IT3105: Project 1 – Using Minimax with Alpha-Beta pruning to play Quarto

Due on Wednesday, September 18, 2013

Lecturer: Keith Downing

Pablo Liste Garcia & Dominik Horb

Contents

1 General Setup

The first decision our team made after we initially heard of the assignment task,was to use Java as our programming language. This decision was made mainly out of familiarity with the language and because we couldn't find any caveats that would complicate fulfilling the assignment.
We also chose to share our code through a Git repository, which you can access through:

- `https://github.com/dom2503/Quarto`

We also wanted to start out with a basic command line interface as it was shown during the lecture and create a graphical interface maybe later on, if there would be time left. You can see an example of the interface we created in Figure 1.

```
Please enter the x-coordinate for your move:
2
Please enter the y-coordinate for your move:
a

y\x    1      2      3      4
 A     ____   (r )   (R )   ____
 B     ____   ____   ____   ____
 C     ____   ____   ____   ____
 D     ____   ____   ____   ____


Player1 please select a piece to give to Player2.

These are the available pieces:
1.[R*]  2.(R*)  3.[R ]  4.[r*]  5.(r*)  6.[r ]  7.[B*]  8.(B*)  9.[B ]  10.(B )  11.[b*]  12.(b*)  13.[b ]  14.(b )
6
Player1 selected [r ]

Player2 please make your move.
y\x    1      2      3      4
 A     ____   (r )   (R )   ____
 B     ____   ____   ____   ____
 C     ____   ____   ____   ____
 D     ____   ____   ____   ____
I made my move to 1C
y\x    1      2      3      4
 A     ____   (r )   (R )   ____
 B     ____   ____   ____   ____
 C     [r ]   ____   ____   ____
 D     ____   ____   ____   ____


Player2 please select a piece to give to Player1.
Player2 selected [B ]

Player1 please make your move.
y\x    1      2      3      4
 A     ____   (r )   (R )   ____
 B     ____   ____   ____   ____
 C     [r ]   ____   ____   ____
 D     ____   ____   ____   ____
Please enter the x-coordinate for your move:
```

Figure 1: Basic interface of our Quarto game.

The general structure of our code is pretty simple. The main class of the application is *QuartoGame*, which contains the game loop and almost all setup and command line interface parts. The actual game is put together through the *Board* and *Piece* classes and some additional enums for the properties of the pieces.
All the different players that we were supposed to write just needed to be able to pick a next piece and to make a move with the piece that was given to them, as can be seen in

the shortened version of our *Player* interface below.

```
1 public interface Player {
2   public String makeMove();
3   public Piece selectPieceForOpponent();
4   public void setGivenPiece(Piece givenPiece);
5 }
```

Listing 1: Player interface

2 Evaluation Function

We first started out thinking about the actual evaluation function for the states of the Quarto game, after we finished the main game logic, the random and the human player. As the main focus was to get the Minimax algorithm working, we wrote a very basic dummy version that just returned a random double value in the beginning.

```
1 public double evaluateBoard(Board board){
2         return rand.nextDouble();
3 }
```

Listing 2: Dummy evaluation function

In the real version later on it should evaluate, whether the current state of the board is good for the last player that made a move though. A positive value meaning that this is the case and a negative one meaning that the opponent has an advantage.
The state of the game as we look at it at this point, is just the board and the pieces that were already placed.
After our implementation had developed into a more mature state we added the two distinctive finishing states of the Quarto game: a draw and the win of the player that set the last piece. As you can see below, we were returning positive infinity for a win, because it is the best outcome that can be reached and 0.0 for a draw, because no player has an advantage there.

```java
public double evaluateBoard(Board board){
    double result;
    if(board.gameWasWon()){
      result = Double.POSITIVE_INFINITY;
    } else if(board.isDraw()){
      result = 0.0;
    } else {
      result = rand.nextDouble();
    }

    return result;
}
```

Listing 3: Very basic evaluation function

Figuring out the next evolutionary step of the evaluation process proofed to be a bit more difficult though. What we somehow concluded was, that leaving a lot of nearly completed rows, that contained three pieces with one or two identical properties for the next player is somehow a bad decision.
Playing in a way that the other player doesn't have a choice but to leave open nearly finished lines however – preferably with only pieces left that would finish these – would be good decision making.
We therfore replaced the random value in the else block with the following code that simply counts the number of lines that fulfill these criterias:

```java
int nearlyFinishedLines = this.getNearlyFinishedLineCount(board);

// the value of 10.0 is arbitrary, it's just to denote that it's good
// that we don't leave nearly finished lines
if(nearlyFinishedLines == 0){
  return 10.0;
}
// nearly finished lines are bad, because, the next user could maybe finish t
// with the right piece
return -1.0 * nearlyFinishedLines;
```

Listing 4: Counting nearly finished lines.

After a while we noticed, that this also leads to problems, because the algorithm then overlooks obvious winning situations at a higher level in favor of winning situations in deeper levels in earlier moves.

By adding the current depth level as a multiplier to the method, we were able to remedy this problem, so that obvious winning situations aren't overlooked anymore.

```
int nearlyFinishedLines = this.getNearlyFinishedLineCount(board);

// the value of 10.0 is arbitrary, it's just to denote that it's good
// that we don't leave nearly finished lines
if(nearlyFinishedLines == 0){
  return depth * 10.0;
}
// nearly finished lines are bad, because, the next user could maybe finish t
// with the right piece
return -1.0 * nearlyFinishedLines * depth;
```

Listing 5: Using depth as a multiplier

3 Novice vs. Random

Runs	Novice	Random	Draws
20	19	1	0

4 Novice vs. Minimax-3

Runs	Novice	Minimax-3	Draws
20	3	17	0

5 Minimax-3 vs. Minimax-4

Runs	Minimax-3	Minimax-4	Draws
20	5	15	0

6 Tournament experiences

Runs	Our Minimax	Marc & Valerio	Draws
20	0	0	0

Runs	Our Minimax	Jan & Tomas	Draws
20	0	0	0

Together with the other groups we decided to use a client-server approach for making the tournament happen. As we were the first group to be ready with the basic application, we inherited the responsibility of being the server, which was surprisingly easy to do

within our environment and yielded the additional benefit, that we could decide about the used protocol.
For the implementation we just needed to extend the above mentioned Player interface a bit, so that it also could be used to send some additional messages and implement it in a *RemotePlayer* class:

```
1 public interface IRemotePlayer extends Player {
2   public void sendMove(int x, int y);
3   public void sendMessage(String message);
4   public String receiveMessage();
5   public void close();
6 }
```

Listing 6: Remote player interface

The only change to our game loop was then to check if the nextPlayer is a remote player, so that we would need to send some information about the made move:

```
1 Point move = this.currentPlayer.makeMove();
2 if(this.nextPlayer instanceof IRemotePlayer){
3   IRemotePlayer remote = (IRemotePlayer) this.nextPlayer;
4   remote.sendMove(move.x, move.y);
5 }
```

Listing 7: Sending move information.

All the other methods of the normal interface could basically be used, just instead of calculating the correct move within *makeMove* for example we just needed to read from the port that was opened on the creation of the object. This setup could also easily be used with two remote players, because every object uses a different port to communicate. The major problem was then just to get everyone together in order to conduct the real tournament and fix the unavoidable bugs. Our server part was just written with a very small dummy script after all.
Seeing how our implemented player performed against others was then really interestings.