Introduction to Artificial Intelligence
Quarto

Jérémie	Blanchard
Anthony	Deveaux
Josselin	Marnat
Clément	Schmit

2016-2017

Table of Contents

1 Introduction

2 Presentation of the game

The following description is adapted from the website http://www.ludoteka.com/quarto-en.html.

Origin: Quarto is a board game recently invented by Blaise Muller and published by Gigamic

Description: This is a game for two players. The board has 16 squares (4x4), and the 16 different pieces that can be constructed combinating the following four characteristics: `colour` (white/black), `size` (long/short), `shape` (round/square), and `hole` (hole/flat).

Objective: The aim of the game is to complete a line with four pieces that are similar at least about one of the four described characteristics (four big pieces, four little, four red, four blue, four circle, four square, four with hole or four without hole). The line may be vertical, horizontal or diagonal. The winner is the player who places the fourth piece of the line.

How the game goes on: Players move alternatively, placing one piece on the board; once inserted, pieces cannot be moved.
One of the more special characteristics of this game is that the choice of the piece to be placed on the board is not made by the same player who places it; it is the opponent who, after doing his move, decides which will be the next piece to place.
So, each turn consists of two actions: 1. Place on the board the piece given by the opponent. 2. Give to the opponent the piece to be placed in the next move.
In the first turn of the game, the player who starts has only to choose one piece for the opponent.

Final: The game finishes in a draw when nobody reaches the objective after placing the 16 pieces.

3 Algorithm of the game-play

If you look at the file named `quarto.pl`, you'll see the following predicate:
`play(Interface,Heuristics1,Heuristics2)`,
who are running the game.
But the actual predicate who does the job is `round(Interface,...)`, and here is how it works:

1. first, we check if the board is full → Game Over
2. then, we check if somebody wins → Game Over
3. if none of this first predicates come `true`, we launch a round, splitted in two parts:
   (a) first part: the player need to choose a piece:
      - we display the board, the player, the pieces *etc.* ;
      - we ask a piece to the player (human or computer) ;

   (b) we swap the players to do the second part of the round:
      - we display the piece to play;
      - we read the position where the player want to put the given piece;
      - then, we put the piece on the board (by getting a `NewBoard`);
      - finally, we run a new round with the baord obtained (return to 1.).

4 Heuristics

Random (`random.pl`): this heuristics is the 'stupid' one.
It selects a piece randomly among the available ones, and do the same thing for the position. This heuristics was created at the very beggining of the project, just to test the game-play, and to have a base to create the other ones.

The following descriptions of the heuristics have been written by each team member.

Anthony (`anthony.pl`): The AI prepare the Board for the first 5 turn, minimizing the number of alignment between each pieces in the board, and by giving pieces sharing the fewer attributes with the piece the AI receive.
Why the five first turn are preparing the board? Because when you mimizing the number of alignement the fifth turn can share attributes between 2 and more lines. Thanks to all pieces giving because you giving all pieces that sharing fewer attributes as possible and the opponement have to pose them choosing by align with other pieces and not. In wich case it can helps us to do more alignement.
For next turns, the AI became more agressive by giving non loosing pieces and maximizing the number of attributes share with the last piece receive by the AI. And choosing position where we can maximizing the number of share aligments with the maximum of share. If the Board is too bad the AI (loosing next turn), then the AI play in a position that can free some pieces to give and possibly can switch the Board into good board (winning next turn).

Josselin (`josselin.pl`):

Clément (`clement.pl`): A main predicate is used in this heuristic. This is the ListWin predicate it goes through all the board and look for winning pieces. These winning positions are put in a list (coordinates with winning caracteristics).
When you need to place the piece that was given to you, the read position predicate look for a position where the piece can win if placed there.
On the other way when you need to choose a piece for your opponnent you use again the ListWin but this time the piece you give must not have a common caracteristics with one of the winnings piece in the ListWin so that your opponent can't win with this piece on his turn.

Example :

1	0	0	10
0	2	11	0
0	0	3	0
12	0	0	0

With this board the winlist is:
`[[white, 4, 4], [short, 4, 4], [black, 3, 2], [short, 3, 2]]`
You can win by either placing a white piece in 4,4 or a short one in 4,4 *etc.*
In this board we cannot find a piece to win we have to loose considering there is a winnable position with a black piece and a white piece. Considering a piece is either white or black there is no solution. In this case we call a random piece because the previous predicate will fail.

Exemple:

1	0	0	10
0	2	11	0
0	0	0	0
12	0	0	0

With this board the winlist is `[[black, 3, 2], [short, 3, 2]]`
So after placing the piece we would choose one that doesn't allow our opponnent to win.
In this case we need a piece that isn't black or that isn't short.

Jérémie (`jeremie.pl`): To make this IA, I decided to create a Min Max algorithm which takes the best position option for the given piece, and gives the enemy a piece that will make him less likely to win. In order to do that I had to calculate the weight of every playable board to know which one I had to choose. To calculate the weight I simply took the board of which I wanted to know the weight,

and added up all the characteristics of all the pieces on the board one by one. Then I had to compare them with each other to take the minimal one. Once I had done that for one board, I had to do it for all the other boards that are linked to the original board, and put the minimal value of each one into a list, and finally I took the minimal value of this list. Then I picked the board that had this minimal value.
To know which piece I have to give my opponent, I used the same system as before, but this time I took the board with the highest weight.

5 User Interfaces

5.1 Inline mode

5.2 GUI: a XPCE adventure...

6 Efficiency and challenges

6.1 Stats foreach heuristics

#win	random	anthony	josselin	clement	jeremie
random	50	2	1	32	x
anthony	100	29	68	100	x
josselin	100	4	100	99	x
clement	58	29	25	55	x
jeremie	47	0	0	41	x

Table 1: each heuristics vs. random, over 100 iterations

TIME	random	anthony	josselin	clement	jeremie
random	0.4 s	4.7 s	33.3 s	0.8 s	x
anthony	4.6 s	43.4 s	49.1 s	4.5 s	x
josselin	39.6 s	48.8 s	89.4 s	39.7 s	x
clement	0.7 s	4.6 s	34.6 s	1.1 s	x
jeremie	87.5 s	75.6 s	109.7 s	77.7 s	x

Table 2: each heuristics vs. random, over 100 iterations

INFERENCES	random	anthony	josselin	clement	jeremie
random	2 M	36 M	304 M	5 M	x
anthony	34 M	398 M	445 M	39 M	x
josselin	341 M	430 M	827 M	342 M	x
clement	5 M	38 M	315 M	8 M	x
jeremie	149 M	171 M	419 M	149 M	x

Table 3: each heuristics vs. random, over 100 iterations

7 Conclusion

`quarto.pl`	Josselin	10-12 h
`inline_interface.pl`	Josselin	3-4 h
`gui.pl`	Josselin	5-7 h
`random.pl`	Josselin	5 m
`human.pl`	Josselin	15 m
`anthony.pl`	Anthony	25-30 h
`clement.pl`	Clément	12-15 h
`jeremie.pl`	Jérémie	30-35 h
`josselin.pl`	Josselin	8-10 h

Table 4: Approximate time for coding each files