Università degli Studi di Camerino

Scuola di Scienze e Tecnologie

Corso di MSc in Computer Science (LM-18)

Distributed Calculus and Coordination Project: The Idiotypic Network

Students
Luca Fuligni
Michele Russo
Marco Zamponi

Supervisore
prof.ssa Emanuela Merelli

A.A. 2020/2021

Indice

1. Introduction

The presented project is contextualized within a generic adaptive immune system represented through immune network theory. In particular, the main purpose of the work is to offer an *agent-based* simulation of the idiotypical network conceived by Giorgio Parisi [*paper Parisi*].

In the Chapter 2 a brief general presentation of the domain will be made: the immune system (in particular the adaptive one), the theory of the immune network and Parisi's model.

In the Chapter 3 the modeling will be discussed as an intermediate step between domain and implementation: how the domain has been interpreted and what assumptions have been made, trying to delineate in an optimal way the agents that are part of the implementation.

In the Chapter 4 will be shown the implementation made through *Java Repast Symphony*, following the *agent-based paradigm*. The general structure of the code will be mentioned: the division in classes, the choice of methods and more.

In the Chapter 5 will be shown how the simulation works. Will be introduced all the various types of parameters and how they affect the latter. Finally, it will be shown the various possible stages of progress both in terms of high-level and low-level vision.

In the Chapters 6 the main difficulties and critical points encountered during the project will be presented, also thinking about possible future developments.

2. Domain

The immune system is a complex system whose main purpose is to prevent and limit infections. Any kind of organism is constantly exposed to external pathogens and the task of the immune system is to activate certain cells and proteins that fight against these invading microorganisms. The immune system operates according to two main phases:

1. pathogen identification;

2. appropriate response to the problem.

In the following, the subdivision of the immune system into its two main categories will be introduced. In particula,r the adaptive immune system will be considered, which represents the specific domain for the project carried out.

2.1 Breakdown of the immune system

As mentioned above, the immune system is fundamental to protect the organism from infections or other pathologies. Depending on the level at which it operates to defend the organism, it can be divided into two complementary types:

- *Innate immune system*: the word *innate* indicates how this part of the immune system is present from birth and is the result of long-term evolution. It is the non-specific part of the immune system, meaning it has similar mechanisms for different pathogens;

- *Adaptive immune system*: the part of the immune system that is created during the life of the organism, through continuous exposure to external substances. Contrary to the innate one, the adaptive one has a very specific response depending on the pathogen considered because the cells of which it is composed are highly specialized.

2.2 Adaptive immunity

The adaptive immune system is concerned with the production of specific cells or antibodies that destroy a particular antigen. An antigen is defined as any element, such as pathogens, food, drugs, pollen, or tissue, that the immune system recognizes as foreign. As a whole, the adaptive immune system is composed of lymphocytes and the antibodies that are produced by them.

2.2.1 Response mechanism in general

When an antigen enters the body, the first and second defenses are the adaptive immune system.

During the first exposure to an unrecognized antigen, the immune response is minimal. However, cells remember the exposure to said antigen and create an immunological memory of the encounter. This immunological memory remains constant over time and allows a defense to be created.

During a second exposure the response is more amplified: lymphocytes and specific antibodies for the pathogen in question are sent.

2.2.2 Adaptive immune system cells

The cells that make up the adaptive immune system are the *B lymphocytes* and the *T lymphocytes*.

T lymphocytes are further divided into two subgroups: the *CD4 helper T lymphocytes* and the *CD8 killer T lymphocytes*.

The former are responsible for the proper functioning of the rest of the immune cells, including CD8 cells; the latter kill pathogens by interacting directly with the invading microbes.

B lymphocytes, on the other hand, produce antibodies, which are designated to recognize and capture antigens that are present on the surface of bacteria and other pathogens. In fact, antibodies do not destroy invaders on their own, but they activate a number of different mechanisms of the immune system so that they can be destroyed by immune cells.

2.2.3 Different types of adaptive immunity response

Adaptive immunity consists of two types of closely related immune responses, both of which are triggered by antigens. In the cell-mediated response some T cells behave like an army of soldiers directly attacking the invading antigen by chemical and physical means. In the antibody-mediated response, B lymphocytes are transformed into plasma cells that synthesize and secrete antibodies; a given antibody can bind and inactivate a specific antigen.

2.3 Immune Network Theory

The functioning of the adaptive immune system can be seen through the *immune network theory*, conceived in 1974 by Niels Kaj Jerne [*N. K. Jerne (1974) Towards a network theory of the immune system. Ann. Immunol. (Inst. Pasteur), 125C, 373-38*].

In its broad conception, the immune network theory states in particular that the production of antibodies, following an external antigen, can be seen as a chain phenomenon: the production of an antibody Ab_1 is able to cause or inhibit the production of an antibody Ab_2, in turn able to cause or inhibit the production of an antibody Ab_3 and so on.

As we will see below, Parisi formulated a simple model that could represent the behavior of antibodies following exposure to an external antigen, based on the theory formulated by Jerne [*citazione paper Parisi*].

2.3.1 Parisi's Model

In 1989, Parisi provided a formalization of the idiotypic network so as to represent antibody behavior in the context of adaptive immunity. In his conception, Parisi wanted to describe the evolution of the network without the internal presence of any antigen. In particular, he focused on two fundamental aspects: the immune system's behaviour and the immunologic memory's evolution. The model is a simple theoretical framework that allows the derivation of results analitically, without the need for simulations. In a common immune system there are more than 10^6 kinds of different molecules that interact with each other. In this scenario, antiidiotypic antibodies are generated in response to external antigens, for example through vaccination.

Initial study of the network

The first step proposed by Parisi is the developing of assumptions that will be used in the model definition. His studies mainly concentrated on some aspects of the functional role of idiotypic networks that are not yet fully understood, like:

- Does it contains a small set of high responder clones or a large set of low responder clones?
- The addition of an antigen in the network modifies the whole network or the perturbation is localized?
- The network is a unique indivisible unit or is it composed by a large number of subnetworks? Is so, are they open or closed?
- Assuming states depend on internal dynamics, how the learning physically happens?
- How large in general is the immunologic memory?

Starting from these key points, Parisi developed a set of assumptions that will define the entire model. In particular, he chose the most extreme hypotesis, for simplifying the model the most. Such set of assumptions are:

- There is a large set of low responder clones.
- The entire network has such an high connectivity that it can be considered a unique entity.
- The immunological memory is a shared property of the whole network.

Known facts

The immune network is composed by a precise number of antibodies that can be produced at any moment. Such number is in the order of $10^6 - 10^7$, while the number of antibodies that is actually produced at any given moment is usually 10 times smaller. With these knowledges, the repertoire of antibodies can be considered complete and can react to every protein. In particular, such proteins that stimulate antibodies, that are considered antigens, can produce two different reactions:

- Tolerance;

– Immunity.

The decision of the reaction that should be taken depends on many factors, for example:

– The amount of antigen introduced inside the immune system;

– The way they enter the organism.

The taken decision is remembered by the immune system for a long time, sometimes lifetime too. Another important factor to consider is the idiotypic cascade, which defines the chain of antibodies created or inhibited inside an organism by the action of other antibodies. However, It's now always true that the same organism produce the same idiotypic cascades, for example idiotypic cascades can be afected by:

– Concentration boost in injection;

– Modification in the chain;

– Possibility of different idiotypic environments.

Model

The main goal of this model is to describe an immune system and its associated network functionally and in an useful way as far memory is concerned. Another goal is to make it simplified to the maximum. For this reason there are no antigens nor B-T cells and the actors are the antibody concentrations. In particular, the state of the system in an istant of time t is:

$$s_t = c \in 0, 1^n \mid n \sim 10^7 \tag{2.1}$$

Instead, the concentration of a single antibody at time t is defined as $c_i(t)$. Conventionally, the value of the concentration is 1 or 0. The state of the system at a given time is given by all the concentrations $c_i(t)$ of the possible antibodies.

We define the following equations:

$$h_i(t) = S + \sum_{k=1}^{N} J_{i,k} c_k(t) \tag{2.2}$$

$$c_i(t + \tau) = \theta[h_i(t)] \tag{2.3}$$

Referring to the formula (2.2), $J_{i,k}$ is a matrix $J \in R^{n \times n}$ that represents the influence of the antibody k on the antibody i. If positive it causes its production, if negative it inhibits it. In addition, τ represents a discretized time of about one week, that corresponds to the average immune response time. The function $\theta(x)$ in (2.3) is described as:

$$\theta(x) = \begin{cases} 0 & if \quad x < 0 \\ 1 & if \quad x \geq 0 \end{cases} \tag{2.4}$$

In general, the variable h_i represents the stimulatory or inhibitory effect of the network on the antibody i. The quantity S governs the dynamics when the terms of J are very small. If for simplicity we were to take zero and the antibody concentration was independent of time, the equations (2.2) and (2.3) can be rewritten as:

$$h_i(t) = \sum_{k=1}^{N} J_{i,k} c_k \tag{2.5}$$

and

$$c_i = \theta[h_i] \tag{2.6}$$

The development of most of the project is based exactly on the model conceived by Parisi. In the next section it will be described how, starting from the domain and from the aforesaid model, it has been arrived to the implementation.

3. Development methodology

To create this MAS we are preceded by following an iterative approach, which started in November and was divided into 3 main iterations at the end of each of which we presented a working prototype of our system.

3.1 First iteration: Conway's game of life

The first iteration of the project began in November and coincided with the realization of point one of our project requirements. To begin to better understand how to approach agent programming we have created a simulation of Conway's life game. So after a thorough study of various possibilities we decided to use repast java as the main development framework. At the end of this activity we had in hand a working simulation of the game of life created in Repast Symphony. The agents involved were of only one type: cells. They were located within a grid and could only have two states: live or dead. Depending on the number of cells in their neighborhood, they could change their state or not.

In this very simple project we could already see a set of agents cooperating one each other, but they still weren't working towards a common goal. This was the purpose of our second iteration.

3.2 Second iteration: idiotypic network

In this second iteration we have adapted our game of life in order to simulate the reaction of an idiotypic network to an external agent as described by Parisi in his paper [parisi paper]. At this point, our cells translated into antibodies, which too could be alive or dead. Each antibody had a specific type (represented for demonstration purposes by an incremental identification number) and an hValue. The positivity or negativity of this hValue resulted respectively in the presence or absence of that given antibody within the idiotypic network.

At the time of antibody creation, a matrix was created (reflective and symmetric) representing how much an antibody of a given type affected the hValue of another antibody. At each step, each live antibody updated the hValue of all the other antibodies present within the idiotypic network, and after being in turn influenced by all the other antibodies it changed its state based on the remaining hValue.

We can therefore see that at the end of this second iteration we have obtained a set of agents collaborating in order to obtain a state of equilibrium within an idiotypic network. This represents what happens when a recognized antigen bypasses the innate immune system and reaches the adaptive one. However, we cannot yet keep track of immunological memory: this was the goal of our third and final iteration.

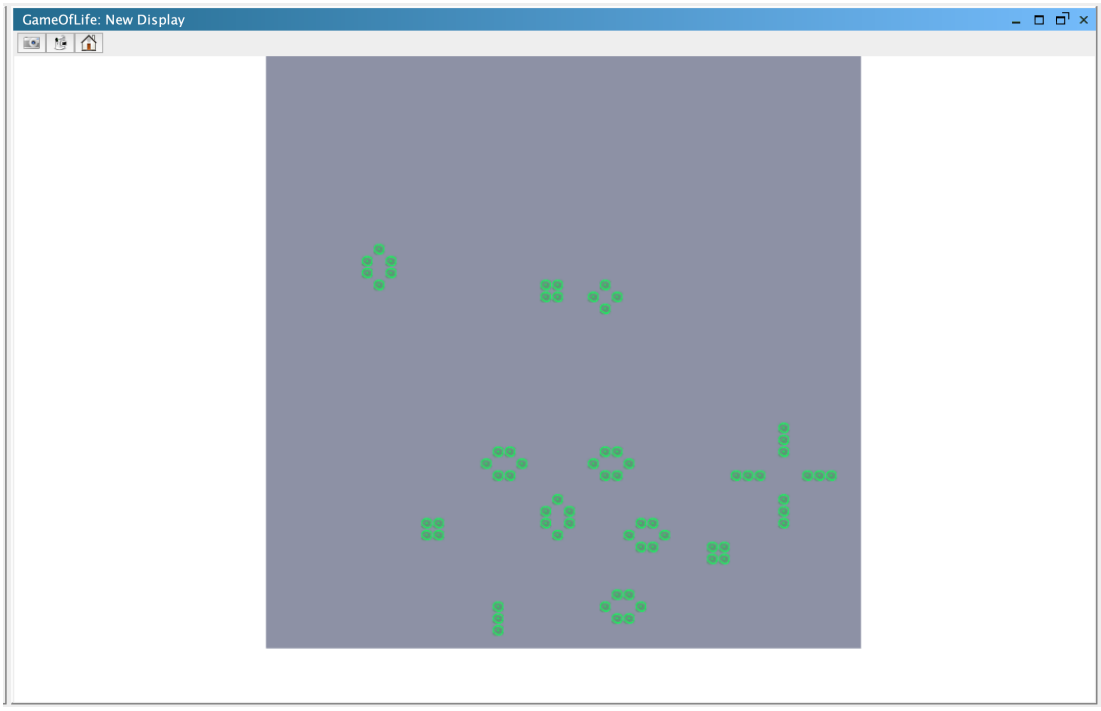


Figura 3.1: How our Game of Life simulation looks like once it reached an equilibrium

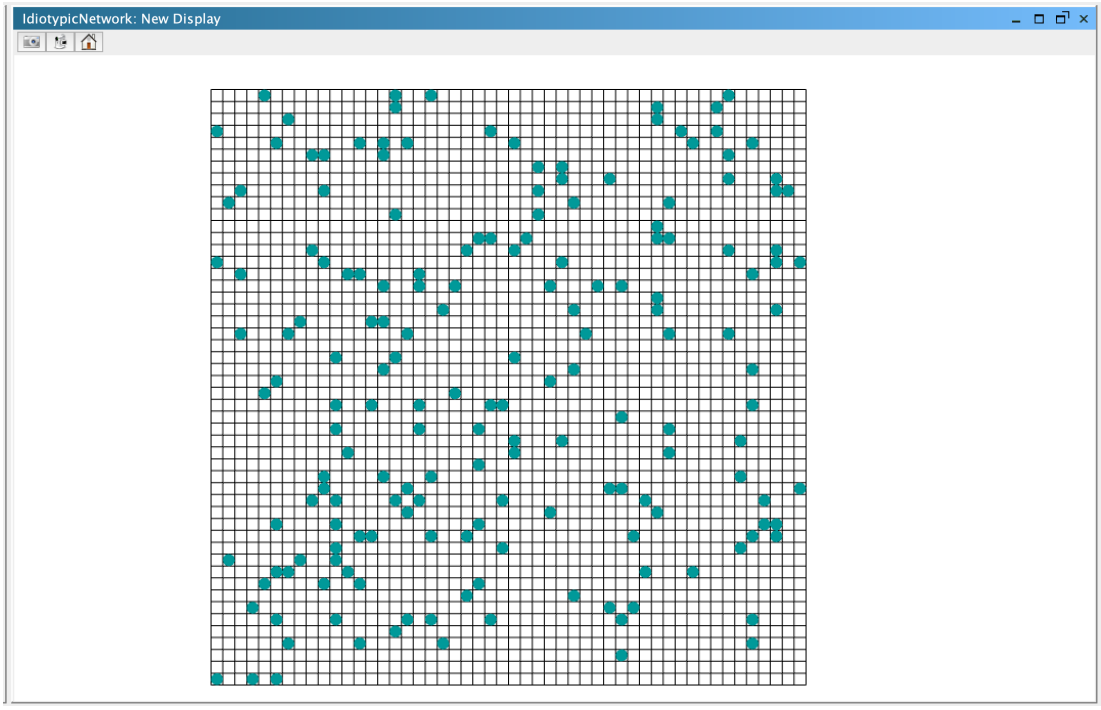


Figura 3.2: How our Idiotypic Network simulation looks like once it reached an equilibrium

3.3 Third iteration: immunological memory

For this third and final interaction, we aimed to maintain immunological memory. To do this, we needed an extra layer of abstraction. In fact, what has been done so far perfectly represented the behavior of an idiotypic network in contact with an antigen already recognized and present within the matrix, but in order to simulate an immunological memory we also had to simulate the presence and influence of the various antigens, recognized or not.

We therefore created a new grid, representing a high-level view of the organism and containing only two agents: the "immune system" and the "external agent". "The external agent" does nothing but check whether the immune system is in a state of equilibrium or not, and according to it, create a new antigen. The immune system at this point will be responsible for managing the Parisi's matrix, and will only react to the presence of an external antigen by losing its state of equilibrium. In case the immune system has never encountered the aforementioned antigen, it will create a new line and column in the Parisi matrix (with random values in our simulation) and a new antibody capable of responding to this antigen by inserting it into the grid related to the idiotypic network.

At this point it will start calculating its equilibrium state again. If it finds it, it will remove the antigen and be ready to fight another, otherwise if - after a number of moves at the user's discretion - it still does not find a state of equilibrium, the simulation will end representing the death of the organism.

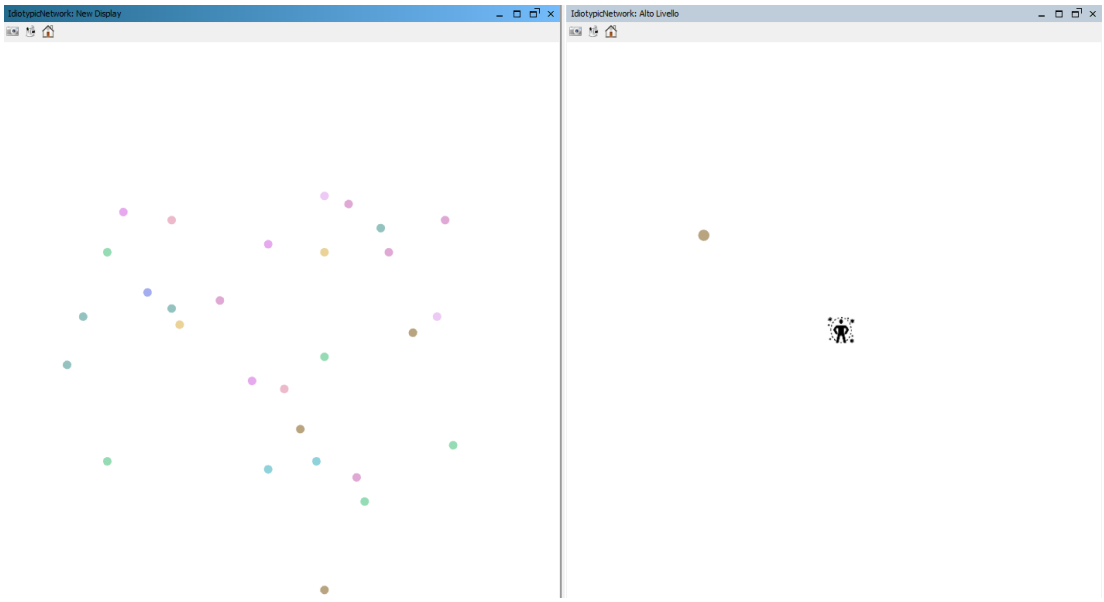


Figura 3.3: How our final simulation looks like once it reached an equilibrium

4. Implementation

The literature reveals many different methods to program agents, to create this project we chose to program object-oriented using agents as behavior controllers.

The Object-Oriented (OO) paradigm regroups the data and capabilities of existing implicit structures in implementations via entities called "objects". Every object created from the same "class" shares the same capabilities, while possessing its own parameters value. Heterogeneous agents fit perfectly to objects description. The translation of an envisioned ABM in OO is therefore often seamless. Aside from the ease of implementation of ABMs, the benefits of OO programming are multiple: ease to maintain, to develop, to communicate, to split into team work packages, to modularize... at the cost of being slightly more demanding in processing power but this trade-off falls short in significance compared to the optimization routines often performed at each timestep when agents have to make decisions. Also, programming languages offer OO specific "services" such as automatic deletion of unused objects, polymorphism, etc., for a more flexible and understandable implementation.

To do this we relied on Java and Repast Symphony. Repast is a widely used free and open-source agent-based modeling and simulation toolkit. Three Repast platforms are currently available, namely, Repast for Java (Repast J), Repast for the Microsoft .NET framework (Repast .NET), and Repast for Python Scripting (Repast Py). Each of these platforms has the same core features. However, each platform provides a different environment for these features. Taken together, the Repast platform portfolio gives modelers a choice of model development and execution environments. Repast Simphony(Repast S) extends the Repast portfolio by offering a new approach to simulation, development, and execution. The Repast S runtime is designed to include advanced features for agent storage, display, and behavioral activation, as well as new facilities for data analysis and presentation.

4.1 Antibody

The antibody is the key agent of our idiotypic network. It is part of an immune system and is characterized by a type and an hValue (as described by Parisi). It can be alive or dead and in a state of balance or not. Not having the possibility to calculate the state of equilibrium with the entropy of the system we tried to simulate this algorithm with a class created specifically by us, called EquilibriumDataStructure. An Object Oriented representation of an Antibody agent is:

```
public class Antibody {

      private boolean alive;
      private int type;
```

```
        private double hValue;
        private EquilibriumDataStructure eq;

}
```

This agent at each step will influence all the other antibodies present within the idiotypic network by updating their hValue, and changing their state from alive to dead or viceversa according to the value of their hValue after being in turn affected by all other antibodies. In code this translates to:

```
@ScheduledMethod(start = 1, interval = 2, priority = 2)
public void step() {

        ImmuneSystem immuneSystem = this.getImmuneSystem();

        if (this.alive && !immuneSystem.isGlobalEquilibrium()) {

                this.getAntibodies().forEach(antibody -> antibody.
                    ↪ hValue += immuneSystem.getMatrix()[this.type
                    ↪ ][antibody.type]);

        }

}

@ScheduledMethod(start = 2, interval = 2, priority = 1)
public void changeStatus() {
        if (!this.getImmuneSystem().isGlobalEquilibrium()) {
                if (this.hValue < 0) {
                        this.alive = false;
                } else {
                        this.alive = true;
                }

                eq.addState(this.alive ? "A" : "D");
                this.hValue=0;
        }
}
```

4.1.1 EquilibriumDataStructure

EquilibriumDataStructure is, as the name suggests, the class that allows us to calculate whether a single agent is in a state of equilibrium or not. To do this, having to find a compromise between performance and effectiveness, we decided to use a simple pattern matching algorithm rather than more sophisticated systems using neural networks. Operation is fairly intuitive. This data structure stores the last n states (decided by the user) in which the agent was found as a string, associating the character 'A' to indicate the fact that the agent is alive and 'D' and indicate that this is dead. If the same letter is repeated n times, or if they alternate n times, the agent will be in a state of equilibrium.

In code the data structure will be as follows:

```
public class EquilibriumDataStructure {

	private String lastStateString;
	private int maxLength;
	private boolean equilibrium;

	private Predicate<String> equilibriumPredicate;

	public EquilibriumDataStructure(int maxLength) {

		this.lastStateString = "";
		this.maxLength = maxLength;

		this.equilibrium = false;

		this.equilibriumPredicate = Pattern.compile(String
		    ↪ .format("A{%d}|D{%d}|(AD){%d}%s|(DA){%d}%s",
		    ↪ maxLength, maxLength,
				maxLength / 2, maxLength % 2 == 0 ? ""
				    ↪  : "A", maxLength / 2, maxLength
				    ↪ % 2 == 0 ? "" : "D")).
				    ↪ asMatchPredicate();
	}

	public void addState(String state) {
		lastStateString = lastStateString.concat(state);

		if (lastStateString.length() > maxLength) {
			lastStateString = lastStateString.substring(
			    ↪ lastStateString.length() - maxLength,
			    ↪ lastStateString.length());
		}
	}

	public boolean isEquilibrium() {
		this.equilibrium = equilibriumPredicate.test(
		    ↪ lastStateString);
		return this.equilibrium;

	}

	public void reset() {
		this.lastStateString = "";
	}

	public boolean isEquilibrium() {
		return equilibrium;
```

```
        }
}
```

4.2 Immune System

The immune system in low-level vision almost performs an environmental function. Seeing it at a higher level, it plays an active role in the simulation. In order to better express it we decided to create both views within our simulation.

The immune system at the time of creation randomly generates the interaction matrix between antibodies. It can be in two states: balanced or not, and infected or not. An Object Oriented representation of the immune system is:

```
public class ImmuneSystem {

      private boolean globalEquilibrium;
      private double[][] matrix;
      private boolean isInfected;
      private int maxEquilibriumStateLength;

      private List<Integer> infectionTimes;
      private int currentInfectionTime;

      public ImmuneSystem(int antibodyTypeCount, int
         ↪ maxEquilibriumStateLength) {
            this.globalEquilibrium = false;

            Uniform uniform = RandomHelper.createUniform(-1,
               ↪ 1);

            this.matrix = new double[antibodyTypeCount][
               ↪ antibodyTypeCount];
            for (int i = 0; i < antibodyTypeCount; i++) {
                  for (int j = 0; j < i; j++) {
                        double initialValue = uniform.
                           ↪ nextDouble(); // value between -1
                           ↪  , 1
                        matrix[i][j] = initialValue;
                        matrix[j][i] = initialValue;

                  }
            }

            this.isInfected = false;
            this.maxEquilibriumStateLength =
               ↪ maxEquilibriumStateLength;

            this.infectionTimes = new ArrayList<>();
            this.currentInfectionTime = 0;
      }
```

At each step, it will do nothing but check if each antibody present in the idiotypic network is in a state of equilibrium or not. if and only if all the antibodies are in equilibrium at the same time then it will itself be in a state of equilibrium.

It will also play the role of reactive agent in our high-level vision. In fact, it will react to the presence of an antigen of a certain type ending its state of equilibrium and considering itself 'infected', it will therefore create an antibody capable of being able to counter that type of antigen in case it had never encountered such a threat. In code:

```
@ScheduledMethod(start = 1, interval = 2, priority = 2)
public void checkEquilibrium() {
	Context<Antibody> context = ContextUtils.getContext(this
	    ↪ );
	this.currentInfectionTime++;
	this.globalEquilibrium = context.getObjectsAsStream(
	    ↪ Antibody.class).allMatch(agent -> agent.getEq().
	    ↪ updateAndGetEquilibrium());
	if (this.globalEquilibrium && this.isInfected) {
		this.removeAntigen();
		this.infectionTimes.add(currentInfectionTime);
		this.isInfected = false;
	}

}

@Watch(watcheeClassName = "it.unicam.dcc.idiotypicnetwork.
    ↪ agent.Antigen", watcheeFieldNames = "moved", query = "
    ↪ within_moore␣1", whenToTrigger = WatcherTriggerSchedule.
    ↪ IMMEDIATE)
public void reactToAntigen() {
	this.getAntibodies().forEach(antibody -> antibody.getEq
	    ↪ ().reset());
	this.globalEquilibrium = false;
	this.isInfected = true;
	this.currentInfectionTime = 0;

	Antigen antigen = this.getAntigen();
	if (antigen != null && antigen.getType() < this.matrix.
	    ↪ length) {
		Context<Antibody> context = ContextUtils.
		    ↪ getContext(this);

		this.addNewAntibodyToMatrix();
		context.add(new Antibody(antigen.getType(), this.
		    ↪ maxEquilibriumStateLength));
	}
}
```

4.3 Antigen

An antigen is characterized by a type. The antigen does not play an active role in this simulation. For graphic reasons we have chosen to bring it closer to the immune system and infect it when these come into contact, but at the level of business logic its presence inside the body would be enough to trigger the immune system's processes in defense .

```
public class Antigen {

int type;
public Grid<Object> grid;
boolean moved;


public Antigen (int type, Grid<Object> grid) {

      private int type;
      private Grid<Object> grid;
      private boolean moved;

}

@ScheduledMethod(start = 1, interval = 1)
public void step() {
      ImmuneSystem immuneSystem = this.getImmuneSystem();
      if(immuneSystem.isGlobalEquilibrium()) {

            GridPoint myPoint = grid.getLocation(this);
            GridPoint isPoint = grid.getLocation(immuneSystem)
               ↪ ;

            float deltaX = isPoint.getX() - myPoint.getX();
            float deltaY = isPoint.getY() - myPoint.getY();
            double angle = Math.atan2( deltaY, deltaX );


            grid.moveByVector(this, 1, angle);
            this.moved = true;
      }
}
```

4.4 External Agent

Finally, we have an agent that we entered simply for simulation purposes. It just randomly creates a new antigen whenever the immune system is not infected and is in a state of equilibrium. It will also check that the immune system is being attacked by only one antigen at a time. In code:

```
@Watch(watcheeClassName = "it.unicam.dcc.idiotypicnetwork.
    ↪ agent.ImmuneSystem", watcheeFieldNames = "
    ↪ globalEquilibrium", whenToTrigger =
    ↪ WatcherTriggerSchedule.IMMEDIATE)
public void createAntigen() {

      Context<Antigen> context = ContextUtils.getContext(this)
         ↪ ;

      if (context.getObjectsAsStream(Antigen.class).count() ==
         ↪  0) {

            if (RandomHelper.nextDoubleFromTo(0, 1) < this.
               ↪ newAntigenPercentage) {
                  context.add(new Antigen(RandomHelper.
                     ↪ nextIntFromTo(0, antigenTypeCount - 1),
                     ↪ this.grid));
            } else {
                  int antigenType = antigenTypeCount;
                  context.add(new Antigen(antigenType, this.
                     ↪ grid));
                  antigenTypeCount++;
            }
      }
}
```

5. Simulation

The simulation can be started through the GUI provided by Repast, when we run the model from Eclipse with the "IdiotypicNetwork Model" run configuration. This GUI will contain a scenario tree, that takes as input the configuration stored in the *scenario.xml* file, inside the *IdiotypicNetwork.rs* folder, which in general contains every detail about the model configuration. In particular, the scenario we defined, contains three entries:

- An entry that specifies the used `ContextBuilder` implementation. In our case the used implementation is stored inside the java class `IdiotypicNetworkBuilder`;
- An entry that represents the display for the low abstraction level visual, whose details are contained inside the file *low-level.xml*;
- An entry that represents the display for the high abstraction level visual, whose details are contained inside the file *high-level.xml*;

Thanks to this configuration, when we load up the model we can have a simultaneous representation of the two different abstraction levels on screen easily. The visual result of this implementation can be viewed in the figure 5.1.

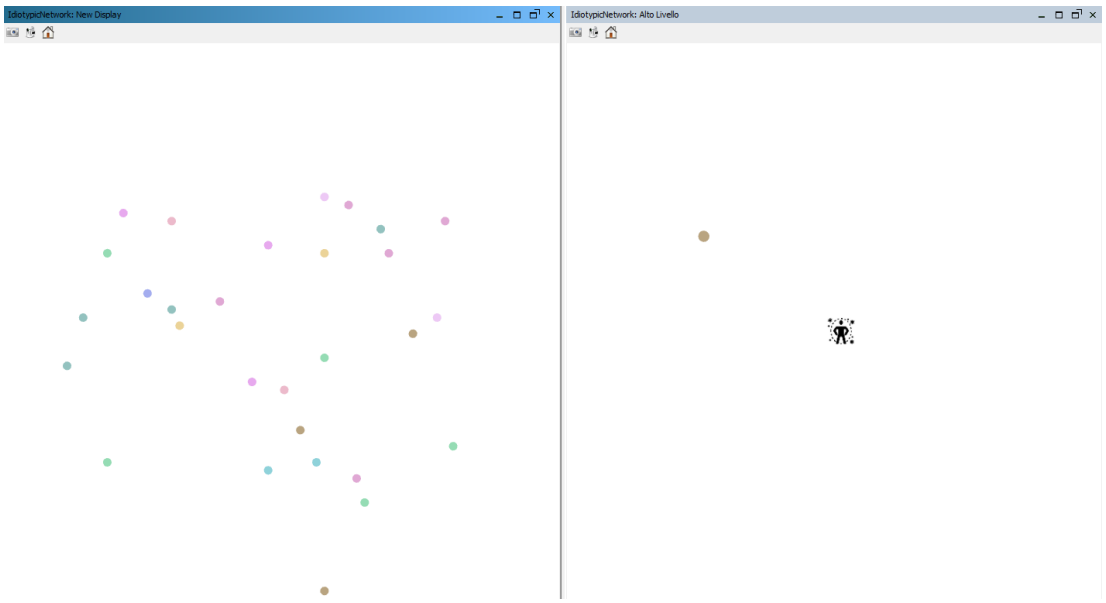


Figura 5.1: Two tabs view of the abstraction levels of the IdiotypicNetwork Model

The simulation provides also a list of parameters that can be configured manually through the repast GUI, in the parameters view tab. Such feature is shown in the

figure 5.2. Outside of the purely aesthetic configuration parameters, like *Grid Height* and *Grid Width*, that only define the size of the low level grid in which antibodies are shown, the parameters control interesting antibody properties, like the initial type count (*Antibody Type Count*), the maximum amount of initial antibodies of each type (*Antibody Max Amount Per Type*), and the length of the string used to define if an antibody is in a state of equilibrium (*Antibody Equilibrium Max Length*). Finally, there is a parameter that defines, every time an antigen is inserted inside the system, the percentage of it being a new antigen type or an old type already known by the immune system (*New Antigen Percentage*).

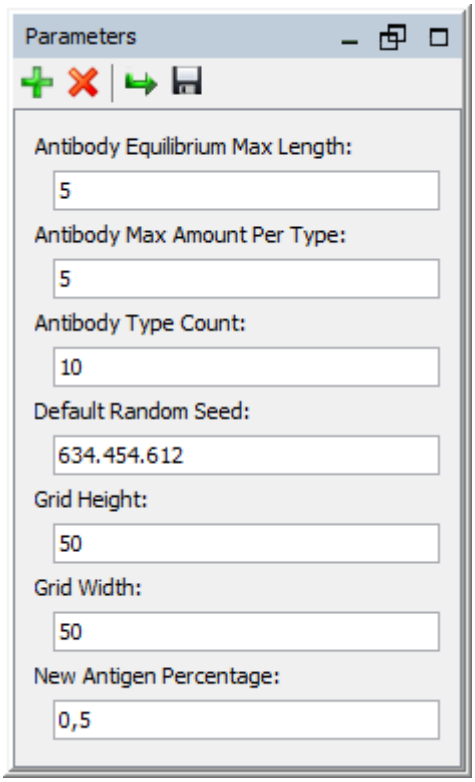


Figura 5.2: Simulation parameters of the IdiotypicNetwork Model

During the simulation we can see the two displays progressing in parallel autonomously. We decided to follow this dual approach to provide a better visualization for what is exactly happening on both stages of the system, the one internal to the immune system in an idiotypic network, represented by the antibodies concentrations and their influences between each other and the one external to the immune system, where the antigen enters the organism to defend. In the following sections we will inspect in detail how both visualizations work.

5.1 Antibody display progression

At the beginning of a simulation, all the antibodies that have been added to the context by the `IdiotypicNetworkBuilder` context builder, are represented as small colored circles. Each antibody type has a color mapped to it. Such color mapping is stored inside the java class `ColorTypeMapping`.

Another important things to notice is that in the first tick, every antibody that has been created is currently alive. This situation, as we will see in the following paragraph, will change tick after tick. This is caused by the fact that at the start of the simulation the value of `hValue` for each instance of `Antibody` present in the context is still equal to 0.

An example of the way the low level display is visualized in the Repast GUI can be found in the figure 5.3

In the following ticks, the whole immune system is trying to achieve a state of equilibrium. All the antibodies contained inside the the context are cooperating with each other, allowing or denying the production of other antibodies in the way described by the immune system interaction matrix. In this stage of the simulation a part of the antibodies may disappear for the screen, this represents the fact that their production has been inhibited by other antibodies. An example of this behaviour can be found in the figure 5.4, which represents the state of the system described initially in the figure 5.3 after a few ticks.

After a certain amount of ticks, the antibodies start finding their equilibrium. In particular, an antibody that is in equilibrium state is marked by a green border, independently from its being inhibited or not. So in a state in which the immune system is in equilibrium, we would see some empty green circles, which represent inhibited antibodies in an equilibrium state and colored circles with borders, which represent working antibodies that are in equilibrium state. An example of an immune system in a state of equilibrium can be found in the figure 5.5;

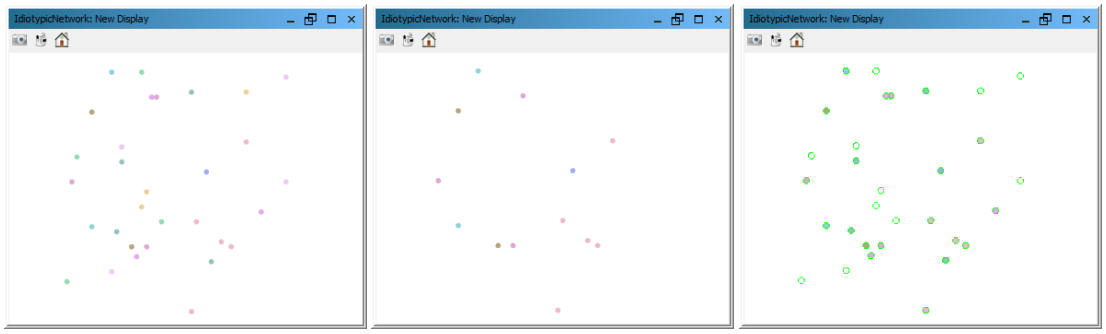


Figura 5.3: Initial state of the antibodies

Figura 5.4: State of the system with inhibited antibodies

Figura 5.5: State of a system in equilibrium

5.2 Immune system display progression

In the first tick of the simulation, in the display containing the high level visualization of the whole system, we can see an immune system, represented by an human icon, and a circle far away from it, which is an antigen. Such antigen is colored with the same color of the antibody designed to fight it. Since such antigen is far away from the immune system, we can say that the immune system is currently not infected. An example of the visualization of a non infected immune system can be seen in the figure **??**.

When an immune system is in equilibrium the antigen will start moving tick after tick always nearer to the immune system, to infect it. Once such antigen reaches the immune system, the latter becomes infected, and starts the reaction to expel such antigen from the body. When this event happens, we could notice that, in the screen where the antibodies are visualized, all the equilibrium states of the antibodies would be reset and that a new antibody, colored the same as the infection antigen, would appear on the grid. Once the whole immune system finds a new equilibrium state, the antigen would be correctly expelled and a new one would be inserted in the grid. An example of the visualization of an infected immune system can be seen in the figure 3.3.

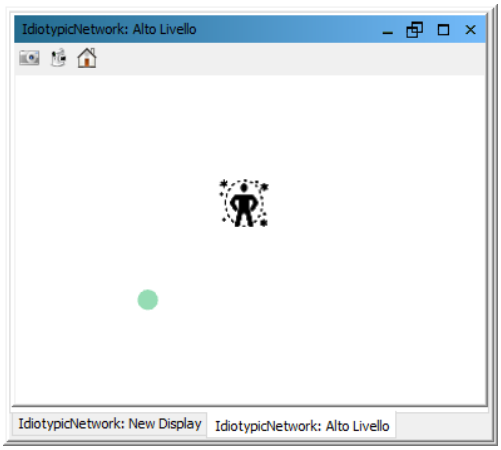


Figura 5.6: An example of not infected Immune System

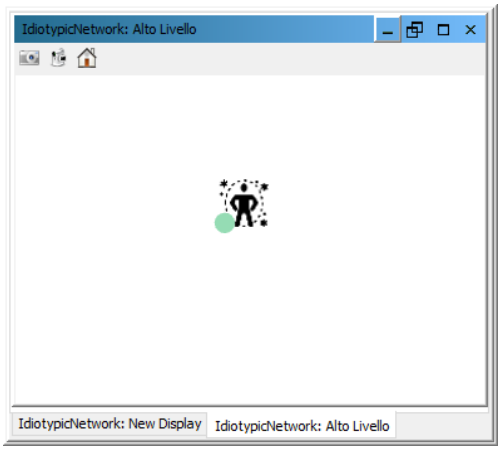


Figura 5.7: An example of infected Immune System

5.3 Collected data

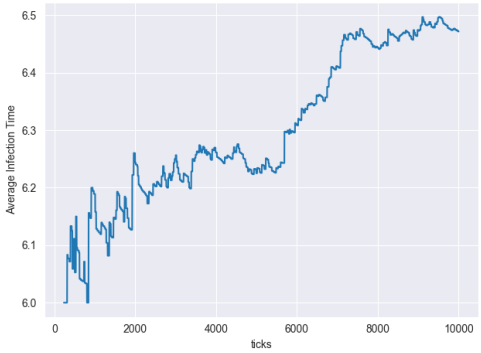
Associated to the context of the model, there are other two entries, that define a way to capture data from the simulation and to store them inside data files. In particular:

- *dataset.xml* defines a dataset that, in each tick of simulation, saves from the instance of `ImmuneSystem` the amount of recognised antigen types and the average duration in ticks of an infection;
- *filesink.yml* defines the saving of the data obtained from the dataset explained above inside a file named *antigen_type_count.csv*.

With the data generated by the simulation, we can create various types of graphs to make analysis on the outcomes of the simulations, based on the given initial parameters. This aspect has been unluckily not analized well yet however. For now, we created a couple of graphs that express the situation of the ImmuneSystem, using the default parameters, that are found in the figure 5.2, too.

In particular, an example of these graphs can be found in the figures 5.8 and 5.9. In the first one we can note how the average infection time slowly increases over time. This can be probably caused by the fact that an higher amount of antibody has a more difficult time in achieving equilibrium than a small amount. In the second graph we

can see the constant growth of the types of antigen recognised by the immune system. This graph is coherent with the fact that, by the parameters, in average one antigen out of two will be a new and unknown one.



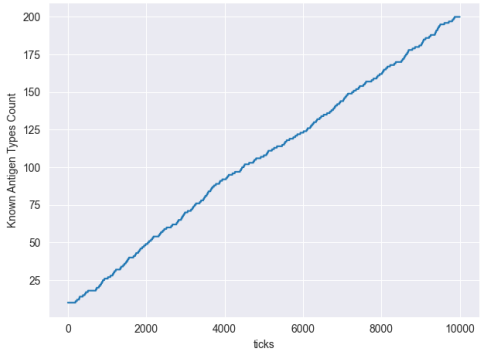


Figura 5.8: The average infection time in ticks over time

Figura 5.9: The amount of recognised antigen types over time

6. Conclusions

In conclusion, the final goal of our project was to try to simulate a sort of immunological memory, a goal that has been achieved. In fact, using a S_n state of our MAS and any S_m state with $m \geq n$ we can interpret the type of antigens present within the idiotypic network. In the moment that the organism has come into contact with new antigens it will present new antibodies created to fight that threat. The same result can be obtained simply by comparing the length of the Parisi's matrices of the two immune systems belonging to S_n and S_m.

The next step in order to further improve our simulation would be to manage in a more sophisticated way (perhaps following a topological approach based on entropy) the search for the state of equilibrium of the idiotypic network. In fact, in this simulation we searched for equilibrium using a data structure created ad hoc, following a very simplistic approach based on pattern matching.

At this point we could continue by gradually adding a greater level of detail, perhaps adding the innate immune defense barrier that makes the first skimming when an antigen comes into contact with the body.

It would also certainly be interesting to test what was created with real data, coming from biologists and experts in the sector. Currently our simulation generates a matrix that reflects the properties described by Parisi in his paper but generated in a totally random way. It might be interesting to see how an idiotypic network behaves knowing the true values of influence between antibodies.