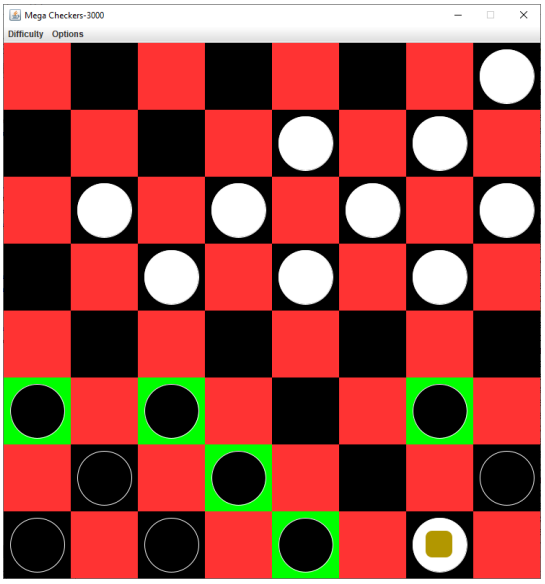

Figure 2 – GUI midway through a game

When the user launches the program, they are presented with the screen visible in Figure 1. Figure 2 is more representative of a game in progress; in which hints for the user are turned on, and a white piece has been promoted to a king after a multi capture move at co-ordinates (*7,8)*.

The underlying algorithm to the AI's gameplay is minimax, in which an adversarial search is performed through all possible succeeding board states, to find that which best benefits itself (and therefore inhibits the human player) in accordance with a heuristic - the difficulty can then simply be adjusted via the maximum permitted search depth the AI can explore.

2. Overview of Program Architecture

The two key classes of this object-oriented checkers implementation are *Game* and *Board.*

Game – This main method can be found in here, where all code execution begins by initialising a *Game* object. Its constructor then creates and provides functionality to the programs menus, as well as initialising a new instance of the *Board* class.

Board – This is by far the most complex class; it is where you'll find the bulk of the game logic. I will go into more detail on some of its key methods in section 3.

In addition to these two classes, there are a few other very lightweight ones such as *Move* or *Checker*, which exists to allow instances of them to be created for various aspects of the program's functionality. Instances of these two of course represent checker pieces, or possible moves that could be made respectively. The purpose of the lightweight CheckerPos class is one that may seem slightly less self-explanatory; however this is to simply associate a checker object with physical position within the GUI. It also uses a co-ordinate system separate to that of the board representation (relative to the GUI as opposed to tiles), therefore allowing the continuous updating of where a checker appears to be whilst click-and-dragging it around the board, without updating the underlying board representation.

3. Description of Program Functionality

1. Gameplay

a. Interactive checkers gameplay (Human user vs. Computer) of some sort

This has been achieved with few shortcomings. The game offers a very intuitive drag-and-drop control system that comes naturally to users, and the AI automatically responds to each move with one of its own in accordance with all the standard rules of checkers. The key shortfall that I must point out is the GUI updates instantly after moves - whilst this leads to snappy feeling gameplay, it can be disorienting during AI-made multi capture moves, or on turns in which a players checker is captured immediately after capturing one belonging to the AI. Within *Board* you can see my attempted implementation of a *pause* method to remedy this, however the pause on each turn would then take place before snapping the players drag-and-dropped checker to the tile centre, meaning that would then only happen at the same time as the AI's move was shown. During AI-made multi capture moves, the pause time would be accumulated before any visual updates, before both moves were once again visually updated at the same time. Because of these two reasons I made the decision to not call this method in my final implementation, as the imperfectly timed delays led to a worse user experience with more clunky feeling gameplay, and I instead eased potential confusion by providing text feedback of the chosen AI move at each step.

b. Different levels of verifiably effective AI cleverness, adjustable by the user

The user has the option of 4 different difficulty levels. As mentioned in section 1, this is controlled by limiting the depth of the adversarial search tree of potential board states that can be evaluated by the AI, this in essence limits how many turns ahead it can look. The selectable difficulties equate to maximum recursion (search) depths of 1, 3, 5 and 15. Differences in behaviour can certainly be seen between the difficulty levels - on the highest setting you will notice the AI is much more likely to protect its pieces by ensuring it has others behind them, particularly with those that could be at risk of being captured, often advancing whilst maintaining a formation much like how the checkers are initially set up. In contrast there appears to be much less logic to the moves made on the lowest difficulty.

2. Search algorithm

a. Appropriate state representation

My underlying state representation is a 2d Char array with 8x8 dimensions to match those of a checkers board. Each element corresponds to a tile, and may contain any of the following characters:

x – A red tile, no checker can ever land on it
o – An empty black tile, open to checkers
w – A standard white checker
b – A standard black checker
W – A King white checker
B – A king black checker

b. Successor function to generate AI moves

The *getAllValidMoves* method within the *Board* class serves as my successor function. It returns an arraylist of all possible *Move* objects for a given team and board state. As well as being used to determine if the games termination conditions have been met, it is called from within my *minimax* method to populate a list of all potential succeeding moves, before their resulting board states are recursively evaluated, and succeeding move chosen accordingly.

c. Minimax evaluation

My *minimax* method can also be found within the *Board* class, it takes as input parameters a board representation, maximum search depth, maximising team, as well as alpha and beta values. For the given board representation, it explores all possible moves for the maximising team, evaluating the resulting board state in accordance with a simple heuristic to determine the best move. This is done recursively on these potential future board states until the maximum search depth dictated by the set difficulty level has been reached, or no more moves are available.

d. Alpha-Beta pruning

My AB pruning can of course be found within the *minimax* method, through the use of *if* statements comparing them to *maxScore*, Alpha keeps track of the best score that the maximising player could currently guarantee at that level or above, whilst Beta does the same with the minimizing player. The loop and therefore that chain of recursion is then broken if alpha becomes greater than or equal to beta (after being initialized at -∞ and ∞ respectively), as these represent suboptimal branches in the game tree for the maximising player, and there is nothing to be gained from exploring and evaluating them. This significantly increases computational efficiency allowing much greater search depths to be exploited.

e. Appropriate use of heuristics

The *getHeuristic* method, also found within the *Board* class, returns the zero-sum score for a given board representation and a given team. I gave each standard checker a weighting of 1, meaning it gives its team a score of 1, and -1 to the opposing team. I decided that 3 was a suitable weighting for a king, as despite having exactly double the movement options, I see those of a king to be more valuable, since with the exception of regicide cases, they begin behind the opposing checkers, from which they are largely defenceless only being able to capture forwards.

3. Validation of moves

a-c. Only valid moves permitted by both human and AI, with specific explanation given to user in the case of an attempted invalid move.

The *validMove* method within the *Board* class takes as input parameters a board representation, as well as initial and final x and y coordinates, and returns a Boolean to dictate whether the described move is valid or not. One final input parameter is the Boolean *errMsg*, which dictates if an explanation should be given if the described move is invalid. This allows the same method to regulate moves made by both the human and AI, without addressing the user with messages regarding attempted moves by the AI.

d. Forced capture - an opportunity to capture an enemy piece has to be taken. If there is more than one capturing opportunity in the same turn, the user may choose which one to take.

getAllValidMoves first populates an array list with possible capturing moves that could be made for the given team and board representation. Only if this array list is empty (no capture moves available) is a second array list then populated with non-capturing moves in accordance with the *validMove* method described above. This way no normal move becomes a valid option if there is a capturing one available, and should there be multiple capturing moves available, they will all be present in the array list, and therefore the users choice as to which one is executed.

4. Other features

a. Multi-step capturing moves for the user

Support for this can be found in the *playTurn* method. If a capture move is performed, and another one now exists at the new co-ordinates of the capturing checker, the Boolean dictating who's turn it is isn't flipped, allowing the user to perform that additional capture move.

b. Multi-step capturing moves for the AI

These moves are handled similarly for the AI, however the relevant code within *playTurn* simply sets a Boolean *AIextraCapMove* to true, which triggers a section of code that follows the call to *playTurn* within the *mouseReleased* event. To implement the drag-and-drop playstyle, much of the top-level functionality had to be put within this event. In the case of permitting the extra AI move, I found that if I left the relevant code all within *playTurn,* a second release of the mouse was needed to trigger the second move (but any user move made by that mouser release wasn't counted). I'm sure I could reduce the coupling between these two methods in this respect, however it achieves the required functionality and I don't see it to be an issue with this implementation.

c. King conversion at baseline (The king's row) as per the normal rules

checkForKing is the method responsible here. It simply takes a board representation as its input and scans through both of the king's rows to determine if there are any non-native checkers there, and if that is the case, they are promoted to kings. This is done both in terms of the board representation, as well as the corresponding checkerPos insance to keep the GUI in sync.

d. Regicide - if a normal piece manages to capture a king, it is instantly crowned king and then the current turn ends.

The code that handles regicide can be found within the *capture* method. This method is called whenever a capturing move is made, and so it seemed an appropriate place to make these checks. If the captured piece was a king, the capturing checker is promoted to a king both in terms of the board representation, and it's associated checkerPos instance.

e. A help facility that provides hints about available moves, given the current game state. For instance, when enabled, any squares containing movable pieces might get a different colour.

As you can see in figure 2 in the Introduction, if hints are toggled on, any tiles containing a movable checker are highlighted in green to clearly indicate them to the user.

4. Appendix

Game.java

```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package istcheckers;

/**
 *
 * @author 213120
 */

import java.awt.EventQueue;
import java.awt.Font;
import java.awt.Graphics;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.ButtonGroup;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;
import javax.swing.JOptionPane;
import javax.swing.JPanel;
import javax.swing.JRadioButtonMenuItem;
import javax.swing.UIManager;



public class Game extends JFrame{
    public static int maxDepth = 1;
    public static boolean firstTurn = true;
    JMenuBar menuBar;

    //Creates new Game instance
    public static void main(String[] args) {
        Runnable r = new Runnable()
                    {
                       @Override
                       public void run()
                       {
                          new Game();
                       }
                    };
      EventQueue.invokeLater(r);
    }

    //Constructor for new Game, initialises new Board
    public Game() {
        super("Mega Checkers-3000");
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        Board board = new Board();
        board.resetBoard();
        setContentPane(board);
        setResizable(false);
        createMenu(board);

        pack();
        setVisible(true);
    }
```

```
    //Creates the menu
    public void createMenu(Board board) {
        menuBar = new JMenuBar();
        //Difficulty drop down
        JMenu difficulty = new JMenu("Difficulty");

        //Difficulty selection radio buttons
        ButtonGroup difficultyGroup = new ButtonGroup();
        JRadioButtonMenuItem rbEasy = new JRadioButtonMenuItem("Piece of Cake");
        JRadioButtonMenuItem rbMedium = new JRadioButtonMenuItem("Let's Rock");
        JRadioButtonMenuItem rbHard = new JRadioButtonMenuItem("Come Get Some");
        JRadioButtonMenuItem rbVHard = new JRadioButtonMenuItem("Damn I'm good!");

        //Action listeners to provide functionality
        rbEasy.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                resetBoard();
                System.out.println("Don't worry, I remember my first time");
                maxDepth = 1;
            }
        });
        rbMedium.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                resetBoard();
                System.out.println("Getting the hang of this game? I'll still play
with one hand tied behind my back");
                maxDepth = 3;
            }
        });
        rbHard.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                resetBoard();
                System.out.println("Let's see what you've got");
                maxDepth = 5;
            }
        });
        rbVHard.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                resetBoard();
                System.out.println("Prepare to be humiliated!");
                maxDepth = 15;
            }
        });

        //Group all radio buttons
        difficultyGroup.add(rbEasy);
        difficultyGroup.add(rbMedium);
        difficultyGroup.add(rbHard);
        difficultyGroup.add(rbVHard);

        //Ensure correct one is selected
        if(maxDepth == 1) {
            rbEasy.setSelected(true);
            rbMedium.setSelected(false);
            rbHard.setSelected(false);
            rbVHard.setSelected(false);
        } else if (maxDepth == 3) {
            rbEasy.setSelected(false);
            rbMedium.setSelected(true);
            rbHard.setSelected(false);
            rbVHard.setSelected(false);
        } else if (maxDepth == 5) {
            rbEasy.setSelected(false);
```

```
            rbMedium.setSelected(false);
            rbHard.setSelected(true);
            rbVHard.setSelected(false);
        } else if (maxDepth == 15) {
            rbEasy.setSelected(false);
            rbMedium.setSelected(false);
            rbHard.setSelected(false);
            rbVHard.setSelected(true);
        }
        //Add the group to difficulty tab
        difficulty.add(rbEasy);
        difficulty.add(rbMedium);
        difficulty.add(rbHard);
        difficulty.add(rbVHard);
        menuBar.add(difficulty);

        //Create Options dropdown
        JMenu options = new JMenu("Options");
        menuBar.add(options);

        //Add reset Board as option
        JMenuItem resetBoard = new JMenuItem("Reset Board");
        options.add(resetBoard);
        resetBoard.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e ) {
                resetBoard();
            }
        });

        //Add instructions to options
        JMenuItem howToPlay = new JMenuItem("How To Play");
        options.add(howToPlay);
        howToPlay.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e ) {
                UIManager.put("OptionPane.messageFont",      new      Font("Arial",
Font.PLAIN, 17));
                String html = "<html><body width='%1s'><h1>Rules</h1>"
                + "<ul>"

                + "<li>Drag and drop checkers with your mouse to play. </li>"
                + "<br>"
                + "<li>The objective of the game is to create a situation in which
it is impossible for your opponent to make any move, usually due to complete
elimination. </li>"
                + "<br>"
                +  "<li>From  their  initial  positions,  checkers  may  only  move
diagonally and forwards - therefore they only ever land on the black squares.</li>"
                + "<br>"
                + "<li>A checker can only move one sqaure, unless they are performing
a capture move. This is where a checker captures another piece by jumping over it
diagonally onto a vacant square.</li>"
                + "<br>"
                + "<li>On a capturing move, a piece may make multiple consecutive
jumps. If after a jump the piece is in a position to make another jump then they may
do so. This means that a player may make several jumps in succession, capturing
several pieces during a single turn.</li>"
                + "<br>"
                + "<li>If a player is in a position to make a capturing move, they
must make a capturing move. If multiple are available, it is the players choice as
to which one.</li>"
                + "<br>"
                + "<li>When a checker reaches the opponents end of the board, they
become a King, signified by the gold crown on the checker.</li>"
                + "<br>"
```

```
                + "<li>Kings gains an added ability to move diagonally
backwards.</li>"
                + "</ul>";
                int w = 500;
                JOptionPane.showMessageDialog(null, String.format(html, w, w));
            }
        });

        //Add hints to options
        JMenuItem showHints = new JMenuItem("Toggle Hints");
        options.add(showHints);
        showHints.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e ) {
                if (firstTurn) {
                    System.out.println("Have a go first! You can move any on the
upper row.");
                } else {
                    board.toggleHints();
                    board.updateGUI();
                }
            }
        });
        //display the completed menu
        this.setJMenuBar(menuBar);
    }


    public void resetBoard() {
        System.out.println("Board Reset");
        dispose();
        new Game();
    }

    //Getters and Setters
    public static int getMaxDepth() {
        return maxDepth;
    }


    public static void infoBox(String infoMessage, String titleBar){

        JOptionPane.showMessageDialog(null,    infoMessage,    ""    +    titleBar,
JOptionPane.INFORMATION_MESSAGE);

    }

}
```

Checker.java

```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package istcheckers;

import java.awt.Color;
import java.awt.Graphics;
import javax.swing.JComponent;

/**
 *
```

```
 * @author 213120
 */
public class Checker{
    private final static int dim = 75;
    public CheckerType checkerType;

    public Checker(CheckerType checkerType) {
        this.checkerType = checkerType;
    }


    public void erase(Graphics g) {
        g.dispose();
    }

    public void draw(Graphics g, int col, int row) {
        int x = col - dim/2;
        int y = row - dim/2;
        //Set Colour
        g.setColor(checkerType == CheckerType.BLACK_REGULAR || checkerType ==
CheckerType.BLACK_KING ? Color.BLACK : Color.WHITE);
        //Draw
        g.fillOval(x, y, dim, dim);
        g.setColor(Color.WHITE);
        g.drawOval(x, y, dim, dim);

      if   (checkerType   ==   CheckerType.WHITE_KING   ||   checkerType   ==
CheckerType.BLACK_KING) {
          g.setColor(Color.decode("#B29700"));
          //g.fillOval(x + dim/4, y + dim/4, dim/2, dim/2);
          g.fillRoundRect(x+dim/4, y+dim/4, dim/2, dim/2, dim/4, dim/4);
           //g.drawString("K", col, row);
      }

    }

    public static int getDimension()
   {
      return dim;
   }

    public static boolean contains(int x, int y, int col, int row)
   {
      return (col - x) * (col - x) + (row - y) * (row - y) < dim / 2 * dim /
2;
   }

}
```

CheckerPos.java

```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package istcheckers;

/**
 *
 * @author 213120
 */
```

```
public class CheckerPos {
    public Checker checker;
    public int x;
    public int y;
    public boolean Lcap;
    public boolean Rcap;
}
```

CheckerType.java

```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package istcheckers;

/**
 *
 * @author 213120
 */
public enum CheckerType {
   BLACK_REGULAR,
   BLACK_KING,
   WHITE_REGULAR,
   WHITE_KING
}
```

AlreadyOccupiedException.java

```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package istcheckers;

/**
 *
 * @author 213120
 */
public class AlreadyOccupiedException extends RuntimeException
{
   public AlreadyOccupiedException(String message)
   {
      super(message);
   }
}
```

Move.java

```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package istcheckers;
```

```
/**
 *
 * @author 213120
 */
public class Move {
    public int startX;
    public int startY;
    public int endX;
    public int endY;
    //public boolean isCap;

    public Move(int x, int y, int X, int Y) {
        this.startX = x;
        this.startY = y;
        this.endX = X;
        this.endY = Y;
        //isCap = cap;
    }

    public int getStartX() {
        return startX;
    }

    public int getStartY() {
        return startY;
    }

    public int getEndX() {
        return endX;
    }

    public int getEndY() {
        return endY;
    }

//    public boolean isCapMove() {
//        return isCap;
//    }
}
```

Board.java

```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package istcheckers;
//Necessary imports
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.RenderingHints;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
import java.awt.event.MouseMotionAdapter;
import static java.lang.Integer.min;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Iterator;
import java.util.List;
import javax.swing.JComponent;
```

```
/**
 *
 * @author 213120
 */
public class Board extends JComponent{
    private final static int TileDim = (int) (Checker.getDimension() * 1.25); //Tile
dimension, 25% larger than a checker
    private final int BoardDim = 8 * TileDim;
    private Dimension dimSize;
    //Click + Drag support
    private boolean beingDragged = false;
    private int disX, disY;                         //distance between checker centre
+ start of drarg
    private CheckerPos checkerPos;                   //start of drag checker pos
    private int oldX, oldY;                          // centre loc of above
    private List<CheckerPos> checkerPoss;           //Array list checker rep for GUI
    List<CheckerPos> toRemove = new ArrayList<CheckerPos>();
    private List<Move> moves = new ArrayList<>();   //possible non capturing moves
    private List<Move> movesCap = new ArrayList<>(); //possible capturing moves
    private List<Move> allValidMoves = new ArrayList<>(); //all possible moves
    //private List<MovesAndScores> successorEvaluations; //List of possible moves
with associated scores
    public boolean occupied[][] = new boolean[80][80];
    private char boardRep[][];                      //Main board representation
    public int maxDepth = Game.getMaxDepth();        //Tree search depth - dictates
difficulty
    int bestOldX, bestOldY, bestNewX, bestNewY;      //Components of selected best
move
    private boolean hintsDisplayed = false;
    boolean playerTurn = true;                      //Tracks who's turn it is
    boolean capturePerformed = false;                //Used to track when a second
turn should be allowed
    boolean AIextraCapMove = false;                  //Tracks when AI is allowed a
second turn
    boolean regicide = false;                       //Used to handle promotion when
king is captured by normal checker


    public Board() {
        checkerPoss = new ArrayList<>();
        dimSize = new Dimension(BoardDim, BoardDim);

        // x = Red tiles, can't be used at any time
        // o = Available black tile
        // w = White Checker, b = Black Checker
        // W = White king, B = Black king
        boardRep = new char[][] {
            {'x', 'w', 'x', 'w', 'x', 'w', 'x', 'w'},
            {'w', 'x', 'w', 'x', 'w', 'x', 'w', 'x'},
            {'x', 'w', 'x', 'w', 'x', 'w', 'x', 'w'},
            {'o', 'x', 'o', 'x', 'o', 'x', 'o', 'x'},
            {'x', 'o', 'x', 'o', 'x', 'o', 'x', 'o'},
            {'b', 'x', 'b', 'x', 'b', 'x', 'b', 'x'},
            {'x', 'b', 'x', 'b', 'x', 'b', 'x', 'b'},
            {'b', 'x', 'b', 'x', 'b', 'x', 'b', 'x'},
        };

        addMouseListener(new MouseAdapter()
                    {
                       @Override
                       public void mousePressed(MouseEvent me)
                       { //Initial Mouse co-ords when clicked
                          int x = me.getX();
                          int y = me.getY();

                          //Find black checker at those co-ords
```

```
                            for (CheckerPos checkerPos: checkerPoss)
                               if     (Checker.contains(x,     y,     checkerPos.x,
checkerPos.y))// && ((checkerPos.checker.checkerType == CheckerType.BLACK_REGULAR ||
checkerPos.checker.checkerType == CheckerType.BLACK_KING)))
                                  {
                                     Board.this.checkerPos = checkerPos;
                                     oldX = checkerPos.x;
                                     oldY = checkerPos.y;
                                     disX = x - checkerPos.x;
                                     disY = y - checkerPos.y;
                                     beingDragged = true;
                                     return;
                                  }
                            }

                            @Override
                            public void mouseReleased(MouseEvent me)
                            {
                               //Update variable
                               if (beingDragged)
                                  beingDragged = false;
                               else
                                  return;
                               //Snap checker to tile centre
                               int x = me.getX();
                               int y = me.getY();
                               checkerPos.x = (x - disX) / TileDim * TileDim +
                                             TileDim / 2;
                               checkerPos.y = (y - disY) / TileDim * TileDim +
                                             TileDim / 2;

                               //Check move is valid
                               if       (!validMove(oldX/TileDim,       oldY/TileDim,
checkerPos.x/TileDim, checkerPos.y/TileDim, boardRep, true)) {
                                   Board.this.checkerPos.x = oldX;
                                   Board.this.checkerPos.y = oldY;
                               } else {
                                   //pause(); //Attempt to pause gui between turns,
kinda works (not very well), gives clunky feel to gameplay
                                   Game.firstTurn = false;
                                   playTurn();

                                   //Handles  the  AI's  consecutive  capture  moves,
without each having to be triggered by mouse release
                                   if (AIextraCapMove) {
                                      minimax(boardRep, 0, "white", Integer.MIN_VALUE,
Integer.MAX_VALUE);

System.out.println(Arrays.deepToString(boardRep));
                                      updateBoardRep(bestOldX*TileDim,
bestOldY*TileDim, bestNewX*TileDim, bestNewY*TileDim);
                                      checkForKing(boardRep);
                                      updateGUI();
                                      repaint();
                                      AIextraCapMove = false;
                                      playerTurn = true;
                                   }

                                   //Console feedback
                                   //System.out.println("Valid   black   moves:   "   +
getAllValidMoves("black", boardRep).size());
                                   //System.out.println("Valid   white   moves:   "   +
getAllValidMoves("white", boardRep).size());
                                   System.out.println("Chosen AI Move: " + (bestOldX +
1) + "," + (bestOldY + 1) + " to " + (bestNewX + 1) + "," + (bestNewY + 1)); //Adjust
from 0 indexing to 1
```

```
                                //Winning conditions
                                if (blackHasWon()) {
                                    Game.infoBox("BLACK WINS - We'll crank up the
difficulty for next time!", "Game Over!");
                                }
                                if (whiteHasWon()) {
                                    Game.infoBox("WHITE  WINS  -  Maybe  stick  to
TicTacToe", "Game Over!");
                                }

                            }
                            checkerPos = null;
                            repaint();
                        }
                    });
        addMouseMotionListener(new MouseMotionAdapter()
                            {
                               @Override
                               public void mouseDragged(MouseEvent me)
                               {
                                  if (beingDragged)
                                  {
                                     // Update checker ventre loc
                                     checkerPos.x = me.getX() - disX;
                                     checkerPos.y = me.getY() - disY;
                                     repaint();
                                  }
                               }
                            });
    }

    //Attempt to pause play + update GUI between turns
    //Pauses before snapping moved checker to centre, and failed to use it to update
GUI between multicap moves
    public void pause() {
        //new Thread(() -> {
            System.out.println("Dramatic paaaauuuuuuse....");
            try {
                //resume = false;
                Thread.sleep(1000);
            } catch (InterruptedException e){

                updateGUI();
                repaint();
            }
            //resume= true;
        //}).start();
    }

    //Controls who should play when - responsible for allowing multicap turns
    public void playTurn(){
        int newX = checkerPos.x;
        int newY = checkerPos.y;
        int capX = ((oldX + newX)/ 2);
        int capY = ((oldY + newY)/2);

        //Apply user turn
        if (playerTurn) {
            updateBoardRep(oldX, oldY, checkerPos.x, checkerPos.y);
            checkForKing(boardRep);

            //If capturing checker is able to capture another, another turn is allowed
            if  (capturePerformed  &&  captureMoveAt(boardRep,  checkerPos.x/TileDim,
checkerPos.y/TileDim)) {
                System.out.println("Keep it going! You can take another checker!");
            } else {
            playerTurn = false;
```

```
            }
        }

        //Computers Turn
        if (!playerTurn){
        minimax(boardRep, 0, "white", Integer.MIN_VALUE, Integer.MAX_VALUE);
        updateBoardRep(bestOldX*TileDim,     bestOldY*TileDim,     bestNewX*TileDim,
bestNewY*TileDim);
        checkForKing(boardRep);
        //If capturing checker is able to capture another, another turn is allowed
            if (capturePerformed && captureMoveAt(boardRep, bestNewX, bestNewY)) {
                System.out.println("ANOTHER GO TO COMPUTER");
                updateGUI();
                repaint();
                //pause();
                AIextraCapMove = true;
                //Thread.sleep(1000);
            } else {
                AIextraCapMove = false;
                playerTurn = true;
            }
        }
        //UI + book keeping needed regardless of who's turn it is
        capturePerformed = false;
        updateGUI();
        repaint();
        //System.out.println(Arrays.deepToString(boardRep));
        regicide = false;
    }

    //Performs a capturing move
    public void capture(char[][] board, int oldX, int oldY, int newX, int newY) {
        int capX = ((oldX + newX)/ 2);
        int capY = ((oldY + newY)/2);
        Iterator<CheckerPos> iter = checkerPoss.iterator();

        //If a capturing move has been made
        if (Math.abs(newX - oldX) == 2 * TileDim && Math.abs(newY - oldY) == 2 *
TileDim) {
            System.out.println("You're   mine!   at   "   +   capX/TileDim   +   ","   +
capY/TileDim);
            capturePerformed = true;

            //Regicide - Promote piece to king if it caps another king
            if (board[capY/TileDim][capX/TileDim] == 'W') {
                regicide = true;
                //System.out.println("KING CAPPED");
                boardRep[newY/TileDim][newX/TileDim] = 'B';

                for (CheckerPos checkerPos: checkerPoss) {
                    if (checkerPos.y == newY && checkerPos.x == newX) {
                        System.out.println("CHECKERPOS UPDATED to king");
                        checkerPos.checker.checkerType = CheckerType.BLACK_KING;
                        repaint();
                    }
                }
            }
            //Regicide for the AI (white) Team
            if (board[capY/TileDim][capX/TileDim] == 'B') {
                regicide = true;
                System.out.println("KING CAPPED");
                boardRep[newY/TileDim][newX/TileDim] = 'W';
                System.out.println("cap: " + Arrays.deepToString(boardRep));
                //Promote correct checkerPos type to king
                for (CheckerPos checkerPos: checkerPoss) {
                    if (checkerPos.y == newY && checkerPos.x == newX) {
                        //System.out.println("CHECKERPOS UPDATED to king");
```

```
                        checkerPos.checker.checkerType = CheckerType.WHITE_KING;
                        repaint();
                    }
                }
            }

            //Update Board Rep + GUI
            board[capY/TileDim][capX/TileDim] = 'o';
            for (CheckerPos checkerPos: checkerPoss) {
                if (checkerPos.x == capX && checkerPos.y == capY) {
                    checkerPoss.remove(checkerPos);
                    //toRemove.add(checkerPos);
                    updateGUI();
                    repaint();
                    break;
                }
            }
        }
    }

    //Promotes checkers who have reached opposing side of board
    public void checkForKing(char[][] board) {
        for (int x = 0; x < 8; x++) {
            if (board[0][x] == 'b') {
                board[0][x] = 'B';
                for (CheckerPos checkerPos: checkerPoss) {
                    if (checkerPos.y == TileDim /2 && checkerPos.checker.checkerType
== CheckerType.BLACK_REGULAR) {
                        checkerPos.checker.checkerType = CheckerType.BLACK_KING;
                        repaint();
                    }
                }
            }
            if (boardRep[7][x] == 'w') {
                boardRep[7][x] = 'W';
                //System.out.println("REP UPDATED");
                for (CheckerPos checkerPos: checkerPoss) {
                    if    (checkerPos.y    ==    BoardDim    -    TileDim/2    &&
checkerPos.checker.checkerType == CheckerType.WHITE_REGULAR) {
                        checkerPos.checker.checkerType = CheckerType.WHITE_KING;
                        repaint();
                    }
                }
            }
        }
    }

    //Dictate whether the described move is valid
    public boolean validMove(int x, int y, int X, int Y, char[][] board, boolean
errMsg) {
        boolean valid = true;
        int oldX = x;
        int oldY = y;
        int newX = X;
        int newY = Y;

        //Capture moves must be made if available
        if(board[oldY][oldX] == 'b' || board[oldY][oldX] == 'B') {
            if (existsCapMove(board, "black") && !captureMovePerformed(board, oldX,
oldY, newX, newY)) {
                if (errMsg) System.out.println("If you can capture a piece, you
must!");
                valid = false;
            }
        }
        if(board[oldY][oldX] == 'w' || board[oldY][oldX] == 'W') {
            //System.out.println("MOVING A WHITE PIECE");
```

```
            if (existsCapMove(board, "white") && !captureMovePerformed(board, oldX,
oldY, newX, newY)) {
                if (errMsg) System.out.println("If you can capture a piece, you
must!");
                valid = false;
            }
        }

        //Prevents moving out of bounds
        if (newX > 7 || newX <0 || newY > 7 || newY < 0) {
            if (errMsg) System.out.println("Are you trying to run away? That's out
of bounds!");
            return false;
        }
         //Checks tile is not already occupied
        if (board[newY][newX] == 'b' || board[newY][newX] == 'w' || board[newY][newX]
== 'B' || board[newY][newX] == 'W') {
            valid = false;
            if (errMsg) System.out.println("Invalid move! This square ain't big
enough for the both of us!");
        }
         //Prevent moving in a straight line
        if (newX == oldX || newY == oldY) {
            valid = false;
            if (errMsg) System.out.println("Nope! You gotta move diagonally!");
        }
        //Only allow moving forwards
        if (board[oldY][oldX] == 'b' && newY > oldY){
            valid = false;
            if (errMsg) System.out.println("You can only move forwards!");
        }
        if (board[oldY][oldX] == 'w' && newY < oldY){
            valid = false;
            if (errMsg) System.out.println("You can only move forwards!");
        }
        //Forbid moving more than one space away`
        if ((Math.abs(newX - oldX) > 1 || Math.abs(newY - oldY) > 1) &&
!captureMoveAt(board, oldX, oldY)){
            valid = false;
            if (errMsg) System.out.println("Woah slow down there buddy! You can't
move more than 1 square away!");
        } else if (Math.abs(newX - oldX) > 2 || Math.abs(newY - oldY) > 2) {
            valid = false;
            if (errMsg) System.out.println("Woah slow down there buddy! You can't
move more than 2 squares away when taking a piece!");
        }
        return valid;
    }

    //Creates an arrayList of all valid Moves for a given team and board
representation
    public List<Move> getAllValidMoves(String team, char[][] board) {
        movesCap.clear();
        moves.clear();
        allValidMoves.clear();
        //Adds all capturing moves
        for (int i = 0; i<8; i++) {
            for (int j = 0; j < 8; j++) {
                if ((team == "white" && (board[i][j] == 'w' || board[i][j] == 'W'))
|| (team == "black" && (board[i][j] == 'b' || board[i][j] == 'B'))) {
                    if (captureMovePerformed(board, j, i, j + 2, i + 2)) {
                            movesCap.add(new Move(j, i, j+2, i+2));
                        }
                        if (captureMovePerformed(board, j, i, j - 2, i + 2)) {
                            movesCap.add(new Move(j, i, j-2, i+2));
                        }
                        if (captureMovePerformed(board, j, i, j + 2, i - 2)) {
```

```
                            movesCap.add(new Move(j, i, j+2, i-2));
                        }
                        if (captureMovePerformed(board, j, i, j - 2, i - 2)) {
                            movesCap.add(new Move(j, i, j-2, i-2));
                        }
                    }
                }
            }
            //Adds non capturing moves if no capturing ones exist
            if (movesCap.isEmpty()) {
                for (int i = 0; i<8; i++) {
                    for (int j = 0; j < 8; j++) {
                        if ((team == "white" && (board[i][j] == 'w' || board[i][j] ==
'W')) || (team == "black" && (board[i][j] == 'b' || board[i][j] == 'B'))) {
                            if (validMove(j, i, j + 1, i + 1, board, false)) {
                                moves.add(new Move(j, i, j+1, i+1));
                            }
                            if (validMove(j, i, j - 1, i + 1, board, false)) {
                                moves.add(new Move(j, i, j-1, i+1));
                            }
                            if (validMove(j, i, j + 1, i - 1, board, false)) {
                                moves.add(new Move(j, i, j+1, i-1));
                            }
                            if (validMove(j, i, j - 1, i - 1, board, false)) {
                                moves.add(new Move(j, i, j-1, i-1));
                            }
                        }
                    }
                }
                //Add to one super-list to ease referencing
                allValidMoves.addAll(moves);
                return moves;
            } else {
                allValidMoves.addAll(movesCap);
                return movesCap;
            }
        }


        //Dictates if the move just performed was a capturing move
        public boolean captureMovePerformed(char[][] board, int x, int y, int newX, int
newY) {
            boolean capMove = false;

            //Black caps to uper left
            try {
            if ((board[y][x] == 'b' || board[y][x] == 'B') && ((board[y-1][x-1] == 'w'
|| board[y-1][x-1] == 'W') && board[y-2][x-2] == 'o')) {
                if (newX == x-2 && newY == y-2) {
                    capMove = true;
                }
            }} catch(ArrayIndexOutOfBoundsException e) {}
            //Black caps to upper right
            try {
            if((board[y][x] == 'b'|| board[y][x] == 'B') && ((board[y-1][x+1] == 'w'||
board[y-1][x+1] == 'W') && board[y-2][x+2] == 'o')) {
                if (newX == x+2 && newY == y-2) {
                    capMove = true;
                }
            }} catch(ArrayIndexOutOfBoundsException e) {}
            //White caps to lower left
            try{
            if ((board[y][x] == 'w' || board[y][x] == 'W') && ((board[y+1][x-1] == 'b'
|| board[y+1][x-1] == 'B') && board[y+2][x-2] == 'o')){
                if (newX == x-2 && newY == y+2) {
                    capMove = true;
                }
```

```
        }} catch(ArrayIndexOutOfBoundsException e) {}
        //White caps to lower right
        try {
        if ((board[y][x] == 'w' || board[y][x] == 'W') && ((board[y+1][x+1] == 'b'
|| board[y+1][x+1] == 'B') && board[y+2][x+2] == 'o')) {
            if (newX == x+2 && newY == y+2) {
                capMove = true;
            }
        }} catch(ArrayIndexOutOfBoundsException e) {}


        //Black king lower left caps
        try {
        if ((board[y][x] == 'B')&& ((board[y+1][x-1] == 'w' || board[y+1][x-1] ==
'W') && board[y+2][x-2] == 'o')) {
            if (newX == x-2 && newY == y+2) {
                capMove = true;
            }
        }}catch(ArrayIndexOutOfBoundsException e) {}
        //black king lower right caps
        try {
        if ((board[y][x] == 'B')&& ((board[y+1][x+1] == 'w' || board[y+1][x+1] ==
'W') && board[y+2][x+2] == 'o')) {
            if (newX == x+2 && newY == y+2) {
                capMove = true;
            }
        }}catch(ArrayIndexOutOfBoundsException e) {}
        //White king upper left
        try {
        if ((board[y][x] == 'W')&& ((board[y-1][x-1] == 'b' || board[y-1][x-1] ==
'B') && board[y-2][x-2] == 'o')) {
            if (newX == x-2 && newY == y-2) {
                capMove = true;
            }
        }}catch(ArrayIndexOutOfBoundsException e) {}
        //White king upper right
        try {
        if ((board[y][x] == 'W')&& ((board[y-1][x+1] == 'b' || board[y-1][x+1] ==
'B') && board[y-2][x+2] == 'o')) {
            if (newX == x+2 && newY == y-2) {
                capMove = true;
            }
        }}catch(ArrayIndexOutOfBoundsException e) {}
        //System.out.println("Capture move performed: " + capMove);
        return capMove;
    }

    //Dictates if a capture move exists at the given co-ordinates
    public boolean captureMoveAt(char[][] board, int x, int y) {
        boolean capMove = false;

        //Black caps to uper left
        try {
        if ((board[y][x] == 'b' || board[y][x] == 'B') && ((board[y-1][x-1] == 'w'
|| board[y-1][x-1] == 'W') && board[y-2][x-2] == 'o')) {
            capMove = true;
        }} catch(ArrayIndexOutOfBoundsException e) {}
        //Black caps to upper right
        try {
        if((board[y][x] == 'b'|| board[y][x] == 'B') && ((board[y-1][x+1] == 'w'||
board[y-1][x+1] == 'W') && board[y-2][x+2] == 'o')) {
            capMove = true;
        }} catch(ArrayIndexOutOfBoundsException e) {}
        //White caps to lower left
        try{
        if ((board[y][x] == 'w' || board[y][x] == 'W') && ((board[y+1][x-1] == 'b'
|| board[y+1][x-1] == 'B') && board[y+2][x-2] == 'o')){
```

```
            capMove = true;
        }} catch(ArrayIndexOutOfBoundsException e) {}
        //White caps to lower right
        try {
        if ((board[y][x] == 'w' || board[y][x] == 'W') && ((board[y+1][x+1] == 'b'
|| board[y+1][x+1] == 'B') && board[y+2][x+2] == 'o')) {
            capMove = true;
        }} catch(ArrayIndexOutOfBoundsException e) {}


        //Black king lower left caps
        try {
        if ((board[y][x] == 'B')&& ((board[y+1][x-1] == 'w' || board[y+1][x-1] ==
'W') && board[y+2][x-2] == 'o')) {
            capMove = true;
        }}catch(ArrayIndexOutOfBoundsException e) {}
        //black king lower right caps
        try {
        if ((board[y][x] == 'B')&& ((board[y+1][x+1] == 'w' || board[y+1][x+1] ==
'W') && board[y+2][x+2] == 'o')) {
            capMove = true;
        }}catch(ArrayIndexOutOfBoundsException e) {}
        //White king upper left
        try {
        if ((board[y][x] == 'W')&& ((board[y-1][x-1] == 'b' || board[y-1][x-1] ==
'B') && board[y-2][x-2] == 'o')) {
            capMove = true;
        }}catch(ArrayIndexOutOfBoundsException e) {}
        //White king upper right
        try {
        if ((board[y][x] == 'W')&& ((board[y-1][x+1] == 'b' || board[y-1][x+1] ==
'B') && board[y-2][x+2] == 'o')) {
            capMove = true;
        }}catch(ArrayIndexOutOfBoundsException e) {}

        return capMove;
        }

    //Dictates if a capture move exists anywhere on the board for a given team
    public boolean existsCapMove(char[][] board, String team) {
        boolean capMove = false;
        if (team == "black") {
            for (int i = 0; i<8; i++) {
                for (int j = 0; j<8; j++){
                    if  (captureMoveAt(board,  i,  j)  &&  (board[j][i]  ==  'b'  ||
board[j][i] == 'B')) {
                        capMove = true;
                    }
                }
            }
        }
        if (team == "white") {
            for (int i = 0; i<8; i++) {
                for (int j = 0; j<8; j++){
                    if  (captureMoveAt(board,  i,  j)  &&  (board[j][i]  ==  'w'  ||
board[j][i] == 'W')) {
                        capMove = true;
                    }
                }
            }
        }
        //System.out.println("Exist cap move: " + capMove);
        return capMove;
    }

   //Updates the underlying board representation
   public void updateBoardRep(int x, int y, int newx, int newy) {
```

```
        int oldX = x / TileDim;
        int oldY = y/ TileDim;
        int newX = newx / TileDim;
        int newY = newy / TileDim;

        capture(boardRep, x, y, newx, newy);

        if (!regicide) {
        boardRep[newY][newX] = boardRep[oldY][oldX];
        }
        boardRep[oldY][oldX] = 'o';

        updateGUI();
        repaint();
    }

   //Updates the graphical user interface based on the board rep
   public void updateGUI() {
       checkerPoss.clear();
       repaint();

       for (int y = 1; y <= 8; y++) {
            for (int x = 1; x <= 8; x++) {

                if (boardRep[y-1][x-1] == 'w') {
                   this.add(new Checker(CheckerType.WHITE_REGULAR), y, x);
                }
                if (boardRep[y-1][x-1] == 'W') {
                   this.add(new Checker(CheckerType.WHITE_KING), y, x);
                }
                if (boardRep[y-1][x-1] == 'B') {
                   this.add(new Checker(CheckerType.BLACK_KING), y, x);
                }
                if (boardRep[y-1][x-1] == 'b') {
                   this.add(new Checker(CheckerType.BLACK_REGULAR), y, x);
                }

            }
        }
       repaint();
   }

   //NOT USED - Initial attemp at minimax. Worked to some extent but did not exhibit
intelligent gameplay
   public int minimaxAttempt1(char[][] board, int depth, String team, int alpha, int
beta) {
        int   maxScore   =   Integer.MIN_VALUE;//team   ==   otherPlayer(team)   ?
Integer.MIN_VALUE : Integer.MAX_VALUE;
        char[][] newBoard = new char[board.length][board.length];
        int oX, oY, nX, nY, newScore;

        //A deep copy of the board must be made to asses the score of it after
possible moves have been applied
        for (int i = 0; i< board.length; i++) {
            newBoard[i] = Arrays.copyOf(board[i], board[i].length);
        }

        //Array list of all the valid moves for a given team and board representation
        getAllValidMoves(team, board);

        //The score for the board at the maximum depth is returned in the deepest
layer of recursion
        if (depth == maxDepth) {
            return getHeuristic(board, otherPlayer(team));
        } else {
        //Otherwise all valid moves are applied to ultimately have their potential
score assessed
```

```
        for (int i = 0; i < allValidMoves.size(); i++) {
            oX = allValidMoves.get(i).startX;
            oY = allValidMoves.get(i).startY;
            nX = allValidMoves.get(i).endX;
            nY = allValidMoves.get(i).endY;

            newBoard = applyMove(newBoard, oX, oY, nX, nY);
            newScore = minimax(newBoard, depth+1, team, alpha, beta);
            //If the move results in a new high-scoring board, the details of this
move are recorded
            if (newScore > maxScore){
                maxScore = newScore;
                bestOldX = oX; bestOldY = oY; bestNewX = nX; bestNewY = nY;
            }

            //AB pruning
            if (maxScore > beta) {
                if (maxScore >= alpha){
                    break;
                } else {
                    beta = maxScore;
                }
            }
            if (maxScore < alpha) {
                if (maxScore <= beta) {
                    break;
                } else {
                    alpha = maxScore;
                }
            }
        }
        }

        return maxScore;
    }

   //Re implementation of minimax. Works significantly better.
   public int minimax(char[][] board, int depth, String team, int alpha, int beta){
       char[][] minimaxBoard = new char[board.length][board.length];
       List<Move> possibleMoves;
       int maxScore, eval;
       maxScore = Integer.MIN_VALUE;

       //Create deep copy of board to evaluate potential moves on
        for (int i = 0; i< board.length; i++) {
            minimaxBoard[i] = Arrays.copyOf(board[i], board[i].length);
        }

       //Populate list of possible moves
       possibleMoves = getAllValidMoves(team, board);

       //Terminating conditions
       if (depth == maxDepth || possibleMoves.isEmpty()){
           return getHeuristic(board, team); //Other player?
       }

       if (team == "white") {
           maxScore = Integer.MIN_VALUE;
           for (int m = 0; m < possibleMoves.size(); m++){
               int oX = possibleMoves.get(m).startX;
               int oY = possibleMoves.get(m).startY;
               int nX = possibleMoves.get(m).endX;
               int nY = possibleMoves.get(m).endY;

               //Applies first possible move, updating local boardRep
               minimaxBoard = applyMove(minimaxBoard, oX, oY, nX, nY);
```

```
                //Recursive minimax call, switching player each time
                eval = minimax(minimaxBoard, depth + 1, otherPlayer(team), alpha,
beta);

                //Log best move found
                if (eval > maxScore) {
                 bestOldX = oX; bestOldY = oY; bestNewX = nX; bestNewY = nY;
                 maxScore = eval;
                }

                //Alpha Beta pruning
                if (maxScore > alpha) {
                    alpha = maxScore;
                }
                if (alpha >= beta) {
                    break;
                }
            }
            return maxScore;
        }
        else {
            maxScore = Integer.MAX_VALUE;
            for (int j=0; j<possibleMoves.size(); j++){
                int oX = possibleMoves.get(j).startX;
                int oY = possibleMoves.get(j).startY;
                int nX = possibleMoves.get(j).endX;
                int nY = possibleMoves.get(j).endY;

                //Applies first possible move, updating local boardRep
                minimaxBoard = applyMove(minimaxBoard, oX, oY, nX, nY);

                //Recursive minimax call, switching player each time
                eval = minimax(minimaxBoard, depth + 1, otherPlayer(team), alpha,
beta);
                maxScore = min(maxScore, eval);

                //Alpha Beta pruning
                if (maxScore < beta) {
                    beta = maxScore;
                }
                if (alpha >= beta) {
                    break;
                }
            }
            return maxScore;
        }
    }

    //Moves a piece within the board representation
    public char[][] applyMove(char[][] board, int oX, int oY, int nX, int nY) {

         //System.out.println(board[oY][oX] + " at " + oX + "," + oY + " Moving to "
+ board[nY][nX] + " at " + nX + "," + nY);
         board[nY][nX] = board[oY][oX];
         board[oY][oX] = 'o';


         return board;
     }

    //This function returns the score of a given board state for the given team
    public int getHeuristic(char[][] board, String maxPlayer) {
         int score = 0;
         checkForKing(board);

         for (int y = 0; y < 8; y++) {
             for (int x = 0; x < 8; x++) {
```

```
                if (board[y][x] == 'w' && maxPlayer == "white") {
                    score++;
                } else if (board[y][x] == 'w' && maxPlayer == "black") {
                    score--;
                }
                if(board[y][x] == 'W' && maxPlayer == "white") {
                    score = score + 3;
                } else if (board[y][x] == 'W' && maxPlayer == "black") {
                    score = score - 3;
                }
                if(board[y][x] == 'b' && maxPlayer == "white") {
                    score--;
                } else if (board[y][x] == 'b' && maxPlayer == "black") {
                    score++;
                }
                if(board[y][x] == 'B' && maxPlayer == "white") {
                    score = score - 3;
                } else if (board[y][x] == 'B' && maxPlayer == "black") {
                    score = score + 3;
                }
            }

        }

        return score;
    }

   //Terminating Conditions
   public boolean blackHasWon() {
        boolean won = false;
        if (getAllValidMoves("white", boardRep).isEmpty()) {
            won = true;
        }
        return won;
    }
    public boolean whiteHasWon() {
        boolean won = false;
        if (getAllValidMoves("black", boardRep).isEmpty()) {
            won = true;
        }
        return won;
    }

    //Called by minimax to alternate between MIN and MAXimising player
    public String otherPlayer(String player) {
        String newPlayer;
        if (player == "white") {
            newPlayer = "black";
        } else {
            newPlayer = "white";
        }
        return newPlayer;
    }

    //Does exactly what it says on the tin
    public void resetBoard() {
        for (int y = 1; y <= 8; y++) {
            for (int x = 1; x <= 8; x++) {
                if ((x + y)%2 == 1 && x <= 3) { //Only place white checkers on black
tiles on the top 3 rows
                    this.add(new Checker(CheckerType.WHITE_REGULAR), x, y);
                }
                if ((x + y)%2 == 1 && x >= 6) { //Only place black checkers on black
tiles on the bottom 3 rows
                    this.add(new Checker(CheckerType.BLACK_REGULAR), x, y);
                }
            }
```

```
        }
    }

    //Adds a new checker, used for setting up the board.
    public void add(Checker checker, int y, int x) {
        if (x < 1 || x > 8 || y < 1 || y > 8)
            throw new IllegalArgumentException("Co-ords out of range!");
        CheckerPos checkerPos = new CheckerPos();
        checkerPos.checker = checker;
        checkerPos.x = (x - 1) * TileDim + TileDim /2;
        checkerPos.y = (y - 1) * TileDim + TileDim /2;
        for (CheckerPos _checkerPos: checkerPoss) {
            _checkerPos.Lcap = false;
            _checkerPos.Rcap = false;
            if (checkerPos.x == _checkerPos.x && checkerPos.y == _checkerPos.y)
                throw new AlreadyOccupiedException("(" + y + "," + x + ") is already
taken!");
        }
        checkerPoss.add(checkerPos);
    }

    @Override
    protected void paintComponent(Graphics g)
   {
      paintBoard(g);
      for (CheckerPos checkerPos: checkerPoss)
         if (checkerPos != Board.this.checkerPos)
            checkerPos.checker.draw(g, checkerPos.x, checkerPos.y);

      if (checkerPos != null)
         checkerPos.checker.draw(g, checkerPos.x, checkerPos.y);
   }

    //Draws the checker board
    private void paintBoard(Graphics g){
      ((Graphics2D)            g).setRenderingHint(RenderingHints.KEY_ANTIALIASING,
RenderingHints.VALUE_ANTIALIAS_ON);
      for (int row = 0; row < 8; row++){
         g.setColor(((row & 1) != 0) ? Color.BLACK : Color.decode("#FF3333"));
         for (int col = 0; col < 8; col++){
            g.fillRect(col * TileDim, row * TileDim, TileDim, TileDim);
            g.setColor((g.getColor()  ==  Color.BLACK)  ?  Color.decode("#FF3333")  :
Color.BLACK);
         }
      }
      //If hints are enabled, all tiles containing a movable checker are filled green
      if(hintsDisplayed){
            if (allValidMoves.size() > 0){
                g.setColor(Color.GREEN);
                for (int m = 0; m < allValidMoves.size(); m++){
                    g.fillRect(allValidMoves.get(m).startX        *        TileDim,
allValidMoves.get(m).startY * TileDim, TileDim, TileDim);
                }
            }
        }
   }

    @Override
   public Dimension getPreferredSize()
   {
      return dimSize;
   }

   //NOT USED - alternative method for displaying hints.
   public void drawHintCircle(Graphics g, int x, int y, int r) {
       x = x -(r/2);
       y = y- (r/2);
```

```
        g.setColor(Color.CYAN);
        g.fillOval(x,y,r,r);
    }

    public void toggleHints(){
        System.out.println("Hints toggled");
        this.hintsDisplayed = !hintsDisplayed;
        updateGUI();
        repaint();
    }

    public char[][] getBoard() {
        return boardRep;
    }
        }
```