ID2209 – Distributed Artificial Intelligence and Intelligent Agents

Assignment 2 – Negotiation and Communication (FIPA)

Group 7
Sofia Krylova
Iosif Koen
22 Nov. 2022

The first task in this assignment is the problem of N queen. What we try to do here is essentially create a QueenSizeXQueenSize size chessboard.

The second task, similar to assignments 1 and 2, this assignment is based on the festival setting with the addition of stages where each stage has its values, and each Guest has the utilities for each stage. Guests communicate with the stages through FIPA to know their individual values. For the challenge, we are required to develop a program so to make each Guest know the values of each stage and calculate his utility for each stage and pick each time stage with the highest utility.

How to run

Run GAMA 1.8 and import three models 1) TheQueenGame.gaml 2) Stages.gaml as new projects. Press the main to run the simulation. Note that when you press the main, the GAMA will pop you to the simulation, where we can see all our agents in their start state. So from now, there are two ways to run the simulation: one is just simply pushing the play button (not recommended), and the other is to push the step-by-step button. In this way, we can see each cycle and what is going on each, and we can examine more detailed the agents' behaviour.

Species

Within this assignment, we have implemented 2 main types of agents:

- Queen - a main actor of the simulation. Its goal is to be in a cell alone and not be another queen to its diagonals, rows and columns.
- Grid - Grid is an actor with is the ChessBoard, actually where the Queens move to and live.

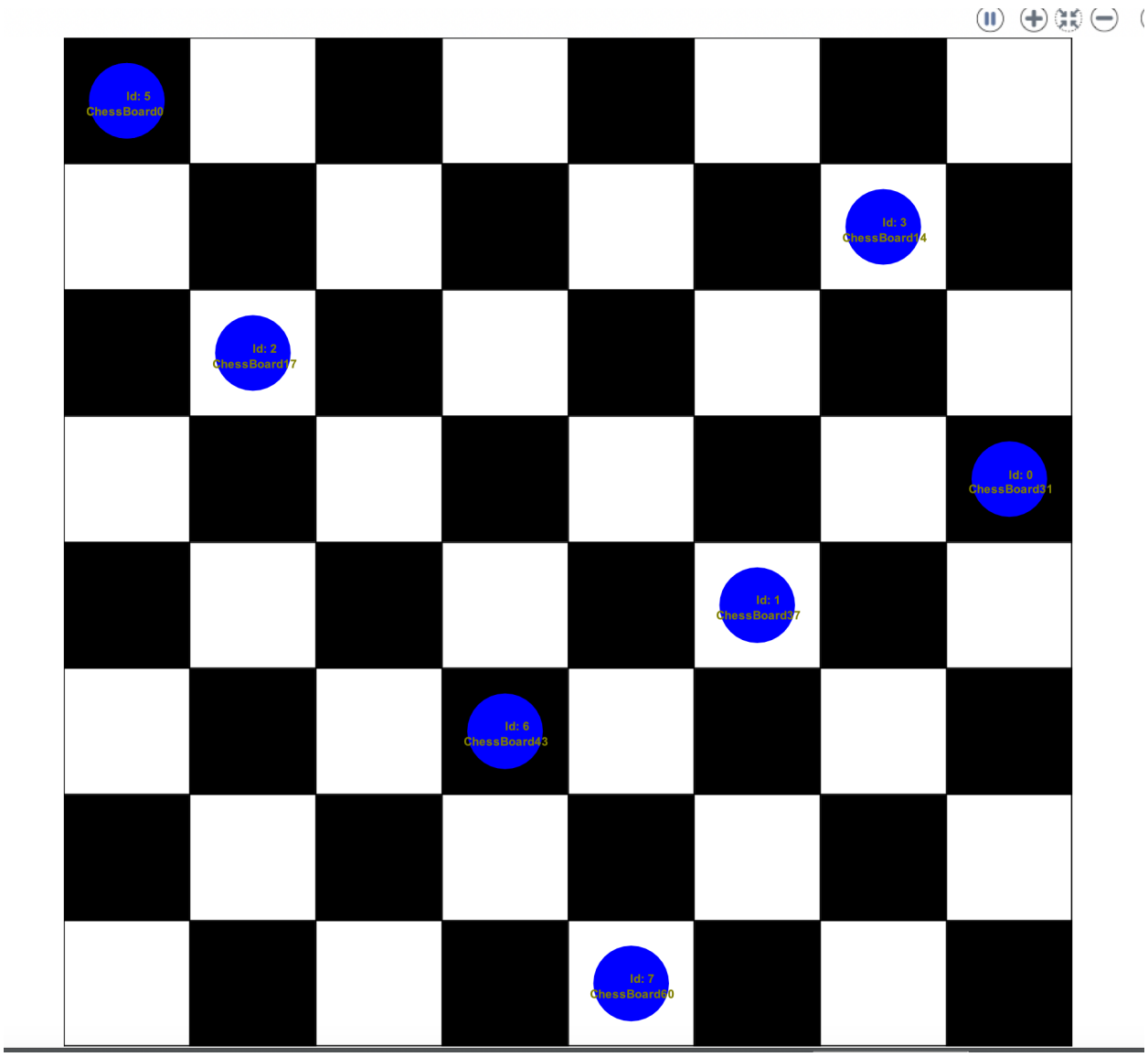
Implementation

The queens are pre-installed on the Chessboar in a diagonal schema and move based on the rules described (No two Queens share the same row, column and diagonal line). Each queen finds a position on the grid to which it moves. Then a call is made to the successor to find and move, and this carries on till the last queen finds its position. If no position is found, then it can go back to a former queen to find a possible solution, and so on. FIPA is used to communicate among the queens, and finally, all queens are placed in the QueenNumberXQueenNumber Chessboard formation iteratively
For the Grid, we have only its own declaration:

```
/*
 * We make our grid QueenNumber x QueenNumber
 * So our queens be exactely as the cells of our grid
 *
 * The neighbors is a build in variable witch returns the list
 * of cells at a distance of 1.
 *
 */
grid ChessBoard width: numberOfQueens height: numberOfQueens neighbors: QueensNeighbors{
    init{
        if(even(grid_x) and even(grid_y)){
            color <- #black;
        }
        else if (!even(grid_x) and !even(grid_y)){
            color <- #black;
        }
        else {
            color <- #white;
        }
    }


}
```

Picture 1 - Grid actor

For the Queen, we have some reflexes witch are those who are doing the whole job of communication and moving on ChessBoard:


Id: 5
ChessBoard0
Id: 3
ChessBoard14
Id: 2
ChessBoard17
Id: 0
ChessBoard31
Id: 1
ChessBoard37
Id: 6
ChessBoard43
Id: 7
ChessBoard60

```
//      We define predecessors and successors of Queens
        loop counter from: 0 to: numberOfQueens - 1 {
            Queen queen <- Queen[counter];
            queen <- queen.setId(counter);
            queen <- queen.initializeCell();
            write ("Init a queen" + queen);
            if counter > 0 {
                queen.predecessor <- Queen[counter - 1];
                write ("Init a queen predecessor " + Queen[counter - 1]);
                queen.predecessor.successor <- queen;
                write ("Init a queen successor " + queen);

            }


        }
        Queen[0].predecessor <- Queen[numberOfQueens-1];
        Queen[0].predecessor.successor <- Queen[0];
        write "OccupiedCells " + OccupiedCells;
//      add Queen[numberOfQueens - 1] to: positioned;
//      do PrintSuccessorsAndPredecessors;
        ask Queen[0] {
            do FindSafePosition;
        }
    }
```

Initialise the successors and predecessors

```
Added Queen0 to cell ChessBoard[0]
Init a queenQueen[0]
Added Queen1 to cell ChessBoard[0]
Init a queenQueen[1]
Init a queen predecessor Queen[0]
Init a queen successor Queen[1]
Added Queen2 to cell ChessBoard[0]
Init a queenQueen[2]
Init a queen predecessor Queen[1]
Init a queen successor Queen[2]
Added Queen3 to cell ChessBoard[0]
Init a queenQueen[3]
Init a queen predecessor Queen[2]
Init a queen successor Queen[3]
OccupiedCells []
```

Queen Initialaisation

```
    }

    reflex AcceptPositionMessage when: !empty(proposes){
        message position <-proposes at 0;
        if (reply) {
            write name + " Forward message to " + successor;
            helped <- false;
            do start_conversation to: list(successor) protocol: 'fipa-contract-net' performative: 'cfp' contents: ["Found position for you", position.contents[0]];
            reply <- false;
        } else {

            Positioned <- true;
            remove myCell from: OccupiedCells;
            myCell <- position.contents[0];
            add myCell to: OccupiedCells;
            location <- myCell.location;
            if !(positioned contains self) {
                    add self to: positioned;
                }
            write positioned;
            write name + " positioned by a propose";
            do start_conversation to: list(successor) protocol: 'fipa-contract-net' performative: 'cfp' contents: ["Find position for yourself"];

        }
    }
```

Accept Position message

```
/=============== Queen3 checking her position ===============/
Queen3 Checking row 0
ChessBoard0 does not fit in row
selfcheck [ChessBoard[1],ChessBoard[4],ChessBoard[5],ChessBoard[15],ChessBoard[3],ChessBoard[7]]
Queen3 Checking row 0
ChessBoard0 does not fit in row
Queen3 Checking row 1
Queen3 Checking column 0
ChessBoard0 does not fit in column
Queen3 Checking row 1
Queen3 Checking column 1
Queen3 Checking diagonals
D Init 1 1
ChessBoard0 does not fit in d1
Queen3 Checking row 3
Queen3 Checking column 3
ChessBoard11 does not fit in column
Queen3 Checking row 0
ChessBoard0 does not fit in row
Queen3 Checking row 1
Queen3 Checking column 3
ChessBoard11 does not fit in column
Queen3 Asking my predecessor to find a position
```

Queen check her position.

```
16
17      bool CheckRow(int RowNumber){
18          write name + " Checking row " + RowNumber;
19
20          loop counter from: 0 to: numberOfQueens - 1 {
21              if(OccupiedCells contains ChessBoard[counter, RowNumber] ){
22                  write ChessBoard[counter, RowNumber].name + " does not fit in row";
23                  return false;
24              }
25          }
26          return true;
27      }
28
29      bool CheckColumn(int ColumnNumber){
30          write name + " Checking column " + ColumnNumber;
31
32          loop counter from: 0 to: numberOfQueens - 1 {
33              if(OccupiedCells contains ChessBoard[ColumnNumber, counter] ){
34                  write ChessBoard[ColumnNumber, counter].name + " does not fit in column";
35                  return false;
36              }
37
38          }
39          return true;
40      }
```

Check Row and Column

Challenge 1

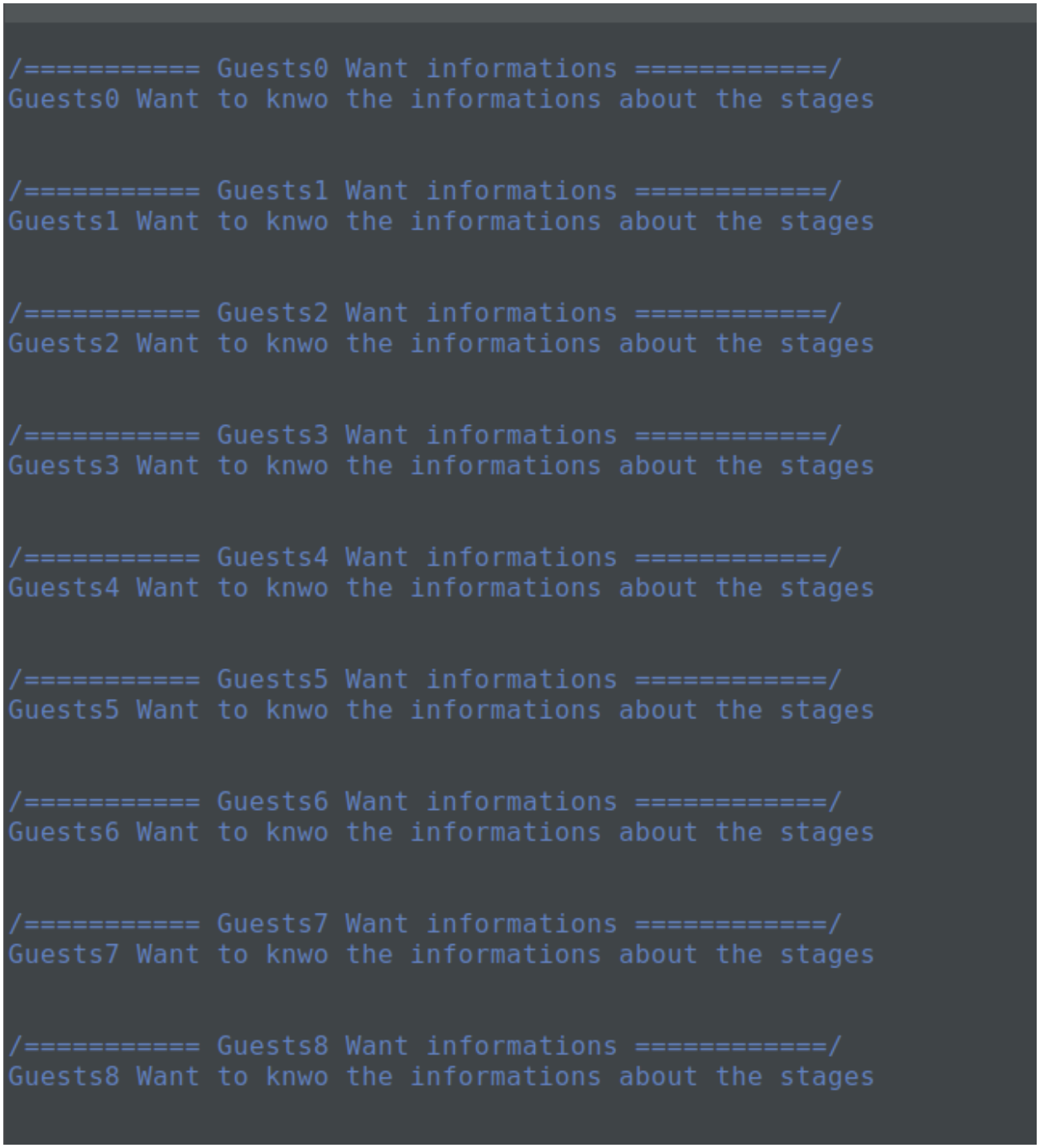
In this challenge, we were asked to develop and introduce a stage festival where the guests can visit stages depending on their preferences. We make the preferences depending on the utilities of each stage and their values of them. The values are random number numbers (like a ranking) of how good a stage is or not from 1 to 20.

We used fipa to make all our actors to communicate with each other and exchange values, asking for information (values). We followed the instruction, and we didn't use music type or bands for guest preferences instead, we used a formula in the reflex of FindMostLikedStage:

```
reflex FindMostLikedStage when: StageValuesChanged {
    write ("\n============ Find the Best Stage for the Guest " + self.name + " ====================/");
    StageValuesChanged <- false;
    UtilityForEachStage <- [0, 0, 0, 0];

    loop stage from: 0 to: length(Stages) - 1 {
        loop value from: 0 to: length(StagesValues) - 1{
            list<int> IndexStageValues <- StagesValues[stage];
            UtilityForEachStage[stage] <- UtilityForEachStage[stage] + IndexStageValues[value] * self.GuestValues[value];
        }
    }
    write ("Utility For Each Stage: " + UtilityForEachStage);
```

The result of this is that each time 30 mod 20 = 0, the preferences of the guests and the values of each stage will gonna change, and guests will ask for a new stage to go.

We implemented 4 stages, each one to the corners of our playground. And an N number of guests (we have 30) each can have as many as its wants.



Picture 2 - In this picture, we can see Guests who send information request to stages

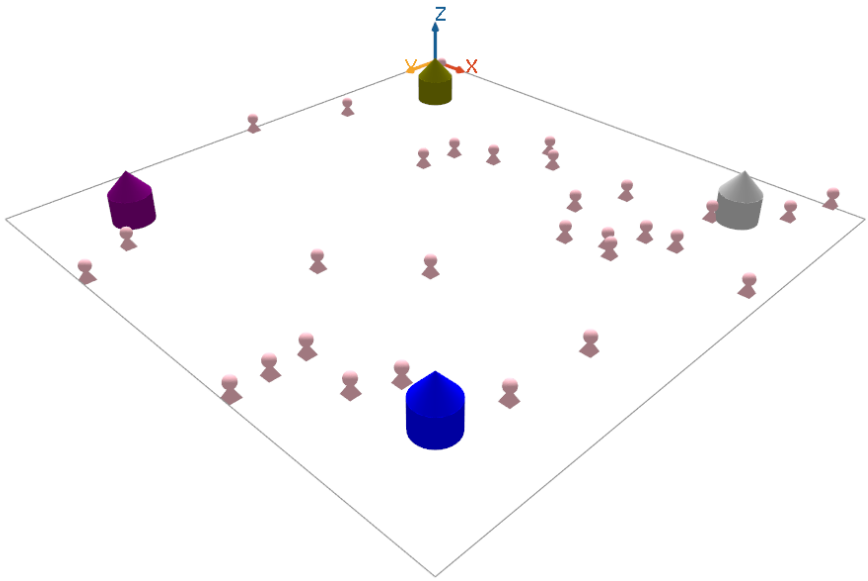
Picture 3 - In this picture, we can see that Stage0 reply to all Guests who ask for its values

```
/============= Guest Guests0 receive Information message ==================/
I Guests0 received new values for Stages[0]

I Guests0 received new values for Stages[1]

I Guests0 received new values for Stages[2]

I Guests0 received new values for Stages[3]
```

Picture 4 - In this picture, we can see that the Guest0 received the values for all stages (from them)

```
============ Find the Best Stage for the Guest Guests0 ====================/
Utility For Each Stage: [481,115,361,256]
The max unitility from all stagies is: 481
```

Picture 4 - In this picture we can see that the Guest0 found the best stage with the highest utility



Discussion / Conclusion

This assignment was very interesting, and both of us learned a lot of things about agent communications. It made us read and deeply understand the theory part for this assignment. Also, it helped us to improve not only our knowledge about this specific topic of the course but

also it helped us to improve our teamwork and creativity. The whole Homework was very interesting, and it was a good challenge making us think and understand how the N Queen problem works. That was the most tricky part of the assignment. Finally, it helps us to obtain hands-on experience on how the agents can work together and even sacrifice their preferences to achieve a common goal.