Intelligent Systems Project

AI Checkers Game

Warren Boult,

Candidate number 186959

Introduction

The goal of this project was to build a checkers game in Java. This included building the GUI to visually represent the game and allow interactive play; the internal workings of the game, to update the game state and verify if a player has won; and an AI controller, which can generate potential moves and evaluate which would be the best, to allow a human to play against this controller.

My approach to this task was broadly to develop an event-driven GUI, wherein the event listeners on the various GUI components would handle the updating of the game state, and updating the GUI. This appeared to me the simplest way to build the game, preventing the possibility of complicated threading issues, and avoiding undesirable things such as a while loop updating the GUI (this can cause unexpected Java component updating). I firstly tackled the building of the GUI, as I felt less comfortable in coding this aspect of the project, and also once I had built the GUI, the empty game state functions I’d specified in the event listeners effectively meant I knew what functions and classes I’d need for my game state.

I essentially split my classes into all the possible discrete objects that exist in a checkers game: the board, the pieces, the moves, and the game state. Along with this, there needed to be a GUI class, and some small helper classes. I decided I didn’t need classes for the player/AI, since the Game class handled everything on this front effectively without confusing the matter.

My main class is the CheckersServer class, which handles all the workings of the game, including the creation and updating of the GUI, and the updating of game state, switching turns, starting/quitting the game etc.

Along with this main class, there is a Board class, which extends JComponent, and this class holds GUI-related information and functions for the game board, including the overrided paintComponent method to allow me to paint the board with the checkers in their correct places.

Similarly, there is a Checker class, which holds information about an individual checker such as its type, colour, board position, and pixel coordinates of its associated square centre, and has a paint method to paint an oval of the correct colour, with a “K” label if it is a King piece.

Then, there is a Move class, which is used when generating valid moves; this holds the type of move (“Move” or “Capture”), and the list of position updates (‘BoardMoves’) associated with the move.

The game internals are stored in the Game class. The game class handles everything to do with updating the game state; this includes a method to initialise the game state, methods to generate valid successors, check if the game is won, do AI moves, minimax evaluation, among others. To make the AI have more unpredictable play, more similar to a human, I make the AI choose one of the available moves randomly if multiple moves have the same score, as otherwise it will always choose the first move in the list leading to the same patterns of play in certain situations.

The Game class initialises a game in startGame by setting an 8x8 array of Checkers, giving each Checker its correct colour, type and position. It updates the game state in updateState when a new valid move is proposed, by removing the moved Checker from its old position in the array, placing it in its new position, and removing any captured pieces, then checking if the game is won.

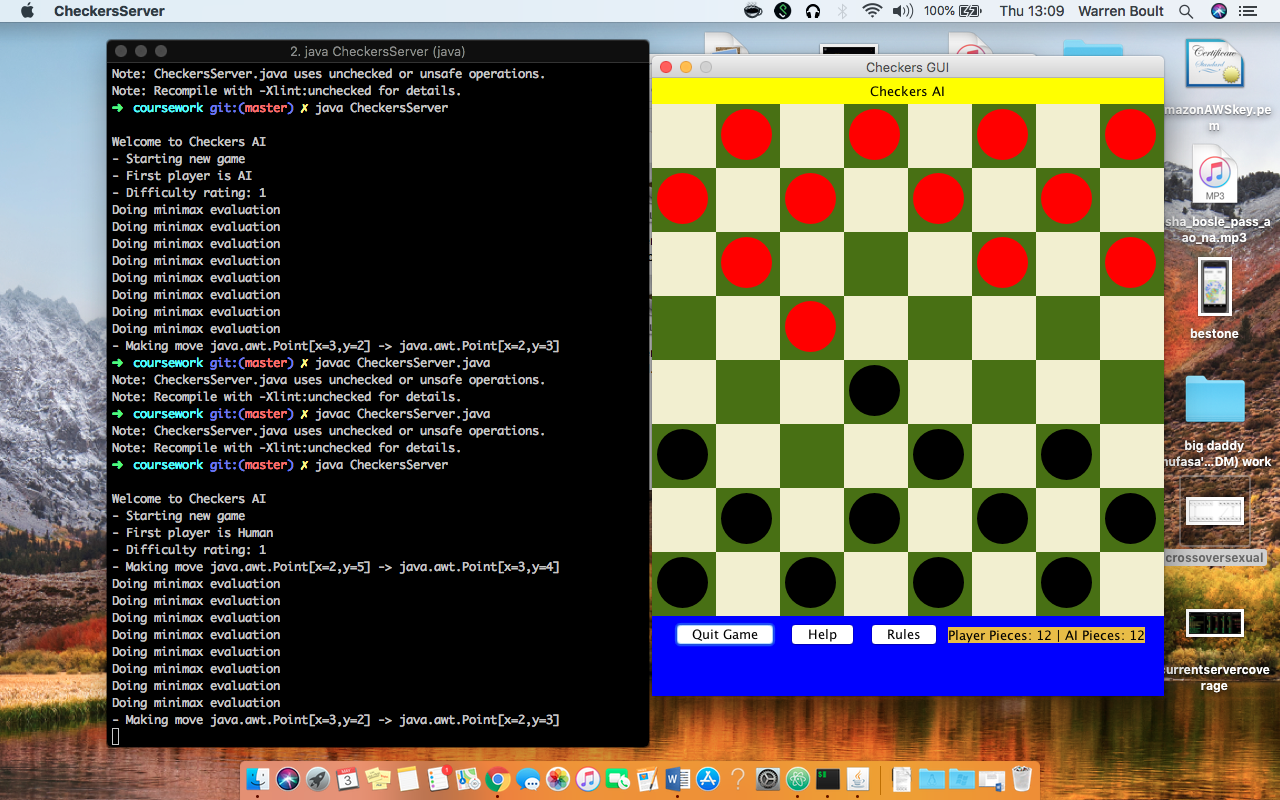


Figure - screenshot of the game in play

To run the program, type javac CheckersServer.java to compile, and then java CheckersServer. Alternatively double click the CheckersServer.class file, though this means you won’t get the associated terminal outputs.

Program Functionality

My program includes interactive gameplay of a human user vs the AI, wherein an event listener on the board allows the human player on their turn to drag a checker to a new position, and the GUI updates reflecting the validity of this move. This event listener functionality takes place between the lines 359-580. There are event listeners on the board to allow a checker to be clicked, dragged around, and released on valid positions. The GUI updating has a few issues, as sometimes the bottom menu panel items don’t refresh and work correctly, and additionally when selecting AI to move first, the board with checkers is displayed with the AI’s first move already made; I wasn’t able to get these issues sorted unfortunately, but they don’t have too big an effect on gameplay.

The program starts by running the main method in the CheckersServer class (line 697), which constructs a CheckersServer object, calling the homeGUI function to display the home screen and initialise a Game object.

The game state representation is held in the checkers array, declared at the start of the Game class, on line 834. The checkers array is initialised in startGame from line 1629, and updated after moves by updateState() from line 1679. The valid successor states are generated by a collection of functions, generateAllValidSuccessors (line 1268), generateValidSuccessors (line 1216), addValidRegMoves (line 1070), addValidCaptureMoves (line 937), and addMultiCapMoves (line 1040). The generateAllValidSuccessors function gives a list of all the valid moves for whoever’s turn it is, and so this is used to verify if a user’s proposed move is valid (line 455), and when highlighting movable squares for the user (line 539).

Similarly, generateAllValidSuccessors is used to generate AI moves for the AI minimax evaluation.

When a move is invalid, popups are displayed giving a general reason why that move is invalid (lines 427, 444, 480, 491). These are not that specific, rather they let the user know if they have tried to move outside of the board, if they are trying to move to occupied or white squares, and if the piece attempted to move has no valid moves available.

The AI controller uses valid minimax evaluation (line 1434) to find the best move, and searches to a game tree depth specified by the given difficulty rating (line 1464). The minimax generates valid successors, returns a heuristic evaluation if the max depth is reached, performs static evaluations, and does recursive dynamic evaluations. To perform the dynamic evaluation, a temporary copy of the game’s state is needed, so that it can be updated each recursive call of the minimax without actually affecting the real game state. This required adding a copy constructor to the Game class (line 857). Furthermore, a helper class was needed for building the list of potential moves and their associated scores: MoveAndScore. The AI controller loops through this list in makeAIMove (line 1388) when finally selecting the move it will play.

The minimax evaluation uses alpha-beta pruning to speed up AI move selection (line 1527). If the alpha score is greater than the beta score, we stop searching down that branch of the tree by doing a ‘break;’, as this means that the best score found down that branch won’t be better than the one we have already found.

The game also supports multi-capture moves for both the AI and player. The multi-capture generation is handled in the addValidCaptureMoves method (line 937), wherein this method is recursively called while there are valid captures available from the end position of the previous capture.

Forcing of takes is also incorporated into the game; this is done by only adding regular moves to the available move list for a given board position if there are no captures available (lines 1231 and 1250), and then only adding capture moves to the list of all valid successors if a capture move has been found (lines 1280-1298).

Finally for the game internals, there is automatic king conversion in updateState (lines 1690-1698). This checks if the piece is at the opposite end to the current player’s starting end, and if so, converts to a King, updating the Checker’s type.

On the GUI side of things, the game supports drag and drop of pieces (line 562), updating the chosen piece to the dropped location if the move is valid, repainting it at its old location if not. It also has a help option (line 303), which when pressed will allow the player to click on its pieces and see the associated movable positions for that piece, highlighting the movable squares (lines 537-543).

The GUI has a menu option to take you to the webpage displaying the rules (lines 150-174). However I wasn’t able to get the GUI to pause for a very long time between steps of a multi-capture move, despite my attempts by making the thread sleep between moves of the multi-capture (lines 655-662).

Appendix

0001: /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

0002: \* AI Checkers. Play against an AI opponent of varying difficulty \*

0003: \* Warren Boult, University of Sussex, 2018. \*

0004: \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

0005:

0006: import java.util.\*;

0007: import javax.swing.\*;

0008: import java.awt.\*;

0009: import java.awt.event.\*;

0010: import java.util.concurrent.TimeUnit;

0011: import java.util.Random;

0012:

0013: import java.awt.Desktop;

0014: import java.net.URI;

0015:

0016: // Main class to build the gui, and handle the playing of the game.

0017: public class CheckersServer extends JFrame {

0018:

0019: JPanel pane, panel1, panel2, panel3, scorePanel;

0020: JLabel label1, scoreLabel;

0021: JButton btnNewGame, btnExit, btnRules, btnQuitGame;

0022: JToggleButton btnHelp;

0023: Board boardPanel;

0024: JComboBox diffOptions;

0025: static Game game;

0026: private static int BOARDSIZE = 512;

0027: private static int WINDOWHEIGHT = (int) ( BOARDSIZE / 0.8 );

0028: private Point oldCenter;

0029: private Point oldPos;

0030: private boolean dragFlag = false;

0031: private Checker checkerAtPos, checkerAtNewPos;

0032: private ArrayList<Move> validMoves = new ArrayList<Move>();

0033:

0034: public CheckersServer()

0035: {

0036:

0037: System.out.println("Welcome to Checkers AI");

0038: game = new Game( BOARDSIZE / 8 );

0039: homeGUI( game );

0040: // game = gameState;

0041:

0042: }

0043:

0044: // Display the initial gui, for not started game.

0045: // Board not yet populated, and cannot be interacted with

0046: public void homeGUI( Game game )

0047: {

0048: //Create and set up the window.

0049: // JFrame frame = new JFrame( "CheckersServer" );

0050: setTitle( "Checkers GUI" );

0051: setDefaultCloseOperation( JFrame.EXIT\_ON\_CLOSE );

0052: setLayout(new BorderLayout());

0053:

0054: pane = new JPanel();

0055: // pane.removeAll();

0056: // GridLayout layout = new GridLayout(1, 2);

0057: // setLayout(layout);

0058:

0059: // Using gridbag layout for menu items, so GridBagConstraints must be set

0060: pane.setLayout(new GridBagLayout());

0061: GridBagConstraints c = new GridBagConstraints();

0062:

0063: panel1 = new JPanel();

0064: // panel1.setLayout(new BorderLayout());

0065: panel1.setBackground( Color.YELLOW );

0066: // panel1.setSize( 800, 800 );

0067: label1 = new JLabel( "Checkers AI" );

0068: // label1.setBackground(Color.CYAN );

0069: // label1.setSize( 800, 800 );

0070: panel1.add(label1);

0071:

0072: c.fill = GridBagConstraints.BOTH;

0073: c.gridx = 0;

0074: c.gridy = 0;

0075: c.weightx = 1.0;

0076: pane.add(panel1, c);

0077:

0078: panel2 = new JPanel();

0079: panel2.setLayout(new FlowLayout());

0080: boardPanel = new Board(game);

0081: boardPanel.setPreferredSize(new Dimension(BOARDSIZE, BOARDSIZE));

0082: // panel2.setBorder(BorderFactory.createEmptyBorder(10,10,10,10));

0083: // panel2.setBackground(Color.CYAN );

0084:

0085: c.fill = GridBagConstraints.BOTH;

0086: c.gridx = 0;

0087: c.gridy = 1;

0088: c.weightx = 1.0;

0089: // c.weighty = 0.8;

0090: pane.add(boardPanel, c);

0091: // pane.add(panel2, c);

0092: // pane.add(panel2, c);

0093:

0094: panel3 = new JPanel();

0095: panel3.removeAll();

0096: panel3.setBackground( Color.BLUE );

0097: // panel2.setSize( 800, 200 );

0098:

0099: btnNewGame = new JButton( "New Game" );

0100: btnNewGame.addActionListener(new ActionListener()

0101: {

0102:

0103: @Override

0104: public void actionPerformed(ActionEvent e)

0105: {

0106: String[] options = { "Human", "AI" };

0107: int n = JOptionPane.showOptionDialog(null,

0108: "Who gets first move?",

0109: "Choose Player 1",

0110: JOptionPane.YES\_NO\_OPTION,

0111: JOptionPane.INFORMATION\_MESSAGE,

0112: null,

0113: options,

0114: options[0]);

0115: String player1 = n == 0 ? "Human" : "AI";

0116: System.out.println( "- Starting new game" );

0117: game.setPlayer( player1 );

0118: System.out.println( "- First player is " + game.whoseTurn() );

0119: System.out.println( "- Difficulty rating: " + game.getDifficulty() );

0120: // setVisible(false);

0121: newGame( game );

0122:

0123: // javax.swing.SwingUtilities.invokeLater(new Runnable() {

0124: // public void run() {

0125:

0126: revalidate();

0127: repaint();

0128: // setVisible(true);

0129:

0130: // }

0131: // });

0132:

0133: }

0134:

0135: });

0136: btnExit = new JButton( "Exit" );

0137: btnExit.addActionListener(new ActionListener()

0138: {

0139:

0140: @Override

0141: public void actionPerformed(ActionEvent e)

0142: {

0143: System.out.println( "- Exiting game" );

0144: setVisible( false );

0145: dispose();

0146: System.exit(0);

0147: }

0148:

0149: });

0150: btnRules = new JButton( "Rules" );

0151: btnRules.addActionListener(new ActionListener()

0152: {

0153:

0154: @Override

0155: public void actionPerformed( ActionEvent e )

0156: {

0157:

0158: if (Desktop.isDesktopSupported())

0159: {

0160:

0161: try

0162: {

0163: Desktop.getDesktop().browse( new URI("http://www.indepthinfo.com/checkers/play.shtml" ));

0164: }

0165: catch ( Exception err )

0166: {

0167: System.err.println("Error getting rules: " + err.getMessage());

0168: }

0169:

0170: }

0171:

0172: }

0173:

0174: });

0175:

0176: String[] diffStrings = { "Difficulty (Default is easy):", "Easy", "Intermediate", "Hard", "Extreme" };

0177: diffOptions = new JComboBox( diffStrings );

0178: diffOptions.addItemListener(new ItemListener()

0179: {

0180:

0181: @Override

0182: public void itemStateChanged( ItemEvent e )

0183: {

0184:

0185: if ( e.getStateChange() == ItemEvent.SELECTED )

0186: {

0187:

0188: String item = (String) e.getItem();

0189: game.setDifficulty( item );

0190:

0191: }

0192:

0193: }

0194:

0195: });

0196:

0197: c.fill = GridBagConstraints.BOTH;

0198: c.anchor = GridBagConstraints.CENTER;

0199: c.gridx = 0;

0200: c.gridy = 0;

0201: c.weightx = c.weighty = 1.0;

0202: panel3.add( btnNewGame, c );

0203: c.gridx = 1;

0204: c.gridy = 0;

0205: // c.insets = new Insets(50, 50, 50, 50);

0206: panel3.add( diffOptions, c );

0207: c.gridx = 2;

0208: // c.insets = new Insets(50, 50, 50, 50);

0209: // c.gridwidth = 1;

0210: // c.weighty = 1.0;

0211: panel3.add( btnExit, c );

0212: c.gridx = 3;

0213: c.gridy = 0;

0214: panel3.add( btnRules, c );

0215: c.gridx = 0;

0216: c.gridy = 2;

0217: // c.weighty = 0.1;

0218: c.insets = new Insets(0, 0, 0, 0);

0219: pane.add( panel3, c );

0220: pane.revalidate();

0221: pane.repaint();

0222:

0223: // pane.add(button, c);

0224: add(pane);

0225: // c.add(panel2);

0226:

0227: setResizable(false);

0228: pack();

0229: setSize( BOARDSIZE, WINDOWHEIGHT );

0230: setLocationRelativeTo( null );

0231: setVisible(true);

0232: repaint();

0233: validate();

0234:

0235: // setVisible(true);

0236:

0237: }

0238:

0239: // Display the in-game gui,

0240: // add event listeners to the board to drive event driven gameplay

0241: public void newGame( Game game )

0242: {

0243:

0244: game.startGame();

0245: // setVisible(false);

0246: // revalidate();

0247: // repaint();

0248:

0249:

0250: // Remove home screen buttons

0251: panel3.remove( btnNewGame );

0252: panel3.remove( diffOptions );

0253: panel3.remove( btnExit );

0254: panel3.remove( btnRules );

0255:

0256: // boardPanel.repaint();

0257: // boardPanel.revalidate();

0258:

0259: btnQuitGame = new JButton( "Quit Game" );

0260: btnQuitGame.addActionListener(new ActionListener()

0261: {

0262:

0263: @Override

0264: public void actionPerformed(ActionEvent e)

0265: {

0266:

0267: int n = JOptionPane.showConfirmDialog(null,

0268: "Are you sure you want to quit?",

0269: "Quitting Game",

0270: JOptionPane.YES\_NO\_OPTION);

0271: if ( n == 0 )

0272: {

0273:

0274: game.endGame();

0275: // panel3.removeAll();

0276: // boardPanel.clearHighlightSquares();

0277: // dispose();

0278: // setVisible(false);

0279: // // panel3.updateUI();

0280: // panel3.revalidate();

0281: // panel3.repaint();

0282: // javax.swing.SwingUtilities.invokeLater(new Runnable() {

0283: // public void run() {

0284: //

0285: // homeGUI( game );

0286: // revalidate();

0287: // repaint();

0288: // setVisible(true);

0289: //

0290: // }

0291: // });

0292: setVisible( false );

0293: dispose();

0294: System.exit(0);

0295:

0296:

0297: }

0298:

0299: }

0300:

0301: });

0302:

0303: btnHelp = new JToggleButton("Help");

0304: btnHelp.addActionListener(new ActionListener()

0305: {

0306:

0307: @Override

0308: public void actionPerformed( ActionEvent e )

0309: {

0310:

0311: JToggleButton tglBtn = (JToggleButton) e.getSource();

0312:

0313: if ( tglBtn.isSelected() )

0314: game.setHelp(true);

0315: else

0316: game.setHelp(false);

0317: boardPanel.clearHighlightSquares();

0318: revalidate();

0319: repaint();

0320:

0321: }

0322:

0323: });

0324:

0325: String labelString = "Player Pieces: " + (12 - game.aiScore()) + " | AI Pieces: " + (12 - game.playerScore());

0326: scoreLabel = new JLabel(labelString);

0327: scoreLabel.setOpaque(true);

0328: scorePanel = new JPanel();

0329: scoreLabel.setBackground( new Color( 232, 189, 71 ) );

0330: // scorePanel.add(scoreLabel);

0331: // scorePanel.revalidate();

0332: // TODO: Add a score panel, maybe other info

0333: GridBagConstraints c = new GridBagConstraints();

0334: c.fill = GridBagConstraints.BOTH;

0335: c.anchor = GridBagConstraints.CENTER;

0336: c.gridx = 0;

0337: c.gridy = 0;

0338: c.weightx = c.weighty = 1.0;

0339: panel3.add(btnQuitGame, c);

0340: c.gridx = 1;

0341: c.gridy = 0;

0342: panel3.add(btnHelp, c);

0343: c.gridx = 2;

0344: c.gridy = 0;

0345: panel3.add(btnRules, c);

0346: c.gridx = 3;

0347: // c.gridwidth = 3;

0348: panel3.add(scoreLabel, c);

0349: c.gridx = 0;

0350: c.gridy = 2;

0351: c.gridwidth = 1;

0352: pane.add(panel3, c);

0353: panel3.revalidate();

0354: boardPanel.revalidate();

0355: pane.revalidate();

0356: pane.repaint();

0357:

0358: // Add listener functionality to the board

0359: boardPanel.addMouseListener(new MouseAdapter()

0360: {

0361:

0362: public void mouseEntered( MouseEvent e ) {}

0363: public void mouseExited( MouseEvent e ) {}

0364: public void mouseMoved( MouseEvent e ) {}

0365:

0366: // When mouse is pressed, find the board position pressed on,

0367: // and generate valid moves.

0368: // Store the board position to repaint checker there if proposed move is invalid

0369: @Override

0370: public void mousePressed( MouseEvent e )

0371: {

0372:

0373: if ( !game.inPlay() )

0374: return;

0375:

0376: if ( game.whoseTurn() == "Human" )

0377: {

0378:

0379: game.clearSuccessors( validMoves );

0380:

0381: // // Obtain mouse coordinates at time of press.

0382: //

0383: int x = e.getX();

0384: int y = e.getY();

0385: Point boardPos = boardPanel.convertToBoardPosition(x,y);

0386: checkerAtPos = game.checkers[boardPos.x][boardPos.y];

0387:

0388: if ( checkerAtPos != null && game.getPlayerColour() == checkerAtPos.getColour() )

0389: {

0390:

0391: // System.out.println("Clicked on your piece");

0392: oldCenter = checkerAtPos.getCenter();

0393: oldPos = checkerAtPos.getPos();

0394: game.clearSuccessors( validMoves );

0395: validMoves = game.generateAllValidSuccessors();

0396: dragFlag = true;

0397:

0398: }

0399:

0400: }

0401:

0402: }

0403:

0404: // When dragging of piece is finished, we check to see if this proposed move is valid

0405: // If invalid, redraw checker at original pos, display reason why invalid.

0406: // If move is valid, update state and gui to reflect move

0407: @Override

0408: public void mouseReleased( MouseEvent e )

0409: {

0410:

0411: if ( !game.inPlay() )

0412: return;

0413:

0414: if ( game.whoseTurn() == "Human" )

0415: {

0416: if ( dragFlag )

0417: dragFlag = false;

0418: else

0419: return;

0420:

0421: int x = e.getX();

0422: int y = e.getY();

0423:

0424: if ( x > 512 || y > 512 )

0425: {

0426:

0427: String messageString = "Attempting to move outside of board, try again";

0428: errorPopup( messageString );

0429: checkerAtPos.setBoardPos( oldPos.x, oldPos.y );

0430: return;

0431:

0432: }

0433:

0434: // TODO: add popup explaining why a move is invalid

0435: Point boardPos = boardPanel.convertToBoardPosition(x,y);

0436: checkerAtNewPos = game.checkers[boardPos.x][boardPos.y];

0437:

0438: if ( checkerAtNewPos == null && (boardPos.x + boardPos.y) % 2 != 0 ) // for white squares x + y is even

0439: {

0440:

0441: if ( validMoves.isEmpty() )

0442: {

0443:

0444: String messageString = "No valid moves available for that piece, try again";

0445: errorPopup( messageString );

0446: checkerAtPos.setBoardPos( oldPos.x, oldPos.y );

0447:

0448: }

0449:

0450: BoardMove newMove = new BoardMove( oldPos, boardPos );

0451:

0452: for ( Move move : validMoves )

0453: {

0454:

0455: if ( move.getMovePos().equals( newMove.newPos ) && move.getStartPos().equals( newMove.oldPos ) ) // If proposed end position matches one of the valid moves, use that valid move

0456: {

0457:

0458: // Do human move, then switch play to do AI move

0459: updateGUI( game, move );

0460:

0461:

0462: // Creating a Runnable instance here:

0463: javax.swing.SwingUtilities.invokeLater(new Runnable() {

0464: public void run() {

0465:

0466: game.setPlayer( "AI" );

0467: Move bestMove = game.makeAIMove();

0468:

0469: updateGUI( game, bestMove );

0470: game.setPlayer( "Human" );

0471:

0472: }

0473: });

0474:

0475: return;

0476: }

0477:

0478: }

0479:

0480: String messageString = "Proposed move not valid, try again";

0481: errorPopup( messageString );

0482: checkerAtPos.setBoardPos( oldPos.x, oldPos.y );

0483: // break;

0484:

0485: }

0486: else if ( boardPos.x == oldPos.x && boardPos.y == oldPos.y )

0487: {}

0488: else

0489: {

0490:

0491: String messageString = "Attempting to move to white or non-empty square, try again";

0492: errorPopup( messageString );

0493: checkerAtPos.setBoardPos( oldPos.x, oldPos.y );

0494:

0495: }

0496:

0497: revalidate();

0498: repaint();

0499:

0500: }

0501:

0502: }

0503:

0504: // Upon clicking a piece, if help is enabled squares that can be moved to

0505: // should get highlighted

0506: @Override

0507: public void mouseClicked( MouseEvent e )

0508: {

0509:

0510: if ( !game.inPlay() )

0511: return;

0512:

0513: if ( game.whoseTurn() == "Human" )

0514: {

0515:

0516: boardPanel.clearHighlightSquares();

0517: game.clearSuccessors( validMoves );

0518:

0519: if ( game.getHelp() )

0520: {

0521:

0522: int x = e.getX();

0523: int y = e.getY();

0524: // System.out.println( "X coord: " + x + "; Y coord: " + y );

0525: Point boardPos = boardPanel.convertToBoardPosition(x,y);

0526:

0527: checkerAtPos = game.checkers[boardPos.x][boardPos.y];

0528: if ( checkerAtPos != null )

0529: {

0530:

0531: if ( game.getPlayerColour() == checkerAtPos.getColour() )

0532: {

0533:

0534: // System.out.println("Clicked on your piece");

0535: validMoves = game.generateAllValidSuccessors();

0536: ArrayList<Point> highlightSquares = new ArrayList<Point>();

0537: for ( Move move: validMoves )

0538: {

0539: if ( move.getStartPos().equals(boardPos) )

0540: highlightSquares.add( move.getMovePos() );

0541: }

0542:

0543: boardPanel.setHighlightSquares( highlightSquares );

0544:

0545: }

0546:

0547: }

0548:

0549: }

0550:

0551: revalidate();

0552: repaint();

0553: }

0554:

0555: }

0556: });

0557:

0558: boardPanel.addMouseMotionListener(new MouseMotionAdapter()

0559: {

0560:

0561: @Override

0562: public void mouseDragged( MouseEvent e )

0563: {

0564:

0565: if ( !game.inPlay() )

0566: return;

0567:

0568: if ( game.whoseTurn() == "Human" && dragFlag )

0569: {

0570:

0571: checkerAtPos.setPos( e.getX(), e.getY() );

0572: repaint();

0573:

0574: }

0575:

0576: }

0577:

0578: });

0579:

0580: revalidate();

0581: repaint();

0582: // If first player is AI, start by letting the AI move

0583: javax.swing.SwingUtilities.invokeLater(new Runnable() {

0584: public void run() {

0585:

0586: if ( game.whoseTurn() == "AI" )

0587: {

0588:

0589:

0590: Move bestMove = game.makeAIMove();

0591: updateGUI( game, bestMove );

0592: game.setPlayer( "Human" );

0593:

0594: }

0595:

0596: }

0597: });

0598:

0599: // }

0600:

0601:

0602: // panel3.repaint();

0603: // panel3.revalidate();

0604: revalidate();

0605: repaint();

0606: // setVisible(true);

0607:

0608: }

0609:

0610: // Update the game state, and reflect the changes by repainting the gui.

0611: // Upon each update, if game is won, then display the winner popup and end game

0612: public void updateGUI( Game game, Move move )

0613: {

0614:

0615: // iterate through all board positions, 1 for a single capture, multiple for mutlicap

0616: for ( BoardMove boardMove : move.moveList )

0617: {

0618:

0619: System.out.println("- Making move " + boardMove.getMove());

0620: game.updateState( move.getMoveType(), boardMove );

0621: boardPanel.clearHighlightSquares();

0622: String labelString = "Player Pieces: " + (12 - game.aiScore()) + " | AI Pieces: " + (12 - game.playerScore());

0623: scoreLabel.setText(labelString);

0624:

0625: revalidate();

0626: repaint();

0627: // revalidate();

0628: // repaint();

0629:

0630: if ( game.isWon() )

0631: {

0632:

0633: winnerPopup();

0634: game.endGame();

0635: panel3.removeAll();

0636: boardPanel.clearHighlightSquares();

0637: dispose();

0638: setVisible( false );

0639: homeGUI( game );

0640: try

0641: {

0642: TimeUnit.SECONDS.sleep(1);

0643: }

0644: catch ( InterruptedException err )

0645: {

0646: System.err.println("Thread interrupted");

0647: }

0648: revalidate();

0649: repaint();

0650: setVisible( true );

0651: break;

0652:

0653: }

0654:

0655: try

0656: {

0657: TimeUnit.SECONDS.sleep(2);

0658: }

0659: catch ( InterruptedException err )

0660: {

0661: System.err.println("Thread interrupted");

0662: }

0663:

0664: }

0665:

0666: }

0667:

0668: // Display a popup if the game is won

0669: public void winnerPopup()

0670: {

0671:

0672: String messageString = game.humanHasWon() ? "Well done, you won!" : "Bad luck, you lost";

0673:

0674: // if ( game.humanHasWon() )

0675: // messageString = "Well done, you won!!";

0676: // else

0677: // messageString = "Unlucky, AI won";

0678:

0679: JOptionPane.showMessageDialog(null,

0680: messageString,

0681: "Game Won",

0682: JOptionPane.PLAIN\_MESSAGE);

0683:

0684: }

0685:

0686: // Display a popup if move is invalid

0687: public void errorPopup( String messageString )

0688: {

0689:

0690: JOptionPane.showMessageDialog(null,

0691: messageString,

0692: "Invalid Move",

0693: JOptionPane.PLAIN\_MESSAGE);

0694:

0695: }

0696:

0697: public static void main( String[] args )

0698: {

0699:

0700: new CheckersServer();

0701:

0702: }

0703: }

0704:

0705: // Board class is used to draw the board, and the checkers on the board.

0706: // Purely a gui-related aspect, does not control any game functionality

0707: class Board extends JComponent {

0708: private static int rows = 8;

0709: private static int columns = 8;

0710: private static Color green = new Color( 72, 112, 20 );

0711: private static Color blue = new Color( 112, 157, 229 );

0712: private static Color cream = new Color( 242, 239, 208 );

0713: // BoardLayout board\_layout;

0714: private Game gameState;

0715: private Point oldCenter;

0716: private Point oldPos;

0717: private boolean dragFlag = false;

0718: private Checker checkerAtPos, checkerAtNewPos;

0719: public ArrayList<Point> highlightSquares = new ArrayList<Point>();

0720:

0721: Board( Game game )

0722: {

0723:

0724: gameState = game;

0725:

0726: }

0727:

0728: public void setHighlightSquares( ArrayList<Point> squares )

0729: {

0730:

0731: for ( Point square: squares )

0732: {

0733:

0734: highlightSquares.add(square);

0735:

0736: }

0737:

0738: }

0739:

0740: public void clearHighlightSquares()

0741: {

0742:

0743: highlightSquares.clear();

0744:

0745: }

0746:

0747: public Point convertToBoardPosition( int xcoord, int ycoord )

0748: {

0749: int height = this.getSize().height/8;

0750: int width = this.getSize().width/8;

0751:

0752: int xboard = xcoord / width;

0753: int yboard = ycoord / width;

0754:

0755: return new Point( xboard, yboard );

0756: }

0757:

0758: @Override

0759: protected void paintComponent( Graphics g )

0760: {

0761:

0762: ((Graphics2D) g).setRenderingHint( RenderingHints.KEY\_ANTIALIASING, RenderingHints.VALUE\_ANTIALIAS\_ON );

0763:

0764: int height = this.getSize().height/8;

0765: int width = this.getSize().width/8;

0766:

0767: for (int i = 0; i < rows; i++)

0768: {

0769: for (int j = 0; j < columns; j++)

0770: {

0771: if (j % 2 == 0)

0772: {

0773:

0774: if (i % 2 == 0)

0775: g.setColor( cream );

0776: else

0777: {

0778: g.setColor( green );

0779: if ( highlightSquares.contains( new Point( j, i ) ) )

0780: g.setColor( blue );

0781: }

0782:

0783: }

0784: else {

0785:

0786: if (i % 2 == 0)

0787: {

0788: g.setColor( green );

0789: if ( highlightSquares.contains( new Point( j, i ) ) )

0790: g.setColor( blue );

0791: }

0792: else

0793: g.setColor( cream );

0794:

0795: }

0796: g.fillRect( j \* height, i \* height, height, height );

0797: // if game.inPlay() {

0798: // if game.board[i][j]

0799: // drawChecker

0800: // }

0801: }

0802: }

0803:

0804: if (gameState.inPlay())

0805: {

0806:

0807: for ( int i=0; i<gameState.checkers.length; i++ )

0808: {

0809: for ( int j=0; j<gameState.checkers[i].length; j++ )

0810: {

0811:

0812: Checker checker = gameState.checkers[i][j];

0813: if (checker != null)

0814: checker.paint( g );

0815:

0816: }

0817: }

0818: }

0819: // this.repaint();

0820: }

0821:

0822: }

0823:

0824: // The Game class holds all the internal game state information,

0825: // includes the functions to build initial state, advance game state,

0826: // generate successors, and check if game is won

0827: class Game {

0828:

0829: private boolean inPlayFlag = false;

0830: private String activePlayer = "Human";

0831: private int difficulty = 1;

0832: private int seLevel = 1;

0833: private boolean help = false;

0834: Checker[][] checkers;

0835: public static int SQUARESIZE;

0836: private String playerColour = "Black";

0837: private String aiColour = "Red";

0838: ArrayList<Point> highlightSquares = new ArrayList<Point>();

0839:

0840: ArrayList<MoveAndScore> aiMoveList = new ArrayList<MoveAndScore>();

0841: int maxDepth = 1;

0842: int seCount;

0843: int deCount;

0844: int pCount;

0845:

0846: // private String playerTurn;

0847:

0848: public Game( int squareSize )

0849: {

0850:

0851: SQUARESIZE = squareSize;

0852: // playerTurn = player;

0853:

0854: }

0855:

0856: // Copy constructor needed to create a temporary game state for AI

0857: public Game( Game original )

0858: {

0859:

0860: // Just copy over all fields from original gameState

0861: this.inPlayFlag = original.inPlay();

0862: this.activePlayer = original.whoseTurn();

0863: this.difficulty = original.getDifficulty();

0864: this.help = original.getHelp();

0865: this.checkers = new Checker[original.checkers[0].length][original.checkers[1].length];

0866: for ( int x = 0; x < checkers[0].length; x++)

0867: {

0868: for ( int y = 0; y < checkers[1].length; y++)

0869: {

0870: if ( original.checkers[x][y] != null )

0871: checkers[x][y] = new Checker( original.checkers[x][y] );

0872: }

0873: }

0874: this.SQUARESIZE = original.SQUARESIZE;

0875: this.playerColour = original.getPlayerColour();

0876: this.aiColour = original.getAIColour();

0877: if ( !original.highlightSquares.isEmpty() )

0878: {

0879: for ( Point square : original.highlightSquares )

0880: this.highlightSquares.add(square);

0881: }

0882: if ( !original.aiMoveList.isEmpty() )

0883: {

0884: for ( MoveAndScore mas : original.aiMoveList )

0885: this.aiMoveList.add( new MoveAndScore( mas ) );

0886: }

0887: this.maxDepth = original.maxDepth;

0888: this.seCount = original.seCount;

0889: this.deCount = original.deCount;

0890: this.pCount = original.pCount;

0891:

0892: }

0893:

0894: // If aiScore is maximal, ie 12, then ai has won the game

0895: public boolean aiHasWon()

0896: {

0897: // System.out.println("AI score: " + this.aiScore() );

0898: if ( this.aiScore() == 12 )

0899: {

0900:

0901: inPlayFlag = false;

0902: return true;

0903:

0904: }

0905: else

0906: return false;

0907:

0908: }

0909:

0910: // AI score is max number of pieces minus number of human player's pieces remaining

0911: // Used to calculate leaf utility and to see if game is won

0912: public int aiScore()

0913: {

0914:

0915: int score = 12;

0916: for ( int x = 0; x < 8; x++ )

0917: {

0918: for ( int y = 0; y < 8; y++ )

0919: {

0920:

0921: if ( checkers[x][y] != null && checkers[x][y].getColour() == this.getPlayerColour() )

0922: score--;

0923:

0924: }

0925: }

0926:

0927: return score;

0928:

0929: }

0930:

0931: // Returns the valid capture moves for a given position

0932: // if conditions check to see if the proposed move end point is within the board bounds,

0933: // if the space is empty, and if the piece to be captured is of the opposite colour

0934: // Recurse for multiCaptures,

0935: // make sure we're not trying to do a multicapture forwards and backwards

0936: // in the same spot with (newX != oldX || newY != oldY) in conditional

0937: private ArrayList<Move> addValidCaptureMoves( int x, int y, String colour, PieceType type, int oldX, int oldY )

0938: {

0939:

0940: ArrayList<Move> validMoves = new ArrayList<Move>();

0941:

0942: if ( this.whoseTurn() == "Human" )

0943: {

0944:

0945: if ( x-2 >= 0 && y-2 >= 0 && (x-2 != oldX || y-2 != oldY) && checkers[x-1][y-1] != null && checkers[x-1][y-1].getColour() != colour && checkers[x-2][y-2] == null ) // left capture move allowed

0946: {

0947:

0948: // Move move = new Move( new Point(x, y), new Point(x-2, y-2) );

0949: // validMoves.add(move);

0950: ArrayList<Move> multiCapMoves = addValidCaptureMoves( x-2, y-2, colour, type, x, y );

0951: addMultiCapMoves( x, y, x-2, y-2, validMoves, multiCapMoves );

0952:

0953: }

0954: if ( x+2 <= 7 && y-2 >= 0 && (x+2 != oldX || y-2 != oldY) && checkers[x+1][y-1] != null && checkers[x+1][y-1].getColour() != colour && checkers[x+2][y-2] == null ) // right capture move allowed

0955: {

0956:

0957: // Move move = new Move( new Point(x, y), new Point(x+2, y-2) );

0958: // validMoves.add(move);

0959: ArrayList<Move> multiCapMoves = addValidCaptureMoves( x+2, y-2, colour, type, x, y );

0960: addMultiCapMoves( x, y, x+2, y-2, validMoves, multiCapMoves );

0961:

0962: }

0963: if ( type == PieceType.BLACK\_KING || type == PieceType.RED\_KING ) // If king piece, also check downward moves

0964: {

0965:

0966: if ( x-2 >= 0 && y+2 <= 7 && (x-2 != oldX || y+2 != oldY) && checkers[x-1][y+1] != null && checkers[x-1][y+1].getColour() != colour && checkers[x-2][y+2] == null ) // left capture move allowed

0967: {

0968:

0969: // Move move = new Move( new Point(x, y), new Point(x-2, y+2) );

0970: // validMoves.add(move);

0971: ArrayList<Move> multiCapMoves = addValidCaptureMoves( x-2, y+2, colour, type, x, y );

0972: addMultiCapMoves( x, y, x-2, y+2, validMoves, multiCapMoves );

0973:

0974: }

0975: if ( x+2 <= 7 && y+2 <= 7 && (x+2 != oldX || y+2 != oldY) && checkers[x+1][y+1] != null && checkers[x+1][y+1].getColour() != colour && checkers[x+2][y+2] == null ) // right capture move allowed

0976: {

0977:

0978: // Move move = new Move( new Point(x, y), new Point(x+2, y+2) );

0979: // validMoves.add(move);

0980: ArrayList<Move> multiCapMoves = addValidCaptureMoves( x+2, y+2, colour, type, x, y );

0981: addMultiCapMoves( x, y, x+2, y+2, validMoves, multiCapMoves );

0982:

0983: }

0984:

0985: }

0986:

0987: }

0988: else if ( this.whoseTurn() == "AI" )

0989: {

0990:

0991: if ( x-2 >= 0 && y+2 <= 7 && (x-2 != oldX || y+2 != oldY) && checkers[x-1][y+1] != null && checkers[x-1][y+1].getColour() != colour && checkers[x-2][y+2] == null ) // left capture move allowed

0992: {

0993:

0994: // Move move = new Move( new Point(x, y), new Point(x-2, y+2) );

0995: // validMoves.add(move);

0996: ArrayList<Move> multiCapMoves = addValidCaptureMoves( x-2, y+2, colour, type, x, y );

0997: addMultiCapMoves( x, y, x-2, y+2, validMoves, multiCapMoves );

0998:

0999: }

1000: if ( x+2 <= 7 && y+2 <= 7 && (x+2 != oldX || y+2 != oldY) && checkers[x+1][y+1] != null && checkers[x+1][y+1].getColour() != colour && checkers[x+2][y+2] == null ) // right capture move allowed

1001: {

1002:

1003: // Move move = new Move( new Point(x, y), new Point(x+2, y+2) );

1004: // validMoves.add(move);

1005: ArrayList<Move> multiCapMoves = addValidCaptureMoves( x+2, y+2, colour, type, x, y );

1006: addMultiCapMoves( x, y, x+2, y+2, validMoves, multiCapMoves );

1007:

1008: }

1009: if ( type == PieceType.BLACK\_KING || type == PieceType.RED\_KING ) // If king piece, also check upward moves

1010: {

1011:

1012: if ( x-2 >= 0 && y-2 >= 0 && (x-2 != oldX || y-2 != oldY) && checkers[x-1][y-1] != null && checkers[x-1][y-1].getColour() != colour && checkers[x-2][y-2] == null ) // left capture move allowed

1013: {

1014:

1015: // Move move = new Move( new Point(x, y), new Point(x-2, y-2) );

1016: // validMoves.add(move);

1017: ArrayList<Move> multiCapMoves = addValidCaptureMoves( x-2, y-2, colour, type, x, y );

1018: addMultiCapMoves( x, y, x-2, y-2, validMoves, multiCapMoves );

1019:

1020: }

1021: if ( x+2 <= 7 && y-2 >= 0 && (x+2 != oldX || y-2 != oldY) && checkers[x+1][y-1] != null && checkers[x+1][y-1].getColour() != colour && checkers[x+2][y-2] == null ) // right capture move allowed

1022: {

1023:

1024: // Move move = new Move( new Point(x, y), new Point(x+2, y-2) );

1025: // validMoves.add(move);

1026: ArrayList<Move> multiCapMoves = addValidCaptureMoves( x+2, y-2, colour, type, x, y );

1027: addMultiCapMoves( x, y, x+2, y-2, validMoves, multiCapMoves );

1028:

1029: }

1030:

1031: }

1032:

1033: }

1034:

1035: return validMoves;

1036:

1037: }

1038:

1039: public void addMultiCapMoves( int x, int y, int newX, int newY, ArrayList<Move> validMoves, ArrayList<Move> multiCapMoves )

1040: {

1041:

1042:

1043: if ( multiCapMoves.isEmpty() )

1044: {

1045:

1046: Move move = new Move( new Point(x, y), new Point(newX, newY) );

1047: validMoves.add(move);

1048:

1049: }

1050: else

1051: {

1052:

1053: for ( Move mcMove : multiCapMoves )

1054: {

1055: Move move = new Move( new Point(x, y), new Point(newX, newY) );

1056: for ( BoardMove bMove : mcMove.moveList )

1057: move.moveList.add( bMove );

1058: validMoves.add(move);

1059: }

1060:

1061: }

1062:

1063: }

1064:

1065: // Returns the valid non-capture moves for a given position

1066: // if conditions check to see if the proposed move end point is within the board bounds,

1067: // and if the space is empty

1068: private ArrayList<Move> addValidRegMoves( int x, int y, String colour )

1069: {

1070:

1071: // System.out.println("Generating regular moves..");

1072: ArrayList<Move> validMoves = new ArrayList<Move>();

1073:

1074: if ( this.whoseTurn() == "Human" )

1075: {

1076:

1077: if ( x-1 >= 0 && y-1 >= 0 && checkers[x-1][y-1] == null ) // regular left move allowed

1078: {

1079:

1080: // System.out.println("Found left move");

1081: Move move = new Move( new Point(x, y), new Point(x-1, y-1) );

1082: validMoves.add(move);

1083:

1084:

1085: }

1086: if ( x+1 <= 7 && y-1 >= 0 && checkers[x+1][y-1] == null ) // regular right move allowed

1087: {

1088:

1089: // System.out.println("Found right move");

1090: Move move = new Move( new Point(x, y), new Point(x+1, y-1) );

1091: validMoves.add(move);

1092:

1093: }

1094: if ( checkers[x][y].getType() == PieceType.BLACK\_KING || checkers[x][y].getType() == PieceType.RED\_KING ) // If king piece, also check downward moves

1095: {

1096:

1097: if ( x-1 >= 0 && y+1 <= 7 && checkers[x-1][y+1] == null ) // regular left move allowed

1098: {

1099:

1100: Move move = new Move( new Point(x, y), new Point(x-1, y+1) );

1101: validMoves.add(move);

1102:

1103: }

1104: if ( x+1 <= 7 && y+1 <= 7 && checkers[x+1][y+1] == null ) // regular right move allowed

1105: {

1106:

1107: Move move = new Move( new Point(x, y), new Point(x+1, y+1) );

1108: validMoves.add(move);

1109:

1110: }

1111:

1112: }

1113:

1114: }

1115: else if ( this.whoseTurn() == "AI" )

1116: {

1117:

1118: if ( x-1 >= 0 && y+1 <= 7 && checkers[x-1][y+1] == null ) // regular left move allowed

1119: {

1120:

1121: // System.out.println("Found left move");

1122: Move move = new Move( new Point(x, y), new Point(x-1, y+1) );

1123: validMoves.add(move);

1124:

1125:

1126: }

1127: if ( x+1 <= 7 && y+1 <= 7 && checkers[x+1][y+1] == null ) // regular right move allowed

1128: {

1129:

1130: // System.out.println("Found right move");

1131: Move move = new Move( new Point(x, y), new Point(x+1, y+1) );

1132: validMoves.add(move);

1133:

1134: }

1135: if ( checkers[x][y].getType() == PieceType.BLACK\_KING || checkers[x][y].getType() == PieceType.RED\_KING ) // If king piece, also check downward moves

1136: {

1137:

1138: if ( x-1 >= 0 && y-1 >= 0 && checkers[x-1][y-1] == null ) // regular left move allowed

1139: {

1140:

1141: Move move = new Move( new Point(x, y), new Point(x-1, y-1) );

1142: validMoves.add(move);

1143:

1144: }

1145: if ( x+1 <= 7 && y-1 >= 0 && checkers[x+1][y-1] == null ) // regular right move allowed

1146: {

1147:

1148: Move move = new Move( new Point(x, y), new Point(x+1, y-1) );

1149: validMoves.add(move);

1150:

1151: }

1152:

1153: }

1154:

1155: }

1156:

1157: return validMoves;

1158:

1159: }

1160:

1161: public void clearSuccessors( ArrayList<Move> validMoves )

1162: {

1163:

1164: validMoves.clear();

1165:

1166: }

1167:

1168: // If game is over, this function is called and resets the state

1169: public void endGame()

1170: {

1171:

1172: checkers = null;

1173: inPlayFlag = false;

1174: activePlayer ="Human";

1175: help = false;

1176: difficulty = 1;

1177:

1178: }

1179:

1180: public String getAIColour()

1181: {

1182:

1183: return aiColour;

1184:

1185: }

1186:

1187: // Captured position can be found by summing old and new pos and dividing by 2

1188: public Point getCapturedPos( BoardMove boardMove )

1189: {

1190:

1191: int oldX = boardMove.getOldPos().x;

1192: int oldY = boardMove.getOldPos().y;

1193: int newX = boardMove.getNewPos().x;

1194: int newY = boardMove.getNewPos().y;

1195:

1196: return new Point( (oldX + newX) / 2, (oldY + newY) / 2 );

1197:

1198: }

1199:

1200: public int getDifficulty()

1201: {

1202:

1203: return difficulty;

1204:

1205: }

1206:

1207: public String getPlayerColour()

1208: {

1209:

1210: return playerColour;

1211:

1212: }

1213:

1214: public ArrayList<Move> generateValidSuccessors( Point pos )

1215: {

1216: String colour;

1217: ArrayList<Move> validMoves = new ArrayList<Move>();

1218: int x = pos.x;

1219: int y = pos.y;

1220:

1221: if ( this.whoseTurn() == "Human" )

1222: {

1223:

1224: colour = this.getPlayerColour();

1225:

1226: if ( checkers[x][y] != null && checkers[x][y].getColour() == colour )

1227: {

1228:

1229: // System.out.println("Getting moves for piece (" + x + ", " + y + ")");

1230: validMoves.addAll( addValidCaptureMoves( x, y, colour, checkers[x][y].getType(), 100, 100 ) );

1231: if ( validMoves.isEmpty() ) // only add regular moves if there are no captures available

1232: {

1233:

1234: // System.out.println("No captures found");

1235: validMoves.addAll( addValidRegMoves( x, y, colour ) );

1236:

1237: }

1238:

1239: }

1240:

1241: }

1242: else

1243: {

1244:

1245: colour = this.getAIColour();

1246: if ( checkers[x][y] != null && checkers[x][y].getColour() == colour )

1247: {

1248:

1249: validMoves.addAll( addValidCaptureMoves( x, y, colour, checkers[x][y].getType(), 100, 100 ) );

1250: if ( validMoves.isEmpty() ) // only add regular moves if there are no captures available

1251: {

1252:

1253: // System.out.println("No captures found");

1254: validMoves.addAll( addValidRegMoves( x, y, colour ) );

1255:

1256: }

1257:

1258: }

1259:

1260: }

1261:

1262: return validMoves;

1263:

1264: }

1265:

1266: public ArrayList<Move> generateAllValidSuccessors()

1267: {

1268:

1269: ArrayList<Move> allValidMoves = new ArrayList<Move>();

1270: boolean captureFound = false;

1271:

1272: for ( int x = 0; x < 8; x++ )

1273: {

1274: for ( int y = 0; y < 8; y++ )

1275: {

1276:

1277: ArrayList<Move> validMoves = generateValidSuccessors( new Point( x, y ) );

1278: for ( Move move : validMoves )

1279: {

1280: if ( captureFound )

1281: {

1282:

1283: if ( move.getMoveType() == "Capture" )

1284: allValidMoves.add( move );

1285:

1286:

1287: }

1288: else if ( move.getMoveType() == "Capture" )

1289: {

1290:

1291: allValidMoves.clear();

1292: captureFound = true;

1293: allValidMoves.add(move);

1294:

1295: }

1296: else

1297: allValidMoves.add(move);

1298: }

1299:

1300: // System.out.println("Valid moves size: "+validMoves.size());

1301:

1302: }

1303: }

1304:

1305: return allValidMoves;

1306:

1307: }

1308:

1309: public boolean getHelp()

1310: {

1311:

1312: return help;

1313:

1314: }

1315:

1316: public boolean humanHasWon()

1317: {

1318:

1319: // System.out.println("Player score: " + this.playerScore() );

1320: if ( this.playerScore() == 12 )

1321: {

1322:

1323: inPlayFlag = false;

1324: return true;

1325:

1326: }

1327: else

1328: return false;

1329:

1330: }

1331:

1332: public boolean inPlay()

1333: {

1334:

1335: return inPlayFlag;

1336:

1337: }

1338:

1339: public boolean isWon()

1340: {

1341:

1342: if ( !inPlay() )

1343: return false;

1344: else

1345: return ( this.humanHasWon() || this.aiHasWon() );

1346:

1347: }

1348:

1349: // Evaluate the utility of a leaf

1350: // Score = (ai\_score + num\_ai\_kings) - (player\_score + num\_player\_kings)

1351: public int leafUtility()

1352: {

1353:

1354: int aiScore = aiScore();

1355: int playerScore = playerScore();

1356: int aiKings = numKings( getAIColour() );

1357: int playerKings = numKings( getPlayerColour() );

1358:

1359: return ( aiScore + aiKings ) - (playerScore + playerKings );

1360:

1361: }

1362:

1363: public int numKings( String colour )

1364: {

1365:

1366: int numKings = 0;

1367: for ( int x = 0; x < 8; x++ )

1368: {

1369: for ( int y = 0; y < 8; y++ )

1370: {

1371:

1372: if ( checkers[x][y] != null && checkers[x][y].getColour() == colour )

1373: if ( checkers[x][y].getType() == PieceType.BLACK\_KING || checkers[x][y].getType() == PieceType.RED\_KING )

1374: {

1375:

1376: numKings++;

1377:

1378: }

1379:

1380: }

1381: }

1382:

1383: return numKings;

1384:

1385: }

1386:

1387: public Move makeAIMove()

1388: {

1389:

1390: int max = -10000;

1391: int ind = -1;

1392: int prev = max;

1393: boolean allSame = true;

1394: ArrayList<MoveAndScore> sameList = new ArrayList<MoveAndScore>();

1395: seCount =0;

1396: deCount=0;

1397: pCount=0;

1398:

1399: if ( this.whoseTurn() != "AI" )

1400: return null;

1401:

1402: aiMoveList = new ArrayList<MoveAndScore>();

1403: minimax( 0, 1, Integer.MIN\_VALUE, Integer.MAX\_VALUE );

1404:

1405: // iterate over successors and return the one with the highest eval result

1406: for ( int i = 0; i < aiMoveList.size(); ++i )

1407: {

1408:

1409: if (max < aiMoveList.get(i).score)

1410: max = aiMoveList.get(i).score;

1411:

1412: }

1413:

1414: // Choose move randomly from those with same score to make AI more unpredictable

1415: for ( int i = 0; i < aiMoveList.size(); ++i )

1416: {

1417:

1418: if ( aiMoveList.get(i).score == max )

1419: sameList.add( aiMoveList.get(i) );

1420:

1421: }

1422:

1423: Random rand = new Random();

1424: ind = rand.nextInt(sameList.size());

1425: return sameList.get(ind).move;

1426:

1427: }

1428:

1429: // AI minimax evaluation

1430: // Generate valid moves, if depth level reached, return utility estimate,

1431: // else start minimax evaluation.

1432: // Create a temporary copy of the game state so that actual game state isn't updated

1433: public int minimax( int depth, int player, int alpha, int beta )

1434: {

1435:

1436: int bestScore;

1437:

1438: if(player == 1)

1439: bestScore = -24;

1440: else

1441: bestScore = 24;

1442:

1443: ArrayList<Move> allValidMoves = this.generateAllValidSuccessors();

1444:

1445: // limit static evaluations acc. to diff. level,

1446: // or not at all at "Impossible" (Level 4)

1447: if(seCount <= seLevel || this.getDifficulty() == 4) {

1448: // determine outcomes and increase SE cost

1449: if (aiHasWon())

1450:

1451: {

1452: seCount++;

1453: return +30; // Static evaluation result for the AI winning

1454:

1455: }

1456: if (humanHasWon())

1457: {

1458:

1459: seCount++;

1460: return -30; // Static evaluation result for the AI losing

1461:

1462: }

1463: if ( depth > maxDepth )

1464: {

1465:

1466: return leafUtility();

1467:

1468: }

1469: }

1470: else

1471: return leafUtility();

1472:

1473: System.out.println("Doing minimax evaluation");

1474:

1475: for ( int i = 0; i < allValidMoves.size(); i++ )

1476: {

1477:

1478: // determine all board positions that aren't occupied

1479: Move move = allValidMoves.get(i);

1480:

1481: // increment dynamic evaluation cost

1482: Game tempGame = new Game( this );

1483: deCount++;

1484:

1485: if ( player == 1 )

1486: { //AI's turn: get the highest result returned by minimax

1487: // place a piece at the first available position

1488:

1489: for ( BoardMove boardMove : move.moveList )

1490: { // iterate through all board positions, 1 for a single capture, multiple for mutlicap

1491: tempGame.updateState( move.getMoveType(), boardMove );

1492: }

1493:

1494: // get the minimax evaluation result for making the previous move

1495: tempGame.setPlayer( "Human" ); // switch player

1496: int currentScore = tempGame.minimax( depth + 1, 2, alpha, beta ); // Increase

1497:

1498: if ( currentScore > bestScore )

1499: bestScore = currentScore;

1500:

1501: alpha = Math.max( alpha, currentScore );

1502: // store a mapping of complete evaluations (at depth 0) and their scores

1503: if ( depth == 0 )

1504: aiMoveList.add(new MoveAndScore(currentScore, move));

1505:

1506: }

1507: else if ( player == 2 )

1508: {//Human's turn: get the lowest result returned by minimax

1509:

1510: for ( BoardMove boardMove : move.moveList ) // iterate through all board positions, 1 for a single capture, multiple for mutlicap

1511: {

1512: tempGame.updateState( move.getMoveType(), boardMove );

1513: }

1514:

1515: tempGame.setPlayer( "AI" ); // switch player

1516: int currentScore = tempGame.minimax(depth + 1, 1, alpha, beta);

1517:

1518: if ( currentScore < bestScore )

1519: bestScore = currentScore;

1520:

1521: beta = Math.min( beta, currentScore );

1522:

1523: }

1524:

1525: // Add AB pruning & count the pruning operations carried out

1526: if( alpha >= beta )

1527: {

1528:

1529: pCount++;

1530: break;

1531:

1532: }

1533:

1534: }

1535:

1536: return bestScore;

1537:

1538: }

1539:

1540: // Player score is max number of pieces minus number of ai player's pieces remaining

1541: // Used to calculate leaf utility and to see if game is won

1542: public int playerScore()

1543: {

1544:

1545: int score = 12;

1546: for ( int x = 0; x < 8; x++ )

1547: {

1548: for ( int y = 0; y < 8; y++ )

1549: {

1550: if ( checkers[x][y] != null )

1551: {

1552:

1553: if ( checkers[x][y].getColour() == this.getAIColour() )

1554: score--;

1555:

1556: }

1557: }

1558: }

1559:

1560: return score;

1561:

1562: }

1563:

1564: public void setColour()

1565: {

1566:

1567: playerColour = activePlayer == "Human" ? "Black" : "Red";

1568: aiColour = playerColour == "Black" ? "Red" : "Black";

1569:

1570: }

1571:

1572: public void setDifficulty( String diffString )

1573: {

1574:

1575: if ( diffString == "Intermediate" )

1576: {

1577:

1578: difficulty = 2;

1579: maxDepth = 5;

1580: seLevel = 3;

1581:

1582: }

1583: else if ( diffString == "Hard" )

1584: {

1585:

1586: difficulty = 3;

1587: maxDepth = 7;

1588: seLevel = 5;

1589:

1590: }

1591: else if ( diffString == "Extreme" )

1592: {

1593:

1594: difficulty = 4;

1595: maxDepth = 10;

1596: seLevel = 10;

1597:

1598: }

1599: else

1600: {

1601:

1602: difficulty = 1;

1603: maxDepth = 2;

1604: seLevel = 1;

1605:

1606: }

1607:

1608: }

1609:

1610: public void setPlayer( String playerString )

1611: {

1612:

1613: activePlayer = playerString;

1614:

1615: }

1616:

1617: public void setHelp( boolean helpFlag )

1618: {

1619:

1620: help = helpFlag;

1621:

1622: }

1623:

1624: // Initialise game state

1625: // Set checkers in the checker board array

1626: // Set game to in play

1627: public void startGame()

1628: {

1629:

1630: inPlayFlag = true;

1631: checkers = new Checker[8][8];

1632: setColour();

1633:

1634: // Top half of board

1635: for ( int y = 0; y < 3; y++)

1636: {

1637: for ( int x = 1; x < 8 ; x+=2 )

1638: {

1639:

1640: int tempX = x;

1641: if ( (y % 2) != 0 )

1642: tempX = x - 1;

1643:

1644: checkers[tempX][y] = new Checker( SQUARESIZE, tempX, y );

1645: if ( activePlayer == "Human" )

1646: checkers[tempX][y].setType( PieceType.RED );

1647: else

1648: checkers[tempX][y].setType( PieceType.BLACK );

1649:

1650: }

1651: }

1652:

1653: // Bottom half of board

1654: for ( int y = 7; y > 4; y--)

1655: {

1656: for ( int x = 0; x < 8 ; x+=2 )

1657: {

1658: int tempX = x;

1659:

1660: if ( (y % 2) == 0 )

1661: tempX = x + 1;

1662:

1663: checkers[tempX][y] = new Checker( SQUARESIZE, tempX, y );

1664:

1665: if (activePlayer == "Human")

1666: checkers[tempX][y].setType( PieceType.BLACK );

1667: else

1668: checkers[tempX][y].setType( PieceType.RED );

1669:

1670: }

1671: }

1672:

1673: }

1674:

1675: // Given a new move, update the internal game state

1676: // Remove captured pieces, move pieces

1677: public void updateState( String moveType, BoardMove boardMove )

1678: {

1679:

1680: int oldX = boardMove.oldPos.x;

1681: int oldY = boardMove.oldPos.y;

1682: int newX = boardMove.newPos.x;

1683: int newY = boardMove.newPos.y;

1684:

1685: checkers[newX][newY] = checkers[oldX][oldY];

1686: checkers[newX][newY].setBoardPos( newX, newY );

1687: checkers[oldX][oldY] = null;

1688:

1689: // Convert piece to king if at the opposite end of board

1690: if ( (this.whoseTurn() == "Human" && newY == 0) || (this.whoseTurn() == "AI" && newY == 7) )

1691: {

1692:

1693: if ( checkers[newX][newY].getType() == PieceType.BLACK )

1694: checkers[newX][newY].setType( PieceType.BLACK\_KING );

1695: else if ( checkers[newX][newY].getType() == PieceType.RED )

1696: checkers[newX][newY].setType( PieceType.RED\_KING );

1697:

1698: }

1699:

1700: // If a capture, find the position of the captured piece, remove from game

1701: if ( moveType == "Capture" )

1702: {

1703:

1704: Point capturedPos = this.getCapturedPos( boardMove );

1705: checkers[capturedPos.x][capturedPos.y] = null;

1706:

1707: }

1708:

1709: }

1710:

1711: public String whoseTurn()

1712: {

1713:

1714: return activePlayer;

1715:

1716: }

1717:

1718: }

1719:

1720: // Class to hold information about an individual checker

1721: // Includes its board position, its associated center in pixel coordinates,

1722: // its colour and piece type

1723: class Checker {

1724:

1725: private int boardPosX = 0;

1726: private int boardPosY = 0;

1727: public int centerX = 0;

1728: public int centerY = 0;

1729: public int coordX = 0;

1730: public int coordY = 0;

1731: private PieceType type;

1732: private String pieceColour;

1733: public static int SQUARESIZE;

1734:

1735: public Checker( int squareSize, int x, int y )

1736: {

1737:

1738: SQUARESIZE = squareSize;

1739: setBoardPos( x, y );

1740:

1741: }

1742:

1743: // Copy constructor

1744: public Checker( Checker original )

1745: {

1746:

1747: Point boardPos = original.getPos();

1748: this.boardPosX = boardPos.x;

1749: this.boardPosY = boardPos.y;

1750: this.centerX = original.centerX;

1751: this.centerY = original.centerY;

1752: this.coordX = original.coordX;

1753: this.coordY = original.coordY;

1754: this.type = original.getType();

1755: this.pieceColour = original.getColour();

1756: this.SQUARESIZE = original.SQUARESIZE;

1757:

1758: }

1759:

1760: public void paint( Graphics g )

1761: {

1762:

1763: int xCoord = centerX - ( SQUARESIZE / 2 ) + 5;

1764: int yCoord = centerY - ( SQUARESIZE / 2 ) + 5;

1765: // int xCoord = coordX + 5;

1766: // int yCoord = coordY + 5;

1767: int pieceSize = (int) ( 0.8 \* SQUARESIZE );

1768:

1769: if ( type == PieceType.BLACK || type == PieceType.BLACK\_KING )

1770: g.setColor( Color.black );

1771: else

1772: g.setColor( Color.red );

1773:

1774: g.fillOval( xCoord, yCoord, pieceSize, pieceSize );

1775:

1776: if (type == PieceType.RED\_KING || type == PieceType.BLACK\_KING)

1777: {

1778:

1779: String text = "K";

1780: FontMetrics fm = g.getFontMetrics();

1781: double textWidth = fm.getStringBounds(text, g).getWidth();

1782: g.setColor( new Color(163, 134, 19) );

1783: g.drawString(text, (int) (centerX-textWidth/2), centerY);

1784:

1785: }

1786:

1787: }

1788:

1789: public String getColour()

1790: {

1791:

1792: return pieceColour;

1793:

1794: }

1795:

1796: public PieceType getType()

1797: {

1798:

1799: return type;

1800:

1801: }

1802:

1803: public Point getCenter()

1804: {

1805:

1806: return new Point(centerX, centerY);

1807:

1808: }

1809:

1810: public Point getPos()

1811: {

1812:

1813: return new Point(boardPosX, boardPosY);

1814:

1815: }

1816:

1817: // Set checker's internal board position, and update its pixel coordinate centers

1818: public void setBoardPos( int x, int y )

1819: {

1820:

1821: if ( x >= 8 || x < 0 || y >= 8 || y < 0 )

1822: throw new IllegalArgumentException( "Piece's X or Y board position is invalid" );

1823:

1824: boardPosX = x;

1825: boardPosY = y;

1826: coordX = boardPosX \* SQUARESIZE;

1827: coordY = boardPosY \* SQUARESIZE;

1828: centerX = (int) ( SQUARESIZE \* boardPosX + SQUARESIZE / 2 );

1829: centerY = (int) ( SQUARESIZE \* boardPosY + SQUARESIZE / 2 );

1830:

1831: }

1832:

1833: public void setPos( int x, int y )

1834: {

1835:

1836: centerX = x;

1837: centerY = y;

1838:

1839: }

1840:

1841: public void setType( PieceType pieceType )

1842: {

1843:

1844: type = pieceType;

1845: if (type == PieceType.BLACK || type == pieceType.BLACK\_KING)

1846: pieceColour = "Black";

1847: else

1848: pieceColour = "Red";

1849:

1850: }

1851: }

1852:

1853: // Class for a game move, which holds the move type ("Move" or "Capture"),

1854: // and a list of the moves (move list size is only > 1 for a multi capture move)

1855: class Move

1856: {

1857: public ArrayList<BoardMove> moveList = new ArrayList<BoardMove>();

1858: public String moveType;

1859:

1860: public Move( Point pos1, Point pos2 )

1861: {

1862:

1863: moveList.add( new BoardMove( pos1, pos2 ) );

1864: if ( (pos2.x - pos1.x == 1 || pos2.x - pos1.x == -1) && (pos2.y - pos1.y == 1 || pos2.y - pos1.y == -1) )

1865: moveType = "Move";

1866: else

1867: moveType = "Capture";

1868:

1869: }

1870:

1871:

1872: public String getMove()

1873: {

1874:

1875: BoardMove move = (this.moveList).get(0);

1876: String string = move.oldPos + " -> " + move.newPos;

1877: // return this.moveList;

1878: return string;

1879:

1880: }

1881:

1882: public Point getMovePos()

1883: {

1884:

1885: BoardMove move = (this.moveList).get(0);

1886: return move.newPos;

1887:

1888: }

1889:

1890: public Point getStartPos()

1891: {

1892:

1893: BoardMove move = (this.moveList).get(0);

1894: return move.oldPos;

1895:

1896: }

1897:

1898: public String getMoveType()

1899: {

1900:

1901: return this.moveType;

1902:

1903: }

1904:

1905: }

1906:

1907: // Board move is a helper class for move, containing the starting position of

1908: // the move, and the end position

1909: class BoardMove

1910: {

1911:

1912: // private Point oldPos;

1913: // private Point newPos;

1914: public Point oldPos;

1915: public Point newPos;

1916: // public Point capturedPos = null;

1917:

1918: public BoardMove( Point pos1, Point pos2 )

1919: {

1920:

1921: oldPos = pos1;

1922: newPos = pos2;

1923:

1924: }

1925:

1926: public Point getOldPos()

1927: {

1928:

1929: return oldPos;

1930:

1931: }

1932:

1933: public Point getNewPos()

1934: {

1935:

1936: return newPos;

1937:

1938: }

1939:

1940: public String getMove()

1941: {

1942:

1943: String string = this.oldPos + " -> " + this.newPos;

1944: // return this.moveList;

1945: return string;

1946:

1947: }

1948:

1949: }

1950:

1951: // Used for AI move evaluation, holds a Move and its associated minimax score

1952: class MoveAndScore {

1953:

1954: int score;

1955: Move move;

1956:

1957: public MoveAndScore(int score, Move move) {

1958:

1959: this.score = score;

1960: this.move = move;

1961:

1962: }

1963:

1964: // Copy constructor

1965: public MoveAndScore( MoveAndScore original ) {

1966:

1967: this.score = original.score;

1968: this.move = original.move;

1969:

1970: }

1971:

1972: }

1973:

1974: // Enum of the possible piece types

1975: enum PieceType

1976: {

1977:

1978: BLACK,

1979: BLACK\_KING,

1980: RED,

1981: RED\_KING

1982:

1983: }