A Chess Application

Report on the final project of COSC 3P71

Torben Krüger and Maike Rees

December 2016

1 Problem Definition

The task for our final project was to code a chess application. This application should be playable by one player who plays against an artificial intelligence. For the artificial intelligence we have to use a tree-search sheme with alpha-beta-pruning. We are using the minmax-strategy.

2 Development

We developed this application using Java for the Logic and JavaFX for the GUI. Our IDE is Eclipse Mars on Windows 7.

We found inspiring articles about evaluating functions in chess on this Website **chessprogramming - evaluation** and the evaluation function that our AI uses ist mostly oriented from here **chessprogramming - Simplified evaluation function**. We chose this approach because we wanted to keep it clear and simple at first but with enough possibilities to develop a more specific evaluation function later on.

In the application there are two play modes, one with a GUI and one with a TUI which has an ASCI representation of the board and lets you make moves by typing in the coordinates. Please consider that the GUI has more options like restart, undo and close.

3 Design Choices

This section explains the design choices we made before and while programming the chess application to give you a better understanding how the application works inside.

3.1 Basics

We chose an object-orientated approach with nearly no complexe datastructures but some enums and abstract classes. For a better overview our project is grouped in different packages.

3.2 Player

The Players are represented by a single enum that contains black and white and a simple *toString*-method.

3.3 The Game

The Game class represents - of course - a Game. The Game contains the current board and a stack of previous boards that gives us the chance to implement an undo function where the player can undo his/her previous moves.

This is one of the reasons why we chose to have a Game class and a Board class. Another one is that here the Board does not have to bother with coordinates that are out of range because the Game will catch them before handing over coordinates to the Board. For every move that is made a new Board is created.

The GUI knows the Game which manages the current Board etc for it.

3.4 Board Representation

Each board has a couple of attributes:

- The current player
- The state of the board (check / checkmate / stalemate / draw / ...)
- The previous board (this is used for the search that is discussed later on)
- Markers for both players that are used for handling *en passant* moves

- The values of both players for the current board, calculated by the evaluation function

The chess board itself is represented by a two-dimensional 8 x 8 Figure array which contains the chess figures on their associated squares or **null** if no piece is currently sitting on the square.

The board can set, remove and find a figure on the itself. After every move it evaluates if a *Promotion* can take place and executes it if so. Additionally, the board can calculate its own value.

3.4.1 Pieces Representation

The Pieces are represented by the abstract class **Figures**. Every Figure knows its owner, board, coordinates and implements the abstract **clone()** and **isSquareReachable()** methods. The second method implements the logic what a legal move for the specific piece is. It knows its material value and its position matrix. Some Figures additional have some special parameters that will be discussed.

The King piece has an attribute called **hasBeenMoved** and overrides the **move** method. Both are needed for the *castling* move. The King's evaluation matrix changes during the course of the game. This means that for the evaluation function a matrix exists for the beginning and middle part of the game as well as another for the end part of the game. Hence the king provides methods to deal with this.

The Rook also provides an attribute called **hasBeenMoved** to deal with the *casteling* move.

The Pawn has to additional attributes (**enPassant, startMove**) to be able to handle the *en passant* move. For the same purpose, it also overrides the **move** method.

3.5 Evaluation Function

The evaluation function that we use is pretty simple but sufficient and fast. It is composed of two parts.

Every type of figure has an assigned value. These values are chosen due to chess strategies that can be read in the linked website above. The unit of those values is called centipawn ($\frac{1}{100}$Pawn).

- Pawn = 100
- Knight = 320
- Bishop = 330
- Rook = 500
- Queen= 900
- King = 20000

The second part is the evaluation matrix. Each piece has its own evaluation matrix that gives penalties for 'bad' squares on the board and rewards for 'good' squares on the board.
An example: The Evaluation Matrix of the Pawn:

```
1  private final static int[][] EVAL = {
2          {  0,   5,   5,   0,   5,  10,  50,   0 },
3          {  0,  10,  -5,   0,   5,  10,  50,   0 },
4          {  0,  10, -10,   0,  10,  20,  50,   0 },
5          {  0, -20,   0,  20,  25,  30,  50,   0 },
6          {  0, -20,   0,  20,  25,  30,  50,   0 },
7          {  0,  10, -10,   0,  10,  20,  50,   0 },
8          {  0,  10,  -5,   0,   5,  10,  50,   0 },
9          {  0,   5,   5,   0,   5,  10,  50,   0 }
10     };
```

Those two parts are calculated together like this:
Value of a piece = Material Value (Pawn = 100) + Position Value (EVAL[x][y])

In the Code (class Pawn):

```
1   @Override
2   public int getValue() {
3       int value = Pawn.VALUE;
4       switch (owner) {
5       case WHITE:
6           value += EVAL[x][y];
7           break;
8       case BLACK:
9           value += EVAL[x][7 - y];
10          break;
11      default:
12          throw new IllegalArgumentException();
13      }
14      return value;
15  }
```

So to get the final Evaluation function of a board-Object for e.g. the white Player you have to calculate: **Value of Board for white = Sum of all values of white pieces - Sum of all values of black pieces**

3.6 Search

Our search uses the MinMax - Strategy in a tree sheme with alpha-beta-pruning.

3.7 Embedding of the AI

3.8 The GUI

For the GUI we chose to use the JavaFX framework for the simple reason that we have already used it earlier in out studies and have some experience with it. Further more the grid layout (*GridPane*) that it provides is pretty straight forward for a chess board. In the GUI you can chose the intelligence of the AI by chosing the ply. Furthermore, you can make use of the buttons undo, close and restart (which generates a completely new Game-Object).