DAIIA Assignment Report

Course: [Distributed Artificial Intelligence and AI Agents]
Assignment: Homework 3

Lorenzo Deflorian, Riccardo Fragale, Juozas Skarbalius
KTH Royal Institute of Technology

November 27, 2025

Contents

Running Instructions

Import the ZIP file given on GAMA and go to the folder assignment3/models. You will find a file called NQueen.gaml where our model for task 1 is implemented. Then, from the interface of GAMA, click on the play button and you will see a simulation. Use the tools provided by the GAMA simulation interface to adjust the speed, read the outputs and verify on screen that everything is working correctly. There is a parameter in the global of the module, called queens, that can be modified and defines the number of queens in the chess board. It is modifiable and should be used to verify that our code works for all N between 4 and 19. Regarding task 1, in the same folder assignment3/models there is a .gaml file called FestivalStages. Also in this case open it and play the simulation.

1. Task1: NQueen

General overview

We have to solve the N Queens problem; in particular the following are the rules of the game:

- Create a NxN size chessboard, placing N queens on it
- No two queens can share the same row
- No two queens can share the same column
- No two queens can share the same diagonal line

Starting from a random situation the Queens should adjust their position so that they do not violate the rules and they found a correct arrangement. This property must be validfor $N \in \{4, \ldots, 19\}$. In our setup we have a ChessBoard that appears on screen and a certain number of Queen agents, defined by the variable queens (as said in the running instructions). Each queen is an agent, they are able to communicate between each other through *"fipa-contract-net"* protocol. In our solution each queen is only able to communicate to its predecessor and to its successor. There is a sort of recursive procedure in which if a queen has no available position, she must let her predecessor know and ask her to reposition her. If also the predecessor has no available positions left, she must message her predecessor and so on and so for up until a correct arrangement is found. Queens in correct positions are depicted in green while queens that have not found the correct placement are depicted in red.

Code

First of all, each Queen is positioned on the chessBoard almost randomically using the following init procedure:

```
chessBoardCell myCell <- one_of (chessBoardCell);
```

```
[...]
 init {
    //Assign a free cell
    loop cell over: myCell.neighbours{
        if cell.queen = nil{
            myCell <- cell;
            break;
        }
    }

    location <- myCell.location;
    myCell.queen <- self;
    add self to: allQueens;
    do refreshOccupancyGrid;
}
```

The procedure starts by calculating the occupancyGrid and verifying whether there are conflicts with respect to the rules. When a conflict is found, queens need to move. The core relocation logic lives in the needToMove action. If a queen is in a conflict (indicated by a red color) and has no immediately free safe cells, it initiates negotiation. Unlike the previous approach based on visibility, the queens are organized in a circular linked list. Each queen is assigned a specific `predecessor` and `successor` during initialization. When stuck, a queen sends a request specifically to its predecessor.

```
// Communicate via chain: ONLY ask predecessor
if predecessor != nil and !awaitingResponse{
    do start_conversation to: [predecessor]
    protocol: 'fipa-contract-net' performative: 'cfp'
    contents: ['request_move', string(self.myCell.grid_x),
               string(self.myCell.grid_y), name];

    awaitingResponse <- true;
    messageContext <- "chain_request";
}
```

The requester includes its own name in the message so that if the request is forwarded down the chain, the eventual helper knows who to reply to.

Incoming CFPs trigger the `handleCFP` reflex. The logic follows a "Chain of Responsibility" pattern:

1. **If the receiver can move:** It moves to a free spot, effectively freeing its old cell. It then sends a `PROPOSE` message directly to the `originalRequester` (extracted from the message content) telling them the position is now available.
2. **If the receiver cannot move:** It acts as a middleman. It forwards the CFP to its own `predecessor`, preserving the `originalRequester`'s name. It then sends a `REFUSE` to the immediate sender to close that specific transaction.

3. **Chain Exhaustion:** If the request circles back to the original sender (Full Circle), the chain is exhausted, and the request is dropped to prevent infinite loops.

```
reflex handleCFP when: !empty(cfps){
    if length(myOptions) > 0 {
        // Move and notify original requester
        do start_conversation to: [originalQueen] ...
        performative: 'propose'
        contents: ['position_available', string(oldCell.grid_x), ...];
    } else {
        // Forward to predecessor
        do start_conversation to: [predecessor] ... performative: 'cfp'
            contents: ['request_move', ..., originalRequester];
    }
}
```

The `handlePropose` reflex processes the resolution. When a queen in the chain finds a spot and moves, it sends a `'position_available'` message to the queen that started the chain.

```
reflex handlePropose when: !empty(proposes) and awaitingResponse{
    if contents[0] = 'position_available'{
        // Move to the cell freed by the helper
        myCell <- freedCell;
        location <- freedCell.location;
        do accept_proposal
        message: proposalMessage contents: ['move_completed'];
    }
}
```

This creates a robust system where a single request can ripple through the entire population of queens until one is found that can shift to accommodate the group, resolving deadlocks that occurred in the visibility-based approach.

Finally, to provide visual feedback on the solution state, the queens update their color dynamically:

- **Red:** The queen is currently in a conflict (occupancy grid value > 0).
- **Green:** The queen is in a safe position (occupancy grid value == 0).

The simulation is considered solved when all queens turn green. The result is clearer separation: messages carry intent, reflexes react to performatives, and actions perform local state updates.

Demonstration

We will show below a couple of situations proving that our implementation is correct using N = 4. In our opinion it would definitely be better for a reader to run many simulations and verify that the N Queen problem is solved for that randomical beginning position.

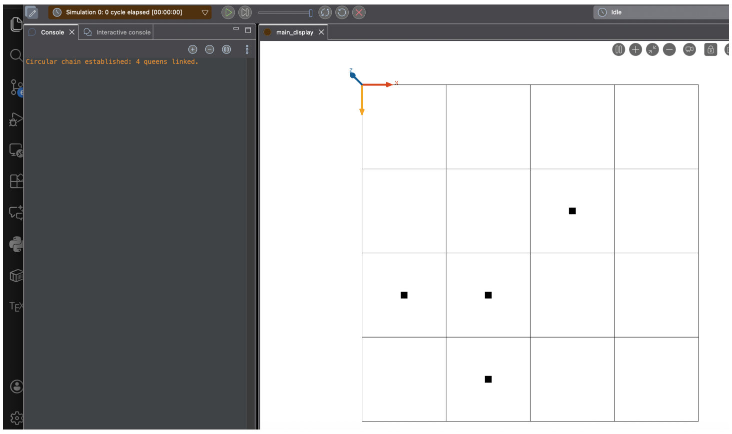

Figure 1: Starting point

As it can be seen from 1 the queens are placed correctly on the map.

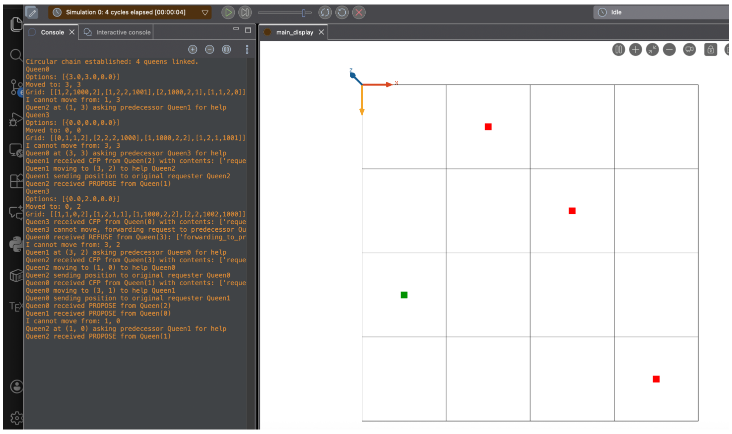

Figure 2: Queens are moving

2 shows that queens are communicating and changing their position in order to find a correct placement. It is clear also that sometimes the request is passed from predecessor to predecessor until someone can move.

Finally, 3 shows that the position in found.

We will also add a picture of what happens inside a run with N=19. Clearly the logs are more and the situation might be definitely more complex so more loops are needed to find the correct placement for all the queens in the chessboard.

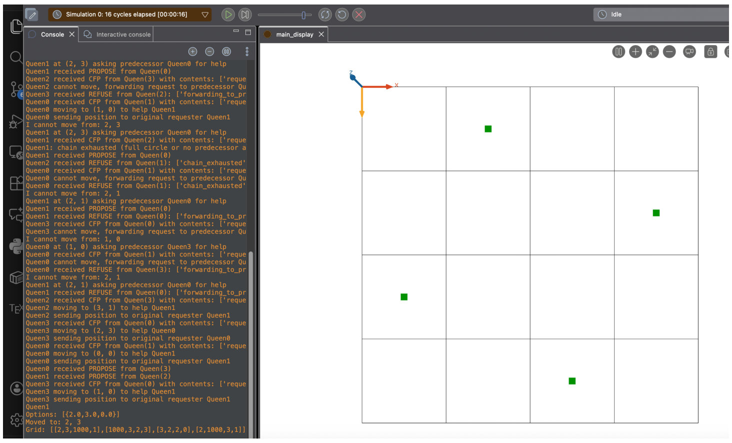

Figure 3: Position found

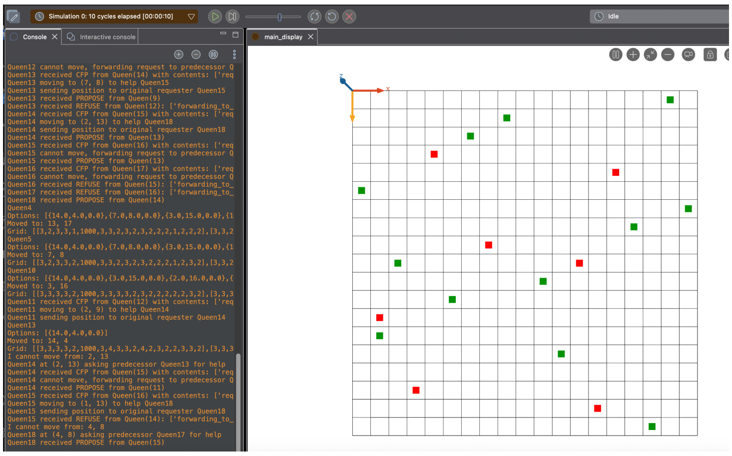

Figure 4: 19 queens moving on the chessboard

2. Task 2: Positioning speakers at main stage

For the purpose of this task we decide to eliminate the features of the previous homeworks related to the festival except for the InformationCenter and the skeleton of the normal guest. We were asked to put some stages into the festival which host an act for a different amount of time. The guest must know their position since the beginning of the simulation and must decide to go and attend an event at a festival based on a utility function. Basically, the decision is based on the guest preference and the characteristic of each stage related to some properties (e.g sound system ,light visuals etc). Guest communicate with stages through FIPA and after a "discussion" they must select the stage with the highest utility.

3.1 Explanation

Describe what this section requires and how you approached solving it.

3.2 Code

During initialization the model creates one InformationCenter, three Stage agents and ten Guests. Each guest receives a list of all stages (and also knows their position) as shown below

```
ask guests {
            set stages <- allStages;
     }
```

Each Stage has three key properties:

- lightShow (0–100)
- speaker (0–100)
- musicStyle (0–100)

These are randomly generated and represent the stage's entertainment characteristics. Stages also have a starting time and a duration, both different for each stage and randomically generated. Each guest has individual preferences (lightShowPreference, speakerPreference, musicStylePreference), all randomly initialized in the interval 0 to 100. Utility for a stage is computed as follows:

```
action compute_utilities(list<int> stats, Stage sender) {
int lightShow <- stats[0];
int speaker <- stats[1];
int musicStyle <- stats[2];
int utility <- lightShowPreference * lightShow
+ speakerPreference * speaker + musicStylePreference * musicStyle;
utilities[sender] <- utility;
}
```

Guests request stage statistics using FIPA Contract Net Protocol. The guest sends a cfp (Call For Proposal) performative to all stages with contents [”stats”].

```
    reflex request_utilities when: hasRequestedUtilities = false {
    write 'Requesting utilities from stages';
    do start_conversation to: list(Stage)
    protocol: 'fipa-contract-net' performative: 'cfp' contents: ["stats"];
    hasRequestedUtilities <- true;
}
```

When a stage receives a CFP with content ”stats”, it responds immediately with a propose performative containing its three attributes as a list: [lightShow, speaker, musicStyle].

```
    reflex receiveCFP when: !empty(cfps) {
    loop cfpMsg over: cfps {
        list contents_list <- list(cfpMsg.contents);
        if (contents_list[0] = "stats") {
            do propose message: cfpMsg contents:
            [lightShow, speaker, musicStyle];
        }
    }
}
```

All proposals are received in the receiveProposal reflex. For each one this reflex calls compute_utilities to score the stage and stores the result in a utilities map.

```
    reflex receiveProposals when: hasReceivedUtilities = false and
    hasRequestedUtilities = true and !empty(proposes){
    loop proposeMsg over: proposes {
        list<int> contents_list <- list<int>(list(proposeMsg.contents));
        do compute_utilities(contents_list, proposeMsg.sender);
    }
    hasReceivedUtilities <- true;
    do select_stage;
}
```

After processing all proposals, the guest invokes select_stage, which iterates through all stages and finds the one with the highest utility score stored in the utilities map.

```
    action select_stage {
    int maxUtility <- 0;
    Stage bestStage <- nil;
    loop stage over: stages {
        if (utilities != nil and utilities contains_key(stage)) {
            int stageUtility <- utilities[stage];
            if (stageUtility > maxUtility) {
                maxUtility <- stageUtility;
                bestStage <- stage;
            }
        }
    }
    selectedStage <- bestStage;
}
```

Once a stage is selected and the current time falls within [startFestivalTime, endFestivalTime], the guest computes a random target location near the stage and moves toward it using the moving skill with movingSpeed = 0.75. Guests navigate to their selected stage, arrive at a random position nearby (within 6–12 units from the stage), and then dance continuously

3.3 Demonstration

I will show below screenshots for two cases;

- Guests request utilities to stages
- Guest select the stage with higher utility

For the purpose of this demonstration I included only 4 guests so that the number of logging messages is reduced.

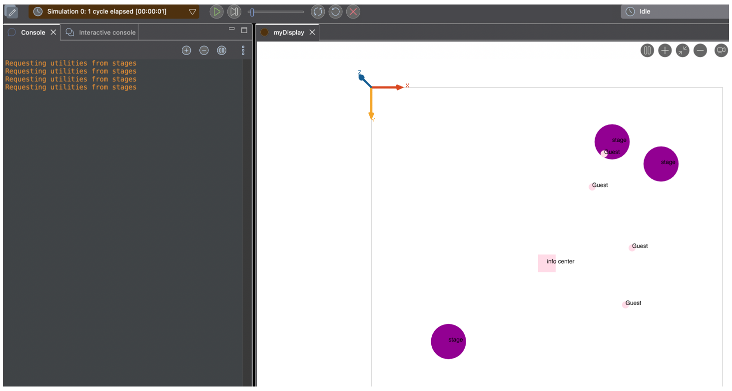

Figure 5: Guest request utilities

Guests request utilities to stages. A message is send to the stages in 5 to request utilities values.

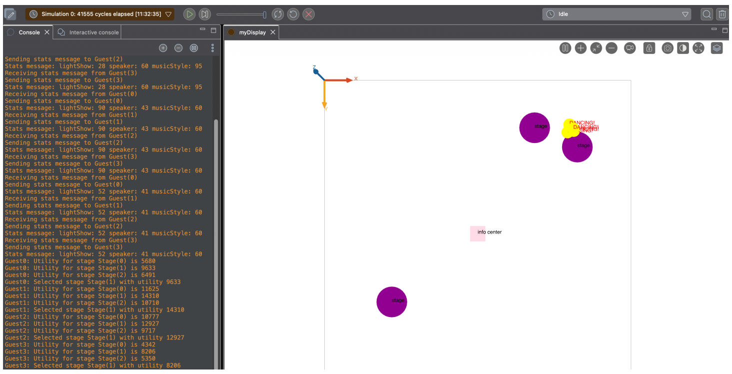

Figure 6: Guest select the stage with higher utility

Guest select the one with maximum utility. As it can be seen in 6 each guest is selecting the stage with the highest utility.

3. Challenge 1: global utility function

4.1 Explanation

Explain the challenge and your approach.

4.2 Code

Show relevant code snippets and screenshots.

4.3 Demonstration

Provide 4 complete use cases:

4. Final Remarks

In this homework we learnt a couple of things related to GAMA language and agents coordination. First of all, we were forced to create a "chain" of messages between agents that were bouncing back messages one to another (in the first task). Regarding the second task and its challenge it was not so easy to find a way to have a leader and to have the agents working together towards an objective following a utility function. We are really happy about our result and we are really looking forward to the final project which will certainly be more challenging.