DAIIA Assignment Report

Course: [Distributed Artificial Intelligence and AI Agents]
Assignment: [Homework 1]

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

November 14, 2025

Contents

1. Running Instructions

Import the ZIP file given on GAMA and go to the folder assignment1/models. You will find a file called Festival.gaml where our model for the homework 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. Note that changing the parameters *numCenter,numShop and numGuests* will change the number of agents that will appear on the screen and be part of the simulation.

2. General Overview

2.1 Agents

We have defined 4 main species:

- InformationCenter
- Shop
- SecurityGuard
- Guest

We also have two subspecies of Guest that are called SmartGuest and BadApple and they have been implemented to obtain the objectives requested by challenge 1 and 2.

The agent of type **InformationCenter** is responsible to give informations to all the agents that are asking him where the shops are. He knows the locations of them and is also connected to a SecurityGuard that will be helpful for the purpose of the second challenge. Regarding the aspect of this agent, it is depicted as a black square of length 5 and its location is right in the center of the field where the simulation is happening. All the other agents know by default the position of the InformationCenter.

A **Shop** is an agent responsible of replenishing food or water to all the guests that are coming to them. They are divided into food shops and water shops and their position changes in each simulation. Water shops are depicted as grey triangles while food shops are red triangles.

The **Guest**, instead, takes part in the festival and basically wanders randomly unless it is either hungry or thirsty. In this case it goes to the InformationCenter and asks where to find food or water to fulfill its need. Then he starts moving to the location given by the InformationCenter and "charges its batteries". Then, it continues moving aimlessly unless it feels hungry or thirsty again(or both). The initial value for hunger and thirst are set to 0, the maximum value reachable is 1000 and the threshold is set to 300. A guest is a pink dot.

As we said before, there is also the **SmartGuest** that has a memory and so it is able to remember the locations of the shops visited. In this way he could move less with respect

to normal guests and it will satisfy its need faster. The SmartGuest appears on screen as a yellow dot.

The other subtype of Guest, called **BadApple** is requested for challenge 2 and it has the ability to follow a guest and attack it, either by changing direction towards the guest or by reducing/adding thirstiness and hungriness. Other guests can report him to the InformationCenter which will use the SecurityGuard to kill the BadApple.

Last but not the least, there is the **SecurityGuard** that is responsible to catch and kill all the badApples that do not follow the rules of the festival and must be eliminated. The security guard is a blue dot and its starting position is north-west with respect to the InformationCenter.

2.2 Assumptions

Our model has 1 InformationCenter, 1 SecurityGuard, 4 Shops (two of which of type *food* and two of type *water*). Regarding the Guests, we have defined 3 SmartGuests, 10 guests without memory and 4 BadApples.

2.3 Goals

The goals of this homework are:

- Introduction to the Gama platform
- Working with agents
- Learning the GAMA syntax
- Creating different types of agents
- Starting basic simulations
- Little bit of movement and behaviour

3. Basic implementation

3.1 Explanation. The basic implementation required us to create a simulation of a festival where Guests get hungry or thirsty. If they do, they should go to an Information Center to ask for the nearest Shop that gives them what they need. Afterward, the Guests should simply keep doing something until they get hungry/thirsty again (they wander randomly).

3.2 Code. For this challenge we implemented all the requested agents and we setup a system through which there is a communication between the agents.

First of all, a thing we need to specify is that every loop the hunger and the thirst are updated in the following way

```
  reflex update_needs {
     hunger <- min(maxHunger, hunger + rnd(0, 1));
        thirsty <- min(maxThirst, thirsty + rnd(0, 1));
    }
```

In fact the need of food and water increase randomically (maximums and threshold are already specified in the previous section of this document).

Going on, this is the portion of code where we designed the interaction between the information center and the guest.

```
reflex reached_info_center when:
        infoCenter != nil
        and onTheWayToShop = false
        and targetShop = nil
        and (location distance_to infoCenter.location) < 1.0 {

        // Check BOTH needs
        bool needFood <- (hunger >= hungerThreshold);
        bool needWater <- (thirsty >= thirstThreshold);

        // If no needs, exit early
        if (!needFood and !needWater) {
            return;
        }

        // Determine which need to address
        string primaryNeed;

        if (hunger = thirsty) {
            // If equal, choose randomly
            primaryNeed <- one_of(["food", "water"]);
        } else if (hunger > thirsty) {
            primaryNeed <- "food";
        } else {
            primaryNeed <- "water";
        }

        targetShop <- infoCenter.getShopFor(need: primaryNeed);

        if (targetShop = nil) {
            write "No target shop found for " + primaryNeed;
            return;
        }

        onTheWayToShop <- true;
        currentAction <- "-> " + primaryNeed + " shop";
        write "Going to " + targetShop + " to get " + primaryNeed + " - hunger: " + hu
 }
```

The info center checks the need of the guest and assign him a shop to go. The guest then starts going to the target shop. Whenever the guests are not thirsty nor hungry they remain on idle and they wander as it can be seven from the piece of code below.

```
  reflex wander when: hunger < hungerThreshold and thirsty < thirstThreshold and onThe
        do wander speed: movingSpeed;
    }
```

.

One important aspect is that guests know the position of the InformationCenter by default.

```
  ask guests {
             infoCenter <- center[0];
         }
```

This ask is called inside the init and will be applied also for the other type of guests defined for the two challenges.

Finally, when the user reaches a shop, the hunger or the thirst are solved thanks to this reflex and the action it is subsequently calling.

```
  reflex reached_shop when: targetShop != nil and (location distance_to targetShop.loc

        // Satisfy the need for this shop type
        do satisfy_needs;

    }

    action satisfy_needs {
  string shopType <- targetShop.getShopType("");
  if (shopType = "food") {
            hunger <- 0;
            write "Ate food! Hunger reset.";
        } else if (shopType = "water") {
            thirsty <- 0;
            write "Drank water! Thirst reset.";
        }
 }
```

3.3 Demonstration. I will be providing a couple of use case for this initial part of the homework.

1. An agent is assigned a food shop when it is hungry
2. An agent gets food and so hungry is set back to 0

For the following two subsections we did the screenshots considering only a normal guest without memory, 4 shops and an InformationCenter at the centre of the map.

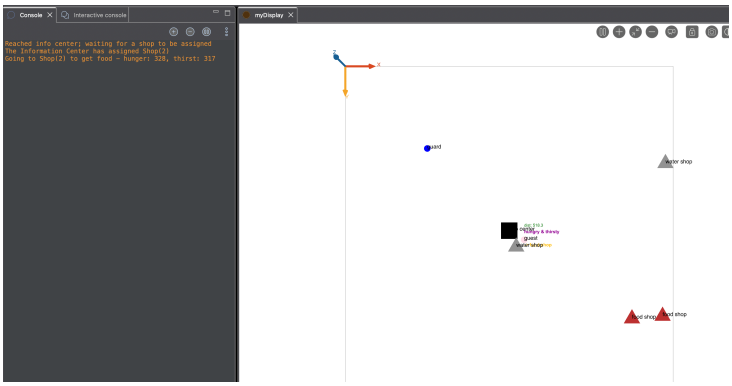

Figure 1: A food shop is assigned to the guest

3.3.1 Shop assignment. As it can be seen, the agent ask the info center where to go and the info center tell him to go to a food shop. Then the guest start moving toward the assigned shop.

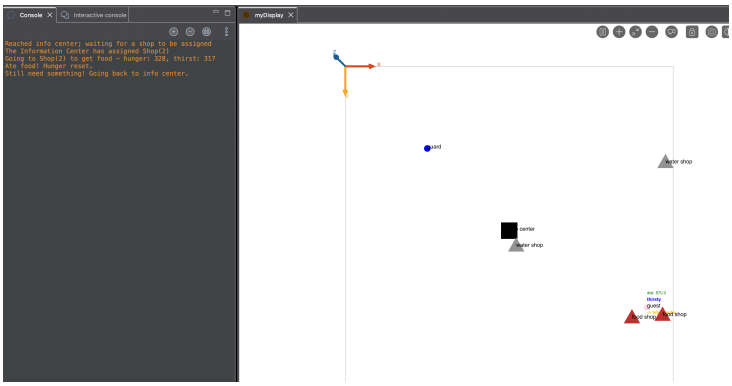

Figure 2: Hungriness is set to 0

3.3.2 Replenish hungriness. As it can be seen, the agent buys and eats food. Then it goes back to the info center sine the thirstiness is above the threshold.

4. Challenge 1

Smart Guests

4.1 Explanation. For this challenge we implemented a new type of Guest, called SmartGuest, that has a memory of the shops he has visited. This way if he gets hungry or thirsty again, he can decide to either go to the InformationCenter to ask for a new shop or go directly to a known shop.

```
species SmartGuest parent: Guest{
    list <Shop> visitedPlaces;

    reflex manage_needs{
        if ((hunger >= hungerThreshold or thirsty >= thirstThreshold) and onTheWayToShop = false and targetShop = nil) {
            // No visited places go to info center
            if (length(visitedPlaces)) <= 0 {
                currentAction <- "-> Info Center";
                do go_infocenter;
                return;
            }

            bool visitNewPlace <- rnd(0, 1);
            if (!visitNewPlace) {
                currentAction <- "-> Info Center";
                do go_infocenter;
                return;
            }

            list shuffledPlaces <- shuffle(visitedPlaces);
            loop i from: 0 to: length(shuffledPlaces) - 1{
                Shop candidateShop <- shuffledPlaces[i];
                if (hunger >= hungerThreshold) {
                    if (candidateShop.trait = "food") {
                        targetShop <- candidateShop;
                        onTheWayToShop <- true;
                        currentAction <- "-> Known " + candidateShop.trait + " shop";
                        return;
                    }

                }

                if (thirsty >= thirstThreshold) {
                    if (candidateShop.trait = "water") {
                        targetShop <- candidateShop;
                        onTheWayToShop <- true;
                        currentAction <- "-> Known " + candidateShop.trait + " shop";
                        return;
                    }

                }
            }

```

Figure 3: Code snippet for SmartGuest behavior

4.2 Code. The code snippet in Figure 3 shows how the SmartGuest decides whether to go to a known shop or ask the InformationCenter for a new one. With a probability of 50%, the SmartGuest chooses to visit a previously known shop from its memory of visited places. It randomly selects one shop that satisfies its current need (food or water) and heads there directly; otherwise it behaves like a normal Guest and goes to the InformationCenter.

Whenever a SmartGuest visits a shop, it adds that shop to its list of visited places so it can remember it for future needs, reducing travel time on repeated needs.

SmartGuests are represented as yellow dots on the simulation screen.

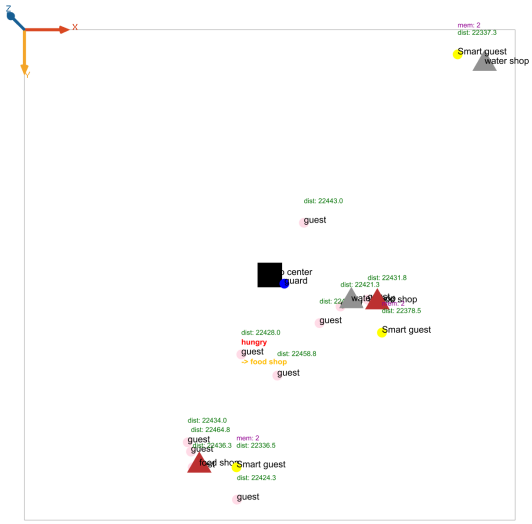

Figure 4: SmartGuests walking less due to memory

As shown in Figure 4, SmartGuests walk less compared to normal Guests because they can go directly to known shops instead of asking the InformationCenter every time they get hungry or thirsty.

4.3 Demonstration. Provided use cases for this challenge:

1. A guest with no memory goes to the info center.
2. A smart guest with memory still goes to the infor center to discover a new shop.
3. A smart guest decides to go to a known food shop from its memory.
4. A smart guest decides to go to a known water shop from its memory.

4.3.1 Guest with no memory

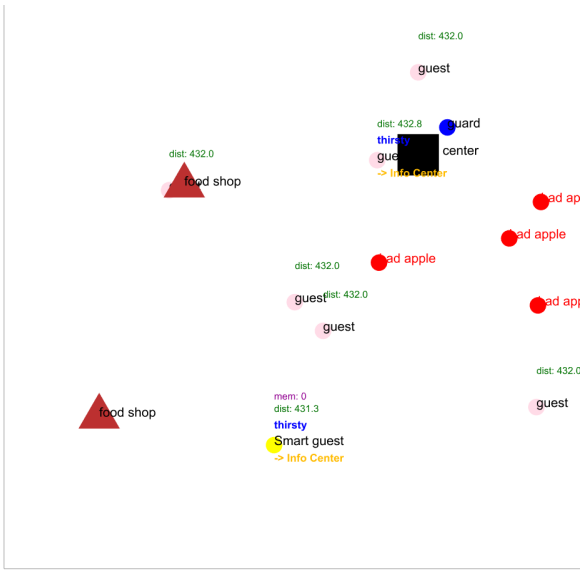

Figure 5: Guest with no memory going to Information Center

In Figure 5, we see a Smart Guest (yellow dot) that has no memory of shops. When it gets hungry, it goes to the Information Center (black square) to ask for the nearest food shop.

4.3.2 Smart Guest discovering new shop

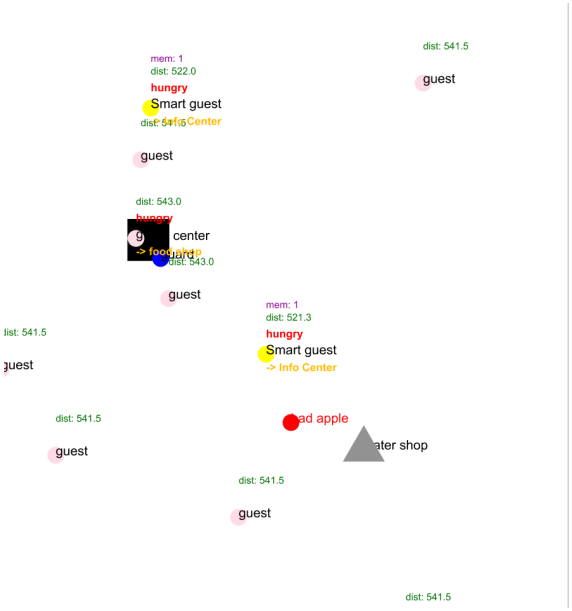

Figure 6: Smart Guest discovering a new shop

In Figure 6, we see a Smart Guest (yellow dot) that already has one shop in its memory. However, it decides to go to the Information Center (black square) to discover a new shop instead of going to the known one.

4.3.3 Smart Guest going to known food shop

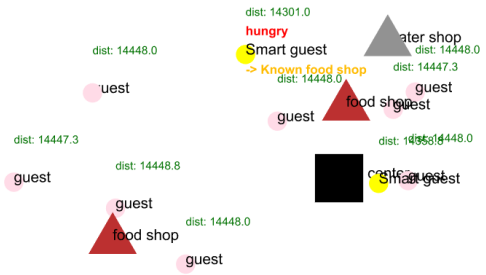

Figure 7: Smart Guest going to known food shop

In Figure 7, we see a Smart Guest (yellow dot) that has a food shop in its memory. When it gets hungry, it decides to go directly to the known food shop (red triangle) instead of

asking the Information Center.

4.3.4 Smart Guest going to known water shop

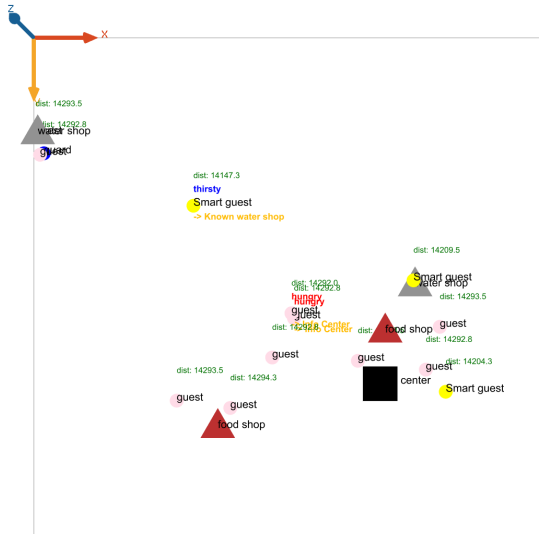

Figure 8: Smart Guest going to known water shop

In Figure 8, we see a Smart Guest (yellow dot) that has a water shop in its memory. When it gets thirsty, it decides to go directly to the known water shop (grey triangle) instead of asking the Information Center.

5. Challenge 2: SecurityGuard and Bad Apples

5.1 Explanation

For this challenge we implemented the behavior of a BadApple agent (a guest type that attacks other guests) and a SecurityGuard agent that eliminates BadApples when reported by other guests. Bad behaving agents are reported by the victim Guest agent to the InformationCenter, which then instructs the SecurityGuard to locate and eliminate the reported BadApple.

5.2 Code

The BadApple agent randomly selects a target Guest to follow and attack. When it gets close enough to the target, it performs an attack by reducing the target's hunger or thirst levels, or by shaking the victim by bumping into it. When a Guest is attacked, it reports the incident to the InformationCenter. The InformationCenter then instructs the SecurityGuard to locate and eliminate the reported BadApple. The SecurityGuard moves towards the BadApple's location and "kills" it upon reaching it.

```
// chase if we have a target of type normal guest
if (targetGuest != nil) {
    do goto target: targetGuest speed: chase_speed;

    // if close enough, "attack"
    if (self distance_to targetGuest < attack_range) {
        BadApple attacker <- self;
        write "Attacking a normal guest";
        ask targetGuest {
            int   hunger_bump    <- 50;    // how much hunger to add
            int   thirst_bump    <- 30;    // how much thirst to add
            float shove_amplitude <- 2.0;  // how much the victim stumbles

            // make life harder
            hunger  <- min(100, hunger  + hunger_bump);
            thirsty <- min(100, thirsty + thirst_bump);

            // cancel their current trip so they re-plan
            onTheWayToShop <- false;
            targetShop <- nil;

            // make them stumble
            do wander amplitude: shove_amplitude;

            // Set attacked flag and go to info center
            beingAttacked <- true;
            attackerRef <- attacker;
            do go_infocenter;
        }

    }
}
```

Figure 9: Code snippet for BadApple attack behavior

5.3 Demonstration

Provided use cases for this challenge:

1. BadApple scans and approaches its targets
2. A BadApple attacks a Guest.
3. InformationCenter is notified about the attack and informs the SecurityGuard.
4. The SecurityGuard locates and eliminates the BadApple.

5.3.1 BadApple scans and approaches its targets. The BadApple agent maintains a list of all other Guest agents in the simulation. It randomly selects one Guest from this list as its target and starts moving towards it.

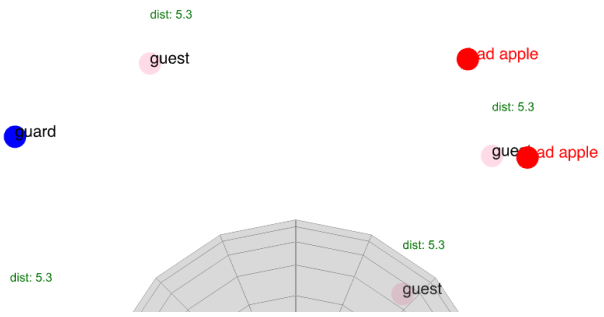

Figure 10: BadApple approaching its target Guest

5.3.2 BadApple Attack. Each BadApple, which is a sub-agent of Guest, maintains a list of all other Guest agents. It randomly selects a target Guest from this list and starts following it. When the BadApple gets within a certain distance of its target Guest, it performs an attack by randomly decreasing the target's hunger or thirst levels, or shaking the victim by bumping into it. Initial attack is illustrated in the figure below. Guest detects when it is being attacked by a BadApple. Upon detection, the victim Guest reports the incident to the InformationCenter by going to InformationCenter and notifying it about the BadApple's behavior.

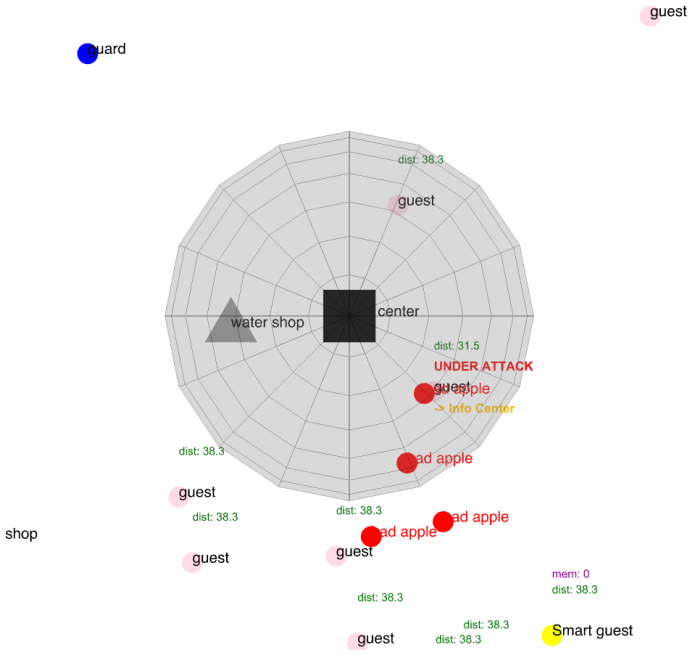

Figure 11: Simulation snippet when BadApple is attacking a Guest

5.3.3 InformationCenter is notified about the attack and informs the SecurityGuard. Upon receiving a report from a victim Guest about a BadApple attack, the InformationCenter records the location of the reported BadApple. The Information Center then instructs the Security Guard to locate and eliminate the reported BadApple.

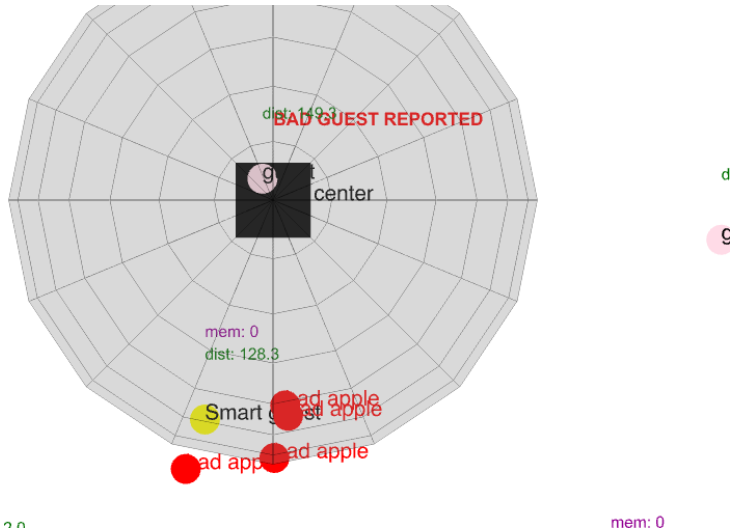

Figure 12: InformationCenter notified about BadApple attack

5.3.4 The SecurityGuard locates and eliminates the BadApple.. The SecurityGuard receives the location of the reported BadApple from the InformationCenter. It puts the BadApple agent into the list of misbehaving agents. Then it approaches and neutralizes the closest BadApple to guard's current position.

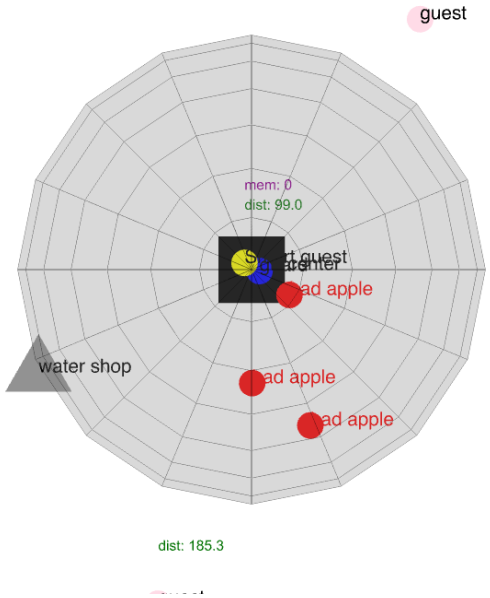

Figure 13: Security Guard eliminating the BadApple

6. Final Remarks

Overall this assignment was great. We had the opportunity to learn the basics of the syntax of GAMA and we were challenged to implemented interesting things. We realized that changing a bit the inputs or minor details might lead to big changes in each simulation.