
**UNIVERSITA' DEGLI STUDI DELL'INSUBRIA**

**Facoltà di Scienze MM.FF.NN.**

Corso di Laurea Specialistica in Informatica

**NORWEGIAN UNIVERSITY OF SCIENCE AND TECHNOLOGY**

**Department of Computer and Information Science**

Master's Thesis in Computer Science




## Stefano Nichele


# Trajectories and attractor basins as a behavioral description and evaluation criteria for artificial EvoDevo systems


Supervisor NTNU

**Gunnar Tufte**

Supervisor UNINSUBRIA

**Claudio Gentile**



Trondheim, June 2009

*The mystery of life isn't a problem to solve,*
*But a reality to experience.*

Frank Herbert, *Dune*

Master Thesis



Università degli Studi dell'Insubria – Varese, ITALY
Facoltà di Scienze Matematiche, Fisiche e Naturali
Corso di Laurea Specialistica in Informatica

NTNU - Norges Teknisk-Naturvitenskapelige Universitet – Trondheim, NORWAY
Faculty of Information Technology, Mathematics and Electrical Engineering
Department of Computer and Information Science

# ABSTRACT

In a traditional von Neumann architecture, computation is based on the principle of a program being sequentially executed on a single complex processor. However, alternative architectures for computation exist and may be more effective (e.g. improved speed, scalability and technology independence).

Cellular Computing is such an alternative computation system exploiting a vast amount of simple computation elements operating in parallel with only local interconnections. In fact, each computing element has access only to the states of a small number of neighbors.

These new paradigms, such as cellular computing, may offer massive computation power. However, the potential is hard to exploit. Logical design of hardware and lack of programming methods make it difficult to unleash the potential computational power.

As an alternative to today's top-down design approach, an adaptive approach such as Evolutionary Algorithm (EA), shows promising results as a design tool for such systems. However, EAs alone lack the scalability required to solve the task of designing the hardware and set up the running conditions required for realistic computation problems. One solution to increasing the level of design complexity of EAs is to take inspiration from nature's way of handling complexity. Nature's process of development, where a single cell can develop to a multi-cellular organism, can be included in an EA approach toward creation of cellular machines.

This massive parallel operation of a cellular computation machine, in combination with the natural parallel process of artificial development, require enormous computation power if it has to be simulated on a traditional computer architecture. To be able to run tests in a realistic time a customized hardware platform for experimentation in artificial development towards computational circuits has been developed and designed at CRAB Lab [1] [2]. This platform is a product of several years of research. It is designed using Field Programmable Gate Arrays (FPGAs). The internal architecture of FPGAs is very close to the computational paradigm using simple units with only local interconnections.

This project is not focused on the hardware but rather on the understanding of trajectories and attractors basins as a behavioral description and evaluation criteria for such artificial evolution and development systems (EvoDevo) [3]. EvoDevo systems are analyzed and interpreted as a discrete dynamical system, where each event represents a point in time in the developmental path from zygote to multi-cellular organism [4]. One dimensional uniform Cellular Automata (CA) and one dimensional non-uniform Cellular Automata are chosen as a theoretical and experimental platform. The first main task is to generate, using a genetic approach, CA rules that can produce a specified trajectory and reach a defined attractor. The second goal is to investigate different time-scales, timing paradigms and state abstractions.

# TABLE OF CONTENTS

ABSTRACT ..... 3

TABLE OF CONTENTS ..... 4

PREFACE ..... 6

Acknowledgements ..... 6

LIST OF FIGURES ..... 7

LIST OF TABLES ..... 8

INTRODUCTION ..... 9

BIO-INSPIRED SYSTEMS ..... 10

Darwin's Theory ..... 10

Evolution, genotype and phenotype ..... 11

Amorphous Computing and Cellular Machines ..... 12

CELLULAR AUTOMATA ..... 13

Formal Definition ..... 13

Uniform CA ..... 15

Non-Uniform CA ..... 16

Reduction to 12 rules for non-uniform CA ..... 16

GENETIC ALGORITHMS ..... 18

DISCRETE DYNAMICS AND BASINS OF ATTRACTION ..... 20

ANALYSIS OF THE PROBLEM ..... 21

CODE DEVELOPMENT ..... 22

Engine for uniform CA ..... 22

Engine for non-uniform CA ..... 25

GA implementation ..... 27

GA for uniform CA ..... 27

GA for non-uniform CA ..... 31
Speed improvements ..... 34
EXPERIMENTS, RESULTS AND ANALYSIS ..... 35
Experiments on uniform CAs ..... 36
Experiments on non-uniform CAs ..... 65
CONCLUSION AND FUTURE WORK ..... 85
BIBLIOGRAPHY ..... 86
APPENDIX ..... 89
Appendix 1: 1D uniform CA engine ..... 89
Appendix 2: 1D non-uniform CA engine ..... 92
Appendix 3: GA for uniform CA ..... 96
Appendix 4: GA for non-uniform CA ..... 105

# PREFACE

This report is the result of my master thesis project carried out at the Norwegian University of Science and Technology (NTNU - Trondheim).

The work herein was performed at the Department of Computer and Information Science, NTNU, under the supervision of Associate Professor Gunnar Tufte (http://www.idi.ntnu.no/~gunnart/) and Associate Professor Claudio Gentile (http://www.dicom.uninsubria.it/~cgentile/).

This master thesis is also fulfilling the last portion of my Master of Science degree at University of Insubria (Università degli Studi dell'Insubria – Varese).

## Acknowledgements

First of all I want to express my gratitude to my supervisor Associate Professor Gunnar Tufte who allowed me to develop this work in such a special context. Thanks also for all the support and advice. I am looking forward to further collaboration in the same research field.

I would also like to thank my co-supervisor Associate Professor Claudio Gentile for the precious help in finding an academic experience abroad.

This paper would not have been possible without the backing of my family. My deepest gratitude goes to my parents Ambrogina and Doriano, my grandmother Giovanna and all my relatives.

Without the friendship of my buddies this experience would have been nothing. Thanks, in random order, to Samuela, Alessia, Daniela, Isabella, Manuel, Fabio, Andrea, Matteo, Domenico.

I also have to thank hundreds of new friends I got to know in Norway who contributed to making this last year unforgettable.

Stefano Nichele
June 30, 2009

# LIST OF FIGURES

Figure 1: Darwinian Evolution (Chimps and Humans share 98.77% of the genome) ..... 11
Figure 2: Cellular Automaton definition ..... 14
Figure 3: 1D Uniform CA Rule 30 computation ..... 15
Figure 4: One-point uniform crossover ..... 19
Figure 5: Roulette-wheel selection ..... 19
Figure 6: RBN and basin of attraction ..... 20
Figure 7: Uniform CA engine - structure of rule 126 ..... 24
Figure 8: Input file for non-uniform CA simulation ..... 25
Figure 9: Non-Uniform CA structure example ..... 33
Figure 10: GA output example (n. evolution steps, final state, binary rule) ..... 37
Figure 11: Comparison graph between experiment 1 and 3 ..... 40
Figure 12: Comparison graph between experiment 4 and 5 ..... 43
Figure 13: Comparison graph among experiment 5, 6 and 7 ..... 46
Figure 14: Comparison graph between experiment 7 and 8 ..... 48
Figure 15: Trajectory through intermediate state ..... 50
Figure 16: Comparison graph between experiment 5 and 9 ..... 51
Figure 17: Trajectory comparison - rules 238 and 252 ..... 55
Figure 18: Trajectory comparison - rules 206 and 238 ..... 60
Figure 19: Comparison graph among experiment 10, 11, 12 and 13 ..... 61
Figure 20: Trajectory with two intermediate states in intervals ..... 63
Figure 21: Comparison graph between experiment 13 and 14 ..... 64
Figure 22: Non-Uniform CA computation example ..... 67
Figure 23: Comparison graph between experiment 4 and 15 ..... 68
Figure 24: Fitness function VS GA evolution cycles ..... 70
Figure 25: Comparison graph between experiment 6 and 16 ..... 71
Figure 26: Comparison graph between experiment 5 and 17 ..... 72
Figure 27: Comparison graph between experiment 17 and 18 ..... 74
Figure 28: 1D non-uniform CA, size 17, experiment 20 ..... 77
Figure 29: Execution of experiment 21 ..... 78
Figure 30: Comparison graph between experiment 20 and 23 ..... 82

# LIST OF TABLES

Table 1: 1D Uniform CA Rule 30 representation ..... 15

Table 2: Reduced rule-set for non-uniform CA ..... 17

Table 3: Comparison between Rule 238 and Rule 252 ..... 52

Table 4: Comparison between Rule 206 and Rule 238 ..... 57

Table 5: Non-uniform CA rule-set example ..... 66

Table 6: Experiment 17 - obtained rule-sets ..... 72

Table 7: Experiment 18 - obtained rule-sets ..... 73

Table 8: Experiment 20 - obtained rule-sets ..... 76

Table 9: Experiment 23 - obtained rule-sets ..... 81

Table 10: Experiment 24 - obtained rule-sets ..... 83

# INTRODUCTION

This project is part of ongoing research work at CRAB-lab [1] in the field of artificial evolution and development (EvoDevo). Artificial developmental systems are analyzed and evaluated by viewing the system as a discrete dynamic system and the development process is treated as series of discrete intermediate events, each representing a point in time on the developmental path from zygote to multi-cellular organism.

This is a highly research oriented project and its main long term goal aims toward computation beyond today's machines and technology, exploiting biologically inspired principles: **rethinking the fundamentals of computation**.

Different types (uniform and non-uniform) of cellular automata (CAs) are chosen as a theoretical and experimental platform.

If the parallel nature and limited local communication of a cellular system is considered in relation with the discrete time update of the system, a developmental system, or here a CA, can be approached as a network of sparsely connected units (cells). Such networks can be modelled and analyzed using the same methods as for Boolean networks, Random Boolean Networks (RBN) or extended to multi-value networks. This opens for the possibility to generate and visualize attractor basins and the trajectories from initial configurations to attractors.

The complete state space (all the possible states that the system can reach) from an initial state to a final state is called basin of attraction. A system trajectory, from the initial state to the attractor, may represent the system behavior. As such, it is important that a system's behavior can be described as points on the trajectory and that it is possible to extract information from the running system to know if the system is actually following a specified trajectory.

Specifying a trajectory can be done at different levels of abstraction from the actual state space of the system. For this reason, it is important to find out on what level of abstraction such trajectories can be described, and still be valid for a given sought behavior.

The project work includes the following tasks:

1- Develop an Evolutionary Algorithm that can generate CA rules to produce a specified trajectory;
2- Investigate different time scales, timing paradigms and state abstractions for evaluation.

The main goal is to understand when it is possible to find the correct rules, using different cellular automata types and initial / final conditions, with an evolutionary approach.

# BIO-INSPIRED SYSTEMS

Is it possible to build computers that are intelligent and alive?

This question has been on the minds of computer scientists since the beginning of the computer age and remains a most compelling line of inquiry. Some would argue that the question makes sense only if we put quotes around “intelligent" and “alive," since we're talking about computers and not biological organisms. As some other scientists point out, the answer to that question can be unequivocally yes, no quotes or other punctuation needed, but that to get there our notions of life, intelligence, and computation will have to be deepened considerably [5].

If we are able to include the main biological peculiarities in our computer paradigm, it is possible to couple the words computer and life together. First of all, we have to take inspiration from biology when building our computation machines and then we have to include the concept of evolution in order to make them work.

## Darwin’s Theory

Be it a butterfly, a sunflower or a human being, nature has designed extremely complex machines, the construction of which is still well beyond our current engineering capabilities. Nature is the finest engineer known to man. Her designs come into existence through the slow process, over millions of years, known as evolution by natural selection.

In 1859 Charles Darwin published his masterpiece *On the Origin of Species by Means of Natural Selection, on the Preservation of Favoured Races in the Struggle of Life*. He described, in few words, that evolution is based on “...one general law, leading to the advancement of all organic beings, namely, multiply, vary, let the strongest live and the weakest die.’’ [6].

From the theory of Darwin, the four basic principles of the evolutionary process may be said to be:

1. Viability of individual organisms differs depending on the environments where they live.
2. This variation is heritable.
3. Individuals tend to produce more offspring than can survive on the limited resources available in the environment.
4. In the struggle for survival, the individuals best adapted to the environment are the ones that will have more chances to survive and to reproduce.

The continual workings of this process over the millennia cause populations of organisms to change, generally becoming better adapted to their environments. Most natural organisms consist of numerous elemental units called cells (for example, a human being is composed of approximately sixty trillion cells). Every cell contains the entire plan of the organism, known as the genome: a one-dimensional chain of deoxyribonucleic acid (DNA) that contains the information necessary for the

making of the individual. The genome (or genotype) thus gives rise to the so-called phenotype: the mature organism that emerges from the ontogenetic process. The genotype-phenotype distinction is fundamental in nature: while it is the phenotype that is subjected to the "survival battle", it is the genotype that retains the evolutionary benefits [7] [8].

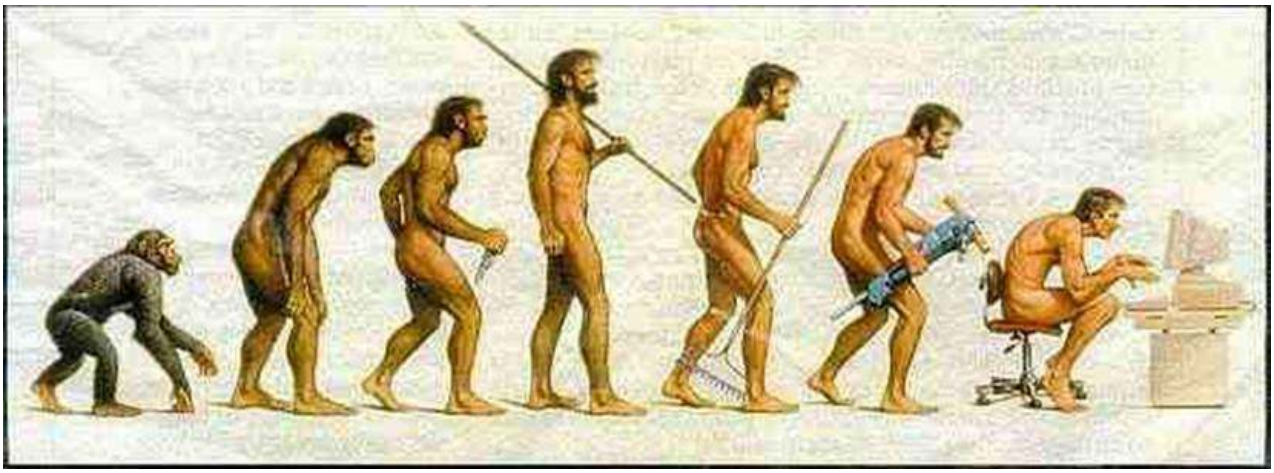


**Figure 1: Darwinian Evolution (Chimps and Humans share 98.77% of the genome)**

## Evolution, genotype and phenotype

An organism's genotype is the set of genes that it carries. An organism's phenotype is all of its observable characteristics, which are influenced both by its genotype and by the environment. So the definition of evolution is really concerned with changes in the genotypes that make up a population from generation to generation. However, since an organism's genotype generally affects its phenotype, the phenotypes that make up the population are also likely to change [9].

In evolutionary computation, the first thing to do is to create a population of organisms (there are no biological details here, just an analogy with the biological process of evolution). As explained before, the organism's genotype is its genetic constitution, the instructions for the making of the individual. The phenotype is the mature organism that emerges through the execution of the instructions written in the genotype. When defining an artificial organism that uses the principles of evolution, it is important to understand how to represent the genome, or how to represent the new species of individuals. After that, the main problem is how to find a genotype that can produce a good individual. Usually the search space is huge and it is infeasible to examine every genotype. It is possible to evolve the population searching for fit elements and then let them reproduce to generate the next generation individuals. This process is called crossover. Even a weak element may possess some important traits in its genome, which may, several generations down the line, combine with other organisms' genetic material to produce a fit individual. This process is iterated for several generations until a good result is achieved. This is how nature works.

## Amorphous Computing and Cellular Machines

The von Neumann architecture, which is based upon the principle of one complex processor that sequentially performs a single complex task at a given moment, has dominated computing technology for the past 50 years. Recently, however, researchers have begun exploring alternative computational systems based on entirely different principles. Although emerging from disparate domains, the work behind these systems shares a common computational philosophy, which is called cellular computing. The main principles of cellular computing are simplicity, vast parallelism and locality [10].

The basic processor used as the fundamental unit of cellular computing, the cell, is simple. Although a current, general-purpose processor can perform quite complicated tasks, a cell by itself can do very little.

Most parallel computers contain no more than a few dozen processors. Cellular computing involves parallelism on a much larger scale. To distinguish this huge number of processors from that involved in classical parallel computing, the term vast parallelism is used.

Cellular computing is also distinguished by its local connectivity pattern between cells. All interactions take place on a purely local basis. A cell can only communicate with a few other cells, most or all of which are physically close by. Further, the connection lines usually carry only a small amount of information. One implication of this principle is that no one cell has a global view of the entire system, there is no central controller.

Amorphous computing is the development of systems based on the cooperation of myriads of unreliable parts that are locally interconnected. An approach of designing and programming amorphous systems is inspired by metaphors from biology and physics. The main advantages of such systems are Scalability and Robustness [11].

Moreover, such systems can be modeled using specific computational machines called Cellular Automata. The metaphor of biological evolution can be exploited on cellular systems because the physical structure is similar to the biological multi-cellular organisms.

In the next chapter, how to use Cellular Automata to model cellular computing machines is explained. Afterwards, it is shown how to adopt natural evolution principles in order to evolve those machines using a specific type of Evolutionary Algorithms called Genetic Algorithms.

# CELLULAR AUTOMATA

Cellular automata (CAs) are idealized versions of the massively parallel, decentralized computing systems described above. A CA is a large network of simple processors, with limited communication among the processors and no central control, which can produce very complex dynamics from simple rules of operation. Cellular Automata were originally conceived by Ulam and von Neumann [12] [13] in the 1940s to provide a formal framework for investigating the behavior of complex, extended systems.

## Formal Definition

Formally, a cellular automaton [14] [15] consists of a countable array of discrete sites or cells *i* and a discrete-time update rule $\Phi$ operating in parallel on local neighborhoods of a given radius *r*. At each time the cells take on values in a finite alphabet *A* of primitive symbols: $\sigma^{i}_{t} \in \{0, 1,..., k- 1\} \equiv A$. The local site-update function is written:

$$\sigma^{i}_{t+1} = \Phi(\sigma^{i-r}_{t}, ...., \sigma^{i+r}_{t})$$

The state $s_t$ of the CA at time *t* is the configuration of the finite or infinite spatial array: $s_t \in A^N$, where $A^N$ is the set of all possible cell value configurations on a lattice of *N* cells. The "extended state space", denoted $A^*$, is the union of all states of any *N*:

$$A^* = \mathrm{U}_{N \geq 0} A^N \qquad \text{with} \qquad A^0 = \emptyset.$$

The CA global update rule $\Phi$: $A^N \rightarrow A^N$ applies $\Phi$ in parallel to all sites in the lattice: $s_t = \Phi\, s_{t-1}$. For finite *N* it is also necessary to specify a boundary condition.

The automata used in this research are "elementary" CAs for which $(k, r) = (2, 1)$; that is, nearest-neighbor interactions and binary cell values, giving a total of $k^{2r+1} = 8$ possible neighborhood patterns. Following the conventional numbering scheme, these patterns are arranged in the order given below. The next cell value is the value obtained by applying $\Phi$ to each pattern. Interpreting the resulting string of 8 symbols as a binary number the rule index is obtained. In the example below the rule with index 18 is shown:

| Neighboorhood | 111 | 110 | 101 | 100 | 011 | 010 | 001 | 000 |
|---|---|---|---|---|---|---|---|---|
| Next cell value | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 |

$\sigma^{i-1}_{t}\ \sigma^{i}_{t}\ \sigma^{i+1}_{t}$

$\sigma^{i}_{t+1}$

In practice, a one-dimensional cellular automaton consists of a one-dimensional lattice of “cells", represented as squares, each of which communicates with the 2 cells that surround it.

These 2 cells along with the cell itself are known as the “neighborhood" of a cell. Each cell starts out at time t=0 as either in the black state (“on" or “1”) or in the white state (“off" or 0).

The whole pattern of states at t=0 is called the “initial configuration".

The cellular automaton iterates over a series of discrete time steps (t = 1; 2; 3; .....; k). At each time step, each cell updates its state as a function of its current state and the current state of the 2 neighbors to which it is connected.

The boundary cells are dealt with by having the whole lattice wrap around into a torus, thus boundary cells are connected to “adjacent" cells on the opposite boundary.

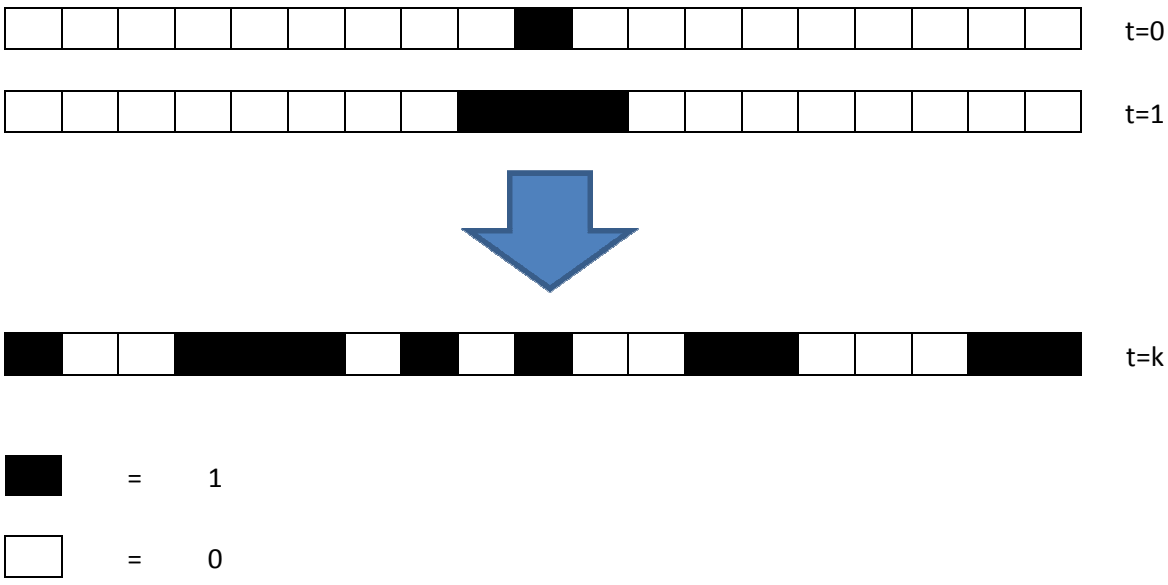



**Figure 2: Cellular Automaton definition**

## Uniform CA

In uniform Cellular Automata each cell uses the same function for updating its state and all cells update synchronously in parallel at each time step. There are two possible values for each cell (0 or 1), and rules that depend only on nearest neighbor values. As a result, the evolution of an elementary cellular automaton can completely be described by a table specifying the state a given cell will have in the next generation based on the value of the cell to its left, the value the cell itself, and the value of the cell to its right.

Since there are 2 x 2 x 2 = $2^3$ = 8 possible binary states for the three cells neighboring a given cell, there are a total of $2^8$ = 256 elementary cellular automata, each of which can be indexed with an 8-bit binary number [16] [17]. For example, the table giving the evolution of rule 30 (in binary 00011110) is illustrated below. In this diagram, the possible values of the three neighboring cells are shown in the top row of each panel, and the resulting value the central cell takes in the next generation is shown below in the center.

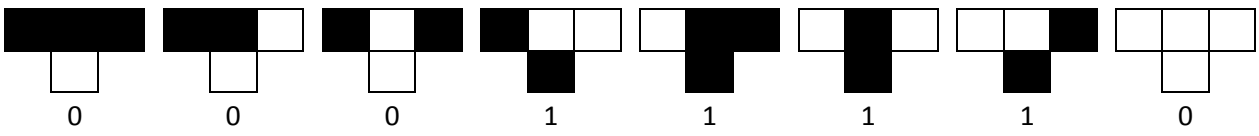



Table 1: 1D Uniform CA Rule 30 representation

The evolution of a one-dimensional cellular automaton can be illustrated by starting with the initial state (generation zero) in the first row, the first generation on the second row, and so on. For example, the figure below illustrates the first 20 generations of the rule 30 uniform cellular automaton, starting with a single black cell in the middle.

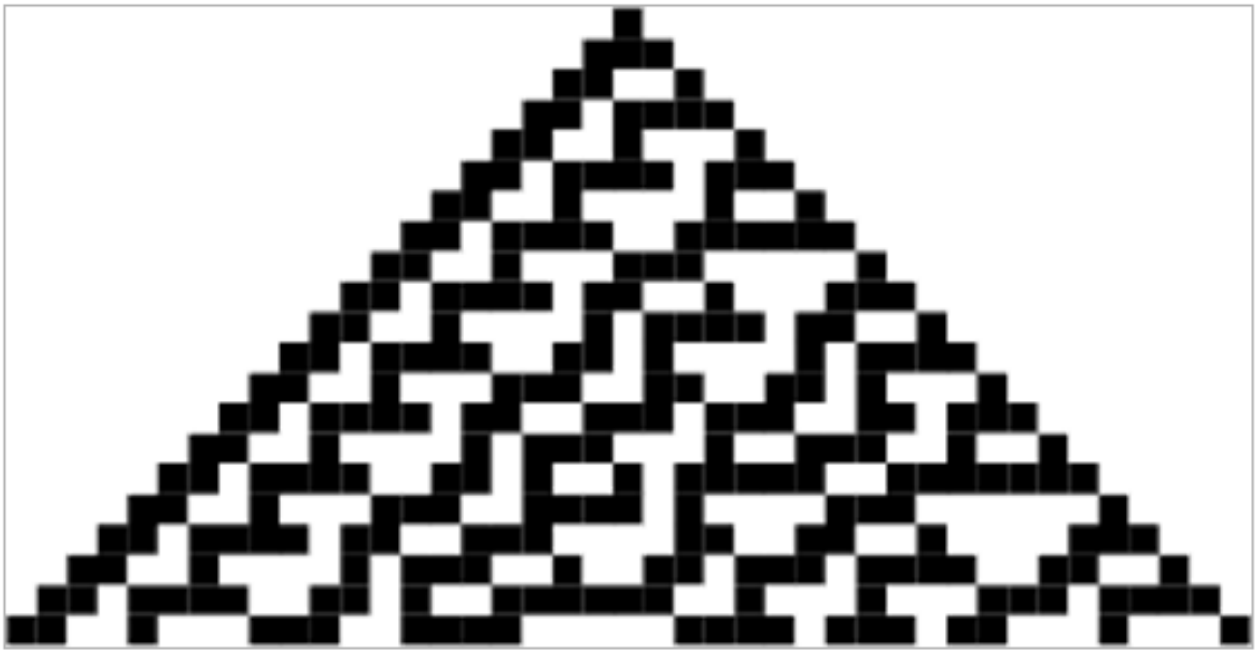


Figure 3: 1D Uniform CA Rule 30 computation

## Non-Uniform CA

In non-uniform Cellular Automata each cell can use a different function for updating its state. All cells update synchronously in parallel at each time step. There are two possible values for each cell (0 or 1), and every rule depends only on nearest neighbor values and the cell type (rule to apply in the specific cell).

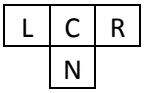
Since there are 2 x 2 x 2 = $2^3$ = 8 possible binary states for the three cells neighboring a given cell, there are a total of $2^8$ = 256 possible rules that can be applied to each cell. This means that the state space of all the possible rule-sets for an automaton with *N* cells has size $256^N$.

As the search space is huge, it can be unfeasible to find a rule that computes a specific function. In the next section, how to reduce the rule-set is explained.

## Reduction to 12 rules for non-uniform CA

As shown in different works of Sipper [8] and Bidlo [18], it is possible to reduce the number of rules in the rule-set without losing the expressiveness of the CA [19] [20]. Instead of using all the 256 possible rules, only 12 rules are used (Propagation, XOR, OR, NAND). The state-space, for a CA with N cells, becomes $12^N$ (it is a permutation with repetitions). If the CA, for example, has size 65, the state-space is $12^{65}$ ($\sim 1.4 \times 10^{70}$ possible rule-sets).

Every cell and neighborhood can be represented as:




with L = left neighbor, C = current cell, R = right neighbor and N = next value.

The reduced set of rules is represented in the following table:

| **RULE CODE / INDEX** | **OPERATION PERFORMED** |
|---|---|
| 0 | Identity of value C |
| 1 | Identity of value L |
| 2 | Identity of value R |

| | |
|---|---|
| 3 | OR between L and C |
| 4 | OR between C and R |
| 5 | OR between L and R |
| 6 | XOR between L and C |
| 7 | XOR between C and R |
| 8 | XOR between L and R |
| 9 | NAND between L and C |
| 10 | NAND between C and R |
| 11 | NAND between L and R |

**Table 2: Reduced rule-set for non-uniform CA**

# GENETIC ALGORITHMS

A Genetic Algorithm (GA) is an idealized computational version of Darwinian evolution. In Darwinian evolution, organisms reproduce at differential rates, with fitter organisms producing more offspring than less fit ones. Offspring inherit traits from their parents; those traits are inherited with variation via random mutation, sexual recombination, and other sources of variation. Thus traits that lead to higher reproductive rates get preferentially spread in the population, and new traits can arise via variation [5] [21] [22].

In GAs, computer "organisms" encoded as bit strings of ones and zeros reproduce in proportion to their fitness in the environment, where fitness is a measure of how well an organism solves a given problem. Offspring inherit traits from their parents with variation coming from random mutation, in which parts of an organism are changed at random, and sexual reproduction, in which an organism is made up of recombined parts coming from its parents.

Assume the individuals in the population are Cellular Automata rules encoded as bit strings. The following is a simple genetic algorithm.

1. Generate a random initial population of M individuals.

Repeat the following for N generations:

2. Calculate the fitness of each individual in the population. The user must define a function assigning a numerical fitness to each individual.

3. Repeat until the new population has M individuals:

    a) Choose two parent individuals from the current population probabilistically as a function of fitness.

    b) Cross them over at a randomly chosen point to produce an offspring. That is, choose a position in each bit string, form one offspring by taking the bits before that position from one parent and after that position from the other parent.

    c) Mutate each value in the offspring with a small probability.

    d) Put the offspring in the new population.

4. Go to step 2 with the new population.

This process is iterated for many generations, at which point hopefully one or more high-fitness individuals have been created. Notice that reproduction in the simple GA consists merely of copying parts of bit strings.

The GA can be tuned using different parameters and techniques. In this research the following strategies are used:

- One-point uniform crossover: a single crossover point on both parents' organism strings is selected. All data beyond that point in either organism string is swapped between the two parent organisms. The resulting organisms are the children.

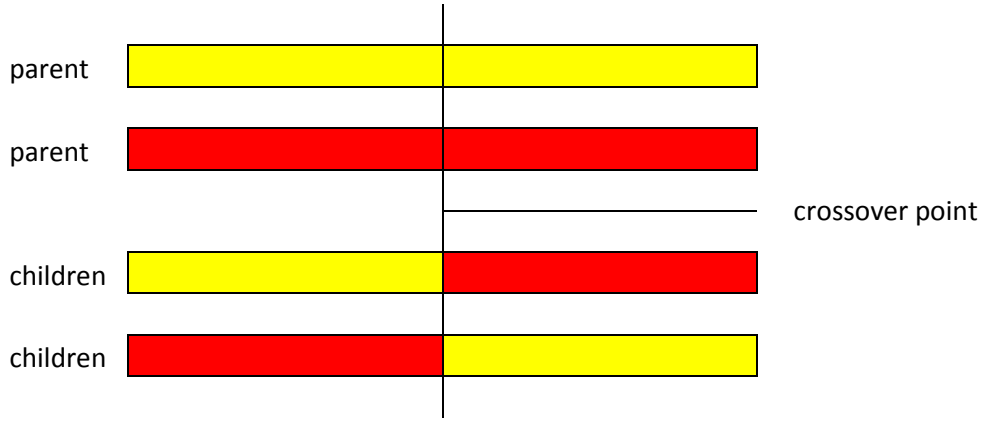



Figure 4: One-point uniform crossover

- Crossover rate: it is simply the chance that two chromosomes will swap their bits or in other words the probability of performing the crossover. A good value is 0,7 [23].

- Mutation rate: it is the chance that a bit within a chromosome will be flipped (0 becomes 1, 1 becomes 0). A good value is 1/L where L is the length of the bit string [24] [25] [26] [27] [28].

- Roulette-wheel selection: it is a way of choosing members from the population of chromosomes in a way that is proportional to their fitness. It does not guarantee that the fittest member goes through to the next generation but merely that it has a very good chance of doing so. As specified in the following image, the sum of all the fitness functions F is calculated and then a random number r is chosen in the interval [0,F). The element with the fitness in the corresponding interval is chosen. Elements with higher fitness have higher probability of being chosen [29] [30].

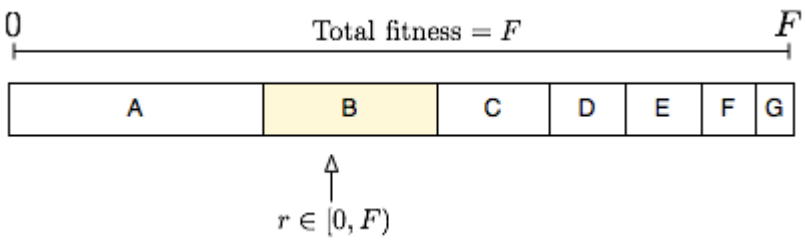



Figure 5: Roulette-wheel selection

# DISCRETE DYNAMICS AND BASINS OF ATTRACTION

Networks of sparsely inter-connected elements with discrete values and updating in parallel are central to a wide range of natural and artificial phenomena drawn from many areas of science, from physics to biology to cognition, to social and economic organization, to parallel computation and artificial life, to complex systems in general [31].

A Boolean network consists of a set of Boolean variables whose state is determined by other variables in the network. They are a particular case of discrete dynamical networks, where time and states are discrete. Boolean and elementary cellular automata are particular cases of Boolean networks, where the state of a variable is determined by its spatial neighbors. The first Boolean networks were proposed by Stuart A. Kauffman in 1969, as random models of genetic regulatory networks [32]

A Random Boolean network (RBN) is a system of N binary-state nodes (representing genes) with K inputs to each node representing regulatory mechanisms. The two states (1/0) represent respectively, the status of a gene being active or inactive. The state of a network at any point in time is given by the current states of all N genes. Thus the state space of any such network is $2^N$. Simulation of RBNs is done in discrete time steps. The state of a node at time t+1 is a function of the state of its input nodes and the boolean function associated with it [33].

In a Cellular Automaton of size N, one of its states B might be 1010….0110. The state-space is made up of all $2^N$ states, the space of possible bitstrings or patterns. A trajectory in state-space, can be described graphically by a Random Boolean Network where the states are represented by nodes connected by arcs and arrows. The state C, in Figure 6, is a successor of B, and A is a predecessor (pre-image) of B, according to the dynamics on the network. Any trajectory must sooner or later encounter a state that occurred previously and thus, since the dynamics are deterministic, enter an attractor cycle or a point attractor. The complete graph is called the basin of attraction.

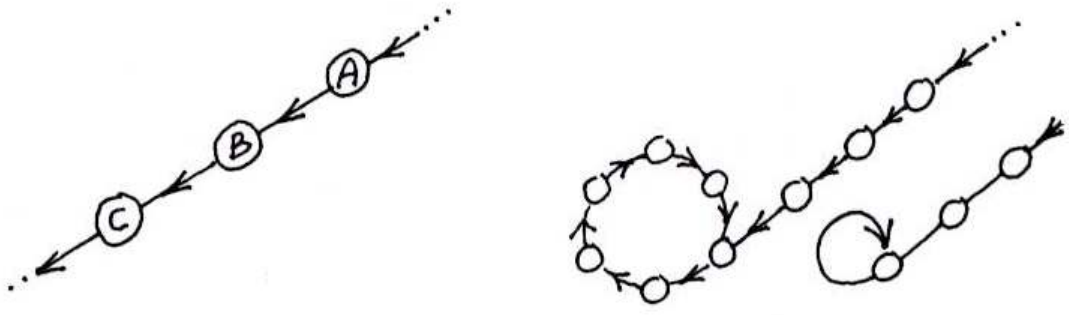



**Figure 6: RBN and basin of attraction**

# ANALYSIS OF THE PROBLEM

Cellular Automata have been studied a lot in the last 40 years [34] [35] [13] [15] and nowadays there are several researches going on. They are used as computational devices, as pseudo-random number generators [36] and for many other purposes. Their complexity and computational power have been analyzed [37], for instance, to solve the majority problem [10] or to compute other complex operations, as explained in [38], [39], [40] and [41].

Evolutionary Algorithms can be combined with Cellular Automata [5]. Such a combination provides a system with a natural way of handling complexity that fits perfectly over a structure which is inspired to cellular organisms [10].

A Cellular Automaton can be considered as an artificial developmental system [4] and therefore it can be analyzed as a discrete dynamic system [42]. More precisely, in the developmental process [3], the sequence of states of the CA can be treated as series of discrete intermediate events representing a point in time on the developmental path from zygote to multi-cellular organism. At another level of abstraction, a developmental system (here a CA) can be approached as a network of sparsely connected units (here cells) and consequently analyzed as a Random Boolean Network [33]. In this way, it is also possible to graphically visualize attractor basins and trajectories. A system trajectory, from an initial state to the attractor, can represent the system behavior.

It is important to specify a trajectory with some intermediate points along it and find rules that can meet the behavior specifications. The final goal of the project is to find (when possible) and study the behavior of Cellular Automata rules that can generate a specified trajectory that crosses some given intermediate states and reach a defined attractor basin, using a Genetic Algorithm approach. In the beginning simple 1-Dimensional Cellular Automata are used in order to gain experience. After, Non-Uniform Cellular Automata are used to investigate more complex behaviors [43].

The first part of the project consists in the development of uniform and non-uniform CA simulators that can generate a computation, given the initial state of the CA and the rule / rule-set.

The second step consists in the development of a Genetic Algorithm that can search for rules / rule-sets able to generate a specified computation on the previously developed CAs. The trajectory is described with an initial state, some intermediate states that are crossed during the process and a final attractor. The intermediate states can be found at specified points in time or into time intervals. Also the length of the computation is given as input parameter. This phase is performed gradually. Initially the trajectory is described by an initial and final state. Then an intermediate state is introduced in a defined point in time. After, time intervals are introduced and in the end a trajectory with two intermediate states is investigated. This is done for both uniform and non-uniform CAs. At the end, the CA simulators are used to check the correctness of the rules found by the GA.

In the performed experiments different combinations of the following parameters and characteristics are investigated: size (number of cells) and CA type (uniform vs. non-uniform), number of evolution steps, initial-intermediate-final states (given or randomly generated), timing intervals, investigation of symmetric rules and symmetry breaking, trajectories with equal or different attractors (different trajectories with same attractor, same trajectories with different attractors). The research is motivated by the possibility to find the CA rules to describe a trajectory using a Genetic Algorithm.

# CODE DEVELOPMENT

In this chapter the code development phase is described. First, the CA simulators for uniform and non-uniform CA are explained. Then, the coding of the Genetic Algorithm is shown for both uniform and non-uniform CAs. In the end the adopted speed improvements are presented.

All the code has been written in C language using lcc-win32 editor and compiler [44], under the Operating System Windows Vista 32 bit.

The machine where the code has been executed is an Intel Core 2 Duo T8100 @ 2.10 GHz with 4 GB of RAM.

## Engine for uniform CA

The engine for a 1 dimensional uniform cellular automaton (full code in the Appendix 1) is used to simulate the computation of the CA, given an initial state and a specified rule. The automaton is evolved for a certain number of cycles and the output is the last state. An example is given in Fig. 7.

In every instant of the computation the required information are the following:

- the size of the Cellular Automaton

```
const int size = 65;
```

- the current state of the CA, an array which is representing the state of every single cell

```
int automata[size];
```

- the next state of the CA

```
int next[size];
```

- the rule used for the computation

```
struct rule
{
        int left;
        int me;
        int right;
        int next;
};

struct rule rules[8];
```

- the number of the required evolution cycles

```
int loops = 64;
```

As defined in the main, the three main parts of the engine are:

- initialization of the rule

```
void init_rules()
```

- initialization of the CA

```
void init_ca()
```

- computation of the automata

```
void run()
```

In the rule's initialization, for every possible configuration of the neighbors and the current cell, an output is given. This is done repeating the following portion of code for each of the eight possible three-bit strings:

```
rules[5].left   = 1;
rules[5].me     = 0;
rules[5].right  = 1;
rules[5].next   = 0;
```

In this example, if a local situation of 101 is found (left and right neighbors' value is 1 and the analyzed cell's value is 0) the next value of the cell will be 0.

In the CA initialization, a value is given to all the cells of the automaton. The standard configuration is to assign a value equal to 0 to every cell except the one in the middle.

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | …… | 0 | 0 | 0 | 1 | 0 | 0 | 0 | …… | 0 | 0 | 0 |

In this way, it is possible to observe the evolution and propagation of the rule behavior in both the sides of the CA.

The core of the engine is the run() function, which performs the computation of the CA. Basically, for each of the evolution cycles, the Left–Centre–Right combination is searched in the rule-set and the Next value is updated accordingly in the new state of the automaton. There are two particular cases: when the analyzed cell is the first, the left neighbor is the last cell of the CA; the other way around if the analyzed cell is the last one, the right neighbor is the first cell of the CA.

This is done in the code by the function find_rule(), for the cell in the specified position "pos":

```
int find_rule (int pos)
```

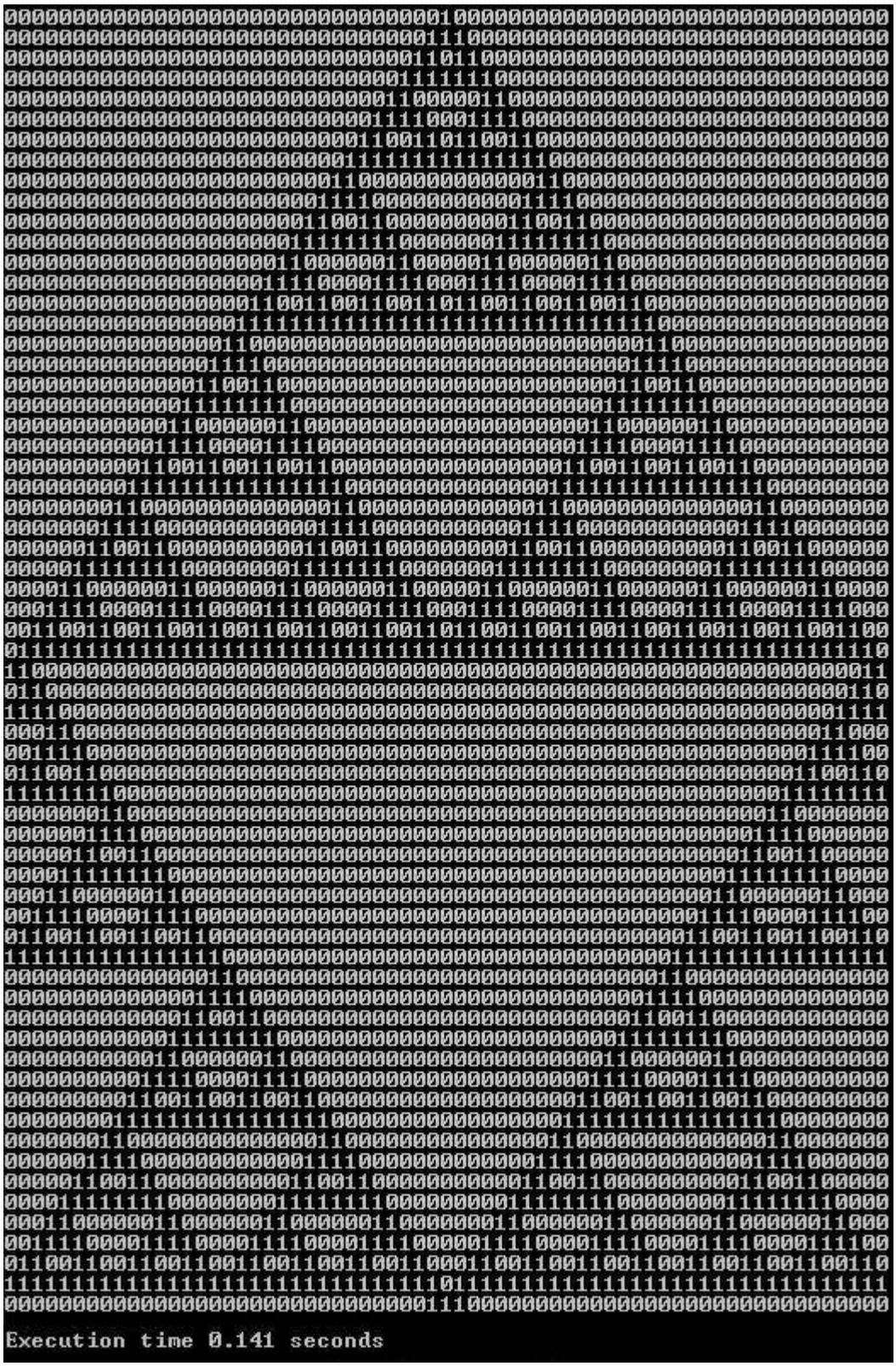



Figure 7: Uniform CA engine - structure of rule 126

## Engine for non-uniform CA

The engine for a 1 dimensional non-uniform cellular automaton (full code in the Appendix 2) is used to simulate the computation of the CA, given an initial state and a specified rule-set (one rule for each cell). The automaton is evolved for a certain number of cycles and the output is the last state.

The same objects as for the uniform CA are instantiated, except for the rule structure. In fact, only an array with the cell types is maintained:

```
int celltype[size];
```

The cell type can be one of the following:

*0* *Identity C*
*1* *Identity L*
*2* *Identity R*
*3* *OR L,C*
*4* *OR C,R*
*5* *OR L,R*
*6* *XOR L,C*
*7* *XOR C,R*
*8* *XOR L,R*
*9* *NAND L,C*
*10* *NAND C,R*
*11* *NAND L,R*

Moreover, an input file is used to read the specific rule-set of the automata.

```
FILE *input_file;
```

In the following image there is an example of input file with a ruleset for a non-uniform CA of size 17:

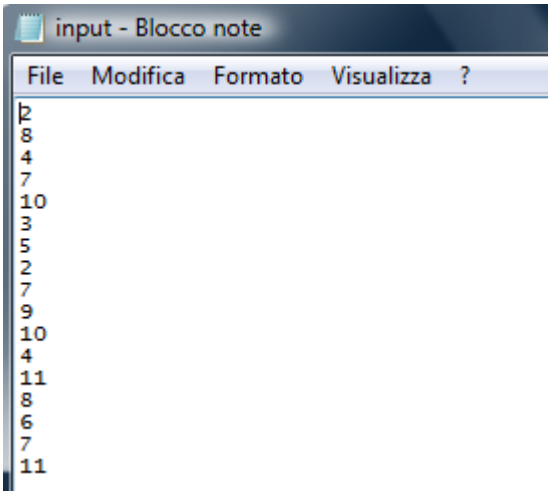



**Figure 8: Input file for non-uniform CA simulation**

The file is opened in “read” mode.

```
input_file = fopen("input.txt", "r");
```

Inside the file “input.txt” a sequential list of the cell types is present. They are processed sequentially and each value is assigned to the relative element in the array:

```
fscanf(input_file, "%d", &celltype[i]);
```

The automaton is initialized as for the uniform CA giving a value equal to 0 to every cell except for the one in the middle.

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | …… | 0 | 0 | 0 | 1 | 0 | 0 | 0 | …… | 0 | 0 | 0 |

In this way, it is possible to observe the evolution and propagation of the rule behavior on both sides of the CA.

The main part of the engine is the run() function, which performs the computation of the non-uniform CA. As for a uniform CA, for each of the evolution cycles each cell is processed but, depending on the cell type, the correct rule is applied and the next value is calculated. When the analyzed cell is the first, the left neighbor is the last cell of the CA; when the analyzed cell is the last one, the right neighbor is the first cell of the CA.

For instance, if the cell is of type 1, the Identity Left rule is applied:

```
switch(celltype[centre])
{
……..
        case 1 :
                next[centre] = automata[left];
                break;
……..
```

## GA implementation

The Genetic Algorithm is implemented to search for specific rules that can compute (on a particular CA) a given trajectory, described with an initial state, a final state and some intermediate states that can be found inside time intervals. As the structure of uniform and non-uniform CA is different, two separate GA are used.

### GA for uniform CA

The full implementation of the Genetic Algorithm for uniform CA is given in the Appendix 3. The most important data structures are the same used in the uniform CA engine:

- size of the CA
- number of evolution steps
- array to keep the current state of the CA
- array to calculate the next state of the CA
- rule structure

Moreover, the following items are declared:

- size of the population (by default 10 elements are used)

```
const int child = 10;
```

In order to keep track of the state of every element in the population, the declaration of the automata states become:

```
int automata[child][size];
```

Also one rule for each element in the population is declared:

```
struct rulestruct ruleset[child];
```

- input values, specifically the initial state of the CA, the final state and the two intermediate states:

```
int input[size];
int output[size];
int inter1[size];
int inter2[size];
```

- intervals for the intermediate states; the first range is defined by position1 and position2 and the second range is between position3 and position4. Those values are defined as constant, before the beginning of the computation:

```
const int position1 = 10;
```

```
const int position2 = 20;
const int position3 = 40;
const int position4 = 50;
```

- input and output files. In the input file the initial, intermediate and final states are defined. In the output file the final rule and the number of required iterations (performed crossover) are written:

```
FILE *input_file, *output_file;
```

- array of the fitness value for every element of the population:

```
int fit[child];
```

- array of the weights of the fitness values for every element of the population:

```
float weight[child];
```

- ranking of the rules according to their fitness value multiplied for the weight:

```
int rank[child] = {0,1,2,3,4,5,6,7,8,9};
```

- indexes for the two best and two worst rules in the ranking:

```
int best1, best2;
int worst1, worst2;
```

The algorithm works as follow: in the beginning the configuration is read using the procedure:

```
void read_conf()
```

Basically, every string in the input file representing initial, final and the two intermediate states is read and saved in the relative data structure. The procedure reads every character in the file until a new line is found. Every char representing the state of a single cell is converted from ASCII code to integer and saved in the array.

For example, the initial state of the CA is retrieved using the code below:

```
while((c = fgetc(input_file)) != '\n')
  {
        input[i] = c - 48;                    //ASCII code of value 0 is 48
        i++;
  }
```

Once the configuration parameters are set up, the population of ten rules is randomly initialized using the procedure:

```
void init_rules()
```

For every element in the population, all the rules are initialized using the code below. For example, for the sub-rule 000 (cell with value 0 and left / right neighbor with value 0) the next state can have value 0 or 1 with probability 0,5 each.

```
ruleset[i].rules[0].left        = 0;
ruleset[i].rules[0].me          = 0;
ruleset[i].rules[0].right       = 0;
j = rand() % 100 + 1;
if(j>50)
        {ruleset[i].rules[0].next =0;}
else
        {ruleset[i].rules[0].next =1;}
```

Before the core of the GA (composed by fitness evaluation, crossover and mutation), another step is required. All the CAs in the population are configured with the initial state, as read before from the input file. At this point, everything is ready for the real computation. As the goal is to perform every simulation several times, the main steps are executed for a specified number of runs. For every simulation, before the new trial, the initial parameters are restored with the above described default values.

The procedure run() evolves every CA in the population for the desired number of steps, checking if the intermediate states are reached during the computation. If they are not found, the weight of the relative fitness function is reduced by 0,3 for every missed checkpoint. The functioning of the procedure is the same described before for the engine of the uniform CA.

With the function fitness(), the fitness value is calculated for every element in the population. The reached final state is compared with the desired final state and the Hamming distance is calculated. The obtained value is weighted using the following formula, where *count* represents the number of matching cells between desired and final state:

```
fit[i]=count * weight[i];
```

The rules are ranked accordingly to their fitness and the two worst rules are substituted by two new elements. The new generation elements are obtained using a procedure of selection, crossover and mutation, as it happens in nature.

The procedure best_rules() performs a selection of the two rules that are used for the crossover.

This is done using a Roulette-Wheel technique [23]. This is a way of choosing members from the population of rules in a way that is proportional to their fitness. It does not guarantee that the fittest member goes through to the next generation; merely that it has a very good chance of doing so. It works like this: imagine that the population's total fitness score is represented by a pie chart, or roulette wheel. Now assign a slice of the wheel to each member of the population. The size of the slice is proportional to that rules fitness score (i.e. the fitter a member is the bigger the slice of pie it gets). Now, the element is chosen spinning the ball and grabbing the rule at the point it stops.

This is performed by the following code. The sum of fitness is calculated. After that, two random values (i and k) are generated and a check is performed in order to verify in which interval the two random values fall:

```
void best_rules()
{
	int sum = 0;
	int i,k,j;

	for(i=0; i<child; i++)
		sum = sum + fit[i];			// sum of the fitness functions

	i = rand() % sum + 1;			//first random value
	k = rand() % sum + 1;			// second random value

	sum = 0;
	j = 0;

	while(j < child)			// for every element in the population
	{
		if((i>sum) && (i<sum+fit[j]+1))	// if the random number is included
		{				// in the current interval, the rule is chosen
			best1 = j;
		}

		if((k>sum) && (k<sum+fit[j]+1))
		{
			best2 = j;
		}

		sum = sum + fit[j];		// calculate the next interval
		j++;
	}
}
```

The function replace() performs a uniform crossover [23], with a probability of 0,7. The crossover is done dividing exactly the chromosomes (bit strings representing the rules) in the middle and swapping all the genes (bits) after that point.

In the following example, the worst ranked rule is replaced by the first four bits of the first best rule (chosen with roulette-wheel) and the last four bits of the second best rule (chosen with roulette-wheel):

```
ruleset[worst1].rules[0].next = ruleset[best1].rules[0].next;
ruleset[worst1].rules[1].next = ruleset[best1].rules[1].next;
ruleset[worst1].rules[2].next = ruleset[best1].rules[2].next;
ruleset[worst1].rules[3].next = ruleset[best1].rules[3].next;

ruleset[worst1].rules[4].next = ruleset[best2].rules[4].next;
ruleset[worst1].rules[5].next = ruleset[best2].rules[5].next;
ruleset[worst1].rules[6].next = ruleset[best2].rules[6].next;
ruleset[worst1].rules[7].next = ruleset[best2].rules[7].next;
```

The last but very important step is the random mutation. With a probability of 0,125 (1/L where L is the length of the bit-string = 8) each single bit of the new generated rules can be flipped (0 becomes 1, 1 becomes 0).

At the end, the fitness of the new generated elements is recalculated and the procedure is restarted, until an element with maximum fitness function is found. This means that the rule found when the computation ends is able to reach the desired final state, starting from the desired initial state, overstepping the two desired intermediate steps.

**GA for non-uniform CA**

The full implementation of the Genetic Algorithm for non-uniform CA is given in the Appendix 4. The main structure is the same shown in the previous chapter for the uniform CAs. In this chapter all the relevant differences are presented.

First of all, one specific rule is present for each cell of the CA. Moreover, to keep the information about the rules, it is sufficient to save the rule code, which represents also the cell type.

As mentioned before, the cell type can be one of the following:

*0 Identity C*
*1 Identity L*
*2 Identity R*
*3 OR L,C*
*4 OR C,R*
*5 OR L,R*
*6 XOR L,C*
*7 XOR C,R*
*8 XOR L,R*
*9 NAND L,C*
*10 NAND C,R*
*11 NAND L,R*

The rule-sets for each of the elements in the population is saved in following matrix, where "child" represents the number of elements in the population and "size" the length of the CA :

```
int ruleset[child][size];
```

Once the input parameters, such as the initial state, final state and intermediate states, are read, the rules are initialized randomly in the procedure init_rules().

With the procedure run() all the CA in the population are evolved for the number of desired steps, checking the presence of the specified intermediate states along the trajectory. The main features of this procedure are the same explained in the chapter about the Engine for non-uniform CA. For every

missed checkpoint the weight for the relative fitness function is reduced according to the following formula:

```
f1 = 0.3 - (0.3 * ((count*1.0)/size));
```

The computation of the next value of the cell for the following state is done applying the relative rule. In the example below, the rule number 3 (OR between the left neighbor and the value of the cell itself) is shown:

```
switch(ruleset[i][centre])
{
        ……..        // other cases
        case 3 :
                if((automata[i][left] == 0) && (automata[i][centre] == 0))
                        next[centre] = 0;
                if((automata[i][left] == 0) && (automata[i][centre] == 1))
                        next[centre] = 1;
                if((automata[i][left] == 1) && (automata[i][centre] == 0))
                        next[centre] = 1;
                if((automata[i][left] == 1) && (automata[i][centre] == 1))
                        next[centre] = 1;
                break;
```

The variable f1 represents the coefficient that is subtracted from the weight of the fitness function for the current rule. The maximum value of the weight is 1 and if the intermediate states are present and 100% correct the value of the coefficient is 0. Otherwise, if there is no matching or partial matching between the intermediate state and the final state, the subtracted coefficient has a value between 0 and 0,3 for each of the nonequivalent intermediate states. The variable "count" represents the number of correct bits, comparing the desired and the actual state. In this way, the subtracted values are proportional to the number of non-matching bits for every intermediate state.

The final fitness, for every couple rule / CA, is calculated multiplying the number of matching bits of the final state with the relative weight (this is done inside the procedure fitness()):

```
fit[i]=count * weight[i];
```

The rule-sets are ranked according to the value of their fitness function. The Genetic Algorithm for non-uniform CA is different from the GA used for uniform CA. In fact, no crossover is performed. Only one rule-set is selected using a Roulette-Wheel technique. The next generation elements are created only mutating the chosen rule-set. All the other rule-sets, except the chosen one, are replaced. In this way, in every step, the population is completely new. The mutation itself consists on a substitution of the rule for a particular cell with a new randomly chosen rule. This is done with a probability of 0,085 (1/L where L is the number of possible rules = 12) and the mutation of the rule for each cell is done independently one after another.

All the described steps are repeated until a rule-set with a fitness value equal to the size of the CA is found. In the output file the number of the GA iterations (performed crossover) and the obtained rule-set are written.

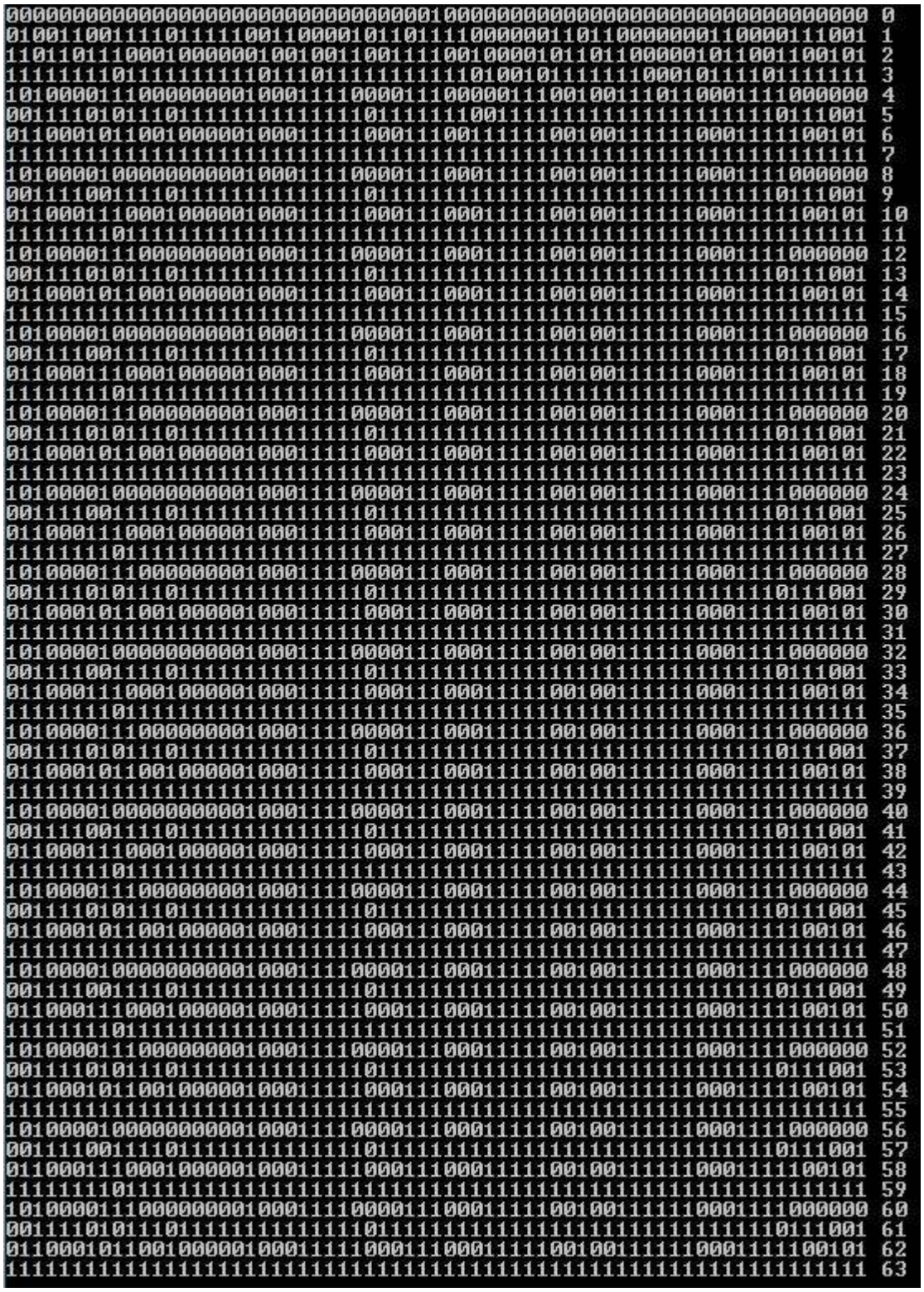


Figure 9: Non-Uniform CA structure example

In Figure 9 an example of the evolution of structure for a non-uniform CA is shown. The computation from the initial state 000000…1…000000 (all 0 and 1 in the middle) to the state 111111…111111 (all 1) is obtained with the rule-set: 2 11 4 8 11 9 6 5 9 11 10 9 7 10 9 9 11 9 4 8 10 9 3 2 2 5 11 8 10 11 4 2 3 6 9 8 5 2 5 4 5 10 11 3 10 11 3 5 3 5 5 4 8 10 11 3 3 1 2 11 10 9 6 6 9.

## Speed improvements

Genetic Algorithms are a sort of search algorithms. As the search space can be huge, such as for the possible rule-sets of a non-uniform CA, it is important to speed-up the computation.

The first improvement that has been introduced is the application of Mergesort algorithm in order to create the ranking of the elements in the population, according to value of their fitness function. Mergesort is a comparison-based sorting algorithm invented by John Von Neumann. It is based on the divide and conquer paradigm. The performances in the worst and average case are O(n log n), where n represents the number of objects in the population.

Another improvement comes out of the tuning of the Genetic Algorithm parameters such as the crossover rate and the mutation rate.

The crossover rate is the probability of the crossover to be performed. As specified in [23], the search space is explored in a optimized way using a crossover rate equal to 0,7.

The mutation rate is the probability that a chromosome (in this case the representation of the rule) is mutated. In [24] it is stated that a good value for the mutation rate is 1/L, where L represents the size of the population which is subject to mutation (the length of the bit-string needed to represent a rule for uniform CA, the number of rules for non-uniform CA).

An additional improvement used to increase the performances of the Genetic Algorithm is the Roulette-Wheel technique, also known as Fitness Proportionate Selection. It is used to choose potential elements in the population that are used in the recombination phase.

In Roulette-Wheel, as in all selection methods, the fitness function assigns a fitness value to possible solutions. This fitness level is used to associate a probability of selection with each individual element. If *fi* is the fitness of individual *i* in the population, its probability of being selected is:

$$p_i = \frac{f_i}{\sum_{j=1}^{N} f_j}$$

where *N* is the number of individuals in the population. While elements with a higher fitness will be less likely to be eliminated, there is still a chance that they may be. This helps the exploration of the search space.

After the introduction of all the described speed up improvements, the computation time is approximately 5 times shorter.

# EXPERIMENTS, RESULTS AND ANALYSIS

In this section all the performed experiments are presented, staring with the tests executed on uniform cellular automata and after with the non-uniform cellular automata. They are listed in the same order as they have been executed, in order to give an idea of the increase of difficulties, in terms of development, execution and needed computational power. Moreover, in every experiment the introduced improvements are explained.

All the rules and their relative behaviors are indexed using Wolfram's rule classification [17] [16] [45].

The experimental setup and the results are presented using the following structure:

**CA characteristics:**

| | |
|---|---|
| Automaton type | uniform or non-uniform |
| Number of cells | size of the CA |
| Number of evolution steps | number of cycles from the initial state to the final state |
| Fixed initial state | initial configuration of the CA |
| Desired final state | final configuration of the CA |

**Results after the simulation:**

| | |
|---|---|
| Mean number of crossovers | average number of crossover needed to find the solution |
| Variance | measure of statistical dispersion |
| Standard deviation | square of the variance, another measure of statistical dispersion |
| Minimum number of crossovers | minimum value |
| Maximum number of crossovers | maximum value |

The term evolution is mostly used to describe the evolution steps of the Cellular Automata. Sometimes it is also referred to the iterations of the Genetic Algorithm, to indicate the evolution of the population from the parents to the offspring.

The most important index is the number of required crossovers. It is a relevant measure of the effectiveness and the speed of the GA. If the standard deviation is large, it means that the results of the experiment are volatile and they tend to vary quite often.

## Experiments on uniform CAs

## Experiment 1

In this first simple experiment a 1D uniform CA is used in order to find the rules that can generate a computation from a specified initial state to a desired final state. The goal is to check the correct behavior of the Genetic Algorithm. It is known that different rules can evolve from the specified initial configuration to the required output. The objective is to generate the correct rules and consequently simulate and check their correctness.

The Genetic Algorithm used during this first phase of the project has only basic features and the performed operations are:

- Generate initial population of 10 random rules;
- Evolve the automata for each of the rules;
- Calculate the best and worst two rules according to the fitness function (hamming distance – number of matching bits between the reached final state and the desired final state);
- Perform uniform crossover between the two best ranked rules;
- Replace the two worst ranked rules with the new generated rules;
- Perform a random mutation in the new generated rules with probability 0,5;
- Repeat all the steps (except the first) until a fitness of 100% is reached.

CA characteristics:

| | |
|---|---|
| Automaton type | 1 dimension uniform CA |
| Number of cells | 65 |
| Number of evolution steps | 64 |
| Fixed initial state | 00000000000000000000000000000001000000000000000000000000000000000 |
| Desired final state | 11111111111111111111111111111110111111111111111111111111111111111 |

Results after 1.000 simulations:

| | |
|---|---|
| Mean number of crossovers | 259,491 |
| Variance | 5.952.946,372 |
| Standard deviation | 2.439,866 |
| Minimum number of crossovers | 0,000 |
| Maximum number of crossovers | 71.009,000 |

Obtained rules:

- 203 11001011
- 217 11011001
- 218 11011010
- 219 11011011

In Figure 10 an example of the output of the GA is shown. In particular, each line indicates the number of performed crossover, the final state reached by the CA and the binary value of the rule found by the GA. The values are separated by commas.

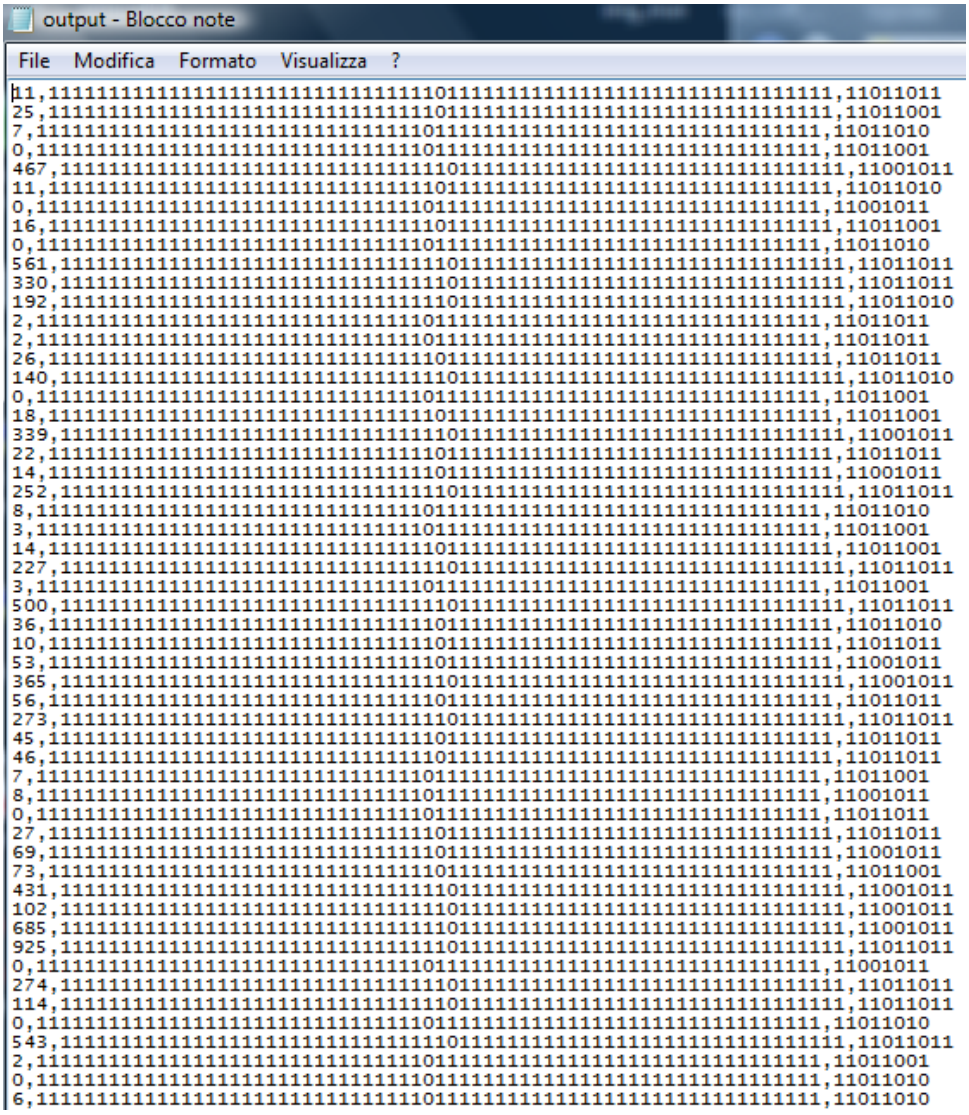



**Figure 10: GA output example (n. evolution steps, final state, binary rule)**

**Analysis:**

The minimum number of evolution steps required to generate the desired rules is 0 and the maximum is 71.009. As the standard deviation is 2.439,372, this means that the majority of the computations generate a number of evolution steps between 0 and 2.700, with a high density around the mean value of 259,491. All the generated rules (203, 217, 218 and 219) reach the desired final state after 64 evolution steps. There are some other rules, for instance rule 90, which can reach the same final state after the same number of computation steps, but they are not found by the algorithm. Therefore, the first version of the Genetic Algorithm works properly but needs to be refined in order to find also the other correct rules. In order to better understand the weaknesses of the GA, another similar experiment is executed (Experiment 2).

## Experiment 2

In this experiment the same initial conditions as the Experiment 1 are used. The desired final state is different and, starting from the same fixed initial state, it can be reached only by the rule number 126. The goal of this test is to find the correct rule to perform the specified computation, using the GA described previously.

CA characteristics:

| | |
|---|---|
| Automaton type | 1 dimension uniform CA |
| Number of cells | 65 |
| Number of evolution steps | 64 |
| Fixed initial state | 00000000000000000000000000000001000000000000000000000000000000000 |
| Desired final state | 00000000000000000000000000000011100000000000000000000000000000000 |

Results:

When the computation ends the rule number 126 is found. However, the experiment failed in some runs. The failed runs resulted in an evolved solution that reached a local maximum, maintaining a fitness value of 64/65 (around 98,45%).

Some modifications to the GA are required in order to increase the performances and the success rate.

**Analysis:**

As the results show, this experiment has partially failed because during some computations the Genetic Algorithm falls into a local solution and loops forever searching for a rule that will never

reach, maintaining a fitness value of 64/65 (around 98,45%). Some other times the computation ends and the rule number 126 is found. After an analysis of the level of fitness during the mutation steps it is possible to observe that, after the crossover, the new generated rules have lower fitness than the other "old" rules in the population and they're often replaced in the next crossover step. In order to avoid this problem, the adopted solution is to reduce the mutation rate of the GA and to implement a roulette-wheel technique for the selection of the rules that will be used to generate the next generation elements.

## Experiment 3

For this experiment the Genetic Algorithm has been modified to use a roulette-wheel selection strategy, different mutation and crossover rates, merge-sort algorithm to rank the population. After the modifications of the GA and the introduction of new key features, the same tests as in the experiment 1 are performed. The goal is to obtain more rules that can compute the same trajectory, from a fixed initial state to a desired final state. In the first experiment the rules n. 203, 217, 218 and 219 were found but not, for instance, the rule n. 90, that is also able to reach the same desired final state after the same number of evolution steps.

The new features and modifications of the GA are, in details:

- Mutation rate modified from 0,5 to 1/L, where L is the length of the binary string needed to represent a rule. In this specific scenario, with a neighborhood of 3 elements L is equal to 8. This corresponds to a mutation rate of 1/8 (0,125) [24];
- Implementation of Roulette-Wheel technique to select the rules that will generate the next generation elements [29];
- Introduction of Crossover Rate with a probability value of 0,7 [23];
- Implementation of Merge-Sort algorithm to speed-up the ranking of the rules, according to their fitness function [46].

CA characteristics:

| | |
|---|---|
| Automaton type | 1 dimension uniform CA |
| Number of cells | 65 |
| Number of evolution steps | 64 |
| Fixed initial state | 00000000000000000000000000000001000000000000000000000000000000000 |
| Desired final state | 11111111111111111111111111111101111111111111111111111111111111111 |

Results after 1.000 simulations (in parenthesis the results of the experiment n. 1):

| | | |
|---|---|---|
| Mean number of crossovers | 20,107 | (259,491) |
| Variance | 650,800 | (5.952.946,372) |
| Standard deviation | 25,510 | (2.439,866) |
| Minimum number of crossovers | 0,000 | (0,000) |
| Maximum number of crossovers | 332,000 | (71.009,000) |
| Computation time | ~ 6 seconds | ~ 30 seconds |

Obtained rules:

- 51 00110011
- 90 01011010
- 105 01101001
- 122 01111010
- 123 01111011
- 203 11001011
- 217 11011001
- 218 11011010
- 219 11011011
- 250 11111010

**Analysis:**

In this experiment, that is a repetition on the first test, the mutation rate for the GA has been reduced to 0,125 (independent for each cell). The minimum number of evolution steps required to generate the desired rules is 0 and the maximum is 332. As the standard deviation is 25,51, this means that the majority of the computations generate a number of evolution steps between 0 and 50, with a high density around the mean value of 20,107. This is translated in a five time faster computation. Besides, more correct rules are found by the GA (in comparison with the experiment 1). This result is a first verification of the consistency of the Genetic Algorithm.

In figure 11 a comparison of experiments 1 and 3 is shown. A logarithmic scale is used.

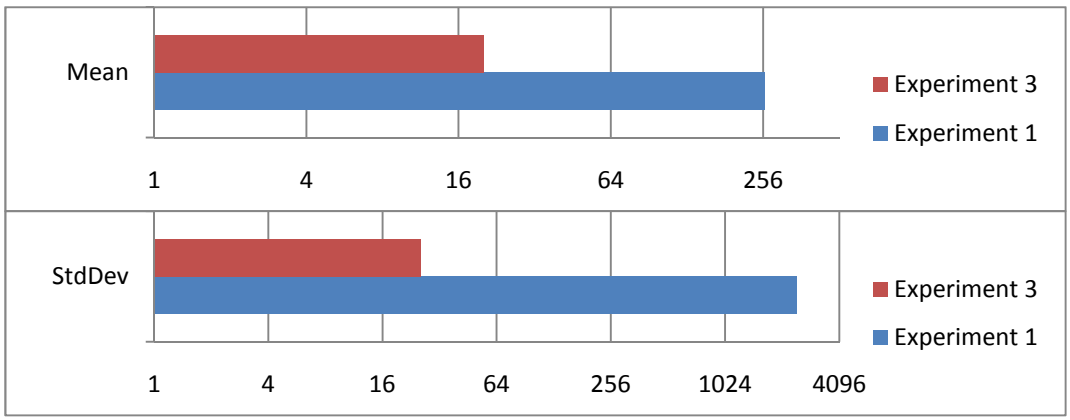



Figure 11: Comparison graph between experiment 1 and 3

## Experiment 4

The experiment 2 was not entirely successful. In this new test the same initial conditions are tested. After the modification of the Genetic Algorithm explained above, the goal is to find the rule n. 126, which is the only one that can reach the desired final state starting from the specified initial state, after 64 evolution steps.

CA characteristics:

| | |
|---|---|
| Automaton type | 1 dimension uniform CA |
| Number of cells | 65 |
| Number of evolution steps | 64 |
| Fixed initial state | 00000000000000000000000000000001000000000000000000000000000000000 |
| Desired final state | 00000000000000000000000000000011100000000000000000000000000000000 |

Results after 100 simulations (no previous results present for comparison):

| | |
|---|---|
| Mean number of crossovers | 1.480,110 |
| Variance | 2.654.941,000 |
| Standard deviation | 1.629,400 |
| Minimum number of crossovers | 2,000 |
| Maximum number of crossovers | 7.162,000 |
| Computation time | ~ 45 seconds |

Obtained rules:

- 126 01111110

**Analysis:**

In this experiment, that is a repetition on the test 2 (that has partially failed), the algorithm is working properly and it converges 100% of times. The desired rule 126 is found at every run. As it is the only rule that can reach the desired final state, starting from the selected initial state and after 64 evolution steps, the number of required iterations for the GA is bigger if compared to the experiment 3.

In the experiment 3, in fact, many rules were able to reach, through different trajectories, the same final state and more results were available in the rule state-space. Nevertheless, the goal is achieved with a mean number of GA evolution cycles of 1.480,11 with a standard deviation of 1.629,4.

## Experiment 5

In the previous experiments the desired rules to compute a trajectory, starting from a specified initial state and reaching a desired final state, have been obtained. In the first case many rules are able to generate the same behavior through different trajectories; in the second case only one rule is able to perform the desired computation.
In this experiment only one rule is able to generate the trajectory that can reach the specified final state (00100110010100000010100000010100101011111001011101101001110101100). This state is the result of the rule number 30 (in binary 00011110) after 64 evolution steps.
The rule n. 30 is supposed to be hard to find because it shows an aperiodic behavior and after a certain number of steps it is generating a unique sequence that cannot be produced by any other rule, starting from the same initial configuration. It is often used for random number generation [36].

CA characteristics:

| | |
|---|---|
| Automaton type | 1 dimension uniform CA |
| Number of cells | 65 |
| Number of evolution steps | 64 |
| Fixed initial state | 00000000000000000000000000000001000000000000000000000000000000000 |
| Desired final state | 00100110010100000010100000010100101011111001011101101001110101100 |

Results after 100 simulations (no previous results present for comparison):

| | |
|---|---|
| Mean number of crossovers | 2.946,840 |
| Variance | 11.311.211,000 |
| Standard deviation | 3.363,214 |
| Minimum number of crossovers | 2,000 |
| Maximum number of crossovers | 19.182,000 |
| Computation time | ~ 138 seconds |

Obtained rules:

- 30 00011110

**Analysis:**

As the results show, this experiment represents an instance that is harder to find. The algorithm is working properly and it converges 100% of times. The desired rule 30 is found in all the 100 simulations, with a mean number of GA evolution cycles of 2.946,84 and a standard deviation of 3.363,124. This result is promising and together with the previous two experiment results constitute a solid base and confirmation of the potential of the Genetic Algorithm and opens to a more consistent use of the GA itself. Investigation of more elaborate trajectories is now possible.

In figure 12 a comparison of experiments 4 and 5 is shown.

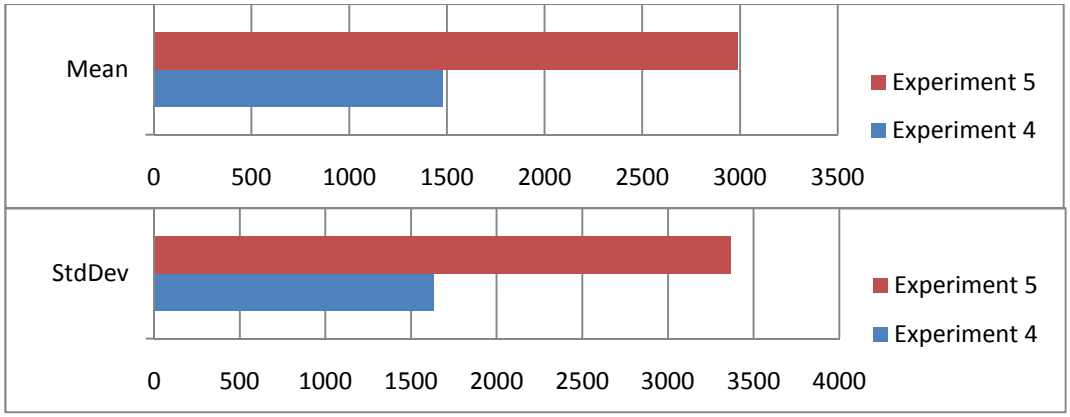



**Figure 12: Comparison graph between experiment 4 and 5**

## Experiment 6

In this experiment the size of the Cellular Automaton is increased in order to analyze and compute bigger instances of the same problem. Moreover, the performances of the Genetic Algorithm are verified. A uniform 1D Cellular Automaton with 129 cells is used. The goal is to find again the rule 30, which is the only one that can reach the desired seemingly-random final state starting from a default initial state, after 64 evolution steps.

CA characteristics:

| | |
|---|---|
| Automaton type | 1 dimension uniform CA |
| Number of cells | 129 (double the size of the previous experiments) |
| Number of evolution steps | 64 |

| | |
|---|---|
| Fixed initial state | 00000000000000000000000000000000000000000000000<br>00000000000000000000010000000000000000000000<br>00000000000000000000000000000000000000000000000 |
| Desired final state | 01101111001101001110100101111001001011111 00<br>10010110011110111101010111110010111011010<br>00100101111000100100011001011111111111111110 |

Results after 100 simulations (in parenthesis the results with a CA of size 65):

| | | |
|---|---|---|
| Mean number of crossovers | 1.196,660 | (2.946,840) |
| Variance | 4.651.759,000 | (11.311.211,000) |
| Standard deviation | 2.156,790 | (3.363,210) |
| Minimum number of crossovers | 1,000 | (2,000) |
| Maximum number of crossovers | 11.773,000 | (19.182,000) |
| Computation time | ~ 123 seconds | (~ 138 seconds) |

Obtained rules:

- 30 00011110

**Analysis:**

The results show that increasing the size of the CA to 129 cells doesn't affect the performances of the Genetic Algorithm. In fact, the computation time is less, even if the size of the instance is bigger, because fewer crossovers are performed to find the same rule (an average of 1.196,66 crossovers instead of 2.946,84 for the previous experiment). Moreover, the maximum number of performed crossovers and the standard deviation are smaller. It is possible that the state-space is explored faster because the fitness function, using a bigger CA, can distinguish better the rules and converge faster to the optimum solution.

## Experiment 7

In this experiment the size of the Cellular Automaton is increased again in order to analyze and compute bigger instances of the same problem. Also the performances of the Genetic Algorithm are verified. A uniform 1D Cellular Automaton with 257 cells is used. The goal is to find again the rule 30, which is the only one that can reach the desired final state starting from a default initial state, after 64 evolution steps.

CA characteristics:

| | |
|---|---|
| Automaton type | 1 dimension uniform CA |
| Number of cells | 257 (double the size of the previous experiments) |
| Number of evolution steps | 64 |
| Fixed initial state | 00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000<br>00000000000000000000000000000000000000000000100000000000000000000000000000000000000000000000000<br>00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000 |
| Desired final state | 00000000000000000000000000000000000000000000000000000000000000000000000011011110011010011101001<br>01011110010010111110010010110011110111101010101111100101110110100010010111100010010001<br>11001011111111111111100000000000000000000000000000000000000000000000000000000000000000000000000000000 |

Results after 100 simulations (in parenthesis the results with a CA of size 129):

| | | |
|---|---|---|
| Mean number of crossovers | 63,320 | (1.196,660) |
| Variance | 7.246,120 | (4.651.759,000) |
| Standard deviation | 85,120 | (2.156,790) |
| Minimum number of crossovers | 0,000 | (1,000) |
| Maximum number of crossovers | 767,000 | (11.773,000) |
| Computation time | ~ 8 seconds | (~ 123 seconds) |

Obtained rules:

- 30 00011110

**Analysis:**

In this experiment the size of the Cellular Automaton is doubled again, up to 257 cells. This change reduces the number of required crossovers because with 64 evolution cycles not all the cells of the CA are affected by the behavior of the rule 30. The result is a final state with a long sequence of 0 in the cells in the first and in the last positions. In the central portion of the CA there is a chaotic distribution of 1 and 0, produced by the computation of the rule 30. This means that the most difficult effort for the Genetic Algorithm is to process and obtain a sequence of "randomly" distributed 1 and 0 instead of long sequences of 1 followed by long sequences of 0 (or vice versa). This result shows that the rule 30 is a difficult instance to find, especially when a number of evolutions cycles that affect all the cells of the CA is chosen.

The average of performed GA evolution cycles is 63,32. Only the 25% of the rules state-space (composed by 256 rules) is explored before finding the correct solution.

In figure 13 a comparison of experiments 5, 6 and 7 is shown. A logarithmic scale is used.

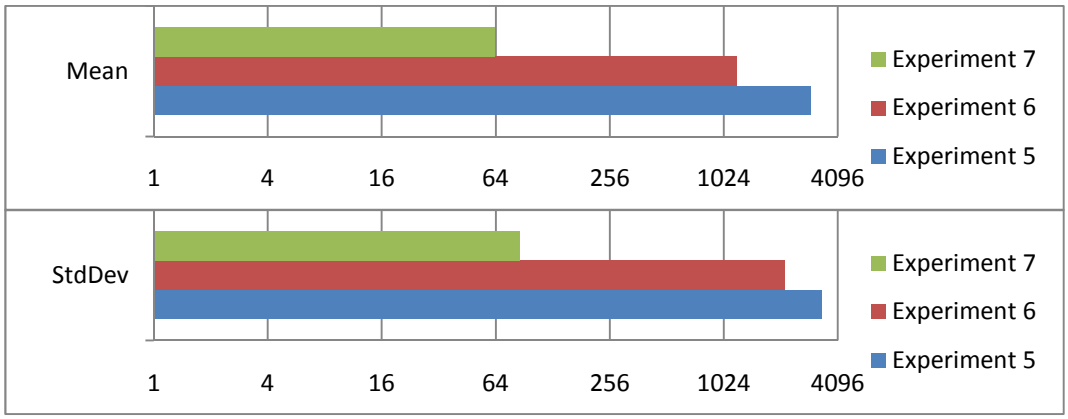



**Figure 13: Comparison graph among experiment 5, 6 and 7**

## Experiment 8

In this experiment the size of the Cellular Automaton is maintained to 257 but the difficulty of the problem is increased. The goal of the experiment is to find the rule n. 30, starting from a random initial state and reaching the desired final state that is known to be generated by the computation of the rule number 30 (previously computed using the uniform CA engine).

CA characteristics:

| | |
|---|---|
| Automaton type | 1 dimension uniform CA |
| Number of cells | 257 |
| Number of evolution steps | 64 |
| Fixed random initial state | 00111001100111001110101001110000010101001001011011000000010110011101011100010001100100<br>10010110100000001110100000110111110010000111110110001001000010000001011001111011100 01<br>10010001111110011001001110101011001100100100011101000010000100101101101001100011001001 |
| Desired final state | 00000110000001011110001010000011001111100110000011000111110010011111100010101011011000<br>01010110011100000100100001110011111110101100110011001110111111001001100001101110000 10<br>00000010100001101101110010101111101110000110110001011111010101110011000100111100010001 |

Results after 100 simulations:

| | | |
|---|---|---|
| Mean number of crossovers | 28.646,630 | |
| Variance | 2.463.460.693,000 | |
| Standard deviation | 49.633,260 | |
| Minimum number of crossovers | 1,000 | |
| Maximum number of crossovers | 216.977,000 | |
| Computation time | ~ 5983 seconds | (around 100 minutes) |

Obtained rules:

- 30 00011110

**Analysis:**

This experiment is a repetition of the previous test, in a way that all the cells are affected by the behavior of the rule 30 after 64 evolution steps. This is done starting the computation of the CA from a randomly generated initial state. This specific instance is supposed to be hard to compute. Nevertheless, the desired rule is found 100% of times, even if the number of required GA cycles is very big (compared to the size of the rules state-space). The explanation is that, starting from a random initial state, there are several rules that are able to generate a final state with a small Hamming Distance, or in terms of fitness function a value close to the maximum. The GA loops in local maximum before falling in the final solution and the state-space is not explored rapidly.

As reported in [27] there is an optimized GA that can explore faster the state-space using large mutation rates and population-elitist selection. Also in [28] it is explained that the crossover rate can be tuned in order to increase the exploration speed of the algorithm. As this is not the main goal of the project, more effort will be put on the investigation and research of specific trajectories, as described in the following experiments.

In figure 14 a comparison of experiments 7 and 8 is shown. A logarithmic scale is used.

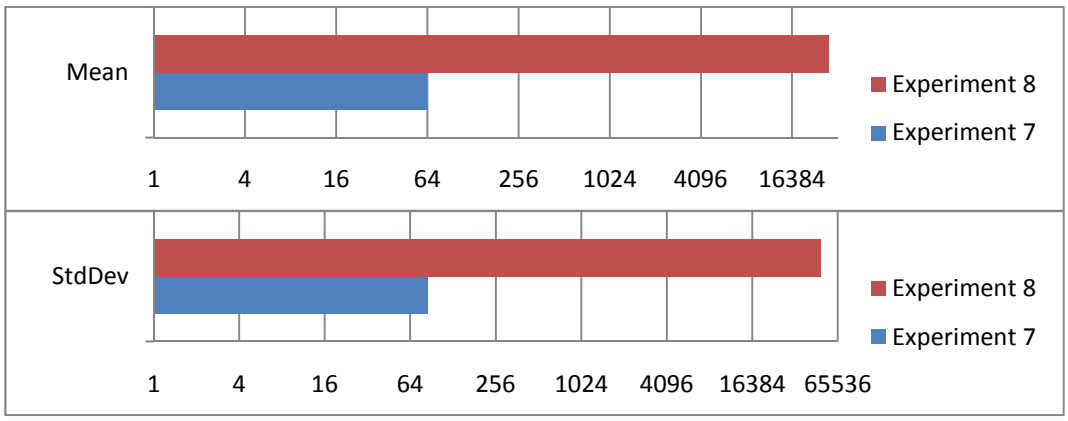



**Figure 14: Comparison graph between experiment 7 and 8**

## Experiment 9

In this experiment the goal is to find the rules that can obtain a specified computation described by the following trajectory, where an intermediate state has to be reached in the middle of the computation at a particular step:

**INITIAL STATE ----- n steps -----> INTERMEDIATE STATE ----- n steps -----> FINAL STATE**

In fact, it is harder for the GA to find the rules because the desired trajectory is described not only by an initial and final state but also it has to reach a well defined intermediate state at a specified evolution step (in the middle of the computation, at the evolution step 32).

CA characteristics:

| | |
|---|---|
| Automaton type | 1 dimension uniform CA |
| Number of cells | 65 |
| Number of evolution steps | 64 |
| Fixed initial state | 00000000000000000000000000000001000000000000000000000000000000000 |
| Desired intermediate state (at evolution step n. 32) | 01101111001101001110100101111000001111000001111011010101111111110 |
| Desired final state | 00100110010100000010100000010100101011111001011101101001110101100 |

Results after 100 simulations (in parenthesis the results without the intermediate state in the trajectory):

| | | |
|---|---|---|
| Mean number of crossovers | 692,740 | (2.946,840) |
| Variance | 2.525.284,000 | (11.311.211,000) |
| Standard deviation | 1.589,110 | (3.363,210) |
| Minimum number of crossovers | 0,000 | (2,000) |
| Maximum number of crossovers | 9.596,000 | (19.182,000) |
| Computation time | ~ 26 seconds | (~ 138 seconds) |

Obtained rules:

- 30 00011110

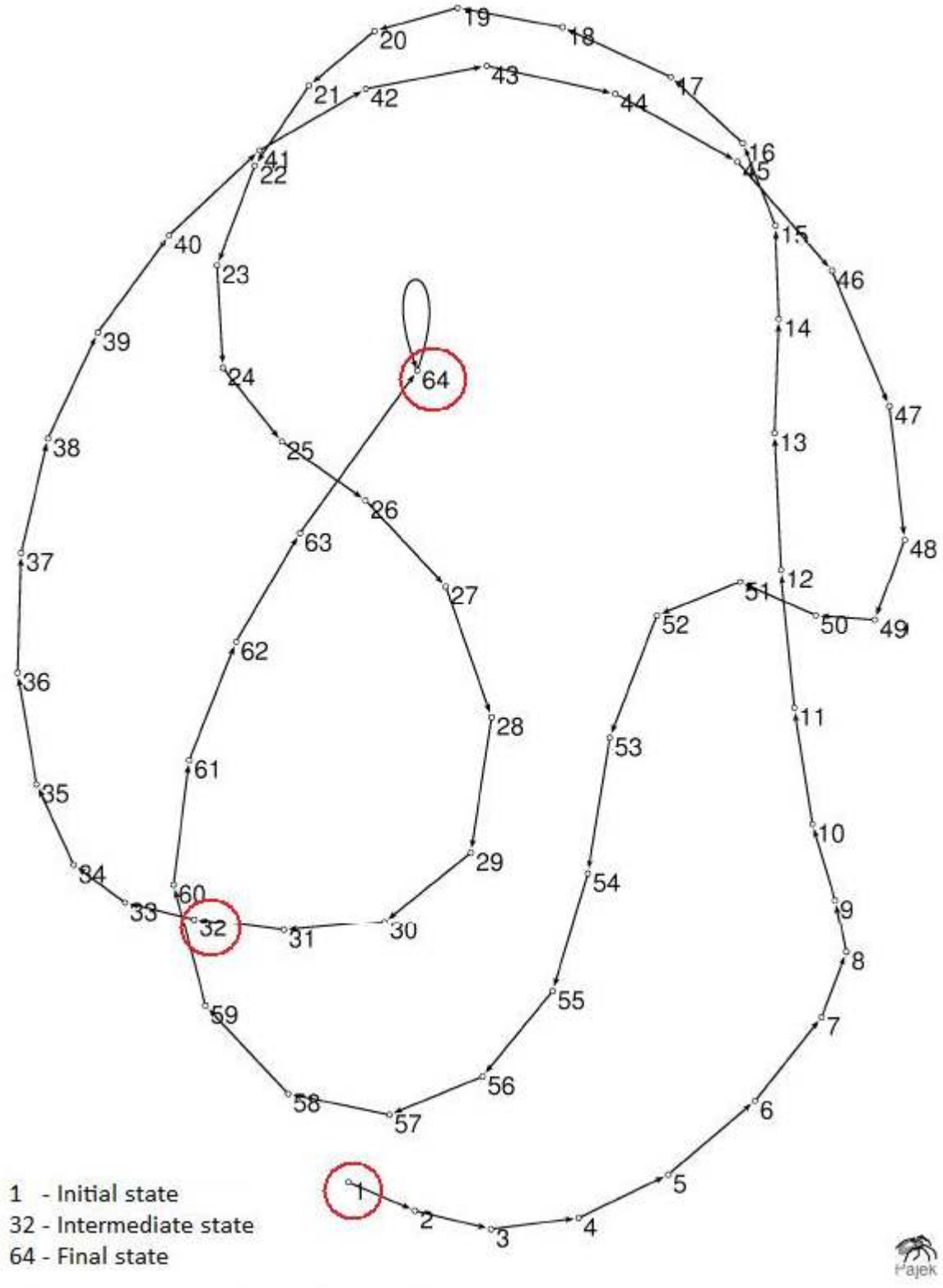



Figure 15: Trajectory through intermediate state

In Figure 15 the obtained trajectory is shown. The highlighted states (initial, intermediate and final) are specified and the rule that can reach them is found.
The plot of the trajectory has been done by PAJEK using the Fruchterman Reingold 3d option.

**Analysis:**

At a first analysis the results seem surprising. Despite the introduction of a specified intermediate state at the step 32 that must be matched in the CA trajectory (in the middle of the computation), the GA converges to the best solution faster. This is done introducing a weight parameter for the fitness function. If the intermediate state does not match, the weight of the final fitness function becomes 0,5, even if the final state matches completely. In this way, lot of importance is given to the first intermediate state in the trajectory. If the intermediate state is found, the rule is close to the correct one with a high degree of probability, even if the final desired state is not found (the first part of the trajectory is correct). In the other case, if the final state is found but not the intermediate state, the rule can be completely different (the intermediate trajectory can be completely dissimilar but it falls in the same final state). Analyzing the numerical results, it is possible to notice that the number of performed crossover is 692,74, the 75% less than without an intermediate state. This is also translated in a reduction of the computation time. Specifying an intermediate state, the GA gains more information in order to perform the desired task.

In figure 16 a comparison of experiments 5 and 9 is shown.

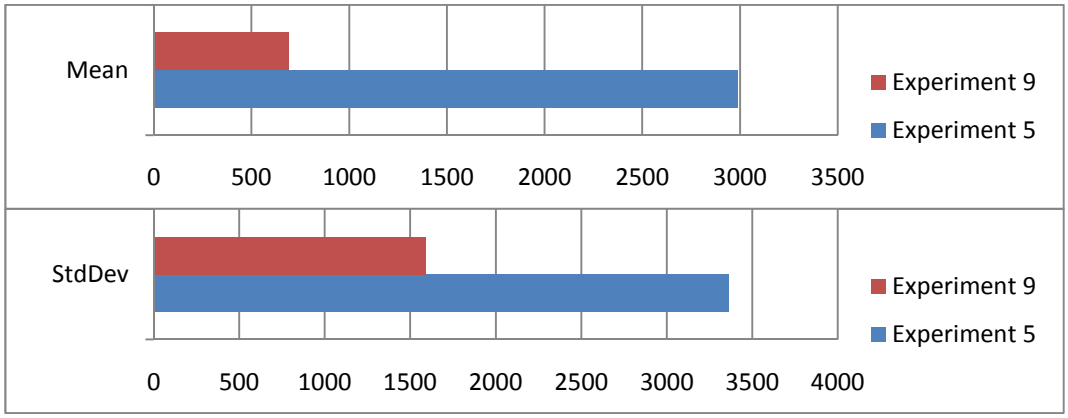



**Figure 16: Comparison graph between experiment 5 and 9**

## Experiment 10

The goal of the experiment 10 (and 11) is to search for rules that can generate a computation with the same final state through different trajectories (and different intermediate states).
In order to understand the problem, a graphic result of the computation of the used rules is given in the following table:

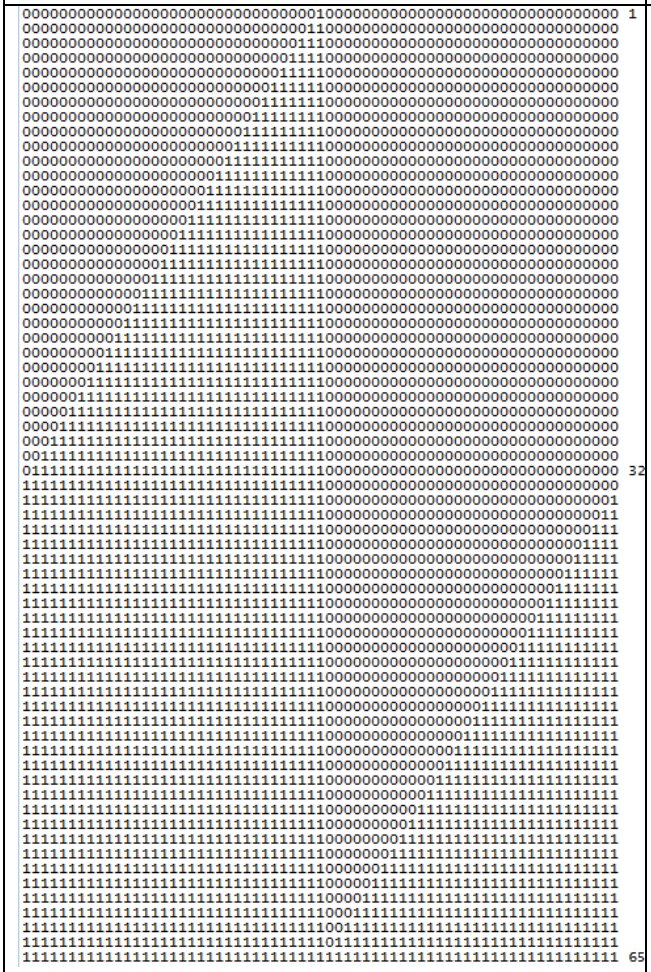
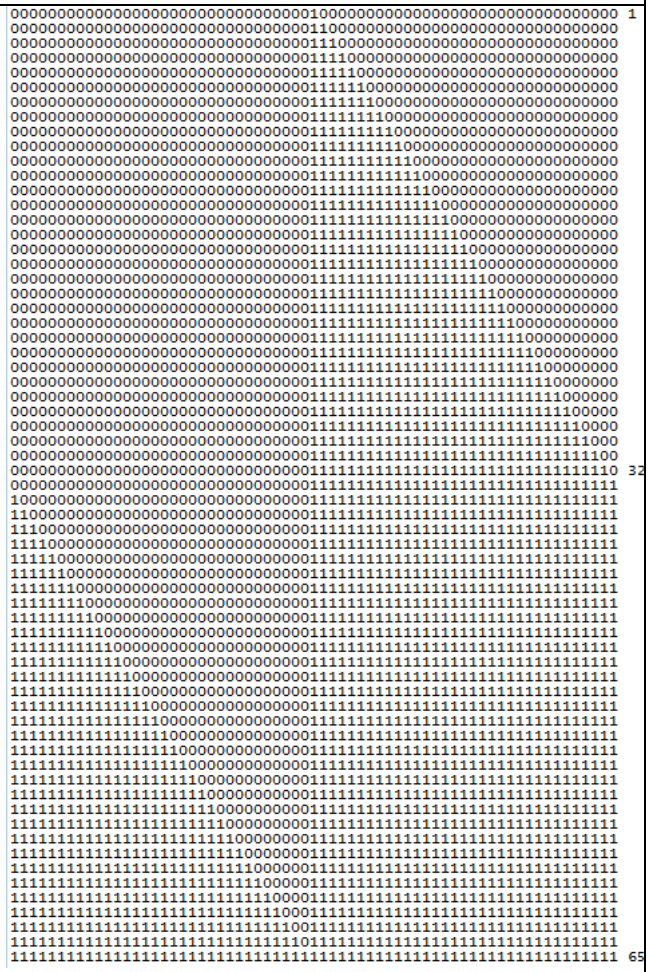
| rule 238 - 11101110 | rule 252 - 11111100 |
|---|---|
|  |  |

Table 3: Comparison between Rule 238 and Rule 252

As shown in the previous table, starting from the same initial state, the rules 238 and 252 can reach the same final state attractor after the same number of steps (65), but through different trajectories and intermediate states. For instance, at the evolution step 32 the bit-string that represents the state is mirrored. In general, it is possible to observe that the two rules have a symmetric behavior but finally they reach the same attractor.

In the experiment 10 the objective is, given the right initial, intermediate and final state, to obtain the rule 238.

CA characteristics:

| | |
|---|---|
| Automaton type | 1 dimension uniform CA |
| Number of cells | 65 |
| Number of evolution steps | 65 |
| Fixed initial state | 00000000000000000000000000000001000000000000000000000000000000000 |
| Desired intermediate state (at evolution step n. 32) | 01111111111111111111111111111110000000000000000000000000000000000 |
| Desired final state | 11111111111111111111111111111111111111111111111111111111111111111 |

Results after 100 simulations:

| | |
|---|---|
| Mean number of crossovers | 81,580 |
| Variance | 4.540,740 |
| Standard deviation | 67,390 |
| Minimum number of crossovers | 0,000 |
| Maximum number of crossovers | 311,000 |
| Computation time | ~ 4,977 seconds |

Obtained rule:

- 238 11101110

## Experiment 11

Using the same set-up as in the experiment 10, except for the intermediate state, the goal is to find the rule 252.

The rule 252 is reaching the same attractor as the rule 238 (starting from the same default configuration) but through different intermediate states. In this experiment the intermediate state of rule 252 at evolution step n. 32 is considered.

CA characteristics:

| | |
|---|---|
| Automaton type | 1 dimension uniform CA |
| Number of cells | 65 |
| Number of evolution steps | 65 |
| Fixed initial state | 00000000000000000000000000000001000000000000000000000000000000000 |
| Desired intermediate state (at evolution step n. 32) | 00000000000000000000000000000001111111111111111111111111111111110 |
| Desired final state | 11111111111111111111111111111111111111111111111111111111111111111 |

Results after 100 simulations:

| | |
|---|---|
| Mean number of crossovers | 84,540 |
| Variance | 9.403,710 |
| Standard deviation | 96,970 |
| Minimum number of crossovers | 0,000 |
| Maximum number of crossovers | 474,000 |
| Computation time | ~ 5,165 seconds |

Obtained rule:

- 252 11111100

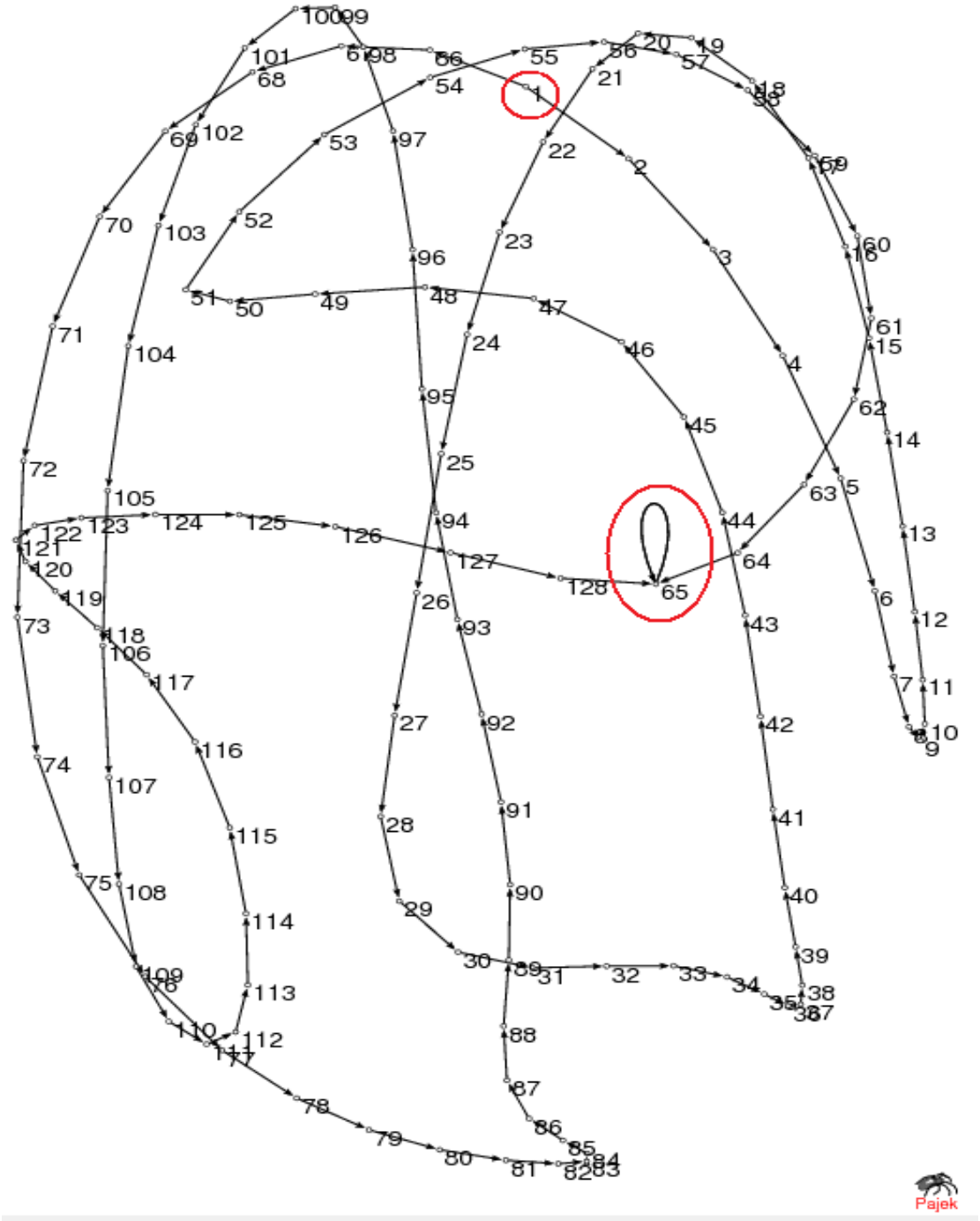



Figure 17: Trajectory comparison - rules 238 and 252

In Figure 17 the trajectories of the rules 238 and 252 are graphically compared.
As in the experiments 10 and 11 the computation starts for both rules from the same state, the initial state is shared and denoted by 1.
The two rules follow completely different trajectories. The second state (at the second evolution step) for the rule 252 is the state denoted by 2 and for the rule 238 it is denoted by 66.
At the end of the computation both fall in the same attractor state denoted by 65 (rule 238 from the state number 128 and rule 252 from the state denoted by 64).

**Analysis:**

In those two experiments the goal was to find different rules using a GA, starting from the same state and arriving in the same attractor through different trajectories (reaching during the computation a different intermediate state). In both the experiments the desired rule is found (238 and 252). The analysis of the computational time and the mean number of crossover performed confirms that the introduction of an intermediate state in the computation of the GA doesn't affect the performances.

## Experiment 12

The goal of the experiment 12 (and 13) is similar to the previous two tests. In fact, the objective is to search for rules that can generate a computation with the same intermediate state, following the same trajectories but reaching a different final state / attractor.
In order to understand the problem, a graphic result of the computation of the used rules is given in the following table:

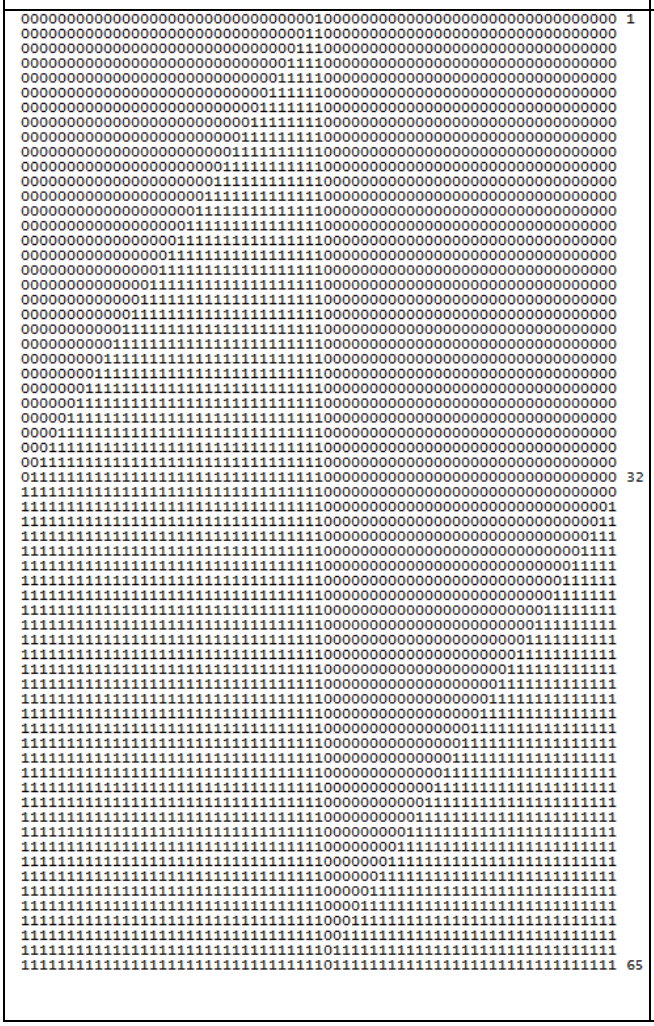
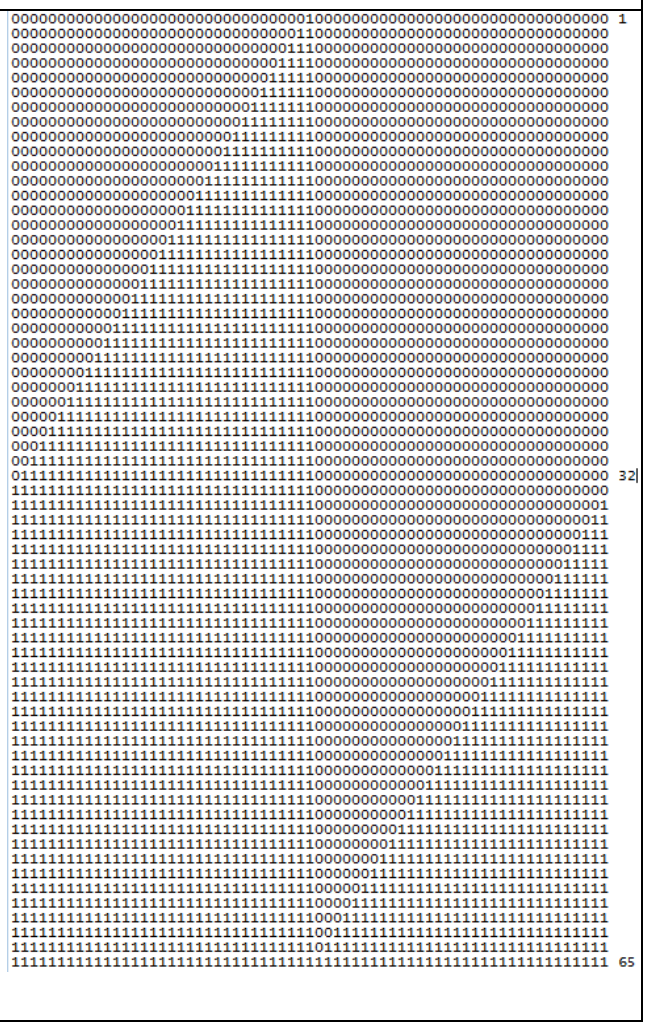
| rule 206 - 11001110 | rule 238 - 11101110 |
| --- | --- |
|  |  |

Table 4: Comparison between Rule 206 and Rule 238

As shown in the previous table, starting from the same initial state, the rules 206 and 238 follow the same trajectory and reach the same intermediate state at the step number 32. After that, the two computations continue in the same path until the evolution step 64. It is not possible to distinguish between the two rules considering the first 64 evolution steps. In the evolution step n. 65 the reached state is different.

The attractor for the rule 206 is:

11111111111111111111111111111111**0**11111111111111111111111111111111

and the attractor for the rule 238 is:

11111111111111111111111111111111**1**11111111111111111111111111111111

(the bit in red is different).

In the experiment 12 the objective is, given the right initial, intermediate and final state, to obtain the rule 206.

CA characteristics:

| | |
|---|---|
| Automaton type | 1 dimension uniform CA |
| Number of cells | 65 |
| Number of evolution steps | 65 |
| Fixed initial state | 00000000000000000000000000000001000000000000000000000000000000000 |
| Desired intermediate state (at evolution step n. 32) | 01111111111111111111111111111110000000000000000000000000000000000 |
| Desired final state | 11111111111111111111111111111101111111111111111111111111111111111 |

Results after 100 simulations:

| | |
|---|---|
| Mean number of crossovers | 113,150 |
| Variance | 17.863,010 |
| Standard deviation | 133,650 |
| Minimum number of crossovers | 0,000 |
| Maximum number of crossovers | 636,000 |
| Computation time | ~ 6,725 seconds |

Obtained rule:

- 206 11001110

## Experiment 13

Using the same set-up as in the experiment 12, except for the desired final state, the goal is to find the rule 238.

The rule 238 is following the same trajectory as the rule 206 (starting from the same default configuration) but at the evolution step 65 they reach different attractors.

CA characteristics:

| | |
|---|---|
| Automaton type | 1 dimension uniform CA |
| Number of cells | 65 |
| Number of evolution steps | 65 |
| Fixed initial state | 00000000000000000000000000000001000000000000000000000000000000000 |
| Desired intermediate state (at evolution step n. 32) | 01111111111111111111111111111110000000000000000000000000000000000 |
| Desired final state | 11111111111111111111111111111111111111111111111111111111111111111 |

Results after 100 simulations:

| | |
|---|---|
| Mean number of crossovers | 83,21 |
| Variance | 8.281,07 |
| Standard deviation | 91,00 |
| Minimum number of crossovers | 0,00 |
| Maximum number of crossovers | 386,00 |
| Computation time | ~ 5,092 seconds |

Obtained rule:

- 238 11101110

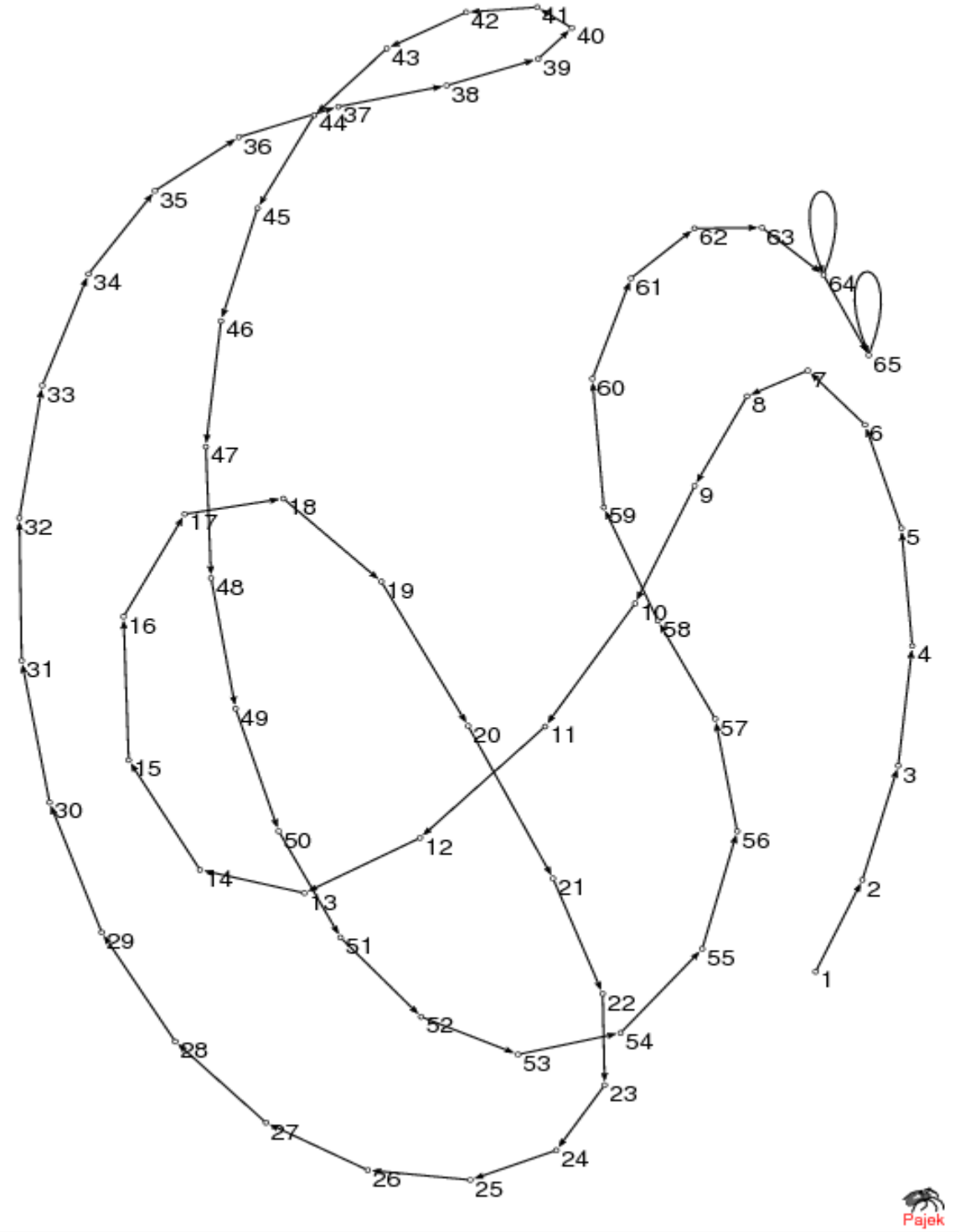



Figure 18: Trajectory comparison - rules 206 and 238

In Figure 18 the trajectories of the rules 206 and 238 are compared graphically.

The rule 206 computes the trajectory from the state denoted by 1 to the state denoted by 64, after it loops in the state attractor.

The rule 238 generates the same trajectory, but from state n. 64 it performs another evolution step until the state denoted by 65, that is its space attractor.

**Analysis:**

In those two experiments the goal was to find different rules using a GA, following the same trajectories and reaching the same intermediate state but falling into two different attractors (the trajectories are completely the same except for the last computation step which falls in different final states). In both the experiments the desired rule is found (206 and 238). The analysis of the computational time and the mean number of crossover performed is consistent with the results shown in the two previous experiments.

In figure 19 a comparison of experiments 10, 11, 12 and 13 is shown.

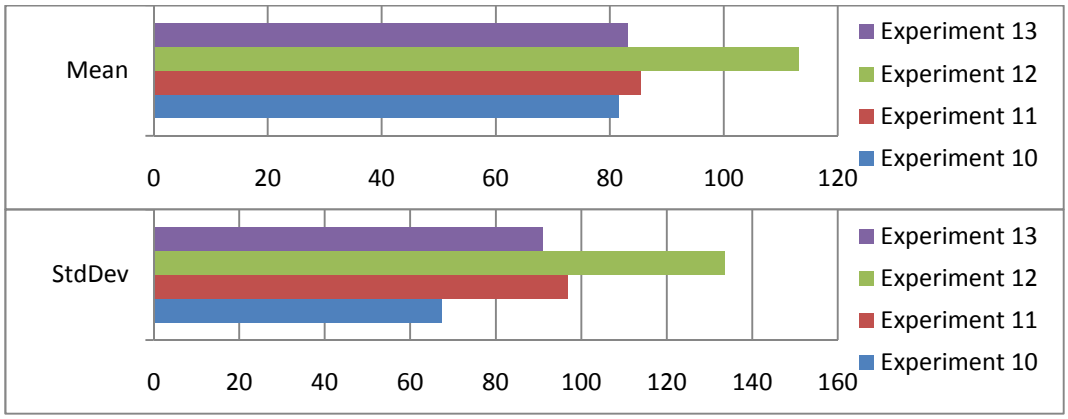



**Figure 19: Comparison graph among experiment 10, 11, 12 and 13**

## Experiment 14

In this experiment a trajectory with two different checkpoints is investigated. Furthermore, the two intermediate states can be found in two different intervals instead of two specific points. The goal of this test is to find the rule 206, given a specific initial state, a defined attractor and two intermediate states that have to be found not in a precise evolution step but inside specified ranges (between step 10 and 20 the first and between step 40 and 50 the second).

CA characteristics:

| | |
|---|---|
| Automaton type | 1 dimension uniform CA |
| Number of cells | 65 |
| Number of evolution steps | 65 |
| Fixed initial state | 00000000000000000000000000000001000000000000000000000000000000000 |
| First intermediate state (between step 10 and 20) | 00000000000000000011111111111111100000000000000000000000000000000 |
| Second intermediate state (between step 40 and 50) | 11111111111111111111111111111110000000000000000000011111111111111 |
| Desired final state | 11111111111111111111111111111101111111111111111111111111111111111 |

Results after 100 simulations:

| | |
|---|---|
| Mean number of crossovers | 120,550 |
| Variance | 14.176,530 |
| Standard deviation | 119,070 |
| Minimum number of crossovers | 0,000 |
| Maximum number of crossovers | 700,000 |
| Computation time | ~ 7,661 seconds |

Obtained rule:

- 206 11001110

In Figure 20 the obtained trajectory is shown.

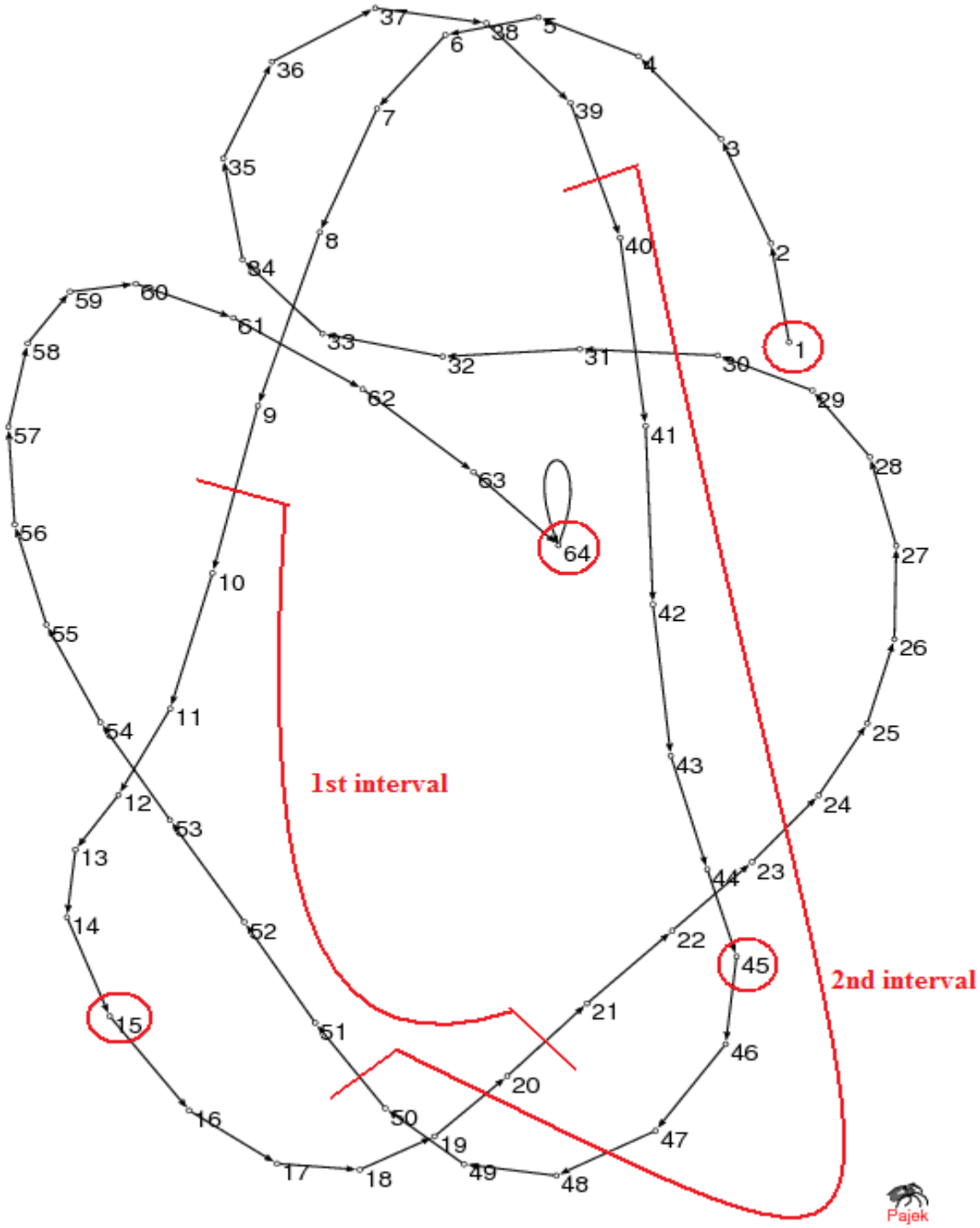



Figure 20: Trajectory with two intermediate states in intervals

1: initial state

15: checkpoint 1 (interval between 10 and 20)

45: checkpoint 2 (interval between 40 and 50)

64: attractor (in the experiment the final state is at the evolution step 65; in order to distinguish with the rule 238, that has the same trajectory until the state 64 but a different attractor at the evolution step 65, a loop is designed graphically in the trajectory at the evolution step 64)

**Analysis:**

In this experiment two intermediate checkpoints have been introduced during the trajectory. The correct positions in which they are supposed to be are at the step n. 15 and at the step n. 45. In order to introduce some flexibility, the algorithm is set up to find the specified intermediate states inside intervals, respectively between steps 10 and 20 and between steps 40 and 50. The GA works as follow: the initial weight of the fitness functions is 1. For every missed checkpoint (the state is not found inside the desired interval) the weight is reduced of 0,3 and at the end of the computation of the CA the fitness function is recalculated and weighted. With this test, it is also proven that only the correct rule, the n. 206, is found. The rule 238, for instance, has the same behavior and the same checkpoints starting from the same initial configuration, but it falls into another attractor at the step n. 65. Even with the introduction of a second intermediate state and two intervals, the computational time and the mean number of performed crossover is not affected.

In figure 21 a comparison of experiments 13 and 14 is shown.

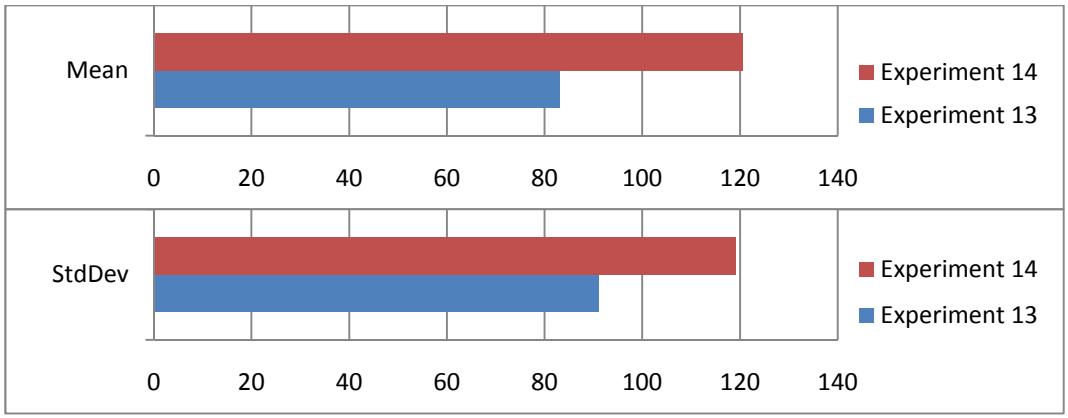



**Figure 21: Comparison graph between experiment 13 and 14**

## Experiments on non-uniform CAs

As explained in the Cellular Automata chapter, a non uniform CA has a different rule-set for every cell.

This means that for and automaton of size 65, each of the cells can have one of the 256 possible rules. This is translated in a huge state-space of size 256^65 (~ 3,43 x 10 ^ 156). Therefore, it can be difficult to search for a specific rule to compute a given trajectory in such a vast space. In order to decrease this problem, a reduced rule-set is considered.

The reduced rule-set is composed by 12 rules based on Logic Boolean operations such as Identity, OR, NAND and XOR. The obtained state-space, for an automaton of the same dimension, has a size of 12^65 (~ 1,4 x 10 ^ 70 – still huge).

A list of the rules with the relative code is given to facilitate the reading of the following results:

- Rule 0 N = IDENTITY C
- Rule 1 N = IDENTITY L
- Rule 2 N = IDENTITY R
- Rule 3 N = OR L, C
- Rule 4 N = OR C, R
- Rule 5 N = OR L, R
- Rule 6 N = XOR L, C
- Rule 7 N = XOR C, R
- Rule 8 N = XOR L, R
- Rule 9 N = NAND L, C
- Rule 10 N = NAND C, R
- Rule 11 N = NAND L, R

C represents the current value of the cell, N the next value, L and R the value of left and right neighbors.

As the structure of the non-uniform CA is different from the structure of the uniform CA, a new Genetic Algorithm is required. Basically the main structure is the same but the new GA works without crossover, using only mutation. The details are explained in the Genetic Algorithm chapter.

In a non-uniform CA the dependency between genes is different from a uniform CA and theoretically more evaluations are needed to search for the desired rule. For this reason, in the following experiments a smaller amount of simulations is executed.

## Experiment 15

In this experiment a 1-dimension non-uniform CA is considered. Starting from a standard configuration with only a cell in the middle, the goal is to achieve the same final state obtained with the computation of the rule 126 for a uniform CA, after 64 evolution steps.

CA characteristics:

| | |
|---|---|
| Automaton type | 1 dimension non-uniform CA |
| Number of cells | 65 |
| Number of evolution steps | 64 |
| Fixed initial state | 00000000000000000000000000000001000000000000000000000000000000000 |
| Desired final state | 00000000000000000000000000000011100000000000000000000000000000000 |

Results after 100 simulations:

| | |
|---|---|
| Mean number of crossovers | 310,350 |
| Variance | 34.339,550 |
| Standard deviation | 185,309 |
| Minimum number of crossovers | 88,000 |
| Maximum number of crossovers | 1.194,000 |
| Computation time | ~ 46 seconds |

The rule-sets obtained in the 100 simulations are totally different. Some examples are given in the next table. The sequence of numbers represents the code of the rule for each of the 65 cells.

| |
|---|
| 2 2 0 1 6 8 2 0 4 7 2 6 5 2 1 4 1 6 1 6 6 0 2 3 3 4 7 3 7 8 3 4 5 5 4 3 1 8 6 0 2 8 1 0 4 8 2 4 6 7 9 2 4 4 6 2 5 5 4 2 6 4 3 5 3 |
| 1 4 8 3 6 5 7 0 6 1 0 8 3 0 5 2 5 3 6 5 2 6 1 7 8 0 7 5 1 7 10 5 10 10 0 4 5 5 5 6 1 4 0 7 4 4 5 3 3 5 8 4 4 7 7 6 2 7 6 3 6 3 0 1 1 |
| 7 8 2 6 7 5 7 3 6 3 8 4 5 2 2 1 5 5 6 8 4 1 3 8 2 5 0 6 5 0 6 8 0 2 9 4 6 6 1 6 8 3 0 5 2 2 3 0 2 8 5 8 7 2 2 7 6 6 0 1 7 8 8 8 0 |
| 6 4 8 4 1 6 4 5 5 4 7 6 1 3 2 1 4 1 0 5 4 1 7 4 2 5 6 3 3 1 6 2 3 11 7 3 8 4 1 7 0 7 2 3 5 3 1 3 5 2 5 2 8 8 8 5 2 1 7 3 7 10 8 8 11 |
| 2 7 3 4 6 3 6 5 7 6 4 0 2 8 1 7 9 4 2 4 4 1 7 9 7 8 4 4 0 7 6 4 0 10 0 5 0 0 6 8 6 3 3 0 1 5 7 0 7 1 8 1 5 3 8 2 7 1 0 4 5 6 1 2 3 |
| 7 4 5 8 0 4 0 3 2 5 5 8 2 3 2 8 7 6 6 0 2 8 8 0 7 2 6 2 3 0 1 2 3 2 9 7 3 3 3 5 2 0 8 3 2 6 3 3 0 5 8 5 7 1 2 3 5 4 8 8 1 3 7 0 6 |
| 0 7 1 0 5 3 5 2 0 2 3 1 7 9 8 9 0 4 4 8 3 2 6 5 0 4 1 1 2 3 3 11 0 3 6 7 7 1 6 6 4 1 3 4 8 5 2 3 8 1 1 2 3 7 3 6 6 0 2 1 6 2 1 3 4 |
| 5 5 7 6 1 0 7 4 2 8 4 6 7 1 6 3 1 2 8 4 2 0 8 8 8 8 2 7 2 7 3 11 11 9 0 2 5 8 2 3 3 2 3 3 7 0 8 1 3 8 8 7 1 1 2 6 4 6 5 4 0 0 8 5 4 |
| 4 6 8 2 4 5 0 4 3 5 4 4 7 2 0 1 2 2 2 5 2 7 5 4 7 1 6 4 8 3 3 4 5 11 0 6 4 1 0 6 1 3 7 3 7 9 7 7 1 0 2 3 8 2 2 2 2 0 0 6 4 2 4 2 7 |
| 3 7 2 5 3 5 0 5 3 2 3 10 6 7 0 0 4 6 8 3 8 5 5 5 8 0 3 1 6 8 1 9 1 1 4 8 6 5 6 8 7 4 0 7 4 5 2 7 2 4 1 3 6 3 7 1 6 6 8 0 4 6 6 1 0 |

**Table 5: Non-uniform CA rule-set example**

**Analysis:**

Starting from this experiment, the investigation of trajectories for one dimensional non-uniform cellular automata is started. Over 100 simulations, in every test a correct rule is found. There are several rules that can reach the same final state through different paths. In fact, the rule-set found are completely different. Most of them have a ripetitive computation.

Two examples are given below in Figure 22:

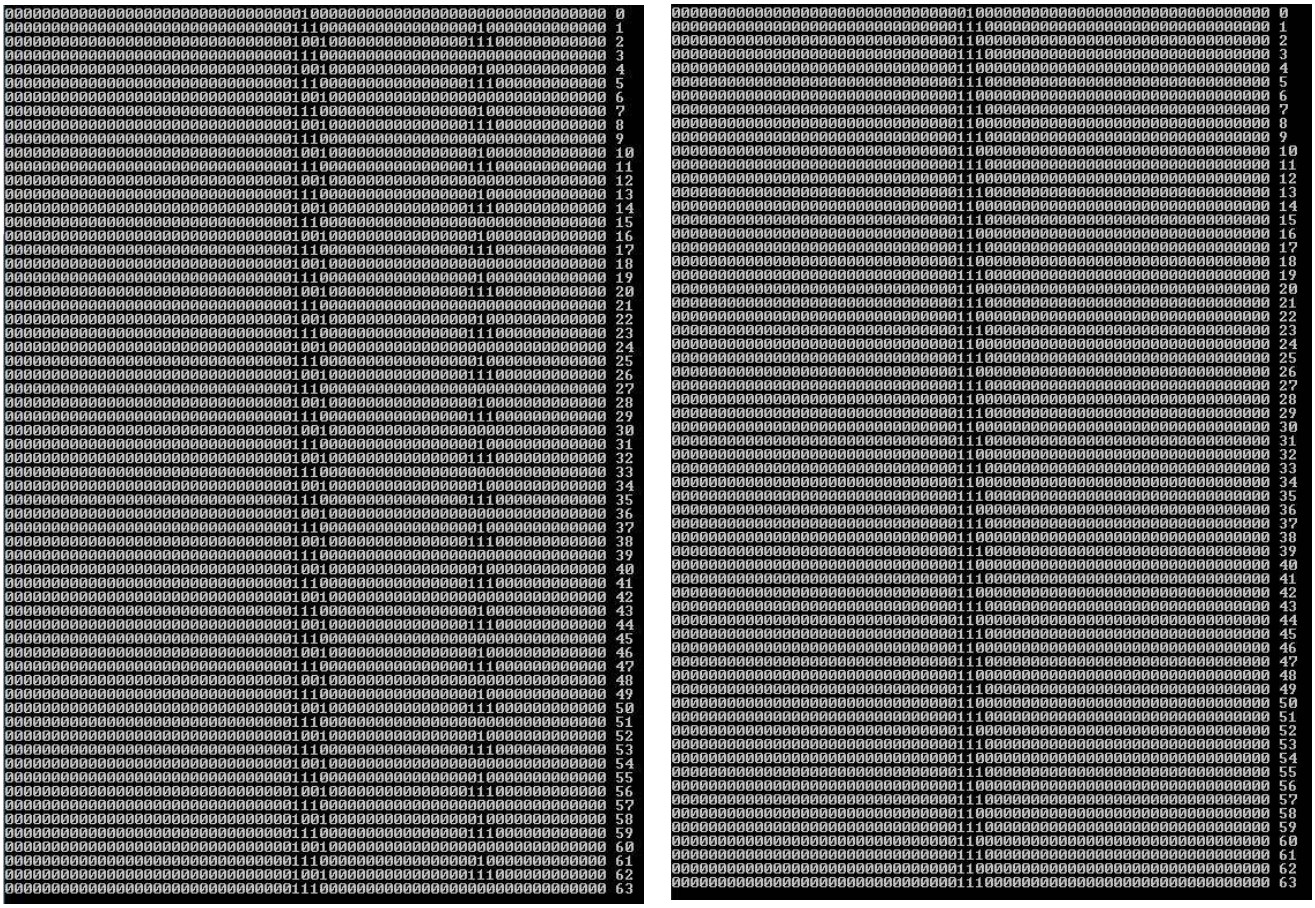


**Figure 22: Non-Uniform CA computation example**

It is possible to see that both rules reach the same final state and both are producing loops. The rule 126 for uniform CAs is reaching the same final state but the computation is more symmetric and the result is propagating from the centre to the left and right regularly.

The computational result is encouraging because the mean number of GA evolution steps is 310 (with minimum 88 and maximum 1194). The computation is fast, the time is similar to the experiment 4 for uniform CAs (46 vs 45 seconds) even if the number of performed crossovers is different.

Consequently, it is possible to proceed with CAs with bigger number of cells.

In figure 23 a comparison of experiments 4 and 15 is shown.

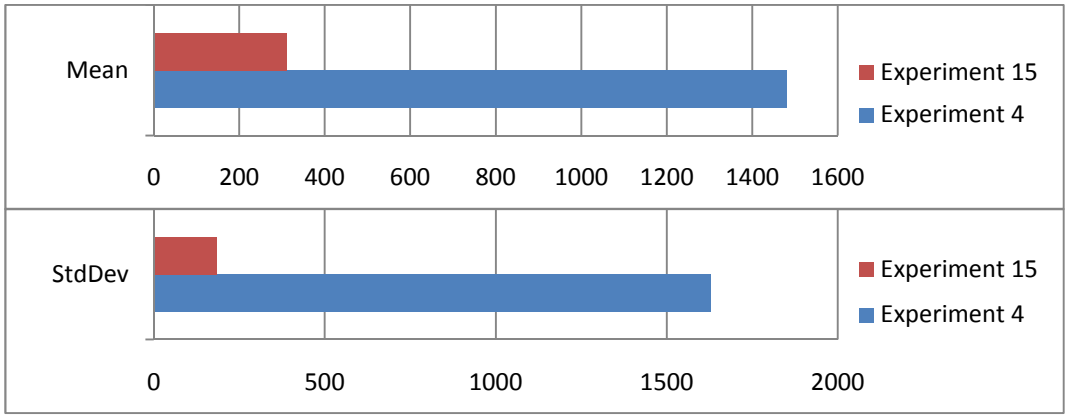



**Figure 23: Comparison graph between experiment 4 and 15**

## Experiment 16

In this experiment, starting from a standard configuration with only a cell in the middle, the goal is to achieve the same final state obtained with the computation of the rule 30 for a uniform CA (showing a pseudo-random behavior), after 64 evolution steps. The considered non-uniform CA has a size of 129 cells.

CA characteristics:

| | |
|---|---|
| Automaton type | 1 dimension non-uniform CA |
| Number of cells | 129 |
| Number of evolution steps | 64 |
| Fixed initial state | 00000000000000000000000000000000000000000000000000000000000010000000000000000000000000000000000000000000000000000000000000 |
| Desired final state | 01101111001101001110100101111001001011111001001011001111011110101011111100101110110100010010111100010010001100101111111111111110 |

Results:

This experiment has been simulated only 4 times because of the computational demand and long execution time.

The algorithm converged after respectively 34.034, 147.023, 190.695 and 80.446 steps of the Genetic Algorithm.

The last run, with 80446 steps, has been analyzed in terms of fitness. This can be useful to understand the convergence frequency:

- 110/129 after ~ 5.000 cycles
- 120/129 after ~ 16.000 cycles
- 125/129 after ~ 33.000 cycles
- 127/129 after ~ 40.000 cycles
- 128/129 after ~ 69.000 cycles
- 129/129 after 80.446 cycles

The maximum fitness value is represented by the size of the CA, where all the resulting bits are matching with the desired final state. If there is a fitness value of 110/129, there are 110 bits matching between the desired and the obtained final state.

**Analysis:**

Even if the computation was slow, this is an interesting result because the state-space for this experiment is 12 ^ 129 (number of possible rule-sets). In the worst simulation the solution is found after ~ 190.000 GA evolution cycles.

In the following image (Figure 24) the relation between the growth of the fitness function (X) and the number of GA evolution cycles (Y) is shown. The analyzed case reaches the maximum fitness function of 129/129 after 80446 cycles.

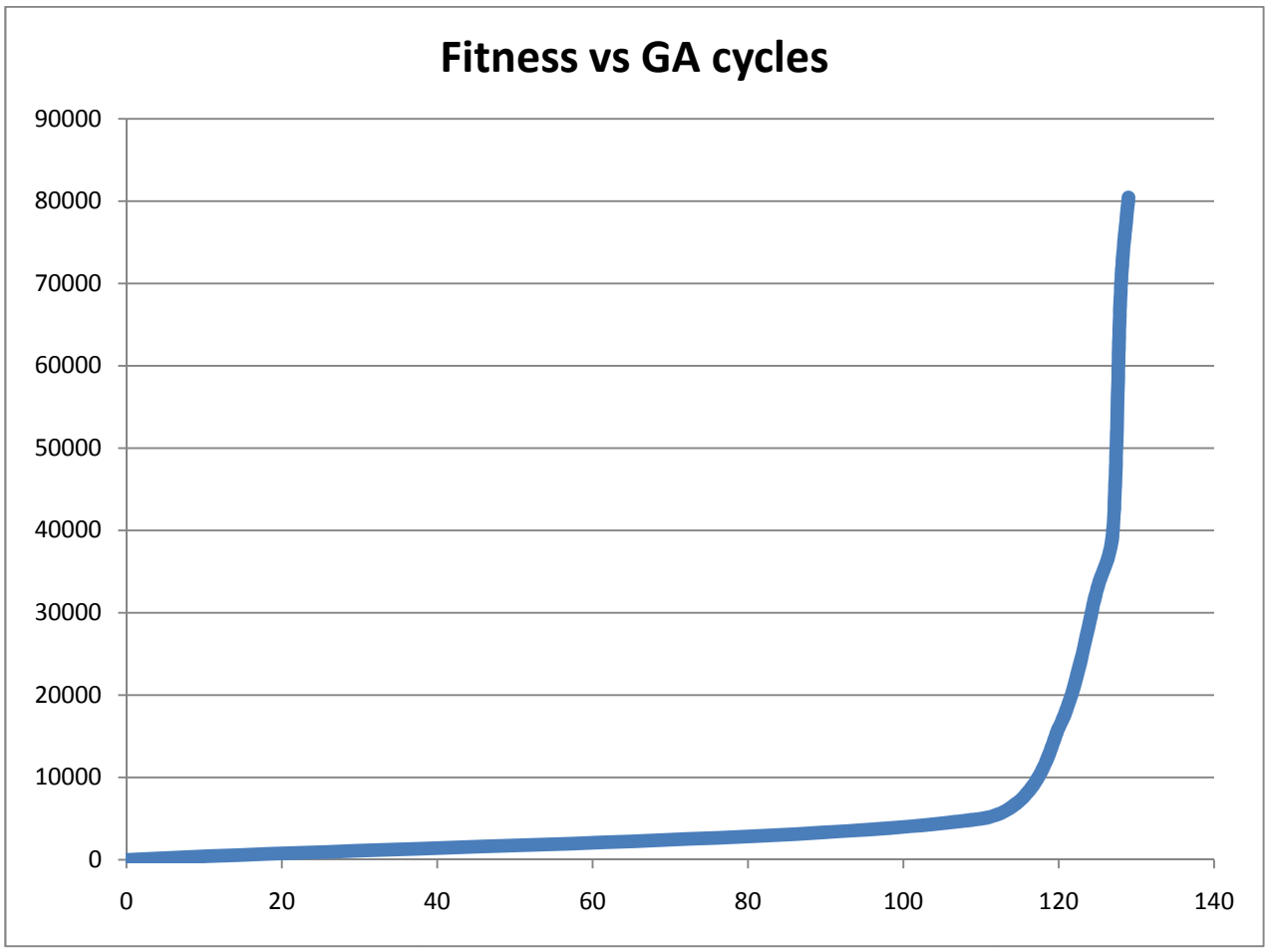



Figure 24: Fitness function VS GA evolution cycles

The chart confirms that in the beginning of the evolution the growth of the fitness function is fast and in the end of the computation, with fitness values close to the maximum, several GA cycles are needed in order to improve the fitness of the rule. The trend is exponential.

In figure 25 a comparison of experiments 6 and 16 is shown. A logarithmic scale is used.

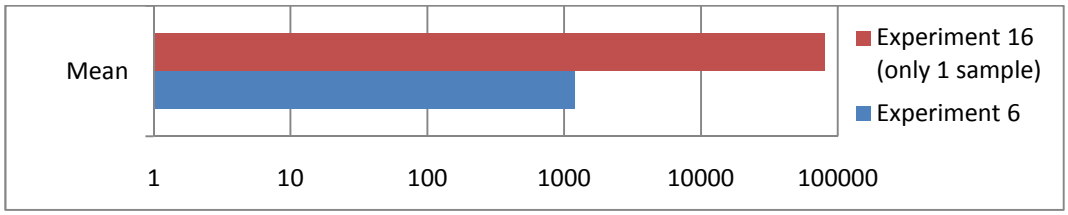



**Figure 25: Comparison graph between experiment 6 and 16**

## Experiment 17

In this experiment the same conditions as the previous test are used, with a reduced CA size of 65 cells. The goal is to achieve the same final state obtained with the computation of the rule 30 for a uniform CA (showing a pseudo-random behavior), after 64 evolution steps.

CA characteristics:

| | |
|---|---|
| Automaton type | 1 dimension non-uniform CA |
| Number of cells | 65 |
| Number of evolution steps | 64 |
| Fixed initial state | 00000000000000000000000000000001000000000000000000000000000000000 |
| Desired final state | 00100110010100000010100000010100101011111001011101101001110101100 |

Results after 10 simulations:

| | |
|---|---|
| Mean number of crossovers | 4.655,800 |
| Variance | 365.9571,560 |
| Standard deviation | 1.913,001 |
| Minimum number of crossovers | 1.949,000 |
| Maximum number of crossovers | 7.306,000 |
| Computation time | ~ 71 seconds |

In the following table the obtained rule-sets are shown.

| |
|---|
| 6 0 11 10 6 11 9 2 3 9 6 5 8 5 4 3 7 10 3 8 3 4 4 5 8 1 0 8 9 3 10 6 5 11 5 0 9 11 10 11 2 9 7 7 10 4 5 7 10 5 8 8 10 2 6 9 2 2 9 2 9 5 5 8 9 |
| 1 3 5 10 8 4 2 9 8 3 0 4 11 10 6 10 6 11 5 9 5 6 2 6 0 5 1 2 9 5 9 6 11 8 3 9 3 3 10 11 1 4 3 10 0 4 10 11 10 5 5 6 1 5 0 10 11 5 10 1 9 3 2 9 0 |
| 1 1 10 2 3 5 4 11 10 1 9 10 0 5 0 2 3 10 3 9 10 7 8 7 3 0 3 5 9 11 0 1 4 8 3 0 11 3 10 11 9 0 0 11 8 9 1 6 6 3 10 10 5 4 0 10 9 8 10 2 9 3 6 8 0 |
| 1 1 9 7 9 10 9 10 6 6 1 9 0 7 6 5 8 0 11 7 10 7 5 8 1 10 8 4 11 1 9 7 5 9 7 7 2 11 9 10 11 2 6 7 2 10 9 6 8 2 5 10 2 9 7 11 9 2 10 3 10 8 8 6 4 |
| 6 1 4 10 8 4 2 9 11 5 11 1 2 2 3 0 7 9 1 9 9 5 7 4 3 5 1 9 8 9 0 8 6 7 10 0 5 2 5 4 11 4 0 4 9 10 11 9 7 7 4 11 3 7 9 10 7 4 9 1 2 8 2 9 2 |
| 2 8 2 7 9 3 1 11 10 3 11 5 0 3 7 9 6 9 11 0 4 9 7 0 4 3 0 5 9 11 7 1 5 6 9 10 5 5 1 1 10 2 6 9 10 1 5 4 9 1 6 10 10 8 10 3 10 10 0 8 9 4 11 2 1 |
| 6 0 10 0 10 8 6 7 3 10 0 10 2 0 1 3 6 1 11 7 10 4 8 7 4 5 6 11 11 3 2 1 8 11 3 0 11 10 9 9 6 7 9 9 5 11 10 9 8 6 11 10 1 7 3 11 1 5 6 3 8 8 5 0 6 |
| 1 6 11 2 1 7 4 9 6 4 9 2 9 0 7 7 6 0 4 10 8 7 1 1 0 4 1 8 9 5 2 1 7 10 2 9 3 7 2 5 5 9 9 8 8 5 4 2 9 1 6 5 8 10 6 5 4 8 8 2 9 3 10 2 4 |
| 4 1 11 7 6 7 10 10 6 5 6 8 0 6 3 4 7 3 8 11 8 4 6 7 4 3 1 4 11 2 9 11 7 0 11 0 9 9 2 5 10 0 5 8 8 10 11 9 1 10 5 11 5 0 0 9 5 4 11 5 2 10 10 7 2 |
| 0 1 9 10 6 1 5 7 10 4 11 10 2 6 1 5 2 0 10 10 1 0 5 2 5 5 1 2 11 8 7 0 7 1 5 0 9 3 3 9 10 7 3 11 0 9 8 8 0 9 5 8 3 7 10 5 3 11 11 3 11 11 1 8 1 |

Table 6: Experiment 17 - obtained rule-sets

**Analysis:**

With a reduced CA size, it is possible to run more simulation in a reasonable time. After 10 simulations the mean number of GA cycles is 4.655,8 with a standard deviation of 1.913,001. This is a small number compared to the rules state-space of size 12 ^ 65. Over 10 runs, 10 different rule-sets are found. This means that for a non-uniform CA there are several rules that can reach the same final state over different trajectories.

In figure 26 a comparison of experiments 5 and 17 is shown.

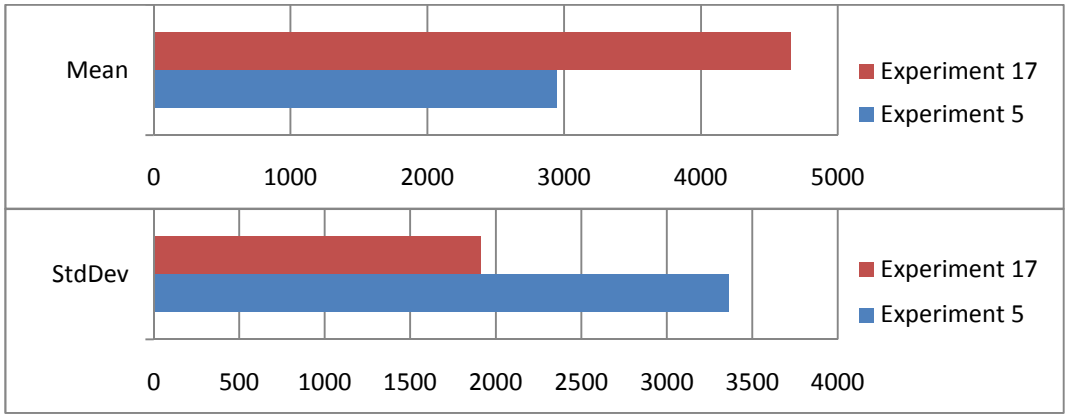



Figure 26: Comparison graph between experiment 5 and 17

## Experiment 18

In this experiment the CA type, size and number of evolution steps are the same as the previous experiment but the initial and final states are randomly generated, in order to verify the robustness of the Genetic Algorithm, even with random configuration parameters.

CA characteristics:

| | |
|---|---|
| Automaton type | 1 dimension non-uniform CA |
| Number of cells | 65 |
| Number of evolution steps | 64 |
| Fixed initial state | 01110110110111000110100000110100101101111001010101101000110111001 |
| Desired final state | 11101100010010110010010010010000010001001001000100100100101000001 |

Results after 10 simulations:

| | |
|---|---|
| Mean number of crossovers | 8.607,100 |
| Variance | 45.179.728,490 |
| Standard deviation | 6.721,587 |
| Minimum number of crossovers | 2.209,000 |
| Maximum number of crossovers | 23.433,000 |
| Computation time | ~ 135 seconds |

In the following table the obtained rule-sets are shown.

| |
|---|
| 3 3 3 6 1 8 1 6 2 10 6 10 4 10 10 8 7 6 6 0 8 1 4 1 9 0 8 10 2 11 8 0 1 8 9 6 11 3 11 8 10 0 0 3 11 10 7 11 11 8 7 5 11 10 2 1 0 7 5 11 6 8 2 7 11 |
| 4 1 0 9 3 11 7 6 1 9 7 6 3 2 2 8 8 8 2 9 8 3 10 6 8 1 1 0 2 11 5 0 1 7 7 6 0 11 9 11 11 0 0 10 7 7 6 11 0 10 0 11 8 3 7 3 9 9 5 11 10 9 7 3 7 |
| 6 8 3 9 11 6 1 0 1 7 1 7 4 8 11 3 9 11 0 0 5 1 0 3 10 0 8 10 6 6 0 1 1 9 10 9 9 6 1 6 3 0 1 7 0 8 0 7 8 2 7 2 11 10 0 1 0 2 10 6 8 8 0 6 6 |
| 4 0 3 9 4 7 7 7 6 11 10 7 0 11 7 10 5 8 3 0 5 5 1 0 11 7 7 0 7 6 2 7 6 3 8 9 11 4 11 8 4 11 10 1 7 6 6 3 2 7 8 8 2 2 7 2 6 5 9 1 11 6 4 0 0 |
| 10 0 4 8 11 5 6 8 9 2 6 7 3 7 9 2 11 7 7 0 7 2 9 8 11 0 8 10 5 8 5 7 6 3 7 11 2 9 10 11 4 4 3 11 0 1 0 11 0 7 0 0 1 3 8 2 9 1 3 8 1 6 7 9 3 |
| 1 6 6 9 2 5 8 7 7 7 0 7 4 11 11 2 9 2 7 1 11 11 7 3 4 11 10 3 7 6 1 2 7 10 5 7 6 4 9 8 0 7 1 10 5 7 6 5 8 11 5 0 1 6 0 3 5 2 2 1 2 2 2 3 9 |
| 4 2 4 6 2 4 6 6 11 5 6 7 4 8 5 1 8 2 7 1 11 4 10 6 1 0 7 0 10 10 7 2 11 5 6 7 6 8 8 7 0 2 1 10 0 10 10 2 7 10 0 7 6 2 9 0 6 6 3 8 6 11 2 8 2 |
| 6 8 0 9 3 0 7 8 10 0 11 7 3 6 1 11 0 8 10 5 8 3 6 7 10 0 10 3 2 8 8 0 1 9 8 9 8 0 6 7 7 7 1 10 1 8 7 1 0 1 0 0 1 4 9 0 5 7 2 11 10 9 9 6 3 |
| 11 11 11 1 2 3 9 8 8 4 7 7 3 6 2 2 7 10 4 0 5 8 0 8 11 7 8 7 2 11 5 8 5 6 8 7 7 2 8 8 9 2 3 0 7 6 5 10 6 10 0 7 6 10 2 1 10 7 7 8 7 6 1 7 2 |
| 9 7 1 1 1 3 11 5 8 0 2 11 4 6 7 4 7 10 0 10 2 1 1 8 5 11 7 5 2 2 0 6 1 6 11 5 0 11 11 7 11 0 0 0 2 2 0 4 7 6 2 11 7 2 9 11 4 11 5 11 9 11 7 11 3 |

**Table 7: Experiment 18 - obtained rule-sets**

**Analysis:**

This result is consistent with the results of the previous three experiments. In fact, the execution time is greater than in the previous test because the initial state is randomly generated. This instance is supposed to be harder to compute. The minimum and maximum values of GA cycles are respectively 2.209 and 23.433, with an average of 8.607,1. Moreover, in this experiment, 10 different rule-sets are found.

In figure 27 a comparison of experiments 17 and 18 is shown.

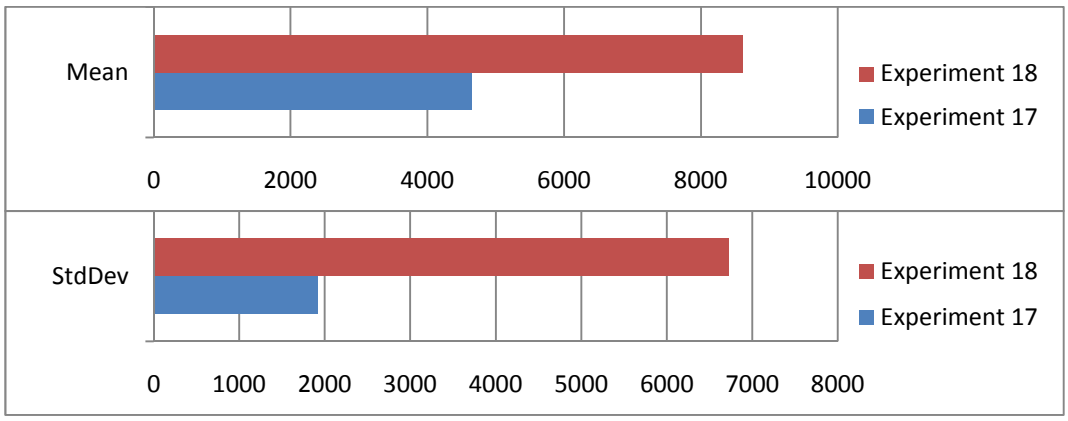



**Figure 27: Comparison graph between experiment 17 and 18**

## Experiment 19

In this experiment an intermediate state is introduced in the middle of the computation at the step 32. The goal is to find a rule-set for the non-uniform CA that can compute a trajectory from a default initial state to a randomly generated final state, through a randomly generated intermediate state.

CA characteristics:

| | |
|---|---|
| Automaton type | 1 dimension non-uniform CA |
| Number of cells | 65 |
| Number of evolution steps | 64 |
| Fixed initial state | 00000000000000000000000000000001000000000000000000000000000000000 |
| Random intermediate state (at evolution step 32) | 10101110101000111010000100110110000011010001110001001000111110110 |
| Random final state | 01011000110110101011000111000000111010010010011010101110111001011 |

Results:

This experiment has partially failed. Anyway, the outcome can be analyzed in order to understand the difficulty of the problem and the trustworthiness of the Genetic Algorithm with a specified intermediate state in the middle of the trajectory.

After 1.000.000 evolution cycles the obtained fitness function is 64/65 (around 98%). This means that both final and intermediate states are almost matching 100% (63/65 for the intermediate and 65/65 for the final or 64/65 for the intermediate and 64/65 for the final or 65/65 for the intermediate and 63/65 for the final).

**Analysis:**

The computation has been stopped because only few computational resources were available. After 1.000.000 evolution cycles the obtained fitness function is around 98%. It can be useful to reduce the size of the CA in order to partially solve the problem and retrieve some concrete results. This is explained in the next experiment's results.

## Experiment 20

In this experiment a 1-dimensional non-uniform CA with 17 cells is used. The goal is the same as the previous test. Specifically the objective is to find a rule-set for the non-uniform CA that can compute a trajectory from a default initial state to a randomly generated final state, through a randomly generated intermediate state. With a reduced CA dimension, the state-space becomes of size 12^17 (~ 2,22 x 10^18).

CA characteristics:

| | |
|---|---|
| Automaton type | 1 dimension non-uniform CA |
| Number of cells | 17 |
| Number of evolution steps | 64 |
| Fixed initial state | 00000000100000000 |
| Random intermediate state (at evolution step 33) | 01101111001101001 |
| Random final state | 00100110010100000 |

Results after 5 simulations:

| | |
|---|---|
| Mean number of crossovers | 786.380,800 |
| Variance | 82.914.725.349,000 |
| Standard deviation | 287.949,172 |
| Minimum number of crossovers | 510.066,000 |
| Maximum number of crossovers | 1.331.125,000 |
| Computation time | ~ 1.096 seconds |

In the following table the number of evolution cycles for each simulation is given, with also the obtained rule-set.

| Evolution cycles | Obtained rule-sets |
|---|---|
| 510.066 | 0 8 10 7 10 5 5 6 7 9 10 4 11 8 6 8 6 |
| 1.331.125 | 0 10 8 0 7 4 4 9 7 9 10 3 11 8 6 8 6 |
| 777.712 | 11 10 1 0 10 3 2 9 7 8 10 4 11 8 6 8 6 |
| 594.595 | 7 2 10 7 10 3 8 1 7 10 10 1 7 10 8 8 10 |
| 718.406 | 7 10 3 7 7 2 5 9 7 9 1 3 11 8 10 10 8 |

Table 8: Experiment 20 - obtained rule-sets

**Analysis:**

Considering the computational difficulties to find a specified trajectory with an intermediate state in a CA of size 65, the problem has been simplified using non-uniform CA of size 17. The resultant state-space has size 12^17 (around 2,22 x 10^18). It is still a huge value but it was possible to collect data from 5 simulations (execution time around 1096 seconds). This is a really encouraging result, considering that the mean number of GA cycles is 786.380,8.

In Figure 28 an example of a correct solution is shown:

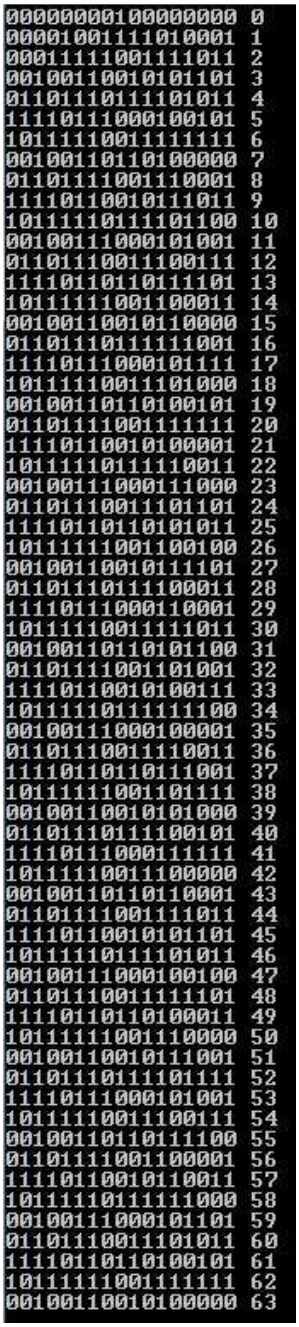



Figure 28: 1D non-uniform CA, size 17, experiment 20

## Experiment 21

In this experiment a 1D non-uniform CA of size 129 is used. Starting from an initial state with only a 1 in the middle, the goal is to reach an intermediate state and a final state that are symmetric. The intermediate and final states can be produced by the rule 90 for a uniform CA. With this experiment, it is investigated if it is possible, for a non-uniform CA, to start from a situation of symmetry, break it and then reach again a symmetric state (intermediate state), then break the symmetry again and reach a symmetric final state in the end of the computation.

CA characteristics:

| | |
|---|---|
| Automaton type | 1 dimension non-uniform CA |
| Number of cells | 129 |
| Number of evolution steps | 64 |
| Fixed initial state | 00000000000000000000000000000000000000000000000<br>00000000000000000000010000000000000000000000000<br>00000000000000000000000000000000000000000000000 |
| Intermediate state<br>(at evolution step 32) | 00000000000000000000000000000000001010101010<br>10101010101010101010101010101010101010101010101<br>01010101010000000000000000000000000000000000 |
| Final state | 01010101010101010101010101010101010101010101010<br>10101010101010101010101010101010101010101010101<br>01010101010101010101010101010101010101010101010 |

Results (only 1 simulation):

| | |
|---|---|
| Number of crossovers | 255.5133 |
| Computation time | ~ 8.891 seconds |

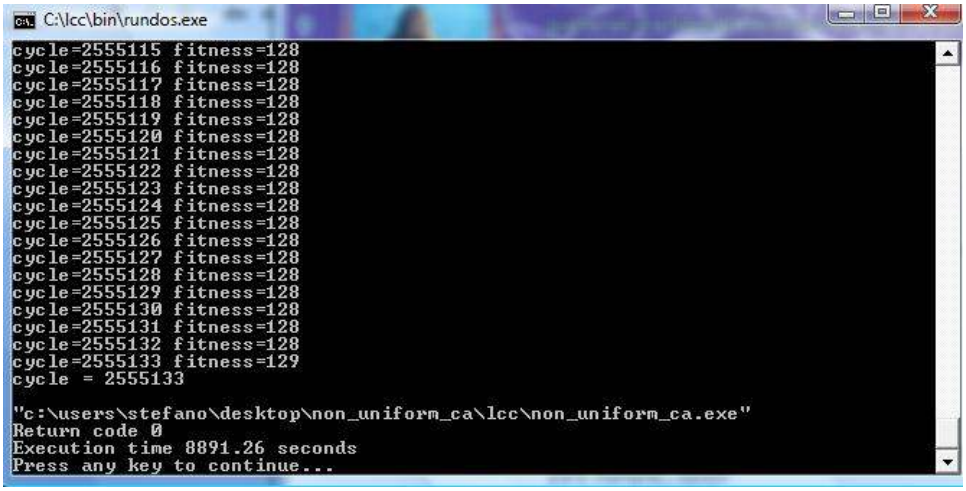



Figure 29: Execution of experiment 21

In Figure 29 the end of the computation of the experiment is shown, with the execution time, number of GA iterations and fitness reached.

Obtained rule-set:

6 8 11 5 0 5 11 2 11 2 11 5 8 2 8 2 11 8 0 5 11 5 11 2 11 1 11 8 8 8 11 1 11 3 0 11 0 9 11 11 0 4 8 10 0 9 11 3 11 11 8 9 11 3 8 3 11 10 0 11 11 4 11 4 8 11 8 10 8 11 11 11 0 11 0 9 0 9 11 3 0 10 0 11 8 3 11 3 0 11 11 10 8 11 11 3 11 5 0 2 11 8 0 5 8 2 8 2 11 8 8 5 11 5 11 5 8 8 8 5 0 5 11 1 8 2 11 5 4

**Analysis:**

This result shows that it is possible, for a non-uniform CA, to find a rule that breaks the symmetry more than once in the same trajectory and then reconstruct it. In fact, starting from a symmetric configuration, the computation reaches another symmetric state (intermediate state) after several not symmetric states. After that, again the symmetry is broken and reached once more in the final state. As a single computation takes more than 2 hours, this experiment has been executed only once. It is enough to prove that the hypothesis is confirmed.

## Experiment 22

In this experiment the set-up is the same as the previous test. The only difference is that the intermediate state can be found in a range and not in a specific evolution step. In fact, some flexibility in the trajectory is added. Starting from an initial state with only a 1 in the middle, the goal is to reach an intermediate state (between evolution step 24 and 40) and a final state that are symmetric. The intermediate and final states can be produced by the rule 90 for a uniform CA. With this experiment, it is investigated if it is possible, for a non-uniform CA, to start from a situation of symmetry, break it and then reach again a symmetric state (intermediate state), then break the symmetry again and reach a symmetric final state in the end of the computation.

CA characteristics:

| | |
|---|---|
| Automaton type | 1 dimension non-uniform CA |
| Number of cells | 129 |
| Number of evolution steps | 64 |
| Fixed initial state | 00000000000000000000000000000000000000000000000<br>00000000000000000000010000000000000000000000<br>00000000000000000000000000000000000000000000000 |
| Intermediate state<br>(between step 24 and 40) | 00000000000000000000000000000000001010101010<br>10101010101010101010101010101010101010101010101<br>01010101010000000000000000000000000000000000 |

Final state

01010101010101010101010101010101010101010
10101010101010101010101010101010101010101
01010101010101010101010101010101010101010

Results:

The experiment has partially failed because after 5.000.000 iterations of the GA, the reached fitness value is 116/129. This means that the reached intermediate state and the reached final state have an average of 116 of the 129 cells that are correct.

**Analysis:**

Even if this experiment has partially failed, the outcomes are useful to make some considerations and analyze the performances of the Genetic Algorithm with a specified intermediate state inside a range. The obtained fitness function is around 90% (116/129). As it is calculated giving a weight of 50% at the fitness of the final state and 50% at the fitness of the intermediate state, this means that each of the fitness values is between 103/129 and 129/129. The introduction of an interval for the intermediate state (between evolution step 24 and 40 of the CA) increases the difficulty of the search problem (even if there are more rules fulfilling the requirements) because a mutation of a rule is reflected on a different fitness value for each of the states inside the interval and not only on a specific state. This can cause possible loops in local maximum solutions and reduce the performances and the speed of the GA. As future work, it can be useful to reduce the size of the CA in order to investigate the problem and understand how to modify the GA to retrieve some concrete results.

## Experiment 23

In this experiment another intermediate state is added. In order to reduce the difficulty of the problem, a non-uniform CA of size 17 is used. The goal is to find a rule that can reach during the computation two intermediate states at some specified points in time. The initial state, intermediate states and final states are calculated using the rule 90 on a uniform CA.

CA characteristics:

| | |
|---|---|
| Automaton type | 1 dimension non-uniform CA |
| Number of cells | 17 |
| Number of evolution steps | 64 |
| Fixed initial state | 00000000100000000 |
| First intermediate state (at evolution step 15) | 01100110001100110 |
| Second intermediate state (at evolution step 45) | 01100110001100110 |
| Desired final state | 00000101010100000 |

Results after 100 simulations:

| | |
|---|---|
| Mean number of crossovers | 2.548,410 |
| Variance | 6.454.467,000 |
| Standard deviation | 2.540,564 |
| Minimum number of crossovers | 479,000 |
| Maximum number of crossovers | 23.621,000 |
| Computation time | ~ 72 seconds |

In the following table some obtained rule-sets are shown:

| | |
|---|---|
| 3 10 6 7 0 8 9 8 7 10 10 1 4 6 7 9 0 | 0 10 6 0 0 9 6 10 6 11 1 11 0 1 7 9 7 |
| 0 10 6 4 1 2 9 9 6 9 7 11 0 6 10 6 4 | 0 10 6 7 6 8 2 10 6 2 7 11 2 3 10 6 0 |
| 0 7 9 2 3 8 9 10 6 10 10 1 4 0 10 6 7 | 0 10 6 0 3 5 9 10 6 11 1 5 4 3 10 6 7 |
| 3 10 6 4 6 2 9 10 6 9 1 10 2 6 7 9 7 | 0 10 6 7 1 2 9 8 6 2 10 3 4 3 10 6 7 |
| 3 7 9 4 1 11 2 10 6 9 10 8 4 1 10 6 4 | 6 10 6 0 0 5 9 11 6 9 10 5 0 0 10 6 2 |
| 1 10 6 7 3 2 2 10 7 9 10 1 4 0 7 9 2 | 6 10 6 7 6 11 6 10 6 11 10 5 4 6 10 6 7 |

Table 9: Experiment 23 - obtained rule-sets

**Analysis:**

With a reduced size of the CA it is possible to find rule-sets that, starting from a symmetric state, can break the symmetry and then reach a symmetric state again twice, before reaching another symmetric final state. This is done introducing two intermediate states at specific evolution steps of the CA. After 100 simulations the mean number of iterations of the GA is 2.548,41. This is a very low value compared to the search space of size 12^17. The minimum and maximum number of iterations are respectively 479 and 23.621, with a standard deviation of 2.540,564. In each of the 100 iterations a different rule-set has been found. The result is remarkable because the introduction of two intermediate states at specific points in the trajectory increases the speed of the exploration of the state-space performed by the GA.

In Figure 30 a comparison of experiments 20 (with only one intermediate step) and 23 is shown. A logarithmic scale is used.

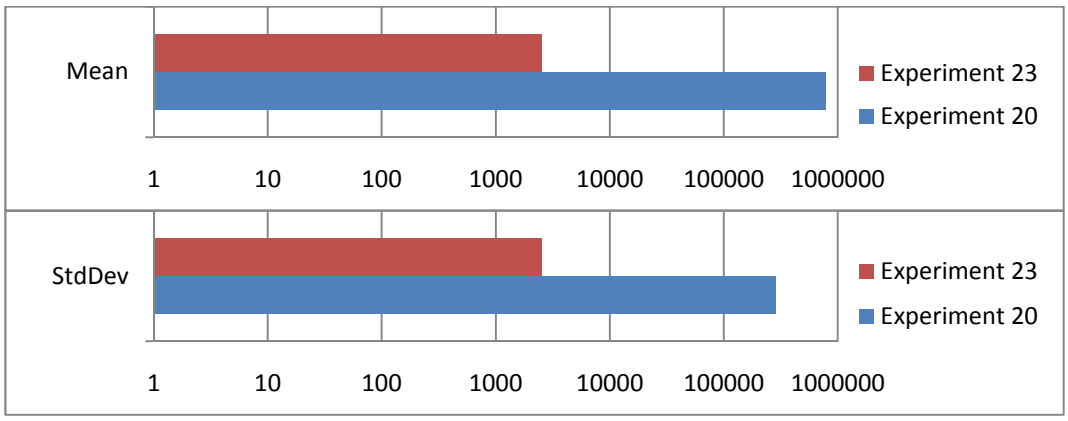



**Figure 30: Comparison graph between experiment 20 and 23**

## Experiment 24

In this experiment the goal is to find the rule that can compute in a single CA evolution step a trajectory from a random state to another randomly generated state.

CA characteristics:

| | |
|---|---|
| Automaton type | 1 dimension non-uniform CA |
| Number of cells | 129 |
| Number of evolution steps | 1 |
| Random initial state | 11001110111111001010110011111011000100011100010001101111000110111011110011011011011000100010100101101000100010100101000010100110 1 |
| Random final state (after only 1 evolution step) | 11011111101001011100011110001111000100001001101100110111101110111011010010101100110100000101110110000100011011001001111110111010 |

Results after 100 simulations:

| | |
|---|---|
| Mean number of crossovers | 11.474,750 |
| Variance | 23.770.640,000 |
| Standard deviation | 4.875,514 |
| Minimum number of crossovers | 8.357,000 |
| Maximum number of crossovers | 31.039,000 |
| Computation time | ~ 106 seconds |

In the following table some obtained rule-sets are shown:

| |
|---|
| 3 1 0 8 6 5 3 5 2 10 2 10 10 5 2 11 10 7 8 11 7 3 10 11 8 11 8 6 9 11 4 11 3 0 2 9 8 6 1 9 10 2 3 10 9 2 9 4 7 9 10 4 11 4 1 3 9 3 0 10 3 10 4 3 4 2 7 1 4 9 8 1 7 5 0 2 9 6 8 6 0 9 5 0 10 0 5 0 4 0 9 8 11 11 10 5 9 8 9 8 |
| 2 10 0 5 11 3 4 10 3 10 2 11 8 4 2 2 3 6 8 8 10 3 10 11 0 6 7 9 9 2 0 8 0 3 5 10 8 0 10 9 5 0 7 8 6 7 9 9 1 2 10 5 7 2 7 10 10 6 6 0 1 7 4 0 5 7 7 0 1 9 8 3 10 11 11 6 9 2 6 2 11 11 10 4 11 0 8 2 5 0 3 8 10 9 11 5 6 6 6 |
| 0 1 4 9 2 4 1 4 6 6 0 11 10 3 4 8 3 2 2 11 10 1 10 2 5 9 10 9 4 9 1 10 7 6 5 8 0 3 1 7 0 2 2 2 9 0 10 9 7 2 4 6 11 1 5 1 11 3 2 10 9 10 5 2 0 3 10 1 2 9 1 3 1 7 8 3 6 7 8 9 11 6 10 4 11 6 2 0 4 3 3 8 11 8 4 8 2 11 6 0 5 4 |

Table 10: Experiment 24 - obtained rule-sets

**Analysis:**

This experiment shows that it is possible, for a non-uniform CA, to reach in only 1 evolution step a randomly generated final state starting from another randomly generated initial state. After 100 simulations the mean number of iterations of the GA is 11.474,75 with a standard deviation of 4.875,514, a minimum of 8.357 and a maximum of 31.039. Even if it is not mathematically proven, it is reasonable to say that experimentally it is possible, for a non-uniform CA with a reduced number of rules, to reach every possible state from a randomly generated initial state, in a single CA evolution step. This is achievable because every cell of the next state depends only on the local state of the neighbors and the value of the cell itself. It is possible to find a rule for each combination of the neighborhood to generate a specific next value of every cell.

# CONCLUSION AND FUTURE WORK

In this research the goal was to exploit the behavior of life-like computing systems such as Cellular Automata, developed together with Evolutionary strategies such as Genetic Algorithms. This work, along with other research on artificial life and artificial intelligence, demonstrates that it is possible to use some of the principles of life, evolution and adaptation in machines.

The following results have been achieved:

- The use of Genetic Algorithms to find CA rules that can follow a specified trajectory can be a remarkable approach, as the search-space is beyond the current computational resources and exhaustive research techniques cannot be adopted;

- The applied methodology can guarantee positive results with uniform cellular automata, even with complex trajectories and with non uniform cellular automata (with specified initial and final state);

- Further investigation is required for non uniform cellular automata with complex trajectories (specifying intermediate states and time intervals);

- It is possible to graphically visualize the trajectories and attractor basins of cellular automata using techniques and tools for the analysis of Random Boolean Networks. This is helpful when the behavior of CA is complex. In this research, all the plots of the trajectories have been done by PAJEK, using the Fruchterman Reingold 3d option [47].

Possible future researches include:

- Analysis of 2-dimensional cellular automata;

- In depth analysis of GA to find rules for non-uniform CA;

- Modification and further tuning of the Genetic Algorithm;

- Scaling the problem, investigating CAs of bigger size.

- Implementation of the CAs in hardware.

# BIBLIOGRAPHY


1. **Haddow, Pauline C.** CRAB Lab. *(Complex, Reliable, Adaptive, Bio-inspired hardware).* [Online] [Cited: June 3, 2009.] http://crab.idi.ntnu.no/.

2. **Tufte, Gunnar and Haddow, Pauline C.** *Towards development on a silicon-based cellular computing machine.* Natural Computing. 2005, Vol. 4, 4 s. 387-416.

3. **Tufte, Gunnar.** *The discrete dynamics of developmental systems.* Los Alamitos : IEEE Computer Society, 2009, IEEE International Conference on E-Commerce Technology, pp. 2209-2216.

4. **Tufte, Gunnar.** *Phenotypic, developmental and computational resources: scaling in artificial development.* Atlanta : ACM, 2008. GECCO '08: Proceedings of the 10th annual conference on Genetic and evolutionary computation.

5. **Mitchell, Melanie.** *Life and Evolution in Computers.* Santa Fe : M. McPeek et al., 2000, Vol. Darwinian Evolution Across the Disciplines.

6. **Darwin, Charles.** *On the Origin of Species by Means of Natural Selection, or the Preservation of Favoured Races in the Struggle for Life.* John Murray, 1859.

7. **Sipper, Moshe.** *Machine Nature.* New York : McGraw-Hill, 2002.

8. **Sipper, Moshe.** *Evolution of Parallel Cellular Machines.* Heidelberg : Springer-Verlag, 1997.

9. **Mayr, Ernst W.** *What Evolution Is.* New York : Basic Books, 2001. ISBN 0-465-04426-3.

10. **Sipper, Moshe.** *The Emergence of Cellular Computing.* 7, Lousanne : IEEE Computer Society, 1999, Computer, Vol. 32, pp. 18-26.

11. **Abelson, Harold et al.** *Amorphous Computing.* Boston : AI Memo 1665, 1999.

12. **Los Alamos National Laboratory.** *Stanislaw Ulam 1909-1984*. Los Alamos: Los Alamos Science, 1987. Vol. 15, pp. 1-318, special issue.

13. **Von Neumann, John.** *Theory and Organization of complicated automata.* A. W. Burks, 1949, pp. 29-87 [2, part one]. Based on transcript of lectures delivered at the University of Illinois in December 1949.

14. **Hanson, James E and Crutchfield, James P.** *The Attractor-Basin Portrait of a Cellular Automaton.* Berkeley : Journal of Statistical Physics, 1992, Vol. 66 p.1415-1462.

15. **McMullin, Barry.** *John von Neumann and the Evolutionary Growth of Complexity: Looking Backward, Looking Forward...* Artificial Life. 2000, Vol. 6, 347-361.

16. **Wolfram, Stephen.** *University and Complexity in Cellular Automata.* Addison-Wesley, 1994, pp. 115-157.

17. **Wolfram Research Inc.** *Elementary Cellular Automaton.* Wolfram Math World. [Online] [Cited: March 17, 2009.] http://mathworld.wolfram.com/ElementaryCellularAutomaton.html.

18. **Bidlo, Michael and Vasicek, Zdenek.** *Gate-Level Evolutionary Development Using Cellular Automata.* Los Alamitos : IEEE Computer Society, 2008. NASA/ESA Conference on Adaptive Hardware and Systems. pp. 11-18.

19. **Laing, Richard.** *Artificial Organisms and Autonomous-Cell Rules.* Ann Arbor : Office of Research Administration, University of Michigan, 1972.

20. **Laing, Richard and Arbib, Michael.** *II Morphogenesis of Simple Artificial Organisms.* Automata Theory and Development. 1967.

21. **Mitchell, Melanie, Crutchfield, James P and Das, Jararshi.** *Evolving Cellular Automata with Genetic Algorithms: A Review of Recent Work.* Russian Academy of Sciences, 1996. In Proceedings of the First International Conference on Evolutionary Computation and Its Applications (EvCA'96).

22. **Mitchell, Melanie, Crutchfield, James P and Hraber, Peter T.** *Evolving Cellular Automata to Perform Computations: Mechanisms and Impediments.* Physica D. 1994, Vol. 75, pp. 361-391.

23. **Buckland, Mat.** *The Genetic Algorithm.* ai-junkie. [Online] [Cited: March 17, 2009.] http://www.ai-junkie.com/.

24. **Back, Thomas.** *Optimal Mutation Rates in Genetic Search.* Morgan Kaufmann, 1993. Proceedings of the fifth International Conference on Genetic Algorithms. pp. 2-8.

25. **Adamopoulos, A V, Pavlidis, N G and Vrahatis, M N.** *Genetic Algorithm Evolution of Cellular Automata Rules for Complex Binary Sequence Prediction.* Leiden : Lecture Series on Computer and Computational Science, 2005, Vols. 1 pp. 1-6.

26. **Muhlenbein, Heinz.** *How Genetic algorithms really work. I. Mutation and Hillclimbing.* Parallel Problem Solving from Nature, 1992, Vol. 2 pp. 15-29.

27. **Shimodaira, Hisashi.** *A New Genetic Algorithm Using Large Mutation Rates and Population-Elitist Selection (GALME).* Washington DC : IEEE Computer Society, 1996. ICTAI '96: Proceedings of the 8th International Conference on Tools with Artificial Intelligence.

28. **Stanhope, Stephen A and Daida, Jason M.** *Optimal Mutation and Crossover Rates for a Genetic Algorithm Operating in a Dynamic Environment.* Berlin : Springer Berlin / Heidelberg, 1998, Evolutionary Programming VII - Lecture Notes in Computer Science, Vol. 1447/1998, pp. 693-702.

29. **Wikimedia Foundation.** *Fitness proportionate selection.* Wikipedia. [Online] [Cited: March 17, 2009.] http://en.wikipedia.org/wiki/Fitness_proportionate_selection.

30. **Wikimedia Foundation.** *Tournament selection.* Wikipedia. [Online] [Cited: March 17, 2009.] http://en.wikipedia.org/wiki/Tournament_selection.

31. **Wuensche, Andrew.** *Tools for Investigating Cellular Automata and Discrete Dynamical Networks.* Artificial Life Models in Software. London : Springer, 2009.

32. **Kauffman, Stuart.** *Metabolic stability and epigenesis in randomly constructed genetic nets.* 1969, Journal of Theoretical Biology, Vol. 22, pp. 437-467.

33. **Gershenson, Carlos.** *Introduction to Random Boolean Networks.* Workshop and Tutorial Proceedings, Ninth International Conference on the Simulation and Synthesis of Living Systems (ALife IX), 2004, pp. 160-173.

34. **Burks, A. W.** *von Neumann's self-reproducing automata.* In [3, Essay One, pp. 3-64]. 1970.

35. **Von Neumann, John.** *Theory of self-reproducing automata,* edited and completed by A.W. Burks. University of Illinois Press. 1966.

36. **Sipper, Moshe and Tomassini, Marco.** *Generating Parallel Random Number Generators By Cellular Programming.* 1996, International Journal of Modern Physics C, Vol. 7, pp. 181-190.

37. **Mainzer, Klaus.** *Thinking in Complexity - Chapter 5: Complex Systems and the Evolution of Computability.* Springer, 2002.

38. **Benjamin, Simon C. and Johnson, Neil F.** *A Possible Nanometer-scale Computing Device Based on an Adding Cellular Automaton.* Applied Physics Letters. 1997, Vol. 70, 17 pp.2321-2323.

39. **Adleman, L. M.** *Molecular Computation of Solutions to Combinatorial Problems.* Science, pp. 1921-1924. 1994.

40. **Lipton, R. J.** *DNA Solution of Hard Computational Problems.* Science, pp. 542-545. 1995.

41. **Chou, H. H. and Reggia, J. A.** *Problem Solving During Artificial Selection of Self-Replicationg Loops.* Phisica D, pp. 293-312. 1998.

42. **Wuensche, Andy.** *Discrete Dynamics Lab.* DDLAB. [Online] [Cited: June 3, 2009.] http://www.ddlab.com/.

43. **Costa Santini, Cristina, Tufte, Gunnar and Haddow, Pauline.** *Bio-inspired Reverse Engineering of Regulatory Networks.* Los Alamitos : IEEE Congress on Evolutionary Computation, 2009, pp. 2716-2723, cec.

44. **Navia, Jacob.** *lcc-win32: A Compiler system for windows.* lcc-win32. [Online] [Cited: July 1, 2009.] http://www.cs.virginia.edu/~lcc-win32/.

45. **Wikimedia Foundation.** *Elementary Cellular Automata.* Wikimedia Commons. [Online] [Cited: March 17, 2009.] http://commons.wikimedia.org/wiki/Elementary_cellular_automata.

46. **Sedgewick, Robert.** *Algorithms. 2nd edition.* Addison-Wesley, 1988.

47. **Batagelj, Vladimir and Mrvar, Andrej.** *pajek [Pajek Wiki].* Pajek - Program for Large Network Analysis. [Online] 1997. [Cited: June 4, 2009.] http://pajek.imfm.si/.

# APPENDIX

## Appendix 1: 1D uniform CA engine

```
/* Nichele Stefano */

// include library

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <iostream.h>

// constant and object instantiation

const int size = 17;            //size of the CA
int automata[size];             //CA current state
int next[size];                 //CA next state
int loops = 64;                 //CA loop updates

struct rule                     //rule structure
{
        int left;
        int me;
        int right;
        int next;
};

struct rule rules[8];           //uniform CA, only 1 rule for all cells

//procedures

void init_rules()
{
        //example for rule 90: 01011010

        rules[0].left       = 0;
        rules[0].me         = 0;
        rules[0].right      = 0;
        rules[0].next       = 0;

        rules[1].left       = 0;
        rules[1].me         = 0;
        rules[1].right      = 1;
        rules[1].next       = 1;

        rules[2].left       = 0;
        rules[2].me         = 1;
        rules[2].right      = 0;
        rules[2].next       = 0;

        rules[3].left       = 0;
        rules[3].me         = 1;
        rules[3].right      = 1;
        rules[3].next       = 1;

        rules[4].left       = 1;
        rules[4].me         = 0;
        rules[4].right      = 0;
        rules[4].next       = 1;
```

```
	rules[5].left		= 1;
	rules[5].me		= 0;
	rules[5].right		= 1;
	rules[5].next		= 0;

	rules[6].left		= 1;
	rules[6].me		= 1;
	rules[6].right		= 0;
	rules[6].next		= 1;

	rules[7].left		= 1;
	rules[7].me		= 1;
	rules[7].right		= 1;
	rules[7].next		= 0;
}


void init_ca()
{
	int i;
	//	standard initialization 0000..1..0000
	for (i=0; i<size; i++)
	{
		automata[i] = 0;
		next[i] = 0;
	}
	automata[size/2] = 1;
}


int find_rule (int pos)
{
	int l,m,r,j;
	m = automata[pos];
	if(pos == 0)
	{
		r = automata[pos + 1];
		l = automata[size - 1];
	} else
	if(pos == size - 1)
	{
		r = automata[0];
		l = automata[pos - 1];
	} else
	{
		l = automata[pos - 1];
		r = automata[pos + 1];
	}
	for (j=0; j<8; j++)
	{
		if((rules[j].left == l) && (rules[j].me == m) && (rules[j].right == r))
			return rules[j].next;
	}
	// rule not found
	return -1;
}


void run()
{
	int i,j,new_state;
	printf("\n");
	for (j=0; j<size; j++)
		printf("%d",automata[j]);
	printf(" 0\n");
	for (i=1; i<loops; i++)
```

```
        {
                for (j=0; j<size; j++)
                {
                        new_state = find_rule(j);
                        if (new_state == -1)
                        {
                                printf("\n Error \n");
                                exit(-1);
                        }
                        next[j] = new_state;
                }
                for (j=0; j<size; j++)
                {
                        printf("%d",next[j]);
                        automata[j]=next[j];
                        next[j]=0;
                }
                printf(" %d\n",i);
        }
}

//main

int main(int argc,char *argv[])
{
        init_rules();
        init_ca();
        run();
        return 0;
}
```

## Appendix 2: 1D non-uniform CA engine

```
/* Nichele Stefano */

// include library

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <iostream.h>
#include <time.h>

// constant and object instantiation

const int size = 129;           //size of the ca
int automata[size];             //ca current state
int celltype[size];             //cell types
int next[size];                 //ca next state
int loops = 64;                 //ca loops

FILE *input_file;               // input ruleset

/*
L C R
  N

L = left neighbour
C = state of the cell itself
R = right neighbour
N = next state of the cell

cell types (rules to generate the next state)

0       Identity C
1       Identity L
2       Identity R
3       OR L,C
4       OR C,R
5       OR L,R
6       XOR L,C
7       XOR C,R
8       XOR L,R
9       NAND L,C
10      NAND C,R
11      NAND L,R
*/

// procedures

void init_rules()
{
        //init from file
        int i;
        for(i=0; i<size; i++)
        {
                fscanf(input_file, "%d", &celltype[i]);
                printf("%d \n",celltype[i]);
        }
}


void init_ca()
{
```

```
	// standard initialization 0000..1..0000
	int i;
	for (i=0; i<size; i++)
	{
		automata[i] = 0;
		next[i] = 0;
	}
	automata[size/2] = 1;
}

void run()
{
	int i,j,new_state,right,left,centre;
	printf("\n");
	for (j=0; j<size; j++)
		printf("%d",automata[j]);
	printf(" 0\n");
	for (i=1; i<loops; i++)
	{
		for (j=0; j<size; j++)
		{
			centre = j;
			if(centre == 0)
				left = size - 1;
			else
				left = centre - 1;
			if(centre == size - 1)
				right = right = 0;
			else
				right = centre + 1;
			switch(celltype[centre])
			{
				case 0 :
					next[centre] = automata[centre];
					break;
				case 1 :
					next[centre] = automata[left];
					break;
				case 2 :
					next[centre] = automata[right];
					break;
				case 3 :
					if((automata[left] == 0) && (automata[centre] == 0))
						next[centre] = 0;
					if((automata[left] == 0) && (automata[centre] == 1))
						next[centre] = 1;
					if((automata[left] == 1) && (automata[centre] == 0))
						next[centre] = 1;
					if((automata[left] == 1) && (automata[centre] == 1))
						next[centre] = 1;
					break;
				case 4 :
					if((automata[centre] == 0) && (automata[right] == 0))
						next[centre] = 0;
					if((automata[centre] == 0) && (automata[right] == 1))
						next[centre] = 1;
					if((automata[centre] == 1) && (automata[right] == 0))
						next[centre] = 1;
					if((automata[centre] == 1) && (automata[right] == 1))
						next[centre] = 1;
					break;
				case 5 :
					if((automata[left] == 0) && (automata[right] == 0))
						next[centre] = 0;
					if((automata[left] == 0) && (automata[right] == 1))
						next[centre] = 1;
					if((automata[left] == 1) && (automata[right] == 0))
```

```
                                        next[centre] = 1;
                                if((automata[left] == 1) && (automata[right] == 1))
                                        next[centre] = 1;
                                break;
                        case 6 :
                                if((automata[left] == 0) && (automata[centre] == 0))
                                        next[centre] = 0;
                                if((automata[left] == 0) && (automata[centre] == 1))
                                        next[centre] = 1;
                                if((automata[left] == 1) && (automata[centre] == 0))
                                        next[centre] = 1;
                                if((automata[left] == 1) && (automata[centre] == 1))
                                        next[centre] = 0;
                                break;
                        case 7 :
                                if((automata[centre] == 0) && (automata[right] == 0))
                                        next[centre] = 0;
                                if((automata[centre] == 0) && (automata[right] == 1))
                                        next[centre] = 1;
                                if((automata[centre] == 1) && (automata[right] == 0))
                                        next[centre] = 1;
                                if((automata[centre] == 1) && (automata[right] == 1))
                                        next[centre] = 0;
                                break;
                        case 8 :
                                if((automata[left] == 0) && (automata[right] == 0))
                                        next[centre] = 0;
                                if((automata[left] == 0) && (automata[right] == 1))
                                        next[centre] = 1;
                                if((automata[left] == 1) && (automata[right] == 0))
                                        next[centre] = 1;
                                if((automata[left] == 1) && (automata[right] == 1))
                                        next[centre] = 0;
                                break;
                        case 9 :
                                if((automata[left] == 0) && (automata[centre] == 0))
                                        next[centre] = 1;
                                if((automata[left] == 0) && (automata[centre] == 1))
                                        next[centre] = 1;
                                if((automata[left] == 1) && (automata[centre] == 0))
                                        next[centre] = 1;
                                if((automata[left] == 1) && (automata[centre] == 1))
                                        next[centre] = 0;
                                break;
                        case 10 :
                                if((automata[centre] == 0) && (automata[right] == 0))
                                        next[centre] = 1;
                                if((automata[centre] == 0) && (automata[right] == 1))
                                        next[centre] = 1;
                                if((automata[centre] == 1) && (automata[right] == 0))
                                        next[centre] = 1;
                                if((automata[centre] == 1) && (automata[right] == 1))
                                        next[centre] = 0;
                                break;
                        case 11 :
                                if((automata[left] == 0) && (automata[right] == 0))
                                        next[centre] = 1;
                                if((automata[left] == 0) && (automata[right] == 1))
                                        next[centre] = 1;
                                if((automata[left] == 1) && (automata[right] == 0))
                                        next[centre] = 1;
                                if((automata[left] == 1) && (automata[right] == 1))
                                        next[centre] = 0;
                                break;
                }
        }
```

```
                        for (j=0; j<size; j++)
                        {
                                printf("%d",next[j]);
                                automata[j]=next[j];
                                next[j]=0;
                        }
                        printf(" %d\n",i);
                }
}

// main

int main(int argc,char *argv[])
{
                input_file = fopen("input.txt", "r");
                if(input_file==NULL)
                {
                        printf("Error: can't open input file.\n");
                        return 1;
                }
                init_rules();
                init_ca();
                run();
                return 0;
}
```

## Appendix 3: GA for uniform CA

```
/* Nichele Stefano */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <iostream.h>
#include <time.h>

// constant and object instantiation

const int size = 65;              //size of the CA
const int child = 10;             //number of childs
int automata[child][size];        //ca current states
int next[size];                   //ca next state
int loops = 65;                   //ca loop updates
int attractor[size];              //attractor

int best1, best2;                 // best 2 rules index
int worst1, worst2;               // worst 2 rules index

int fit[child];                                   //array of fitness values, one for each rule
int rank[child] = {0,1,2,3,4,5,6,7,8,9};          // ranking of the fitness functions
float weight[child];                              // weight of the fitness function

int input[size];                  // input from configuration file
int output[size];                 // output from configuration file
int inter1[size];                 // intermediate state 1 from configuration file
int inter2[size];                 // intermediate state 2 from configuration file

const int position1 = 10; // range 1 first value
const int position2 = 20; // range 1 second value
const int position3 = 40; // range 2 first value
const int position4 = 50; // range 2 second value

FILE *input_file, *output_file;           // input and output files

struct rule
{
        int left;
        int me;
        int right;
        int next;
};

struct rulestruct
{
        struct rule rules[8];
};


struct rulestruct ruleset[child];         //uniform ca, only 1 rule for all cells

// procedures

void init_rules()
{
        int i,j;
        for(i=0; i<child; i++)
        {
                ruleset[i].rules[0].left        = 0;
                ruleset[i].rules[0].me          = 0;
                ruleset[i].rules[0].right       = 0;
                j = rand() % 100 + 1;
```

```
if(j>50)
        {ruleset[i].rules[0].next =0;}
else
        {ruleset[i].rules[0].next =1;}

ruleset[i].rules[1].left        = 0;
ruleset[i].rules[1].me          = 0;
ruleset[i].rules[1].right       = 1;
j = rand() % 100 + 1;
if(j>50)
        {ruleset[i].rules[1].next =0;}
else
        {ruleset[i].rules[1].next =1;}

ruleset[i].rules[2].left        = 0;
ruleset[i].rules[2].me          = 1;
ruleset[i].rules[2].right       = 0;
j = rand() % 100 + 1;
if(j>50)
        {ruleset[i].rules[2].next =0;}
else
        {ruleset[i].rules[2].next =1;}

ruleset[i].rules[3].left        = 0;
ruleset[i].rules[3].me          = 1;
ruleset[i].rules[3].right       = 1;
j = rand() % 100 + 1;
if(j>50)
        {ruleset[i].rules[3].next =0;}
else
        {ruleset[i].rules[3].next =1;}

ruleset[i].rules[4].left        = 1;
ruleset[i].rules[4].me          = 0;
ruleset[i].rules[4].right       = 0;
j = rand() % 100 + 1;
if(j>50)
        {ruleset[i].rules[4].next =0;}
else
        {ruleset[i].rules[4].next =1;}

ruleset[i].rules[5].left        = 1;
ruleset[i].rules[5].me          = 0;
ruleset[i].rules[5].right       = 1;
j = rand() % 100 + 1;
if(j>50)
        {ruleset[i].rules[5].next =0;}
else
        {ruleset[i].rules[5].next =1;}

ruleset[i].rules[6].left        = 1;
ruleset[i].rules[6].me          = 1;
ruleset[i].rules[6].right       = 0;
j = rand() % 100 + 1;
if(j>50)
        {ruleset[i].rules[6].next =0;}
else
        {ruleset[i].rules[6].next =1;}

ruleset[i].rules[7].left        = 1;
ruleset[i].rules[7].me          = 1;
ruleset[i].rules[7].right       = 1;
j = rand() % 100 + 1;
if(j>50)
        {ruleset[i].rules[7].next =0;}
else
        {ruleset[i].rules[7].next =1;}
```

```
	}
}

void read_conf()
{
	char c;
	int i;
	// read the initial state configuration from input file
	i=0;
	while((c = fgetc(input_file)) != '\n')
	{
		input[i] = c - 48;		//ascii code of 0 is 48
		i++;
	}
	// read the final state from the configuration file
	i=0;
	while((c = fgetc(input_file)) != '\n')
	{
		output[i] = c - 48;		//ascii code of 0 is 48
	 i++;
	}
	// read the intermediate state 1 from the configuration file
	i=0;
	while((c = fgetc(input_file)) != '\n')
	{
		inter1[i] = c - 48;		//ascii code of 0 is 48
		i++;
	}
	// read the intermediate state 2 from the configuration file (file must finish with new line \n character)
	i=0;
	while((c = fgetc(input_file)) != '\n')
	{
		inter2[i] = c - 48;		//ascii code of 0 is 48
		i++;
	}
}

void init_ca()
{
	int i,j;
	for (j=0; j<child; j++)
	{
		for (i=0; i<size; i++)
		{
			automata[j][i] = input[i];
		}
		weight[j] = 1;
	}
	for (i=0; i<size; i++)
	{
		next[i] = 0;
	}
}

int find_rule (int row, int col)
{
	int l,m,r,j;
	m = automata[row][col];
	if(col == 0)
	{
		r = automata[row][col + 1];
		l = automata[row][size - 1];
	} else
	if(col == size - 1)
	{
		r = automata[row][0];
		l = automata[row][col - 1];
```

```
		} else
		{
			l = automata[row][col - 1];
			r = automata[row][col + 1];
		}
		for (j=0; j<8; j++)
		{
			if((ruleset[row].rules[j].left == l) && (ruleset[row].rules[j].me == m) && (ruleset[row].rules[j].right == r))
				return ruleset[row].rules[j].next;
		}
		return -1;
}

void run()
{
		int l,i,j,new_state,count,found1,found2;
		for (i=0; i<child; i++)
		{
			found1 = 0;
			found2 = 0;
			for (l=1; l<loops; l++)
			{
				for (j=0; j<size; j++)
				{
					new_state = find_rule(i,j);
					if (new_state == -1)
					{
						printf("\n Error \n");
						exit(-1);
					}
					next[j] = new_state;
				}
				for (j=0; j<size; j++)
				{
					automata[i][j]=next[j];			//replace
					next[j]=0;
				}
				//		if the loop is where the 1st intermediate state is supposed to be then check if it's equal
				if((l >= position1) && (l <= position2))
				{
					count = 0;
					for(j=0; j<size; j++)
					{
						if(inter1[j]==automata[i][j]) count++;
					}
					if(count==size) found1 = 1;
				}
				//		if the loop is where the 2nd intermediate state is supposed to be then check if it's equal
				if((l >= position3) && (l <= position4))
				{
					count = 0;
					for(j=0; j<size; j++)
					{
						if(inter2[j]==automata[i][j]) count++;
					}
					if(count==size) found2 = 1;
				}
			}
			if(found1 == 0) weight[i] = weight[i] - 0.3;
			if(found2 == 0) weight[i] = weight[i] - 0.3;
		}
}

void run_worst()		// evolve the 2 new generated automata with the new rules
{
		int l,j,new_state,count,found1,found2;
		// for rule worst1 - worst ranked
```

```
found1 = 0;
found2 = 0;
for (l=1; l<loops; l++)
{
	for (j=0; j<size; j++)
	{
		new_state = find_rule(worst1,j);
		if (new_state == -1)
		{
			printf("\n Error \n");
			exit(-1);
		}
		next[j] = new_state;
	}
	for (j=0; j<size; j++)
	{
		automata[worst1][j]=next[j];			//replace
		next[j]=0;
	}
	//	if the loop is where the 1st intermediate state is supposed to be then check if it's equal
	if((l >= position1) && (l <= position2))
	{
		count = 0;
		for(j=0; j<size; j++)
		{
			if(inter1[j]==automata[worst1][j]) count++;
		}
		if(count==size) found1 = 1;//weight[worst1] = 0.5;
	}
	//	if the loop is where the 2nd intermediate state is supposed to be then check if it's equal
	if((l >= position3) && (l <= position4))
	{
		count = 0;
		for(j=0; j<size; j++)
		{
			if(inter2[j]==automata[worst1][j]) count++;
		}
		if(count==size) found2 = 1;
	}
}
if(found1 == 0) weight[worst1] = weight[worst1] - 0.3;
if(found2 == 0) weight[worst1] = weight[worst1] - 0.3;
found1 = 0;
found2 = 0;
// for rule worst2 - 2nd worst ranked
for (l=1; l<loops; l++)
{
	for (j=0; j<size; j++)
	{
		new_state = find_rule(worst2,j);
		if (new_state == -1)
		{
			printf("\n Error \n");
			exit(-1);
		}
		next[j] = new_state;
	}
	for (j=0; j<size; j++)
	{
		automata[worst2][j]=next[j];  //replace
		next[j]=0;
	}
	//	if the loop is where the 1st intermediate state is supposed to be then check if it's equal
	if((l >= position1) && (l <= position2))
	{
		count = 0;
		for(j=0; j<size; j++)
```

```
                        {
                                if(inter1[j]==automata[worst2][j]) count++;
                        }
                        if(count==size) found1 = 1;//weight[worst1] = 0.5;
                }
                //      if the loop is where the 2nd intermediate state is supposed to be then check if it's equal
                if((l >= position3) && (l <= position4))
                {
                        count = 0;
                        for(j=0; j<size; j++)
                        {
                                if(inter2[j]==automata[worst2][j]) count++;
                        }
                        if(count==size) found2 = 1;
                }
        }
        if(found1 == 0) weight[worst2] = weight[worst2] - 0.3;
        if(found2 == 0) weight[worst2] = weight[worst2] - 0.3;
}

void init_attractor()                   // set the attractor from the config file
{
        int i;
        for(i=0; i<size; i++)
        attractor[i] = output[i];
}

void fitness()                          // calculate the fitness for all the automata
{
        int i,j,count;
        for(i=0; i<child; i++)
        {
                count=0;
                for(j=0; j<size; j++)
                {
                        if(attractor[j]==automata[i][j]) count++;
                }
                fit[i]=count * weight[i];
        }
}

// calculate the fitness for the 2 previous worst ranked automata, executed now with the new generated rules
void fitness_worst()
{
        int j,count;
        // for worst1
        count=0;
        for(j=0; j<size; j++)
        {
                if(attractor[j]==automata[worst1][j]) count++;
        }
        fit[worst1]=count * weight[worst1];
        // for worst2
        count=0;
        for(j=0; j<size; j++)
        {
                if(attractor[j]==automata[worst2][j]) count++;
        }
        fit[worst2]=count * weight[worst2];
}

void best_rules()
{
        //rank is ordered by mergesort -- applying roulette-wheel technique
        int sum = 0;
        int i,k,j;
        for(i=0; i<child; i++)
```

```
                sum = sum + fit[i];
        i = rand() % sum + 1;
        k = rand() % sum + 1;
        sum = 0;
        j = 0;
        while(j < child)
        {
                if((i>sum) && (i<sum+fit[j]+1))
                {
                        best1 = j;
                }
                if((k>sum) && (k<sum+fit[j]+1))
                {
                        best2 = j;
                }
                sum = sum + fit[j];
                j++;
        }
}

void worst_rules()
{
        // rank is ordered by mergesort
        worst1 = rank[0];
        worst2 = rank[1];
}

void replace()
{
        int i,j;
        i = rand() % 100 + 1;
        if(i<31)                                // crossover rate = 0,7
        {
                ruleset[worst1].rules[0].next = ruleset[best1].rules[0].next;
                ruleset[worst1].rules[1].next = ruleset[best1].rules[1].next;
                ruleset[worst1].rules[2].next = ruleset[best1].rules[2].next;
                ruleset[worst1].rules[3].next = ruleset[best1].rules[3].next;
                ruleset[worst1].rules[4].next = ruleset[best1].rules[4].next;
                ruleset[worst1].rules[5].next = ruleset[best1].rules[5].next;
                ruleset[worst1].rules[6].next = ruleset[best1].rules[6].next;
                ruleset[worst1].rules[7].next = ruleset[best1].rules[7].next;

                ruleset[worst2].rules[0].next = ruleset[best2].rules[0].next;
                ruleset[worst2].rules[1].next = ruleset[best2].rules[1].next;
                ruleset[worst2].rules[2].next = ruleset[best2].rules[2].next;
                ruleset[worst2].rules[3].next = ruleset[best2].rules[3].next;
                ruleset[worst2].rules[4].next = ruleset[best2].rules[4].next;
                ruleset[worst2].rules[5].next = ruleset[best2].rules[5].next;
                ruleset[worst2].rules[6].next = ruleset[best2].rules[6].next;
                ruleset[worst2].rules[7].next = ruleset[best2].rules[7].next;
        }
        else
        {
                ruleset[worst1].rules[0].next = ruleset[best1].rules[0].next;
                ruleset[worst1].rules[1].next = ruleset[best1].rules[1].next;
                ruleset[worst1].rules[2].next = ruleset[best1].rules[2].next;
                ruleset[worst1].rules[3].next = ruleset[best1].rules[3].next;
                ruleset[worst1].rules[4].next = ruleset[best2].rules[4].next;
                ruleset[worst1].rules[5].next = ruleset[best2].rules[5].next;
                ruleset[worst1].rules[6].next = ruleset[best2].rules[6].next;
                ruleset[worst1].rules[7].next = ruleset[best2].rules[7].next;

                ruleset[worst2].rules[0].next = ruleset[best2].rules[0].next;
                ruleset[worst2].rules[1].next = ruleset[best2].rules[1].next;
                ruleset[worst2].rules[2].next = ruleset[best2].rules[2].next;
                ruleset[worst2].rules[3].next = ruleset[best2].rules[3].next;
                ruleset[worst2].rules[4].next = ruleset[best1].rules[4].next;
```

```
                ruleset[worst2].rules[5].next = ruleset[best1].rules[5].next;
                ruleset[worst2].rules[6].next = ruleset[best1].rules[6].next;
                ruleset[worst2].rules[7].next = ruleset[best1].rules[7].next;
        }

        // mutation
        for(j=0;j<8;j++)
        {
                i = rand() % 100 + 1;
                // for first new rule
                if(i<13)   // 1/population = 1/8 = 12,5
                {
                        if (ruleset[worst1].rules[j].next == 1) ruleset[worst1].rules[j].next = 0;
                        else ruleset[worst1].rules[j].next = 1;
                }
                // for second new rule
                if(i<13)   // 1/population = 1/8 = 12,5
                {
                        if (ruleset[worst2].rules[j].next == 1) ruleset[worst2].rules[j].next = 0;
                        else ruleset[worst2].rules[j].next = 1;
                }
        }
}

void mergesort(int a[], int low, int high)              //mergesort for fitness ranking
{
        int mid;
        if(low<high)
        {
                mid=(low+high)/2;
                mergesort(a,low,mid);
                mergesort(a,mid+1,high);
                merge(a,low,high,mid);
        }
}


void merge(int a[], int low, int high, int mid)         //function merge (part of mergesort algorithm)
{
        int i, j, k, c[child];
        i=low;
        j=mid+1;
        k=low;
        while((i<=mid)&&(j<=high))
        {
                if(fit[a[i]]<fit[a[j]])
                {
                        c[k]=a[i];
                        k++;
                        i++;
                }
                else
                {
                        c[k]=a[j];
                        k++;
                        j++;
                }
        }
        while(i<=mid)
        {
                c[k]=a[i];
                k++;
                i++;
        }
        while(j<=high)
        {
                c[k]=a[j];
```

```
                        k++;
                        j++;
                }
                for(i=low;i<k;i++)
                {
                a[i]=c[i];
                }
}

// main

int main(int argc,char *argv[])
{
                int c,i;
                int runs;
                input_file = fopen("input.txt", "r");
                output_file = fopen("output.txt", "w");
                 if(input_file==NULL)
                {
                         printf("Error: can't open input file.\n");
                         return 1;
                }
                if(output_file==NULL)
                {
                        printf("Error: can't open output file.\n");
                        return 1;
                }
                srand(time(NULL));
                read_conf();
                for(runs=0; runs<100; runs++)
                {
                        c=0;
                        init_rules();
                        init_ca();
                        run();
                        init_attractor();
                        fitness();
                        mergesort(rank, 0, child - 1);
                        while(fit[rank[child-1]] != size)
                        {
                                best_rules();
                                worst_rules();
                                replace();
                                init_ca();
                                run_worst();
                                fitness_worst();
                                c++;
                                mergesort(rank, 0, child - 1);
                        }
                        init_ca();
                        run();
                        fprintf(output_file,"%d,",c);
                        for(i=0;i<size;i++)
                                fprintf(output_file,"%d",automata[rank[child-1]][i]);
                        fprintf(output_file,",");
                        fprintf(output_file,"%d%d%d%d%d%d%d%d\n",ruleset[rank[child-1]].rules[7].next,ruleset[rank[child-
                        1]].rules[6].next,ruleset[rank[child-1]].rules[5].next,ruleset[rank[child-1]].rules[4].next,ruleset[rank[child-
                        1]].rules[3].next,ruleset[rank[child-1]].rules[2].next,ruleset[rank[child-1]].rules[1].next,ruleset[rank[child-
                        1]].rules[0].next);
                }
                fclose(input_file);
                fclose(output_file);
                return 0;
}
```

## Appendix 4: GA for non-uniform CA

```
/* Nichele Stefano */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <iostream.h>
#include <time.h>

// constant and object instantiation

const int size = 17;              //size of the ca
const int child = 10;             //number of childs
int automata[child][size];        //ca current states
int ruleset[child][size];         //ca current ruleset / cell type
int next[size];                   //ca next state
int loops = 64;                   //ca loop updates
int attractor[size];              //attractor

int best1, best2;                 // best 2 rules index
int worst1, worst2;               // worst 2 rules index

int fit[child];                           //array of fitness values, one for each rule
int rank[child] = {0,1,2,3,4,5,6,7,8,9};  // ranking of the fitness functions
float weight[child];                      // weight of the fitness function

int input[size];                  // input from configuration file
int output[size];                 // output from configuration file
int inter1[size];                 // intermediate state 1 from configuration file
int inter2[size];                 // intermediate state 2 from configuration file

const int position1 = 15; // range 1 first value
const int position2 = 15; // range 1 second value
const int position3 = 45; // range 2 first value
const int position4 = 45; // range 2 second value

FILE *input_file, *output_file;           // input and output files

// procedures

void init_rules()                         // random initialization of the rule-sets
{
        int i,j;
        for(i=0; i<child; i++)
        {
                for(j=0; j<size; j++)
                {
                        ruleset[i][j] = rand() % 12;
                }
        }
}

void read_conf()
{
        char c;
        int i;
        // read the initial state configuration from input file
        i=0;
        while((c = fgetc(input_file)) != '\n')
        {
                input[i] = c - 48;              //ascii code of 0 is 48
                i++;
        }
        // read the final state from the configuration file
```

```
	i=0;
	while((c = fgetc(input_file)) != '\n')
	{
		output[i] = c - 48;		//ascii code of 0 is 48
		i++;
	}
	// read the intermediate state 1 from the configuration file
	i=0;
	while((c = fgetc(input_file)) != '\n')
	{
		inter1[i] = c - 48;		//ascii code of 0 is 48
		i++;
	}
	// read the intermediate state 2 from the configuration file (file must finish  with new line \n character)
	i=0;
	while((c = fgetc(input_file)) != '\n')
	{
		inter2[i] = c - 48;		//ascii code of 0 is 48
		i++;
	 }
}

void init_ca()
{
	int i,j;
	for (j=0; j<child; j++)
	{
		for (i=0; i<size; i++)
		{
			automata[j][i] = input[i];
		}
		weight[j] = 1;
	}
	for (i=0; i<size; i++)
	{
		next[i] = 0;
	}
}

void run()
{
	int l,i,j,new_state,count,found1,found2,right,left,centre;
	float f1,f2;
	for (i=0; i<child; i++)
	{
		found1 = 0;
		found2 = 0;
		for (l=1; l<loops; l++)
		{
			for (j=0; j<size; j++)
			{
				centre = j;
				if(centre == 0)
					left = size - 1;
				else
					left = centre - 1;
				if(centre == size - 1)
					right = right = 0;
				else
					right = centre + 1;
				switch(ruleset[i][centre])
				{
					case 0 :
						next[centre] = automata[i][centre];
						break;
					case 1 :
						next[centre] = automata[i][left];
```

```
                break;
        case 2 :
                next[centre] = automata[i][right];
                break;
        case 3 :
                if((automata[i][left] == 0) && (automata[i][centre] == 0))
                        next[centre] = 0;
                if((automata[i][left] == 0) && (automata[i][centre] == 1))
                        next[centre] = 1;
                if((automata[i][left] == 1) && (automata[i][centre] == 0))
                        next[centre] = 1;
                if((automata[i][left] == 1) && (automata[i][centre] == 1))
                        next[centre] = 1;
                break;
        case 4 :
                if((automata[i][centre] == 0) && (automata[i][right] == 0))
                        next[centre] = 0;
                if((automata[i][centre] == 0) && (automata[i][right] == 1))
                        next[centre] = 1;
                if((automata[i][centre] == 1) && (automata[i][right] == 0))
                        next[centre] = 1;
                if((automata[i][centre] == 1) && (automata[i][right] == 1))
                        next[centre] = 1;
                break;
        case 5 :
                if((automata[i][left] == 0) && (automata[i][right] == 0))
                        next[centre] = 0;
                if((automata[i][left] == 0) && (automata[i][right] == 1))
                        next[centre] = 1;
                if((automata[i][left] == 1) && (automata[i][right] == 0))
                        next[centre] = 1;
                if((automata[i][left] == 1) && (automata[i][right] == 1))
                        next[centre] = 1;
                break;
        case 6 :
                if((automata[i][left] == 0) && (automata[i][centre] == 0))
                        next[centre] = 0;
                if((automata[i][left] == 0) && (automata[i][centre] == 1))
                        next[centre] = 1;
                if((automata[i][left] == 1) && (automata[i][centre] == 0))
                        next[centre] = 1;
                if((automata[i][left] == 1) && (automata[i][centre] == 1))
                        next[centre] = 0;
                break;
        case 7 :
                if((automata[i][centre] == 0) && (automata[i][right] == 0))
                        next[centre] = 0;
                if((automata[i][centre] == 0) && (automata[i][right] == 1))
                        next[centre] = 1;
                if((automata[i][centre] == 1) && (automata[i][right] == 0))
                        next[centre] = 1;
                if((automata[i][centre] == 1) && (automata[i][right] == 1))
                        next[centre] = 0;
                break;
        case 8 :
                if((automata[i][left] == 0) && (automata[i][right] == 0))
                        next[centre] = 0;
                if((automata[i][left] == 0) && (automata[i][right] == 1))
                        next[centre] = 1;
                if((automata[i][left] == 1) && (automata[i][right] == 0))
                        next[centre] = 1;
                if((automata[i][left] == 1) && (automata[i][right] == 1))
                        next[centre] = 0;
                break;
        case 9 :
                if((automata[i][left] == 0) && (automata[i][centre] == 0))
                        next[centre] = 1;
```

```
                                        if((automata[i][left] == 0) && (automata[i][centre] == 1))
                                                next[centre] = 1;
                                        if((automata[i][left] == 1) && (automata[i][centre] == 0))
                                                next[centre] = 1;
                                        if((automata[i][left] == 1) && (automata[i][centre] == 1))
                                                next[centre] = 0;
                                        break;
                                case 10 :
                                        if((automata[i][centre] == 0) && (automata[i][right] == 0))
                                                next[centre] = 1;
                                        if((automata[i][centre] == 0) && (automata[i][right] == 1))
                                                next[centre] = 1;
                                        if((automata[i][centre] == 1) && (automata[i][right] == 0))
                                                next[centre] = 1;
                                        if((automata[i][centre] == 1) && (automata[i][right] == 1))
                                                next[centre] = 0;
                                        break;
                                case 11 :
                                        if((automata[i][left] == 0) && (automata[i][right] == 0))
                                                next[centre] = 1;
                                        if((automata[i][left] == 0) && (automata[i][right] == 1))
                                                next[centre] = 1;
                                        if((automata[i][left] == 1) && (automata[i][right] == 0))
                                                next[centre] = 1;
                                        if((automata[i][left] == 1) && (automata[i][right] == 1))
                                                next[centre] = 0;
                                        break;
                                }
                        }
                        for (j=0; j<size; j++)
                        {
                                automata[i][j]=next[j];  //replace
                                next[j]=0;
                        }
                        //      if the loop is where the 1st intermediate state is supposed to be then check if it's equal
                        if((l >= position1) && (l <= position2))
                        {
                                count = 0;
                                for(j=0; j<size; j++)
                                {
                                        if(inter1[j]==automata[i][j]) count++;
                                }
                                f1 = 0.3 - (0.3 * ((count*1.0)/size));
                        }
                        //      if the loop is where the 2nd intermediate state is supposed to be then check if it's equal
                        if((l >= position3) && (l <= position4))
                        {
                                count = 0;
                                for(j=0; j<size; j++)
                                {
                                        if(inter2[j]==automata[i][j]) count++;
                                }
                                f2 = 0.3 - (0.3 * ((count*1.0)/size));
                        }
                }
                weight[i] = weight[i] - f1 - f2;
        }
}

void init_attractor()                   // set the attractor from the config
{
        int i;
        for(i=0; i<size; i++)
        attractor[i] = output[i];
}

void fitness()                          // calculate the fitness for all the automata
```

```
{
	int i,j,count;
	for(i=0; i<child; i++)
	{
		count=0;
		for(j=0; j<size; j++)
		{
			if(attractor[j]==automata[i][j]) count++;
		}
		fit[i]=count * weight[i];
	}
}

void best_rules()		//rank is ordered by mergesort -- applying roulette-wheel technique
{
	int sum = 0;
	int i,k,j;
	for(i=0; i<child; i++)
		sum = sum + fit[i];
	i = rand() % sum + 1;
	k = rand() % sum + 1;
	sum = 0;
	j = 0;
	while(j < child)
	{
		if((i>sum) && (i<sum+fit[j]+1))
		{
			best1 = j;
		}
		if((k>sum) && (k<sum+fit[j]+1))
		{
			best2 = j;
		}
		sum = sum + fit[j];
		j++;
	}
}

void worst_rules()		// rank is ordered by mergesort
{
	worst1 = rank[0];
	worst2 = rank[1];
}

void replace()
{
	int i,j,r,k;
	// generate all new mutated rulesets except the one with best fitness
	for(k=0;k<child;k++)
	{
		if((rank[child-1]!=k) && (rank[best1]!=k) )
		{
			for(j=0;j<size;j++)
			{
				i = rand() % 100 + 1;

				if(i<9)	// 1/population = 1/12 = 8,3
				{
					r = rand() % 12;
					ruleset[k][j] = r;
				}
				else
				{
					ruleset[k][j] = ruleset[best1][j];
				}
			}
		}
```

```
	}
}

void mergesort(int a[], int low, int high)		//mergesort for fitness ranking
{
	int mid;
	if(low<high)
	{
		mid=(low+high)/2;
		mergesort(a,low,mid);
		mergesort(a,mid+1,high);
		merge(a,low,high,mid);
	}
}

void merge(int a[], int low, int high, int mid)		//function merge (part of mergesort algorithm)
{
	int i, j, k, c[child];
	i=low;
	j=mid+1;
	k=low;
	while((i<=mid)&&(j<=high))
	{
		if(fit[a[i]]<fit[a[j]])
		{
			c[k]=a[i];
			k++;
			i++;
		}
		else
		{
			c[k]=a[j];
			k++;
			j++;
		}
	}
	while(i<=mid)
	{
		c[k]=a[i];
		k++;
		i++;
	}
	while(j<=high)
	{
		c[k]=a[j];
		k++;
		j++;
	}
	for(i=low;i<k;i++)
	{
	a[i]=c[i];
	}
}

// main

int main(int argc,char *argv[])
{
	int c,i;
	int runs;
	input_file = fopen("input.txt", "r");
	output_file = fopen("output.txt", "w");
	if(input_file==NULL)
	{
		printf("Error: can't open input file.\n");
		return 1;
	}
```

```
	if(output_file==NULL)
	{
		printf("Error: can't open output file.\n");
		return 1;
	}
	srand(time(NULL));
	read_conf();
	for(runs=0; runs<1; runs++)
	{
		c=0;
		init_rules();
		init_ca();
		run();
		init_attractor();
		fitness();
		mergesort(rank, 0, child - 1);
		while(fit[rank[child-1]] != size)
		{
			best_rules();
			worst_rules();
			replace();
			init_ca();
			run();
			fitness();
			c++;
			mergesort(rank, 0, child - 1);
		}
		init_ca();
		run();
		printf("cycle = %d \n",c);
		fprintf(output_file,"%d,",c);
		for(i=0;i<size;i++)
			fprintf(output_file,"%d",automata[rank[child-1]][i]);
		fprintf(output_file,",");
		for(i=0;i<size;i++)
			fprintf(output_file,"%d ",ruleset[rank[child-1]][i]);
		fprintf(output_file,"\n");
	}
	fclose(input_file);
	fclose(output_file);
	return 0;
}
```