NTNU
Norwegian University of
Science and Technology

Evaluating the Influence of Network Structure on Boolean Networks and Cellular Automata

Harald Hvaal

Master of Science in Computer Science
Submission date: June 2010
Supervisor: Gunnar Tufte, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Problem Description

Many papers have been published respectively on the qualities of Boolean networks and Cellular Automata, but little work has been done on comparing these networks to each other. Network parameters such as input count and choice of Boolean functions are often fixed in preparation of the experiments with less regard to what effect that choice has.

A broader overview of how the choice of network structure and network parameters will affect the behavior of the network is sought. Differences and similarities between the behavior of the networks are to be compared, especially how the networks differ when specific parameters are changed.

Assignment given: 22. January 2010
Supervisor: Gunnar Tufte, IDI

Abstract

While there have been many papers respectively on the qualities of Boolean networks and Cellular Automata, little work has been done on comparing these networks to each other. Network parameters such as input count and choice of Boolean functions are often fixed in preparation of the experiments with less regard to what effect that choice has.

In this paper a broader overview of how the choice of network structure and network parameters will affect the behavior of the network is given. Metrics such as iterations until stabilization (intermediary state count) and complexity of network behavior over time (functional complexity) are proposed, and evaluated for a set of 15 different network configurations. CA networks are observed to have much less functional complexity than BN, and in general BN seems to have more potential for complex behavior. It is also observed that for increasing values of dimension count/input count the functional complexity decreases.

Contents

Chapter 1

Introduction

1.1 Sparsely connected networks

In the recent years, much research has been devoted to the study of gene networks. Although initially appearing as a way of modeling and simulating concepts found in biology[18], recent years have seen studies directing the field towards networks for general computation.

The networks are built up from quantities of simple Boolean functions. The functions themselves are easily specified and understood by simple truth tables, however, as they are assembled together into networks of Boolean functions it becomes apparent that the network as a whole has an emerging behavior not visible by the simple sub-elements alone. This is known as the *emergent property.*

Both Kauffman and Wolfram's research reflects this quality in their own ways. Kauffman's research asserts that complex mechanisms can be modeled through few components put together. An example of this is the yeast cell-cycle network[19], a regulatory network of the budding yeast, modeled using Boolean networks. Wolfram's research displays how small, almost trivial networks are still able to perform seemingly very complex work. He demonstrates this through his 1-dimensional Cellular Automata that are able to generate complex patterns through elementary, homogeneous rule sets.

Observing the mechanisms found in nature, it has become apparent that the biological gene networks are sparsely, rather than densely, connected, with average number of connections per gene as less than two[22]. These sparsely connected networks are robust in the face of natural selection, even though early studies have pointed to denser networks being more evolutionary robust[23].7

1.2 Problem description

While there have been many papers respectively on the qualities of Boolean networks and Cellular Automata, little work has been done on comparing these networks to each other. Network parameters such as input count (K in [2]) and choice of Boolean functions are often fixed in preparation of the experiments with less regard to what effect that choice has.

In this paper a broader overview of how the choice of network structure and network parameters will affect the behavior of the network is sought. Factors such as iterations until stabilization (intermediary state count) and complexity of network behavior over time (functional complexity) are considered in these regards.

1.3 How the problem was approached

From the beginning, a hands-on approach involving mainly simulation for verification was chosen. Theories were made, and tested for consistency on a large set of random networks through a simulation library.

The reason for using simulation was also based in the practical approach of the topic; we are more interested in discoveries that are proven realistic through experimentation rather than ones based purely on theory. Also, with little pre-existing theoretical work on the subject, the first step towards this is experimentation.

In order to keep the problem size manageable, ultimately limitations had to be applied. These are discussed in section 6.2.

Chapter 2

Background

2.1 Parallel/Distributed computing

After the CPU clock-rate craze of the early 2000s have faded, there has been several rising trends in the general field of increased computing capacity. The two trends of parallel computing and distributed computing, are going in the direction of increasing the computing units and often simplifying the same units. Computation performed on these systems are performed in a parallel rather than serial fashion, targeting a quite different problem domain than before.

2.1.1 A new paradigm for massive parallelization

Although it is relatively easy to build these systems in hardware, the classical development process of think, design, implement is difficult to employ in massively parallel systems as it is significantly harder to predict the outcome from out-of-order execution. Sometimes a system will even suffer performance-wise only in order to make the design reasonable to humans. The need for a new development paradigm is thus showing to be increasingly significant as parallel computing units become more widespread.

Much of the problem with these new systems has been to utilize them efficiently. Not all problems perform as well when split into a large number of sub-problems due to parts that need to be sequentially executed, and as Amdhahl's law[6] points out, these problems will not get any less significant as we increase the number of computational units.

2.1.2 Types of problems

Although Amdahl's law gives a rather pessimistic perspective on parallelization of existing problems, there are reasons for not giving up on parallelization of problems just yet; Gustafson's law[7] points this out by suggesting that we adjust the problem size, thus instead altering the problem to fit our parallel computing systems. Image processing applications, for example, is a category of problems that can be very efficiently parallelized due to their favorable locality properties.

2.2 Emergence and self-organizing systems

Self-organization is defined in [3] as "the spontaneous emergence of global coherence out of local interactions". This global coherence is referred to as the *emergent property*. The essence of this concept is that there is no external control or influence that is leading all the participants in the same direction; the goal the system as a whole will advance towards is implicitly defined from the collective choice of all participants.

This kind of systems is very commonly found in nature, two basic examples (from [3] p5-6) being the polarity of magnetic objects and Bénard rolls. With magnetization, one can observe how the smaller elements in magnetic materials tend to be influenced by and follow the same magnetization as the surrounding ones, eventually resulting in the object having global magnetic polarity. Bénard rolls is the phenomenon of heated particles in a liquid moving in a set of parallel "rolls".

2.2.1 System applications

Systems employing the mechanisms above generally share some traits: they work with a large body of simple computational units, and deal with noise. This noise could be units sometimes returning the wrong value, it could be units that stop returning any values at all, or even invalid values as a result of invasion from a malicious presence. In general, the noise poses a negative influence that has to be dealt with.

2.2.2 Nanoscale machinery

This is a recently advancing field where machinery of sizes smaller than the human eye can see is assembled at an molecular level[16]. It is relevant because a system of this size is hard, if not impossible, to repair or build by hand. In addition, external noise also becomes especially important when you're considering a machine that can be destroyed by the push of a finger.

For this field we can imagine an immune system that governs the construction of new machines and rejects the ones that do not conform to specifications. The machines will perform simple work at the individual level, but seen as a complete system a sense of intelligence will emerge from the mass of machines.

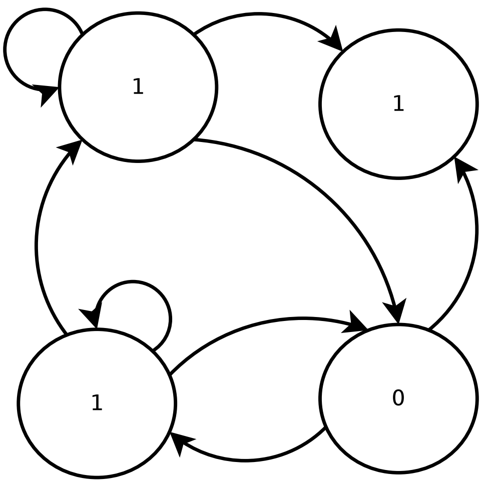


Figure 2.1: Example of a Boolean network, with # of nodes N=4 and # of inputs pr node K=2. The circles represent nodes in the network, with a corresponding Boolean function (not shown in the figure). The arrows pointing to other nodes represents the inputs from other nodes. The current state is shown in the center of each node. Note how the node can also be “aware” of it’s own state by including itself as an input.

2.3 Boolean networks

The Boolean network, first proposed by Stuart A. Kauffman[1] in 1969, consists of a system of N binary-state nodes and K inputs to each of these nodes. The evaluation of the inputs, producing a new state, is done according to a logic function, typically stored as a look-up table of size 2^K (with certain[1] exceptions), for small sizes of K. The network is initialized with an initial binary state over all nodes, and iterated a number of times (typically until the state has converged). Iterating a network refers to according to some scheme, choose one or more nodes and update their internal state according to their Boolean function and inputs.

Although originally developed as a model of genetic regulatory networks, it can be used to model arbitrarily complex mechanism by varying the initial state and inputs[2]. Boolean networks are currently at a state where we know that the network has large computing potential, but one is unsure of how to control the computation, or in other words how to program the Boolean network.

[1]Depending on whether the logic function makes a difference out of which input is connected where, this table might be smaller. For example, for the case of the AND logic function, the output of (0,1) and (1,0) is the same. Both situations will be assumed equally relevant should the case appear.

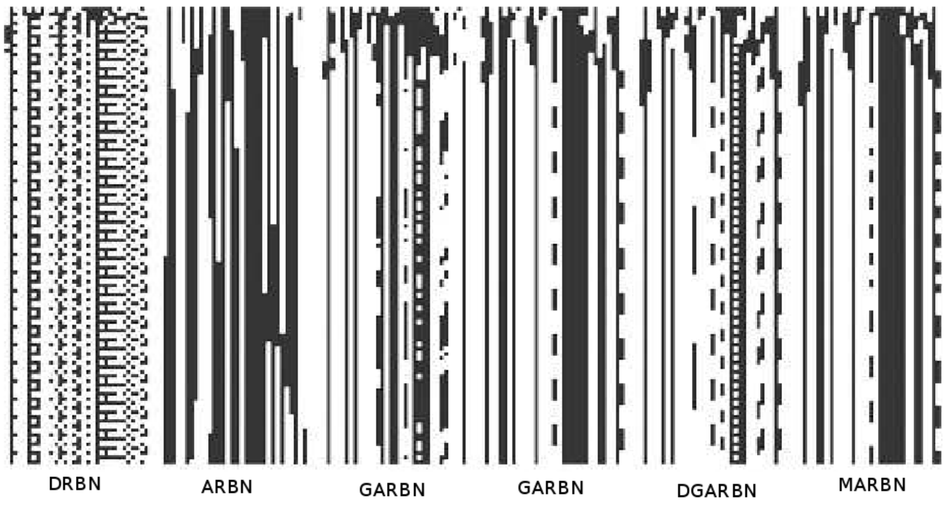


Figure 2.2: Visualization of various types of RBN updating schemes over time. One horizontal line is the state over all nodes at a point in time, black/white corresponding to on/off state. On the very left is a Deterministic RBN (DRBN), clearly showing that the system has stabilized by its generating the same pattern repeatedly. Here all nodes are updated synchronously once per iteration. The other kinds to the right show lesser amounts of stability due to various degrees of asynchronicity in the updating scheme. For more information on these other types of RBN, see [2] p10, where this diagram is borrowed from.

2.3.1 RBN (Random Boolean Networks)

RBN is a further variation of the Boolean networks where the connectivity between the nodes is generated randomly. Used together with artificial evolution methodology as an approach to the problem of programming Boolean networks, randomization is repeated while testing against a specified fitness function until the desired result has been reached.

2.3.2 Cellular Automata

Cellular Automata[5](CA) is a network where all nodes are arranged in a grid, and their inputs are defined to be their immediate neighbors. The classical example of this type of network is *Conway's Game of Life*[4], a Cellular Automata defined from a small set of simple rules. Conway's Game of Life is known as being Turing complete, thus having the capability of general computation.

CA is a special case of Boolean networks, with each node's inputs defined to be a set of its neighbors. Given how conveniently a CA is organized with regards to spatial locality (all nodes by definition will only read from those close to themselves), CA is an architecture that can prove to be efficient when implemented in hardware.

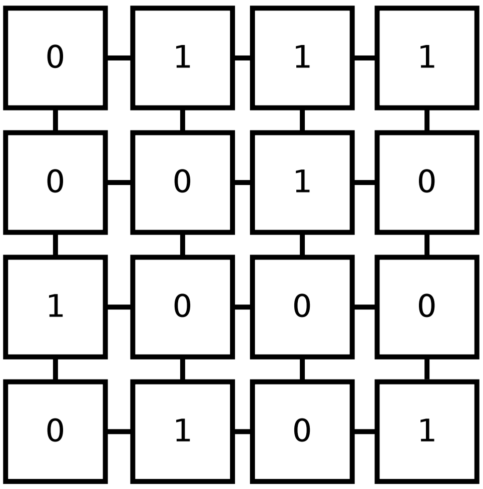


Figure 2.3: Example of a two-dimensional two-state Cellular Automata network. The squares represents cells in the network, and the lines are connections to other cells. The cells have a state, and are connected to the neighbors in their vicinity. This means that when the network is operating, each node will use the state of its neighbors as input for the next iteration. While this is a two-dimensional CA, one-dimensional and three-dimensional are two other popular types of CA networks.

2.3.3 Updating scheme

Boolean networks may also be categorized according to the kind of updating scheme they follow. The updating scheme decides what point in time and in what order the nodes are iterated. Synchronous Boolean networks (SBN) iterate synchronously over all nodes, giving a deterministic and stable behavior. Asynchronous Boolean networks (ABN) follows a scheme in which any node may update at any time, giving a more unpredictable behavior. ABN can be divided into further subcategories (see [2] p7-8), placing restrictions (for example defining time slots where the nodes may update) on the updating scheme.

SBN, while easy to resonate with due to their deterministic nature, are a less realistic model than ABN when used for simulating real-world phenomenons. As an example, the gene networks that Kauffman's networks originally set out to model do not at all update in a synchronous fashion.

Another perspective: imagine a large-scale Boolean network that should be implemented in hardware. In order to update all nodes at the exact same interval significant care must be taken when designing the hardware. The question of whether SBN or ABN is more relevant is outside the scope of this research, but nevertheless the distinction is an significant one, so both cases are be compared.

2.3.4 Threshold nodes

Threshold nodes is a kind of Boolean function defined via the following rule:

$$S_i(t+1) = \begin{cases} 1, & \sum_j a_{ij} S_j(t) > T \\ 0, & \sum_j a_{ij} S_j(t) < T \end{cases}$$

a_{ij} denotes the existence of a connection between node i and node j, and S_j is the state of node j. T is a threshold value set for the node. Intuitively, this can be explained as “the next state is 1 if more than T of a node’s inputs are in the enabled state at the current time, otherwise 0”. A variant that retains the current state if $\sum_j a_{ij} S_j(t) = T$ is used in [19].

There is also another type: negative threshold nodes. These are the inverted case, and can be defined as follows:

$$S_i(t+1) = \begin{cases} 0, & \sum_j a_{ij} S_j(t) > T \\ 1, & \sum_j a_{ij} S_j(t) < T \end{cases}$$

Threshold nodes are especially relevant to the members of the author’s research department, and for this reason it has received extra attention in this thesis.

2.3.5 Complexity

A central topic in this research is the notion of complexity. Although by itself a quite abstract measure, it can be divided into functional structural and functional complexity for a bit more concrete definition.

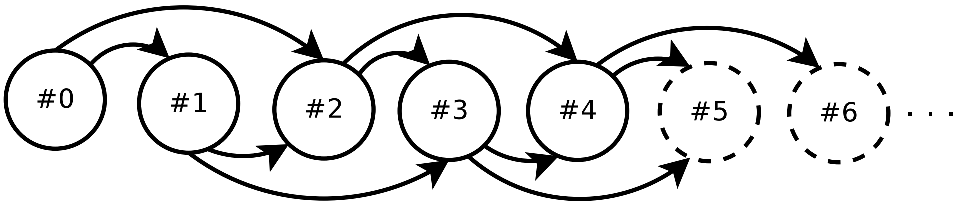
Structural complexity refers to the complexity of the network structure itself. Examples of low and high structural complexity are shown in Figure 2.4.

Functional complexity is the complexity seen only from the behavior of the network. For example, a network that stabilizes after few number of iterations can be said to have a low functional complexity, conversely one that stabilizes after a long time can be said to have a high functional complexity.

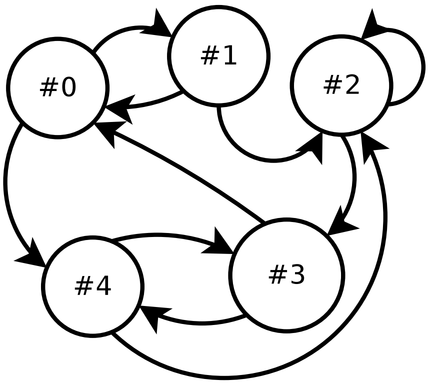
2.3.5.1 Kolmogorov complexity

The above definitions of structural and functional complexity are both instances of the more general notion of complexity known as Kolmogorov complexity. For this reason the general definition is included here.

Given a piece of data, the minimum number of computational resources needed to specify it in some universal description language is known as the Kolmogorov complexity. An example is given in Figure 2.5. It is impossible ([20]) to calculate this complexity, so in practice an approximation is used.



(a) An example network with low structural complexity. One can trivially see that there is a pattern in this particular network, specified by the rule "node N is connected to node N+1 and node N+2". Given the relative simplicity of this description we say that the network has a low structural complexity.



(b) A network with high structural complexity. It is easy to see that the connections here are more erratic, and therefore harder to describe.

Figure 2.4: Examples of structural complexity. The #-notation is used for identification of the individual nodes. Although the functions (eg. AND/OR/XOR/etc) of the nodes is also part if the network's structure, they have been omitted for simplicity.

```
abababababababababababababababababababababababababababababababab
4c1j5b2p0cv4w1x8rx2y39umgw5q85s7uraqbjfdppa0q7nieieqe9noc4cvafzf
```

Figure 2.5: Two examples of Kolmogorov complexity. Two strings are presented, each 64 characters long and composed of only lowercase letters and numbers. Using English as the universal description language, we can encode the first string as simply “ab 32 times”, giving it a low Kolmogorov complexity. The second string, however, does not seem to have any simple description other than the string itself and subsequently can be said to have a high Kolmogorov complexity.

An example real life implementation (which is tested later) is using the zlib compression library to encode the data. The compressability is reflected by the length of the compressed data, and this is used as an approximation of the Kolmogorov complexity. This is most interesting when lengths are compared with each other; specifically how much the library is able to deflate the data is less interesting.

2.3.6 Robustness

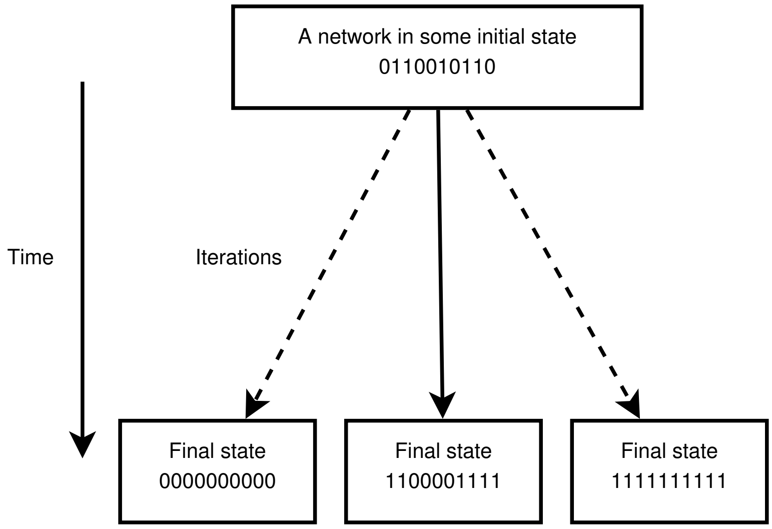
Robustness is the property of to which degree a system is unaffected by perturbations, usually caused by mutations or other random noise. Seen in the context of Boolean networks, this could mean nodes sometimes flipping their state, ignoring inputs or generally showing erratic behavior, but the network as a whole would still move towards the same end state. An illustration of perturbations to a system is found in Figure 2.6.

Robustness is observed in all biological systems in nature, and is a key aspect in biologically inspired models ([17]). A related concept is *redundancy*, often used as a means to attain robustness through duplication of calculation units. The robustness seen in nature, however, typically relies on a more complex integrated fault-tolerance rather than simple duplication. An example is the yeast network, presented in [19], conveniently titled “The yeast cell-cycle network is robustly designed”.

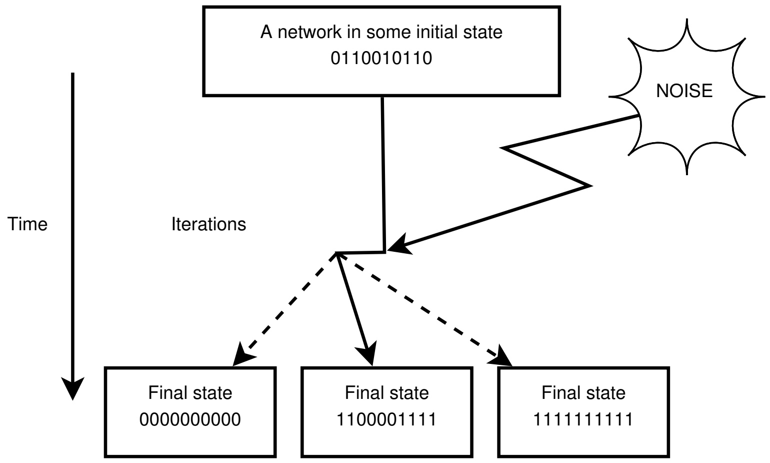
2.3.7 Stability

The stability [18] of a network refers to the tendency to settle in a stable state. While stable, the state of the network will not change unless otherwise perturbed by an external force.

For a synchronous network, determining whether a network is in this state is a trivial: once the network has stayed in the state continuously for more than one iteration it is stable (given that there is no external force acting upon it). For an asynchronous network, this is much more computationally expensive: from one network state to the next there is often a very high number of valid following states,



(a) Without noise



(b) With noise

Figure 2.6: A robust network with and without perturbations. The dotted lines represent possible paths from the initial state, while the solid one is the actual execution. When the network is perturbed, the execution path is noticeably disturbed, but the final state is still the same. If the network was less robust, or the noise was more significant, the network might end up in a different end state. A robust system naturally aims to minimize this kind of error.

and an approximation by observing whether a certain number of successive states are equal, is used instead.

Chapter 3

Theory

3.1 Methodology

While Boolean networks and artificial development recently has seen increasing attention, the field is still relatively new and at an early stage. There is still a lack of theoretical foundation and universally accepted methodology for the research. We aim to advance the field by proposing a clear overview of the various network structures and their effect on robustness. The main focus of this paper is to better evaluate what kinds of networks that are suitable for practical use in future applications of Boolean networks.

In order to the achieve this we will apply the scientific method through simulation for verification. The need for simulation appears because there is not yet any actual implementation of these networks in use.

Initially a hypothesis on the behavior of the networks is established, for example "Random Boolean networks in general have more complex behavior than Cellular Automata". Next an experiment testing this idea is designed, such as "Test 100 random Boolean networks and 100 Cellular Automata and observe their behavior over 1000 iterations each". Using the simulation library developed for this research, the experiment can then be implemented and executed, producing detailed data on how the networks behave. Finally, this data is processed with statistical tool such as R[1] and GraphViz[2].

Building upon these results, further experiments may be devised, leading the way towards a better understanding of the Boolean networks. More specifically, it is hoped that it may show which kinds of network are more likely to carry robust characteristics. This is especially relevant in order to find out to what degree the network type actually influences the potential for robustness.

[1]The R Project for Statistical Computing, `http://www.r-project.org/`
[2]Graphviz - Graph Visualization Software, `http://www.graphviz.org/`

3.2 Network structure

It is quickly evident from experimenting with simulation packages that the behavior of a network varies significantly depending on it's network structure. One of the overall goals is to elaborate further on these differences and get a sense of which types of networks that allow for complex, yet robust operation. For network structure, the following will be examined:

- Cellular Automata in 1, 2, and 3 dimensions
- Random Boolean networks with varying number of inputs

For node types (basically the truth table of the node), the following groups will be examined:

- Threshold nodes
- Negative threshold nodes
- AND/OR/XOR

3.3 Metrics

For the comparison of the network structures, a set of metrics is proposed.

3.3.1 Basin evaluation

A good way to understand the stability of a network is looking collectively at the final states. The *basin evaluation* method does this by N times resetting the network to a random initial state and iterate it until it stabilizes. The basin corresponding to each end state is incremented, and the end result with N set sufficiently high is a set of basins approximating the distribution of the states the network will usually end in. This is illustrated in Figure 3.1.

This computed set of basins can be used for further analysis, for example as a fitness for evolutionary algorithms.

3.3.2 Functional complexity

In [20], *zlib* compression is used to approximate Kolmogorov complexity in order to measure the structural complexity of networks. Inspired by this method, zlib is also used for our measure of functional complexity. Put more specifically, it is done as follows: for every iteration of the network, the state over all the nodes is combined into a single string of 0's and 1's, such as 00110101. Next, these strings (which are generated once per iteration) are concatenated and the resulting long string is run through zlib. The size of this deflated string compared to the size of the original concatenated string, is the metric we are interested in.

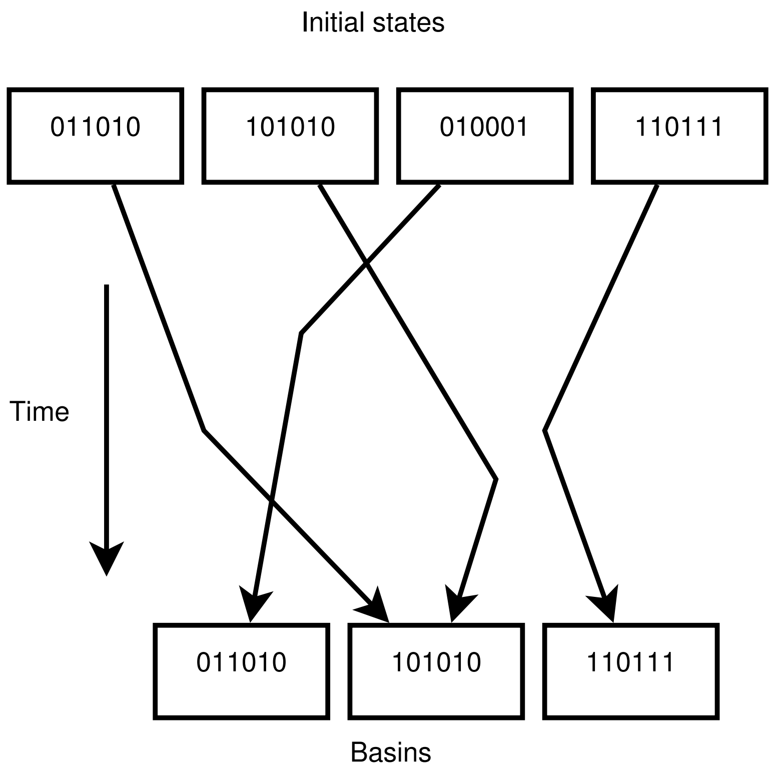


Figure 3.1: An example basins evaluation for 4 different initial states, shown at the top. For each unique end state, a basin is created, and the number of init states leading to this basin is counted. In this case, the end result is three basins of size respectively 1, 2, and 1.

The functional complexity tells us more about how a network behaves in operation, as opposed to properties like how big it is or how it is connected. This is especially important, because as Kauffman and Wolfram showed in their research, even small-scale trivial machines can exhibit surprisingly complex behavior.

3.3.3 Intermediary state count

With the right set of parameters, most network permutations will stabilize after a number of iterations. Determining the average value of this number is the goal of the *intermediary state count* metric. Given a certain network, it works by N times resetting it to a random state, and each time let it iterate T times until the network has stabilized. The average of needed T iterations (once for all N resets) becomes the final metric.

3.3.4 Robustness

This is a metric derived from the basin evaluation of a certain network and the basin evaluation of its perturbed variants. It relies on the premise that a robust network will function much the same way even when perturbed. As such, the results of the basins evaluation on the initial network, and the perturbed variant should ideally be largely the same. The robustness is then measured by the difference between the original and perturbed sets of basins.

While generating the basin sets is a relatively trivial task, comparing them is more of a challenge. Average basin size, basin size distribution and hamming distance between basins, are some properties that may be considered and weighted in order to get a well balanced metric for the robustness. In the following sub-sections these will be explained in detail. Note that these properties operate specifically on multiple sets of basins in a comparative manner (in contrast with properties that evaluate a single basin set of its own such as the basin entropy).

3.3.4.1 Average basin size difference

The average basin size is calculated by calculating the average basin size over a set of basins, once for the initial set and once for the perturbed set(s) of basins. The difference between the initial average basin size and the perturbed average basin size becomes the final metric.

3.3.4.2 Basin size distribution

Basin size distribution refers to the comparison of the individual basins, from the initial and the perturbed set. Two identical basin sets would give a top score, and degrees of different basin sizes will give corresponding degrees of penalties to the score.

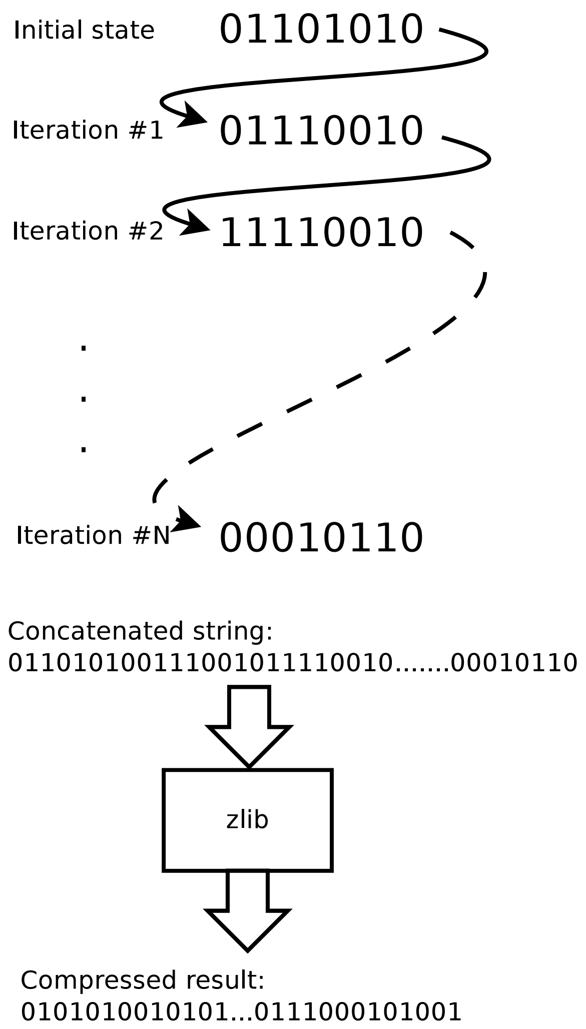


Figure 3.2: Calculation of functional complexity. The string at the top is the initial state over the network, and following the arrows downwards are the following state-strings for each iteration, which are concatenated and sent through the zlib compressor. At the bottom the compressed result is shown. The final metric of interest is the length of this compressed string divided by the length of the concatenated string.

3.3.5 Basin entropy

Yet another basin evaluation derived metric; the basin entropy measure takes a set of basins as input, and calculates a single value that reflects how entropic this set is. Small sets of large basins will yield large values, while larger set of smaller basins will yield lower values and can be said to be more entropic.

Chapter 4

Simulation

4.1 The simulation library

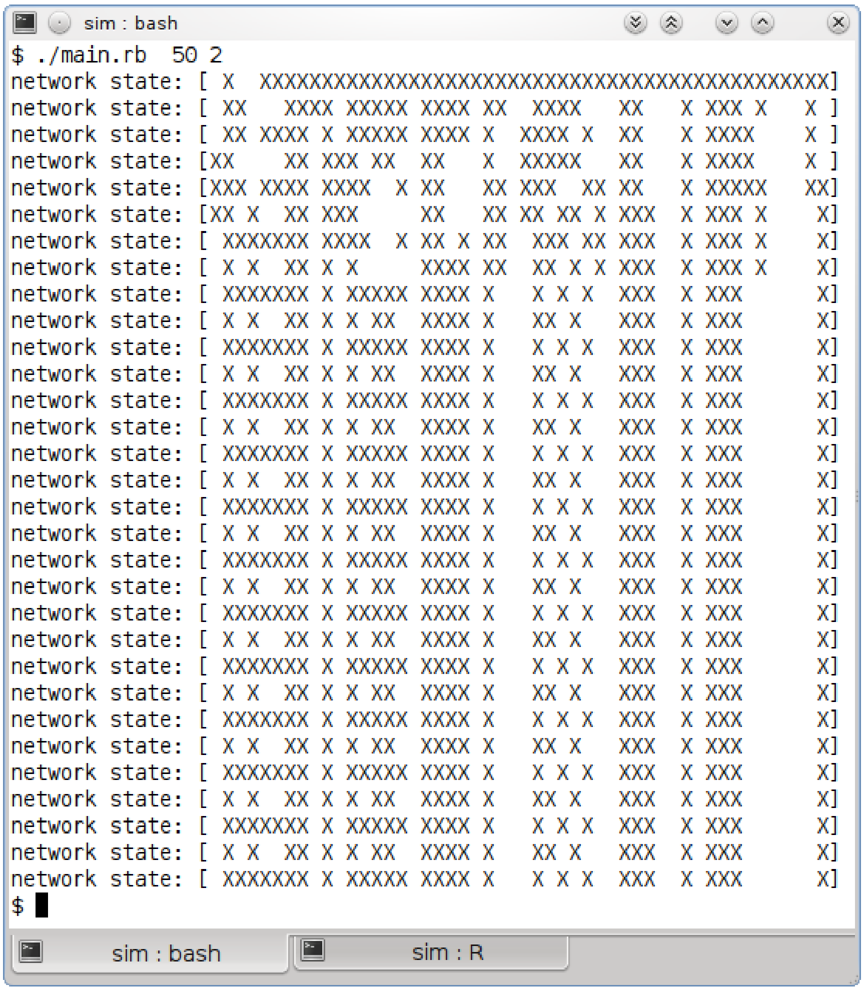
A software package was written to carry out the simulations. It allowed for continuous iteration of Boolean networks with an arbitrary K inputs. It also allowed for specifying the logic function for all nodes or individual ones. Being written in Ruby, an interpreted high-level language, for the purpose of assisting in prototyping and generating test data, it has been helpful in several simulations already.

The state across the network over time will be initialized, and then after every iteration read from the nodes and written to file. Next, the data was analyzed using R, a much used programming language for statistical computing. Theories and assumptions from earlier sections will be tested based on this work. Some examples of the output from the simulation software, processed with R and graphed is found in Figure 4.1.

In addition, a graphical user interface was also written in order to assist the frequent experimentation with different kinds of networks. A screen-shot is shown in Figure 4.3. This lets the user quickly visualize the effect of certain network parameters, as well as perform live modifications such as changing certain node types or apply simulated noise. It is also able to output the network structure into a format that can be read and rendered into a graph by GraphViz.

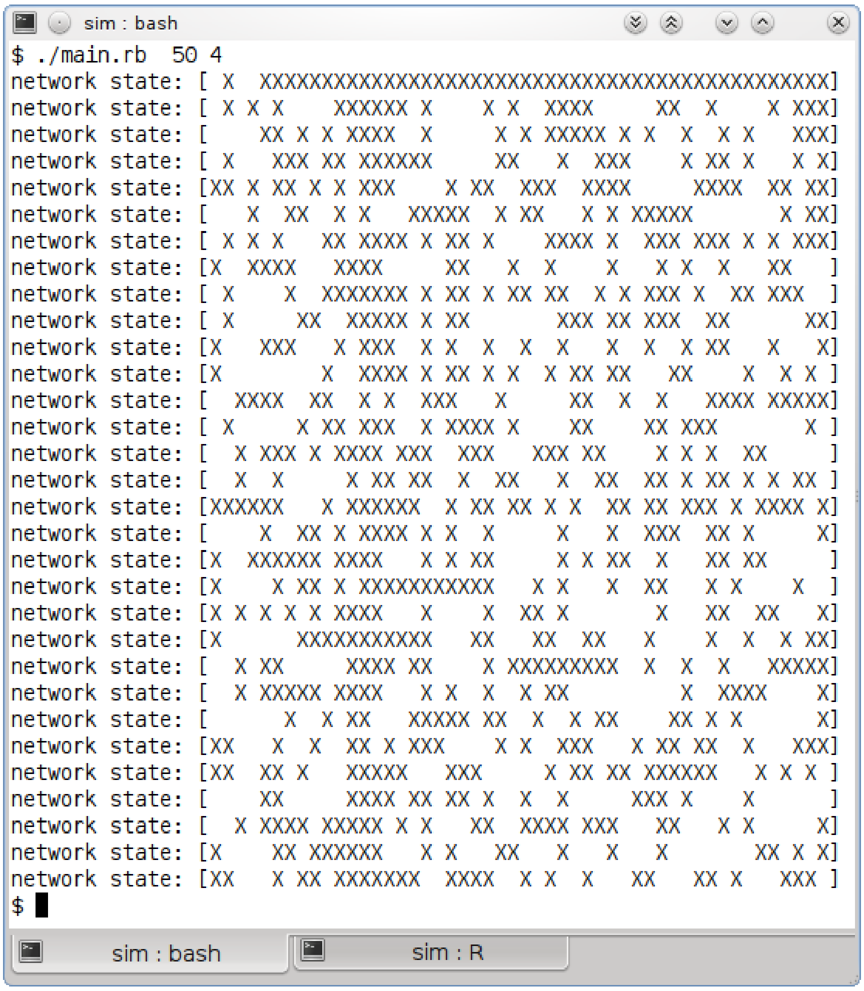
4.1.1 Choice of language

When the time to choose language came around, the choices were two-fold: compiled languages such as Java/C/C++, or interpreted language such as Python/Ruby. Running simulations often becomes a very CPU-intensive task due to the large scale of the networks, and a compiled language would make this execution significantly more efficient. However, also taking the need for rapid development and prototyping into consideration also puts the interpreted languages into a good position.



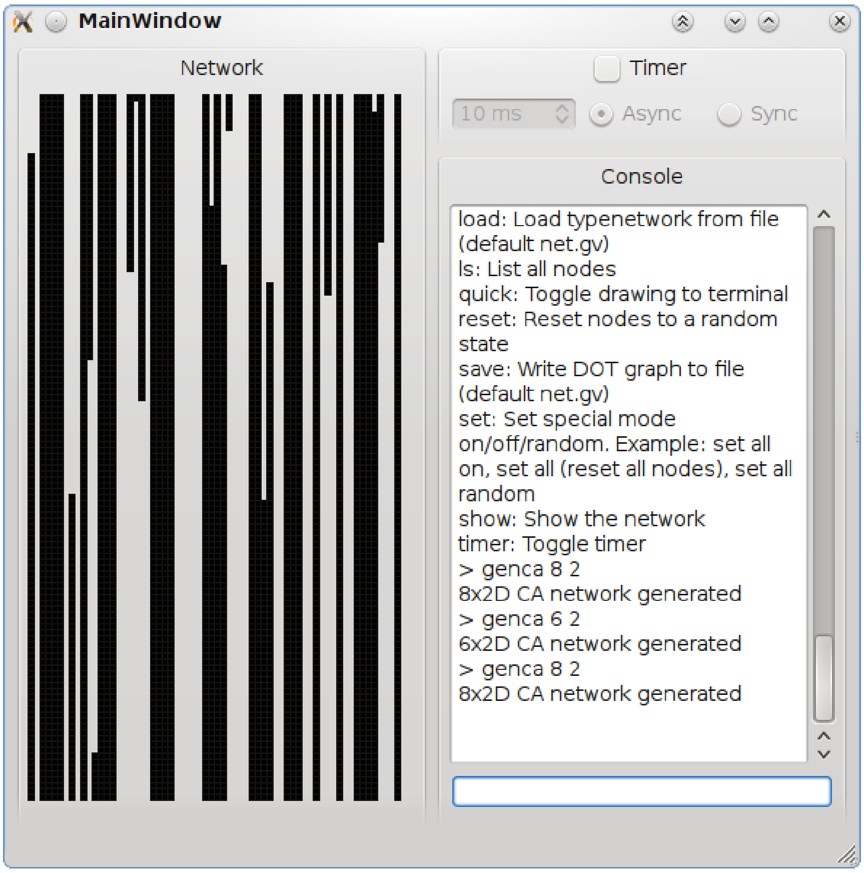
```
$ ./main.rb  50 2
network state: [ X  XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX]
network state: [ XX   XXXX XXXXX XXXX XX  XXXX   XX   X XXX X   X ]
network state: [ XX XXXX X XXXXX XXXX X  XXXX X  XX   X XXXX    X ]
network state: [XX    XX XXX XX  XX   X  XXXXX   XX   X XXXX    X ]
network state: [XXX XXXX XXXX  X XX   XX XXX  XX XX   X XXXXX   XX]
network state: [XX X  XX XXX     XX   XX XX XX X XXX  X XXX X    X]
network state: [ XXXXXXX XXXX  X XX X XX  XXX XX XXX  X XXX X    X]
network state: [ X X  XX X X     XXXX XX  XX X X XXX  X XXX X    X]
network state: [ XXXXXXX X XXXXX XXXX X   X X X  XXX  X XXX      X]
network state: [ X X  XX X X XX  XXXX X   XX X   XXX  X XXX      X]
network state: [ XXXXXXX X XXXXX XXXX X   X X X  XXX  X XXX      X]
network state: [ X X  XX X X XX  XXXX X   XX X   XXX  X XXX      X]
network state: [ XXXXXXX X XXXXX XXXX X   X X X  XXX  X XXX      X]
network state: [ X X  XX X X XX  XXXX X   XX X   XXX  X XXX      X]
network state: [ XXXXXXX X XXXXX XXXX X   X X X  XXX  X XXX      X]
network state: [ X X  XX X X XX  XXXX X   XX X   XXX  X XXX      X]
network state: [ XXXXXXX X XXXXX XXXX X   X X X  XXX  X XXX      X]
network state: [ X X  XX X X XX  XXXX X   XX X   XXX  X XXX      X]
network state: [ XXXXXXX X XXXXX XXXX X   X X X  XXX  X XXX      X]
network state: [ X X  XX X X XX  XXXX X   XX X   XXX  X XXX      X]
network state: [ XXXXXXX X XXXXX XXXX X   X X X  XXX  X XXX      X]
network state: [ X X  XX X X XX  XXXX X   XX X   XXX  X XXX      X]
network state: [ XXXXXXX X XXXXX XXXX X   X X X  XXX  X XXX      X]
network state: [ X X  XX X X XX  XXXX X   XX X   XXX  X XXX      X]
network state: [ XXXXXXX X XXXXX XXXX X   X X X  XXX  X XXX      X]
network state: [ X X  XX X X XX  XXXX X   XX X   XXX  X XXX      X]
network state: [ XXXXXXX X XXXXX XXXX X   X X X  XXX  X XXX      X]
network state: [ X X  XX X X XX  XXXX X   XX X   XXX  X XXX      X]
network state: [ XXXXXXX X XXXXX XXXX X   X X X  XXX  X XXX      X]
network state: [ X X  XX X X XX  XXXX X   XX X   XXX  X XXX      X]
network state: [ XXXXXXX X XXXXX XXXX X   X X X  XXX  X XXX      X]
$ 
```

Figure 4.1: Simulation of a Boolean network of width N=50 and number of inputs K=2. Similar to in figure 2.2, each line represents one iteration and the state of each node in the network. A space-character and an “X”-character respectively signifies a 0- and 1-state.



```
$ ./main.rb  50 4
network state: [ X  XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX]
network state: [ X X X    XXXXXX X    X X  XXXX     XX  X    X XXX]
network state: [    XX X X XXXX  X     X X XXXXX X X  X  X X   XXX]
network state: [ X   XXX XX XXXXXX     XX   X  XXX    X XX X   X X]
network state: [XX X XX X X XXX    X XX  XXX  XXXX     XXXX  XX XX]
network state: [   X  XX  X X   XXXXX  X XX   X X XXXXX       X XX]
network state: [ X X X   XX XXXX X XX X    XXXX X  XXX XXX X X XXX]
network state: [X  XXXX   XXXX     XX   X  X    X   X X  X   XX   ]
network state: [ X    X  XXXXXXX X XX X XX XX  X X XXX X  XX XXX  ]
network state: [ X     XX  XXXXX X XX       XXX XX XXX  XX      XX]
network state: [X   XXX   X XXX  X X  X  X  X   X  X  X XX   X   X]
network state: [X        X  XXXX X XX X X  X XX XX   XX    X  X X ]
network state: [  XXXX  XX  X X  XXX   X     XX  X  X   XXXX XXXXX]
network state: [ X     X XX XXX  X XXXX X    XX    XX XXX       X ]
network state: [  X XXX X XXXX XXX  XXX   XXX XX    X X X  XX     ]
network state: [  X  X     X XX XX  X  XX   X  XX  XX X XX X X XX ]
network state: [XXXXXX   X XXXXXX  X XX XX X X  XX XX XXX X XXXX X]
network state: [    X  XX X XXXX X X  X     X   X  XXX  XX X     X]
network state: [X  XXXXXX XXXX   X X XX     X X XX  X   XX XX     ]
network state: [X    X XX X XXXXXXXXXXX   X X   X  XX   X X    X  ]
network state: [X X X X X XXXX   X    X  XX X       X   XX  XX   X]
network state: [X      XXXXXXXXXXX   XX   XX  XX   X    X  X  X XX]
network state: [  X XX     XXXX XX    X XXXXXXXXX  X  X  X   XXXXX]
network state: [  X XXXXX XXXX   X X  X  X XX         X  XXXX    X]
network state: [      X  X XX   XXXXX XX  X  X XX    XX X X      X]
network state: [XX   X  X  XX X XXX    X X  XXX   X XX XX  X   XXX]
network state: [XX  XX X   XXXXX   XXX     X XX XX XXXXXX   X X X ]
network state: [    XX     XXXX XX XX X  X  X     XXX X    X      ]
network state: [  X XXXX XXXXX X X   XX  XXXX XXX   XX   X X     X]
network state: [X    XX XXXXXX   X X   XX   X   X   X       XX X X]
network state: [XX   X XX XXXXXXX  XXXX  X X  X   XX   XX X   XXX ]
$ █
```

Figure 4.2: Simulation of a Boolean network of width N=50 and K=4, Note that this time there is no clear visible pattern in the network's behavior (the network does not stabilize for K $>$ 3, consistent with [2]).



```
load: Load typenetwork from file
(default net.gv)
ls: List all nodes
quick: Toggle drawing to terminal
reset: Reset nodes to a random
state
save: Write DOT graph to file
(default net.gv)
set: Set special mode
on/off/random. Example: set all
on, set all (reset all nodes), set all
random
show: Show the network
timer: Toggle timer
> genca 8 2
8x2D CA network generated
> genca 6 2
6x2D CA network generated
> genca 8 2
8x2D CA network generated
```

Figure 4.3: A Qt-based graphical user interface built to easier experiment with generation and simulation of networks. On the left is the on/off state of the network's nodes over time, and on the right a command-line interface for controlling the simulation.

As for the specific choice of interpreted language, the differences are less significant. The author has more experience with Ruby, and this made it the final choice.

4.1.2 Structure

The library is centered around a set of base classes suitable for derivation. BooleanFunction represents a single Boolean function holding an on/off state, accepting an arbitrary number of inputs and a function that can generate the next state based on the inputs. BooleanNetwork represents a Boolean network, holding a set of connected Boolean functions. While the functions allow different number of inputs, the network will for simplicity only allow a homogeneous number of inputs.

By sub-classing these base classes, the library allows for much customization. As a part of the thesis, used for experimenting, the following specializations of BooleanNetwork and BooleanFunction were created:

YeastNetwork In order to test the capabilities of the library, as well as verify the robustness of the Yeast Network[?], this special case Boolean network was created.

TypeNetwork/TypeFunction Because the networks used in the experiments mainly were built up of common Boolean operators such as AND/OR/XOR/etc, these classes were created to rapidly execute experiments with these types of networks.

TableFunction Often the behavior of a Boolean function is specified by a truth table. This class enables the BooleanFunction class to also have that capability.

4.1.3 Experiment module

As a part of the paper many experiments were to be carried out, and for this reason an experiment module was also created. This module handles the external parameters passed on to any experiment carried out, allowing the experiments to be reproduced, as well as the parameters themselves be logged.

All parameters have sensible default values, while additional ones may be specified through the command line.

4.2 Network parameters

Most experiments rely on randomly generated networks. The generation of these networks is in turn influenced by network parameters. These parameters are described in this section.

Note, the parameters listed here apply to the generation of the network as a whole, and the nodes it consists of.

4.2.1 Network size

The network size is the number of nodes in the network. While this is a parameter that should not directly influence the behavior of the network, it is important to make sure the conclusions drawn are consistent irregardless of network scale.

4.2.2 Node type

For many experiments, the type of nodes is randomly chosen from a small set of candidate types. This set may be supplied as the node type set, typically a subset of the available node types. The three candidate sets that were used in this paper are [And, Or, Xor], [Threshold-1, Threshold-2, ...][1], and [NThreshold-1, NThreshold-2, ...].

4.2.3 K (input count)

K is the number of inputs to each node (see [2] for more on how this affects Boolean networks). This can be be both uniformly or non-uniformly set, but for simplicity we will only consider the uniform case.

4.3 Simulation parameters

4.3.1 Updating scheme

The updating scheme is what specifies which nodes are updated at each iteration step under normal operation. Although many schemes exist, the most important distinction is that of synchronous versus asynchronous iteration (more in section 2.3.3).

4.3.2 Iteration count

After each network state reset, how many iterations are performed on the network. The definition of an iteration depends on the updating scheme.

4.3.3 Reset count

For a single network, how many times the state is reset to a random one.

[1]Specifically how many types are in these sets varies according to the input count in the network. For example, it does not make sense to allow a Threshold-4 node in a network with K = 3, because the number of active inputs would never pass the threshold, and the node would be in a constant state.

Chapter 5

Experiments

Using the simulation library, a series of experiments were performed that hopefully will collectively establish a basis for theories and improved understanding of Boolean networks. These experiments are centered around a set of 15 different network configurations that are related on some points. These network configurations were created by setting up many different permutations of the following parameters:

BN/CA Boolean network or Cellular Automata

#x#D Width and dimension count, 1 to 3, for CA

K# Input count, from 2 to 4, for BN

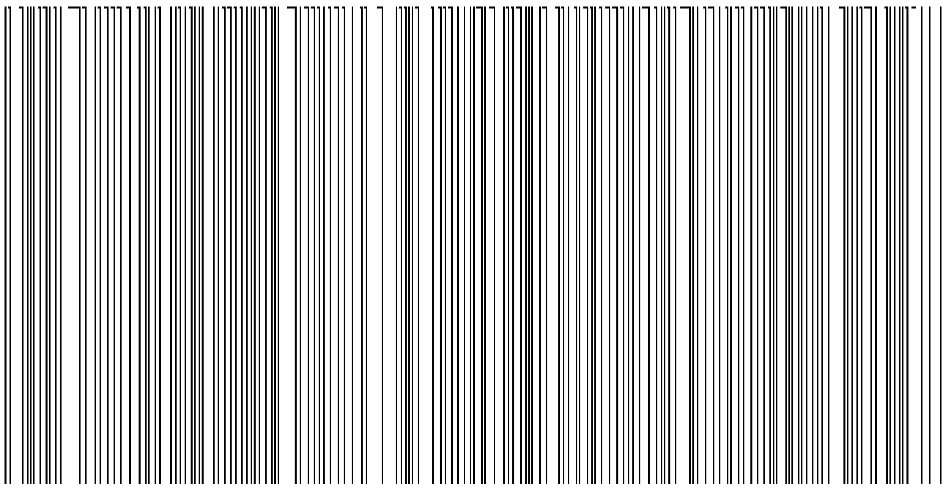
Threshold/NThreshold/AndOrXor Node types, for BN

In the experiments the configurations are denoted by a single dash-adjoined string. An example configuration is BN-K3-NThreshold for a Boolean network with input count K = 3 and negative threshold nodes, or CA-8x2D-Threshold for a two-dimensional Cellular Automata network of width 8, with threshold nodes.

5.1 Complexity in Wolfram's 1D CA

The functional complexity measure was proposed, but has not been tested. To see whether this metric reflects the qualities we are interested in, it was tested with the networks from Wolfram's 1D CA experiments[21]. These networks work by homogeneously assigning a single rule-set to all nodes, taking two adjacent nodes and the node itself as an input. Thus each node has 8 (2^3) different possible input value combinations. Then, for each combination the next state of the node may be 0 or 1, resulting in 256 (2^8) possible rule-sets. Simulation of two networks generated using this method is shown in Figure 5.1.

1118 networks evenly divided into the 256 different rules were generated, and for each network a functional complexity measurement using zlib was carried out. The


(a) Rule #68, showing a quickly stabilizing network.

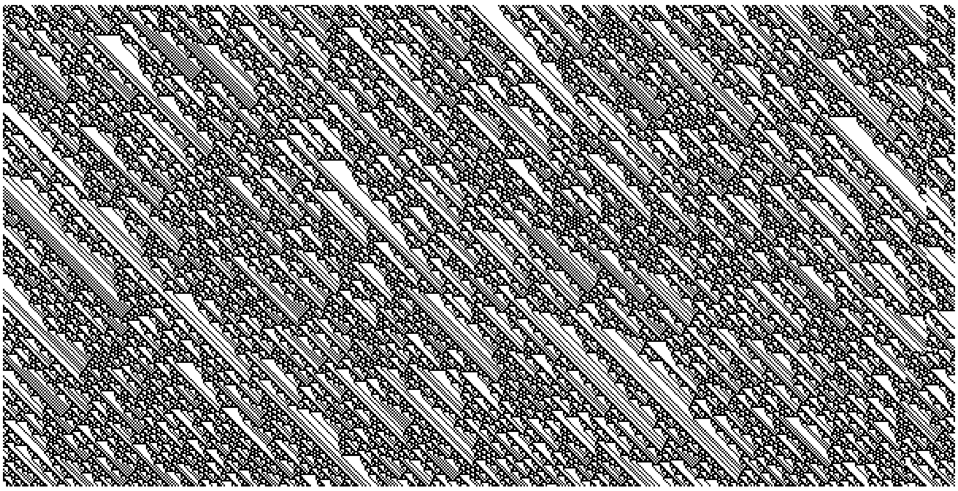

(b) Rule #120, showing a network with significantly complex interaction.

Figure 5.1: Two examples of running Wolfram's 1D Cellular Automata, generated using the Java applet at `http://www.cs.uic.edu/~wilkinson/Applets/automata.html`. Each CA is initiated to a random state (the first line at the top), and moving downwards the state of the network at each step can be seen changing. The most striking observation from this is the span in the complexity between the two, even though they are both based in similar simplistic rules.

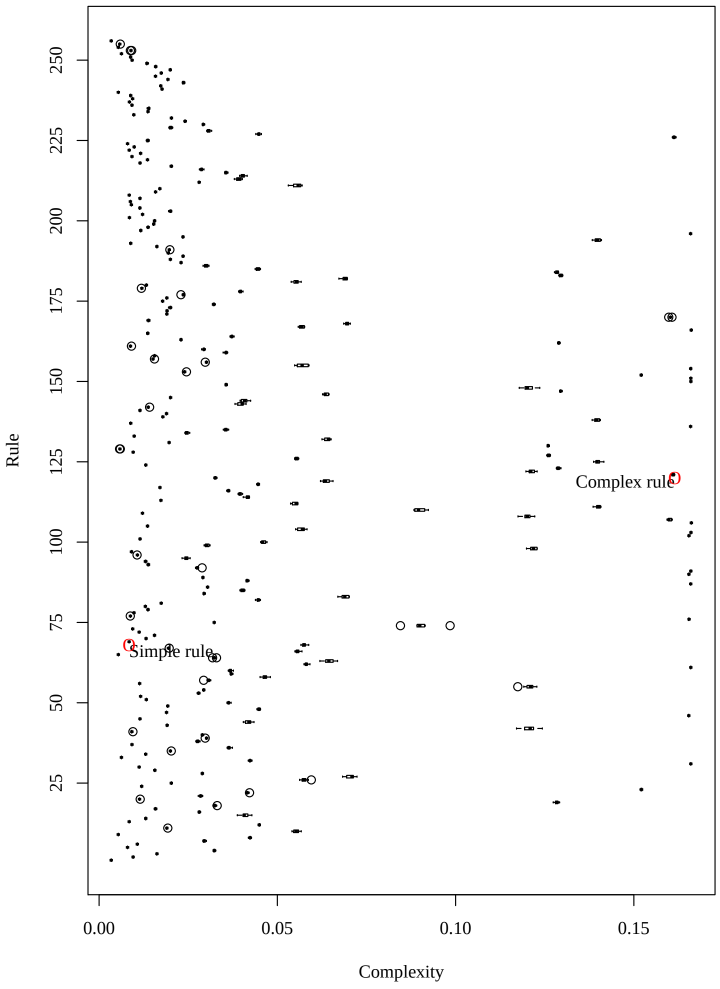


Figure 5.2: Wolfram complexity experiment

results are shown in Figure 5.2. Judging from the small height of the boxes in the box plot, one can see that the measure gives a consistent evaluation of the complexity. Also, the two networks in Figure 5.1 (68 and 120 on the vertical axis) have been marked in the figure. The simple rule has a low complexity value, while the complex rule has a high complexity value.

5.2 Basin evaluation

As an initial experiment, a simple count of the basins was performed. About 10 000 Boolean networks of 15 different sorts were randomly created. Next, the basin evaluation was carried out for each network, and the resulting number of basins counted. The results are shown in Figure 5.3a. While the Boolean networks mainly have low basin counts, the CA networks (in particular 1D and 2D CA) frequently touch the upper limit of basin counts (set to 200). This points to those networks having entered chaotic state space (leading to arbitrary end states and as a result large number of small basins).

In order to get rid of these uninteresting small basins, a second experiment (shown in Figure 5.3b) filtering out all basins smaller than a certain threshold, was carried out. This time it is clear that the CA networks in question do not produce any large basins: only a few outlier cases have basins larger than the threshold.

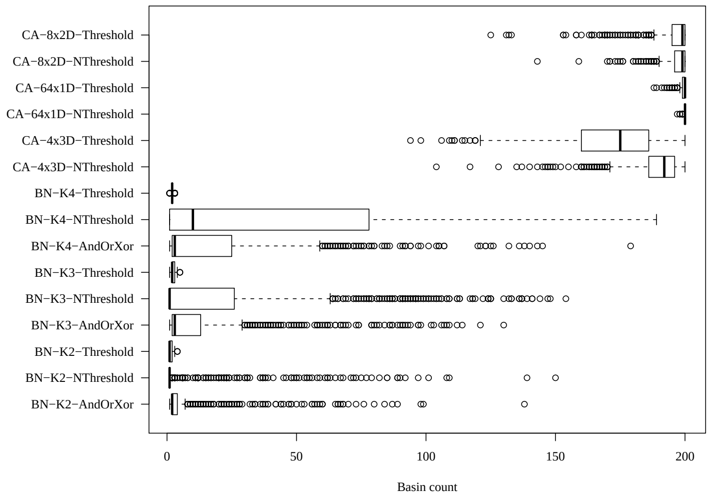
5.3 Intermediary state count

Next up is a look at the intermediary state count. Using the method from section 3.3.3, the results shown in Figure 5.4 were produced. In the same way that the basin evaluation shows networks that do not stabilize with very high numbers of basins, the intermediary state count shows these networks with the count approaching the upper limit of the metric.

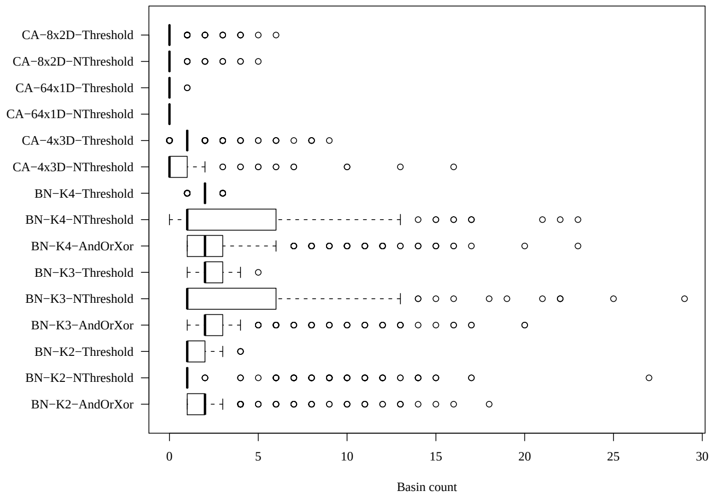
5.4 Functional complexity

Similar to basin evaluation and intermediary state count, also the functional complexity was measured for all the network types. The measurement was executed using zlib as described in section 3.3.2, and the results are shown in Figure 5.5.

The most noticeable observation is the large difference between CA and BN network types; the Boolean networks in general have higher functional complexity than CA irrelevant of their node types.



(a) Basin count



(b) Filtered basin count. All basins of smaller size than 5 were excluded in the count.

Figure 5.3: The basin count experiment. About 700 random networks of each type was generated, and the number of basins generated by the basin evaluation (section 3.3.1) is counted.

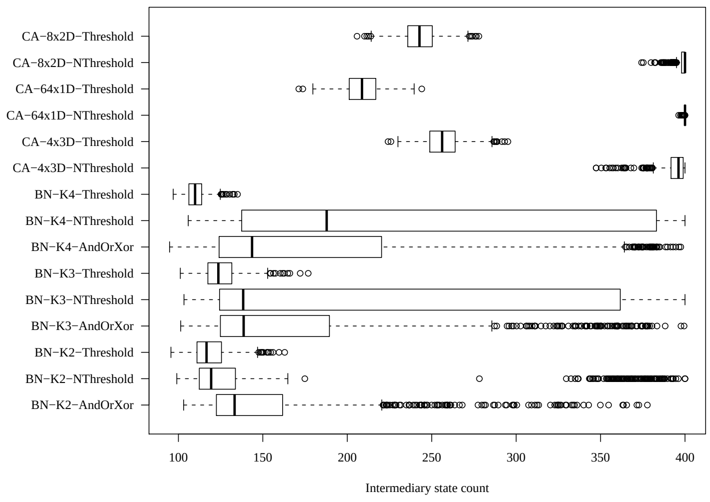


Figure 5.4: Intermediary state count. Each network was iterated for a maximum of 400 iterations, and the stabilization threshold (how many iterations the network must stay unchanged to be deemed stable) was at 60 iterations. This threshold will also thus act as a lower bound; there is no network that achieves a lower intermediate state count than 60.

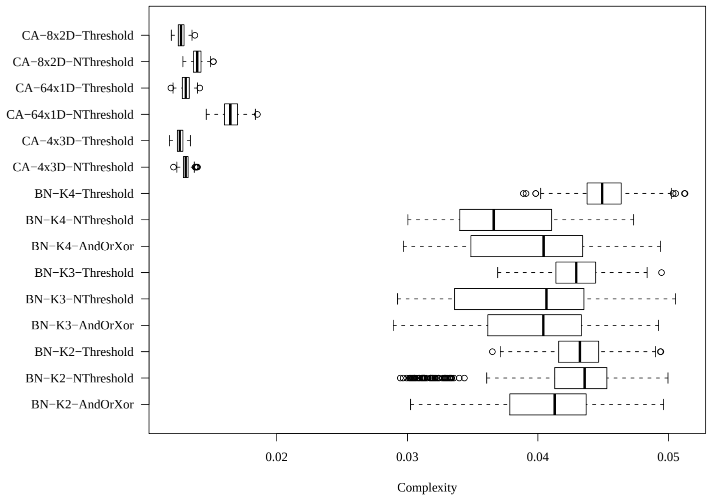


Figure 5.5: Functional complexity. The horizontal axis denote the deflation factor of the zlib compression, where higher values represent higher functional complexity.

5.5 Intermediary state count versus functional complexity

While intermediate state count and functional complexity are interesting metrics by themselves, it is natural to question the existence of a relation between them. That was the purpose of this experiment.

The various usual types of networks this time have their intermediary state count and functional complexity measured, and are plotted in Figure 5.6.

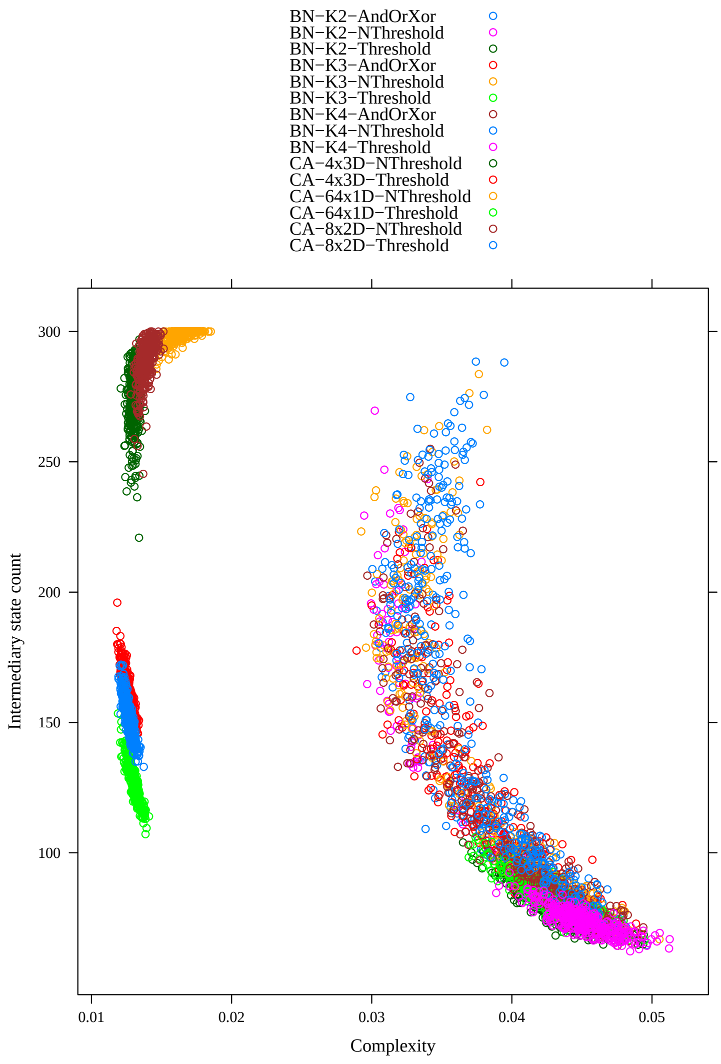


Figure 5.6: Intermediate state count versus functional complexity. The network types are divided into mainly three groups: on the right the BN types, on the upper left the negative threshold CA and lower left threshold CA. While the intermediary state count and the functional complexity measures alone correspond to the previous experiments, the combination of these shows interesting trends. CA networks mostly have the same functional complexity, and the intermediary state count is only slightly more varied. BN networks, however, show a tendency for highly complex networks to have low intermediary state count, but as the complexity decreases, the intermediary state count increases.

Chapter 6

Discussion

Next, the results from the experiments are discussed. General trends and theories that can be derived from the observations are discussed in 6.1, and the scope and limitations that one needs to be aware about are discussed in 6.2.

6.1 General observations

6.1.1 Network structure influence on network behavior

Compared to properties such as BN input count, CA dimension count, or node types, the type of network structure had considerably more influence on the behavior of the networks in the various experiments. This is seen from the large difference in BN and CA in most of the experiments. Even when both the BN and CA were using the same threshold nodes, the gap was significantly large.

6.1.2 BN in general exhibits more complex behavior

The biggest difference between BN and CA is how they are connected: BN can be freely connected, while CA has a fixed, homogeneous connection scheme between all nodes. This gives BN an extra "degree of freedom" in the building of networks, and this is visible in the experiments: In both the intermediary state count experiment and the functional complexity experiment, the spans of the sample points are consistently larger in the BN case than the CA case.

6.1.3 Dimension count versus input count

Dimension count (1D, 2D, 3D) and input count (K=2, K=3, K=4) are related in the way that they define the density of the connections in the network. For the basin count experiment, increasing these values lead to more basins. This is also true for the intermediary state count: the networks spend more time stabilizing. However, for the functional complexity the reverse trend is actually observed: for increasing values of dimension/input count the functional complexity decreases.

6.2 Limitations

Boolean networks in general encompasses a wide range of network structures, with a likewise wide range of differing characteristics. Iconic of this is the method development in this field is usually done at this time: by searching through an exceedingly large fitness landscape and progressively narrowing down the number of paths traversed. There are simply so many possible permutations of networks that an exhaustive is impossible in any practical sense.

In a similar fashion, while a complete overview of this landscape is desired, certain limitations had to be applied in order to keep the scope of the research manageable. In this section these limitations are discussed.

6.2.1 Network structures

Only N-dimensional CA networks and RBN networks were considered, but there are other connection schemes that could be tested as well. For example, a locality threshold that only connects nodes that are (assuming they're positioned in some N-dimensional space) close to each other, could allow the creation of networks that have the same locality and "change propagation" properties (in the sense that a state changing on one node will not affect other parts of the network until it has propagated that far) of the CA, while still permitting the freer connectivity of the BN scheme.

6.2.2 Node types

In Wolfram's experiments with rule sets for 1-dimensional Cellular Automata shows, small changes in the truth tables make large impacts to the functional complexity of the resulting network. While it is reasonable to believe that this also holds for other types of networks, and that followingly many different node types must be evaluated for these as well, nevertheless for simplicity a few representative ones were chosen.

6.2.3 Feedback loops

The definition of stability used so far in this paper defined stable state as one that does not allow any changes in states. However, certain definitions also include having entered a feedback loop as a stable state. This means that although nodes are still changing states, they repeatedly at some point in time go through the same state.

Efficiently detecting whether one has entered such a feedback loop is considerably more challenging than the opposite case (which is simply checking if the previous states are equal to the current one). In addition, detecting them in the case of asynchronous networks is also magnitudes more complex. For these reasons, feedback loops were not considered a stable state.

Chapter 7

Conclusion

In this paper, methods for evaluating behavior of Boolean networks were proposed. The basin evaluation allows a view of the different end states that a certain network ends in. The intermediary state count measures how long it takes the network to stabilize. The functional complexity measures how complex the iteration in the network is.

Using these measures, a set of 15 different network configurations were tested and discussed. CA networks were observed to have much less functional complexity than BN, and in general BN seems to have more potential for complex behavior. It was also observed that for increasing values of dimension count/input count the functional complexity decreases.

7.1 Future work

The Limitations section of the previous chapter describes the many constraints that were applied throughout the research of this paper. A perhaps obvious continuation to the work would then be to remove these constrains and broaden the set of network parameter permutations in which the experiments were executed.

There are also, however, areas that were excluded due to time pressure. The most significant of these is the experiments including noise simulation: The simulation library allows nodes to have a failure probability assigned, and for every update this dictates whether the node's update should be work as expected, or simulated failed. An experiment applying this setting to the nodes at different probabilities was executed, but there was not enough time to include the results here. Because the program for the experiment is included, this experiment is left as future work.

Bibliography

[1] S.A. Kauffman, Self-organization and adaptation in complex systems, in: The Origins of Order: Self-Organization and Selection in Evolution, Oxford University Press, New York, 1993, pp. 173–235.

[2] C. Gershenson, Introduction to Random Boolean Networks, 2004

[3] Francis Heylighen, Center "Leo Apostel", Free University of Brussels, Belgium: The Science of Self-organization and Adaptivity

[4] M. Gardner, “The fantastic combinations of John Conway’s new solitare game “life””, Sci. Am. 223(4), 1970

[5] E.F. Codd, “Cellular Automata”, Academic Press, 1968

[6] Amdahl, G.M.. “Validity of single-processor approach to achieving large-scale computing capability”, Proceedings of AFIPS Conference, Reston, VA., pp. 483-485, 1967

[7] Gustafson, J.L., “Reevaluating Amdahl’s Law”, CACM, 31(5), pp. 532-533., 1988

[8] D. Floreano, C. Mattiussi, “Bio-inspired artificial intelligence”, Ch. 7 Collective systems, pp 515-584, 2008

[9] D. Floreano, C. Mattiussi, “Bio-inspired artificial intelligence”, Ch. 5 Immune systems, pp 335-396, 2008

[10] C. C. Santini, G. Tufte, P. Haddow, “Bio-inspired Reverse Engineering of Regulatory Networks”

[11] C. Erbas, A. D. Pimentel, M. Thompson, S. Polstra, “A Framework for System-Level Modeling and Simulation of Embedded Systems Architectures”, 2007

[12] D. Thomas, W. Luk, “FPGA Accelerated Simulation of Biologically Plausible Spiking Neural Networks”, 2009

[13] S. Gobron, F. Devillard, B. Heit, “Retina simulation using cellular automata and GPU programming”, 2007

[14] S. Borkar, ”Designing Reliable Systems from Unreliable Components: the Challenges of Transistor Variability and Degradation”, 2005

[15] S. Huang, “Gene expression profiling, genetic networks, and cellular states: an integrating concept for tumorigenesis and drug discovery”, Journal of Molecular Medicine, vol. 77, no. 6, pp. 1432 - 1440, 1999

[16] ”Molecular devices and machines”, V. Balzani, A. Credi, M. Venturi, Nano Today, 2007

[17] Kitano, 2004 H. Kitano, Biological robustness, Nature Reviews Genetics 5 (2004), pp. 826–837

[18] Kauffman, S.A., Metabolic stability and epigenesis in randomly constructed genetic nets, J. Theo- retical Biology, Vol. 22, 437-467, 1969.

[19] Li F, et al. The yeast cell-cycle network is robustly designed. Proc. Natl Acad. Sci. USA (2004)

[20] Lehre, P., Haddow, P.C.: Developmental mappings and phenotypic complexity. Proceeding of CEC 2003, 62–68 (2003)

[21] Wolfram, S. (2002). A New Kind of Science, Champaign, IL: Wolfram Media, Inc.

[22] Leclerc, R. D.: Survival of the sparsest: robust gene networks are parsimonious, Mol Syst Biol. 2008

[23] Wagner A (1996.) Does evolutionary plasticity evolve? Evolution 50: 1008–1023.