169414

Draughts with Minmax employed Artificial Intelligence

Abstract

The purpose of this report is to outline and explore an implementation of an artificially intelligent agent serving as an opponent in the classic strategic board game, Draughts. The agent will be implemented with a Minmax algorithm, an algorithm centred around minimising the possible loss for a worst-case scenario.

1. Introduction

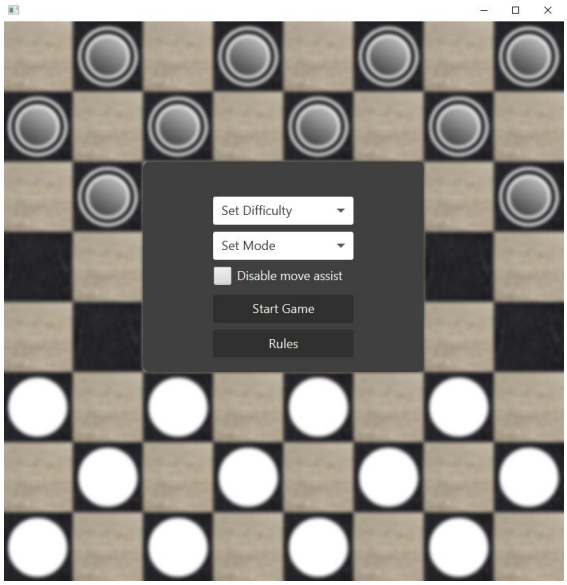
Draughts as commonly played is a player vs player board game involving diagonal moves and mandatory captures by jumping an opposing piece. Whilst the rules of draughts are relatively simple, features such as king pieces of which can move diagonally in any direction and a rule that allows the killing piece of a king to become one itself, adds a further layer of complexity to the game.

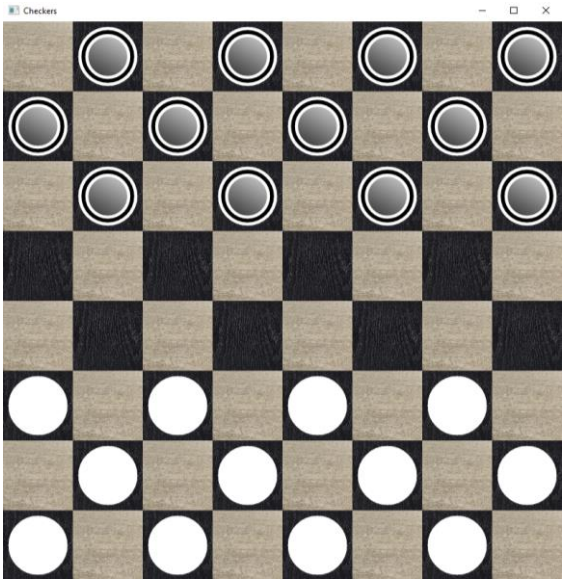
Whilst designing and creating the game of Draughts is challenging, creating an agent that plays the game intelligently will be the true challenge. The first two methods that come to mind for implementing this behavior are; trying all possible combinations of moves, a form of exhaustive search, and secondly, defining a limited set of behaviors with if-then statements (Grau, 2019). However, trying all the possible combinations is unfeasible due to the complexity of the search space and whilst defining a limited set of behaviours might work for simplistic games, it could perhaps be too simplistic for draughts. This leads on to a new design of implementation, a form of adversarial search, a minmax algorithm.

The minmax algorithm splits the game in to two parts; the player who starts plays to win and pursues the path of highest utility and the player who follows plays to minimize the other players result. The underlying mechanism of the algorithm is a depth-first search of the search tree, with the best score then passed up the tree to the root node. Further explanation of the minmax algorithm and heuristics used can be found within the implementation section.

2. Analysis and Design

To create a well-designed and performing project, it is important that the requirements for the project are laid out to allow for appropriate testing. Appendix item 1 shows the specification table for the project. These specification points will be referred to throughout the report.





The design of the project was kept simple. A dark theme was used for a modern look with the menu GUI fading out into the board once the user has selected to start the game. Dropbox selections are

used for the difficulty as well as the game mode. Move assist is disabled with a click of a checkbox. Finally, the rules of the game can be easily seen by clicking the “Rules” button.

3. Implementation

3.1 Game control

Control of the game will be handled by mouse input. To move a piece the player will be required to click the piece they would like to move and then select the location they would like to move the piece to.

A feature of the game will be a move assist option, if enabled, upon the first click of a piece its possible moves will be displayed to the user. Constraint checks will be necessary here to ensure that the appropriate moves are displayed to the user.

3.2 Class Structure

The following section will explore how the classes are structured, whilst also explaining and showing implementation for the specification features.

3.2.1 Main Class

The main App class handles most board operations as well as the game mechanics. The code as outlined below shows how the board is created. The board screen is a JavaFX scene set to display the grid as configured below. Although this was a working implementation, I believe that creating a grid of buttons as opposed to images would have made some aspects of the project easier as it wouldn’t be necessary to get the piece based on X and Y coordinates of the click.

```
1.  public void displayPieces() {
2.      for (int j = 0; j < 8; j++) {
3.          for (int i = 0; i < 8; i++) {
4.              Pane tile = new Pane();
5.              if ((j + i) % 2 == 0) {
6.                  tile.getChildren().add(new ImageView("whitebg.png"));
7.              }
8.              if ((j + i) % 2 != 0) {
9.                  tile.getChildren().add(new ImageView("blackbg.png"));
10.             }
11.             tile.setMinSize(100, 100);
12.             grid.add(tile, i, j);
13.         }
14.     }
15.     for (Piece piece : board.pieces) {
16.         grid.add(piece.getImage(), piece.getX(), piece.getY());
17.         grid.add(piece.getImage(), piece.getX(), piece.getY());
18.     }
19. }
```

The method for handling the click of the mouse was complicated and has room for improvement but does allow me to handle the necessary constraint checks and requirements as laid out in the specification. User moves are validated with simple checks to ensure that they are; a legal move, it is the users turn and that there are no forced moves to be made. On the click of a valid piece, its moves are obtained with simple logic if statements below. This piece of code is extended to obtain possible jumps by checking the piece diagonal is an opposing player and the location diagonal to the opposing player is free. Having the grid allowed me to quite simply work out the moves on an X and Y basis.

```
1.  public ArrayList<Moves> getMovesUp() {
2.          ArrayList<Moves> possibleMoves = new ArrayList<Moves>();
3.          /* Left side check */
4.          if (App.board.getPiece(x - 1, y - 1) == null && isLegal(x - 1, y - 1)) {
5.              Moves m = new Moves();
6.              m.newx = x - 1;
7.              m.newy = y - 1;
8.              m.originalx = x;
9.              m.originaly = y;
```

```
            m.piece = this;
            possibleMoves.add(m);
        }
        /* Right side check */
        if (App.board.getPiece(x + 1, y - 1) == null && isLegal(x + 1, y - 1)) {
            Moves m = new Moves();
            m.newx = x + 1;
            m.newy = y - 1;
            m.originalx = x;
            m.originaly = y;
            m.piece = this;
            possibleMoves.add(m);
        }
        return possibleMoves;
    }
```

User moves are validated by comparing the tried move to the array of returned possible moves above.

From this, forced capturing is relatively easy to implement. A simple loop to see if there is a move that can kill piece, if so, forced capturing is turned on and the player must make a move that results in a kill. The possible moves on the GUI are also updated to only display the forced moves available.

```
public void hasToTake(PieceValue value) {
        for (Piece piece : pieces) {
            piece.getMoves();
            if (piece.getValue() == value) {
                for (Moves move : piece.moves) {
                    if (move.killedPiece != null) {
                        forceTake = true;
                        forcedMoves.add(move);
                    }
                }
            }
        }
    }
```

Next, kings were implemented. The king pieces were identified with an isKing Boolean in the piece class. Conversion of a normal piece to a king was simple and dependent on the Y value of the piece. If a white piece is at Y = 0 or a red piece is at Y = 7, then set them to kings, as shown below. A similar logic was used for the AI kings.

```
if (board.activePiece.getValue() == PieceValue.WHITE && board.activePiece.y == 0) {

    board.activePiece.isKing = true;
    move.wasking = true;
}
if (board.activePiece.getValue() == PieceValue.RED && board.activePiece.y == 7) {
    board.activePiece.isKing = true;
    move.wasking = true;
}
```

This was extended to allow for regicide, whereby the killing piece of a king becomes one itself. A simple check to see if the killed piece was a king is performed, if so, set the killing piece to a king.

```
if (move.killedPiece != null && move.killedPiece.isKing == true) {
    board.activePiece.isKing = true;
    board.pieces.remove(move.killedPiece);
}
```

Users are alerted to an invalid move. This is dependent on the Boolean badMove being true. This Boolean is set to true if the clicked square wasn't either a valid move or a clickable piece.

```
if (board.badMove == true) {
    Alert alert = new Alert(AlertType.ERROR);
    alert.setTitle("Invalid Move");
    alert.setHeaderText("This is an invalid move, please refer to the rules for hel
p.");
    alert.showAndWait();
}
```

The final specification point addressed in the main class, aside from the applications GUI, is the adjustable difficulty of the AI. This was achieved by stopping the minmax recursion when it reaches a depth equivalent to the difficulty. The difficulty was set via a JavaFX ComboBox and handled as outlined below.

```
ComboBox<String> comboBox = new ComboBox<String>();
comboBox.getItems().addAll("1 - Easy", "2", "3", "4", "5 - Difficult");
comboBox.setValue("Set Difficulty");
comboBox.setLayoutX(300);
comboBox.setLayoutY(250);

comboBox.setOnAction((e) -> {
    if (comboBox.getValue() == "1 - Easy") {
        App.board.difficulty = 2;
    }
    if (comboBox.getValue() == "2") {
        App.board.difficulty = 3;
    }
    if (comboBox.getValue() == "3") {
        App.board.difficulty = 4;
    }
    if (comboBox.getValue() == "4") {
        App.board.difficulty = 5;
    }
    if (comboBox.getValue() == "5 - Difficult") {
        App.board.difficulty = 7;
    }
});
```

Project specifications 5, 7, 12, 16, 17 and 18 have been implemented as outlined above.

3.2.2 Board Class

The board class contains all the specification for the board object, as well as containing several useful functions; switchTurn() whereby the ai is instructed to make its move, getPiece(int x, int y) of which returns a piece at the given coordinates and finally, hasWon() of which checks if the game is over.

The board class also handles the initial setup of the board by assigning the correct pieces to the correct location. Below is the information held within board.

```
ArrayList<Piece> pieces = new ArrayList<Piece>();
Piece activePiece;
Minmax ai;
int turn;
int difficulty = 1;
boolean forceTake = false;
boolean showMoves = true;
boolean aiIsPlaying = true;
boolean badMove = false;
boolean gameOver = false;
```

3.2.3 Piece Class

The piece class contains the structure of a piece object as well as the functions used to obtain the moves of a piece. The general characteristics of a piece are as below.

```
ArrayList<Moves> moves = new ArrayList<Moves>();
PieceValue value;
int x;
int y;
boolean isKing;
```

3.2.4 Moves Class

A move object allows for tracing of moves during the game. The object contains the following characteristics:

```
/* Piece being moved */
Piece piece;
/* Piece killed */
Piece killedPiece;
/* Original position */
int originalx;
int originaly;
/* New position */
int newx;
int newy;
/* Move score */
int score;
/* Was piece a king */
boolean wasking = false;
```

3.2.5 Minmax Class

The backbone of the AI was controlled by the Minmax class. The class features an implementation of the minmax algorithm with alpha-beta pruning. The base algorithm is as below, a more in-depth explanation to the implementation can be found within the code files of the appendix, class Minmax.

```
minmax(node, depth, a, B, player)
    if depth == 0 or node == leaf
        return node;
    if player == MAX
        v = -infinity
        for each child of node
            v = max(v, minmax(child, depth - 1, a, B, MIN))
            a = max(a, v)
            if a >= B
                break
        return v
    if player == MIN
        v = infinity
        for each child of node
            v = min(v, minmax(child, depth - 1, a, B, MAX))
            b = min(B, v)
            if a >= B
                break
        return v
```

The algorithm as implemented features alpha-beta pruning. Alpha beta pruning seeks to decrease the number of nodes evaluated. It works by preventing further evaluation of a move when there is at least one possibility that the move is worse than a previous one. Moves are generated with state representation through the premade functions as explored in the Piece class and then assessed by the successor function.

Whilst implementing the algorithm was relatively painless, there was a few scenarios that arose whilst doing so. The first; the algorithm was not correctly evaluating the moves of both players. The second; King pieces for the AI player would often not correctly update. The third; King pieces occasionally behave out of the ruleset, more on this can be found in section 4.1.

One of the most important elements of the minmax evaluation was the heuristics used to evaluate the current board state. To begin with the only evaluation was if the game was over. This meant that the AI performed almost randomly until towards the end of the game, where it begun to show some intelligent behavior behind its decision. The next addition was to add a scoring for the number of pieces on each side. This caused the AI to develop some nature of preserving its life, as well as an urge to take pieces where necessary. Finally, the same was implemented for king pieces. By the end of development, the AI was showing promising behavior.

4. Program Analysis

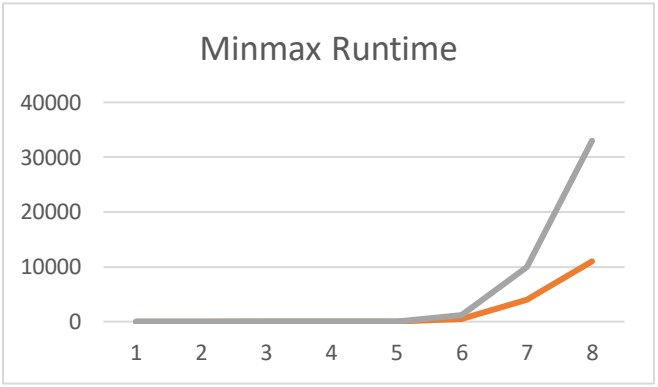
4.1 Program Performance

The project as a program works well and behaves as it should. The game allows users to select the difficulty, toggle the helper mode and play versus the AI or another player. However, it must be recorded that there is a reoccurring bug that has been difficult to replicate. It appears that occasionally when the minmax algorithm runs and creates a king piece in the predicted state space, it will not revert the change. Despite several attempts to alleviate this problem it was unsuccessful, although the reoccurrence of the bug has decreased, it is still occurring in roughly10 % of games played versus the AI.

A final flaw in the design of the system is the launcher window. The window does not close and creates the appearance that multiple games can be run. However, it must be noted that to restart another game, the launcher must be closed and the whole program run again.

4.2 Minmax Analysis

The algorithm performs as expected, with its efficiency increased drastically upon implementing Alpha-beta pruning. Run times differ extensively after pruning. Figure 3 below shows the run time of the minmax algorithm (grey line) and the runtime of the pruned version (orange line).



4.3 Analysis against the specification

The project fulfils specification points 1 to 13 and 16 to 25. Missing specification points 14 and 15 were difficult to implement. Multi step moves consist of allowing the user to jump again if there is a piece to be taken. This was easy to implement for the user-controlled side, but I was unable to let the AI evaluate these multi-step moves. To prevent the game from being unfair and weighted towards the human player, I removed this functionality all together.

Apart from this it is believed that the project has been well created and designed, allowing the user to test their draughts ability against an AI opponent with the only limiting factor being the computing power of the host PC.

5. Conclusion

To conclude, it was a fun and engaging project to complete. Debugging has played a vital role in the success of the project and allowed for a greater understand of the problem at hand. Although there are several improvements that could be made, it is believed the following would be the most necessary:

- Add position-based scoring of the pieces, pieces toward the edge will be weighted more than those in the center
- Create a player class that allows for a greater control of turns and modes
- Expand the board class to better support gameplay functionality such as prevention of invalid moves and clicks

Appendix

1	Interactive checkers gameplay (user vs. AI) of some sort
2	Valid state representation
3	Successor function of some sort
4	Valid successor function
5	Validation of user moves
6	Successor function generates AI moves
7	Rejection of invalid user moves, with explanation given
8	Elements of Minimax evaluation present
9	Valid Minimax evaluation
10	Minimax evaluation uses elements of pruning
11	Minimax evaluation uses valid Alpha-Beta pruning
12	Variable level of verifiable AI cleverness, adjustable by the user (e.g. a difficulty setting)
13	Entirely valid AI moves
14	Multi-step user moves
15	Multi-step AI moves
16	Forced capture (If there is a capture opportunity, the capturing move must be made)
17	Automatic king conversion
18	Regicide (If a normal piece manages to capture a king, it is itself upgraded to a king)
19	Board representation displayed on screen
20	The interface properly updates the display after completed moves (User and AI moves)
21	Fully interactive GUI that uses graphics
22	Mouse interaction focus, e.g., click to select & click to place, or drag & drop (better)
23	GUI pauses appropriately to show the intermediate steps in multi-step moves
24	Dedicated display of the rules (e.g., a corresponding button opening a pop-up window)
25	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

App.java

```
1.  package draughts;
2.
3.  import javafx.application.Application;
4.  import javafx.event.ActionEvent;
5.  import javafx.event.EventHandler;
6.  import javafx.scene.Scene;
7.  import javafx.scene.control.Alert;
8.  import javafx.scene.control.Alert.AlertType;
9.  import javafx.scene.control.Button;
10. import javafx.scene.control.CheckBox;
11. import javafx.scene.control.ComboBox;
12. import javafx.scene.image.ImageView;
13. import javafx.scene.input.MouseEvent;
14. import javafx.scene.layout.GridPane;
15. import javafx.scene.layout.Pane;
16. import javafx.scene.paint.Color;
17. import javafx.scene.shape.Rectangle;
18. import javafx.stage.Stage;
```

```
19.
20. public class App extends Application {
21.
22.     GridPane grid;
23.     static Board board;
24.     boolean clicked = false;
25.
26.     @SuppressWarnings("exports")
27.     @Override
28.     public void start(Stage stage) {
29.         stage.setScene(startMenu());
30.         stage.show();
31.     }
32.
33.     public void handleGui() {
34.         board = new Board();
35.         grid = new GridPane();
36.         displayPieces();
37.         events();
38.     }
39.     /** Display startMenu **/
40.     @SuppressWarnings("exports")
41.     public Scene startMenu() {
42.         Pane pane = new Pane();
43.         ComboBox<String> comboBoxPlayer = new ComboBox<String>();
44.         comboBoxPlayer.getItems().addAll("Player vs Computer", "Player vs Player");

45.         comboBoxPlayer.setValue("Set Mode");
46.         comboBoxPlayer.setLayoutX(300);
47.         comboBoxPlayer.setLayoutY(300);
48.         comboBoxPlayer.setStyle("-fx-text-fill: white; -fx-font-size: 18px; -fx-
    background-color: white; -fx-max-width:200;");
49.
50.         ComboBox<String> comboBox = new ComboBox<String>();
51.         comboBox.getItems().addAll("1 - Easy", "2", "3", "4", "5 - Difficult");
52.         comboBox.setValue("Set Difficulty");
53.         comboBox.setLayoutX(300);
54.         comboBox.setLayoutY(250);
55.         comboBox.setStyle("-fx-text-fill: white; -fx-font-size: 18px; -fx-
    background-color: white; -fx-min-width:200;");
56.
57.         CheckBox cb = new CheckBox("Disable move assist");
58.         cb.setLayoutX(301);
59.         cb.setLayoutY(350);
60.         cb.setStyle("-fx-background-color: #404040; -fx-text-fill: white; -fx-font-
    size: 17.5px; ");
61.
62.         Button button = new Button("Start Game");
63.         button.setLayoutX(300);
64.         button.setLayoutY(390);
65.         button.setMinWidth(200);
66.         button.setStyle("-fx-background-color: #303030; -fx-text-fill: white; -fx-
    font-size: 18px; ");
67.
68.         String rules = new String("http://www.indepthinfo.com/checkers/play.shtml")
    ;
69.         Button helpButton = new Button("Rules");
70.         helpButton.setLayoutX(300);
71.         helpButton.setLayoutY(440);
72.         helpButton.setMinWidth(200);
73.         helpButton.setStyle("-fx-background-color: #303030; -fx-text-fill: white; -
    fx-font-size: 18px; ");
74.
75.         Rectangle r = new Rectangle();
76.         r.setX(200);
77.         r.setY(200);
```

```
78.         r.setWidth(400);
79.         r.setHeight(300);
80.         r.setArcWidth(20);
81.         r.setArcHeight(20);
82.         r.setFill(Color.web("404040"));
83.
84.         pane.getChildren().add(new ImageView("bluurredbg.jpg"));
85.         pane.getChildren().add(r);
86.         pane.getChildren().add(button);
87.         pane.getChildren().add(helpButton);
88.         pane.getChildren().add(cb);
89.         pane.getChildren().add(comboBox);
90.         pane.getChildren().add(comboBoxPlayer);
91.
92.         Scene scene = new Scene(pane, 800, 800);
93.         handleGui();
94.         button.setOnAction(new EventHandler<ActionEvent>() {
95.             public void handle(ActionEvent event) {
96.                 Stage stage = new Stage();
97.                 stage.setTitle("Checkers");
98.                 stage.setScene(new Scene(grid, 800, 800));
99.                 stage.show();
100.
101.                }
102.            });
103.            comboBox.setOnAction((e) -> {
104.                if (comboBox.getValue() == "1 - Easy") {
105.                    App.board.difficulty = 2;
106.                }
107.                if (comboBox.getValue() == "2") {
108.                    App.board.difficulty = 3;
109.                }
110.                if (comboBox.getValue() == "3") {
111.                    App.board.difficulty = 4;
112.                }
113.                if (comboBox.getValue() == "4") {
114.                    App.board.difficulty = 5;
115.                }
116.                if (comboBox.getValue() == "5 - Difficult") {
117.                    App.board.difficulty = 7;
118.                }
119.
120.            });
121.
122.            helpButton.setOnAction((e) -> {
123.                getHostServices().showDocument(rules);
124.
125.            });
126.            comboBoxPlayer.setOnAction((e) -> {
127.                if (comboBoxPlayer.getValue() == "Player vs Computer") {
128.                    App.board.aiIsPlaying = true;
129.                }
130.                if (comboBoxPlayer.getValue() == "Player vs Player") {
131.                    App.board.aiIsPlaying = false;
132.                }
133.            });
134.            cb.setOnAction((e) -> {
135.                if (cb.isSelected()) {
136.                    App.board.showMoves = false;
137.                }
138.            });
139.            return scene;
140.
141.        }
142.        /** Display pieces and board **/
143.        public void displayPieces() {
```

```
144.                 for (int j = 0; j < 8; j++) {
145.                     for (int i = 0; i < 8; i++) {
146.                         Pane tile = new Pane();
147.                         if ((j + i) % 2 == 0) {
148.                             tile.getChildren().add(new ImageView("whitebg.png"));
149.                         }
150.                         if ((j + i) % 2 != 0) {
151.                             tile.getChildren().add(new ImageView("blackbg.png"));
152.                         }
153.                         tile.setMinSize(100, 100);
154.                         grid.add(tile, i, j);
155.                     }
156.                 }
157.                 for (Piece piece : board.pieces) {
158.                     grid.add(piece.getImage(), piece.getX(), piece.getY());
159.                     grid.add(piece.getImage(), piece.getX(), piece.getY());
160.                 }
161.             }
162.             /** Display available moves **/
163.             public void displayMoves(Piece piece) {
164.                 for (Moves move : piece.moves) {
165.                     if (piece.getValue() == PieceValue.RED) {
166.                         if (piece.isKing == false) {
167.                             if (piece.isLegal(move.newx, move.newy)) {
168.                                 ImageView image = new ImageView("blackOpaque.png");

169.                                 grid.add(image, move.newx, move.newy);
170.                             }
171.                         } else {
172.                             if (piece.isLegal(move.newx, move.newy)) {
173.                                 ImageView image = new ImageView("opaqueBlackKing.png
    ");
174.                                 grid.add(image, move.newx, move.newy);
175.                             }
176.                         }
177.
178.                     }
179.                     if (piece.getValue() == PieceValue.WHITE) {
180.                         if (piece.isKing == false) {
181.                             ImageView image = new ImageView("opaqueWhite.png");
182.                             if (piece.isLegal(move.newx, move.newy)) {
183.                                 grid.add(image, move.newx, move.newy);
184.                             }
185.                         } else {
186.                             if (piece.isLegal(move.newx, move.newy)) {
187.                                 ImageView image = new ImageView("opaqueWhiteKing.png
    ");
188.                                 grid.add(image, move.newx, move.newy);
189.                             }
190.                         }
191.                     }
192.                 }
193.             }
194.             /** Display forced captures **/
195.             public void displayForcedMoves() {
196.                 for (Moves move : board.forcedMoves) {
197.                     if (move.piece.getValue() == PieceValue.RED) {
198.                         if (move.piece.isKing == false) {
199.                             if (move.piece.isLegal(move.newx, move.newy)) {
200.                                 ImageView image = new ImageView("blackOpaque.png");

201.                                 grid.add(image, move.newx, move.newy);
202.                             }
203.                         } else {
204.                             if (move.piece.isLegal(move.newx, move.newy)) {
```

```
205.                                 ImageView image = new ImageView("opaqueBlackKing.png
    ");
206.                                 grid.add(image, move.newx, move.newy);
207.                             }
208.                         }
209.
210.                     }
211.                     if (move.piece.getValue() == PieceValue.WHITE) {
212.                         if (move.piece.isKing == false) {
213.                             ImageView image = new ImageView("opaqueWhite.png");
214.                             if (move.piece.isLegal(move.newx, move.newy)) {
215.                                 grid.add(image, move.newx, move.newy);
216.                             }
217.                         } else {
218.                             if (move.piece.isLegal(move.newx, move.newy)) {
219.                                 ImageView image = new ImageView("opaqueWhiteKing.png
    ");
220.                                 grid.add(image, move.newx, move.newy);
221.                             }
222.                         }
223.                     }
224.                 }
225.             }
226.             /** Reset display **/
227.             public void clearDisplay() {
228.                 grid.getChildren().clear();
229.             }
230.             /** Handle mouse click of board **/
231.             public void events() {
232.                 grid.addEventHandler(MouseEvent.MOUSE_CLICKED, e -> {
233.                     int x = (int) Math.floor(e.getX() / 100);
234.                     int y = (int) Math.floor(e.getY() / 100);
235.                     Piece piece = board.getPiece(x, y);
236.                     if (piece != null) {
237.                         if (board.aiIsPlaying && piece.getValue() == PieceValue.RED)
    {
238.                         } else {
239.                             piece.moves.clear();
240.                             board.forcedMoves.clear();
241.                             board.hasToTake(piece.getValue());
242.                             if (board.forcedMoves.isEmpty()) {
243.                                 clearDisplay();
244.                                 displayPieces();
245.                                 piece.getMoves();
246.                                 if (board.showMoves == true) {
247.                                     displayMoves(piece);
248.                                 }
249.                                 board.forcedMoves.clear();
250.                             } else {
251.                                 clearDisplay();
252.                                 piece.getMoves();
253.                                 displayPieces();
254.                                 if (board.showMoves == true) {
255.                                     displayForcedMoves();
256.                                 }
257.                                 board.forcedMoves.clear();
258.                             }
259.                             board.badMove = false;
260.                             board.activePiece = piece;
261.                             ImageView activeImage = new ImageView("activePiece.png")
    ;
262.                             grid.add(activeImage, board.activePiece.getX(), board.ac
    tivePiece.getY());
263.                         }
264.                     } else {
265.                         board.badMove = true;
```

```
266.                     }
267.                     /* Looping to see if click matches up with a currently displayed
     move */
268.                     if(board.activePiece != null) {
269.                     for (int i = 0; i < board.activePiece.moves.size(); i++) {
270.                         Moves move = board.activePiece.moves.get(i);
271.                         board.hasToTake(board.activePiece.getValue());
272.                         if (x == move.newx && y == move.newy) {
273.                             if (move.killedPiece != null || board.forcedMoves.isEmpt
    y()) {
274.                                 board.activePiece.x = x;
275.                                 board.activePiece.y = y;
276.                                 board.forcedMoves.clear();
277.                                 App.board.badMove = false;
278.                                 if (move.killedPiece != null && move.killedPiece.isK
    ing == true) {
279.                                     board.activePiece.isKing = true;
280.                                     board.pieces.remove(move.killedPiece);
281.                                 }
282.                                 if(move.killedPiece != null) {
283.                                     board.pieces.remove(move.killedPiece);
284.                                 }
285.                                 if (board.activePiece.getValue() == PieceValue.WHITE
     && board.activePiece.y == 0) {
286.                                     board.activePiece.isKing = true;
287.                                     move.wasking = true;
288.                                 }
289.                                 if (board.activePiece.getValue() == PieceValue.RED &
    & board.activePiece.y == 7) {
290.                                     board.activePiece.isKing = true;
291.                                     move.wasking = true;
292.                                 }
293.                                 board.activePiece.moves.clear();
294.                                 if (App.board.hasWon(PieceValue.RED)) {
295.                                     Alert alert = new Alert(AlertType.ERROR);
296.                                     alert.setTitle("Game over");
297.                                     alert.setHeaderText("The game is over");
298.                                     alert.showAndWait();
299.                                     App.board.gameOver = true;
300.                                 }
301.                                 if (App.board.hasWon(PieceValue.WHITE)) {
302.                                     Alert alert = new Alert(AlertType.ERROR);
303.                                     alert.setTitle("Game over");
304.                                     alert.setHeaderText("The game is over");
305.                                     alert.showAndWait();
306.                                     App.board.gameOver = true;
307.                                 }
308.                                 board.switchTurn();
309.                                 clearDisplay();
310.                                 displayPieces();
311.                                 board.forcedMoves.clear();
312.                                 board.forceTake = false;
313.                             }
314.                         }
315.                     }
316.                     if (board.badMove == true) {
317.                         Alert alert = new Alert(AlertType.ERROR);
318.                         alert.setTitle("Invalid Move");
319.                         if(board.forceTake == true) {
320.                         alert.setHeaderText("There is a forced capture available.");

321.                         }else {
322.                             alert.setHeaderText("This is an invalid move.");
323.                         }
324.                         alert.showAndWait();
325.                     }
```

```
                }
            });
        }

        public static void main(String[] args) {
            launch(args);
        }

```

Board.java

```
package draughts;

import java.util.ArrayList;

import javafx.scene.control.Alert;
import javafx.scene.control.Alert.AlertType;

public class Board {

    ArrayList<Piece> pieces = new ArrayList<Piece>();
    Piece activePiece;
    Minmax ai;
    int turn;
    int difficulty = 1;
    boolean forceTake = false;
    boolean showMoves = true;
    boolean aiIsPlaying = true;
    boolean badMove = false;
    boolean gameOver = false;
    /** Board constructor **/
    public Board() {
        for (int j = 0; j < 8; j++) {
            for (int i = 0; i < 8; i++) {
                if ((j + i) % 2 != 0) {
                    if (j < 3) {
                        Piece piece = new Piece(PieceValue.RED, i, j);
                        pieces.add(piece);
                    } else if (j > 4) {
                        Piece piece = new Piece(PieceValue.WHITE, i, j);
                        pieces.add(piece);
                    }
                }
            }
        }
        this.ai = new Minmax();
    }
    /** Evaluate any forced captures **/
    ArrayList<Moves> forcedMoves = new ArrayList<Moves>();
    public void hasToTake(PieceValue value) {
        for (Piece piece : pieces) {
            piece.getMoves();
            if (piece.getValue() == value) {
                for (Moves move : piece.moves) {
                    if (move.killedPiece != null) {
                        forceTake = true;
                        forcedMoves.add(move);
                    }
                }
            }
        }
    }
    /** Switch turn of player **/
    public void switchTurn() {
        if (turn == 0) {
```

```
            turn = 1;
            if (aiIsPlaying == true) {
                if (gameOver == false) {
                    ai.start();
                }
            }
        }else {
            if (App.board.hasWon(PieceValue.RED)) {
                Alert alert = new Alert(AlertType.ERROR);
                alert.setTitle("Game over");
                alert.setHeaderText("The game is over");
                alert.showAndWait();
                App.board.gameOver = true;
            }
            if (App.board.hasWon(PieceValue.WHITE)) {
                Alert alert = new Alert(AlertType.ERROR);
                alert.setTitle("Game over");
                alert.setHeaderText("The game is over");
                alert.showAndWait();
                App.board.gameOver = true;
            }
            turn = 0;
        }
    }
    /** Get piece at X, Y
     * @return pieice
     *  **/
    public Piece getPiece(int x, int y) {
        for (Piece piece : pieces) {
            if (piece.getX() == x && piece.getY() == y) {
                return piece;
            }
        }
        return null;
    }
    /** Determine if there is a winner
     * @return true or false
     *  **/
    public boolean hasWon(PieceValue value) {
        for (Piece piece : App.board.pieces) {
            if (piece.value == value) {
                return false;
            }
        }
        return true;
           }
       }
```

Minmax.java

```
package draughts;

import java.util.ArrayList;

public class Minmax {
    ArrayList<Moves> successorEvaluations;
    int difficulty;

    /** Start minmax evaluation **/
    public void start() {
        startEvaluation();
        Moves move = getBestMove();
        makeMove(move);
        App.board.switchTurn();
    }
```

```
16.     /** Evaluate the returned moves **/
17.     private Moves getBestMove() {
18.         int max = -1000;
19.         int best = -1;
20.         for (int i = 0; i < this.successorEvaluations.size(); i++) {
21.             if (max < ((Moves) this.successorEvaluations.get(i)).score) {
22.                 max = ((Moves) this.successorEvaluations.get(i)).score;
23.                 best = i;
24.             }
25.         }
26.         return this.successorEvaluations.get(best);
27.     }
28.     /** Start evaluation **/
29.     public void startEvaluation() {
30.         this.successorEvaluations = new ArrayList<>();
31.         minmax(0, 1, -1000, 1000);
32.     }
33.     /** Perform the move **/
34.     public void makeMove(Moves m) {
35.         Piece piece = App.board.getPiece(m.originalx, m.originaly);
36.         if (piece.isKing) {
37.             m.wasking = true;
38.         }
39.         if (m.killedPiece != null && m.killedPiece.isKing == true) {
40.             m.piece.isKing = true;
41.         }
42.         piece.setX(m.newx);
43.         piece.setY(m.newy);
44.         if (m.killedPiece != null) {
45.             App.board.pieces.remove(m.killedPiece);
46.         }
47.
48.     }
49.     /** Get moves for the piece **/
50.     public ArrayList<Moves> getMoves(PieceValue value) {
51.         ArrayList<Moves> moves = new ArrayList<>();
52.         App.board.forcedMoves.clear();
53.         App.board.hasToTake(value);
54.         if (App.board.forcedMoves.isEmpty()) {
55.             for (Piece piece : App.board.pieces) {
56.                 if (piece.getValue() == value) {
57.                     piece.getMoves();
58.                     for (Moves m : piece.moves) {
59.                         moves.add(m);
60.                     }
61.                 }
62.             }
63.         } else {
64.             moves.addAll(App.board.forcedMoves);
65.         }
66.         return moves;
67.     }
68.     /** Minmax with Alpha-Beta pruning **/
69.     public int minmax(int depth, int player, int alpha, int beta) {
70.         int bestScore = 0;
71.         if (player == 0) {
72.             bestScore = -26;
73.         } else if (player == 1) {
74.             bestScore = 26;
75.         }
76.         if (App.board.hasWon(PieceValue.RED)) {
77.             return bestScore = 26;
78.         }
79.         if (App.board.hasWon(PieceValue.WHITE)) {
80.             return bestScore = -26;
81.         }
```

```
82.         if (depth == App.board.difficulty) {
83.             int redScore = 0;
84.             int whiteScore = 0;
85.             int kingRedScore = 0;
86.             int kingWhiteScore = 0;
87.             for (Piece p : App.board.pieces) {
88.                 if (p.getValue() == PieceValue.RED) {
89.                     redScore++;
90.                     if (p.isKing) {
91.                         kingRedScore++;
92.                     }
93.                 } else {
94.                     whiteScore++;
95.                     if (p.isKing) {
96.                         kingWhiteScore++;
97.                     }
98.                 }
99.             }
100.                    redScore = redScore + kingRedScore;
101.                    whiteScore = whiteScore + kingWhiteScore;
102.                    return redScore - whiteScore;
103.                }
104.                ArrayList<Moves> positions;
105.                if (player == 1) {
106.                    positions = getMoves(PieceValue.RED);
107.
108.                } else {
109.                    positions = getMoves(PieceValue.WHITE);
110.                }
111.                for (Moves m : positions) {
112.                    if (player == 1) {
113.                        makeMove(m);
114.                        int currentScore = minmax(depth + 1, 0, alpha, beta);
115.                        if (currentScore > alpha)
116.                            alpha = currentScore;
117.                        if (currentScore > bestScore)
118.                            bestScore = currentScore;
119.                        if (depth == 0) {
120.                            m.score = currentScore;
121.                            this.successorEvaluations.add(m);
122.                        }
123.                    } else if (player == 0) {
124.                        makeMove(m);
125.                        int currentScore = minmax(depth + 1, 0, alpha, beta);
126.                        if (currentScore < bestScore)
127.                            bestScore = currentScore;
128.                        if (currentScore < beta)
129.                            beta = currentScore;
130.                    }
131.                    Piece piece = App.board.getPiece(m.newx, m.newy);
132.                    piece.setX(m.originalx);
133.                    piece.setY(m.originaly);
134.                    if (m.killedPiece != null) {
135.                        App.board.pieces.add(m.killedPiece);
136.                    }
137.                    if (m.wasking == false) {
138.                        piece.isKing = false;
139.                    }
140.                }
141.                return bestScore;
142.            }
143.
144.        }
```

Moves.java

```java
package draughts;

public class Moves {
    /* Piece being moved */
    Piece piece;
    /* Piece killed */
    Piece killedPiece;
    /* Original position */
    int originalx;
    int originaly;
    /* New position */
    int newx;
    int newy;
    /* Move score */
    int score;
    /* Was piece a king */
    boolean wasking = false;
}
```

Piece.java

```java
package draughts;

import java.util.ArrayList;

import javafx.scene.image.ImageView;
import javafx.scene.layout.Pane;

public class Piece extends Pane {
    ArrayList<Moves> moves = new ArrayList<Moves>();
    PieceValue value;
    int x;
    int y;
    boolean isKing;

    /** Constructor for the piece object, requires value of piece and location
     * @param Value of the piece
     * @param X location
     * @param Y location
     * **/
    public Piece(PieceValue value, int x, int y) {
        this.x = x;
        this.y = y;
        this.value = value;
    }
    /** Returns piece X coordinate
     * @return X coordinate
     * **/
    public int getX() {
        return x;
    }
    /** Set piece X coordinate **/
    public void setX(int x) {
        this.x = x;
    }
    /** Returns piece Y coordinate
     * @return Y coordinate
     * **/
    public int getY() {
        return y;
    }
    /** set piece Y coordinate
     * **/
    public void setY(int y) {
        this.y = y;
```

```
45.         if (y == 0 && value == PieceValue.WHITE) {
46.             this.isKing = true;
47.         }
48.         if (y == 7 && value == PieceValue.RED) {
49.             this.isKing = true;
50.         }
51.     }
52.     /** Obtains the appppropriate draught image
53.      * @return Image
54.      * **/
55.     @SuppressWarnings("exports")
56.     public ImageView getImage() {
57.         ImageView image = new ImageView();
58.         if (value == PieceValue.RED) {
59.             if (isKing == true) {
60.
61.                 image = new ImageView("blackKing.png");
62.
63.             } else
64.                 image = new ImageView("red.png");
65.         }
66.         if (value == PieceValue.WHITE) {
67.             if (isKing == true) {
68.                 image = new ImageView("whiteKing.png");
69.             } else
70.                 image = new ImageView("white.png");
71.         }
72.         return image;
73.     }
74.     /** Get the value of the piece
75.      * @return Piece value
76.      */
77.     public PieceValue getValue() {
78.         return value;
79.     }
80.     /** Set the value of the piece */
81.     public void setValue(PieceValue value) {
82.         this.value = value;
83.     }
84.     /** Get the moves of a piece
85.      * @return The moves
86.      * */
87.     public ArrayList<Moves> getMoves() {
88.         this.moves.clear();
89.         switch (value) {
90.         case RED:
91.             if (isKing == false) {
92.                 moves.addAll(getMovesDown());
93.                 moves.addAll(getJumpsDown());
94.             } else {
95.                 moves.addAll(getMovesUp());
96.                 moves.addAll(getJumpsUp());
97.                 moves.addAll(getMovesDown());
98.                 moves.addAll(getJumpsDown());
99.             }
100.                 break;
101.             case WHITE:
102.                 if (isKing == false) {
103.                     moves.addAll(getMovesUp());
104.                     moves.addAll(getJumpsUp());
105.                 } else {
106.                     moves.addAll(getMovesUp());
107.                     moves.addAll(getJumpsUp());
108.                     moves.addAll(getMovesDown());
109.                     moves.addAll(getJumpsDown());
110.                 }
```

```
111.                     break;
112.                 }
113.                 return moves;
114.             }
115.
116.             /** Get the moves of a white piece
117.              * @return The moves
118.              * */
119.             public ArrayList<Moves> getMovesUp() {
120.                 ArrayList<Moves> possibleMoves = new ArrayList<Moves>();
121.                 /* Left side check */
122.                 if (App.board.getPiece(x - 1, y - 1) == null && isLegal(x - 1, y - 1
    )) {
123.                     Moves m = new Moves();
124.                     m.newx = x - 1;
125.                     m.newy = y - 1;
126.                     m.originalx = x;
127.                     m.originaly = y;
128.                     m.piece = this;
129.                     possibleMoves.add(m);
130.
131.                 }
132.                 /* Right side check */
133.                 if (App.board.getPiece(x + 1, y - 1) == null && isLegal(x + 1, y - 1
    )) {
134.                     Moves m = new Moves();
135.                     m.newx = x + 1;
136.                     m.newy = y - 1;
137.                     m.originalx = x;
138.                     m.originaly = y;
139.                     m.piece = this;
140.                     possibleMoves.add(m);
141.                 }
142.                 return possibleMoves;
143.             }
144.             /** Get the jumps of a white piece
145.              * @return The moves
146.              * */
147.             public ArrayList<Moves> getJumpsUp() {
148.                 /* Left side jump check */
149.                 ArrayList<Moves> possibleMoves = new ArrayList<Moves>();
150.                 if (App.board.getPiece(x - 1, y - 1) != null) {
151.                     if (App.board.getPiece(x - 1, y - 1).getValue() != value && App.
    board.getPiece(x - 2, y - 2) == null
152.                             && isLegal(x - 2, y - 2)) {
153.                         Moves m = new Moves();
154.                         m.newx = x - 2;
155.                         m.newy = y - 2;
156.                         m.originalx = x;
157.                         m.originaly = y;
158.                         m.piece = this;
159.                         m.killedPiece = App.board.getPiece(x - 1, y - 1);
160.                         possibleMoves.add(m);
161.                     }
162.                 }
163.                 /* Right side jump check */
164.                 if (App.board.getPiece(x + 1, y - 1) != null) {
165.                     if (App.board.getPiece(x + 1, y - 1).getValue() != value && App.
    board.getPiece(x + 2, y - 2) == null
166.                             && isLegal(x + 2, y - 2)) {
167.                         Moves m = new Moves();
168.                         m.newx = x + 2;
169.                         m.newy = y - 2;
170.                         m.originalx = x;
171.                         m.originaly = y;
172.                         m.piece = this;
```

```
173.                        m.killedPiece = App.board.getPiece(x + 1, y - 1);
174.                        possibleMoves.add(m);
175.                    }
176.                }
177.                return possibleMoves;
178.            }
179.            /** Get the moves of a black piece
180.             * @return The moves
181.             * */
182.            public ArrayList<Moves> getMovesDown() {
183.                ArrayList<Moves> possibleMoves = new ArrayList<Moves>();
184.                /* Left side check */
185.                if (App.board.getPiece(x - 1, y + 1) == null && isLegal(x - 1, y + 1
    )) {
186.                    Moves m = new Moves();
187.                    m.newx = x - 1;
188.                    m.newy = y + 1;
189.                    m.originalx = x;
190.                    m.originaly = y;
191.                    m.piece = this;
192.                    possibleMoves.add(m);
193.                }
194.                /* Right side check */
195.                if (App.board.getPiece(x + 1, y + 1) == null && isLegal(x + 1, y + 1
    )) {
196.                    Moves m = new Moves();
197.                    m.newx = x + 1;
198.                    m.newy = y + 1;
199.                    m.originalx = x;
200.                    m.originaly = y;
201.                    m.piece = this;
202.                    possibleMoves.add(m);
203.                }
204.                return possibleMoves;
205.            }
206.            /** Get the jumps of a black piece
207.             * @return The moves
208.             * */
209.            public ArrayList<Moves> getJumpsDown() {
210.                /* Left side jump check */
211.                ArrayList<Moves> possibleMoves = new ArrayList<Moves>();
212.                if (isLegal(x - 1, y + 1) && App.board.getPiece(x - 1, y + 1) != nul
    l) {
213.                    if (App.board.getPiece(x - 1, y + 1).getValue() != value && App.
    board.getPiece(x - 2, y + 2) == null
214.                            && isLegal(x - 2, y + 2)) {
215.                        Moves m = new Moves();
216.                        m.newx = x - 2;
217.                        m.newy = y + 2;
218.                        m.killedPiece = App.board.getPiece(x - 1, y + 1);
219.                        m.originalx = x;
220.                        m.originaly = y;
221.                        m.piece = this;
222.                        possibleMoves.add(m);
223.                    }
224.                }
225.                /* Right side jump check */
226.                if (isLegal(x + 1, y + 1) && App.board.getPiece(x + 1, y + 1) != nul
    l) {
227.                    if (App.board.getPiece(x + 1, y + 1).getValue() != value && App.
    board.getPiece(x + 2, y + 2) == null
228.                            && isLegal(x + 2, y + 2)) {
229.
230.                        Moves m = new Moves();
231.                        m.newx = x + 2;
232.                        m.newy = y + 2;
```

```
                        m.killedPiece = App.board.getPiece(x + 1, y + 1);
                        m.originalx = x;
                        m.originaly = y;
                        m.piece = this;
                        possibleMoves.add(m);
                    }
                }
                return possibleMoves;
            }
            /** Check if a coordinate is on the board
             * @return True or false
             * */
            public boolean isLegal(int x, int y) {
                if (x >= 0 && x <= 7 && y >= 0 && y <= 7) {
                    return true;
                }
                return false;
            }

        }
```

PieceValue.java

```
package draughts;

public enum PieceValue {
    RED, WHITE
}
```