UNIVERSITÀ DEGLI STUDI DI TORINO

DIPARTIMENTO DI MATEMATICA GIUSEPPE PEANO

SCUOLA DI SCIENZE DELLA NATURA

Corso di Laurea Magistrale in Matematica


Tesi di Laurea Magistrale

# CELLULAR EVOLUTION:
## A SYMBIOSIS BETWEEN CELLULAR AUTOMATA AND GENETIC ALGORITHMS

Relatore: Cerruti Umberto
Correlatore: Murru Nadir

Candidato: Dutto Simone

ANNO ACCADEMICO 2015/2016

# Index

# Chapter 1

# Introduction

Science purpose has always been understanding the world and trying to control it: we humans desire to predict, and sometimes prevent or make happen, every phenomenon that occurs around us. There are still lots of things that we can not handle, also because behind every new discovery stand lots of never studied evolutions and knowledges.

Everybody knows that in mankind history there are some specific events that changed completely the way of thinking and living. One of the most important revolution of this kind concerns the advent of electronic computers: thanks to Alan Turing and his successors now we can overcome the limitation of human brain and reach results that no one would have imagined. Aside from common technologies, like personal computers and smartphones, by which everybody has possibilities never thought before, this wind of change brought a big boost in every scientific field. Now it is possible make simulations of every kind of environment and even create new form of life with behaviours similar to the real ones. With these instruments we can try to simulate and study situations difficult to examine in real life or prevent future developments of current problems.

Cellular automata are one of the most used tool in these situations. They were introduced by von Neumann in the '50s to resolve a problem of self-replicating automata and they quickly evolved in the following year: in '60s they were studied as particular dynamical systems, in '70s Conway introduced his renowned Game of Life and from the '80s to the present day, thanks especially to Wolfram, they took a role into every discipline of science. Now, thanks to their versatility, you can find a sign of cellular automata in biology, chemistry, Earth science, physics, informatics and lots of other fields of study.

A connected but different approach is given by genetic algorithms. They were invented in the '60s by Holland that wanted to study the phenomenon of adaptation and implement it into computer systems: he took inspiration by Darwin's natural selection and obtained a procedure that now is applied particularly in computer science to reach better results in optimization and search problems.

Since structures of cellular automata and genetic algorithms are similar, one can easily think to merge them in a new kind of instrument that combine the advantages of both and can be used to find new interesting study cases. Hence we introduce our system: a bi-dimensional cellular automata that works with rules of genetic algorithms. Our purpose is to explain its behaviour and make this new procedure available to everyone, in the hope that today's prolific community will find useful applications, especially in other disciplines.

# Chapter 2

# Cellular automata

Here we want to introduce some simple notions about cellular automata. After considering the generic definition and some important features, we will see how a cellular automaton can be mathematically formalized. In particular we will concentrate on one-dimensional cellular automata, because of their simple and explicative behaviour, and on bi-dimensional cellular automata, since our work is focused on them. See [Ilac01] and [Wolf02] for more details and specifications.

## 2.1 Definition and characteristics

Cellular automata were first introduced in the early '50s by John von Neumann that wanted to create simple models of biological self-reproduction. In the years the study of these systems has generated great interest, thanks to their ability to develop lots of very complex patterns of behaviour, starting with relatively simple well defined rules. Another important achievement is given by the coherence between them and real systems, especially in some essential features of complex self-organizing cooperative behaviour.

**Definition 2.1.1.** *Cellular automata* (CA) are deterministic mathematical systems characterized by:

- a discrete lattice of simple homogeneous components called *cells*;
- finite number of discrete states taken by cells;
- an intrinsic rule that at each discrete unit of time determines interactions between every cell and its neighbourhood.

Despite the simplicity of these rules, such systems are capable of extremely complicated behaviour. Thanks to that, besides the direct work in mathematics and computer science, there have been applications of CA in lots of disciplines. Some specific examples of phenomena that have been modelled by CA include fluid and chemical turbulence, plant growth, DNA evolution, the propagation of infectious diseases, social dynamics, forest fires and patterns of electrical activity in neural networks.

## 2.2 System formalization

Although each CA is defined or selected to fit the requirements of a particular model, the definition of any of these specific systems requires the specification of the following generic characteristics:

- **discrete cellular state space $\mathcal{L}$**
  the discrete lattice of cells that constitutes the structure of the CA. It can be one-dimensional with simple shaped cells, two-dimensional with every kind of cells shape (like rectangular or hexagonal), three-dimensional with the same liberty on cells form, or even random;

- **local value space $\Sigma$**
  a finite set of different values $\Sigma \equiv \{0, 1, 2, \dots k-1\}$ that every cells in $\mathcal{L}$ can assume. We can distinguish each cell using its position in $\mathcal{L}$, indicated by a multi-index, and consider the value of the cell $\underline{i}$ at the instant $t$ given by
  $$\sigma_{\underline{i}}(t) \in \Sigma \tag{2.2.1}$$
  Usually $\Sigma$ is any finite commutative ring (like $\mathbb{Z}_k$). If $\mathcal{L}$ is a finite lattice of $N$ cells, then the total number of *global states* is finite (given by $k^N$);

- **boundary conditions**
  although CA are assumed to live on infinite lattices, computer simulations must necessarily run on finite sets. For a one dimensional lattice with $N$ cells it is common to use periodic boundary conditions, in which $\sigma_{N+1} = \sigma_1$ (as in a ring), or to consider two extra cells, over the boundaries, with fixed value 0. Similarly, in two dimensions, it is usual to have the dynamics take place on a torus, so that if the lattice is $M \times N$ we have $\sigma_{M+1,j} = \sigma_{1,j}$ and $\sigma_{i,N+1} = \sigma_{i,1}$, or to consider a boundary with fixed value 0, or also to have a boundary on two sides and the link on the other two (like a cylinder). Anyhow boundary conditions play an important role in shaping the form of the resulting dynamics;

- **dynamical rule $\Phi$**
  the evolution of the CA is described by a function $\Phi : \Sigma^n \longrightarrow \Sigma$ where $n$ specifies the number of cells in the chosen kind of neighbourhood. The transition rule is most generally written as

$$\sigma_{\underline{i}}(t+1) = \Phi(\sigma_{\underline{j_1}}(t), \sigma_{\underline{j_2}}(t), \ldots, \sigma_{\underline{j_n}}(t)) \tag{2.2.2}$$

  where, for all $k \in \{1, 2, \ldots n\}$, $\underline{j_k} \in \mathcal{N}(\underline{i})$ neighbourhood of the cell $i$. One iteration step of the dynamical evolution is achieved after the simultaneous application of this rule to each cell in the lattice $\mathcal{L}$.

### 2.2.1 One-Dimensional CA

In one-dimensional CA we can choose an arbitrary range $r$ that determines the dimension of neighbourhoods. Given that the dynamic rule becomes

$$\sigma_i(t+1) = \Phi(\sigma_{i-r}(t), \ldots, \sigma_i(t), \ldots, \sigma_{i+r}(t)) \tag{2.2.3}$$

If there are $N$ cells, after choosing some specific initial global state

$$\{\sigma_1(0), \sigma_2(0), \ldots, \sigma_N(0)\} \tag{2.2.4}$$

the temporal evolution of this CA is then given by the simultaneous application of $\Phi$ to each of the cells.

*Example* 2.2.1. Considering neighbourhoods of range 1, we can study the simplest one-dimensional CA called *elementary cellular automata* by Stephen Wolfram, who has extensively studied their amazing properties. Their behaviour can be completely described by simple rules given by choosing one of the two possible results for each of the $2^3 = 8$ possible neighbourhoods. Hence there are only $2^8 = 256$ possible elementary CA.
Each neighbourhood is an ordered list of 3 bits so we can consider the associate decimal number $n$ and, calling the relative result $r_n$, we can describe the used rule with the integer given by

$$\sum_{n=1}^{8} n^{r_n} \tag{2.2.5}$$

In the figure 2.1 we can see *rule 30* (00011110) and its first 20 generations starting with a single alive cell (each line is a new generation). This cellular automata is of special interest because Wolfram in 2002 proved that it is chaotic and hence it can be used as random number generator.

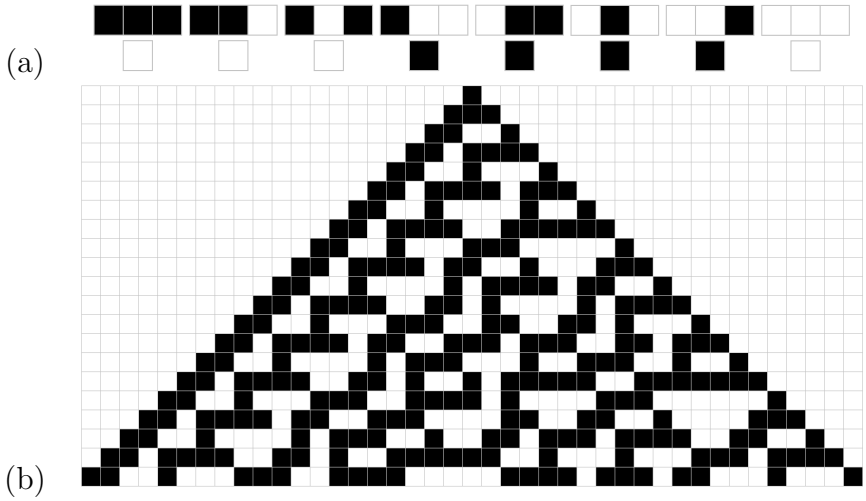


**Figure 2.1:** (a) all possible one-dimensional neighbourhoods with relative results decided according to rule 30: they are, in order, 0 0 0 1 1 1 1 0; (b) first 20 iterations of rule 30.

### 2.2.2 Two-Dimensional CA

The extension of generic CA systems to two dimensions is significant for two reasons: first, the extension brings with it the appearance of many new phenomena involving behaviours of the boundaries of two-dimensional patterns that have no simple analogues in one-dimension. Secondly, two-dimensional dynamics permit easier comparison to real physical systems.

There is a variety of different lattices and neighbourhood structures that can be used in two-dimensional CA. An important factor is the shape of cells: regular polygon are usually used (like triangular, squared or hexagonal) but one can also consider irregular shapes providing that every cell has the same number of edges (like Penrose tiling, see [OwSt10]).

Regarding neighbourhoods there are two simple configurations that are generally used and can be adapted to every choice of cells shape:

- von Neumann neighbourhood (figure 2.2(a)), consisting of the center cell and all the cells which have a common edge with it;
- Moore neighbourhood (figure 2.2(b)), consisting of the center cell and all cells which have at least one common vertex with it.

Talking about the dynamic rule, the equation (2.2.2) is so generic that can describe an impressive variety of complex behaviours. You can see a simple but really important example of two-dimensional CA in chapter 5.

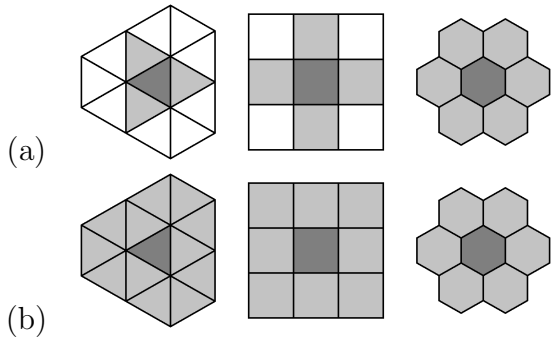


**Figure 2.2:** (a) von Neumann neighbourhoods with different cell shapes; (b) Moore neighbourhoods in the same cases.

## 2.3 Meaningful approaches

Now we want to provide a general list of what are some of the most important and interesting generalizations of the basic CA systems. As the following ones, the purpose of the system that we will introduce in chapter 4 is to improve the simple concept of cellular automaton in order to obtain new powerful behaviours.

**Reversible rules**
Almost any rule will define a non-invertible dynamic, so that it will be impossible to determine what local configuration at the previous time step gave rise to a particular value at the current step. Since physical dynamical laws are microscopically reversible any simulation of real physical systems can be made only if the underlying CA is itself reversible. Hence the best way to write down a general form for such rules is given by

$$\sigma_{\underline{i}}(t+1) = [\Phi(\sigma_{\underline{j_1}}(t), \ldots, \sigma_{\underline{j_n}}(t)) - \sigma_{\underline{i}}(t-1)](\mathrm{mod}\ k) \tag{2.3.1}$$

where $\Phi$ is a generic dynamic rule and, for all $k \in \{1, 2, \ldots n\}$, $\underline{j_k} \in \mathcal{N}(\underline{i})$. For such rules any value of $\sigma_{\underline{i}}$ can be determined if the values at site $\underline{i}$ are known for two consecutive time steps, therefore the total information contained in the initial state has to be preserved for all time.

**Probabilistic CA**
Deterministic results of dynamic rules may be replaced with specifications of the probabilities of value assignments:

$$\mathbf{P}\big(\{\sigma_{\underline{i}}(t+1) = \alpha,\ \text{given } \sigma_{\underline{i}}(t) \text{ and some values in } \mathcal{N}(\underline{i})\}\big) \tag{2.3.2}$$

valuing it on every $\alpha \in \Sigma$. Instead of studying particular evolutions of arbitrary initial states, such rules determine ensembles of CA trajectories.

**Non-homogeneous CA**
Every CA rule defined so far was homogeneous which means that each cell of the system evolves according to the same rule. The simplest example of non-homogeneous CA is characterized by two different rules which are randomly distributed throughout the lattice but it is also possible to study the extreme version in which the lattice is randomly populated with all $2^{2^k}$ possible boolean functions of $k$ inputs.

**Structurally dynamic CA**
Another possibility is to make the lattice itself changing with the dynamical evolution of the system. Considering changing neighbourhoods $\mathcal{N}(\underline{i}, t)$ and calling $A(\mathcal{L}, t)$ the adjacency matrix of the lattice $\mathcal{L}$ at the instant $t$, the system to be studied is given by

$$\begin{cases} \sigma_{\underline{i}}(t+1) = \Phi\left[\sigma_{\underline{j}}(t) \,|\, \underline{j} \in \mathcal{N}(\underline{i}, t), a_{\underline{i},\underline{j}}(t) \in A(\mathcal{L}, t)\right] \\ a_{\underline{i},\underline{j}}(t+1) = \Psi\left[\sigma_{\underline{j}}(t) \,|\, \underline{j} \in \mathcal{N}(\underline{i}, t), a_{\underline{i},\underline{j}}(t) \in A(\mathcal{L}, t)\right] \end{cases} \tag{2.3.3}$$

# Chapter 3

# Genetic algorithms

Let us now take a look at genetic algorithms and how do they work. Starting from their origins, we begin with a biological approach that brings us to the specific definition. In the second part we will observe how genetic algorithms mathematically work and the causes of their power in problem solving. To see in deep these argumentations you can follow [Gold89].

## 3.1 First ideas and definitions

As we said in chapter 1, genetic algorithms were invented by John Holland in the '60s when he introduced an algorithm based on natural selection and genetic operators like crossover and mutation. His goals were mainly two: to summarize and explain the adaptive process of natural systems and to reproduce the important mechanisms of this process in artificial systems.

Since genetic algorithms were inspired by natural systems, let us start with some simple biological definitions. All living organisms consist of cells, and each cell contains the same set of one or more *chromosomes* (strings of DNA) that determines the characteristics of the organism. A chromosome can be divided into *genes* each of which is located in a particular position, called *locus*, and encodes a particular trait of the organism. A natural system can be seen as a population of chromosomes and during its evolution there are generally three step. First an operator called *fitness function* associates a measure of goodness to each chromosome: the fitter the chromosome, the more times it can be selected by another one to reproduce. During the *coupling* a locus is randomly chosen and the sub-sequences before or after that locus are exchanged between the couple of chromosomes to create two offsprings. Finally, with some very small probability, *mutation* may occur so that some of the genes in a chromosome randomly change.

**Definition 3.1.1.** A *genetic algorithm* (GA) is a higher-level procedure described by these features:

- population of *chromosomes*;
- *reproduction* according to a *fitness function*;
- *crossover* to produce new offspring;
- random *mutation* of new offspring.

Although their simple definition, GA are really useful in optimization and search problems, especially thanks to qualities like:

1. easy and fast operators;
2. robust processing (high balance between efficiency and efficacy);
3. the use of a coding of the parameter set instead of the parameters themselves;
4. the evolution of a population of points and not of a simple point;
5. the independence from derivatives or other auxiliary knowledge;
6. the use of probabilistic transition rules instead of the deterministic ones.

Every system that presents the characteristics in definition 3.1.1 is a genetic algorithm but now we want to concentrate on one of the simplest models, directly inspired by biological ones. It considers bit strings of fixed length $l$ as chromosomes (in which locus can assume the values 0 or 1), representing the real values of the system, and its operators work as follows:

- the reproduction operator select strings that enter into a mating pool for the following step. It may be implemented in lots of way and one of the easiest is make a weighted choice according to the fitness values, so that more highly fit strings have a higher number of offspring in the succeeding generation;
- given the selected strings they are randomly paired and the crossover begins: for each couple a random position $k$ in the range $[1, l-1]$ is selected and two new strings are created by swapping all characters in positions $k+1, \ldots, l$ between the initial strings;
- finally, with really low probability, one of the characters of the resulting string may change state (from 0 to 1 or vice versa) and, after that, we have our new population.

## 3.2 Mathematical foundations

Even though the operation of GA is remarkably straightforward, this processing of strings really causes the implicit processing of the system. To see how this works we need to introduce some simple notation.

In the GA on which we want to focus, strings are constructed over the binary alphabet $V = \{0, 1\}$. We will refer to them by capital letters while their individual characters will be indicated with lower-case letters subscripted by their position, so that a generic string of length $l$ will be represented as:

$$A = a_1 a_2 \dots a_l \tag{3.2.1}$$

Meaningful genetic search requires populations of strings, indicated with bold capital letters, which depend on time, hence we have

$$\mathbf{A}(t) = \{A_1, A_2, \dots, A_n\} \tag{3.2.2}$$

Since important similarities among highly fit strings can help guide a search, Holland introduced *schemata*: similarity templates describing subsets of strings with similarities at certain string positions. Practically a schema $H$ is a string constructed over the alphabet $V^* = \{0, 1, *\}$ where the additional symbol $*$ matches either a 0 or a 1 at a particular position. A string $A$ as in (3.2.1) is an example of the schema $H = h_1 h_2 \dots h_l$ if $\forall i \in \{1, 2, \dots, l\}$

$$a_i = h_i = 0 \quad \vee \quad a_i = h_i = 1 \quad \vee \quad h_i = *,\ a_i \in V \tag{3.2.3}$$

Considering binary strings of length $l$ there are $3^l$ possible schemata defined over them and if a population contains $n$ strings there are at most $n \cdot 2^l$ schemata represented in it, because each string is an example of $2^l$ schemata.

We can characterize a schemata through the following properties:

- the *order* of a schema $H$, denoted by $o(H)$, is the number of his fixed positions (number of 0s and 1s);
- if $H$ is a schema with $o(H) > 0$, his *defining length*, denoted by $\delta(H)$, is the difference between the last and first specific (non $*$) string position. Clearly it may assume values from 0 to $l - 1$.

*Example* 3.2.1. Considering the string of length 6 $A = 100101$ two of the schemata of which is an example are $H_1 = 1*****$ and $H_2 = *00**1$. Clearly the second one is more specific and spans more of the total string length: we have $o(H_1) = 1$ and $\delta(H_1) = 0$ while $o(H_2) = 3$ and $\delta(H_2) = 6 - 2 = 4$.

Now we want to study the individual and combined effect of GA operators on schemata represented by a population of strings: suppose at a given time step $t$ there are $m$ examples of a particular schema $H$ within the population $\mathbf{A}(t)$ as in (3.2.3). Hence we will write $m = m(H,t)$. During reproduction a string $A_i \in \mathbf{A}(t)$ gets selected with probability

$$p_i = \frac{f_i}{\sum_{j=1}^{n} f_j} \tag{3.2.4}$$

where, for all $j \in \{1, 2, \ldots, n\}$, $f_j$ is the fitness value of the string $A_j$.
After reproducing a new population of $n$ strings we expect that a particular schema grows as the ratio of the average fitness of the schema $f(H)$ to the average fitness of the entire population $\overline{f}$. In fact we have

$$m(H, t+1) = n \cdot \sum_{i=1}^{m(H,t)} p_i = n \cdot \frac{f(H) \cdot m(H,t)}{\sum_{j=1}^{n} f_j} = m(H,t) \cdot \frac{f(H)}{\overline{f}} \tag{3.2.5}$$

It is important to observe that this expected behaviour is carried out with every schema contained in the population in parallel.

During the crossover it is easy to see that a schema survives when the cross site falls outside the defining length, hence the survival probability is

$$p_s = 1 - \frac{\delta(H)}{l-1} \tag{3.2.6}$$

Since crossover is itself performed by random choice, say with probability $p_c$, the survival probability becomes

$$p_s \geq 1 - p_c \cdot \frac{\delta(H)}{l-1} \tag{3.2.7}$$

Finally we have mutation: supposing that its probability is $p_m$, in order for a schema to survive all of its fixed positions must themselves survive. Since a single character survives with probability $1 - p_m$ the scheme survive with a probability of $(1-p_m)^{o(H)} \sim 1 - o(h) \cdot p_m$ because $p_m << 1$.

Hence the combined effect on the expected number of examples that a schema $H$ receive in the next generation is given by

$$m(H, t+1) \geq m(H,t) \cdot \frac{f(H)}{\overline{f}} \cdot \left[1 - p_c \cdot \frac{\delta(H)}{l-1} - o(h) \cdot p_m\right] \tag{3.2.8}$$

This conclusion is called the *Fundamental Theorem of Genetic Algorithms* and let us see that short, low-order, above-average schemata receive exponentially increasing trials in subsequent generations.

## 3.3 A simple example

Now you can see how genetic algorithms work practically. We will consider a little population and a simple fitness function so that computation will not become too complex and long.

Our chromosomes are bit strings of length 5, the population size is 4 and the fitness function $f$ is simply the square of the decimal number $x$ corresponding to the string. The reproduction operator will choose 4 strings according the approximation of $f(x)/\overline{f}$. For the crossover we use the method explained in the previous sections with $p_c = 1$ and we choose the mutation probability $p_m = 0,001$. In parallel we study the evolution of two schemata.

| String | $x$ | $f(x)$ | $f(x)/\overline{f}$ | Count | Reprod. | Offspring | $x$ | $f(x)$ |
|---|---|---|---|---|---|---|---|---|
| 01101 | 13 | 169 | 0,58 | 1 | 0110\|1 | 01100 | 12 | 144 |
| 11000 | 24 | 576 | 1,97 | 2 | 1100\|0 | 11001 | 25 | 625 |
| 01000 | 8 | 64 | 0,22 | 0 | 11\|000 | 11011 | 27 | 729 |
| 10011 | 19 | 361 | 1,23 | 1 | 10\|011 | 10000 | 16 | 256 |
| Min | | 64 | | | | | | 144 |
| Average | | 293 | | | | | | 439 |
| Max | | 576 | | | | | | 729 |

| | Before Rep. | | After Rep. | | After All | |
|---|---|---|---|---|---|---|
| Schema | $m(H)$ | $f(H)$ | Exp. $m(H)$ | $m(H)$ | Exp. $m(H)$ | $m(H)$ |
| $1****$ | 2 | 469 | 3,20 | 3 | 3,20 | 3 |
| $*10**$ | 2 | 320 | 2,18 | 2 | 1,64 | 2 |
| $1***0$ | 1 | 576 | 1,97 | 2 | 0,00 | 0 |

As you can see after only one step our system has already improved: in fact minimum, average and maximum fitness values are increased compared to the initial ones. Talking about schemata the estimations made in the previous section are completely respected.

# Chapter 4

# Cellular evolution

Here we introduce our cellular automaton inspired by genetic algorithms. This work started in 2015 with the brilliant idea of my supervisor of merge the two concepts: the results were really surprising and fascinating. Now we want to continue this approach optimizing the process and studying its behaviour more deeply. In this chapter we want to specify our implementation first discursively, explaining how we linked cellular automata and genetic algorithms, and then using a pseudo-code, so that everyone can try to implement our idea.

## 4.1 A new idea and its implementation

Starting by the definitions given in the previous chapters, it is easy to see that cellular automata and genetic algorithms are closely correlated. Hence we want to define *Cellular Evolution*, a dynamic system characterized by the structure of a cellular automata and a behaviour similar to that of genetic algorithms. The purpose is to obtain a new kind of process that can mix together the advantages of cellular automata and genetic algorithms, and maybe find some interesting and useful results.

While in simple genetic algorithms each cell evolves according to a dynamical rule based only on the states of cells in its neighbourhood (as in (2.2.2)), now we want a fitness function that allow each cell to select the best mate in its neighbourhood, a crossover operator that defines couples interactions and the resulting evolution of cells states and a final possible mutation.

As every kind of process seen before, given an initial population, it evolves in discrete time steps.

Our principal implementations use bi-dimensional lattices of squared cells, but it is possible to generalize the process to every kind of structure. In order to obtain a cellular automata, during computations of fitness values and the crossover operations we still consider limited neighbourhoods instead of mating pools as in GA. In particular we will use Moore neighbourhoods but it is a completely indifferent choice. We will study every kind of boundary condition and we also want to consider fixed nodes grids.

The local value space $\Sigma$ is, as in section 2.2, a finite set of different values, not necessarily numeric, assumable by each cell. As in CA, we usually consider a set of numeric values like $\mathbb{Z}_k$. We maintain the notation $\sigma_{\underline{i}}(t)$ for the state of cell $\underline{i}$ at the instant $t$, as introduced in 2.2.1.

As we said, the innovative part concerns the dynamical rule. Since it is inspired by genetic algorithms, we can divide this process in three steps:

- given a cell $\underline{i} \in \mathcal{L}$ and its neighbourhood $\mathcal{N}(\underline{i}) = \{\underline{j_1}, \underline{j_2}, \ldots, \underline{j_9}\}$ we call the *configuration* of $\underline{i}$ at time $t$

$$\mathcal{C}_{\underline{i}}(t) = (\sigma_{\underline{j_1}}(t), \sigma_{\underline{j_2}}(t), \ldots, \sigma_{\underline{j_9}}(t)) \tag{4.1.1}$$

  and we consider a *fitness function* that takes these configurations and returns values in an ordered set $\mathcal{F}$ containing possible fitness values

$$f : \Sigma^9 \longrightarrow \mathcal{F}\ , \quad f(\mathcal{C}_{\underline{i}}(t)) = f_{\underline{i}}(t) \tag{4.1.2}$$

  In this way, at each time step, every cell receives a fitness value that depends on states of the cells in its neighbourhood. After that, each cell $\underline{i}$ selects randomly $\underline{j}$, one of the fittest cells in $\mathcal{N}(\underline{i})$, to interact with it. Hence fittest cells have higher chances of being chosen for couplings;

- now that each cell has a mate it is time for the *crossover*. Since we want to keep the system as generic as possible we consider the following operator that takes states and fitness values of the two cells and returns the new value of the cell $i$:

$$\chi : (\Sigma \times \mathcal{F})^2 \longrightarrow \Sigma\ , \quad \chi\big(\sigma_{\underline{i}}(t), f_{\underline{i}}(t), \sigma_{\underline{j}}(t), f_{\underline{j}}(t)\big) = \sigma_{\underline{i}}(t+1) \tag{4.1.3}$$

  Unlike what happens in genetic algorithms, the result of the crossover will be only one offspring that take the place of cell $\underline{i}$, otherwise the system would not be a cellular automata;

- finally we have *mutation* that changes randomly the state of some cells, as always with a very small probability.

The resulting process is still a cellular automaton because composing the part of selection with the crossover operator and the mutation step we obtain a dynamic rule. We observe that, despite during each single step we considered Moore neighbourhoods, the dynamic rule of our system seen as CA uses different neighbourhoods: for the cell $\underline{i}$ it takes the states of the cells in

$$\overline{\mathcal{N}}(\underline{i}) = \bigcup_{\underline{j} \in \mathcal{N}(\underline{i})} \mathcal{N}(\underline{j}) \tag{4.1.4}$$

and returns the new state of $\underline{i}$. Hence the neighbourhood used by the dynamic rule of Cellular Evolution is the one in figure 4.1.

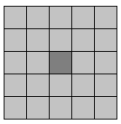

**Figure 4.1:** Neighbourhood used by Cellular Evolution as CA.

## 4.2 The algorithm step by step

We implemented Cellular Evolution in two different languages: first using Matlab, that allows us to obtain a powerful and fast way to run our process, and at a later time we create a little program in Python, because it is open-source and hence gives the possibility to everyone to try our system. Our works can be found in [CEDr17] and are completely open-source.

Trying to keep the argumentation as generic as possible, now we use a pseudo-code to describe our methods of implementation. Let us start with the practical definition of the used operators:

- the initial population is given randomly or inserted by the user and has to be a matrix `pop` of size `m` $\times$ `n` with values in the range of possible state values `[0,1,...,nstates-1]`. It will evolve for `ngen` steps and can be customized with the following parameters:
  - a variable `type` describes the shape of our world: `type=0` means toroidal world, `type=1` closed, `type=2` stands for a vertical cylinder shape (closed on upper and lower sides) and `type=3` is for horizontal cylinder (closed on left and right sides);
  - the presence of a fixed nodes grid, that is given by a boolean variable `grid` and, if `grid=1`, the size of its meshes is specified by two numeric values called `r` and `c`.

- the fitness function has to be an operator `fitfun` that takes the state values of the nine cells in a neighbourhood and returns an element of `[0,1,...,nfit-1]` as fitness values for the center cell. We can consider a function that at each step takes the matrix `pop` and returns a matrix `fit` of dimensions `m` $\times$ `n` that contains all fitness values. Each cell selects the fittest mate in its neighbourhood and, in case of cells with same fitness value, it chooses randomly one of them using a seed `s`;

- the crossover operator that describes interactions between cells (considering states and fitness values) can be seen as a 4-dimensional matrix `co` of size `nstates` $\times$ `nfit` $\times$ `nstates` $\times$ `nfit` containing the result of every possible combination of two states and two fitness values;

- regarding the mutation operator it will depends on two variables: `fmut` and `pmut` that give respectively its frequency (how many steps pass between a mutation and another) and the probability of its occurring.

Now that we have explained all the used variables we can show the entire algorithm and, afterwards, see what every step does.

```
1) define function:  N(M) = southshift(M)
   define function:  S(M) = northshift(M)
   define function:  W(M) = eastshift(M)
   define function:  E(M) = westshift(M)
2) define function:  shuffle(M,seed) = permuteelements(M,seed)
3) rand = [0:8]
4) for {t in [1:ngen]} do
5)   fit = fitfun(pop)
6)   print(pop), print(fit)
7)   rand = shuffle(rand,s)
8)   for {{x in [0:m-1]} and {y in [0:n-1]}} do
9)     pp[rand[0],x,y] = pop[x,y]
       pp[rand[1],x,y] = N(pop)[x,y]
       pp[rand[2],x,y] = N(E(pop))[x,y]
       pp[rand[3],x,y] = E(pop)[x,y]
       pp[rand[4],x,y] = E(S(pop)[x,y]
       pp[rand[5],x,y] = S(pop)[x,y]
       pp[rand[6],x,y] = S(W(pop))[x,y]
       pp[rand[7],x,y] = W(pop)[x,y]
       pp[rand[8],x,y] = W(N(pop))[x,y]
```

```
10)     ff[rand[0],x,y] = fit[x,y]
        ff[rand[1],x,y] = N(fit)[x,y]
        ff[rand[2],x,y] = N(E(fit))[x,y]
        ff[rand[3],x,y] = E(fit)[x,y]
        ff[rand[4],x,y] = E(S(fit))[x,y]
        ff[rand[5],x,y] = S(fit)[x,y]
        ff[rand[6],x,y] = S(W(fit))[x,y]
        ff[rand[7],x,y] = W(fit)[x,y]
        ff[rand[8],x,y] = W(N(fit))[x,y]
11)   for {{x in [0:m-1]} and {y in [0:n-1]}} do
12)     best[x,y] = min(i | ff[i,x,y] = max(ff[[0:8],x,y]))
        selp[x,y] = pp[best[x,y],x,y]]
        self[x,y] = ff[best[x,y],x,y]]
13)   if {{type = 0} and {grid = 0}} then
14)     for {{x in [0:m-1]} and {y in [0:n-1]}} do
15)       pop[x,y] = co[pop[x,y],fit[x,y],selp[x,y],self[x,y]]
16)       if {pmut > 0} then
17)         if {mod(t,nmut) = 0} then
18)           if {random(0,1) < pmut} then
19)             pop[x,y] = int(random(0,nstates-1))
20)   if {{type = 0} and {grid = 1}} then
21)     for {{x in [0:m-1]} and {y in [0:n-1]}} do
22)       if {{mod(x,r) > 0} or {mod(y,c) > 0}} then
23)         pop[x,y] = co[pop[x,y],fit[x,y],selp[x,y],self[x,y]]
24)         if {pmut > 0} then
25)           if {mod(t,nmut) = 0} then
26)             if {random(0,1) < pmut} then
27)               pop[x,y]=int(random(0,nstates-1))
28)   if {{type = 1} and {grid = 0}} then
29)     for {{x in [1:m-2]} and {y in [1:n-2]}} do
30)       pop[x,y] = co[pop[x,y],fit[x,y],selp[x,y],self[x,y]]
31)       if {pmut > 0} then
32)         if {mod(t,nmut) = 0} then
```

```
33)           if {random(0,1) < pmut} then
34)             pop[x,y] = int(random(0,nstates-1))
35)   if {{type = 1} and {grid = 1}} then
36)     for {{x in [1:m-2]} and {y in [1:n-2]}} do
37)       if {{mod(x,r) > 0} or {mod(y,c) > 0}} then
38)         pop[x,y] = co[pop[x,y],fit[x,y],selp[x,y],self[x,y]]
39)         if {pmut > 0} then
40)           if {mod(t,nmut) = 0} then
41)             if {random(0,1) < pmut} then
42)               pop[x,y]=int(random(0,nstates-1))
43)   if {{type = 2} and {grid = 0}} then
44)     for {{x in [1:m-2]} and {y in [0:n-1]}} do
45)       pop[x,y] = co[pop[x,y],fit[x,y],selp[x,y],self[x,y]]
46)       if {pmut > 0} then
47)         if {mod(t,nmut) = 0} then
48)           if {random(0,1) < pmut} then
49)             pop[x,y] = int(random(0,nstates-1))
50)   if {{type = 2} and {grid = 1}} then
51)     for {{x in [1:m-2]} and {y in [0:n-1]}} do
52)       if {{mod(x,r) > 0} or {mod(y,c) > 0}} then
53)         pop[x,y] = co[pop[x,y],fit[x,y],selp[x,y],self[x,y]]
54)         if {pmut > 0} then
55)           if {mod(t,nmut) = 0} then
56)             if {random(0,1) < pmut} then
57)               pop[x,y]=int(random(0,nstates-1))
58)   if {{type = 3} and {grid = 0}} then
59)     for {{x in [0:m-1]} and {y in [1:n-2]}} do
60)       pop[x,y] = co[pop[x,y],fit[x,y],selp[x,y],self[x,y]]
61)       if {pmut > 0} then
62)         if {mod(t,nmut) = 0} then
63)           if {random(0,1) < pmut} then
```

```
64)            pop[x,y] = int(random(0,nstates-1))
65)   if {{type = 3} and {grid = 1}} then
66)     for {{x in [0:m-1]} and {y in [1:n-2]}} do
67)       if {{mod(x,r) > 0} or {mod(y,c) > 0}} then
68)         pop[x,y] = co[pop[x,y],fit[x,y],selp[x,y],self[x,y]]
69)         if {pmut > 0} then
70)           if {mod(t,nmut) = 0} then
71)             if {random(0,1) < pmut} then
72)               pop[x,y]=int(random(0,nstates-1))
```

Obviously this language has to be translated depending on the program used, but this list of passage may be useful to understand what the system does during a run. Let us take a look at each step of the algorithm.

**Initialization**
During the first two steps we have to define some functions that will be useful in the next steps. The functions in step 1 are simple shift of matrix elements in the indicated direction. We name them with the opposite letter for a good reason: for example `N` is the south shift because in the position `(i,j)` the matrix `N(M)` must have the value located in the position above `(i,j)` in `M`. In step 2 we define a random shuffle (depending on a seed) of the elements of a vector and in step 3 we consider an array `rand` containing all the positions in neighbourhoods.

**Defining fitness values**
In step 4 the main cycle starts. The first thing to do is define the fitness value of each cell (step 5). After that we can print the images of the current population and fitness matrices (step 6).

**Selection**
This operation occupies steps 7-12. First we update `rand` shuffling it with the given seed `s` (step 7). This will randomize the choice of the fittest cell. Now we define two three-dimensional matrices of size $9 \times m \times n$ (steps 8, 9, 10) that allows us to detect more easily the fittest cell: each vector `pp[[0:8],x,y]` contains the states in the neighbourhood of the cell `(x,y)` while each `ff[[0:8],x,y]` contains their fitness values (in both cases in the random order defined by `rand`). During steps 11 and 12 we select the best fitness value for every cell and we determine the matrices of selected states (`selp`) and selected fitness values (`self`).

**Crossover and mutation**
For these operators there are eight different cases, depending on choices of `(type,grid)`. The only difference between them is a set of cells that do not evolve (that includes border ones and/or grid ones). In particular case `(0,0)` is studied in steps 13-19, case `(0,1)` during steps 20-27, `(1,0)` in steps 28-34, `(1,1)` in steps 35-42, `(2,0)` in steps 43-49, `(2,1)` in steps 50-57, `(3,0)` in steps 58-64 and finally case `(3,1)` during steps 65-72. Each case is different because of the cells that can evolve. After the selection of those ones (first two or three steps depending on the case), they evolve according to the value in the crossover matrix: cell `(x,y)` takes the state in position `(pop[x,y],fit[x,y],selp[x,y],self[x,y])` in the matrix `co`. Finally the last four steps of each case determine, if mutation is possible, its results.

Some interesting and wonderful results of this process can be seen in the last chapter of this paper.

# Chapter 5

# Game of Life and its implementation

As we told in chapter 2 here we want to introduce a special example of two-dimensional cellular automaton: Conway's Game of Life (see [Gard70]). We will see its definition, some characteristics of its behaviour and how to prove its most important property. After this introductory part, we will show one of the biggest results reached with our system: it is possible to implement Conway's rule using Cellular Evolution. At the end of this chapter there will be a presentation of two cellular automata, created by us, describing two possible interactions between two lives in the same environment. We will conclude with an explanation of how to implement them with Cellular Evolution, in order to let you understand its potentiality.

## 5.1 Rules and behaviour

This is probably the most famous and studied cellular automaton, thanks to its simple definition, its evolution difficult to predict and the emerging of interesting structures and complex systems. The first two characteristics were exactly the ones that, in the '60s, John H. Conway wanted for his rule. To this end, Conway concentrated on meeting the following three criteria:

- it should be difficult to prove that a pattern grows without limit;
- not all simple initial states should immediately yield trivial final states;
- there should exist simple patterns that evolve for many iterations before settling into a simple final state.

After a great deal of experimentation, Conway finally settled on the well known two-dimensional rule.

**Definition 5.1.1.** *Game of Life* (GoL) is a two-dimensional cellular automaton that evolves in a squared lattice of cells binary-valued and uses Moore neighbourhoods. Its rules are the following ones:

- the *birth* of a cell (passage from 0 to 1) occurs if it has exactly three living cells in its neighbourhood;
- a cell encounters *death* (passage from 1 to 0) for isolation if it has less than two living cells in its neighbourhood, or for overcrowding when they are more than three;
- a living cell *survives* if it is surrounded by two or three living cells.

We can write these rules with the following formalism:

$$\begin{aligned} \sigma_{\underline{i}}(t+1) &= \Phi_{GoL}[\sigma_{\underline{j}}(t) \,|\, \underline{j} \in \mathcal{N}(\underline{i})] = \\ &= \begin{cases} 1 & \text{if } \sum_{\underline{j}\in\mathcal{N}(\underline{i})} \sigma_{\underline{j}}(t) = 3, \\ \sigma_{\underline{i}}(t) & \text{if } \sum_{\underline{j}\in\mathcal{N}(\underline{i})} \sigma_{\underline{j}}(t) = 4, \\ 0 & \text{otherwise.} \end{cases} \end{aligned} \tag{5.1.1}$$

What distinguishes this system from all the others and makes this rule truly remarkable is, as we will see, that GoL has been proven to be capable of *universal computation*. The principal consequence of this statement is that GoL can carry out arbitrary algorithmic procedures, so that it can be used in place of every standard digital computer. Furthermore this property gives us an important information about GoL dynamical complexity: this rule is actually capable of displaying arbitrarily complicated behaviour and generally there is no short-cut route to the final outcome of its evolution. Hence, in the past years, various initial configurations were analysed and catalogued, starting obviously from the simplest. Now, in order to understand the behaviour of the most famous CA, we will see some of these structures.

**Three live cells initial states**

Because live cells survive only if surrounded by 2 or 3 other live cells and dead cells become alive only if surrounded by exactly three live cells, initial states consisting of an isolated cell or two adjacent live cells are obviously destined to die after a single step. Hence we start with three adjacent live cells. Excluding possible symmetries and rotations there are only two cases: one returns a stable pattern called *block* while the other is a 2-period oscillator called *blinker* (see figure 5.1).

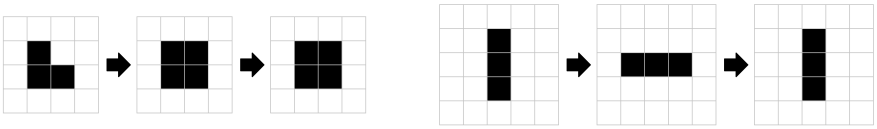

**Figure 5.1:** First steps of the possible three live cells initial states: on the left we obtain a block while on the right we can see a blinker.

### Stable and periodic patterns

As we have just seen in GoL there are patterns that remain unchanged unless other structures come closer to them and other that, if not disturbed, maintain a periodic behaviour. In figure 5.2 you can see some of these cases.

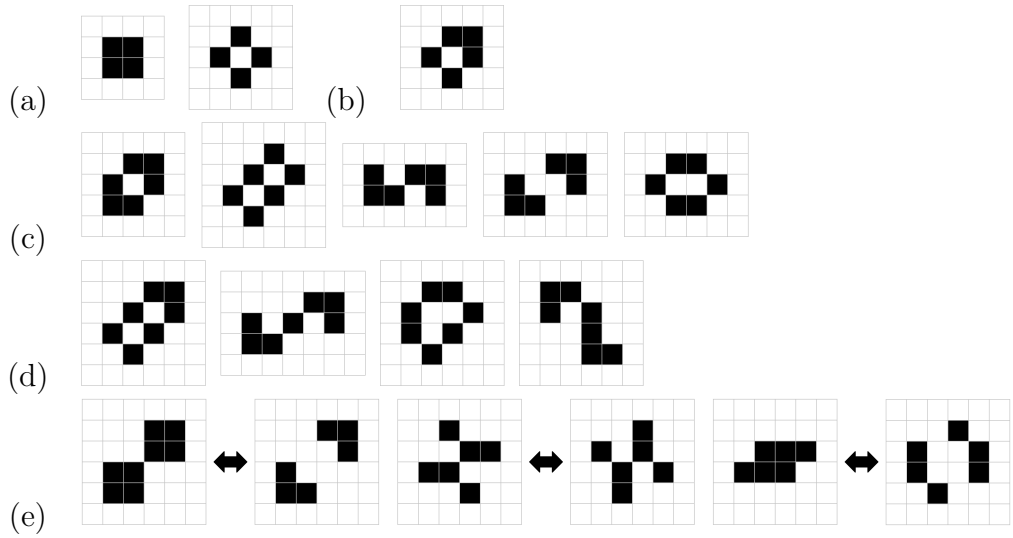


**Figure 5.2:** (a) stable patterns with four cells: a block and a tub; (b) a boat, the only stable five cells structure; (c) stable six cells patterns; (d) stable seven cells patterns; (e) some two-period oscillators: a beacon, a clock and a toad.

### Four live cells initial states

As you can see in figure 5.3, giving four initial live cells there are five possible configurations: one is the block (introduced previously), three of them after some steps return a stable pattern already seen in figure 5.2 called *beehive* and the last one after nine evolutions gives us a set of four blinkers.

### Five live cells initial states

Of the twelve possible initial states with five live cells, five yield the null state within four steps, two reach a stable state and four lead to the same group of four blinkers seen in figure 5.3. However, in marked contrast to these simple

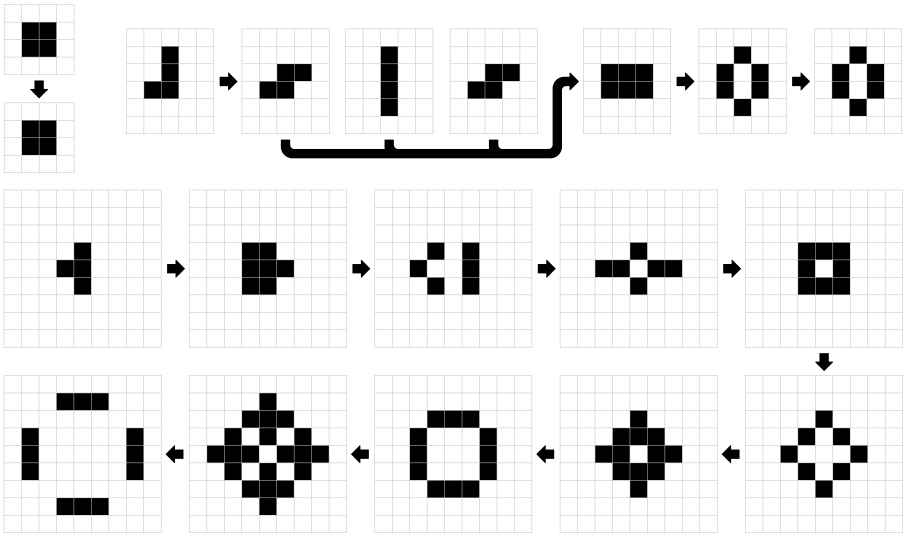

**Figure 5.3:** Evolution of four live cells patterns: the block, the beehive reached by three initial patterns and the bottom one that returns four blinkers.

behaviours, the remaining pattern, called the *R-pentamino*, evolves in a considerably fascinating way: in figure 5.4 we choose to show only its situation every ten steps, but its instability is clearly visible. This behaviour continues for 1103 steps, marking a time beyond which all of the various local patterns remain forever isolated and non-interacting.

**Gliders**
During the development of the R-pentamino a new kind of pattern starts running diagonally at generation 69 and we can see one of its frame in the last slide of figure 5.4. It is called *glider* because after two steps it produces a pattern that is both reflected in a diagonal line and down displaced by one site so that patterns separated by two steps are related by a glide reflection. Consequently the original pattern is reproduced in a diagonally displaced position every four iterations.

**Glider shuttle**
An interesting combination of distinct patterns consists of a single glider and two identical and stationary but oscillatory 12 live-cell patterns. Since the combination effect is that the glider continually shuffles its way from one to the other of these larger patterns, this pattern is called *glider shuttle*.

**Eaters**
A fascinating structure with the ability to destroy nearby patterns without damaging the integrity of its own form, is called the *eater*. For example in

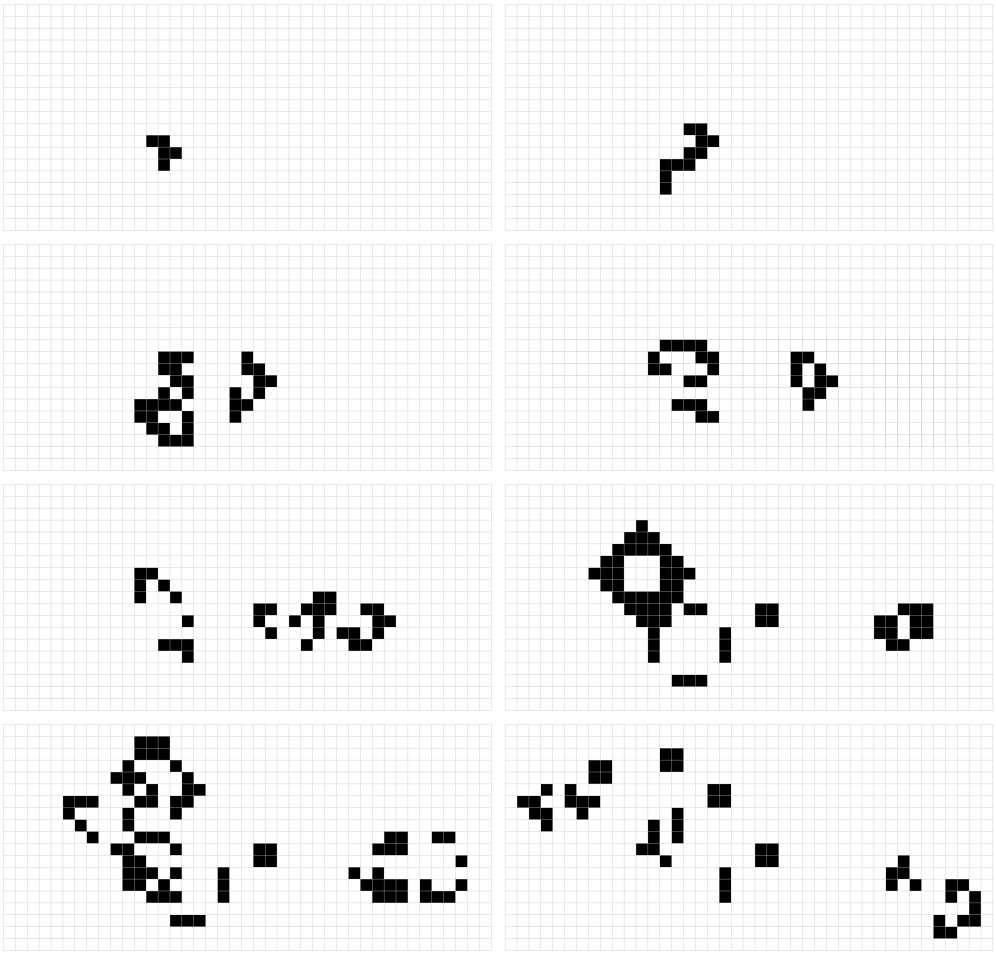

**Figure 5.4:** The evolution of the R-pentamino: steps 0, 10, 20, 30, 40 ,50, 60 and 70. In this last one a glider is visible in the bottom-right corner.

figure 5.7 we can see how an eater kills a glider in four time steps and repairs itself in the process. Another interesting behaviour is what happens when two eaters are made to attack each other: they produce an oscillating pattern in which each takes a 'bite' out of the other and quickly repairs itself before taking another bite. Although not everything can be successfully eaten by this pattern (for example the block is indestructible), the number of digestible patterns is impressively large.

**Glider-glider collisions**

While results of collisions with eaters are easily predictable, a glider-glider collision may have surprisingly different results: in all 73 distinct two-glider collisions are possible. In figure 5.8 we concentrate on two particular cases that we will use later: an *annihilation reaction*, in which both the gliders

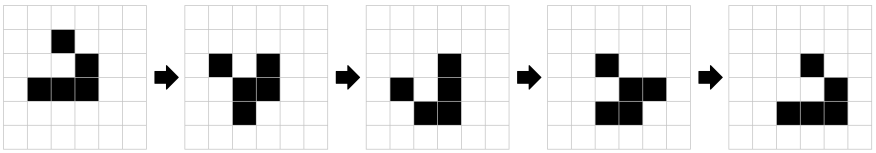

**Figure 5.5:** A glider in one of its shifts: after two steps it is diagonally reflected and shifted down by one site while after four steps its diagonal shift is completed.

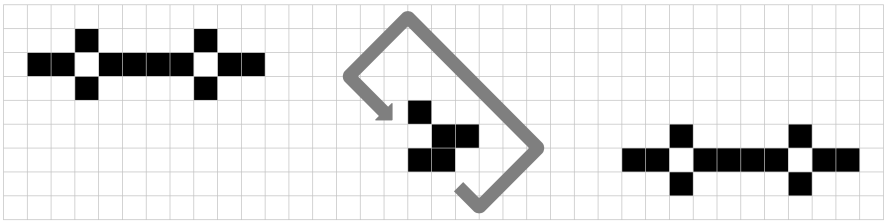

**Figure 5.6:** A glider shuttle in which a glider is bounced between two oscillators of period 15 called pentadecathlon.

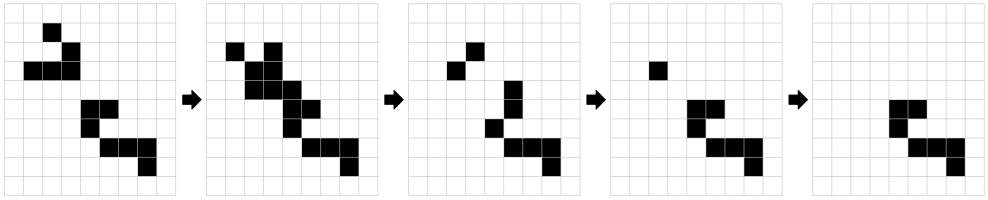

**Figure 5.7:** A glider that encounters an eater and is destroyed while the eater repairs itself.

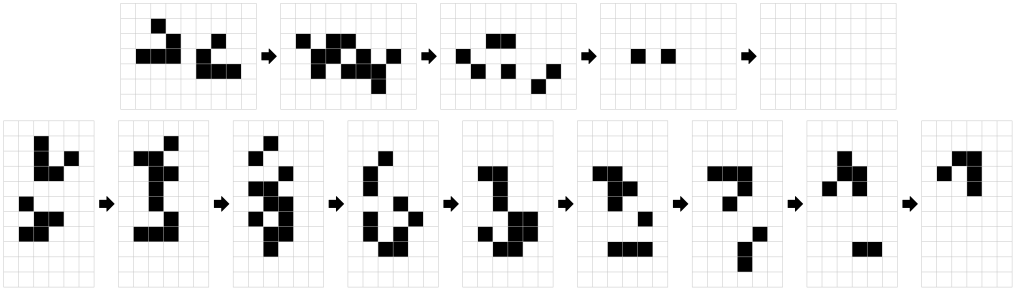

**Figure 5.8:** An annihilation and a kickback reaction in a glider-glider collision: in the first case both gliders die, in the second one a glider dies while the other one is bounced back.

die, and another in which one of the original gliders disappears and the other emerges out of the reaction zone shifted in space by half a diagonal lattice position and moving in the reversed direction. From this behaviour this second collision is called *kickback reaction*.

**Glider Gun**
Conway originally conjectured that no pattern can grow without limit, offering 50$ to the first person who either proved or disproved this claim. Although the task at first appeared to be very daunting the prize was won within a year of its announcement by a group of MIT students working for the Artificial Intelligence Project. After a great deal of experimentation, the group find an oscillatory configuration that each 30 steps produces a new glider: they had discovered a *glider gun* (figure 5.9). After collecting their prize, the members of this MIT group discovers that the glider gun itself can be manufactured out of a collision of 13 gliders.

Here we conclude our digression about the fauna that can be found in GoL. Obviously this was only a little part of it, containing the simplest, most important and useful patterns that this powerful system can generate. Nowadays all configurations with few cells have already been studied and classified but the community is still prolific, especially in studying the behaviour of bigger patterns.

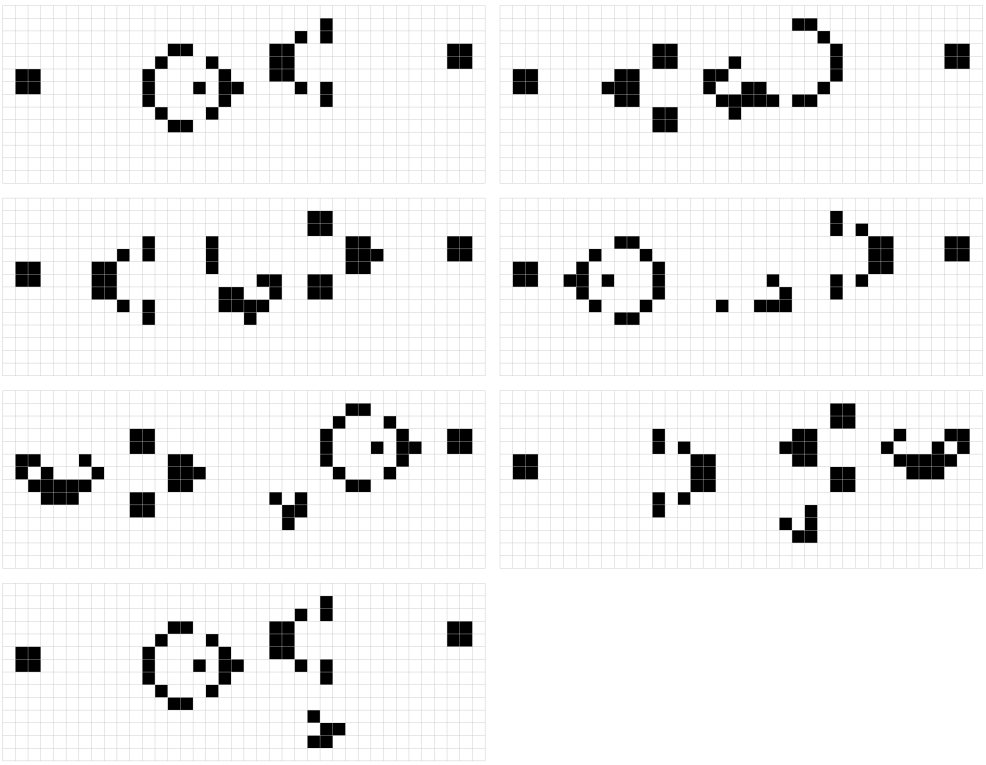

**Figure 5.9:** The evolution of a glider gun: starting at *t=0* we see its pattern each 5 steps. At *t=15* the first glider has just appeared and after more 15 steps we return to the initial configuration.

## 5.2 GoL's universality

Thanks to the previous section we have collected all the instruments to demonstrate that GoL is capable of universal computation. We will prove this property showing that Conway's rule is formally equivalent to another system that has already been proven to be a universal computer. In particular we will prove that each of the essential elements for computation (storage, transmission and processing of information) can be implemented by the evolution of patterns containing only gliders, glider guns, blocks and eaters.

While the precise design of a computer may be complex, its basic ingredients are relatively simple and few in number. It must have:

- some form of bit stream signals to work with;
- conduits for those signals, like wires;
- a way to route those signals, such as by redirection or sending multiple copies of a given bit stream into different directions;
- an internal system clock, or timer, to introduce any necessary delays in building circuits;
- a memory that can store arbitrarily large numbers;
- a set of universal logic gates, such as NOT, AND and OR, from which all other logical functions and operations can be obtained.

Once it can be shown that a given system supports these computational primitives, the construction of the actual working circuitry of a conventional computer becomes a simple formal exercise.

**Gliders as bits**
We begin by postulating the equivalence between, on the one hand, electrical pulses and bits of a pulse stream signal in physical computers and, on the other hand, gliders and glider streams in Conway's rule CA: in GoL we will interpret the presence or absence of an individual glider in a well-defined slot of a glider stream as denoting the corresponding bit value in a physical pulse stream signal. We already know that glider guns can create a stream of gliders, producing one glider every 30 iteration steps with 7,5 sites of distance between two of them. However, two or more glider streams moving on intersecting ways cannot cross without interferences. In order to build circuits we need non-interacting glider streams, hence we will produce a special glider gun that can create glider streams of arbitrarily long periods.

**Thin glider gun**

This particular glider gun is relatively easy to construct and depends on only four basic constructs: a basic glider gun, an eater, a glider-glider annihilation reaction and a kickback reaction. In figure 5.10 there is an diagram that wants to explain how it works. The thin glider gun uses two parallel but oppositely directed glider stream outputs of conventional glider guns (arrows 1 and 2 in the figure, triangles are gliders). A single glider also shuttles between those two glider streams. The timing is such that whenever the shuttling glider collides with a glider stream it undergoes a kickback reaction: it annihilates a glider in the glider stream and is itself shifted (upward if it collides on the left stream and downward otherwise) and reversed in direction. While the glider stream generated by gun 1 is not needed, and is

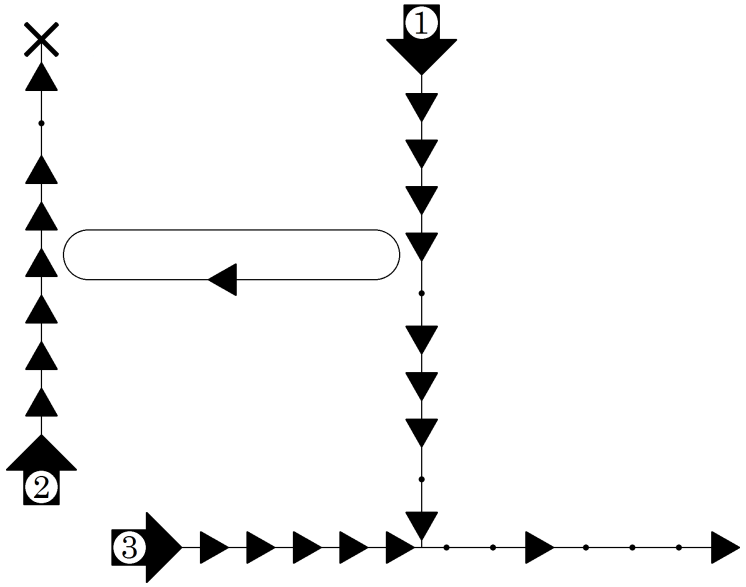


**Figure 5.10:** Structure of a thin glider gun with $n = 4$: arrows are glider guns, triangles gliders and the cross is an eater.

annihilated by an eater (represented with a cross), the stream generated by gun 2 encounters with a mutually annihilating reaction a third glider stream (arrow 3 in the figure). Supposing that every $n$-th glider on the right stream is annihilated by the single glider shuttling back and forth between kickback reactions on the left and right, the resulting glider stream that finally emerges out of the thin glider gun consists of a thinned set of gliders spaced $30 \cdot n$ steps apart. In order to get the right phase timing, $n$ must be divisible by four, but can be arbitrarily large. In this way we can therefore construct glider streams with gliders spaced arbitrarily far apart.

**Logic gates**
We can build a set of universal logic gates using the preceding constructs. We will start with **NOT**, seen in figure 5.11(a). First we assume that input A is some coded glider stream. Once in the gate, all of the gliders in this input stream annihilate when they collide with gliders in a complete glider stream produced by a thin glider gun constructed to have the same frequency of the input. The only gliders that emerge from the **NOT** gate are those that were in the original glider gun stream and were not annihilated by gliders in A hence a particular position of the output stream contains a glider if and only if the corresponding position in A did not.
In the **AND** gate (figure 5.11(b)) two input streams with same frequency, A and B, enter the gate in parallel. Gliders from a glider gun move on a perpendicular course and into an eater, with the timing such that the collisions between gliders in this glider stream with those in the two inputs are both annihilations. A glider makes it into the output stream if and only if both of the corresponding positions in A and B contain gliders.
The operation of the **OR** gate, the schematic of which is shown in figure 5.11(c), proceeds in a similar way: starting from the **AND** gate we have only to eliminate with an eater the previous resulting stream and put a thin glider gun, with the frequency of A and B, whose stream collides with the one that comes from the upper part.

**Time Delays**
In order to make logical gates operate properly, the gliders must be delayed by a time equal to the distance between the two annihilation reactions. Time delays of a signal are easily constructed out of **NOT** gates. Thanks to its 90° rotation of the stream it is possible to build a detour consisting of as many steps as needed for the delay, and then routing the signal stream back to its original course. Obviously, in order to return the original signal **NOT** gates must be used in oven quantities. Other configurations of NOT gates can delay and reroute signals by 180° and/or shift them into parallel paths. However, no combination of **NOT** gates can be made to turn a stream sideways. Fortunately, this capability comes with the following construction.

**A Glider-Stream Copier**
The set of tools that we have defined thus far allow us only to reroute single glider streams into parallel directions of motion but there exists as yet no provision either for offsetting the direction of a given signal by 90° or for making copies of a signal. One of the simplest circuits that solves both of these problems is due to Conway and is shown in figure 5.12. The idea is to use the kickback reaction to send gliders back into their own stream.

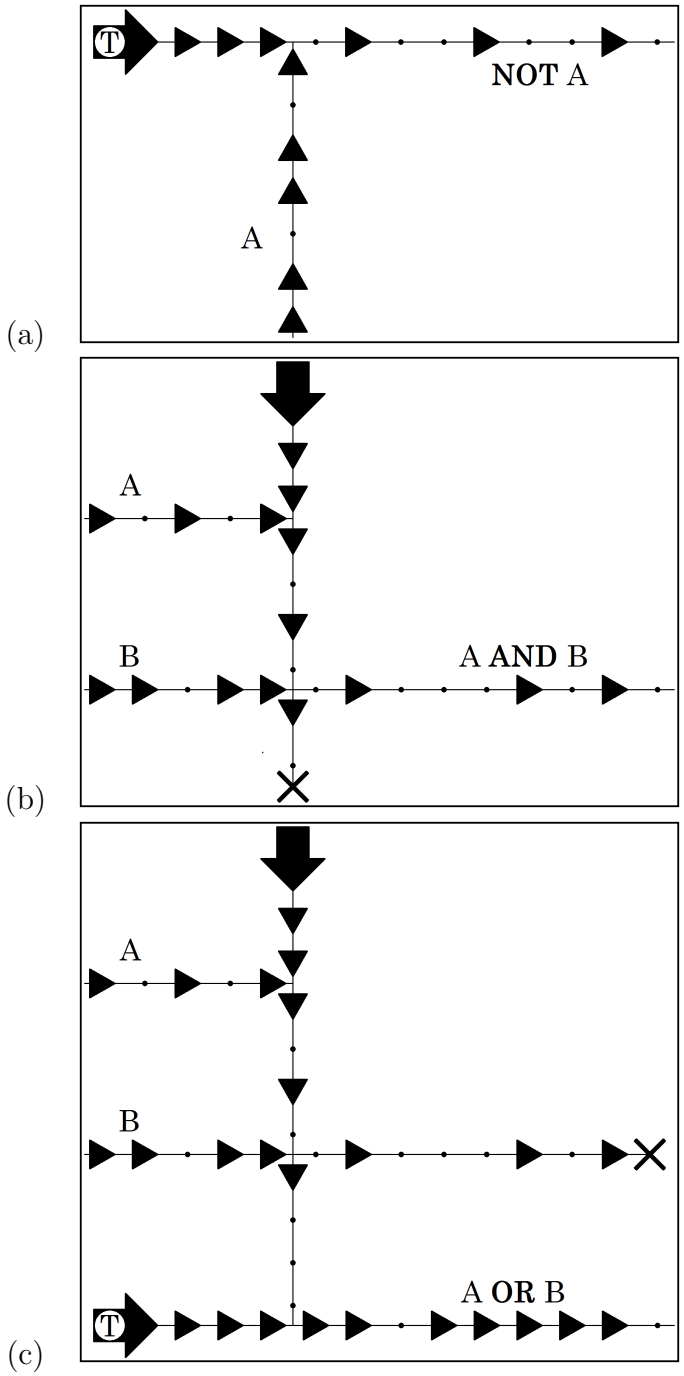


**Figure 5.11:** (a) **NOT** gate with A=110110...; (b) **AND** gate between A=1010... and B=110110...; (c) **OR** gate between A=1010... and B=110110... . In (a) and (c) the arrow signed with T is a thin glider gun.

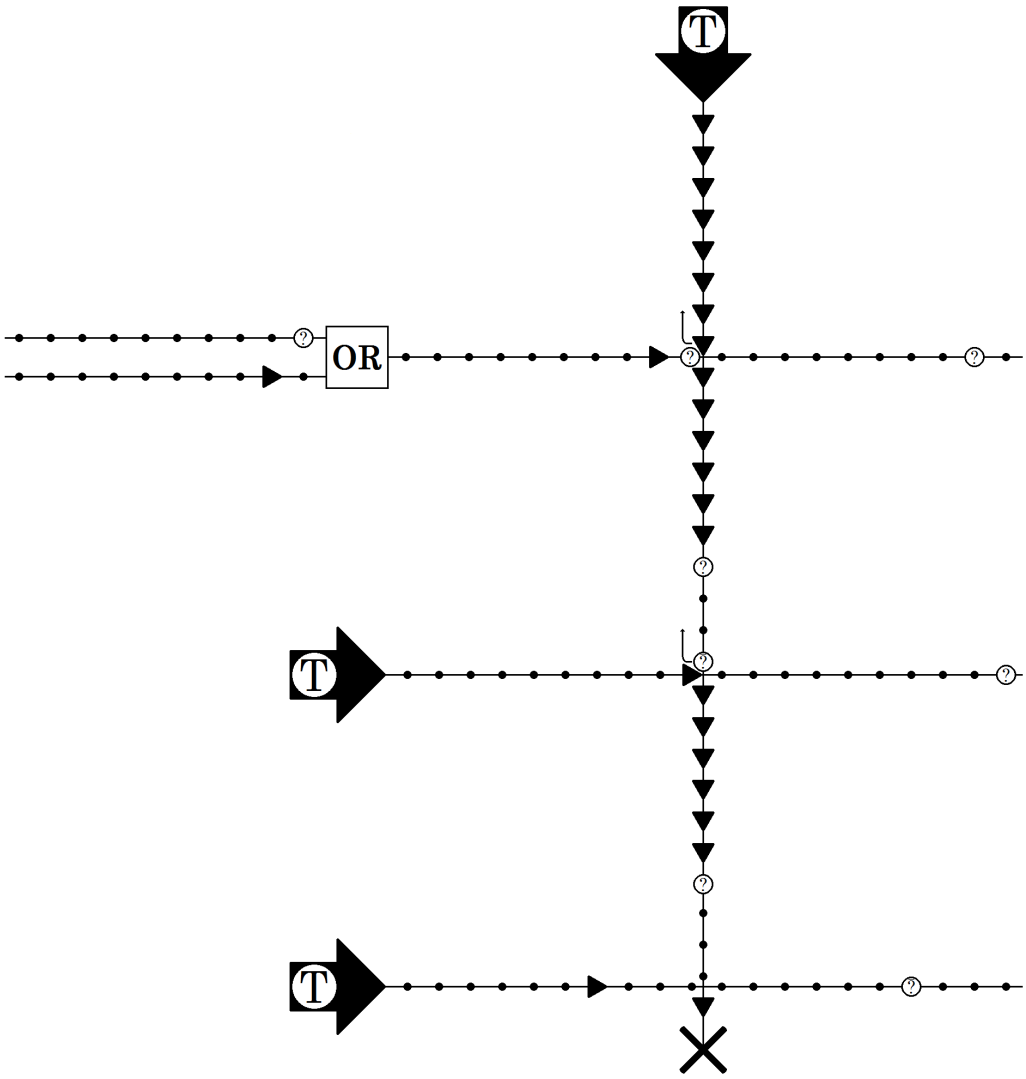


**Figure 5.12:** Conway's stream copier: the input is the first stream entering the **OR** gate and there are three outputs. As before arrows signed with T are thin glider guns

We need a factor 4 thin glider gun, whose complete glider stream is four time less frequent than one of a normal glider gun. We also want that the coded signal stream has been thinned by another factor of 10 (40 total), so that nine positions are always empty and it is only the tenth position that may or may not contain a glider. In the figure, we have indicated the value of this tenth glider-bit position with a question-mark (000000000?). Conway's circuit functions as follows: the input data stream that is to be copied enters an **OR** gate with the output of a thin-glider-gun but whose glider stream

is such that the ninth glider position always contains a glider (0000000010). Hence the output consists of a glider stream that always has the same glider set as does the original coded signal and always contains a glider in the ninth position (000000001?).
Now consider what happens when this string encounters a kickback reaction: if ? = 1, it will kickback the first glider in the downward stream which creates a pile-up collision that will annihilate three gliders in the vertical stream. Also the coded glider is annihilated but the glider behind it will pass through the kickback reaction region unaffected. Hence the output is the string 0000000010. If ? = 0, the first glider in the descending stream will pass through unaffected but the ninth glider behind the coded glider position will kickback the second one which causes the same reaction as before. In this case the output stream contains no gliders (000000000). In short, if the coded glider position contains a glider bit ?, the output of the kickback-reaction is given by 00000000?0, then it contains the same information as the original signal but the coded glider bit is delayed by one position.
Let us see what happens to the downward moving glider stream after it enters the kickback reaction region. As we said this collision always removes exactly three gliders. In particular, if ? = 1 then the kickback reaction strips the first three gliders off of the downward stream while if ? = 0 then the leading glider of the downward stream is left alone but the next three gliders are stripped off by the kickback reaction. Schematically, the output is given by $111111?00\bar{?}$, where $\bar{?} = \mathbf{NOT}(?)$. This descending stream collides with a transverse thin glider stream consisting of all empty glider positions except for the tenth, which always contains a glider (0000000001). This collision is again a kickback whose output stream can have a glider only in his last position, and only when the $\bar{?}$ position in the downward stream is empty. In other words, the leading position of the output at right is $\mathbf{NOT}(\mathbf{NOT}(?)) =?$, so that the output is a perfect replica of the original coded glider stream: 000000000?.
In a similar way, the collision between the downward output of this collision (111111?000) and a transverse thinned signal 0000001000 produces a negated copy of the original signal ($000000\bar{?}000$).

Conway's circuit therefore has a total of three outputs: one is an exact replica, another is a faithful copy but with the glider carrying the information shifted by one position, and the third is a negated copy of the original signal with the information glider shifted by three positions. Notice that all three outputs remain parallel to the direction of motion of the original signal. Hence we can use Conway's basic signal copying circuit with a **NOT** gates to reroute a signal by 90°.

**Memory Storage**
While a finite memory is fairly easy to implement with wires and logic gates (for example, glider stream informations can be made to circulate around a memory circuit contained within the computer) the construction of an arbitrarily large memory requires a bit more work. A universal Turing machine uses an arbitrarily long tape as a potentially infinite memory storage device. Instead, for his proof, Conway used Minsky's idea that a potentially infinite memory can also be obtained by storing arbitrarily large numbers in memory registers.
A simple way of keeping track of large numbers that the computer may have to remember is to use stable blocks as value markers in an auxiliary memory storage register. The distance of the block in a given register with respect to an arbitrarily designated level 0 defines the quantity stored in that register. Suppose register $x$ contains a block at location $N$. In order for the computer to read this value, it must perform the loop

- decrease contents of register $x$ by 1;
- test to see if contents of register $x = 0$;

until the contents of register $x$ is equal to zero. The number of internal iterations required to decrease the distance of the given block from 0 then yields the desired register memory value. Conway showed that all of the required block movements (push to register and pull to read) could be accomplished by a suitable flotilla of gliders: while there does not exist any single glider-block collision capable of displacing a block by its site, it may be shown that ten gliders can be coerced into moving a block by one diagonal lattice site (see [Berk82]).

Having demonstrated, by construction, that each of the computational elements required of a conventional digital computer for its own computation (namely digital bit stream signals, wires, redirection circuits, an internal system clock, a potentially infinite memory and a set of universal logic gates containing **NOT**, **AND** and **OR**) is supported by GoL's dynamics, we have thus proven that Life is universal. Indeed, from the above discussion, it is clear that the circuitry of any computer base on Conway's rule can be built entirely out of gliders, glider-guns, eaters and blocks. Hence Conway's rule can really be used for computation and every kind of algorithm can be implemented with it.

## 5.3 GoL with Cellular Evolution

Now that we are more familiar with Conway's rule it is possible to talk about its implementation using Cellular Evolution. Our system can be customized changing its two operators: the fitness function and the crossover matrix. Here we want to show one of the possible choices to obtain Game of Life in Cellular Evolution.

Since GoL cells can assume only two values, we consider $\Sigma = \{0, 1\}$. We remind that, according to Conway's rule, a dead cell becomes alive if there are exactly two live cells in its neighbourhood while a cell stays alive if there are two or three live cells in its neighbourhood. In particular we observe that the future state of a cell depends only by the state of its neighbourhood. Hence we choose to use a fitness function with values in $\mathcal{F} = \{0, 1\}$ that describes the will of each cell: if its fitness value is 0 then the cell wants to die or stay dead while if it is 1 the cell will become or stay alive. Unfortunately the simplicity of GoL rule leads to a trivialization of the crossover operator: given a fitness value it simply makes the wish of the cell comes true, without regard to the state or the fitness value of the mate chosen by the cell.

Practically the operators for implement GoL in Cellular Evolution are:

- $f : \Sigma^9 \longrightarrow \mathcal{F}$ given by

$$f_{\underline{i}}(t) = \Phi_{GoL}(\mathcal{C}_{\underline{i}}(t)) = \begin{cases} 1 & \text{if } \sum_{\underline{j} \in \mathcal{N}(\underline{i})} \sigma_{\underline{j}}(t) = 3, \\ \sigma_{\underline{i}}(t) & \text{if } \sum_{\underline{j} \in \mathcal{N}(\underline{i})} \sigma_{\underline{j}}(t) = 4, \\ 0 & \text{otherwise.} \end{cases} \tag{5.3.1}$$

- $\chi : \Sigma \times \mathcal{F} \times \Sigma \times \mathcal{F} \longrightarrow \Sigma$ where

$$\chi(\sigma_{\underline{i}}(t), f_{\underline{i}}(t), \sigma_{\underline{j}}(t), f_{\underline{j}}(t)) = f_{\underline{i}}(t) \tag{5.3.2}$$

  or, giving all the possible cases, we have the table

| $\chi$ | | $(\sigma_{\underline{i}}(t), f_{\underline{i}}(t))$ | | | |
|---|---|---|---|---|---|
| | | (0,0) | (0,1) | (1,0) | (1,1) |
| $(\sigma_{\underline{j}}(t), f_{\underline{j}}(t))$ | (0,0) | 0 | 1 | 0 | 1 |
| | (0,1) | 0 | 1 | 0 | 1 |
| | (1,0) | 0 | 1 | 0 | 1 |
| | (1,1) | 0 | 1 | 0 | 1 |

In conclusion we were able to obtain Conway's rule in our system, which means that, just like it, Cellular Evolution is capable of universal computation.

## 5.4 Two evolutions of GoL

Here we want to introduce two new evolutions of Conway's Game of Life. In particular these new rules maintain the bi-dimensional lattice but consider more than two possible states while the evolutions are based on the original one. These systems are inspired by natural interactions between two species. As we did before we will implement these new CA with our system, in order to show its potentiality.

### 5.4.1 Three states evolution

In this case there are three possible states: $\Sigma = \{0, 1, 2\}$ where 0 means dead while 1 and 2 are the two possible living states. The evolution is described by the following rules:

- a dead cell ($\sigma_{\underline{i}}(t) = 0$) changes state if there are exactly 3 live cells in its neighbourhood. In that case its state value becomes 1 if in its neighbourhood there are more cells with state value 1 than whose with state value 2, and 2 otherwise;
- a live cell ($\sigma_{\underline{i}}(t) > 0$) stay unchanged if there are 2 or 3 live cells in its neighbourhood, otherwise it dies.

We can summarize the evolution of the system considering for each cell $\underline{i}$ two values depending on the states of the cells in its neighbourhood namely $\mathbf{1}_{\underline{i}}(t) = |\{\underline{j} \in \mathcal{N}_{\underline{i}}(t) | \sigma_{\underline{j}}(t) = 1\}|$ and $\mathbf{2}_{\underline{i}}(t) = |\{\underline{j} \in \mathcal{N}_{\underline{i}}(t) | \sigma_{\underline{j}}(t) = 2\}|$. Hence the dynamic rule becomes

$$\sigma_{\underline{i}}(t+1) = \Phi_3[\sigma_{\underline{j}}(t) \,|\, \underline{j} \in \mathcal{N}(\underline{i})] = \\ = \begin{cases} 1 & \text{if } \sigma_{\underline{i}}(t) = 0,\ \mathbf{1}_{\underline{i}}(t) + \mathbf{2}_{\underline{i}}(t) = 3 \text{ and } \mathbf{1}_{\underline{i}}(t) > \mathbf{2}_{\underline{i}}(t), \\ 2 & \text{if } \sigma_{\underline{i}}(t) = 0,\ \mathbf{1}_{\underline{i}}(t) + \mathbf{2}_{\underline{i}}(t) = 3 \text{ and } \mathbf{1}_{\underline{i}}(t) < \mathbf{2}_{\underline{i}}(t), \\ \sigma_{\underline{i}}(t) & \text{if } \sigma_{\underline{i}}(t) > 0 \text{ and } 1 < \mathbf{1}_{\underline{i}}(t) + \mathbf{2}_{\underline{i}}(t) < 4, \\ 0 & \text{otherwise.} \end{cases} \tag{5.4.1}$$

For the implementation using Cellular Evolution we can use operators similar to whose used for original GoL:

- $f : \Sigma^9 \longrightarrow \mathcal{F}$ given by $f_{\underline{i}}(t) = \Phi_3(\mathcal{C}_{\underline{i}}(t))$;
- $\chi : \Sigma \times \mathcal{F} \times \Sigma \times \mathcal{F} \longrightarrow \Sigma$ where

$$\chi(\sigma_{\underline{i}}(t), f_{\underline{i}}(t), \sigma_{\underline{j}}(t), f_{\underline{j}}(t)) = f_{\underline{i}}(t) \tag{5.4.2}$$

  whose possible cases are described in the table of the previous section.

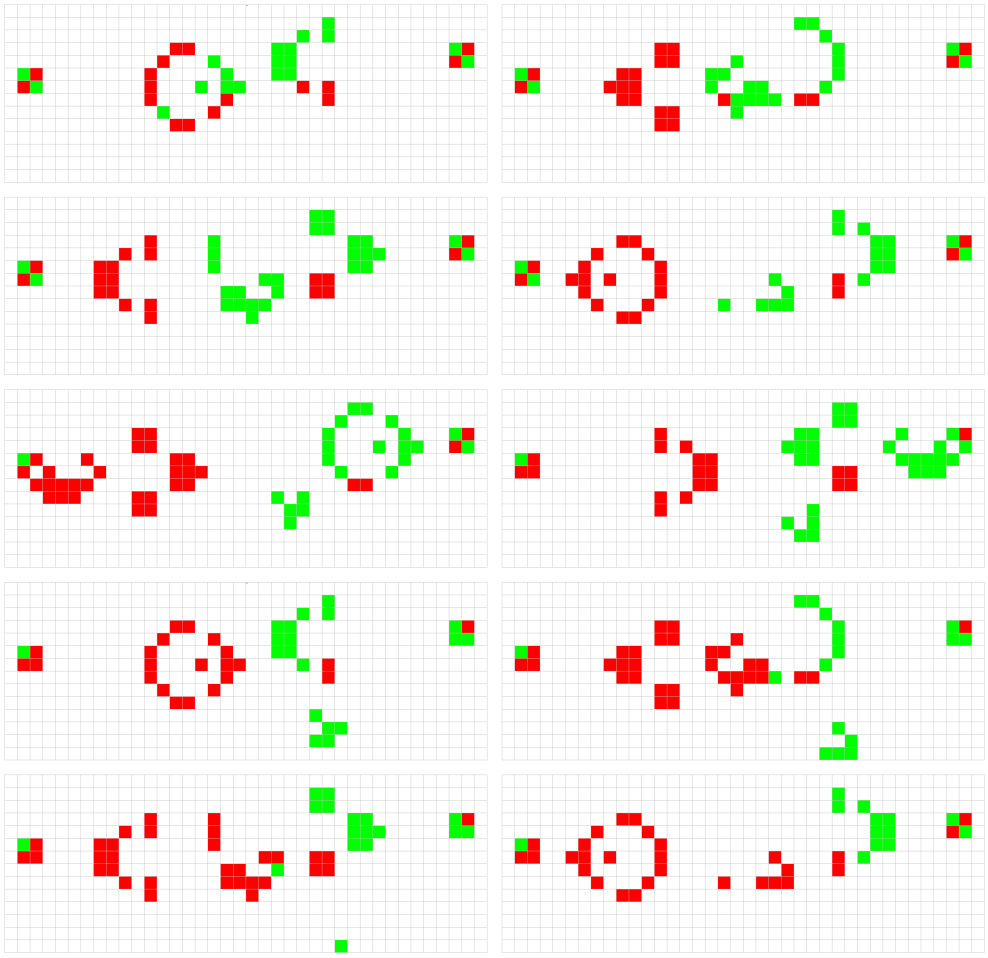

**Figure 5.13:** A version of the glider gun in three colors GoL seen each 5 time-steps: it spawns one green glider followed by infinite red ones.

### 5.4.2 Four states evolution

This case differs from the previous one since there are four possible state value: $\Sigma = \{0, 1, 2, 3\}$ where 0 and 1 are death and live states of the first form of life while 2 and 3 are death and live states of the second one. In this case the two form of life will evolve and live in symbiosis hence they will help each other to stay alive. The rules that describe the behaviour of this system are given by:

- a dead cell of the first form of life ($\sigma_{\underline{i}}(t) = 0$) will become alive (with state value 1) if in its neighbourhood there are exactly 3 cells with state value 1 or if there are 3 generic live cells (states 1 or 3) (or equivalently 5 generic dead cells);

- for a dead cell of the second form of life ($\sigma_{\underline{i}}(t) = 2$) the behaviour is analogous: it will become alive ($\sigma_{\underline{i}}(t+1) = 3$) if in its neighbourhood there are exactly 3 cells with state value 3 or 3 generic live cells (states 1 or 3) (or equivalently 5 generic dead cells);
- a live cell of the first form of life ($\sigma_{\underline{i}}(t) = 1$) stay alive if in its neighbourhood there are 2 or 3 cells with its state value or if there is a quantity between 1 and 4 of cells with state value 3, otherwise it dies;
- as for the dead case, the behaviour of a live cell of the second form of life ($\sigma_{\underline{i}}(t) = 3$) is similar: it stay alive if in its neighbourhood there are 2 or 3 cells with state value 3 or if there is a quantity between 1 and 4 of cells with state value 1, otherwise it dies.

As we did before, in order to summarize the evolution of this system, we will consider for each cell $\underline{i}$ two values that describe the situation of its neighbourhood: $\mathbf{1}_{\underline{i}}(t) = |\{\underline{j} \in \mathcal{N}_{\underline{i}}(t) | \sigma_{\underline{j}}(t) = 1\}|$ and $\mathbf{3}_{\underline{i}}(t) = |\{\underline{j} \in \mathcal{N}_{\underline{i}}(t) | \sigma_{\underline{j}}(t) = 3\}|$. They allow us to write the dynamic rule as follows:

$$\sigma_{\underline{i}}(t+1) = \Phi_4[\sigma_{\underline{j}}(t) \,|\, \underline{j} \in \mathcal{N}(\underline{i})] = \\ = \begin{cases} 0 & \text{if } \sigma_{\underline{i}}(t) = 1,\ \mathbf{1}_{\underline{i}}(t) \notin \{2,3\} \text{ and } \mathbf{3}_{\underline{i}}(t) \notin \{1,2,3,4\}, \\ 1 & \text{if } \sigma_{\underline{i}}(t) = 0 \text{ and } \mathbf{1}_{\underline{i}}(t) = 3 \text{ or } \mathbf{1}_{\underline{i}}(t) + \mathbf{3}_{\underline{i}}(t) = 3, \\ 2 & \text{if } \sigma_{\underline{i}}(t) = 3,\ \mathbf{3}_{\underline{i}}(t) \notin \{2,3\} \text{ and } \mathbf{1}_{\underline{i}}(t) \notin \{1,2,3,4\}, \\ 3 & \text{if } \sigma_{\underline{i}}(t) = 2 \text{ and } \mathbf{3}_{\underline{i}}(t) = 3 \text{ or } \mathbf{1}_{\underline{i}}(t) + \mathbf{3}_{\underline{i}}(t) = 3, \\ \sigma_{\underline{i}}(t) & \text{otherwise.} \end{cases} \tag{5.4.3}$$

Now we can try to implement this system with Cellular Evolution. We may use the same operator used since now but instead we choose to change a bit both of them. As we did in the original GoL case we still use a fitness function that describes the will a cell to become or stay alive but its possible values are only two: $\mathcal{F} = \{0, 1\}$. While the fitness value gives the will, the crossover matrix decides, depending on the current state of the cell, what is its form of life and consequently what will be its new state value. In formulas we have:

- $f : \Sigma^9 \longrightarrow \mathcal{F}$ given by
$$f_{\underline{i}}(t) = \Phi_4(\mathcal{C}_{\underline{i}}(t)) (\text{mod } 2) \tag{5.4.4}$$
- $\chi : \Sigma \times \mathcal{F} \times \Sigma \times \mathcal{F} \longrightarrow \Sigma$ where
$$\chi(\sigma_{\underline{i}}(t), f_{\underline{i}}(t), \sigma_{\underline{j}}(t), f_{\underline{j}}(t)) = \begin{cases} f_{\underline{i}}(t) & \text{if } \sigma_{\underline{i}}(t) \in \{0,1\}, \\ f_{\underline{i}}(t) + 2 & \text{if } \sigma_{\underline{i}}(t) \in \{2,3\}. \end{cases} \tag{5.4.5}$$

Considering every possible case we have the table

| $\chi$ | | $(\sigma_{\underline{i}}(t), f_{\underline{i}}(t))$ | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | (0,0) | (0,1) | (1,0) | (1,1) | (2,0) | (2,1) | (3,0) | (3,1) |
| $(\sigma_{\underline{j}}(t), f_{\underline{j}}(t))$ | (0,0) | 0 | 1 | 0 | 1 | 2 | 3 | 2 | 3 |
| | (0,1) | 0 | 1 | 0 | 1 | 2 | 3 | 2 | 3 |
| | (1,0) | 0 | 1 | 0 | 1 | 2 | 3 | 2 | 3 |
| | (1,1) | 0 | 1 | 0 | 1 | 2 | 3 | 2 | 3 |
| | (2,0) | 0 | 1 | 0 | 1 | 2 | 3 | 2 | 3 |
| | (2,1) | 0 | 1 | 0 | 1 | 2 | 3 | 2 | 3 |
| | (3,0) | 0 | 1 | 0 | 1 | 2 | 3 | 2 | 3 |
| | (3,1) | 0 | 1 | 0 | 1 | 2 | 3 | 2 | 3 |

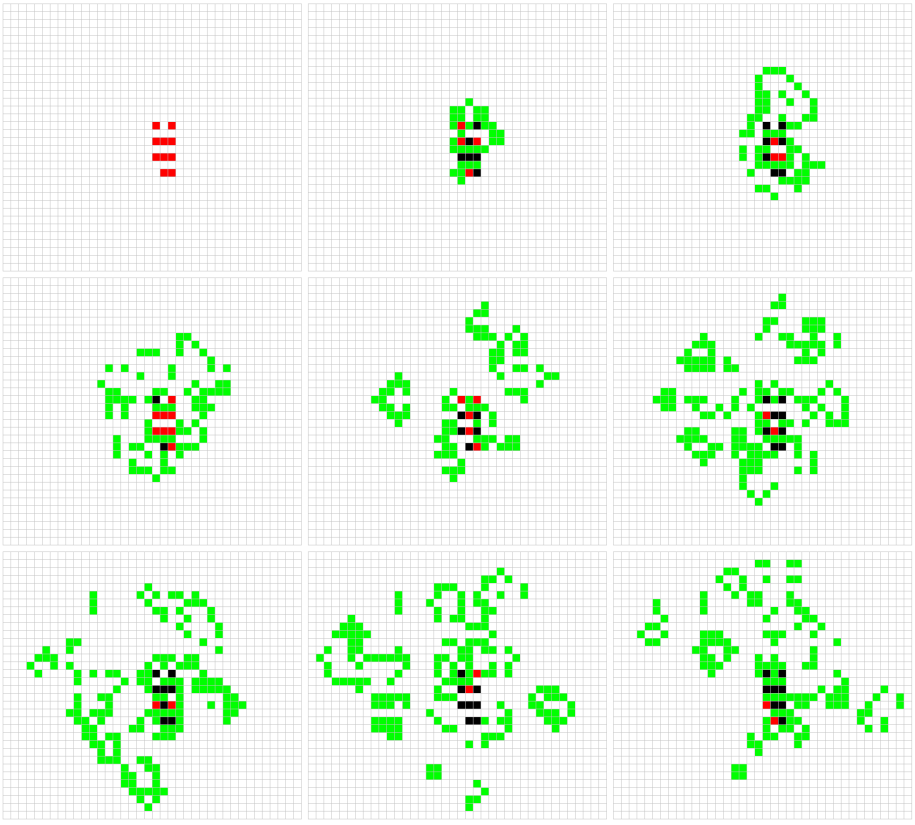

**Figure 5.14:** Steps 0, 10, 20, ... of a life generator in the four colors evolution of GoL: these simple red structure creates infinite patterns of green cells (like the R-pentamino seen in the original GoL).

This two examples have really interesting behaviour and certainly they can still be studied in order to find other wonderful properties, but at the moment we use them only to give us an idea of the potential of Cellular Evolution.

# Chapter 6

# Conclusions

In this work we wanted to introduce and give some properties of a new kind of cellular automaton inspired by genetic algorithms: we created a powerful, flexible and interesting system that has really surprising and amazing behaviours and we called it Cellular Evolution.

In particular we concentrated on its definition and on some useful implementations, like the well known Game of Life created by Conway that gives us the property of universal computation, but this brand new system still hides lots of curious and maybe useful developments.

However, in this paper, you have seen only a little part of our studies: we showed only trivial fitness functions and crossover matrices in order to obtain some specific results, but the work behind this new creation has just begun.

Considering also the evolutions of Game of Life, our future goals are primarily three:

- deepen the knowledges about GoL evolutions, for example by classifying the behaviours of arbitrary structures;
- consider cases of medium complexity characterized by crossover functions depending only on state values;
- study the most generic cases with crossover operator depending on both fitness and state values.

Now we are concentrating on the second step and the following pictures, created with our open-source program, show some of the resulting situations that can be reached with this kind of functions and the evolution of a particular case. If you are interested, our works (in Matlab and Python) can be found in [CEDr17] where we upload and keep updated all our files.

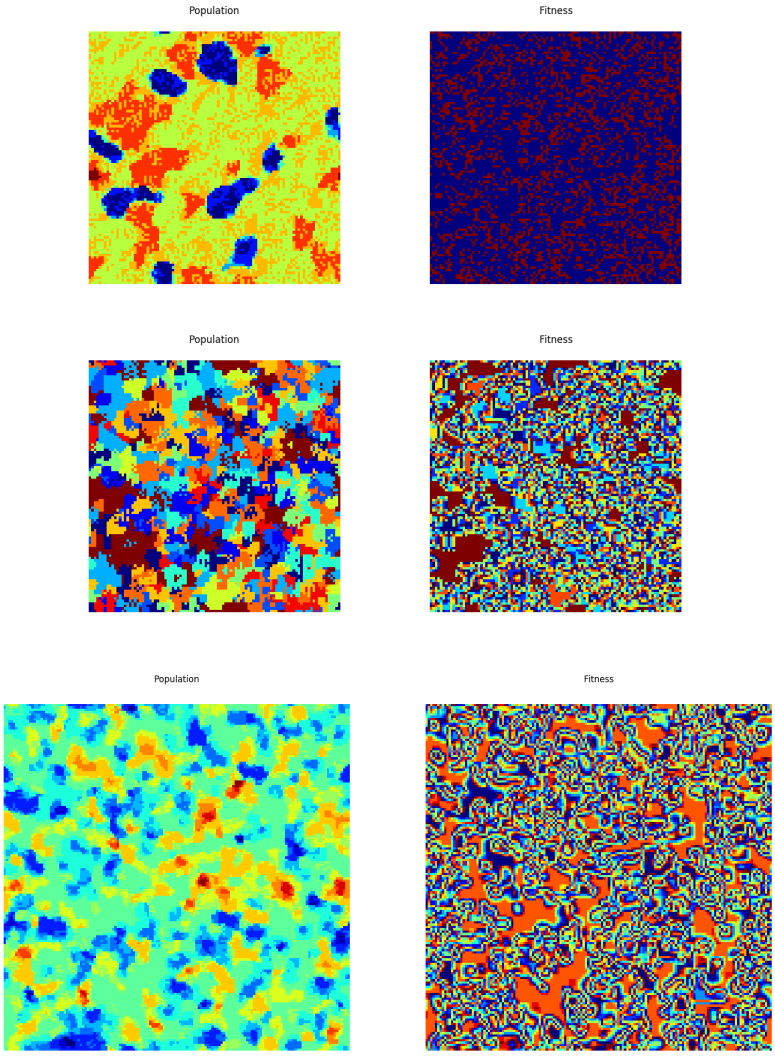


**Figure 6.1:** Some awesome shots generated by Cellular Evolution.

As you can see the results are really amazing and sometimes they may be clearly associated to natural situations but we have still to continue studying the system to say something more specific. Anyhow it has a big potential and we hope that, if not us, the community will find some specific applications that can improve predictions and creation of models in other disciplines.

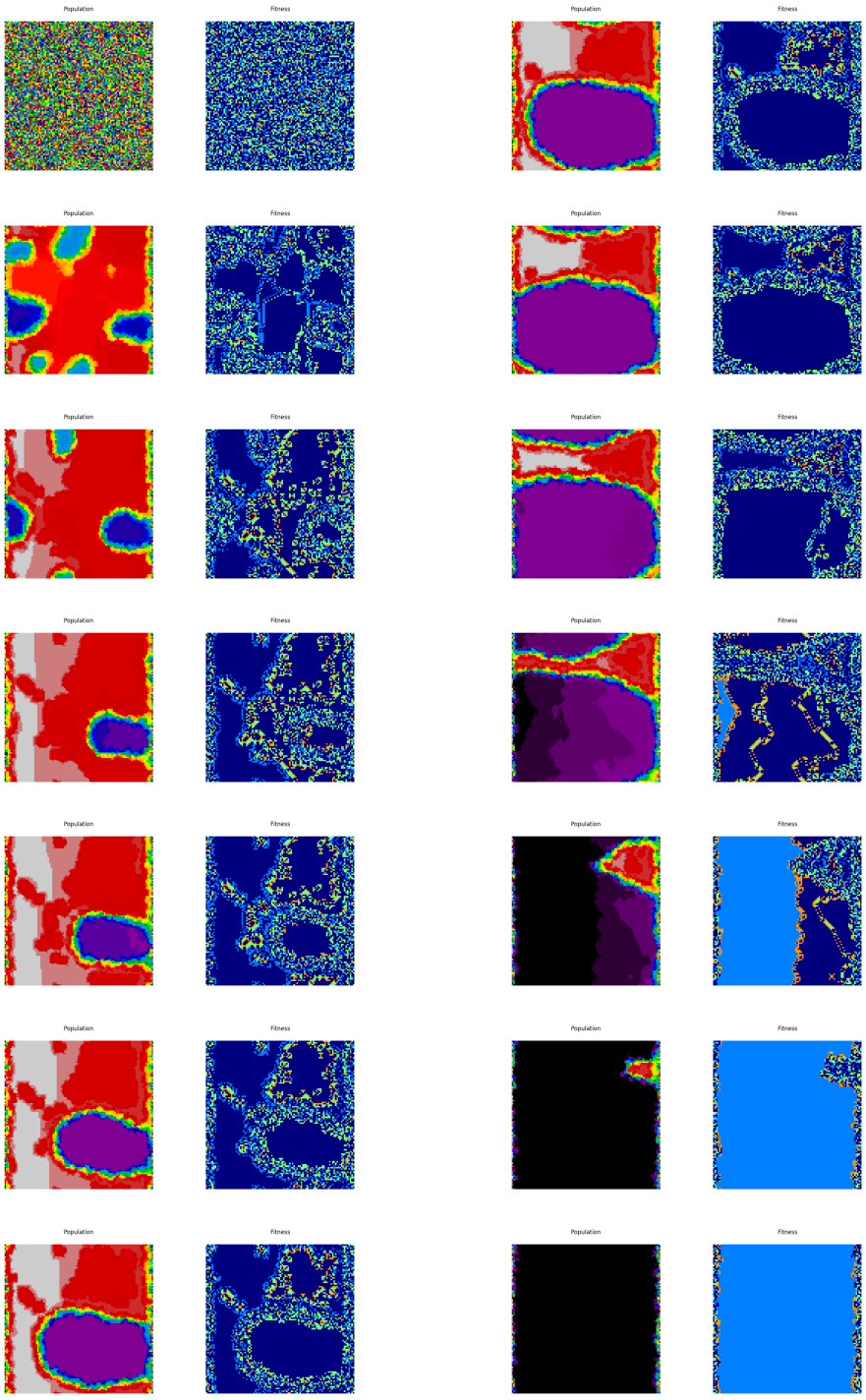


**Figure 6.2:** An interesting case developed with Cellular Evolution.

# Chapter 7

# Bibliography


- Berkelamp, E.R., Conway, J.H., & Guy, R.K. (1982). *Winning Ways for Your Mathematical Plays*. Academic Press. [Berk82]
- Gardner, M. (1970). *Mathematical Games: The Fantastic Combinations of John Conway's New Solitaire Game 'Life'*. Scientific American 223(4), 120-123. [Gard70]
- Goldberg, D.E. (1989). *Genetic algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Publishing Company. [Gold89]
- Ilachinski, A. (2001). *Cellular automata: a discrete universe*. World Scientific Publishing Co Inc. [Ilac01]
- Owens, N., & Stepney, S. (2010). *The Game of Life rules on Penrose tilings: still life and oscillators*. Game of Life Cellular Automata, 331-378. [OwSt10]
- Wolfram, S. (2002). *A new kind of science*. Champaign: Wolfram media. [Wolf02]
- Link to our works [CEDr17]:
  https://drive.google.com/drive/folders/
  0B9aDuGCGOiwYdERTUGs4WG9Sb3M