Project Report

Introduction

This is a Mini-Checker Game for a human to play against a computer. Details are listed as following:

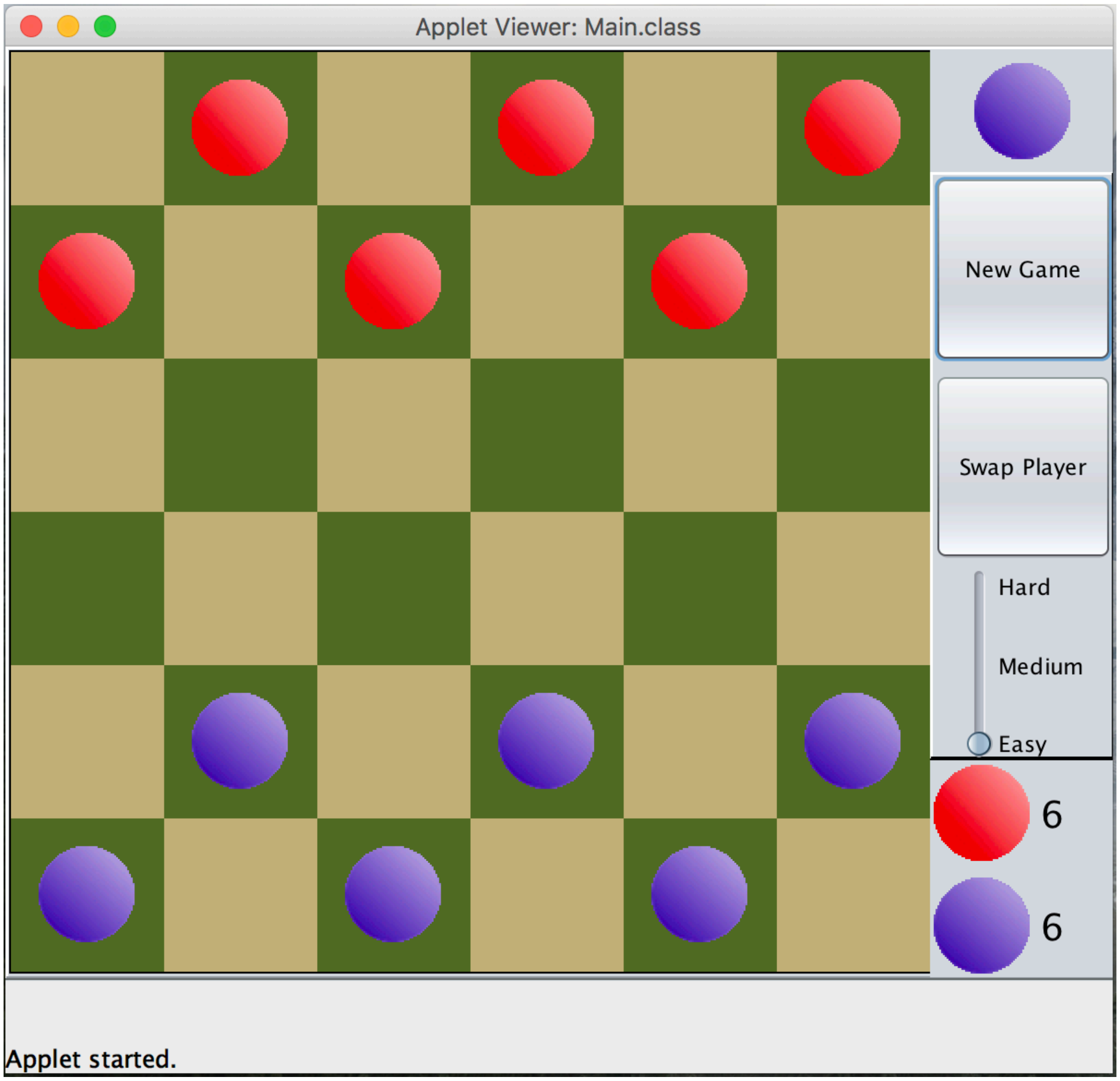


Figure 1

- The checker board consists of 6 x 6 alternating green and yellow squares as shown in **Figure 1**.
- Each player starts with six playing pieces, with the red pieces for the computer and the blue pieces for the human, arranged as shown in **Figure 1**.
- At the start of the game, the human player makes the first move by default. However, users can make the computer to go first by clicking the "Swap Player" button as shown in **Figure 1**.

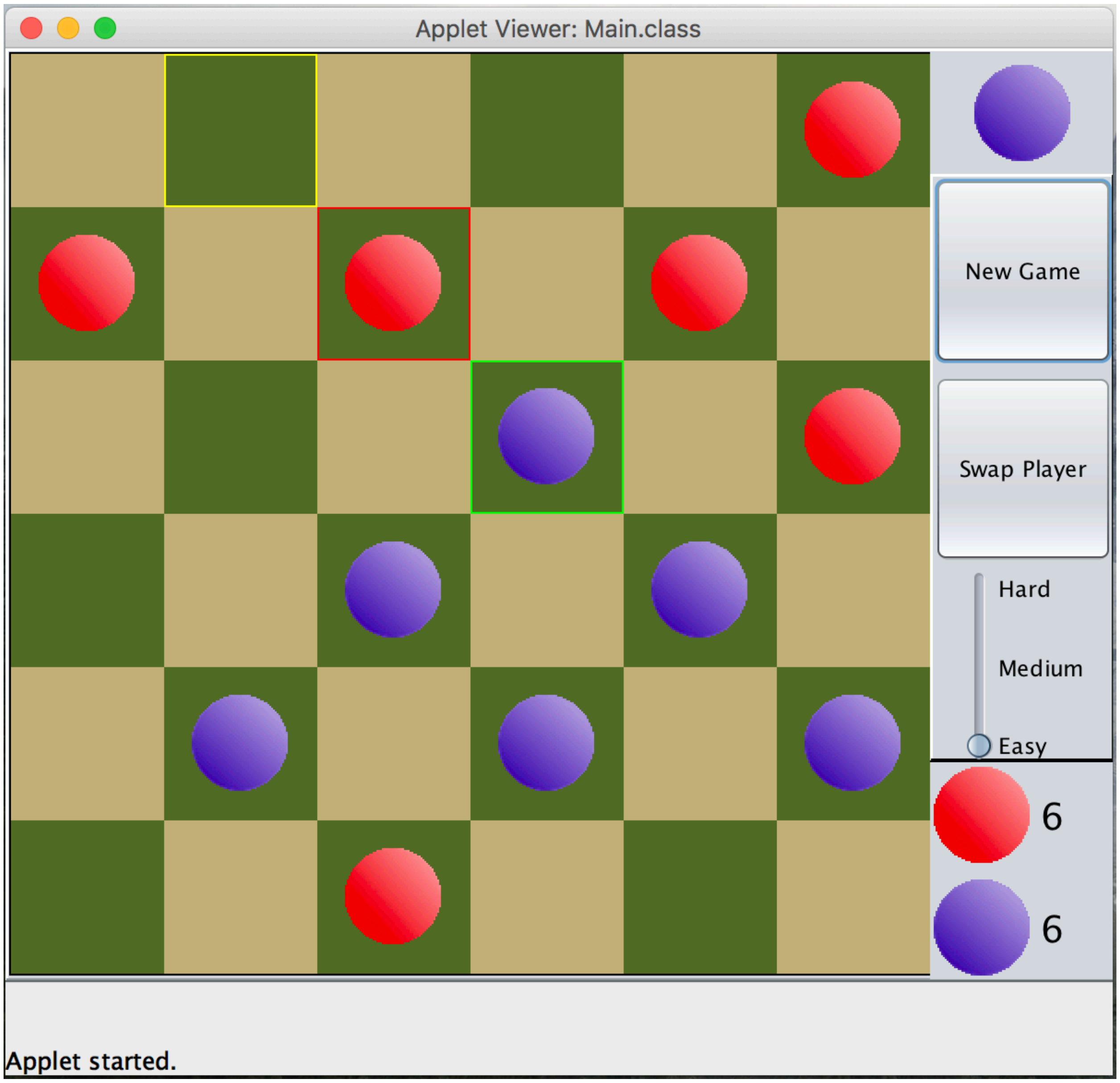
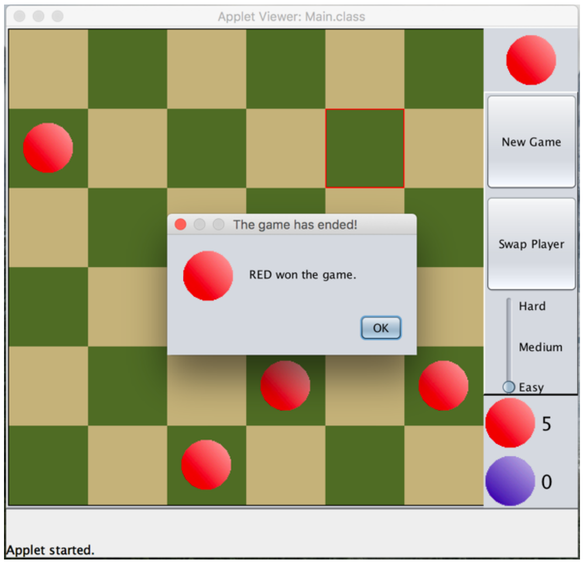


Figure 2

- Each player takes turn to make a move. Moves consists of regular move and capture move.
  - For regular move, a piece can move forward diagonally to an empty adjacent square.
  - For capture move, a piece can jump over and capture an opponent's piece and land on an empty square. However, consecutive jumps are forbidden. In addition, every opportunity to jump must be taken, as shown in **Figure 2** where the blue piece has to jump over the red piece. If there are more than one possible capture moves, the player can choose which one to take.
  - No vertical, horizontal or backward moves are allowed.
- If a player has no legal move left, that turn will be forfeited and the other player will continue.



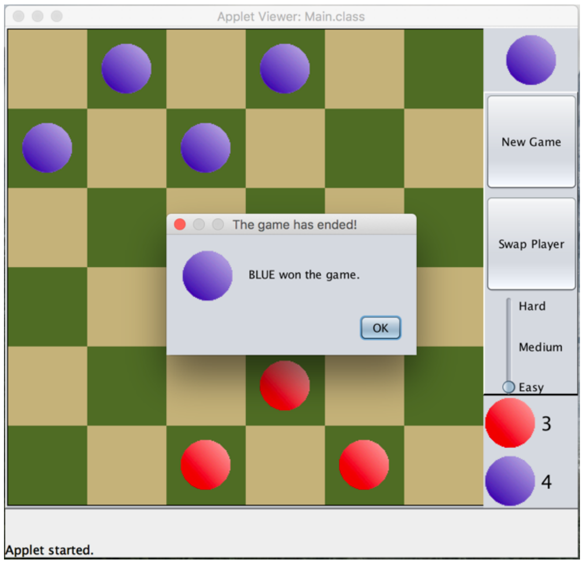
The game has ended!

RED won the game.

OK

Applet started.

- A player wins when it captures all of the other player's pieces as shown the left.



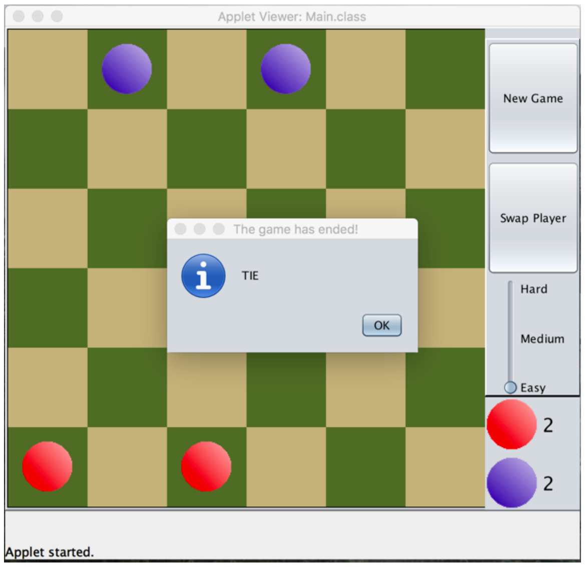
The game has ended!

BLUE won the game.

OK

Applet started.

- If both players do not have any legal move to take, the game will end and the player with the most number of pieces left wins as shown in the left.



The game has ended!

TIE

OK

Applet started.

- If the two players have the same number of pieces left, the game is a draw as shown in the left.

Instruction

In order to compile the program, users can simply follow steps as listed below:

1) Import the entire Java project into Eclipse (or other IDEs)
2) Set up JRE System Library as Java-SE 1.8
3) Select package Mini-Checkers
4) Select Main class and right click to Run it As Java Applet

In order to run the program, there are few tips as listed below:

1) Once running the Main class as Java Applet, a user interface will pop up
2) Users should be able to start playing by dragging pieces on the board
3) Users can click “Swap Player” button to make the computer go first
4) Users can click “New Game” button to start a new game
5) Users can change difficulty setting by dragging the slider on the right

Design

This program is designed based on Minimax Algorithm, which uses evaluation function computing values of all valid moves in order to determine the best move among them. Due to the requirements of decision making within 15 seconds, depth-limit is introduced into the program. In other words, the program will search the generated game tree for certain levels and acquire values of current game state, then the program will pick the best values among them and execute the move accordingly. Also, the Alpha-Beta Pruning is also designed as part of the algorithm to achieve better search efficiency.

There are 5 classes in total in my program, their brief introductions are as following:

Main	GUI including action listeners, game initialization
Game	game rules implementation, utilize the class of Piece, Square and Board
Board	game Board, consists of objects of Square that can contain objects of Piece
Square	game Square, can contain objects of Piece
Piece	game Piece

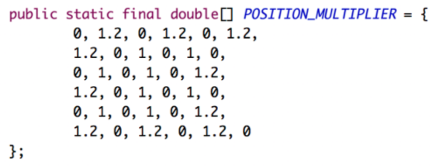
Alpha-Beta Search

Terminal State	Utility Value
No red piece left (AI wins)	-1000
No blue piece left (Human wins)	1000
Same number of red/blue pieces left (Tie)	0

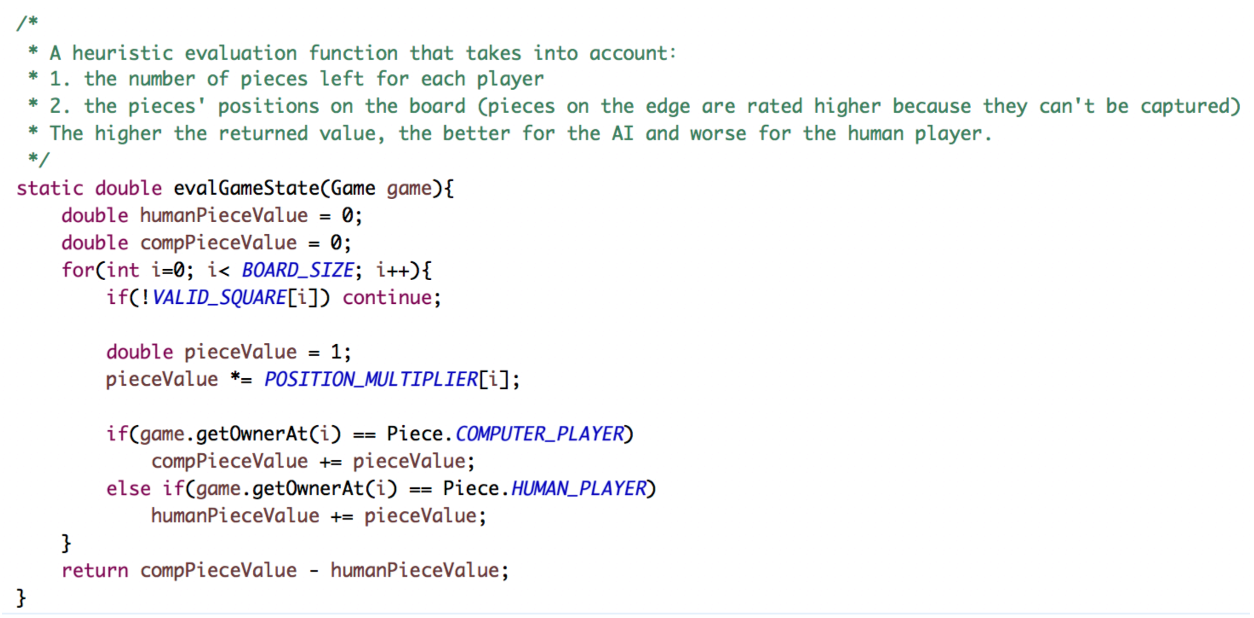
Evaluation function:

As shown in **Figure 3**, the evaluation function that I use in this program takes two parts into account.

- The number of pieces left for each player
- The pieces' positions on the board (pieces on the edge are rated higher because they can't be captured as shown on the right)



```
public static final double[] POSITION_MULTIPLIER = {
        0, 1.2, 0, 1.2, 0, 1.2,
        1.2, 0, 1, 0, 1, 0,
        0, 1, 0, 1, 0, 1.2,
        1.2, 0, 1, 0, 1, 0,
        0, 1, 0, 1, 0, 1.2,
        1.2, 0, 1.2, 0, 1.2, 0
};
```



```
/*
 * A heuristic evaluation function that takes into account:
 * 1. the number of pieces left for each player
 * 2. the pieces' positions on the board (pieces on the edge are rated higher because they can't be captured)
 * The higher the returned value, the better for the AI and worse for the human player.
 */
static double evalGameState(Game game){
    double humanPieceValue = 0;
    double compPieceValue = 0;
    for(int i=0; i< BOARD_SIZE; i++){
        if(!VALID_SQUARE[i]) continue;

        double pieceValue = 1;
        pieceValue *= POSITION_MULTIPLIER[i];

        if(game.getOwnerAt(i) == Piece.COMPUTER_PLAYER)
            compPieceValue += pieceValue;
        else if(game.getOwnerAt(i) == Piece.HUMAN_PLAYER)
            humanPieceValue += pieceValue;
    }
    return compPieceValue - humanPieceValue;
}
```

Figure 3

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

Difficulty Levels Implementation

There are three different levels designed for difficulty aspect of this game. They are implemented by assigning different depth-limits of Alpha-Beta Search. Because AI would obtain more information (a.k.a., become smarter) with deeper Alpha-Beta Search. Hence the intelligence of AI can be limited by setting depth-limits of Alpha-Beta Search. Furthermore, different settings of depth-limit could differentiate AI with different levels of intelligence, which achieve the goal of implementing difficulty levels as well.

The depth-limit settings in my program is list as following:

	EASY_AI	MEDIUM_AI	HARD_AI
Depth-limit	4	5	6