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 24, 2025

Contents

1 Task1: NQueen 2

2 Section Reports 4

3 Challenge / Bonus Section (Optional) 5

4 Final Remarks 5

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.

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.

Code

First of all, each Queen is positioned on the chessBoard almost randomcally 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, which computes threats via calculateOccupancyGrid. If a queen has no immediately free safe cells, it initiates negotiation through FIPA instead of directly modifying another agent's state through ask instructions.

The requester sends a FIPA CFP using start_conversation to a visible queen found by findQueenInSightbyLocation.

```
do start_conversation to: [sight] protocol: 'fipa-contract-net'
performative: 'cfp'

contents: ['request_position', string(self.myCell.grid_x),
string(self.myCell.grid_y)];
```

The requester sets awaitingResponse ¡- true and stores messageContext ¡- ”position_request” to guard message handling.

```
// Handle CFP messages - respond with position information
reflex handleCFP when: !empty(cfps){
 message requestMessage <- cfps[0];
 write name + " received CFP from " +
   requestMessage.sender + " with contents: " + requestMessage.contents;

 // Respond with PROPOSE containing current position
 do propose message: requestMessage contents:
   ['position_info', string(myCell.grid_x), string(myCell.grid_y)];
}
```

Incoming CFPs trigger reflex handleCFP for the receiver , which reads the CFP contents and replies with a PROPOSE containing its position. The proposer uses do propose message: requestMessage contents: ['position_info', string(myCell.grid_x), string(myCell.grid_y)]. The requester watches for proposes and the handlePropose reflex (guarded by awaitingResponse and messageContext) processes proposals.

```
    reflex handlePropose when: !empty(proposes)
     and awaitingResponse and messageContext = "position_request"
```

handlePropose parses ['position_info', x, y], looks up the corresponding chessBoardCell, and scans its neighbours for the nearest empty target. If a valid target is found the requester updates its local myCell and location, and sends ACCEPT_PROPOSAL to confirm the move.

```
    do accept_proposal message: proposalMessage contents: ['move_completed'];
```

If no valid target exists, the requester sends REJECT_PROPOSAL so the proposer knows the negotiation failed.

```
    do reject_proposal message: proposalMessage contents: ['no_valid_target'];
```

After processing proposals the requester resets awaitingResponse ¡- false and clears messageContext, preventing stale reflex activations. The amIsafe reflex is guarded with !awaitingResponse and !isCalculating to avoid concurrent negotiations or reentrancy. This message-driven pipeline replaces direct ask calls, ensuring all state changes happen locally in response to explicit FIPA messages. Key flags (awaitingResponse, messageContext) act as triggers so reflexes run only when a real message exchange occurred. 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.

2. Section Reports

Repeat the structure below for each part of the assignment.

Section X: [Section Name]

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

3.2 Code. Include relevant code snippets below:

```
# Example code snippet
def example():
    print("Hello, DAIIA!")
```

Optionally include screenshots with:

3.3 Demonstration. Provide two use cases for this section.

1. **Input:** Describe the input.
2. **Screenshot:** Include program execution/output.
3. **Interpretation:** Briefly explain the result.

3. Challenge / Bonus Section (Optional)

If you attempted bonus tasks, describe them here.

Challenge X: [Challenge Name]

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:

1. Input description.
2. Screenshot of program execution/output.
3. Short explanation of results.

4. Final Remarks

Summarize what you learned, any limitations, and potential improvements. Optionally include any extra comments.