Tool for creation and analysis of two dimensional cellular automata

Sunday 26th March, 2023 - 20:35

Jason Billard
University of Luxembourg
Email: jason.billard.001@student.uni.lu

This report has been produced under the supervision of:
Nicolas Guelfi
University of Luxembourg
Email: nicolas.guelfi@uni.lu

Abstract

Cellular automata have been known for the large variety of ways they can behave, even with very limited amount of rules. They have been used as means to simulate various physical models, and other real life systems. The goal of this paper is to enable an intuitive way to create new cellular automata by developing two programs. Incidentally, the paper proposes a way to approximate the distribution of the states on the grid as well as the ratio of cells that change state at some time-step. This paper is done as BSP for the first semester of the BICS program, and is submitted with the associated program and video reports in english and in french.

1. Introduction

The main motivation for this BSP is the high interest in cellular automata. This interest comes from videos of people showing their work with them, ranging from very complex and beautiful creation in Game of life [1] to creating a complete computer that runs machine code with the Wireworld automata [2]. Both of these automata are based on very simple rules, yet the possibilities of what can be achieved with them are endless.

Cellular automata are not limited to two dimension and discrete number of states. They can be applied to a large number of fields. An concrete example of their flexibility would be where their use in combination of neural networks to recreate images [3].

Other more practical applications would be simulating the evolution of wildlife to assist fire management [4] or using them in protein bioinformatics [5]. It is onerous to grasp how many ways cellular automata can be applied, however their modus operandi is remarkably simple.

On the amusement side, there exists a website called Pixels Fighting, which runs a cellular automaton with two state. Seeing what state becomes more dominant on the grid after some time is interesting to observe. And being able to compute this as well as the ratio of the cell that change each step is what has been done in the scientific deliverable.

Software to create cellular automata exist. An example would be Golly, which allows cellular automata in 3 dimensions, on hexagonal grids, very complex rules, etc... However, this complexity of this software does is not fit for someone who starts in this domain. Thus, as technical deliverable, an editor where one can make cellular automata with an intuitive interface was made. To complement this, a simulator where the automata created in the editor can be loaded and run has been developed.

2. Project description

2.1. Domains

2.1.1. Scientific

The domain of the scientific deliverable is automata theory. From [6]:

> *Automata theory is the study of abstract machines and automata, as well as the computational problems that can be solved using them.*

In other words, this means that automata theory consists in analyzing models which can perform different logical, arithmetical operations or other computing process. Cellular automaton fall under this domain, which is the topic of this BSP.

2.1.2. Technical

The technical work touches two different domains. The first domain is simulation. Simulation consists in computing

the evolution of a model. By definition [9], a simulation consists in recreating the behaviour of real world systems as close as possible. However, in this technical deliverable, the simulation computes how some type of abstract automata evolve, cellular automata. The simulator is thus more generic, and can simulation a range of different automata of a specific type.

The other domain of this technical work is object oriented programming [10]. In this style of programming, the program is structured around data and objects. To create an object oriented program, the precise type of objects that will be used have to be defined. Then they are implemented as classes, with attributes containing data related to each object, and methods which are function that manipulate the corresponding object.

2.2. Targeted Deliverables

2.2.1. Scientific deliverables

The targeted scientific deliverable consists in calculating the activity and the state distribution of a cellular automata by analysing the rules.
In chapter 2 of [7], they describe a cellular automata as having four different components: a grid, a neighbourhood, the states of a single cell and a local function. In this scientific deliverable, two of these components are fixed. The grid will always be a two dimensional square grid, and the neighbourhood will be the Moore neighbourhood [8]; the central cell and it's eight surrounding cells. Given an a cellular automata with an arbitrary number of possible states and a local function (the rules of that cellular automata), can one calculate the activity and the distribution of the states for any timestep?

If this work is successful, it will be possible to see how a cellular automata behaves. Specifically, it will be possible to know if some cellular automata die down after a few time steps (if the activity goes to 0), or continue evolving over a longer period. It will also be possible to find out what state will be more present after a certain amount of time, or if the distribution stays similar over time.

2.2.2. Technical deliverables

The targeted technical deliverable is a software which allows the user to create and run cellular automata. The software consists in two distinct parts, the simulator and the editor.

The simulator is the part of the software that runs the cellular automaton. Here, the user can see the simulation of the running cellular automaton in a grid and can interact with it in multiple ways.

There are options to manipulate the grid, like shuffling or changing some cells individually. If the user arrives at some grid states that is interesting, he can save the grid, and open it again to try different manipulation.

To have an overview of the running simulation, two graphs will be displayed. One of which will display the activity (number of cells that changed state in one time step) over time. And the other one will show the state distribution on the grid over time.

The other part of the software is a cellular automata editor. In this program, the user can create his own cellular automata with the help of a user interface. The cellular automaton created in this editor can be saved in a file. This is also the file that the simulator will load to define the current running automaton.

Overall, this software will allow the user to work with cellular automata. It will be a tool which will help analyse this kind of system. The reason for this technical deliverable is to support the scientific deliverable. With the tool created for the technical part, the scientific part can be verified.

3. Pre-requisites

3.1. Scientific pre-requisites

To start the scientific work of this BSP, it is necessary to understand how cellular automata work. It is explained again in the deliverable, but already having in insight on cellular automata is greatly recommended. A cellular automaton is a system which changes the states of cells on a grid depending on their neighbours. In two dimensions, each cell will have 8 neighbours. The local function takes in 9 states, the state of the cell of interest and its neighbours, and outputs the state that this cell will have in the following time step.

To understand why this scientific deliverable is written, it is important to understand that cellular automata can, even with very simple rules and a limited amount of states, behave in very complex ways. An example of this would be Conway's Game of Life, which can create patterns and shape that move around the grid, propagate, expand, or become oscillators. This cellular automata however, only contains two possible states are a handful number of rules.

3.2. Technical pre-requisites

For the pre-requisites of the technical deliverable, it is first recommended to have a good understanding Java, or a similar language like c sharp. However it is not imperativ, having enough experience with object oriented programming is sufficient. To be more exact, if one understands how

different data structures work, or has used inheritance and polymorphism before, it will be possible to understand the concepts that will be used in the software. Not all these concepts are necessary to understand the written part of the technical deliverable, but they are required to understand the entire software.

Apache Netbeans 11.1 is used as the integrated development environment to create this technical deliverable. For the user interface, the editor of Netbeans to design Swing ui-s will be utilised.

During the technical project, no external Java library is used. However, some internal library will be, like the aforementioned javax.swing to create the interface or to use timers.

An understanding of the class diagrams (UMLs) is also required.

4. Computing the state distribution and the activity of a cellular automaton

4.1. Requirements

The requirement of this scientific deliverable is to create two functions (f_d and f_v) and to explain with a precise scientific text how these function work.

The first function f_d is the state distribution function, and it computes the state distribution of the next time step given the current state distribution. The second function f_v is the activity function, which computes the activity to the next time step with respect to the current state distribution. The state distribution and activity are defined mathematically in the production section.

The text of this deliverable should be written in different parts. The explanation is broken down into smaller problems for an easier readability. The smaller problems are solved step by step mathematically, and the explanations should be intuitive, concise and comprehensible.

4.2. Design

In this subsection, the steps taken to create the functions described in requirements are listed. The production is written in a mathematical way, and uses algebra, set theory and probabilities.

Cellular automaton are defined mathematically at the start of the scientific deliverable. This is necessary for a precise mathematical notation in the explanations.

4.2.1. The state distribution function

The explanation on how the state distribution function works is divided in 5 parts. First the state distribution is defined mathematically, then there are 4 subsections. The first subsection explains the concept of a batch, and derives some formulas about them. The second subsection describes how to compute the probability of a cell in one state going to another state. This part only computes an approximation, and the reason is also explained. The following subsection explains how to compute the state distribution of a single state, given the current state distribution and the previously computed probability. Finally, the last part explains how this function can be viewed as a matrix transformation.

4.2.2. The activity function

In the activity function section, the activity is defined, and then the activity function is expressed with some idea used in the state distribution function. Because it relies on same ideas, the activity function is also an approximation.

4.3. Production

4.3.1. Cellular Automata

The following elements are notions required to define cellular automata mathematically.

- *The time* is the notion that events happen after each other. The time t corresponds to the number of time steps passed. A time-step contains all the events happen from t to $t+1$.
- *The grid* G is a two dimensional grid of squares, which contains all the cells. Cellular automata can be defined on a more generic grid type and the calculation done in this work can be generalized, the choice of a two dimensional grid is for simplification reasons.
- *A cell* c is a cell of the grid and has one state. The cell $c_{i,j}$ is the cell on the grid at column i and row j.
- *A neighbourhood* n is the set of cells that are neighbours to some cell. n_c denotes the neighbourhood of an arbitrary cell c and the neighbourhood of the cell $c_{i,j}$ is noted as $n_{i,j}$. The neighbourhood is defined with the Moore neighbours [8], so the neighbours of $c_{i,j}$ are $n_{i,j} = \{c_{i-1,j-1}, c_{i,j-1}, c_{i+1,j-1}, c_{i-1,j}, c_{i+1,j}, c_{i-1,j+1}, c_{i,j+1}, c_{i+1,j+1}\}$.
- *A state* s is an element of the set of all the possible states S. The size of S is defined by the cellular automaton. To differentiate the states, let's note s_i (with $1 \leq i \leq |S|$, $|S|$ denotes the number of possible states) a unique state of S. The state of a cell c is $s(c)$. To specify the time, the notation is $s_{t=..}(c)$.
  $n(c, s)$ is the function that returns the number of neighbours of c in the state s, this is computed with $n(s, c) = |\{v | \forall v \in n_c \text{ where } s(v) = s\}|$.
- *The local function* of a cellular automata $f_a(c)$ computes the state of c at $t+1$. The local function depends on the number of neighbours in some state and not their

position.
According to this, let the cells c_1 and c_2 be two different cells. If for every state $s \in S$, $n(c_1, s) = n(c_2, s)$, then $f_a(c_1) = f_a(c_2)$.
The function is applied in the following way: $s(c_{t+1}) = f_a(c_t)$ where c_t is a cell at time t and c_{t+1} is the same cell on the next time step. Another way of writing this is give as parameter a neighbourhood and the state of the cell $f_a(n, s)$, $s(c_{t+1}) = f_a(ct) = f_a(n_{c,t}, s(c_t))$

4.3.2. The state distribution function

The distribution of states D represents how present the different states are on the grid. It is a list containing a value between 0 and 1 for each state, representing the ratio of the cells in this state compared to the grid. D_s corresponds to the distribution value for the state s, meaning the ratio of cells in state s compared to all the cells. Mathematically, we can write:

$$D_s = \frac{|\{c|\forall c \in G \text{ and } s(c) = s\}|}{|G|}$$

To be able to define the distribution at different time, D_t will denote the distribution of the states on the grid at the time t. The state distribution function f_d is a function that takes the current state distribution, and returns the state distribution at the next time step; $D_{t+1} = f_d(D_t)$.

Batch

To help find the function to approximate the state distribution, the concept of a batch is created.

A batch b is a set of neighbourhood, where the number of cells in each state is the same. To represent a batch, a list with the size $|S|$ can be used. The entry at the index i is the number of cells in the neighbourhood in state s_i. This is noted as b_i. Since the total number of neighbours is 8, we have to have:

$$\sum_{i=1}^{|S|} b_i = 8$$

The reason why batches are used, and not directly the neighbourhood, is because there exists a lot more possible neighbourhood than batches. To compute the state distribution function later on, The local function on every possible batch will have to be computed, which is a lot less than the total number of neighbourhood.

The number of possible neighbourhood is $8^{|S|}$. It grows with the power of the number of states. To compute the number of possible batches, the following formula can be used: $\binom{|S|+7}{|S|-1}$.

The explanation of this formula is that this problem can be represented in distributing 8 cells to the $|S|$ states. Let "$*$" be a cell, and "/" be the sign that separates the cells between the different states. As example, suppose there are 5 states. Writing "$** / *** // *** /$" represents that the first state has 2 cells, the second state 3, the third none, the fourth 3 and the last none either. The expression has 4 "/"s because there are 5 states and $5 - 1$ divisions are needed to separate the cells "$*$". The expression to represent how the cells are divided between the state will always be $4 + 8 = 12$ characters long. The number of ways to divide the cells, corresponds in choosing 4 places out of 12 where "/" will be put, and the rest will be filled with "$*$", which will represent every way to divide the 8 cells to the 5 different states. The binomial coefficient formula can compute this $\binom{12}{4}$.

Applying this to choosing separating 8 cells to $|S|$ different states, we have:

$$|B| = \binom{8 + |S| - 1}{|S| - 1} = \binom{|S| + 7}{|S| - 1}$$

To calculate the size of a batch, it comes down to multiplying binomial coefficients. Take b_1 out of 8, which corresponds to taking b_1 of the 8 cells that will in the state s_1. Now take b_2 out of $8 - b_1$, then take b_3 out of $8 - b_1 - b_2$. This is repeated until $b_{|S|}$. These values have to be multiplied with each other. The mathematical expression that represents this computation is:

$$|b| = \prod_{i=1}^{|S|} \binom{8 - \sum_{j=1}^{i-1} b_{sj}}{b_{si}}$$

Probability of cell in one state going to another state

To break down the problem of computing the state distribution for the next time step, the probability of single cell going to another state comes up. Indeed, when the probability of each cell going to another state is known, computing D_s comes down to summing the probabilities of each state going to the state s.

To compute the probability of a cell in state s_i to go in the state s_j, take the number of neighbourhood n such that $f_a(n, s_i) = s_j$ (noted $N_{i \to j}$) and divide it by all the possible neighbourhood. It is in this section that the approximation occurs, because some neighbourhood are less likely to occur than others, but taking this into account would add another layer of complexity. The reason why some neighbourhood are less likely than others is because of the state distribution. If there are very little cells in state s, then neighbourhoods that have the state s will also be less frequent to occur.

The number of possible neighbourhood ($8^{|S|}$) is known. To compute $N_{i \to j}$, take all the batches b, so that if $n \in b$, $f_a(n, s_i) = s_j$, and write this as $B_{i \to j}$. These batches can be found with the rules of the cellular automaton. Then with the previously defined formula, the number of neighbourhood that

are in a batch can be calculated. Thus, $N_{i\to j}$ can be computed;

$$N_{i\to j} = \sum_{b\in B_{i\to j}} |b|$$

Let $p_{i\to j}$ be the approximation of the probability of a cell in one state to another state. This probability is expressed with:

$$p_{i\to j} = \frac{N_{i\to j}}{8^{|S|}}$$

State distribution of a single state

Computing $D_{s,t+1}$ requires D_t, which is known according to our requirements. By definition:

$$D_{s,t+1} = \frac{|\{c|\forall c \in G_{t+1} \text{ and } s(c) = s\}|}{|G_{t+1}|}$$

This is equal to summing the ratio of the different states going to the state s. For every state, take the probability of a cell being in that state and multiply it by the probability of this cell going to the state s:

$$D_{si,t+1} = \sum_{j=1}^{|S|} D_{sj} \times p_{j\to i}$$

State distribution function as matrix transformation

Let M be a square matrix of size $|S|$ with $m_{i,j} = p_{j\to i}$, and take the distribution D as a column vector where at row i there is D_{si}. Multiplying M with D gives a column vector. The result at the row i gives:

$$\sum_{j=1}^{|S|} D_{si} \times p_{j\to i}$$

This is precisely $D_{si,t+1}$. According to this, we have $D_{t+1} = M \times D$. Which is the state distribution at the next time step. The state distribution function is thus: $f_d(D) = M \times D$

The advantage of this method is that the matrix M only has to be computed once at the start, and then computing the next state distribution comes down to a single matrix multiplication. The downside is that this is an approximation, that assumes that each neighbourhood is always equally likely.

4.3.3. Activity function

The activity A is the ratio of cells on the grid8 changing to another state on the next time step.

$$A = \frac{|\{c|\forall c \in G \text{ and } s(c_{t+1}) \neq s(c)\}|}{|\{c|\forall c \in G\}|}$$

The activity function f_v takes the state distribution and returns the activity.

Another way to view the activity, is the state distribution, but the probability of the cell staying the same, and then summing all the values. This works because the state distribution represent all the cell and then the probability of the states staying in the same states is removed, keeping only the cells that changed states. To "remove" the probability of the states staying in the same state, it suffices to set the entries in the diagonal of the matrix M to 0, because these correspond to the probability of a cell in state going to the same state $p_{s\to s}$. Note the matrix with the diagonal of 0 the matrix M'. The function f_v can be expressed with:

$$f_v(D) = \sum_{i=1}^{|S|} (M' \times D)_i$$

4.4. Assessment

For the assessment of this project, the limitations of the developed formulas will be explained. Additionally, what could have been done better will be covered and the requirements of this deliverable will be verified.

First of all, it is necessary to mention the flaw that the function f_d and f_v are only approximations. The approximation comes from the fact that some neighbourhood were more likely than others, but this was not taken into account. This, the functions are only correct for the very first time step where the state distribution is the same for all state and this condition is true.

The second limitation of this approach is that it is necessary to have the state distribution of a previous time step to compute the next one. This means that it is more a sequence that relies on the previous value.

The possible improvement are removing the approximation. Which means the likelihood of a batch should be taken into account. This is not impossible, the likelihood could be calculated with a multiplication of the batch and the state distribution, which would give how much more likely with this distribution than with a uniform distribution. Doing this however would mean rebuilding the matrix for every timestep. This is also the reason as to why this approach was not explained in this scientific deliverable.

According to the requirements, this work was successful. The text explaining how the functions work is a precise text, and answers the question.

5. Cellular Automatool

5.1. Requirements

The technical deliverable consists of two separate complementing programs. The first program is a cellular

automata simulator, and the other is a cellular automata editor.

Both of these program are written in Java and use swing user interfaces. They are also both written in an object oriented programming style.

5.1.1. The simulator

The simulator is the software that runs the cellular automaton. Its main purpose is to run the cellular automata on a grid. However, there are different features available to manipulate and analyse this simulation.

First of all, it is possible to speed up or slow down the simulation with a slider. There is a limit to how slow the simulation can run. However, it is possible to pause and resume the simulation in case a specific state of the grid wants to be looked at closely. If wished, the grid can be saved in a text file and loaded again later. It is only possible to load a file if the number if state of the automata that ran on that grid is not greater than the number of states of the running automata.

If the user wants to skip a large number of steps a very specific number of step, the user has the option to enter a number and compute this amount of steps, without showing what the grid looks like for each time step.

Each time the grid is updated, the program collects data about the grid. It counts the number of cells in each state and computes the state distribution. Additionally, it counts the number of cells who changed state during that update (the activity). This data, both the cell distribution and the activity, are displayed as graphs on the right side of the program. This data can be cleared and saved as a csv file, if the user wants to use an external spreadsheet program like excel to analyse it.

The program contains features for manipulation of the grid. It has an option to shuffle the grid, which sets each cell to a random state of the automaton. The user can also select a state on a list, and set all cells to that state. It is also possible to zoom and pan around the grid, and click a specific cells to only set these cells to the selected state.

5.1.2. The editor

The editor is the program that facilitates the creation of cellular automaton. With the interface of this program, the user can change the number of states there are and add and remove rules.

For each rule that is added, there are three different parameters to set. The cell state, which is the state of the cell for which this rule applies to. The state in which the cell changes to if the condition of this state holds. And finally the condition, which most of the time is related to the neighbours of that cell.

To set the condition, there are several options that the user can choose from:

- *less than*: This condition is true if there are less neighbours in some state than some number.
- *more than*: This condition is true if there are more neighbours in some state than some number.
- *equal*: This condition is true if the number of neighbours in some state is equal to a specific value.
- *and*: This condition is true of the two inner condition are true.
- *or*: This condition is true of at least one inner condition is true.
- *true*: This condition is always true.

The automata files are saved as XML files, which is easily human readable.

5.2. Design

Developing relatively complex programs like the simulator, the process was to start with an easy program and build up, adding features step by step. In most cases, creating a program in this manner works well, since there is no moment where a lot of code has to be tested at the same time. Small pieces are added one after each other, and each time the program can be tested. Making a program by writing the entire code, and only running it once at the very end hoping it works, is likely to end in hours wasted in debugging. Incidentally, if the development had to stop, the program would at least have some functionalities that would certainly be working, in contrast to creating the entire program at once.

In OOP, defining the classes in which the program will be divided is a key point point in the development of the program. In this Design section, the final classes that are used to implement the main functionalities are presented; the existence of the different classes are argued and the interactions between the different objects are explained.

5.2.1. The simulator program

The core functionality of this entire technical deliverable is to run any cellular automaton on a two dimensional square grid and display it. The restrictions of the cellular automaton are precised in the requirements of the scientific deliverable. Since the editor is only a program that complements the simulator; creates the cellular automaton that are simulated, it's design and implementation will not be detailed.

Here is the list of the main classes (also see the class diagram in the appendix 2):

- Automata
- Simulator
- SimulatorDisplayer
- ModularAutomata
- Rule
- Condition
- MainFrame

Automata

The class Automata is a simple interface, but it's role is to represent all cellular automata. When a class implement this class, it means that these classes can work as cellular automata: they have a local function which can compute the next state of a cell if the current state of that cell and the number of neighbours in the different possible state that this cellular automata can have. In the program, this very simple interface only defines two function that have to be implemented, a function which returns the number of states that this cellular automata can have and the local function.

Simulator

The class Simulator is the class containing the grid, and updates it. It contains an instance of an Automata, and uses it to compute the next state of each cell in the grid. It also counts the number of cell that changed cells when the grid is updated, and tracks how many cells are in each state. This is needed for the two graphs that display the activity and the state distribution.

It contains other features, which enhances the interactivity with the grid like allowing to set specific cells to some states, set all the cells to the same state and randomize the states of the cells.

SimulatorDisplayer

The SimulatorDisplayer contains an instance of a Simulator, and displays it on as a grid. It contains a function that translates a position on the displayed grid to the index of the cell, which is used to set specific cells to some state. It allows panning, zooming. The main function remains to draw the grid with a Graphics2D instance. If this were a smaller program, and both the Simulator and the SimulatorDisplayer class were less complex, both could be merged into a single class. But here the separation of the graphics and the logic is made, to keep the structure of the program cleaner.

ModularAutomata

The only class that implements the interface Automata is this ModularAutomata class. This cellular automata works with a system of rules: when evaluating the local function, it looks if any of the rules that are in this ModularAutomata instance. The reason for the name of this class is because it is possible to add and remove Rules, and thus changing the local function of the cellular automata. This is the class that is customized and saved to an XML file in the editor, and loaded in the Simulator for simulating.

Rule

In the ModularAutomata, Rules have been mentioned as a way to save behaviours of cellular automaton. To be more precise, rules apply to cells of specific states, have a Condition that depends on the number of neighbours in the different states, and a state that the cell goes to if the condition is true.

Condition

Condition is again an interface. It has a single function: isTrue, which returns a boolean value whether the condition is true or not depending on the parameters. The parameters are an array of the number of neighbours in the different states, and the state of the cell.

There are multiple classes that implement this interface in different ways, which are used for a larger variety of rules.

MainFrame

The user interface is contained in the MainFrame class. It contains all the function that are called when the different buttons are called, or when the states of sliders are changed. It manages all the input of the user. This class contains an instance of the Simulator, and the user can interact with it.

5.2.2. The editor program

The editor is a program whose entire role is to be able to configure the ModularAutomata class. This program thus also contains all the classes of used in ModularAutomata, however, only the skeleton and not the function they implement since they would not be used. The entire difficulty of the editor was to make the interface usable and intuitive enough.

The editor has a main interface, called MainFrame, which contains buttons that allow to add or remove rules or states. To add a Rule, a new interface opens, and again to set the Condition of this rule. Since there exists conditions like Or and And, the condition interface can open recursively, which is why this editor uses a lot of interfaces and not only one window. With a single window, making an interface without pup-out windows where one could set a Condition in another Condition and so on is a difficult task.

5.3. Production

In this section, the most critical or interesting parts of the program's code are shown, and then explained it in depth. The reasons as to why the code has been written this way will also be explained. The usage of the program will also be shown.y

5.3.1. The update function

Arguably, the most important function of the entire program is the function which updates the grid. It evaluates the local function on all cells, and updates the grid. This function is part of the Simulator class, and requires the existence of 4 different variables and the automata:

- `int gridWidth`: number of columns on the grid
- `int gridHeight`: number of rows on the grid
- `int[][] grid`:
  A two dimensional array where `grid[i][j]` contains the state of the cell at row `i` and column `j`.
- `int[][][] gridCount`:
  A three dimensional array where `gridCount[i][j][s]` contains the number of neighbours in state `s` of the cell at row `i` and column `j`
- `automata`: The instance of the `ModularAutomata` that is currently running in the `Simulator`.

The array `gridCount` can always be computed with the `grid` array. When a value of this array is required, it would be possible to look in the `grid` array and count it then. When profiling the two different ways, the version where we store the counts in a three dimensional array performed better, so it was kept.

The update of the grid is done is several steps. First, the arrays that will contain the values for the next time step are instantiated with the appropriate size.

```
int[][] newGrid = new int[gridHeight][gridWidth];
int[][][] newGridCount = new
     int[gridHeight][gridWidth][automata.numStates()];
```

Then, two loops, one looping over the rows and the other looping over the columns, go over all the cells. For each cell, the next value of that cell is computed with the evaluate function of the `automata` (1). The new grid is then updated, as well as the new grid that counts the number of neighbours (2). This is done by using a function called `setCellValue`. This function will be explained later on.

```
for (int i = 0; i < gridHeight; i++) {
for (int j = 0; j < gridWidth; j++) {
  int nextState = automata.evaluate(
            gridCount[i][j], grid[i][j]);
  setCellValue(j, i, nextState,
            newGrid, newGridCount);
}}
```

After all the cells have been looped through, the current grids are set to the new grids that have been computed:

```
grid = newGrid;
gridCount = newGridCount;
```

This function also collects data about the state distribution and the activity, however those are not the most important part of this function. In its entirety, the update function is the following:

```
public void update() {
  //for stats
  int activity = 0;
  int[] cellCount = new int[automata.numStates()];

  //grid in next time step
  int[][] newGrid = new int[gridHeight][gridWidth];
  int[][][] newGridCount = new int[gridHeight][gridWidth][automata.numStates()];


  for (int i = 0; i < gridHeight; i++) {
     for (int j = 0; j < gridWidth; j++) {
       int nextState = automata.evaluate(gridCount[i][j], grid[i][j]);
       setCellValue(j, i, nextState, newGrid, newGridCount);

       //tracking stats
       cellCount[grid[i][j]]++;
       if (grid[i][j] != nextState)activity++;
  }}

  //updating grid
  grid = newGrid;
  gridCount = newGridCount;

  //saving stats
  stats.addStepValues(activity, cellCount);
}
```

The `setCellValue` function is used to set a the state of a cell a the grid, and also update the associated grid that counts the neighbour. It does this by modifying the entries of all the neighbouring state by adding 1 to the counter of the corresponding state. For simplification, the conditions that check that this value is not out of bound of the array have been removed:

```
public void setCellValue(int col, int row,
int state, int[][] grid, int[][][] gridCount){
   grid[row][col] = state;

   gridCount[row - 1][col - 1][state]++;
   gridCount[row + 1][col - 1][state]++;
   gridCount[row - 1][col + 1][state]++;
   gridCount[row + 1][col + 1][state]++;
   gridCount[row][col - 1][state]++;
   gridCount[row][col + 1][state]++;
   gridCount[row - 1][col][state]++;
   gridCount[row + 1][col][state]++;
}
```

5.3.2. Automata evaluation function

As mentioned in the design part, the evaluate function of classes implementing the `Automata` interface represent the local function of a cellular automata. The class `ModularAutomata`, which implements this interface, was described as having an evaluate function which loops through all the rules and applies the first whose condition apply. The code for this is straightforward:

```
@Override
public int evaluate(int[] neighbourStates, int cellState) {
   for (Rule rule : rules)
   if (rule.isApplied(neighbourStates, cellState))
      return rule.toState;

   return cellState;
}
```

5.3.3. Drawing the state distribution

A less important piece of code, but with an interesting solution, is used when displaying the graph of the state distribution. The graph is drawn with multiple polygons:

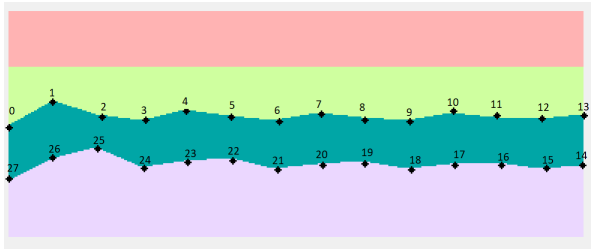

Fig. 1: A polygon of the graph with black dots on the vertices

Here, the `fillPolygon` function of the java `Graphics2D` class is used. It is used to draw the different polygons that draw display the different states. To use this

function, the coordinates of the vertices are given via two arrays, containing the x and the y coordinates respectively. The x-coordinates are the sme for each polygon, so the same array is used. For the y-coordinates, a two-dimensional array is used since there is an array for each state.

```
//number of vertices
int nVertex = activity.size() * 2;
//vertices coordinates
int[] polygonsX = new int[nVertex];
int[][] polygonsY = new int[numStates][nVertex];
```

The height of the polygon at each time step divided by the entire width of the graph represents how many cells of the entire grid are in that states. The sum the number of cells in each state is the number of cells on the entire grid. The corresponding calculation on this graph is to sum the height of all polygons at some time step, resulting in the height of the entire graph, which is always constant.

To find the x-coordinate of the vertices, the width of each time-step `dw` has to be computed. The vertex i and the i-last vertex have the same x-position, and it is simply multiplying n by `dw`. (`i` is the current time-step):

```
polygonsX[i] = polygonsX[nVertex - i - 1] = (int)(i * dw);
```

For the y-coordinates of the top vertex, the height of all the polygons above have to be computed. For this reason, a loop goes through all the states, and sums the height of that polygon to a variable `h`. The y-coordinates of the vertex at the top is `h`, and the vertex at the bottom of this polygon is `h`+ the height of the polygon. The entire function looks like the following:

```
public void showCellCount(Graphics2D g, int width, int
     height) {
//number of vertices
int nVertex = activity.size() * 2;
//vertices coordinates
int[] polygonsX = new int[nVertex];
int[][] polygonsY = new int[numStates][nVertex];
//step width
float dw = (float) width / (activity.size()-1);

for(int i = 0; i < activity.size(); i++) {
   polygonsX[i] = polygonsX[nVertex-i-1] = (int)(i * dw);

   float h = 0;
   for(int state = 0; state < numStates; state++) {
     //top vertex of polygon
     polygonsY[state][i] = (int)(h * height);

     h += cellCounting.get(i)[state] / ((float)gridWidth *
          gridHeight);
     //bottom vertex of polygon
     polygonsY[state][nVertex - 1 - i] = (int)(h * height);
       }
   }

   //draw the background and the polygons
   g.setColor(Color.white);
   g.fillRect(0, 0, width, height);
   for(int state = 0; state < numStates; state++) {
      g.setColor(MiscUtil.colorFromState(state, numStates));
      g.fillPolygon(polygonsX, polygonsY[state], nVertex);
   }
}
```

`cellCounting.get(i)[state]` contains the numbers of cell in state `state` at time `i`.

A screenshot of the simulator can be found in the appendix, see Figure 3.

5.3.4. The editor

As mentioned in the design part, creating the editor program consists mainly in creating interfaces to configure the rules and most importantly conditions. The most interesting part of the editor, is that it creates pop-ups recursively for the And and Or condition.
To make this possible in Java, the address of the condition has to be given to the newly created window `ConditionFrame`. However, in the `ConditionFrame`, the condition has to be instantiated, since there are multiple different classes of condition that the user chooses of. In Java, when a new instance is created, it is not possible to specify at which address it will be saved to. Therefore, the solution to pass the address of the condition does not work. In such a case, a wrapper class has to be made. This class stores the address of the condition, which can change. But the address of the wrapper class doesn't. (The following code snippets are part of the function that sets the condition on the right of an And condition.)

```
ConditionWrapper wrapper = new
     ConditionWrapper(andCondition.getConditionRight());
```

The `wrapper` is then given as parameter when creating a new `ConditionFrame`. This window stays in the foreground until it is closed. On the closing event of this frame, the condition in the wrapper has now been edited by the user.

```
JFrame frame = new ConditionFrame(parent, automata,
     wrapper);
frame.addWindowListener(new WindowAdapter() {
   @Override
   public void windowClosing(WindowEvent windowEvent) {
     andCondition.setConditionRight(wrapper.condition);
     rightTextField.setText(wrapper.condition.toString());
   }
});
```

5.4. Assessment

This section presents the current limitations of the programs developed for this deliverable and expands on the possible improvements that could be added.

The editor and simulator do not have limitations. Every two dimensional automata can be created with the editor, and can be run on very large grids with the ability to zoom, edit the grid, save it. The data on the graphs can also be saved. There is no limit to the possibilities of what can be done with both of these programs.

However, there are still a lot of improvement that could be made. The user interface could be better. More precisely, the editor does not allow a very good workflow and overview over everything that has been done. Incidentally, some conditions are more difficult to achieve because of the lack of logic

condition, like ”not”, ”xor” and so on.

For the simulator, an improvement would be to be able to choose the colours of the states. This is a small convenience but has a big influence on the user's will to continue to use the program. This feature is also easy to implement thanks to the separation of the logic and the graphics in the program.

Overall, the requirements are all satisfied and this project can be considered a success.

Acknowledgment

The author would like to thank Nicolas Guelfi for the great guidance given during the development of this work. The author would also like to give credit to Johannes Vander Stichele, a first year BiCS student that helped for some math in the scientific deliverable.
Incidentally, the entire year one BiCS class has been a great help by keeping the author motivated.

6. Conclusion

In the scientific deliverable, two function that approximated the state distribution and the activity depending on the current state distribution have been created. Both function use the multiplication of the state distribution with a matrix where the elements correspond to the probability of a cell going from one state to some other state.

On the technical side, two programs have been created that conjointly work on enable an intuitive experience on creating and running cellular automata. An editor that allows the user to design his own cellular automata by adding rules, and the simulator where the user can load his automata, run them, pan around the grid and edit the cells, as well as look at the graphs displaying the state distribution and activity.

Both deliverable have met their requirements, however that does not imply that no more work can be done. As explained in the introduction, the domain of cellular automata is very fast. It is always possible to expand on the ideas touched in this BSP. The editor could for example allow the user to create cellular automata based on neural networks. Another possibility would be changing the type of the grid that is worked on, a grid with hexagons could be used. Or the neighbourhood of the cell could be varied, why should it be restricted to only the cells strictly next to the cell of interest? The same stands for the scientific deliverable, a lot of improvements and further research are possible.

The domain of cellular automata is already vast, but has yet finished the grow and the author wishes that some excitement towards this has been stirred up.

7. Plagiarism statement

I declare that I am aware of the following facts:

- As a student at the University of Luxembourg I must respect the rules of intellectual honesty, in particular not to resort to plagiarism, fraud or any other method that is illegal or contrary to scientific integrity.
- My report will be checked for plagiarism and if the plagiarism check is positive, an internal procedure will be started by my tutor. I am advised to request a pre-check by my tutor to avoid any issue.
- As declared in the assessment procedure of the University of Luxembourg, plagiarism is committed whenever the source of information used in an assignment, research report, paper or otherwise published/circulated piece of work is not properly acknowledged. In other words, plagiarism is the passing off as one's own the words, ideas or work of another person, without attribution to the author. The omission of such proper acknowledgement amounts to claiming authorship for the work of another person. Plagiarism is committed regardless of the language of the original work used. Plagiarism can be deliberate or accidental. Instances of plagiarism include, but are not limited to:

  1) Not putting quotation marks around a quote from another person's work
  2) Pretending to paraphrase while in fact quoting
  3) Citing incorrectly or incompletely
  4) Failing to cite the source of a quoted or paraphrased work
  5) Copying/reproducing sections of another person's work without acknowledging the source
  6) Paraphrasing another person's work without acknowledging the source
  7) Having another person write/author a work for oneself and submitting/publishing it (with permission, with or without compensation) in one's own name (‘ghost-writing’)
  8) Using another person's unpublished work without attribution and permission (‘stealing’)
  9) Presenting a piece of work as one's own that contains a high proportion of quoted/copied or paraphrased text (images, graphs, etc.), even if adequately referenced

  Auto- or self-plagiarism, that is the reproduction of (portions of a) text previously written by the author without citing that text, i.e. passing previously authored text as new, may be regarded as fraud if deemed sufficiently severe.

References

[1] Martin Gardner Scientific American 223 (October 1970) MATHEMATICAL GAMES The fantastic combinations of John Conway's new solitaire game "life" Available at: http://ddi.cs.uni-potsdam.de/HyFISCH/Produzieren/lis_projekt/proj_gamelife/ConwayScientificAmerican.htm

[2] Anonymous Author Quinapalus: The Wireworld Computer Available at: https://www.quinapalus.com/wires11.html

[3] Mordvintsev, et al. Growing Neural Cellular Automata Distill, 2020

[4] Freire, J. G. and DaCamara, C. C.: Using cellular automata to simulate wildfire propagation and to assist in fire management Nat. Hazards Earth Syst. Sci., 19, 169–179 Available at: https://doi.org/10.5194/nhess-19-169-2019 2019

[5] Xiao X, Wang P, Chou KC. Cellular automata and its applications in protein bioinformatics Curr Protein Pept Sci. 2011 Sep;12(6):508-19 doi: 10.2174/138920311796957720 PMID: 21787298

[6] Automata Theory Wikipedia. (2021) Available at: https://en.wikipedia.org/wiki/Automata_theory

[7] Karl-Peter Hadeler, Johannes Müller and Springer, International Publishing Cellular automata: analysis and applications. Cham Springer, 2107

[8] Moore neighbourhood. LifeWiki (2019) Available at: https://www.conwaylife.com/wiki/Moore_neighbourhood

[9] Computer Simulation. Wikipedia. (2021) Available at: https://en.wikipedia.org/wiki/Computer_simulation

[10] Alexander S. Gillis, Sarah Lewis Object Oriented Programming TechTarget. (2021) Available at: https://searchapparchitecture.techtarget.com/definition/object-oriented-programming-OOP

[BiCS(2021)] BiCS Bachelor Semester Project Report Template. https://github.com/nicolasguelfi/lu.uni.course.bics.global University of Luxembourg, BiCS - Bachelor in Computer Science (2021).

[BiCS(2021)] Bachelor in Computer Science: BiCS Semester Projects Reference Document. Technical report, University of Luxembourg (2021)

8. Appendix

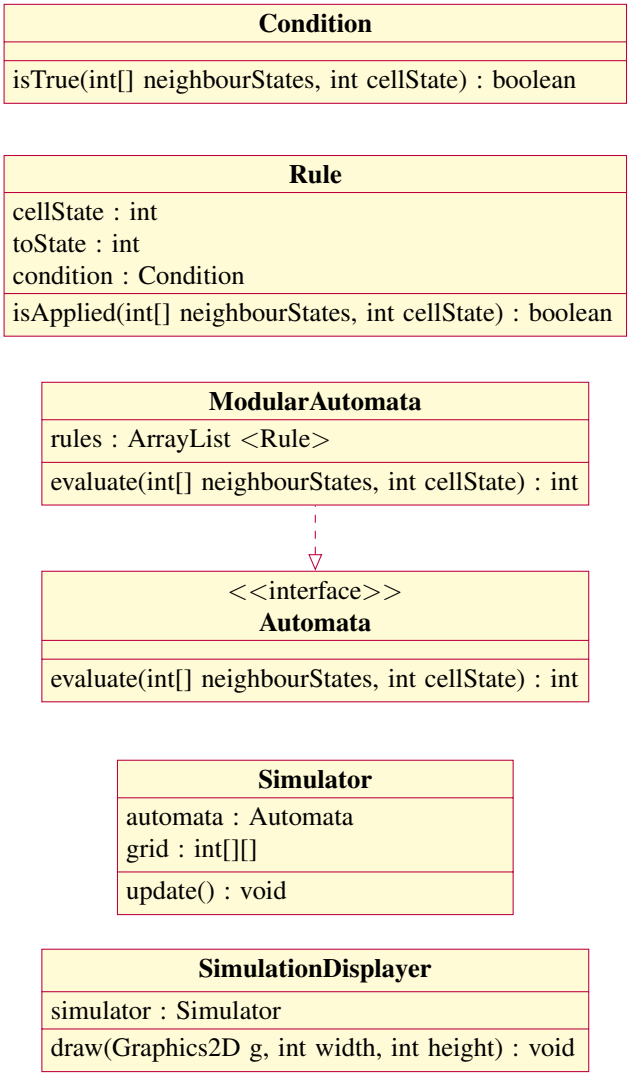

Fig. 2: Class diagram of the main classes in the simulator program

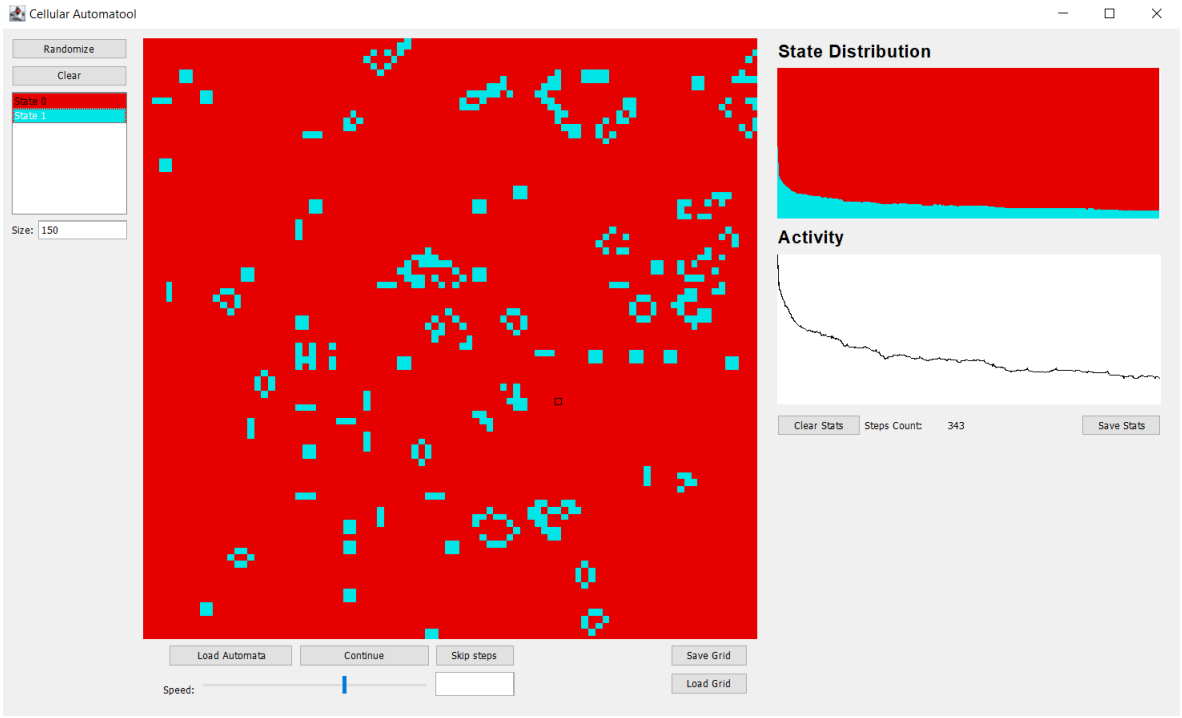

Fig. 3: A screenshot of the simulator program running game of life