

A Study and Comparison of First and Second Order Cellular Automata with Examples

Lauritz Vesteraas Thaulow

Master of Science in Mathematics
Submission date: June 2010
Supervisor: Nils A. Baas, MATH

Problem Description

First give a general introduction to cellular automata, then implement and discuss some examples. Introduce higher order cellular automata, examine whether and how the examples can be extended by applying 2-dynamics and/or 2-morphology, and study the effects.

Preface

This diploma thesis is the conclusion of my masters degree at the Norwegian University of Science and Technology.

I would like to thank my supervisor, Professor Nils A. Baas for his help and input, and my fellow master student Pål Davik, who has been writing a diploma on the same subject this year, for inspiration and support.

Trondheim, June 2010
Lauritz Vesteraas Thaulow

Abstract

This thesis will give an introduction to the concepts of cellular automata and higher order cellular automata, and go through several examples of both. Cellular automata are discrete systems of cells in an n -dimensional grid. The cells interact with each other through the use of a rule depending only on local characteristics, which lead to some global behaviour. Higher order cellular automata are hierarchical structures of cellular automata with added possibilities for dynamic local interaction.

We first give an introduction for non-mathematicians. A mathematical definition of cellular automata follows, and we illustrate the many possibilities with a few examples.

Higher order cellular automata are introduced and defined, and we look at the consequences higher order cellular automata has on optimization of computer implementations. Finally we apply higher order structures to some of the examples, and study the effects.

Contents

Notation and Abbreviations	1
Introduction	3
1 Cellular Automata	5
1.1 Introduction	5
1.1.1 A kind of machine	5
1.1.2 Rule 30 from the inside	6
1.1.3 Giving names to the concepts	7
1.2 History	10
1.2.1 A self-reproducing automaton	10
1.2.2 The Game of Life	10
1.2.3 Stephen Wolfram	11
1.3 Formal definition	12
1.3.1 Cellular automata	12
1.3.2 The local transition function	12
1.3.3 The global function	12
1.3.4 Reversibility, support and the Garden of Eden	13
1.3.5 Neighbourhoods	14
1.3.6 Wolfram numbering	14
1.4 Examples	18
1.4.1 1-dimensional binary cellular automata	18
1.4.2 Other 1-dimensional cellular automata	23
1.4.3 Activation-inhibition model	27
1.4.4 The Game of Life	30
1.4.5 The hodgepodge machine	33
1.4.6 WATOR-World	37
2 Higher Order Cellular Automata	43
2.1 Introduction	43
2.2 Definition	44
2.2.1 2-dynamics	44
2.2.2 2-morphology	45
2.3 Problems of optimization	47
2.4 Examples	49

2.4.1	1-dimensional cellular automata	49
2.4.2	The Game of Life	54
2.4.3	Hodgepodge and activation-inhibition	57
2.4.4	WATOR-World	58
3	Conclusions	63
	References	66
A	Application documentation	67
A.1	General documentation	67
A.2	Prerequisites	67
A.3	Vector graphics cellular automata	68
A.4	CA Explorer	68
A.5	Game of Life	69
A.6	Hodgepodge	69
A.7	WATOR-World	70
A.8	HOCA	70
B	Radius 1 binary 1-CA	71
	Index	75

Notation and Abbreviations

Throughout this paper, the notation listed below will be used.

S, N	Upper-case letters denote sets
\mathcal{N}, \mathcal{R}	Script upper-case letters are mostly used to denote sets of sets
\mathbf{a}, \mathbf{n}	Bolded letters are maps
c, s	Lower-case letters are single values or functions
\vec{z}	\vec{z} is a vector and z_i is its i th axial component
(a, b)	(a, b) is a vector or tuple with each axial component explicitly specified, and is equal to $\begin{bmatrix} a \\ b \end{bmatrix}$
$\{a, b, c\}$	$\{a, b, c\}$ is a set with the elements a, b and c
\mathbb{Z}	The set of all integers
\mathbb{Z}^+	The set of non-negative integers $\{0, 1, 2, \dots\}$
\mathbb{Z}_n	The set $\{0, 1, 2, \dots, n-1\}$
S^n	The set $\underbrace{S \times S \times \dots \times S}_{n \text{ times}}$
$\mathcal{P}(S)$	The set of all finite subsets of S .
$ S $	The cardinality of the set S , $\text{Card}(S)$
$\lceil x \rceil$	The nearest integer of x
$\lfloor x \rfloor$	The integer part of x
$\vec{a} + \vec{b}$	Vector addition, equals $(a_0 + b_0, a_1 + b_1, \dots, a_{d-1} + b_{d-1})$ for d -dimensional vectors \vec{a} and \vec{b} .
$\ \vec{z}\ _1$	The manhattan norm, $\ \vec{z}\ _1 = \sum_{i=1}^m z_i $ for an m -dimensional vector \vec{z} .
$\ \vec{z}\ _\infty$	The maximum norm, $\ \vec{z}\ _\infty = \max\{ z_i \mid i \in \{1, \dots, m\}\}$ for an m -dimensional vector \vec{z} .
$g \circ h$	The function g composed with the function h , so that $(g \circ h)(x) = g(h(x))$.

And these are the most commonly used abbreviations:

CA	cellular automaton
HOCA	higher order cellular automaton
LTF	local transition function
d-CA	d -dimensional cellular automaton
IBM	individual-based model
CPU	central processing unit
RAM	random access memory

Introduction

How does complexity arise from simple beginnings? The universe was born, according to astrophysicists, from extremely uniform and well-ordered beginnings, yet the result is staggeringly complex. Smoke rising from a blown out candle is orderly and simple, until it suddenly becomes chaotic and unpredictable.

New Scientist wrote in its article *Seven questions that keep physicists up at night* [17]:

From the unpredictable behaviour of financial markets to the rise of life from inert matter, Leo Kadanoff, physicist and applied mathematician at the University of Chicago, finds the most engaging questions deal with the rise of complex systems. Kadanoff worries that particle physicists and cosmologists are missing an important trick if they only focus on the very small and the very large. "We still don't know how ordinary window glass works and keeps it shape," says Kadanoff. "The investigation of familiar things is just as important in the search for understanding." Life itself, he says, will only be truly understood by decoding how simple constituents with simple interactions can lead to complex phenomena.

Cellular automata are prime examples of systems with simple constituents and simple interactions which lead to complex phenomena, and they are excellently suited for detailed study and analysis on exactly how it happens, because they can be simulated easily in a computer.

In this thesis we will use the first chapter to introduce, define and give examples of cellular automata, and show how the very simple can quickly become very complex, even though every step on the way is obvious and simple. We will also give examples of cellular automata that simulate processes from real life, some with very accurate results.

In chapter 2 we will introduce the concept of higher order cellular automata. We will consider what consequences adding higher order structures and 2-dynamics have on our ability to optimize computer implementations of such cellular automata. Finally, we will apply higher order structures or 2-dynamics to some of the examples from chapter 1, and consider the effects.

Chapter 1

Cellular Automata

1.1 Introduction

The concept of a cellular automaton is rather simple. However, the use of mathematical notation and lingo in this and other papers, makes it unnecessarily hard for non-mathematicians to grasp. This section is therefore written in a more informal language, as a gentle introduction on the subject, for those who have little or no higher mathematical education.

1.1.1 A kind of machine

A cellular automaton is like a machine – you can put something into it, and it will use that input to produce some output. In our case, this “machine” is going to take rows of black and white squares as input, and produce an equally long row of black and white squares as output.

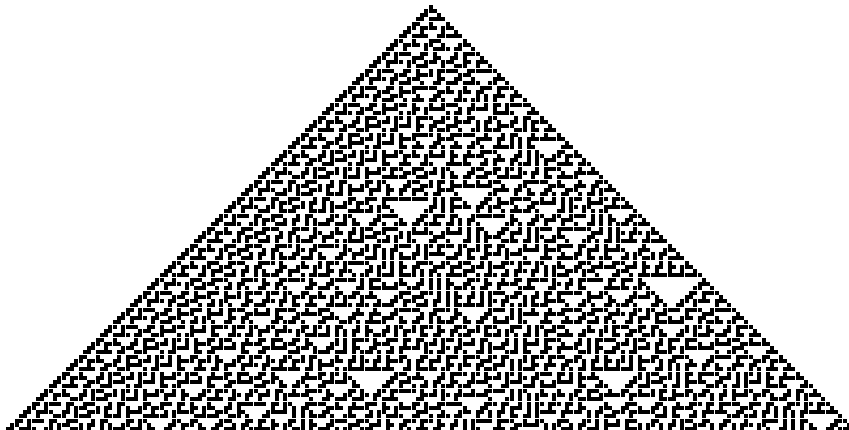


Figure 1.1: A cellular automaton

Figure 1.1 is an image produced by a cellular automaton called *rule 30*. The topmost row, with a single black square, is used as the input to rule 30, and it produces the second row as output. We then use that row as a new input to the same rule, and get the third row as output, and so on.

Rule 30 is just one example of a cellular automaton. They come in a huge variety of shapes and sizes. This text will first tell you how to make such patterns of your own, using only pen and paper. Then, for those who wish to read more on the subject, an introduction to the lingo of cellular automata will be given, based on the concepts we've used.

1.1.2 Rule 30 from the inside

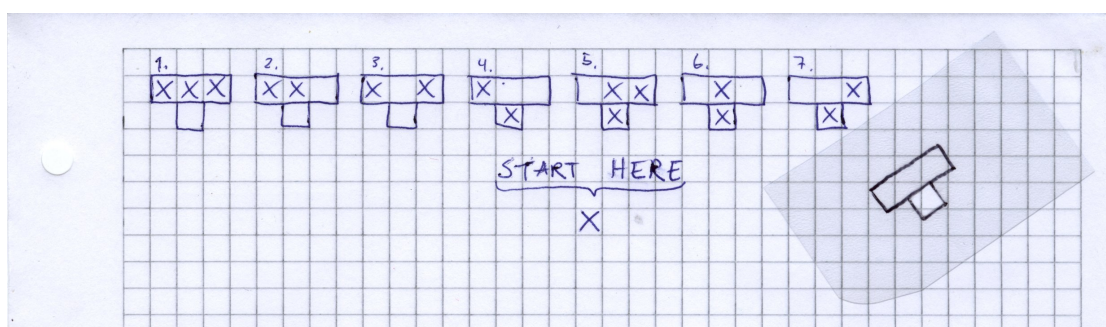






Figure 1.2: The instructions

Imagine you were given a sheet of graph paper as in 1.2. The paper contains a row of 7 instructions, which we will use to create the same pattern as in figure 1.1. This means, of course, that the 7 instructions correspond to rule 30, but we need to understand how to use them.

To that end, you've been given a loose piece of transparency (also shown in figure 1.2), on which a figure has been drawn in black ink. Inside the smallest box a hole has been cut through the plastic. The piece of transparency can be moved around, so imagine that you place it as in the left image of figure 1.3. Now notice that the pattern in the three squares outlined in black matches those in the instruction numbered 7 in figure 1.2. However, that drawing also has a cross in the square hanging underneath, so they do not match completely... yet. The transparency has a convenient hole where the missing cross should be, so we draw an  there to make them equal.

Continuing, we move the transparency one square to the right, as shown to the right of figure 1.3, and observe that the upper three squares now has content matching instruction 6. This also has an  in the square underneath, so we draw an  where the hole is, to make them similar. Finally we move the transparency to the right once more and use instruction number 4 to place another  on the graph paper, which is now looking like in figure 1.4 to the left.

For the next row, we'll need to use instructions 7, 5, 1, 2 and 4, and this will produce

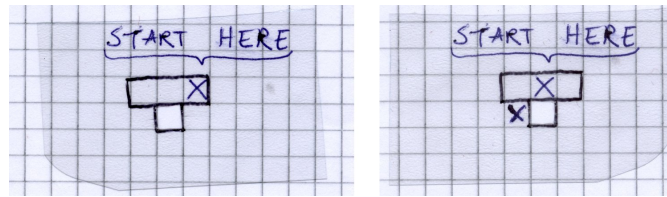


Figure 1.3: Using the piece of transparency.

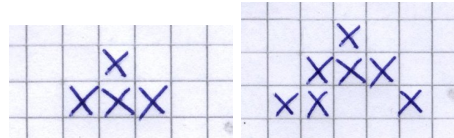


Figure 1.4: The second and third row.

a drawing like the one to the right in figure 1.4. Continuing like this for a while we'll get a drawing like the one in figure 1.5. Notice that the order in which we “calculated” the rows of squares did not matter, we could have done them right-to-left, or in an arbitrary order, instead of left-to-right.

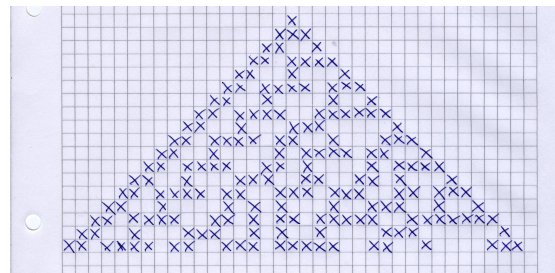


Figure 1.5: After 18 rows.


We can now easily see the similarity between our drawing and figure 1.1. As indicated by the number 30 in rule 30, it is but one of many such rules¹.



1.1.3 Giving names to the concepts



We've now presented the basic parts of a cellular automaton, but we still have not written down exactly what it is. A cellular automaton is made up of exactly four pieces of information:

- An *alphabet*: the different symbols we can use; for us it was only two, \square and \boxtimes .
- A *lattice*: the rows of squares that we draw on.

¹The logic behind the numbering is explained in section 1.3.6

- A *neighbourhood*: the three squares that were outlined by our transparency, and how they're positioned relative to the hole we may draw an  in.
- A *function* or *rule*: tells us what symbol to draw in every case, like our 7 instructions.

Each square is called a *cell*, and each symbol, like  and , is called a *state*. We say that the cell *has* a state. The cellular automaton we've drawn is called a 1-dimensional cellular automaton, even though the drawing we've made fills a two-dimensional grid. This is because we regard the vertical axis as *time*, with time increasing downwards. So just like we live in a three-dimensional world with time, this cellular automaton “lives” in a 1-dimensional world with time. Each row is then the whole world at one particular time, this world is what we call the lattice. At each time step, the lattice has a particular pattern of states, and each such pattern is called a *global state* or *configuration*.


The three cells in a row that was our neighbourhood can contain eight different patterns² of 's and 's, and each of these patterns we will call a *neighbourhood state*. It might be tempting to call it a neighbourhood configuration instead, but this term will be needed for something else later.

The neighbourhood concept is perhaps a little difficult to grasp, since it may seem to be just a part of what makes up a rule. It really isn't, though the two concepts depend on each other to some degree. In our example, the neighbourhood was the three cells in a row, centered on the one we were to find the new state of. However, we could have changed the neighbourhood while keeping the rule by for example shifting the three cells to the right or left, or even by spreading them out so that they are neither adjacent nor symmetric³. The rule needs to be compatible with the neighbourhood though; using our seven instructions with a neighbourhood consisting of five cells would make no sense, since there'd be no matching instructions for the neighbourhood states we'd encounter.

Neighbourhoods come in many different shapes and sizes, although some are more commonly used than others. We say that ours had a *radius* of 1, since it included 1 cell on each side of the cell we were finding the next state for.

We have not talked about what happens when the pattern reaches the edges of the paper, or lattice. Throughout this thesis we will mostly use lattices that wrap. This means that what goes out on one side, comes in on the other. A way of visualizing this is of taping the left and right edge of the grid paper together to form a cylinder. Then there'd no longer be any edge, and the transparency would be usable all the way around, as illustrated in figure 1.6.

We've now covered the basic concepts that make up a cellular automaton. These can be built upon and extended to make things more interesting. We may add another dimension, for example, and get a 2-dimensional cellular automaton. To draw such a thing with pen and paper, you would need a book of graph paper, one page for each time

²I purposefully left out one instruction from those in the beginning of this section, which is the instruction for when there is no 's in any of the cells. This did not matter for that rule, since it implicitly said to leave such cells alone.

³We'd need to imagine them as being adjacent to match them up with the instructions.

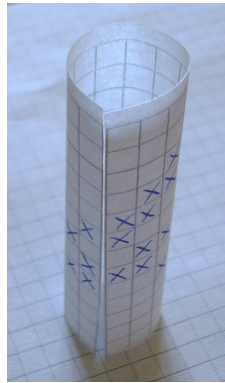


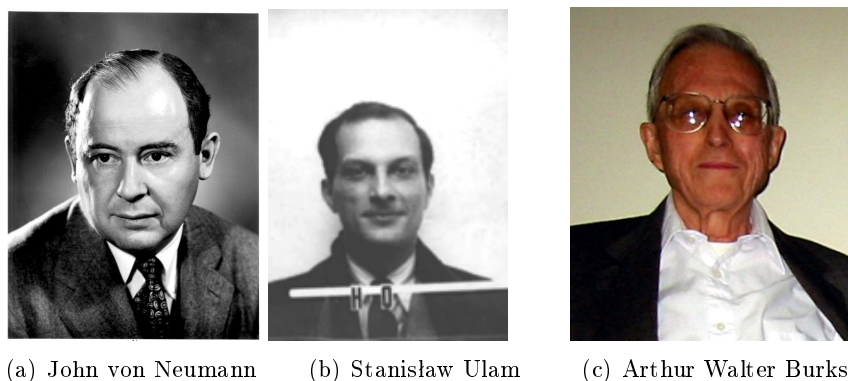
Figure 1.6: What happens on the edges.

step. You would also need a set of instructions for a 2-dimensional cellular automaton, and an initial pattern on the first page. After drawing some pages, you could look at your animation by flipping through the pages.

Animations and java applets depicting various cellular automata can be found on the internet, and they are often fascinating to watch. As a starting point, here are some google searches that gives nice results: “Game of Life”, “forest fire cellular automaton”, “hodgepodge machine”, “cyclic space” and “traffic cellular automaton”.

1.2 History

1.2.1 A self-reproducing automaton



(a) John von Neumann (b) Stanisław Ulam (c) Arthur Walter Burks

Figure 1.7: Important figures from the history of cellular automata. These pictures are in the public domain.

One of the (many) interests of the Hungarian-born mathematician John von Neumann was how to create self-replicating automatic factories. At first he considered how to make such a thing in practice, using robotics or toy mechanic sets, but he soon found the problem too complex. After conferring with Stanisław Ulam on the matter, he decided to follow his friends advice and instead consider a simple abstract mathematical model [19, p. 876]. He then devised what is now known as cellular automata (CA), and outlined a particular kind of 2-dimensional CA that would be capable of self replication if set up in a certain way. It was a rather complicated one; each cell could be in one of 29 different states, with each state having a certain meaning. He went on to outline an initial configuration consisting of 200 000 cells that he claimed would be able to make copies of itself.

His work remained unpublished until his death in 1957, but it was edited and published in 1966 by Arthur Walter Burks [18]. The subject of self-reproducing configurations of cellular automata caught the interest of others, and simpler examples were soon found; Edgar Frank Codd worked out a variant of von Neumann’s original example that needed only eight states [11, p. 1]. Also, during the 1960s various theorems related to the formal computational capabilities of cellular automata were proved.

1.2.2 The Game of Life

The field of Cellular Automata was given its poster child in 1970, when Martin Gardner⁴ presented some results of John Conway in his column *Mathematical Games* in Scientific American. Gardner called the game “Life” , and it was therefore subsequently known as Conway’s Game of Life. Many readers of the column were inspired to do more or

⁴Martin Gardner died just 10 days prior to the deadline for this thesis, on May 22, 2010.



Figure 1.8: More important figures from the history of cellular automata. (a) Photograph by Konrad Jacobs, (b) photograph by Thane Plambeck, (c) photograph by Stephen Faust.

less serious research on the Game of Life, and for three years there was even a quarterly newsletter dedicated to discoveries about the behaviour of this particular cellular automaton.

Around the same time, microcomputers became available and affordable, and for many aspiring programmers, the Game of Life was the inspiration for the first applications they made. However, according to Stephen Wolfram, little of direct scientific value came of the whole thing:

“An immense amount of effort was spent finding special initial conditions that give particular forms of repetitive or other behaviour, but virtually no systematic scientific work was done (perhaps in part because even Conway treated the system largely as recreation), and almost without exception only the very specific rules of Life were ever investigated.” [19, p. 877]

1.2.3 Stephen Wolfram

Apart from his contributions to the field of cellular automata, Stephen Wolfram is today best known as the father of the Mathematica computer algebra system and the Wolfram|Alpha⁵ computational knowledge engine. Back in 1981 he started studying cellular automata, and instead of concentrating on one particular cellular automaton, as von Neumann and the Game of Life enthusiasts had done, he undertook a sweeping survey of many kinds of cellular automata, and devised a way to classify any cellular automaton into one of four classes.

Stephen Wolfram went on to write the controversial [15] 1200 pages long *A New Kind of Science* [19] in 2002. There he claims that what he calls *simple programs*, of which cellular automata is one example, will be the basis for a whole new approach and methodology of science within a number of different scientific fields.

⁵<http://www.wolframalpha.com/>

1.3 Formal definition

There is no agreed-upon standard formalisation or notation for cellular automata. The notation used in this thesis is based on the formalisations of Delorme and Mazoyer [5], but with modifications to allow for later adaption to higher order cellular automata, and some other small changes that was deemed appropriate.

1.3.1 Cellular automata

We need to define a few concepts to start with. A *lattice* is an ordered grid, typically \mathbb{Z}^d , and we denote it \mathcal{L} . A *neighbourhood* is a finite ordered subset of \mathcal{L} , and we denote it by N . A *cell* is an object which has one *state* at a time. The set of all allowable states is a finite set called the *alphabet*, denoted by S . There's one cell at each node of the lattice.

Definition. A d -dimensional *cellular automaton* \mathcal{A} (a d -CA) is a 4-tuple (\mathcal{L}, S, N, f) , where \mathcal{L} is a lattice, S is an alphabet, N is a neighbourhood, and f is any function $f : S^{|N|} \rightarrow S$.

The alphabet S is very often the set $\{0, 1, \dots, n-1\}$ or some variation thereof, though this is by no means mandatory. Among the states are, sometimes, distinguished states s , called *quiescent states*, such that $f(s, \dots, s) = s$.

For the rest of section 1.3, we'll let (\mathcal{L}, S, N, f) be a general d -CA unless otherwise specified.

1.3.2 The local transition function

The function $f : S^{|N|} \rightarrow S$ is called a local transition function, abbreviated *LTF*. The set $S^{|N|}$ is the set of all possible neighbourhood states. Each element of $S^{|N|}$ is an ordered set of states $\{s_0, s_1, \dots, s_{|N|-1}\}$. For some cellular automata, the LTF can be defined with a formula, for example $f(s_0, s_1, s_2) = s_0 + s_1 + s_2 \pmod{2}$. In this case, S is most often a subset of the integers (though it can be any finite set on which the necessary mathematical operations are defined).

In other cellular automata, the LTF is so unwieldy that there is no practical way to express it as a formula, and it may then be necessary to fall back to defining it by explicitly tabulating all the values in the domain and their associated value in the range. This is not always practicaly possible, however. Even though the domain has to be finite, it can be so large that no computer currently in existence could store a function on it in tabular form. Some of these types of cellular automata can fortunately be studied in spite of this, namely those whose LTF can be defined as a computer algorithm. Such an algorithm can be almost arbitrarily complex, and have a large number of parameters.

1.3.3 The global function

A cellular automaton evolves with time. The states of all the cells in the lattice are updated simultaneously, so that time is discrete. We define a function $\mathbf{a}^t : \mathcal{L} \rightarrow S$

that gives the state of each cell of the lattice at time t . We call \mathbf{a}^t the *global state* or *configuration* at time t , and denote the state of the cell at \vec{z} by $\mathbf{a}^t(\vec{z})$. The sequence $(\mathbf{a}^0, \mathbf{a}^1, \mathbf{a}^2, \dots)$ is called the *behaviour*, *action* or *evolution* of the cellular automaton.

For a cellular automaton $\mathcal{A} = (\mathcal{L}, S, N, f)$ with neighbourhood $N = \{\vec{n}_0, \dots, \vec{n}_{n-1}\}$, we define \mathbf{a}^{t+1} as the global state such that the following statement is true:

$$\mathbf{a}^{t+1}(\vec{z}) = f(\mathbf{a}^t(\vec{z} + \vec{n}_0), \mathbf{a}^t(\vec{z} + \vec{n}_1), \dots, \mathbf{a}^t(\vec{z} + \vec{n}_{n-1})) \quad \text{for all } \vec{z} \in \mathcal{L}.$$

We say that \mathbf{a}^{t+1} *follows* \mathbf{a}^t . We can construct a function G such that $G(\mathbf{a}^t) = \mathbf{a}^{t+1}$. We call G the *global function* of \mathcal{A} , as it maps each global state to the global state that follows it. Each cellular automaton has its own associated global function.

The domain of G is the set of all possible global states that the cellular automaton might have, which is equal to $S^{|\mathcal{L}|}$. We denote this set by C . The function G is obviously an endofunction of C , that is, $G : C \rightarrow C$.

In this thesis we'll adopt this convention: if we need to refer to multiple unrelated elements of the set C , we will use subscripted numbers instead of superscripted numbers. This is to distinguish global states that are part of the evolution of a cellular automaton from global states that are unrelated, or might be. For example, $\mathbf{a}^0, \mathbf{a}^1 \in C$ implies that $G(\mathbf{a}^0) = \mathbf{a}^1$, while $\mathbf{a}_0, \mathbf{a}_1 \in C$ does not put any restriction on the relationship between \mathbf{a}_0 and \mathbf{a}_1 .

1.3.4 Reversibility, support and the Garden of Eden

It is now natural to consider the properties of the global function. The configurations that are not in the codomain of G are called the *Garden-of-Eden configurations*, because they can only appear as initial conditions, and will never appear during the evolution of the cellular automaton. If a global function G is bijective, and if G^{-1} is the global function of some cellular automaton, we say that both are the global function of a *reversible* cellular automaton. It is easy to show that a reversible cellular automaton has no Garden-of-Eden configurations.

For a cellular automaton with a quiescent state s , we define the *support* of a configuration \mathbf{a} as follows:

$$\text{supp}(\mathbf{a}) = \{\vec{z} \mid \mathbf{a}(\vec{z}) \neq s\}$$

In words, the support of \mathbf{a} is the subset of \mathcal{L} that has cells with non-quiescent states. We now define a *finite configuration* as a configuration with finite support. This definition is used in the following theorem, due to Edward F. Moore and John Myhill:

Theorem 1.3.1 (Garden-of-Eden Theorem). *A cellular automaton with a quiescent state is surjective if and only if it is injective when restricted to finite configurations.*

For finite configurations, an injective global function is necessarily also surjective, because the global function must be defined for every element in the domain, and the domain and the codomain is the same size, since the global function is an endofunction.

Conversely, a surjective global function must be injective because the domain equals the codomain.

Another way of stating this theorem is that a finite cellular automaton with a quiescent state has a Garden-of-Eden configuration if and only if there exists two configurations \mathbf{a}_a and $\mathbf{a}_b \in C$ such that $G(\mathbf{a}_a) = G(\mathbf{a}_b)$.

1.3.5 Neighbourhoods

First we will address the question on whether the neighbourhood includes the cell whose neighbourhood it is ($\vec{0}$) or not. Different conventions exist [5, p. 7], but we choose in this thesis to use neighbourhoods that explicitly include the cell itself.

The most frequently used neighbourhood for one-dimensional cellular automata is referred to as the *first neighbours neighbourhood*. It consists simply of the central cell of the neighbourhood, plus its right and left neighbours: $\{-1, 0, 1\}$. Extending this concept, we get the *radius neighbourhood*, which we will denote N_r :

$$\{-r, -r+1, \dots, -1, 0, 1, \dots, r-1, r\}$$

For an m -dimensional cellular automaton where $m > 1$, there are two neighbourhoods that need to be mentioned. Recall⁶ that for a vector \vec{z} , we have the Manhattan norm $\|\vec{z}\|_1$ and the maximum norm $\|\vec{z}\|_\infty$. From these two we get the associated metrics d_1 and d_∞ , which we can use to define the *von Neumann neighbourhood* N_{vN} and the *Moore neighbourhood* N_M , respectively:

$$\begin{aligned} N_{vN}(\vec{z}) &= \{\vec{x} \mid \vec{x} \in \mathbb{Z}^d, d_1(\vec{z}, \vec{x}) \leq 1\} \text{ with a given order} \\ N_M(\vec{z}) &= \{\vec{x} \mid \vec{x} \in \mathbb{Z}^d, d_\infty(\vec{z}, \vec{x}) \leq 1\} \text{ with a given order} \end{aligned}$$

In 1 dimension, the von Neumann and Moore neighbourhoods are identical to the radius 1 neighbourhood. The 2-dimensional variants are depicted in figure 1.9. We need to give the elements of the neighbourhoods an order, as we've done for the first neighbours and radius neighbourhoods. Since we will only use the 2-dimensional variants, we will only specify the order for those. We'll let the first element be the origin $(0, 0)$, and then let the rest of the elements follow, starting with $(1, 0)$ and going in a counterclockwise direction.

1.3.6 Wolfram numbering

As mentioned in section 1.3.2, the domain of the local transition function $S^{|N|}$ is a set of ordered sets of the form $\{s_0, s_1, \dots, s_{|N|-1}\}$, where $s_i \in S$. If we give S a total order relation, we may order the set $S^{|N|}$ lexicographically in descending order, making it an

⁶See page 1.

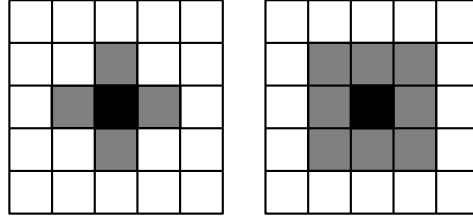


Figure 1.9: The 2-dimensional von Neumann (left) and Moore (right) neighbourhoods. The black cell is the central cell – the cell whose neighbourhood is depicted – and it is also part of the neighbourhood.

ordered set of ordered sets. For example, if $S = \mathbb{Z}_2$ and $|N| = 2$, we would order $S^{|N|}$ as follows: $\{\{1, 1\}, \{1, 0\}, \{0, 1\}, \{0, 0\}\}$.

Let η_i denote an arbitrary neighbourhood state, that is, $\eta_i = \{s_1, s_2, \dots, s_{|N|}\}$, and let $n = |S|^{|N|} - 1$. We may then write $S^{|N|} = \{\eta_n, \eta_{n-1}, \dots, \eta_1, \eta_0\}$. For any η_i , the number i uniquely identifies a certain neighbourhood state, since $S^{|N|}$ is ordered. If $S = \mathbb{Z}_m$ we may easily recover the neighbourhood state of any given η_i by writing i as a base m number with $|N|$ digits. Conversely, given any neighbourhood state we can calculate the index i by interpreting it as a base m number⁷.

We may use this to construct a canonical lookup table for an LTF. It will have the ordered set $S^{|N|}$ in the upper row, and the state that the LTF maps each neighbourhood configuration to in the row below, as in table 1.1. The bottom row can now be concatenated into a word $s_n s_{n-1} \dots s_1 s_0$, which uniquely⁸ identifies the local transition function, provided S is also known. We'll call this word the *explicit form* of the LTF.⁹

Neighbourhood state	η_n	η_{n-1}	\dots	η_0
New central state	s_n	s_{n-1}	\dots	s_0

Table 1.1: A canonical lookup table.

Stephen Wolfram formalized a compact way of labeling these rules which has become very popular. He observed that if $S = \mathbb{Z}_m$, the word $s_n s_{n-1} \dots s_1 s_0$ can be interpreted as a single number in base m . Using this convention, the number uniquely identifies the rule, provided that N and m is also specified.

⁷For $S = \mathbb{Z}_2$ and $|N| = 3$, $S^{|N|}$ would have $\eta_7 = \{1, 1, 1\}$, $\eta_6 = \{1, 1, 0\}$, $\eta_5 = \{1, 0, 1\}$, $\eta_4 = \{1, 0, 0\}$, $\eta_3 = \{0, 1, 1\}$, $\eta_2 = \{0, 1, 0\}$, $\eta_1 = \{0, 0, 1\}$, and $\eta_0 = \{0, 0, 0\}$. We may now easily verify that, for instance, 6 base 10 is equal to 110 base 2.

⁸Please note that the word we've defined here is in the opposite order of the word as defined in [5], where $S^{|N|}$ is sorted in ascending order. The way we've done it, the digits of the word are in the traditional order for a number, that of the most significant digit to the left, and can therefore immediately be read off as a base m number.

⁹The expressions “canonical lookup table” and “explicit form” are coined here by the author, as replacements for the generic terms “table” and “word” used in [5] to describe these constructs. This is to make it easier to refer back to these concepts later.

Neighbourhood state	111	110	101	100	011	010	001	000
New central state	0	0	0	1	1	1	1	0

Table 1.2: Rule 30 as a canonical lookup table.

Let us clarify these concepts by looking at an example. Take the rule from the introduction, rule 30, pictured in figure 1.1. It has $N = \{-1, 0, 1\}$ and $S = \{0, 1\}$, so $m = 2$. Its LTF is given as a canonical lookup table in table 1.2. The set $S^{|N|} = (\mathbb{Z}_2)^3$ is listed in the conventional order in the upper row, and the state that each neighbourhood is mapped to is listed below each neighbourhood state. The explicit form is then the word 00011110, and its base 10 equivalent is 30, which is why the rule is called “rule 30”. Some properties of the rule can be immediately read from the explicit form of the rule:

- The last digit is 0 so the neighbourhood $\{0, 0, 0\}$ will be mapped to 0. This means 0 is a quiescent state of this rule.
- The first digit is 0, so the neighbourhood $\{1, 1, 1\}$ will be mapped to 0. This means that the density of cells with state 1 in the lattice can not exceed a certain threshold for more than a single time step.

The explicit form quickly becomes unpractical for larger neighbourhoods and bigger alphabets, since the number would be too big to print¹⁰. In computers however, the explicit form is the most compact way of storing a generic LTF, and it has the added bonus that calculating the result of the rule given some neighbourhood state is only a matter of calculating which digit to look up.

A 3-state 1-CA with a radius 1 neighbourhood would have 27 different neighbourhood states in its lookup table, and therefore the equivalent of a 27-digit base 3 number in its second row. The largest 27-digit base 3 number is $222 \cdots 222 = 7.6 \cdot 10^{12}$, which is therefore also the total number of different rules that exist for this CA. We can create a formula for calculating this number for different S and N :

$$\Theta = |S|^{|S|^{|N|}}$$

The integer Θ can very easily be an extraordinarily large number. It can be argued that the rules and their properties are the primary object of study in the field of cellular automata, and this then presents a problem: they are very numerous. For example, a 2-dimensional binary cellular automaton with the Moore neighbourhood has $|N| = 9$ and $|S| = 2$, so $\Theta = 2^{2^9} = 1.3 \cdot 10^{154}$, which is a ridiculously large number¹¹.

¹⁰ In section 1.4.2 we will encounter a rule that would have been about 4932 digits long, written in explicit form.

¹¹For the sake of argument, suppose $1 \cdot 10^{24}$ of these rules exhibited some kind of revolutionary, hitherto unseen new behaviour. The chance of finding one of them if you picked a rule at random is about $1 \cdot 10^{-131}$. Let's say you used a computer that could search a million rules a second. The chance of finding one of them in 10 years of searching is still in the order of $1 \cdot 10^{-116}$. For comparison, the chance of winning the “guess which atom in the universe I'm thinking of” game is only $1 \cdot 10^{-81}$.

These overwhelming numbers motivates narrowing down the field of study somewhat, and the normal way of doing so is to replace the local transition function f with a composition of two functions, call them g and h , so that $f = g \circ h$. We first decide on a function h from $S^{|N|}$ to some smaller set, let's call it P , and then apply the second function g from P to S . By fixing h and studying all possible functions $g : P \rightarrow S$, we can narrow down the desired field of study to an arbitrary degree, and also classify the cellular automata according to the fixed function h .

One frequently used such class¹² of cellular automata called *totalistic* cellular automata can be defined by fixing h as the sum of the states of the cells in the neighbourhood. Stephen Wolfram adapted his labeling system to this kind of cellular automata by tabulating the possible sums in the top row and the new value for the central cell in the bottom. In a similar manner as before, the numbers in the bottom row can now be interpreted as a single number with the same base as the number of states.

Neighbourhood sum	6	5	4	3	2	1	0
New central state	2	0	1	2	0	2	0

Table 1.3: Code 1599 lookup table

For example, a radius 1 cellular automaton with $S = \mathbb{Z}_3$ has a maximal sum of 6, so we can read off a 7 digit base 3 number from the bottom row, as shown in table 1.3. To distinguish between these totalistic numbers and the rule numbers, these numbers are called *codes*. Together with the alphabet S and the neighbourhood N , the code will completely determine the cellular automaton.

Outer totalistic cellular automata are totalistic *outside* of the central cell, so they depend on both the sum of the states of the outer cells, and on the state of the central cell. In other words, the function h is of this form:

$$h : S^{|N|} \rightarrow \{0, 1, \dots, |S| \cdot |N|\} \times S$$

Consider a d -cellular automaton (\mathcal{L}, S, N, f) with a global function G . Recall that C is the set of all possible global states (which we may also write $S^{|\mathcal{L}|}$). Set $\Omega^0 = C$, and let $\Omega^t = G(\Omega^{t-1}) = \{G(\mathbf{a}) \mid \mathbf{a} \in \Omega^{t-1}\}$. The *limit set* of \mathcal{A} is the set $\Omega(\mathcal{A}) = \Omega^0 \cap \Omega^1 \cap \Omega^2 \cap \dots$, which corresponds to the periodic parts of the phase space.

¹²This is of course a different kind of classification than the wolfram classification, which we will cover in section 1.4.1.

1.4 Examples

1.4.1 1-dimensional binary cellular automata

Cellular Automaton 1 Radius 1 binary 1-CA

Lattice: $\mathcal{L} = \mathbb{Z}$ or \mathbb{Z}_n
Alphabet: $S = \mathbb{Z}_2 = \{0, 1\}$
Neighbourhood: $N = \{-1, 0, 1\}$ (first neighbours)
LTF: $f : S^3 \longrightarrow S$

One of the simplest kinds of cellular automata is the 1-dimensional binary cellular automata with a radius 1 neighbourhood. There are 256 distinct rules of this kind, but it can be argued that there are only 88 *different* rules. From each rule we might construct up to 3 very similar rules: the mirror image rule, the inverted rule, and the rule that is both mirrored and inverted. Any fact that is known of one of these variations can easily be translated into a similar fact about the others, so they can be considered isomorphic.

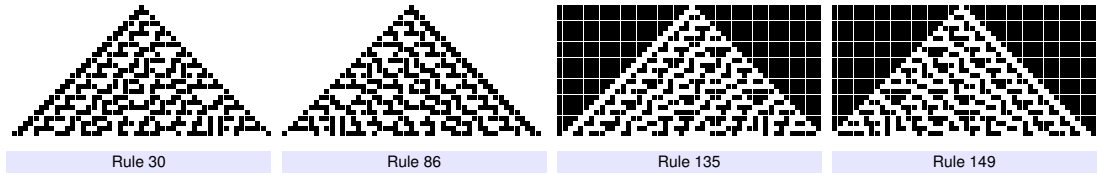


Figure 1.10: Equivalent rules of rule 30.

Take rule 30 as an example. Its mirror image rule is rule 86, its inversion is rule 135, and rule 149 is the mirror image of rule 135 and the inversion of rule 86. For some rules, some or all of these variations coincide; for instance, it's obvious that a symmetric rule is its own mirror image. The details of these relationships is listed in appendix B.

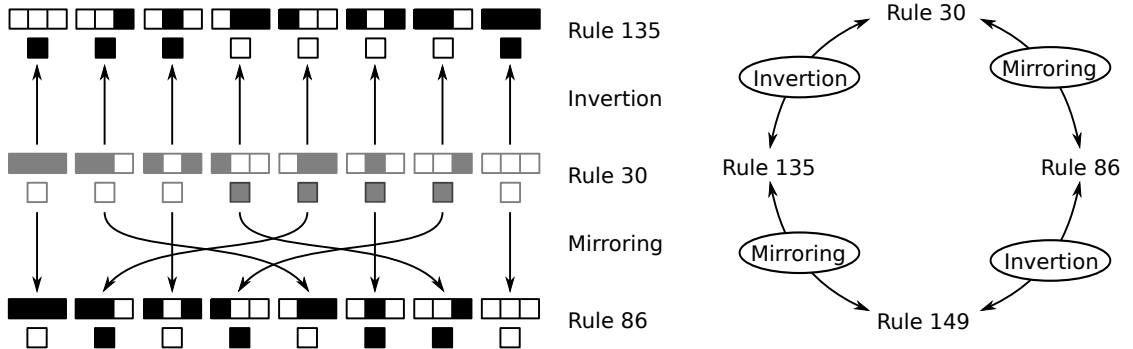


Figure 1.11: Rule 30 and its similar rules

More generally, for any rule with explicit form¹³ $abcdefgh$, the mirror image is $aecg-bfdh$. Using capital letters to denote inversion (1 to 0 and 0 to 1), the inversion is $HGFEDCBA$, and the mirror-inverse is $HDFBGCEA$ ¹⁴. The patterns produced by rules that are mirror images and/or inversions of each other are easily recognizable as such, especially if the initial global state is similarly mirrored and/or inverted, as can be seen in figure 1.10.

The classification system of Stephen Wolfram

Stephen Wolfram invented a classification system for cellular automata, with which the vast majority of cellular automata can be classified into one of four classes. The classification must however be done by manually inspecting the patterns, so the classification might to some degree depend on the person doing the classification and the initial conditions she studies. [19, p. 240]

Class I



Figure 1.12: Class I

This kind of system quickly evolves to a homogeneous global state, after which no further state changes happen. The most obvious example of this kind is rule 0, which instantly discards all information, though rule 8, 32, 40 and 160 are also of this kind. A cellular automata is classified as class I even if there exist certain special initial conditions that leads to a nonhomogenous final state; it is enough that the homogenous state is reached for *almost* all initial states [19, p. 231]. Any changes in the initial global state of a cellular automaton of this kind will have no effect on the resulting global state.

Class II

Cellular automata of this kind are those that after a while starts to repeat themselves. There's a slight problem in the fact that all cellular automata on a finite lattice must at some point reach a global state that it has been in before (there are only finitely many global states), after which it cannot help but repeat itself. This would be the case also for rules that would otherwise be categorized as class III or IV, had they been on an infinite lattice. It's therefore customary to only consider cellular automata that would be of class II on an infinite lattice as belonging to class II.

¹³See section 1.3.6 page 15.

¹⁴This does not depend on the order in which the transformations are applied.



Figure 1.13: Class II

The most trivial examples of class II cellular automata are those that reach some unchanging global state, like rule 4 or rule 12. It bears mentioning at this point that rules that produce left- or right-shifting patterns, like rule 24, present an apparent problem when one attempts to categorize them, since on an infinite lattice such a rule seems not fall into any of the four classes¹⁵. The solution is to observe that it is equivalent to a rule where the neighbourhood is shifted one step to the left or right, depending on the rule, as in Figure 1.14. This will then produce a pattern that is clearly recognizable as that of a class II cellular automaton, and which can easily be transformed back into the original pattern produced by rule 24 by shifting each row the same number of steps to the right as the number of rows above it.

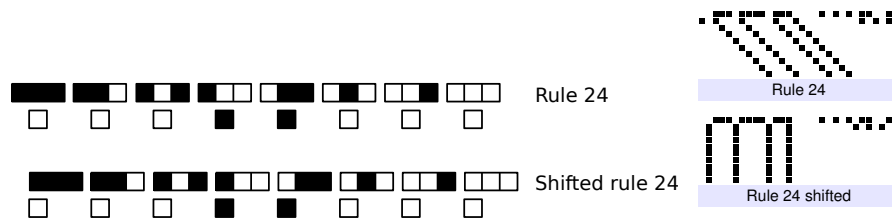


Figure 1.14: How to classify rule 24

Any small local change in the initial global state of a cellular automaton of this kind will yield only a small local change in the resulting pattern. [19, p. 252]

Class III

Class III cellular automata are characterised by their chaotic nature. Almost all initial conditions will lead to patterns that never repeats, with the same finite lattice caveat as for class II. A classical example of this kind of cellular automaton is rule 30, and other examples are rule 90 and rule 150.

For this kind of system, a small change in the initial system will “typically spread at a uniform rate, eventually affecting every part of the system”. [19, p.252]

¹⁵ It does not quickly evolve into a homogenous global state, so it's not class I. It does not ever repeat itself unless the initial condition also does so, so it is not class II. It is highly predictable, so it is not chaotic, and therefore not class III. Lastly, it does not have stationary patterns, so it is not of class IV.



Figure 1.15: Class III

Class IV

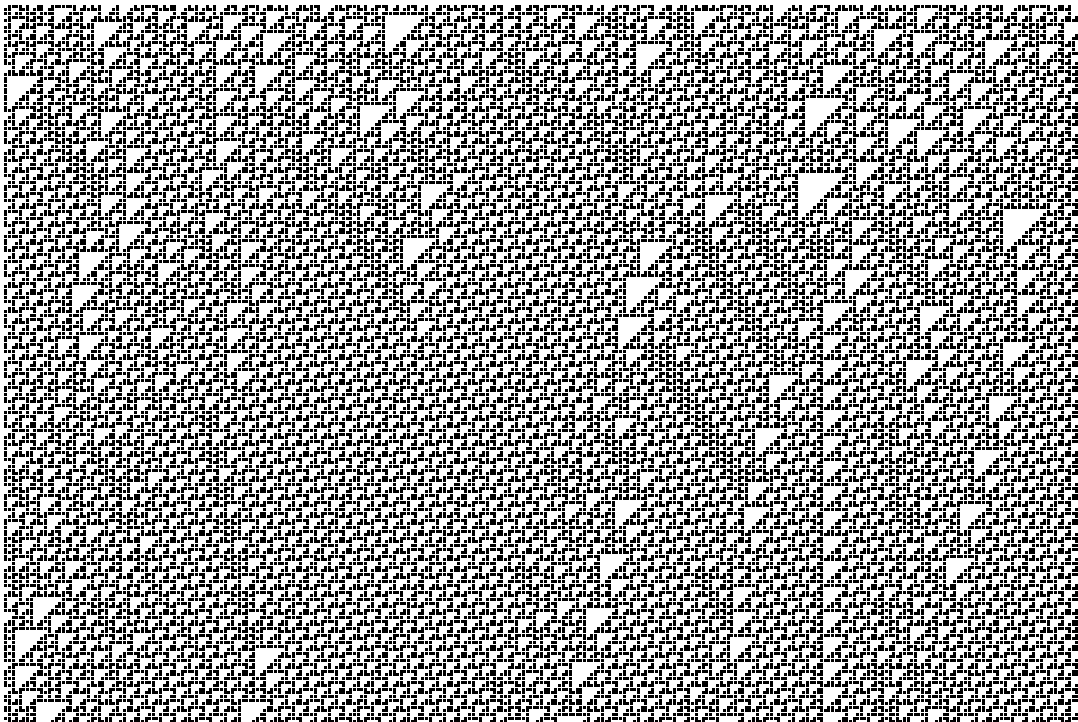


Figure 1.16: Rule 110

This kind of system can be seen as being on the border between the order of class II and the chaos of class III. These kinds of cellular automata are the kinds that could be capable of universal computation, because they have both patterns that move about and patterns that are stationary. Only one of the radius 1 binary 1-dimensional cellular automata is known to be of this type, and that is rule 110 and its equivalent rules. It was proved by Matthew Cook around 2000 that Rule 110 is turing complete, meaning that it can in principle be used to calculate anything, just like a computer. [19, p. 1115]

A small change in such a system can have either no impact, local impact, or a non-local impact on the evolution of the cellular automata. [19, p. 252]

Apart from making pretty patterns, some of these rules also have practical applications. Rule 30 is used as a pseudo-random number generator [19, p. 975], and seems to have properties making it well-suited for cryptographic purposes such as hashing and encryption. If we take a binary string from any source – for example the rows of a black and white image – we may set it as the initial configuration of a CA such as this and run it once or twice. In this way we may use these rules to filter such data for various features; for instance rule 4 can be used to mark each place where the bits formed the pattern 010, while rule 72 marks the beginning and end of each stretch of more than one cell with state 1, so 011010111110 would filter to 011000100010.

Considering that CAs of this type are of the most simple CAs possible, they have been a surprisingly rich store of different and complex behaviour. In the next section we will increase the radius of the neighbourhood and the number of states, and see what happens.

1.4.2 Other 1-dimensional cellular automata

Cellular Automaton 2		Radius r , m -state 1-CA
Lattice:	$\mathcal{L} = \mathbb{Z}$ or \mathbb{Z}_n	
Alphabet:	$S = \{0, 1, \dots, m-1\}$	
Neighbourhood:	$N = N_r = \{-r, -r+1, \dots, r-1, r\}$	
LTF:	$f : S^{2r+1} \longrightarrow S$ (general) or $f = g \circ h : S^{2r+1} \longrightarrow \mathbb{Z}_{(2r+1)(m-1)+1} \longrightarrow S$ (totalistic)	

The general cellular automata of this kind is represented by a map f whose explicit form is $k = m^{2r+1}$ digits long: $s_{k-1}s_{k-2}\dots s_1s_0$, $s_i \in S$. It is then straightforward to look up the new state corresponding to any neighbourhood state, as described in section 1.3.6. For the totalistic variant, the maximal sum of the cells in the neighbourhood is $\sigma = (2r+1)(m-1)$, so our function $h : S^{2r+1} \longrightarrow \mathbb{Z}_{\sigma+1}$ is defined as

$$h(s_0, s_1, \dots, s_{2r}) = \sum_{i=0}^{2r} (s_i)$$

and we may now let $g : \mathbb{Z}_{\sigma+1} \longrightarrow S$ be any function.

Cellular automata with larger neighbourhoods than the nearest neighbours neighbourhood are in principle a superfluous kind of cellular automata to be studying, due to the following fact: “Every d -cellular automaton can be simulated by a d -cellular automaton with the nearest neighbors neighborhood” [5, p.8]. For example, a cellular automaton with radius 4 and 2 states can be simulated by a cellular automaton with radius 1 and 16 states, as in figure 1.17.

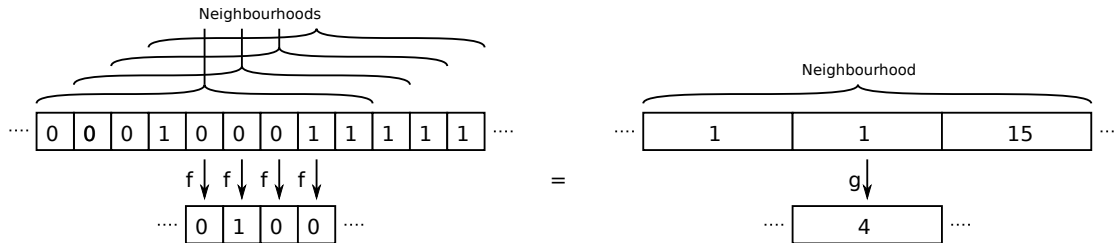


Figure 1.17: How the radius 4 rule with 2 states and code 273 can be converted to a 16-state radius 1 rule.

It should therefore in principle be enough to study all cellular automata with the nearest neighbours neighbourhood. However, a concrete example reveals some problems with this approach.

Figure 1.18 shows a radius 4 binary cellular automaton and its equivalent¹⁶ with a radius 1 neighbourhood. It is evident that much more information can be read from the

¹⁶The equivalent radius 1 rule is not a totalistic rule for a number of reasons. For example, neighbourhood states 888, 444, 222 and 111 should now all give the “sum” 3, but they clearly don’t.

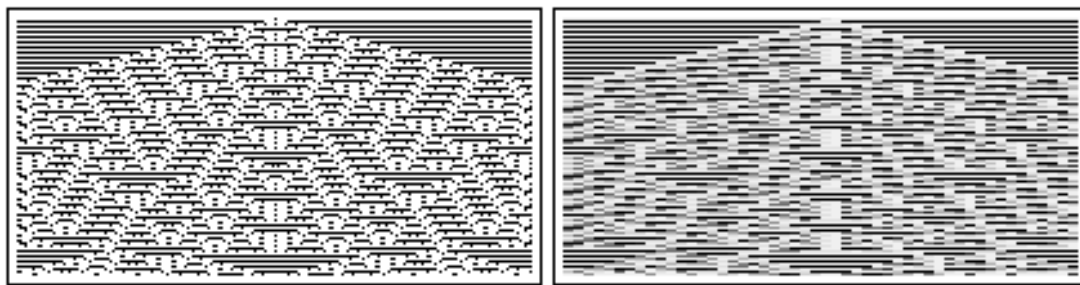


Figure 1.18: Left: The totalistic radius 4 binary cellular automaton with code 273. Right: its equivalent radius 1 cellular automaton with 16 states.

left representation than from the right, even though, theoretically, the two representations are equivalent and store exactly the same information. The left is simply a more expressive and humanly readable way of displaying this automaton. For this reason, we will not limit ourselves to radius 1 CAs.

Totalistic rules are necessarily symmetric, but that does not mean that all patterns produced by them are symmetric. This is only the case if the initial configuration is also symmetric. Figure 1.19 shows a totalistic CA where the LTF causes a particular pattern (11011101) to shift one cell to the left every nine time steps. The mirror image of the pattern (10111011) would shift right instead of left.

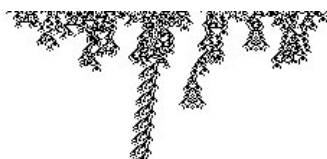


Figure 1.19: Totalistic rules are symmetric, yet can display unsymmetric behaviour. Displayed is $N = N_{r=2}$, $S = \mathbb{Z}_2$ for code 20 with a random initial configuration.

We could equally well have introduced the wolfram classes presented in section 1.4.1 here, with examples taken from totalistic CAs, as they feature all the four classes. This would also have removed the problem that rule 24 created for us. That rule discarded a lot of information and then simply shifted what was left to the right. Any totalistic rule with left- or right-shifting patterns will be of class III or IV, since such patterns means information can be transmitted both left and right.

The explicit form of the equivalent rule, written as an hexadecimal number, is 4096 digits long: 01327644fecc8888fecc888800000001...2310100010000000100000008888ccef. From the first digit we can conclude that the neighbourhood fff (which in the radius 1 equivalent is twelve cells with state 1 in a row: 111111111111) results in the new state 0 (which is equivalent to four cells with state zero: 0000). Similarly, from the last digit we see that the neighbourhood 000, (≈ 000000000000), gives new state f (≈ 1111). From this alone we can conclude that a CA with this rule and homogenous initial configuration would have alternating horizontal black and white stripes.

As with most types of CA, the parameter space is vast, so for this example we will just pick a couple of interesting cases and comment on those. Figure 1.20 shows the 3-state cellular automaton with code 1599 and radius 1, starting from a global state of all cells in state 0 except a single cell in state 1. Its lookup table can be found in table 1.3 on page 17. It shows how delicately poised between expansion and stagnation a cellular automaton can be. It evolves and expands for 8282 steps of highly complicated behaviour before it finally reaches a simple repetitive state. A small sample of various lattice sizes and random initial states shows this to be typical behaviour.

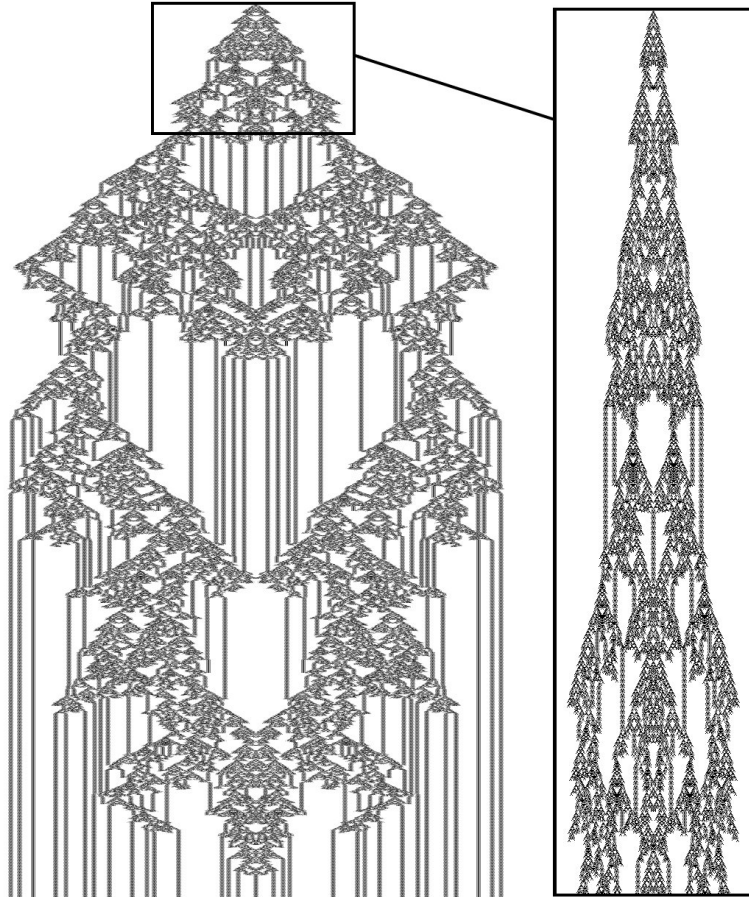


Figure 1.20: The totalistic 3-state radius 1 CA with code 1599, starting with a single cell in state 1. The picture has been squashed along the vertical axis: each pixel in the large image is the average of a column of 7 pixels, as shown in the blowup to the right.

Randomly browsing totalistic CA with $|S| > 2$ and $r > 1$ can be frustrating, because most patterns will simply look like a big mess, and only once in a while will a pattern that is interesting in some way crop up. We will end this section with an example of the latter; the pattern in figure 1.21. It has unusually large structures of similarly textured cells, which in and of itself is not that unusual, but these move about in a complex

manner, instead of staying in one place in the lattice.

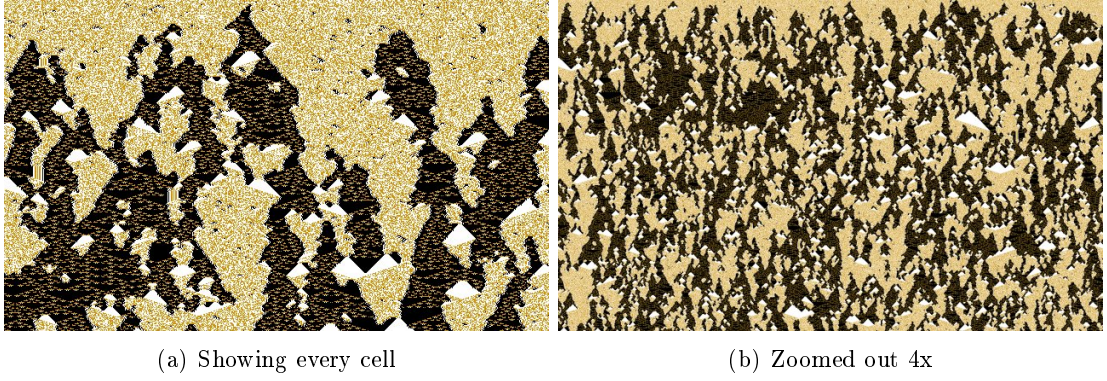


Figure 1.21: A totalistic CA with $N = N_{r=3}$, $S = \mathbb{Z}_3$ and code 12739548.

1.4.3 Activation-inhibition model

Cellular Automaton 3	Algorithmic 1-CA
Lattice:	$\mathcal{L} = \mathbb{Z} \text{ or } \mathbb{Z}_n$
Alphabet:	$S = \mathbb{Z}_2 \times \mathbb{Z}^+$
Neighbourhood:	$N = \{-\max(r_a, r_i), \dots, \max(r_a, r_i)\}$ (radius neighbourhood)
LTF:	$f : S^{ N } \longrightarrow S$ (algorithmic)

The following cellular automaton was proposed by Ingo Kusch and Mario Markus [10], and presented in [16]. It reproduces a number of patterns very similar to some found in nature, for instance on seashells. Each cell is either active or inactive, and there's a certain concentration of inhibitor in each cell. Active cells produce inhibitor, which diffuse to nearby cells. If there is too much inhibitor in an active cell, it becomes inactive.

Each cell has a tuple (c, i) as its state, where $c \in \mathbb{Z}_2$ indicates if the cell is active (1) or not (0), and $i \in \mathbb{Z}^+$ is the concentration of inhibitor¹⁷. We define two projections $\pi_c((c, i)) = c$ and $\pi_i((c, i)) = i$, so that for example, $\pi_i(\mathbf{a}^t(\vec{z}))$ is the amount of inhibitor in the cell at \vec{z} in time step t .

For the sake of overview we will first summarize the algorithm and name the various parameters.

1. Inhibitor decays, either decreasing by 1 or by a factor of d .
2. Each inactive cell activates with probability p .
3. All active cells produce w_1 units inhibitor.
4. Cells with more than $\lfloor m_0 + m_1 i \rfloor$ active cells in their radius r_a neighbourhood are activated.
5. The inhibitor is diffused across the cells in their radius r_i neighbourhood.
6. All active cells whose inhibitor levels are higher than w_2 are deactivated.

Starting with a global state \mathbf{a}^t , we will need two temporary global states \mathbf{a}_1 and \mathbf{a}_2 to calculate the next global state \mathbf{a}^{t+1} , and we need to make 3 passes, one from \mathbf{a}^t to \mathbf{a}_1 , one from \mathbf{a}_1 to \mathbf{a}_2 , and one from \mathbf{a}_2 to \mathbf{a}^{t+1} .

1. Pass 1. For every $x \in \mathcal{L}$, let $(c, i) = \mathbf{a}^t(x)$, and:

- a) Let $i_1 = \max(0, i - 1)$.
- b) If $c = 0$, then with probability p , let $c_1 = 1$, otherwise let $c_1 = 0$.
- c) Set $\mathbf{a}_1(x) = (c_1, i_1)$.

¹⁷Though the alphabet is given as $\mathbb{Z}_2 \times \mathbb{Z}^+$, which is an infinite set and therefore not allowed as an alphabet, in practice the size of i is bounded by some of the parameters, so it's finite

2. Pass 2. For every $x \in \mathcal{L}$, let $(c_1, i_1) = \mathbf{a}_1(x)$, and:
 - a) If $c_1 = 1$ then $i_2 = i_1 + w_1$, otherwise $i_2 = i_1$.
 - b) If $c_1 = 0$ and $\sum_{i=-r_a}^{r_a} (\pi_c(\mathbf{a}_1(x+i))) > \lfloor m_0 + m_1 i_2 \rfloor$, then $c_2 = 1$, otherwise $c_2 = c_1$.
 - c) Set $\mathbf{a}_2(x) = (c_2, i_2)$.
3. Pass 3. For every $x \in \mathcal{L}$, let $(c_2, i_2) = \mathbf{a}_2(x)$, and:
 - a) Let $i = \left\lfloor \frac{1}{2r_i+1} \sum_{i=-r_i}^{r_i} (\pi_i(\mathbf{a}_2(x+i))) \right\rfloor$.
 - b) If $i \geq w_2$, then $c = 0$, otherwise, $c = c_2$.
 - c) Set $\mathbf{a}^{t+1}(x) = (c, i)$.

For $0 < d < 1$, we replace step 1.a) above by

$$\text{Let } i_1 = \max(0, \lfloor (1-d)i \rfloor).$$

That was the algorithm, now we will look at what kind of output it produces. Figure 1.22 shows four different patterns, with parameters given in the figure text. There are many more beautiful patterns to be discovered, and playing with the parameters of this CA is a pastime recommended¹⁸.

Note that the figures hide one part of the state, namely the concentration of inhibitor. This information can be guessed at by observing where the cells spontaneously turn white, because this is due to high concentrations of inhibitor. The inhibitor will then gradually decay and diffuse, and then the cells will either spontaneously activate again once the inhibitor concentration is low enough, or they will be activated because many of their neighbouring cells are activated.

With certain specific parameters, the patterns found on many different kinds of seashells have been reproduced, and some impressive side by side comparisons are shown in Schiff [16], p. 152. We may conclude that CAs can be used to model real-world phenomena quite successfully, and help deepen our understanding of how things work by enabling us to study it in an easily controlled environment – a computer simulation.

¹⁸See appendix A.4.

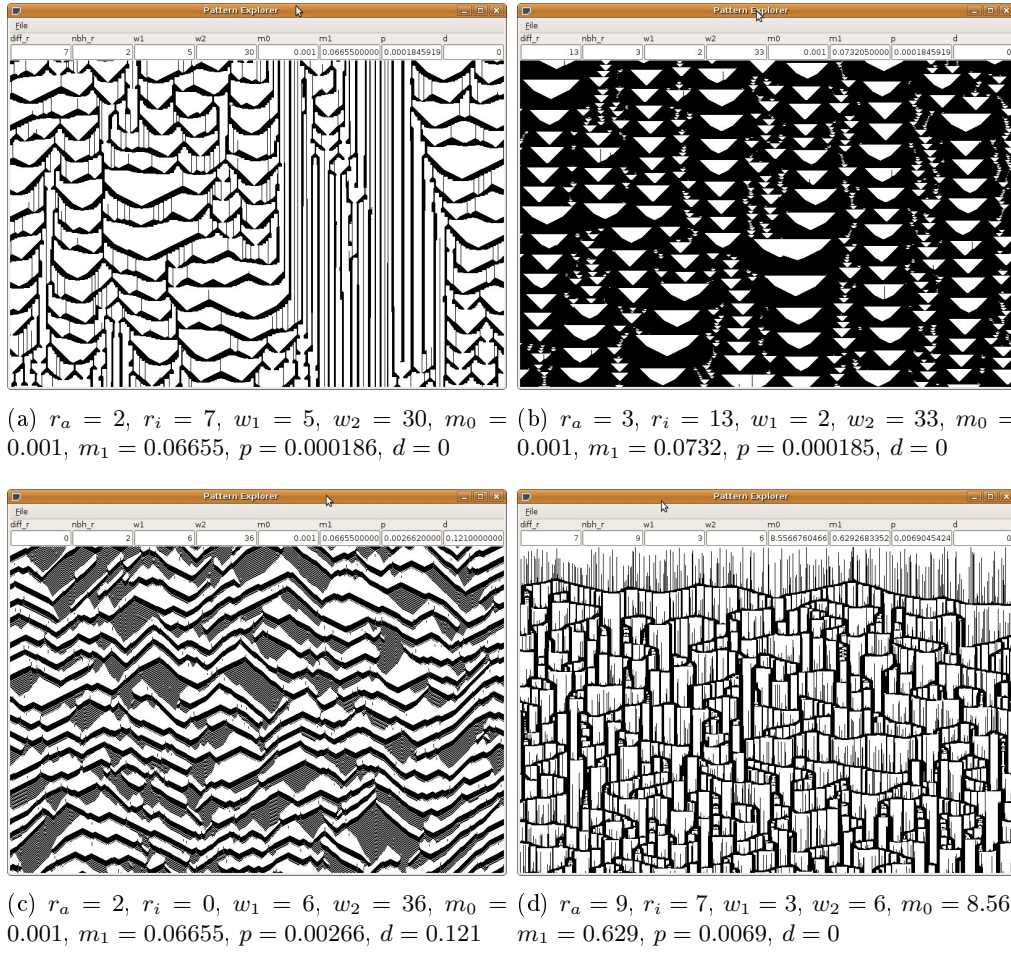


Figure 1.22: Kusch-Markus patterns

1.4.4 The Game of Life

Cellular Automaton 4 Binary outer totalistic 2-CA

Lattice:	$\mathcal{L} = \mathbb{Z} \times \mathbb{Z}$ or $\mathbb{Z}_n \times \mathbb{Z}_m$
Alphabet:	$S = \mathbb{Z}_2$
Neighbourhood:	N_M (2-dimensional Moore neighbourhood)
LTF:	$f = g \circ h : S^9 \longrightarrow \mathbb{Z}_9 \times \mathbb{Z}_2 \longrightarrow S$ (outer totalistic)

This is one of the most famous cellular automata, and its history has already been mentioned in section 1.2.2. It was devised with a clear set of goals in mind: no commonly occurring pattern should grow without bound¹⁹, and there should be many patterns that do not quickly die out or become predictable. [11, p. 3]

This CA is outer totalistic, so we fix the function $h : S^9 \longrightarrow \mathbb{Z}_9 \times \mathbb{Z}_2$ as

$$h(s_0, s_1, \dots, s_7, s_8) = \left(\sum_{i=1}^8 (s_i), s_0 \right) \quad (1.1)$$

The rules of the Game of Life are as follows. Each cell can have only two states; alive (1) or dead (0). For $h(s_0, \dots, s_8) = (\sigma, s)$, we define

$$g((\sigma, s)) = \begin{cases} 1 & \text{if } \sigma = 2 \text{ and } s = 1 \\ 1 & \text{if } \sigma = 3 \\ 0 & \text{otherwise} \end{cases}$$

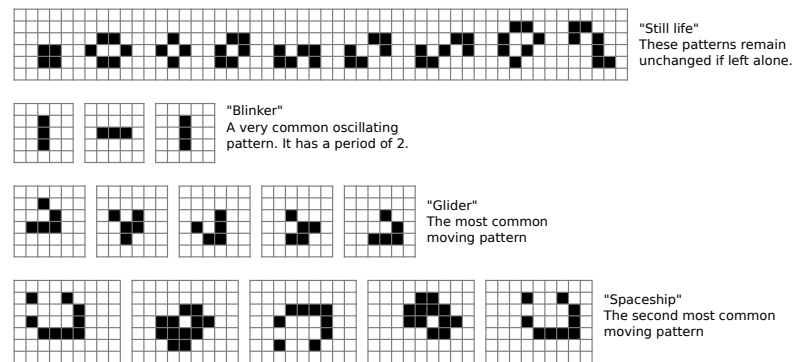


Figure 1.23: A sampling of rather common patterns seen in the Game of Life cellular automaton.

This simple rule gives rise to a stunning menagerie of behaviours, a small sampling of which will be mentioned here.²⁰ Referring to figure 1.23, at the top we have some

¹⁹It was soon proved by counterexample that there did exist patterns that grew without bound. More recently it has been shown that you need at least 10 or more live cells in the initial global state to have a hope of boundless growth. [19, p. 965]

²⁰Any greater thirst for knowledge on the subject of the standard Game of Life may be quenched at <http://conwaylife.com/wiki/>.

examples of *still life*, which are patterns that do not change at all from one time step to the next. It's possible to combine or extend many of the shown patterns into larger examples of still life, even creating tilings that may cover the entire lattice.

Then there are stationary oscillators, where we show only the most simple, the period 2 *blinker*. However, there exists such patterns for every period up to 18 [19].²¹ Thirdly, there are the moving repeating structures, called *spaceships*. The two most common are shown in the figure, the glider and the eponymously named “spaceship” spaceship.

Spaceships may be categorized by the speed with which they move, expressed in fractions of c , where c denotes the so-called *speed of light* – one cell per time unit. The glider has a speed of $\frac{c}{4}$ both horizontally and vertically, and the spaceship has a speed of $\frac{c}{2}$. The most spectacular example of a spaceship is the monstrous *caterpillar*, pictured in figure 1.24, which has a speed of $\frac{17c}{45}$ and consists of roughly 12 million live cells in a 4195×330721 bounding box.

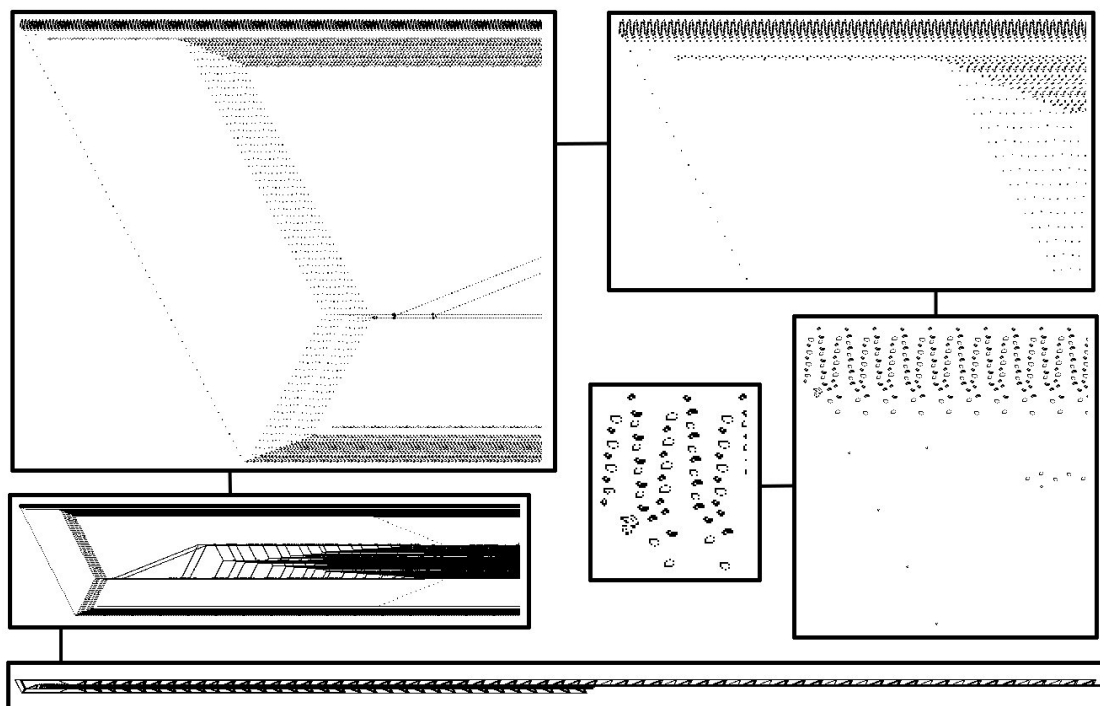


Figure 1.24: The caterpillar. Starting at the bottom left and going in clockwise direction, each frame contains a magnified part of the picture in the frame preceding it.

The standard Game of Life has been analyzed in extreme detail by many, many others, so we will not cover it in any greater detail here. We will come back to it when

²¹A Life enthusiast called David J. Buckingham has written an online article [4] where he describes how to create stationary oscillators of every period larger than 57. Along with other stationary oscillators found by him and others, only periods 19, 23, 31, 38, 41, 43 and 53 now remain to be found. However, this is according to [13], which is an open wiki, so it should be taken with a grain of salt.

we give it 2-dynamics in section 2.4.2. We will now make the leap to 2 dimensions.

1.4.5 The hodgepodge machine

Cellular Automaton 5	Cyclic n -state 2-CA
Lattice:	$\mathcal{L} = \mathbb{Z} \times \mathbb{Z}$ or $\mathbb{Z}_k \times \mathbb{Z}_l$
Alphabet:	$S = \mathbb{Z}_{n+1} = \{0, 1, \dots, n\}$
Neighbourhood:	$N = N_M$ (2-dimensional Moore neighbourhood (see 1.3.5))
LTF:	$f : S^9 \longrightarrow S$ (algorithmic)



Figure 1.25: The Belousov-Zhabotinsky reaction. Image by Stephen Morris, <http://www.flickr.com/photos/nonlin/4013035510/in/photostream/>. Used with permission.

This cellular automata was suggested by Martin Gerhardt and Heike Schuster [16, p. 130], to simulate an oscillating chemical reaction on excitable media. The reaction is called Palladium Oxidation; palladium crystals on a surface absorb CO and O_2 , and produce CO_2 at a rate that increases with the amount of CO available. This produces a cyclic pattern of spiral waves. A very similar pattern (made of waves of a chemical used for intercellular signal transduction) is produced by slime mold when feeding [6]. A third example of a reaction producing such a pattern is the so-called Belousov-Zhabotinsky reaction, which is depicted in figure 1.25.

The CA is a 2-dimensional cellular automata (\mathcal{L}, S, N, f) with $n + 1$ states: $S = \{0, 1, \dots, n\}$. A cell in state 0 is “healthy”, a cell in state n is “ill”, and the states in between are termed “infected” states.

In addition to three parameters k_1 , k_2 and g , the local transition function depends on the state of the central cell, denoted by s , the number of infected cells²² in the neighbourhood N_{inf} , the number of ill cells in the neighbourhood N_{ill} , and the sum of all the states in the neighbourhood Σ . For the sake of readability we write the function f as if it is a function of these derived variables instead of (s_0, s_1, \dots, s_8) , $s_i \in \mathbb{Z}_{n+1}$, since all the mentioned variables can easily be calculated.

²²An ill cell is also considered to be infected.

$$f(s, N_{\text{inf}}, N_{\text{ill}}, \Sigma) = \begin{cases} \left\lfloor \frac{N_{\text{inf}}}{k_1} + \frac{N_{\text{ill}}}{k_2} \right\rfloor & \text{if } s = 0 \\ \min\left(g + \left\lfloor \frac{\Sigma}{N_{\text{inf}}} \right\rfloor, n\right) & \text{if } 0 < s < n \\ 0 & \text{if } s = n \end{cases}$$

The parameters k_1 and k_2 influence how easily a cell is infected, and g controls how fast the cells progress towards becoming ill once they've become infected. With parameters set to appropriate values (for example $n = 100$, $g = 20$, $k_1 = 3$, $k_2 = 2$), a random initial configuration will after a while become spiralling waves, as in the simulation in figure 1.26.

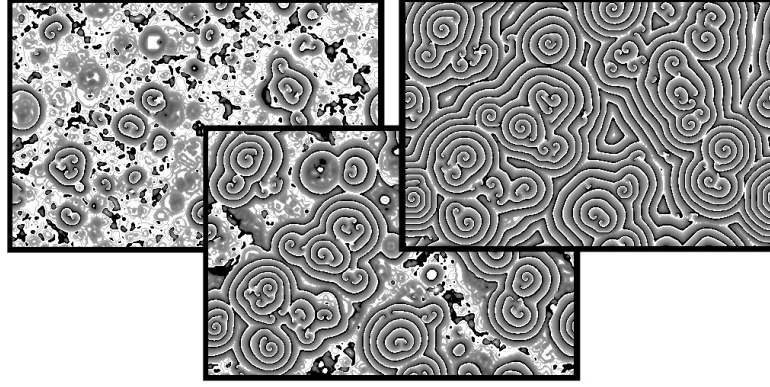


Figure 1.26: A typical evolution of the hodgepodge machine. The white cells are healthy, the gray cells are infected, and the black cells are ill.

We'll now explore how the parameters influence the behaviour of this cellular automaton, with the exception of the parameter n , which we will not investigate at this time.²³ The results are displayed in figure 1.27. As can be seen, whether spirals appear or not is largely dependent on the parameters, though the initial configuration may also have some influence, particularly for the borderline cases. When spirals don't appear, there are various other kinds of patterns that emerge, some of which are depicted in figure 1.28. Once a spiral appears though, it will eventually dominate the whole lattice, ever spreading its spiralling waves outwards²⁴.

The averaging of states that is done all the way from a state becomes infected until it is ill is crucial to the richness in behaviour. If we just increased the state value by g each time step, most of the interesting patterns shown here would disappear and be replaced by chaotic patterns. We used the same principle in the activation-inhibition patterns of section 1.4.3, and we will use it again later in this thesis, and it always seems to produce results that are at pleasing to the eye, or natural, in some sense.

²³I suspect it is the ratio $\frac{g}{n}$ that matters, not the magnitude of the numbers g or n .

²⁴My initial attempts at visualizing the parameter space failed because I set no borders to separate the parameters, allowing the spirals that invariably appeared to destroy any other weaker pattern that might appear.

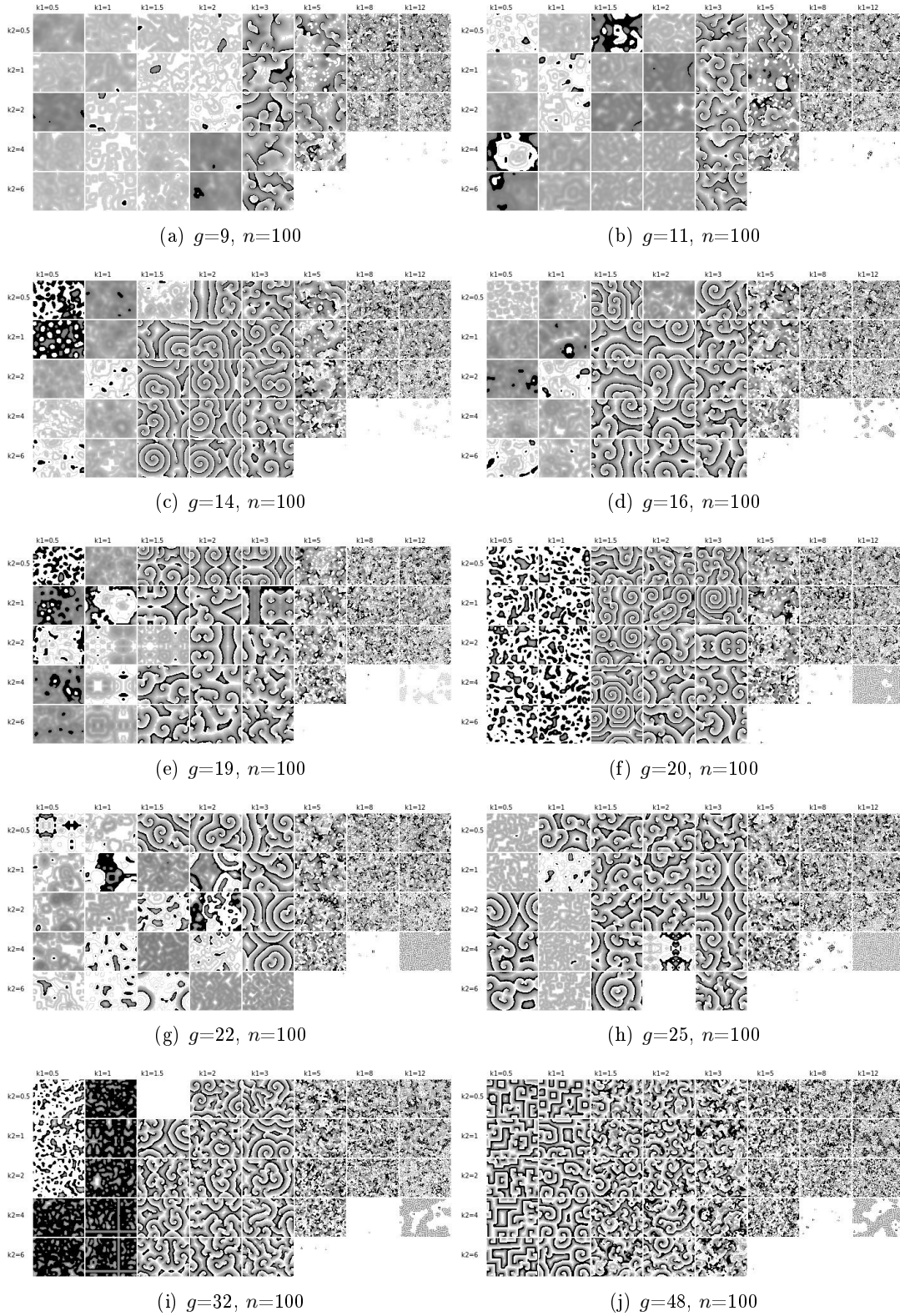


Figure 1.27: The hodgepodge machine after 1200 steps for various parameter combinations. The initial configuration is identical for all simulations with the same parameter g , so any difference between the patterns is only due to the difference in parameters. The row and column headers may be hard to read, so to clarify; along the x axis the parameter k_1 has the values 0.5, 1, 1.5, 2, 3, 5, 8 and 12, and along the y axis k_2 takes on the values 0.5, 1, 2, 4 and 6. These values were selected by trial and error so as to show as many different behaviours as possible.

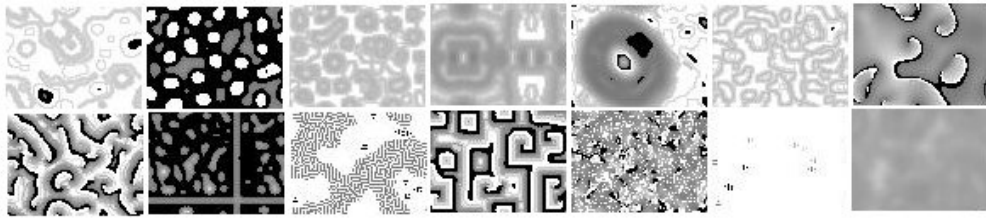


Figure 1.28: Some of the other patterns that the hodgepodge machine may produce.

1.4.6 WATOR-World

Cellular Automaton 6		A predator-prey simulation CA
Lattice:	$\mathcal{L} = \mathbb{Z} \times \mathbb{Z}$ or $\mathbb{Z}_n \times \mathbb{Z}_m$	
Alphabet:	$S = \mathbb{Z}_3 \times \mathbb{Z}_{\max(b_f, b_s)+1} \times \mathbb{Z}_{s+1} \times \mathbb{Z}_2$	
Neighbourhood:	$N = N_{vN}$ (2-dimensional von Neumann)	
LTF:	$f_w : S^5 \longrightarrow S^5$ (asynchronous, algorithmic)	

The following CA was proposed by Alexander Keewatin Dewdney in an article in Scientific American [16, p. 154]. It is called WATOR-World, and is a discretization of the Lotka-Volterra differential equations, which model the relationship between the populations of a predator and its prey. Since such relationships are discrete in nature, a CA model will probably yield more realistic answers than differential equations, all other things being equal. At the very least, it will solve the so-called atto-fox problem of [12, p. 32].²⁵

In this CA, each cell is either empty water, a small fish, or a shark. The sharks and the fish swim randomly about, and if a shark comes across a fish, the shark eats the fish. Both fish and sharks may reproduce once they reach a certain age. They breed by simply depositing an offspring in the cell they leave – a mate is not needed. At the same time, their “age” is reset, so as to avoid adult fish leaving behind a continuous trail of offspring. Finally, sharks will die if they go too long without food.

Stretching the CA definitions

This CA necessitates stretching the definitions of a CA a bit. Schiff introduces this model as a CA in [16], but later states that it is in fact an *IBM*, or individual-based model. An IBM focuses on the individuals that make up the model, instead of the locations that they occupy, as we do in cellular automata. Where we would iterate over the entire lattice, an IBM typically iterates over its population.

In forcing this into the CA framework, we get a problem because the subjects (fish and sharks) often wander in a random direction. The obvious way of doing this in a CA is by swapping the states of two cells, where one is a fish and the other is its empty destination. Though doable, such procedures must be considered carefully in a CA, because both the cells involved in the swapping must be in “agreement” that it is taking place. If they are not, the result is a cloned fish, or a fish that has disappeared.

The fact that the direction of the swapping is random makes it much worse, if not impossible, to implement as a standard CA. Take for example a seemingly simple problem, will a fish leave its cell or not, depicted in figure 1.29. As can be seen, at the very least we’ll need a bigger neighbourhood, and even then it is not so simple. If the next movements of the other four fish are not known, the fish in the middle may have a free cell to move to, or it may not.

²⁵An atto-fox is $1 \cdot 10^{-18}$ of a fox, which is the number of foxes per square kilometer that allowed the population to rebound in the in the cited article.

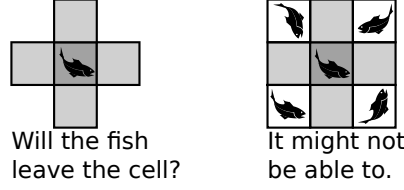


Figure 1.29: One of the problems faced when implementing WATOR-World as a proper CA.

The solution is to enable the LTF to change the state of neighbouring cells as well as its own, and do the update in-place, or asynchronously. This means that the LTF will decide the new position of a swimming entity based on where the fish around it are “now”, as opposed to where they were one time step ago. We still keep the concept of a time step by denoting an update of every cell in the lattice as one time step.

There is also the problem that a fish might be moved several times in one time step if it moves into an as of yet not updated cell. This we will mitigate by marking the fish as “moved”, and making sure not to move fish that has been marked as moved already.

The algorithm

Let $\mathbf{a} : \mathcal{L} \longrightarrow \mathbb{Z}_3 \times \mathbb{Z}_{\max(b_f, b_s)+1} \times \mathbb{Z}_{s+1} \times \mathbb{Z}_2$ denote the global configuration map. We denote the state of an arbitrary cell by (w, a, l, m) , where w is the presence of water (0), fish (1) or shark (2), a is the number of time steps since the fish or shark was born, l is the number of time steps since the last meal of a shark, and m is the flag indicating that the fish or shark has been moved. Further, let π_w, π_a, π_l and π_m be the projections from the alphabet S to each of the smaller sets. For example, $\pi_w((w, a, l, m)) = w$, and similarly for a, l and m . Finally, the parameters of the CA are these: the breeding age of fish and sharks we denote by b_f and b_s , respectively, and sharks starve after s time steps without food.

Now let's specify the algorithm in detail. Because of the vagueness of the description in [16], some details have been filled in or clarified as the author saw fit.

Each time step consists of two passes over the cells. The first pass resets the value m to 0 for all the cells in the lattice, indicating that no fish has moved yet. The second pass is done in random order while still making sure only to update each cell once. This is to avoid the tendency of tight packs of fish to move towards the starting point of the pass. Using the von Neumann neighbourhood $N_{vN} = \{\vec{n}_0, \vec{n}_1, \vec{n}_2, \vec{n}_3, \vec{n}_4\}$ we do the following for each $\vec{z} \in \mathcal{L}$,

1. Denote the current state of the cell by (w, a, l, m) .
2. If $w = 0$, do nothing more for this cell.
3. If $w = 1$ and $m = 0$, then:
 - a) If $a \leq b_f$, then increase a by one, aging the fish.

- b) Let $D = \{\vec{n} \in N_{vN} \mid \pi_w(\mathbf{a}(\vec{z} + \vec{n})) = 0\}$.
 - c) If $D = \emptyset$, the fish has nowhere to move, so update the state of the cell with the new age by setting $\mathbf{a}(\vec{z}) = (1, a, 0, 1)$, and do nothing more for this cell.
 - d) If $D \neq \emptyset$, then pick a $\vec{n}_x \in D$ at random.
 - e) If $a = b_f$ then the fish will breed, so we set $\mathbf{a}(\vec{z} + \vec{n}_x) = (1, 0, 0, 1)$ and $\mathbf{a}(\vec{z}) = (1, 0, 0, 1)$.
 - f) If $a < b_f$, the fish simply moves, so we set $\mathbf{a}(\vec{z} + \vec{n}_x) = (1, a, 0, 1)$ and $\mathbf{a}(\vec{z}) = (0, 0, 0, 1)$.
4. If $w = 2$ and $m = 0$, then:
- a) If $a \leq b_s$, then increase a by one, aging the shark.
 - b) If $l = s$, then the shark dies from hunger, so set $\mathbf{a}(\vec{z}) = (0, 0, 0, 0)$, and do nothing more for this cell.
 - c) If $l < s$, then increase l by one, making the shark hungrier.
 - d) Let $D = \{\vec{n} \in N_{vN} \mid \pi_w(\mathbf{a}(\vec{z} + \vec{n})) = 1\}$.
 - e) If $D \neq \emptyset$, then pick a $\vec{n}_x \in D$ at random, and set $l = 0$, since the shark will eat the fish at $\vec{z} + \vec{n}_x$.
 - f) If $D = \emptyset$, then let $D = \{\vec{n} \in N_{vN} \mid \pi_w(\mathbf{a}(\vec{z} + \vec{n})) = 0\}$ instead.
 - g) If $D = \emptyset$ even now, the shark has nowhere to move (*very* unlikely), update the state of the cell with the new age and hunger by setting $\mathbf{a}(\vec{z}) = (2, a, l, 1)$, and do nothing more for this cell.
 - h) If $l \neq 0$, pick a $\vec{n}_x \in D$ at random.
 - i) If $a = b_f$ then the shark will breed, so we set $\mathbf{a}(\vec{z} + \vec{n}_x) = (2, 0, l, 1)$ and $\mathbf{a}(\vec{z}) = (2, 0, 0, 1)$. The baby shark is born with a full stomach.
 - j) If $a < b_f$, the shark simply moves, so set $\mathbf{a}(\vec{z} + \vec{n}_x) = (2, a, l, 1)$ and $\mathbf{a}(\vec{z}) = (0, 0, 0, 0)$.

Schiff introduces two more parameters, n_f and n_s , the starting number of fish and sharks, respectively. A typical evolution of the WATOR world is pictured in figure 1.30. The sharks eat almost all the fish, and after a while great numbers starve to death since most of them find no more food. A small remnant of fish sustain an even smaller number of sharks, and for a while the fish breed faster than the sharks can eat. Soon the fish have multiplied to cover almost the whole lattice, while the sharks are becoming more numerous again, feasting on the fish. All of a sudden, the sharks have eaten almost all the fish, and the loop repeats, as shown in figure 1.31.

In this example we will study just one set of arbitrarily chosen parameters, namely $b_f = 8$, $b_s = 20$ and $s = 20$. We will consider the influence the lattice size has on the way the CA behaves. The parameters n_f and n_s vary according to the size of the lattice, and to a small extent between simulations, but the $\frac{\text{fish}}{\text{water}}$ and $\frac{\text{shark}}{\text{fish}}$ proportions will be very similar for all the simulations.

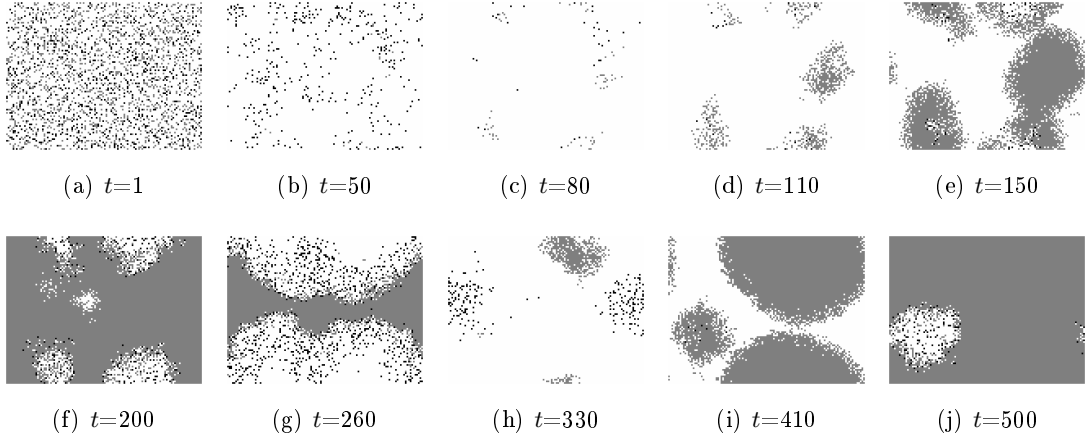


Figure 1.30: A WATOR-World evolution on an 120×90 lattice with $b_f = 8$, $b_s = 20$ and $s = 20$. Black cells are sharks, gray cells are fish and white cells are water. To construct the initial configuration, 20% of the lattice cell states were set to state $(1,0,0,0)$, then 15% were set to state $(2,0,0,0)$, possibly overwriting some of the previously set values.

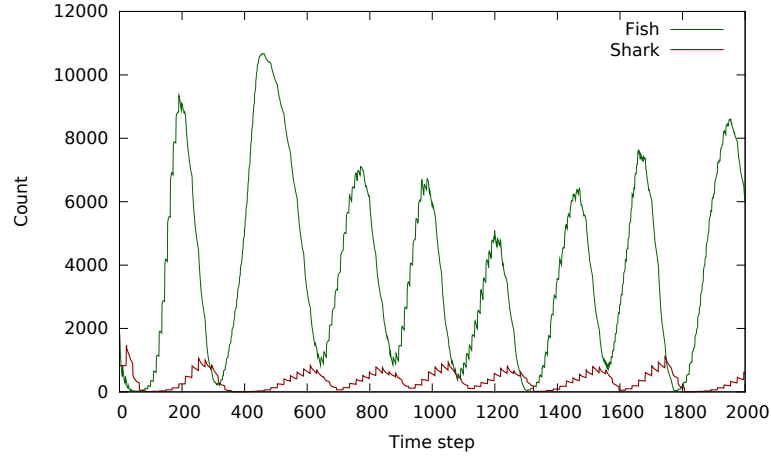


Figure 1.31: A graph showing the population of fish and sharks. This graphs the data from the simulation shown in figure 1.30. The graph lines are saw-toothed because the individuals all tend to breed simultaneously; fish every 8th time step and sharks every 20th. The fish curve gets smoother as time progresses because fish trapped on all sides will delay breeding until there's room enough, thereby also delaying the breeding schedule for all descendants. Sharks almost always have room to breed.

The results are graphed in figure 1.32, and as we can see, there is a gradual change happening between the smallest lattice size and the largest. In the smallest, only one of ten simulations had any shark alive after 100 steps. In the largest, only one of ten didn't, and only one more didn't make it to the end of the simulation at 1000 steps.

It might be tempting to conclude from this data that threatened species that have a predator-prey relationship need large natural reserves for them to have any chance of surviving. However, while there may be some truth to this thinking, the model is far too coarse for us to apply the results obtained through it to the real world. Here are some ways in which we could have improved on it, and as the attentive reader will realize, some of them would probably have helped the sharks in the small lattice survive in the long run:

- Get actual data for breeding ages of both species.
- Also get data for how long a predator can live without prey.
- Take into account that time to first breeding and time to consequent breedings might not be the same.
- Give the predators a finite stomach size.
- Have the individuals die of old age at some point.
- Factor in how the prey might hide from or successfully escape the predator.
- Have the prey gather in groups, as they often do in nature, to protect themselves from predators.
- Factor in the influence of seasons.
- Add additional types of prey that the predator might catch and eat if it is hungry enough, or seasonal prey that sometimes ease the pressure off of the primary prey.
- Add additional types of predators or omnivores that may target the same type of prey as the main predator.
- Have some upper bound on the number of prey, as they will not grow totally without bound in nature.
- Give the predators some sense of smell or sight so they may target prey and move about more purposefully.

We will come back to the last point of this list in section [2.4.4](#).

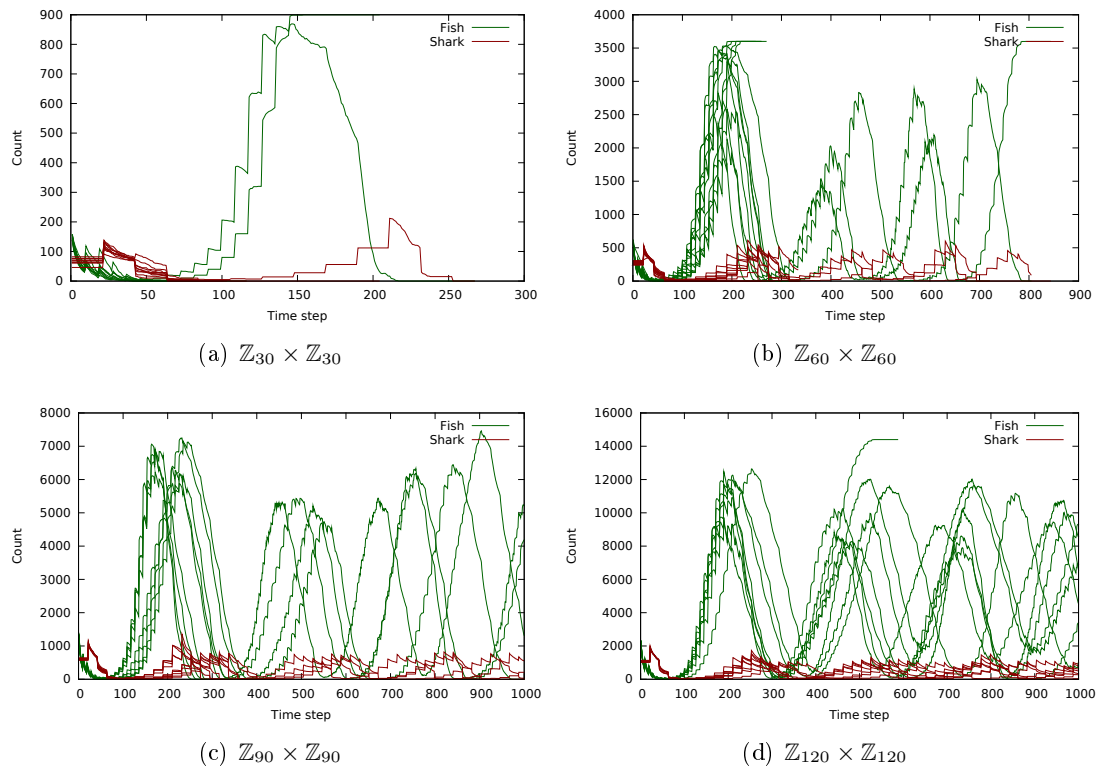


Figure 1.32: 10 runs of four different lattice sizes. The parameters and initial configurations are as described in the caption of figure 1.30

Chapter 2

Higher Order Cellular Automata

2.1 Introduction

In section 1.4.2 we created a very complicated radius 1 generic CA from a simple radius 4 totalistic CA, and observed that the simple rule produced a pattern much better suited for human eyes than the complicated equivalent. The logical connections of the advanced rule were hidden from human eyes. It is natural then to consider how we can put a greater number of the very many complicated CAs¹ into equivalent simple CAs that are easier to study and understand. To do that we need to extend the concept of a CA, but do so in a way that is structured easily reduced into its component pieces, so that the whole thing can be “picked apart” and studied.

According to [3], Torbjørn Helvik introduced the concept of higher order cellular automata [9], abbreviated HOCA, in 2001. He based it on the concept of hyperstructures introduced by Nils A. Baas in [1] in 1994. In brief, a HOCA is a CA with additional dynamics and/or hierarchical structure. With this framework it is possible for instance to combine several different normal CAs into one, or to make groups of cells that influence each other in some way, independent of the local transition function.

We will see that this enables us to design (randomly or with purpose) and understand CAs whose explicit form LTF would have been close to undecipherable. It uncovers a treasure trove of very interesting and unexpected behaviour, and gives us tools we may use to study simple CAs. Finally, we may use HOCAs to meaningfully alter the behaviour of existing CAs to suit our needs.

¹What we mean here by “complicated CA” is a CA with a large number of states and/or a large neighbourhood, and a LTF that seems only to be expressible as a lookup table; that is, there seems to be no formula or algorithm that does the same job. These kinds of CA are naturally extremely plentiful, and a small subset of them probably do some very neat things.

2.2 Definition

In this section we will use the definitions of [2] with some small changes in notation. We start with giving some already defined symbols a little subscripted number: Let \mathcal{L}_1 be a lattice, S_1 be an alphabet and $\mathbf{a}_1 : \mathcal{L}_1 \rightarrow S_1$ be the state configuration of \mathcal{L}_1 . In all these symbols, the subscript 1 refers to the first *morphological level* of the HOCA. A morphological level is what we call each “floor” of the hierarchical structure of the HOCA. We may drop the subscript if there is only one such morphological level.

2.2.1 2-dynamics

In a normal cellular automaton, every cell of the lattice has the same neighbourhood and the same local transition function. Higher order cellular automata with 2-dynamics may assign different neighbourhoods and LTFs to each cell. We now introduce the necessary concepts to deal with this.

Let $\mathcal{N}_1 = \{N_1, N_2, \dots, N_k\}$ denote the set of all the possible neighbourhoods that a cell might have. Then we may define a map that maps each cell to the neighbourhood it uses, $\mathbf{n}_1 : \mathcal{L}_1 \rightarrow \mathcal{N}_1$. The function \mathbf{n}_1 is called the first level *neighbourhood configuration*, and the set \mathcal{N}_1 is the first level *2-neighbourhood*.

Now we must assign one local transition function to each cell. However, the LTF of each cell must be “compatible” with the neighbourhood of that cell, meaning that if a cell has the neighbourhood N_x , then the LTF of that cell must have $S^{|N_x|}$ as domain. To facilitate this, we define a set

$$M = \{|N| \mid N \in \mathcal{N}_1\}$$

In words, M is the set of the different cardinalities that all the possible neighbourhoods have. We now define the first level *2-rule* $\mathcal{R}_1 = \{f_1, f_2, \dots, f_k\}$ as a set of local transition functions such that

1. Each function is of the form $f_i : S_1^m \rightarrow S_1$ for some $m \in M$.
2. For each $m \in M$ there is at least one function $f_j \in \mathcal{R}$ with domain S_1^m .

This lets us divide the sets \mathcal{R}_1 and \mathcal{N}_1 into disjoint subsets according to the cardinality of the neighbourhood:

$$\begin{aligned} \mathcal{R}_1^m &= \{f \in \mathcal{R}_1 \mid f : S_1^m \rightarrow S_1\} \\ \mathcal{N}_1^m &= \{N \in \mathcal{N}_1 \mid |N| = m\} \end{aligned}$$

Now we may finally define the function $\mathbf{r}_1 : \mathcal{L}_1 \rightarrow \mathcal{R}_1$ that maps one local transition function to each cell. It must be such that

$$\mathbf{r}_1(\vec{z}) \in \mathcal{R}_1^m \text{ if and only if } \mathbf{n}_1(\vec{z}) \in \mathcal{N}_1^m \text{ for all } \vec{z} \in \mathcal{L}_1$$

To introduce the 2-dynamics, we need a function ϕ_1 that updates the functions \mathbf{n}_1 and \mathbf{r}_1 each time step, in the same way that \mathbf{a}_1 is updated by the LTF each time step in a regular cellular automaton. To put into mathematical notation the restriction that the functions \mathbf{n}_1 and \mathbf{r}_1 must always be compatible, we first define the target set Φ_1 of this function:

$$\Phi_1 = \{(N, f) \in \mathcal{N}_1 \times \mathcal{R}_1 \mid (N, f) \in \mathcal{N}_1^m \times \mathcal{R}_1^m \text{ for some } m \in M\}$$

For each $m \in M$ we define a function

$$\phi_1^m : S_1^m \times \mathcal{N}_1^m \times \mathcal{R}_1^m \longrightarrow \Phi_1$$

These functions may then be combined to form the *2-transition map* $\phi_1 = (\phi_1^m)_{m \in M}$.

2.2.2 2-morphology

We now go on to group the cells of the lattice \mathcal{L}_1 into *2-cells* or *organs*. To achieve this we define the second order lattice, \mathcal{L}_2 , and a map $M : \mathcal{L}_2 \longrightarrow \mathcal{P}(\mathcal{L}_1)$. For an arbitrary organ, let's say it is at $\vec{z} \in \mathcal{L}_2$, the set $M(\vec{z})$ contains the coordinates of the cells in \mathcal{L}_1 that are contained in that organ. We also define the reciprocal map $M^* : \mathcal{L}_1 \longrightarrow \mathcal{P}(\mathcal{L}_2)$, such that for an arbitrary cell $\vec{z} \in \mathcal{L}_1$, $M^*(\vec{z})$ is the set of organs² that contain the cell at \vec{z} :

$$M^*(\vec{z}) = \{\vec{x} \in \mathcal{L}_2 \mid \vec{z} \in M(\vec{x})\}$$

Each 2-cell has a single state, and the set of all possible states is the *second level alphabet* S_2 . The state of each 2-cell is given by the second level state configuration $\mathbf{a}_2 : \mathcal{L}_2 \longrightarrow S_2$. Now we might alter the domain of the 2-transition map so that the state of the 2-cells may influence it:

$$\phi_1^m : S_1^m \times S_2 \times \mathcal{N}_1^m \times \mathcal{R}_1^m \longrightarrow \Phi_1$$

Similarly we might make the rules in the 2-rule of the first level also take S_2 as an input³. At this point we may choose to do one of three things:

- We may define a function $f_2 : S_1^C \longrightarrow S_2$, where S_1^C is some kind of aggregate value of the 1-cells in the organ⁴. This means that we're using the 2-cell only as a storage mechanism for some aggregate information, which then influences the first level rule and 2-transition map one time step later.

²This means, of course, that a cell may be in several organs at once.

³If the same 1-cell may be in several 2-cells, we need to alter the functions slightly to take that into account. Exactly how this is done will vary from problem to problem, depending on what the HOCA should model or what its desired behaviour is.

⁴The set S_1^C need not be equal to the set S_1 ; it could be a subset of \mathbb{Z} and be the result of a sum, or a binary value indicating the presence or absence of some condition, or the product space of several such indicators.

- We may define a single second level neighbourhood N_2 and a local transition function $f_2 : S_2^{|N_2|} \times S_1^C \longrightarrow S_2$, giving dynamics to the second level.
- Lastly, we may bestow 2-dynamics on the second level as well, by defining
 - a second level 2-neighbourhood \mathcal{N}_2 ,
 - a second level neighbourhood configuration $\mathbf{n}_2 : \mathcal{L}_2 \longrightarrow \mathcal{N}_2$,
 - a second level 2-rule \mathcal{R}_2 ,
 - a second level rule configuration $\mathbf{r}_2 : \mathcal{L}_2 \longrightarrow \mathcal{R}_2$, and
 - second level 2-transition maps $\phi_2^m : S_2^m \times S_1^C \times \mathcal{N}_2^m \times \mathcal{R}_2^m \longrightarrow \Phi_2$.

How to build additional levels should now be evident. If we get as far as adding 3-cells to the model, we may of course modify the second level rules and 2-transition map to depend on the state of the 3-cells that the 2-cells belong to, just like we did with the rules and 2-transition map on the first level after adding the 2-cells.

2.3 Problems of optimization

We will see that we gain a lot of possibilities and flexibility with the concepts of 2-dynamics and 2-morphology, but sadly, there is also a price to be paid in terms of performance. There are obvious costs, like the need to store and retrieve rules and neighbourhoods from the 2-rule and 2-neighbourhood, and the added complexity of the programming⁵.

Obviously, CPU caching increases performance when working with small lattices, but even simulations with large lattices will take some advantage of the cache, since the data is stored sequentially, and is often brought into the cache along with data fetched from main memory in response to an earlier request for data. This advantage may be diminished to some degree for cellular automata on lattices in more than one dimension, and to a larger degree by HOCAs, where the program typically needs to jump around quite a lot to compute the next state of each cell, for reasons we will discuss below.

For an impression of what the cost of fetching data from memory is, here are the numbers for the Pentium M processor: it takes 3 CPU cycles to retrieve something from the L1 cache⁶, 14 cycles for L2 cache, and 240 for RAM access[7, p. 16].

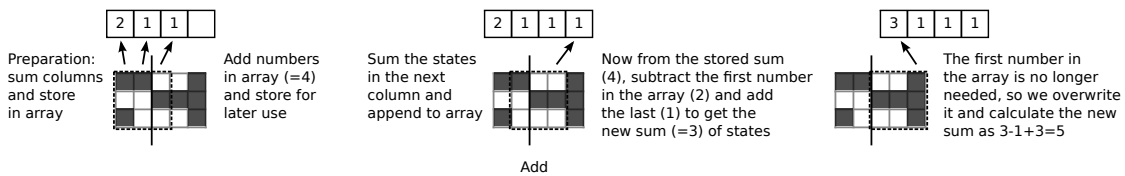


Figure 2.1: A simple way to optimize Game of Life computation performance. The vertical black line indicates where the lattice wraps.

A standard and straightforward way of optimizing the Game of Life (or any totalistic CA) is illustrated in figure 2.1. As a preparation to calculating a row of cells, one considers the neighbourhood of the cell located to the left of the first cell in the row. This is the rightmost cell, since the lattice typically wraps around. The sum of the three cells of each column of the neighbourhood is then calculated and stored in an array, and the sum of this array is stored too. Then, once in the loop of the program, it's only necessary to calculate the sum of the rightmost column of the new neighbourhood and store it in the array, add that number to the sum, and subtract the sum stored for the column that is now no longer part of the neighbourhood. Instead of accessing nine numbers from memory, this method needs only to access four numbers, one of which (the number to subtract) is almost certainly in cache⁷.

⁵Also, as this author has repeatedly struggled with, the dilemma of whether to write code that is optimized, or code that is readable, maintainable and extendable.

⁶The cache is typically hierarchical, with a very fast but small amount of memory inside the CPU itself, called the L1 cache, and a larger and maybe somewhat slower bank of memory called the L2 cache very near the CPU.

⁷As noted above, memory access may be just a little slower, or much much slower than the calculating speed of the CPU, depending on whether or not the required data is in cache.

For HOCAs with more than one possible neighbourhood, this optimization technique cannot easily be used. For totalistic HOCAs, the `rule` function⁸ would presumably call a `sum` member function belonging to the `neighbourhood` object or the `lattice` object, which would then calculate the sum from scratch and return it. Not only does this remove the ability to implement the above optimization technique, it also incurs the added overhead of a function call, which for some programming languages can be considerable.

A more advanced optimization technique applicable to many types of cellular automata was introduced by William Gosper in a 1984 article [8]. This technique has become known as hashlife when used to optimize the Game of Life. It stores the cell states⁹ in a tree representation called a *quadtree*, rather than in a matrix, to allow the lattice to expand as much as needed to fit the pattern of non-quiescent cells. This has the added advantage that it can be combined with hashing to *canonicalize* the contents, meaning every occurring state pattern needs only be stored once, and this pattern is then just pointed to whenever it occurs in the lattice. Lastly *memoization* is used, which means that the function computing the next time step of a part of the tree, stores the results for later use¹⁰.

Hashlife, properly implemented and optimized, can speed up the simulation by several orders of magnitude, and enable the simulation of huge lattices of Game of Life for “trillions of generations as they grow to billions of cells” [14]. Of course, the design of this algorithm hinges on a couple of assumptions: The underlying CA must be deterministic, or else memoization could not be used. The set of commonly occurring sub-patterns must be small enough to fit in memory, so that canonicalization actually gives an advantage. CAs without these properties would probably fare worse if implemented with Gospers algorithm rather than the naive one, since they would get the added overhead for canonicalization and memoization without reaping the benefits.

If one wished to implement a HOCA using this algorithm, the “state” of a cell, when viewed from the perspective of the algorithm, would need to include not only the state of the cell, but also its rule, its neighbourhood, and the states, rules and neighbourhoods of all its organs. This would of course dramatically increase the number of possible sub-patterns that the algorithm would need to cope with. For all but the most simple HOCAs then, this would make both canonicalization and memoization impractical, rendering also this optimization technique moot.

In summary, giving a cellular automaton 2-morphology or 2-dynamics may slow the computation down to a much larger degree than the extra necessary computation should warrant, making the study of large-scale HOCAs, be it in space or time, impractical, time-consuming or expensive.

⁸Here we are using the word “function” to refer to the programming construct, rather than the mathematical notion.

⁹More precisely: the states of the cells which are in the support of the configuration.

¹⁰Garbage collection is also needed to prune infrequently needed results from the stored results, thereby avoiding running out of memory.

2.4 Examples

2.4.1 1-dimensional cellular automata

We start out by looking at a simple variation of the first example given in Baas and Helvik[2], where instead of totalistic rules, we use the kind of rules presented in section 1.4.1. As an overview then, we have the following system:

Cellular Automaton 7		Radius 1 binary 1-HOCA with 2-dynamics
Lattice:	$\mathcal{L} = \mathbb{Z}$ or \mathbb{Z}_n	
Alphabet:	$S = \mathbb{Z}_2$	
2-neighbourhood:	$\mathcal{N} = \{\{-1, 0, -1\}\}$ (first neighbours)	
2-rule:	$\mathcal{R} = \{f_a, f_b\}$ (f_a and f_b as in section 1.4.1)	
2-transition map:	$\phi : S^3 \times \mathcal{R} \longrightarrow \mathcal{R}$ (totalistic)	

The 2-transition map has two parameters, c_a and c_b , and depends on the sum $s \in \mathbb{Z}_4$ of the states of the cells in the neighbourhood, and the current rule $f \in \mathcal{R}$:

$$\phi(s, f) = \begin{cases} f_a & \text{if } f = f_b \text{ and } s = c_b \\ f_b & \text{if } f = f_a \text{ and } s = c_a \\ f & \text{otherwise} \end{cases}$$

This simple HOCA enables much more complex behaviour than its constituent parts might suggest. As an example, let f_a be rule 255 and f_b rule 28. Rule 255 sets the cell state to 1 no matter the input, so it is natural to assume that the combined HOCA would behave somewhat similar to rule 28, since rule 255 seems not to add anything. But as we can see in figure 2.2, this is not the case.

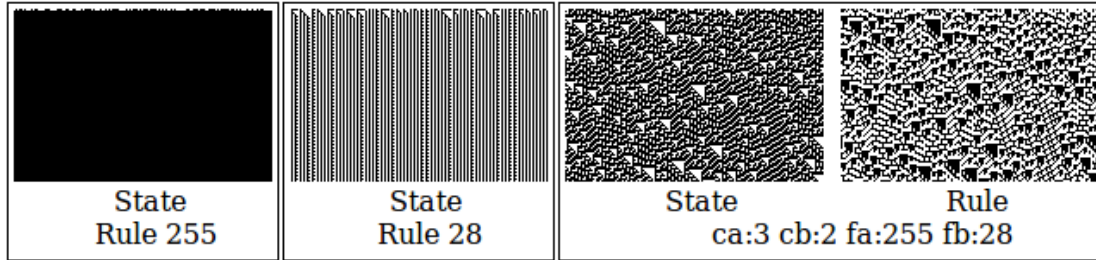


Figure 2.2: A combination of rule 255 and rule 28 where $c_a = 3$ and $c_b = 2$. Examples of how the two rules look by themselves are to the left, while the HOCA that is the combination as described above is in the rightmost frame. Since there is now also a pattern formed by which rule is applied in each cell, the type of pattern (State or Rule) is now specified under each pattern. Then the parameters for the HOCA as a whole is specified beneath. All cells have a random state and (where applicable) rule as initial configuration. Black is 1 and white is 0 for the state patterns, and black is f_b and white is f_a in the rule pattern.

Rule 255 is class I and rule 28 is class II, but as we can see, the combination seems to be class III or perhaps IV. The pattern also creates quite a few different patterns; the state pattern has diagonal stripes, triangles and vertical stippled stripes, while the rule pattern has all of these plus some brick-like pattern.

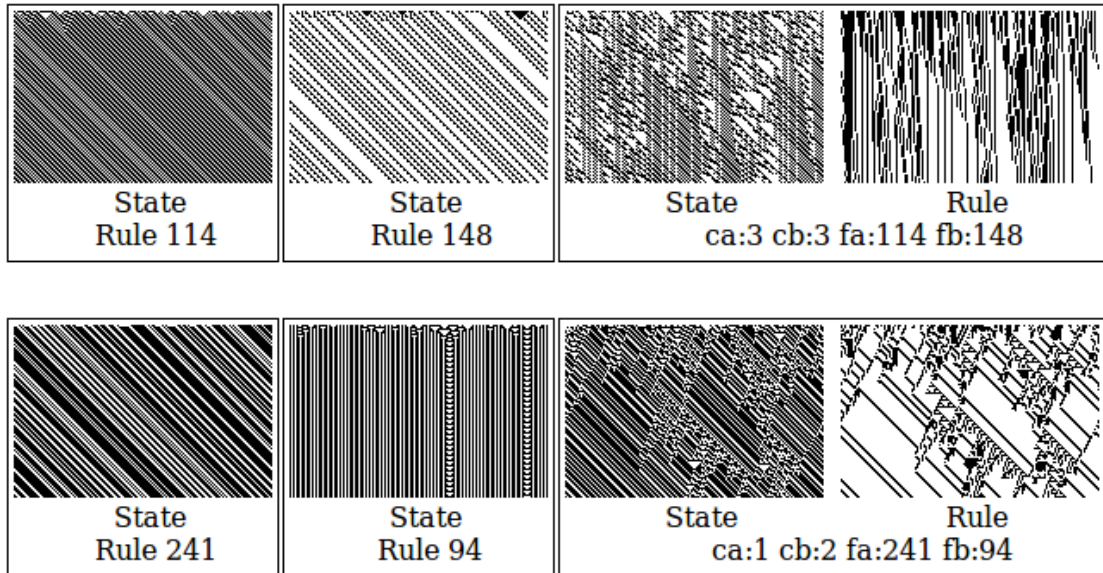


Figure 2.3: Is the randomness due to the random initial conditions or is it inherent to the HOCA?

Two other examples are pictured in figure 2.3. Both seem to exhibit complex behaviour, but one cannot help but wonder whether the complexity is due to the rule, or if maybe the random initial condition is the only source of complexity, and the rule is only shuffling it around. For most patterns exhibiting complex or chaotic behaviour examined by the author, such behaviour would also emerge when started from some very simple seed, like a single cell with state 1 that was surrounded by only state 0 cells¹¹. But some few appeared at first to exhibit only simple behaviour. However, most of them would show complex behaviour with some other simple input, like a state 0 cell surrounded by only state 1 cells, or some small number of state 0 cells in a row.

Shown in figure 2.4 is what happens to the HOCA in 2.3(a) if we start it with all cells set to state 1 with rule f_a , except one, which we give state 0. We can see that it is initially behaving in a very predictable manner, but as soon as the pattern has wrapped around completely, complex behaviour emerges.

Another specimen that was particularly hard to force into exhibiting non-simple behaviour is shown in figure 2.5. For most initial configurations, a pattern like 2.5(a) was

¹¹Many of these did not start behaving in a complex way until the information propagating left and right had met after wrapping around the lattice, which means that on an infinite lattice, they would have simple behaviour, if started from such a simple initial configuration.

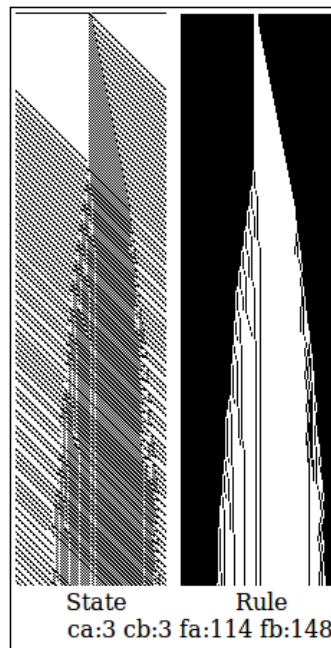


Figure 2.4: Complex behaviour arising because of a wrapping lattice.

the result, and such patterns stay predictable even on finite wrapping lattices. But for certain special initial configurations, some very different patterns would emerge, and for some, it even depended on the width of the lattice, as is apparent from figure 2.5(b) and 2.5(c). It is hard to believe that all these patterns could have been made by the same CA.

The HOCA in this example can also be used to extract or highlight patterns created by any other CA, by setting $f_a = f_b$. Now the pattern created will not be influenced at all by which rule is used, since they're both the same, but the rule pattern will show a pattern based on the evolution of the neighbourhood configuration of each cell. Some examples are shown in figure 2.6.

This HOCA has been a surprisingly rich source of different behaviours, and might be worth further study.

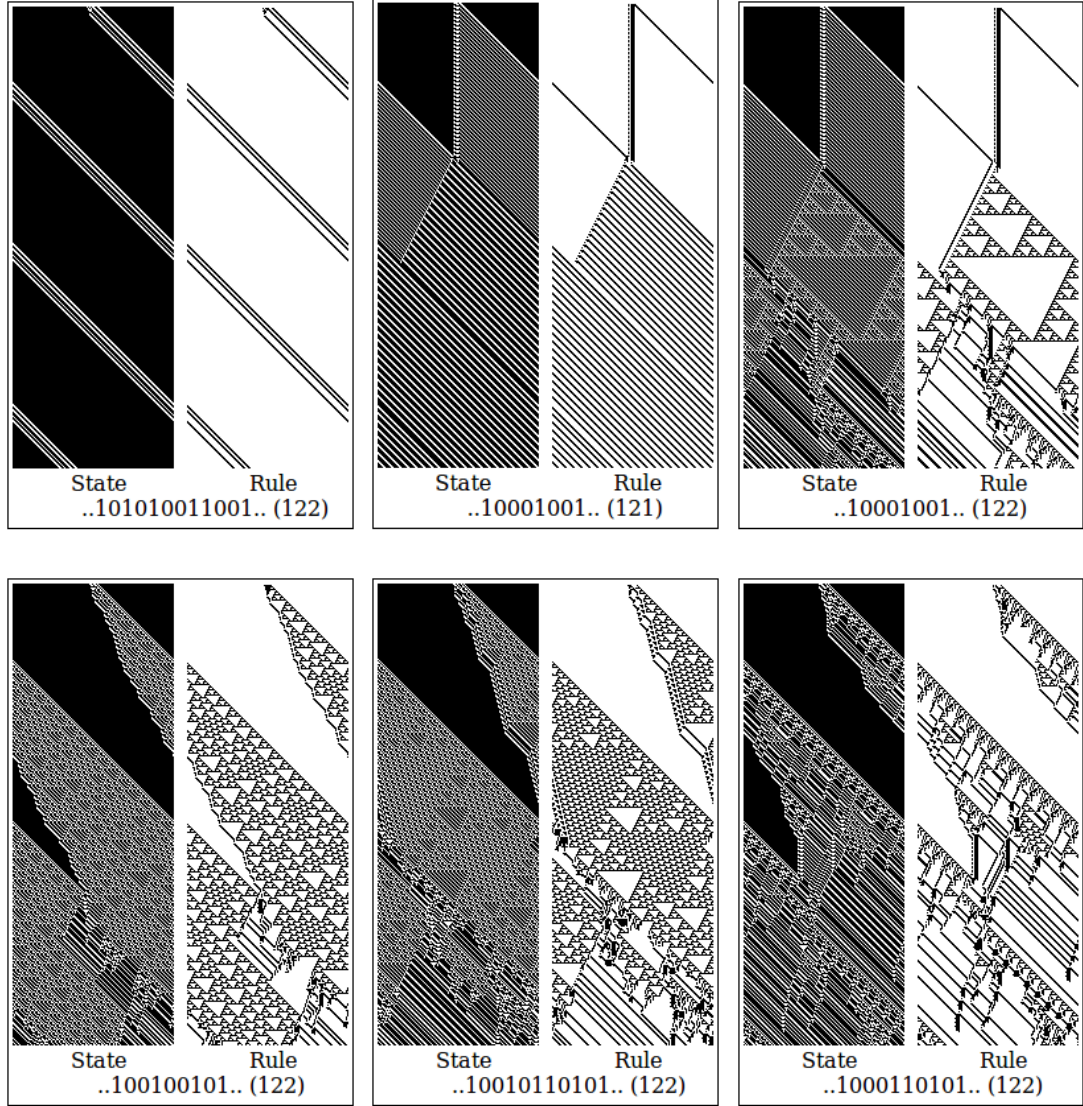
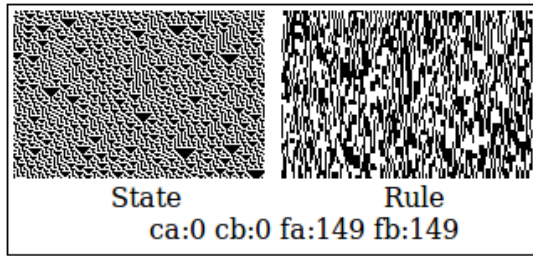
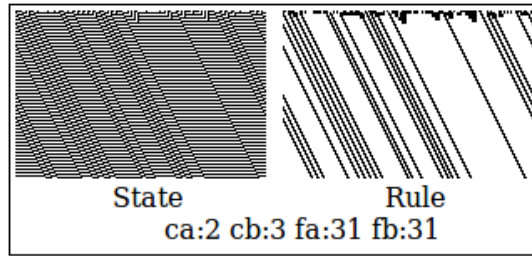


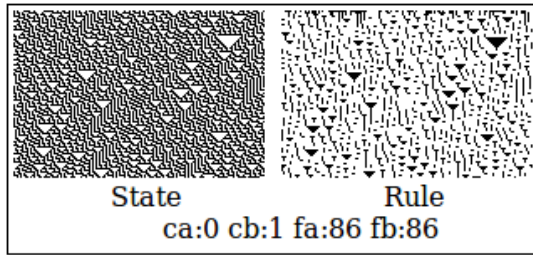
Figure 2.5: Complex behaviour arising out of simple initial conditions. All the images are of the same HOCA with $f_a = 241$, $f_b = 94$, $c_a = 1$, $c_b = 2$. Each image subtext specifies the pattern of initial states with which the initial configuration was made, placed in the “center” of a lattice that has state 1 (black) elsewhere. The number in paranthesis is the width of the lattice. The rule is initially set to f_a for all cells.



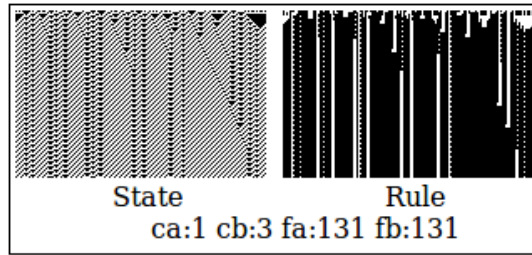
(a) Creating new pattern



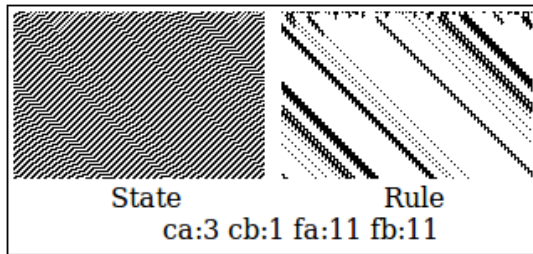
(b) Highlighting features



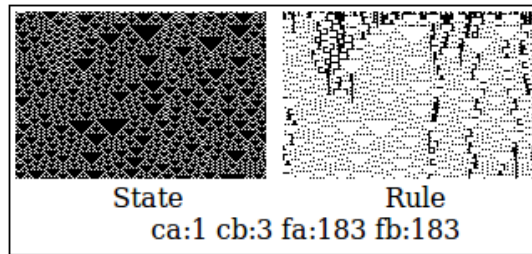
(c) Highlighting features



(d) Highlighting features



(e) Highlighting features



(f) Showing hidden features

Figure 2.6: Some examples of the HOCA presented in section 2.4.1

2.4.2 The Game of Life

With a general HOCA framework implemented it is rather easy to change an existing CA in some way. The following example shows how adding very simple 2-dynamics to the Game of Life will change its long-term behaviour, while still keeping it deterministic and behaving mostly as before on a short time scale.

When starting the standard Game of Life from a random initial configuration, there is at first very much activity, but soon much of it dies down and settles into blinkers or still life. It is inevitable that some activity starting somewhere else on the lattice crashes into these fixtures during its evolution, which of course causes it to behave differently than it would have, had the unmoving life not been there. What if we could remove the clutter if it was left alone for some time, so that the moving patterns could unfold naturally? We can use 2-dynamics to do this.

Cellular Automaton 8 Binary outer totalistic 2-CA

Lattice:	$\mathcal{L} = \mathbb{Z} \times \mathbb{Z}$ or $\mathbb{Z}_n \times \mathbb{Z}_m$
Alphabet:	$S = \mathbb{Z}_2$
2-Neighbourhood:	$\mathcal{N} = \{N_M\}$ (2-dimensional Moore neighbourhood)
2-Rule:	$\mathcal{R} = \{f_0, f_1, \dots, f_k, f_{k+1}, \dots, f_{k+l}\}$
LTFs:	$f_i = g \circ h : S^9 \longrightarrow \mathbb{Z}_9 \times \mathbb{Z}_2 \longrightarrow S$ for $i < k$ $f_i : S^9 \longrightarrow \{0\}$ for $k \leq i \leq k+l$
2-Transition map:	$\phi : S^9 \times \mathcal{R} \longrightarrow \mathcal{R}$

All the f_i LTFs for $i \in \{0, 1, \dots, k-1\}$ are the normal Game of Life LTF, completely unchanged. The rest of the f_i functions are, as is obvious from their codomain, the null function. The 2-transition map depends on the states of the cells in the Moore neighbourhood and the current rule. We reuse the function f from section 1.4.4 to define the 2-transition map:

$$\phi(s_0, s_1, \dots, s_8, f_i) = \begin{cases} f_{i+1} & \text{if } i < k \text{ and } f(s_0, s_1, \dots, s_8) = s_0 \text{ and } s_0 = 1 \\ f_{i+1} & \text{if } k \leq i < k+l \\ f_0 & \text{otherwise.} \end{cases}$$

So for a total of $k-1$ time steps, a cell that remains unchanged and alive will behave as in the normal game of life. But then in the k th time step, such a longlived cell will suddenly become dead, and remain dead no matter what happens around it for a further l time steps¹². After that, it will behave normally again. If the long-lived cell dies by normal means before k time steps have passed, it will start from “scratch” again and use f_0 as its local transition function, so even if it immediately comes alive again and stays that way, it will have another $k-1$ time steps before it’s killed off by our null functions.

¹²This is to kill off the very stable 2×2 squares that are so common – they will rebound if just one cell is killed at a time, so we need to wait until more than 2 are dead to actually remove such patterns from the lattice.

It's quite straightforward to implement this HOCA as a standard CA by changing the alphabet to something like $S = \mathbb{Z}_2 \times \mathbb{Z}_{k+l}$ and changing the LTF to also count the number time steps a cell has been alive. However, any CA with only 2-dynamics and not 2-morphology is conjugate to an ordinary CA – this is just a case that can very easily be seen as such. [2]

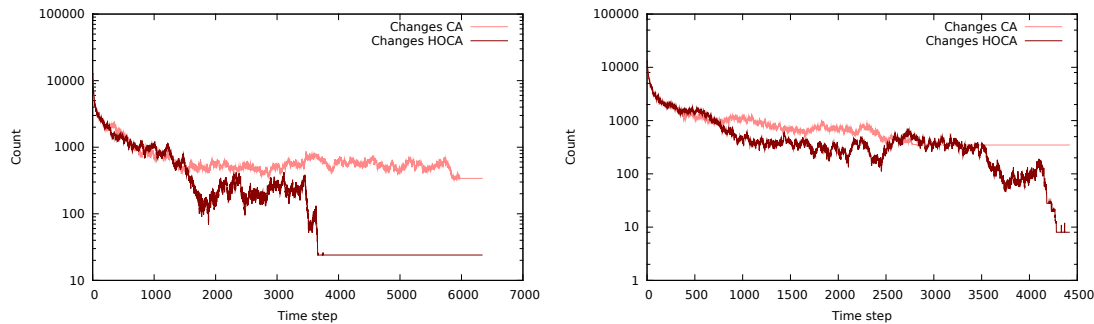


Figure 2.7: A couple of simulations with the same initial configuration for the traditional Game of Life CA and our HOCA variant. Both figures show a simulation on a $\mathbb{Z}_{200} \times \mathbb{Z}_{200}$ lattice. The initial configuration is constructed by setting the state of each cell to alive with 30% probability, and otherwise dead.

To study what effect the 2-dynamics has on the CA, we run the CA and the HOCA simultaneously with the same initial configuration, and for each time step, we count the number of cells that change from alive to dead or from dead to alive, and plot this number vs. time. The results of two such simulations are shown in the subfigures of figure 2.7. The HOCA always seems to end up with just some small number of gliders, while the CA stabilizes into the usual clutter of still life and blinkers – a glider would soon collide with the clutter, so there are no gliders left when the CA stabilizes. We haven't got enough information to draw any conclusions yet – we need more data.

So instead of plotting the number of changes vs. time, we make a note of the time step at which the number of changes stabilized for both the CA and its equivalent HOCA. Then we plot this data as in figure 2.8.

Here we see a definite tendency, which is that the HOCA will stabilize *sooner* than its equivalent CA. So the still life and blinkers are actually not preventing the moving life from living, but helping it stay moving for a longer period than it would otherwise have done.

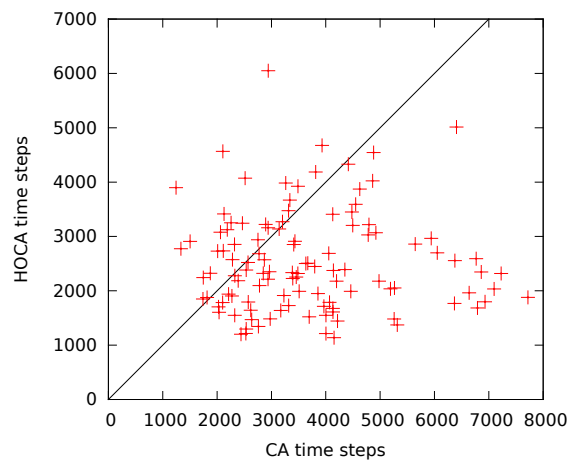


Figure 2.8: The time to stabilization for a Game of Life CA vs. its HOCA equivalent with the same initial configuration. The lattice and initial configuration are as described in figure 2.7.

2.4.3 Hodgepodge and activation-inhibition

The author has not managed to think of any ways these two CAs can meaningfully be given 2-dynamics or 2-morphology without breaking away from the purpose the CA was designed for. The hodgepodge machine was made to simulate a chemical reaction, and it does that job well. There are no hierarchies involved in the reaction, and using the same rule and neighbourhood for all cells is sufficient, more dynamics is not needed.

The activation-inhibition model was also made for a specific purpose which it fulfills nicely; to reproduce patterns found on snail shells. These patterns are produced by rows of glands around the base of the snail as they secrete the shell itself. The existing model obviously does a good job of it, and there are no obvious hierarchical structures we may add here either.

We therefore move on to the last example, which is much better suited for higher order extensions.

2.4.4 WATOR-World

Sharks are known for their excellent sense of smell, especially when it comes to blood, and they use this to find places nearby where other sharks have found food. We will add this behaviour to our WATOR-World CA by adding 2-morphology and altering the algorithm from section 1.4.6.

Cellular Automaton 9 A predator-prey simulation HOCA	
Lattices:	$\mathcal{L}_1 = \mathbb{Z} \times \mathbb{Z}$ or $\mathbb{Z}_{3n} \times \mathbb{Z}_{3m}$ $\mathcal{L}_2 = \mathbb{Z} \times \mathbb{Z}$ or $\mathbb{Z}_n \times \mathbb{Z}_m$
Alphabets:	$S_1 = \mathbb{Z}_3 \times \mathbb{Z}_{\max(b_f, b_s)+1} \times \mathbb{Z}_{s+1} \times \mathbb{Z}_2 \times \mathbb{Z}_2$ $S_2 = \mathbb{Z}_{9900} \times N_{vN}$
2-Neighbourhoods:	$\mathcal{N}_1 = \{N_{vN}\}$ (first neighbours) $\mathcal{N}_2 = \{N_{vN}\}$ (first neighbours)
2-rules:	$\mathcal{R}_1 = \{f_w\}$ $\mathcal{R}_2 = \{f_d\}$
Local Transition Functions:	$f_w : S_1^5 \times S_2 \longrightarrow S_1^5$ (asynchronous, algorithmic) $f_d : S_2^5 \times S_1^C \longrightarrow S_2$ (algorithmic)

We have appended $\times \mathbb{Z}_2$ to the alphabet of the first level to serve as the indicator of blood having been spilt in a cell during that time step. We will denote the new state of a 1-cell by (w, a, l, m, b) , where the first four are as before, while b is 1 if a fish was eaten in this cell this turn, and 0 if not.

As the first level lattice is 3 times larger than the second level lattice, we will use the functions 2.1 and 2.2 to map each 2-cell to the 1-cells it contains and vice versa. In these formulas, $\left\lfloor \frac{(x,y)}{3} \right\rfloor = (\left\lfloor \frac{y}{3} \right\rfloor, \left\lfloor \frac{x}{3} \right\rfloor)$.

$$M(\vec{z} \in \mathcal{L}_2) = \{\vec{x} \in \mathcal{L}_1 \mid \left\lfloor \frac{\vec{x}}{3} \right\rfloor = \vec{z}\} \quad (2.1)$$

$$M^*(\vec{x} \in \mathcal{L}_1) = \left\lfloor \frac{\vec{x}}{3} \right\rfloor \quad (2.2)$$

The second level state is a two-tuple that we'll denote by (c, d) , where $c \in \mathbb{Z}_{9900}$ is the concentration of blood in the water of that cell¹³, and $d \in N_{vN}$ is the direction of the greatest increase of blood concentration. We'll let π_c and π_d be the projections of the state set down to its factors, so that $\pi_c((c, d)) = c$ and $\pi_d((c, d)) = d$. The projections from section 1.4.6 we will alter so that they project from the new alphabet set, and we will add a projection for the new b indicator, which will then be $\pi_b((w, a, l, m, b)) = b$.

Now we are ready to list the modified first level algorithm, which we've called f_w . Unaltered portions are grayed out.

1. Denote the current state of the cell by (w, a, l, m, b) .

¹³9900 is a coarse upper bound for the concentration of blood in the water that can be achieved with the algorithm we'll use.

2. If $w = 0$, do nothing more for this cell.
3. If $w = 1$ and $m = 0$, then:
 - a) If $a \leq b_f$, then increase a by one, aging the fish.
 - b) Let $D = \{\vec{n} \in N_{vN} \mid \pi_w(\mathbf{a}_1(\vec{z} + \vec{n})) = 0\}$.
 - c) If $D = \emptyset$, the fish has nowhere to move, so update the state of the cell with the new age by setting $\mathbf{a}_1(\vec{z}) = (1, a, 0, 1, \mathbf{0})$, and do nothing more for this cell.
 - d) If $D \neq \emptyset$, then pick a $\vec{n}_x \in D$ at random.
 - e) If $a = b_f$ then the fish will breed, so we set $\mathbf{a}_1(\vec{z} + \vec{n}_x) = (1, 0, 0, 1, \mathbf{0})$ and $\mathbf{a}_1(\vec{z}) = (1, 0, 0, 1, \mathbf{0})$.
 - f) If $a < b_f$, the fish simply moves, so we set $\mathbf{a}_1(\vec{z} + \vec{n}_x) = (1, a, 0, 1, \mathbf{0})$ and $\mathbf{a}_1(\vec{z}) = (0, 0, 0, 1, \mathbf{0})$.
4. If $w = 2$ and $m = 0$, then:
 - a) If $a \leq b_s$, then increase a by one, aging the shark.
 - b) If $l = s$, then the shark dies from hunger, so set $\mathbf{a}_1(\vec{z}) = (0, 0, 0, 0, \mathbf{0})$, and do nothing more for this cell.
 - c) If $l < s$, then increase l by one, making the shark hungrier.
 - d) Let $D = \{\vec{n} \in N_{vN} \mid \pi_w(\mathbf{a}_1(\vec{z} + \vec{n})) = 1\}$.
 - e) If $D \neq \emptyset$, then pick a $\vec{n}_x \in D$ at random, and set $l = 0$ and $b = 1$, since the shark will eat the fish at $\vec{z} + \vec{n}_x$.
 - f) If $D = \emptyset$, then let $D = \{\vec{n} \in N_{vN} \mid \pi_w(\mathbf{a}_1(\vec{z} + \vec{n})) = 0\}$ instead, and set $b = 0$, since the shark did not find any food.
 - g) If $D = \emptyset$ even now, the shark has nowhere to move (*very* unlikely), update the state of the cell with the new age and hunger by setting $\mathbf{a}_1(\vec{z}) = (2, a, l, 1, b)$, and do nothing more for this cell.
 - h) If $l \neq 0$, then:
 - i) Let $\vec{n}_p = \pi_d(\mathbf{a}_2(M^*(\vec{z})))$.
 - ii) If $\vec{n}_p \in D$ then let $\vec{n}_x = \vec{n}_p$.
 - iii) If $\vec{n}_p \notin D$ then let $\vec{n}_x = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \vec{n}_p$.¹⁴
 - iv) If $\vec{n}_x \notin D$ then let $\vec{n}_x = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} \vec{n}_p$.
 - v) If $\vec{n}_x \notin D$ then let $\vec{n}_x = \begin{bmatrix} -1 & 0 \\ 0 & -1 \end{bmatrix} \vec{n}_p$.
 - i) If $a = b_s$ then the shark will breed, so we set $\mathbf{a}_1(\vec{z} + \vec{n}_x) = (2, 0, l, 1, b)$ and $\mathbf{a}_1(\vec{z}) = (2, 0, 0, 1, \mathbf{0})$. The baby shark is born with a full stomach.

¹⁴This amounts to rotating the vector \vec{n}_p in a counterclockwise direction. The next step then attempts a clockwise rotation instead, and the last one is a 180° turn. Thus, when there is a definite gradient in the 2-cell and no food around, the movement of the sharks is wholly deterministic.

- j) If $a < b_f$, the shark simply moves, so set $\mathbf{a}_1(\vec{z} + \vec{n}_x) = (2, a, l, 1, \mathbf{b})$ and $\mathbf{a}_1(\vec{z}) = (0, 0, 0, 0, 0)$.

To summarize the changes: we added a bloodspill indicator to a cell whenever a shark eats a fish, and we made the shark swim in the direction of the gradient acquired from the 2-cell, if it could not find food. If unable to swim in that direction, it would try to go around the obstacle.

The second level algorithm, which we've called f_d is rather more straightforward. We'll define the aggregate $k(\vec{z}) \in S_1^C$ of the 1-cells in each 2-cell as $k(\vec{z}) = \sum_{\vec{x} \in M(\vec{z})} \pi_b(\mathbf{a}_1(\vec{x}))$. Then we do the following for each $\vec{z} \in \mathcal{L}_2$:

1. Let $\vec{n}_g \in N_{vN}$ be a vector such that $\pi_c(\mathbf{a}_2(\vec{z} + \vec{n}_g)) > \pi_c(\mathbf{a}_2(\vec{z} + \vec{n}))$ for all $\vec{n} \in N_{vN}$ such that $\vec{n}_g \neq \vec{n}$.
2. If no such vector exists, then let $\vec{n}_g = (0, 0)$.
3. Let

$$c = \sum_{\vec{n} \in N_{vN}} \left(\frac{\pi_c(\mathbf{a}_2(\vec{z} + \vec{n}))}{1.1 \cdot 5} \right) + 100k(\vec{z}).$$

4. The next state of the 2-cell at \vec{z} will be (c, \vec{n}_g) .

What happens is that any shark killing a fish will cause a sharp increase of the blood concentration in the 2-cell containing that cell. After only one time step that concentration will have diffused completely across five 2-cells, so if a 2-cell has a concentration of 100 in time step 1, all cells in its von Neumann neighbourhood will have a concentration of 20 at time step two, barring other sharks eating nearby. That is not quite the truth however, because we also divide by a decay factor of 1.1, so that the actual concentration of all five cells at time step two is 18.

Because of the decay factor, the smell of an old kill will not linger for very long, and the concentration of blood in the water is bounded. A typical run of this new algorithm results in quite different behaviour than the original algorithm, as is shown in figure 2.9 and 2.10(a). What is more, the lattice size dependency that we noticed in section 1.4.6 is now very much diminished, as figure 2.10(b) shows clearly when compared with figure 1.32(a). The HOCA model reaches the 1000th time step all 10 times without the sharks dying out, compared with 0 times for the plain CA.

Whether this is a step in the right direction can be debated however. The population graphs of figure 2.10(b) no longer displays the characteristic predator-prey curves that the Lotka-Volterra model produces, which are known to occur in nature.

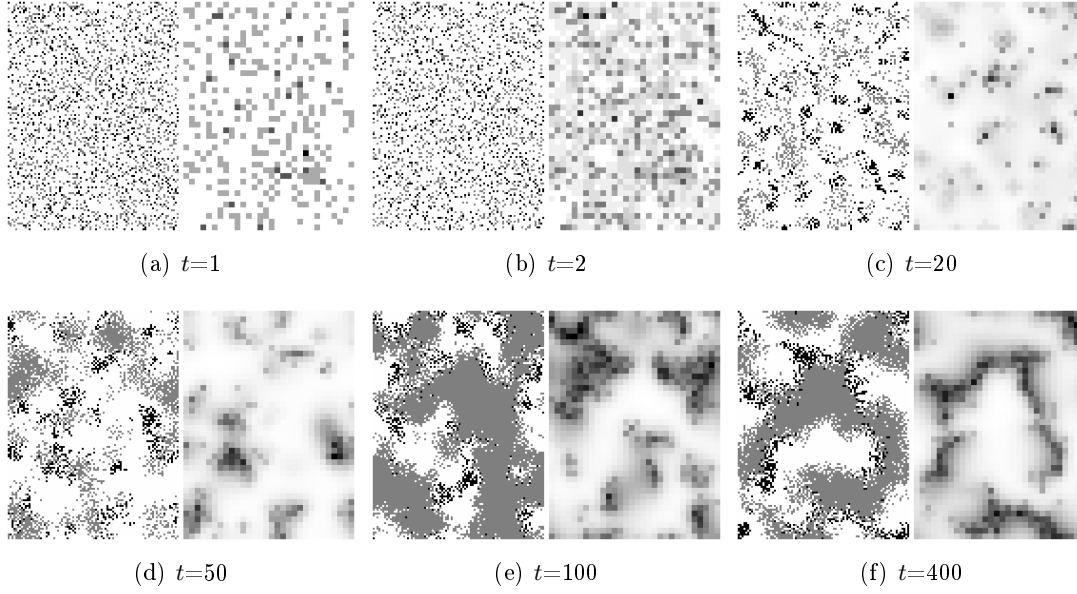
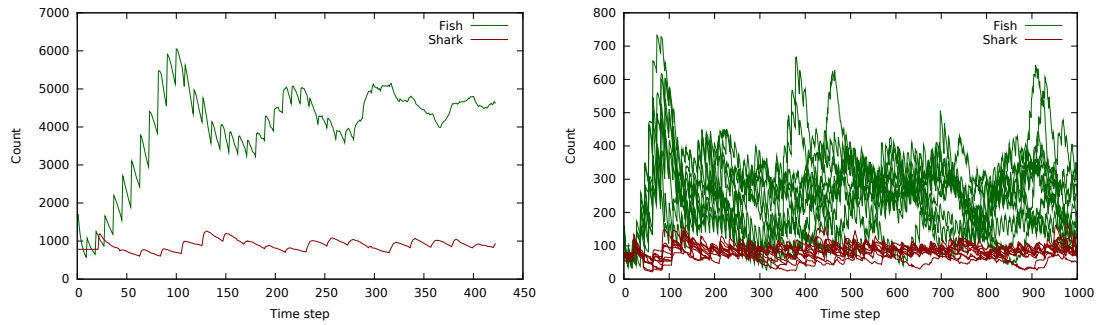


Figure 2.9: The evolution of the HOCA presented in this chapter. Compare with the simulation from section 1.4.6 in figure 1.30, and notice that the introduction of 2-morphology resulted in quite different behaviour. Apart from the modified algorithm and the rotation of the lattice by 90° , nothing is changed. The right part of each subfigure shows the relative concentration of blood in the 2-cells, where darker means more. Observe that the regions where shark and fish meet match up with the regions with high concentrations of blood.



(a) Population graph for simulation in figure 2.9.

(b) 10 simulations on a $\mathbb{Z}_{30} \times \mathbb{Z}_{30}$ lattice.

Figure 2.10: Population graphs.

Chapter 3

Conclusions

In this thesis we have given an introduction to the concepts of cellular automata and higher order cellular automata. We have seen some examples of the widely different and sometimes unexpected patterns that cellular automata can produce.

Then we went on to introduce higher order cellular automata, and considered how, while giving a lot of advantages, they also slow the implementation down to some degree because the presumptions that various optimisation techniques rest upon are no longer satisfied.

We have observed how some cellular automata are in a sense complete, and giving higher order structures or 2-dynamics to them seems unnatural. Other cellular automata very naturally accept higher order structures, and can rather easily be extended to better approximate the real world phenomena the model is based upon to give better and more accurate answers.

Bibliography

- [1] Baas, N. A. [1994], Emergence, hierarchies and hyperstructures, *in* C. G. Langton, ed., ‘Artificial Life III, Santa Fe Studies in the Sciences of Complexity’, Vol. XVII, Santa Fe Institute Studies in the Sciences of Complexity, Addison-Wesley, pp. 515–537.
- [2] Baas, N. A. and Helvik, T. [2005], ‘Higher order cellular automata’, *Advances in Complex Systems* **8(2-3)**, 169–192.
- [3] Barker, D. M. L. [2008], A study of higher order cellular automata with examples, Master’s thesis, NTNU.
- [4] Buckingham, D. J. [1996], ‘My experience with b-heptominoes in oscillators’.
URL: <http://www.radicaleye.com/lifepage/patterns/bhept/bhept.html>
- [5] Delorme, M. and Mazoyer, J. [1999], *Cellular Automata, A Parallel Model*, Kluwer Academic Publishers.
- [6] DeSouza, K. [2006]. Presentation.
URL: http://doursat.free.fr/docs/CS790R_S06/CS790R_S06_6_Pattern_Formation2.pdf
- [7] Drepper, U. [2007], ‘What every programmer should know about memory’.
URL: <http://people.redhat.com/drepper/cpumemory.pdf>
- [8] Gosper, W. [1984], ‘Exploiting regularities in large cellular spaces’, *Physica D: Non-linear Phenomena* **10(1-2)**, 75–80.
- [9] Helvik, T. [2001], Emergence and dynamical structures of higher order, Master’s thesis, NTNU.
- [10] Kusch, I. and Markus, M. [1996], ‘Mollusc shell pigmentation: Cellular automaton simulations and evidence for undecidability’, *Journal of Theoretical Biology* **178**, 333–340.
- [11] McIntosh, H. V. [1987], *Linear Cellular Automata*.
URL: <http://delta.cs.cinvestav.mx/~mcintosh/comun/lcau/lcau.pdf>

- [12] Mollison, D. [1991], ‘Dependence of epidemic and population velocities on basic parameters’, *Math Biosciences* **107**, 255–287.
URL: http://sjsu.rudyrucker.com/~rudy.rucker/wolfram_review_AMM_11_2003.pdf
- [13] *Omniperiodic* [n.d.].
URL: <http://conwaylife.com/wiki/index.php?title=Omniperiodic>
- [14] Rokicki, T. G. [2006], ‘An algorithm for compressing space and time’.
URL: <http://www.drdoobs.com/high-performance-computing/184406478>
- [15] Rucker, R. [2003], ‘Review of a new kind of science’, *American Mathematical Monthly* pp. 851–861.
URL: http://sjsu.rudyrucker.com/~rudy.rucker/wolfram_review_AMM_11_2003.pdf
- [16] Schiff, J. L. [2008], *Cellular Automata*, Wiley Interscience.
- [17] Semeniuk, I. [2009], ‘Seven questions that keep physicists up at night’.
URL: <http://www.newscientist.com/article/dn18041-seven-questions-that-keep-physicists-up-at-night.html>
- [18] von Neumann, J. [1966], ‘Theory of self-reproducing automata’.
- [19] Wolfram, S. [2002], *A New Kind of Science*, Wolfram Media, Inc.

Appendix A

Application documentation

A.1 General documentation

This thesis includes a digital appendix with source code for the applications used to produce the results presented. Some of the applications are written only using the python programming language. Some are written in python but use python extension modules written in C for the heavy processing. Other applications are written using C++.

All applications depend only on free and open libraries. They were compiled and executed on the GNU/Linux Ubuntu operating system, and are only tested on such a system. They are *not* tested on any Windows system, and it will probably be much easier to just temporarily install Linux than to try to make the programs work on Windows. How to set up a Linux system and install the prerequisite packages is beyond the scope of this documentation. Resources for this are readily available on the internet. The Ubuntu distribution has a Windows installer that can be downloaded from <http://www.ubuntu.com/desktop/get-ubuntu/windows-installer>.

A.2 Prerequisites

These are the names of the Ubuntu packages required to compile and/or use the applications, as far as the author is aware:

python	python-numpy	python-wxgtk2.8	python-imaging
python-setuptools	python-dev	python-reportlab	g++
qt4-designer	qt4-dev-tools	libboost1.38-dev	

A.3 Vector graphics cellular automata

Folder: pdf-ca

The python-reportlab module is used to generate PDF figures of cellular automata. PDF graphics (or vector graphics) are better suited for displaying small lattices than pixel-based image formats like jpeg. In this thesis these figures are used for illustration purposes; only very basic cellular automata were illustrated using this program. Each figure is generated by a small python script that uses the `ca.py` library to generate the CAs and the `pdfdraw.py` library to draw them.

- Figure 1.1 on page 5 is generated by `rule30.py`.
- Figure 1.10 on page 18 is generated by `rule30-equivalents.py`.
- Figures 1.12, 1.13 and 1.15 is generated by `classes.py`.
- The right subfigure of figure 1.14 is generated by `rule-24-shifted.py`.
- Figure 1.16 is generated by `rule110.py`.
- The overview of all 88 radius 1 binary 1-CAs in appendix B is generated by `similar-1d-rules.py`.

A.4 CA Explorer

Folder: ca-explorer

The CA Explorer program has a user interface written in the python programming language, and uses a python extension module written in C as a backend. This backend does the heavy calculations necessary, while python deals with everything else. The backend needs to be compiled before the application will work, and the script called `compile.sh` in the `cca` subfolder will do so automatically when run. It needs to be executed once before running the application itself.

The program presents an interactive view of three different cellular automata: the radius 1 binary 1-CA, any totalistic 1-CAs, and the Kusch-Markus CA from section 1.4.3. It has the following controls:

- Scrollwheel scrolls up and down the pattern.
- Shift-scrollwheel scrolls left and right.
- Ctrl-scrollwheel zooms in and out.

In addition, the parameters for the CA can be adjusted either by manually typing in an alternative number and pressing enter, or by hovering the mouse over the parameter

field while using the scrollwheel. Holding down the shift button while using the scrollwheel like this will cause the numbers to change in smaller increments than they would otherwise do, while holding down control will cause them to change in larger increments.

The program will pick a rule or code at random for you if you click in the parameter box for the rule or code and hit the R button. To go back to a previously viewed rule or code, hit Ctrl-Z.

Finally, any portion of the CA can be saved to a PNG image using the File→Save menu choice, where you will first be asked to name the file to save. Then you have to type in which specific part you want saved, by specifying the x and y coordinates of the upper left corner of a box with width w and height h . The desired coordinates can be found by looking at the coordinates in the lower right corner of the program before saving, which show the coordinates of the mouse pointer at any time.

The figures below were generated with this program:

- Figure 1.19 on page 24.
- Figure 1.20 on page 25.
- Figure 1.21 on page 26.
- The subfigures in figure 1.22 on page 29.

A.5 Game of Life

Figure 1.24 in section 1.4.4 was generated with the Golly program, which is freely available from <http://golly.sourceforge.net>, using the RLE pattern for the caterpillar spaceship, available from http://www.yucs.org/~gnivasch/life/article_cat/caterpillar.zip.

A.6 Hodgepodge

Folder: hodgepodge

This C++ program was programmed using the Qt library. The author has not made it an interactive program like CA Explorer, but rebuilt it with various code changes to serve different purposes. The publicly available Qt “mandelbrot” demo served as a starting point for this program, so the copyright notices and license of the program files have been kept unchanged as required by the GNU GPL license, which the demo was licensed under.

Using the Qt Creator application it is relatively easy to change the behaviour of the program. Simply open the project file `hodgepodge.pro`, doubleclick on the file `renderthread.cpp`, and alter the commented constants defined in the upper part of that file. Then click the green “play” button to the lower left, and the program will compile and run, provided your changes did not introduce any errors.

- The state patterns in figure 1.26 on page 34 was generated by this program and composed into one picture with the GNU Image Manipulation Program (GIMP).
- If run without modification, this program will generate the subfigures in figure 1.27 on page 35. The initial configurations will be slightly different each time, as they are randomly generated.

A.7 WATOR-World

Folder: `wator`

This program is written in pure python. It is rather slow, but very flexible. The program generates images at specified intervals, and stores the population data to a `.dat` file each time step. The program also creates a custom `.p` file that can be executed by the *gnuplot* program to generate the population graphs.

There are several small programs that each execute one or more specific simulations. All use the `WatorInterface.py` library to display the CAs while the simulation(s) are running. The CAs are generated by the `WatorHOCA.py` library.

- The subfigures of figure 1.30 on page 40 and the data for the graph in figure 1.31 is generated by `wator.py`.
- The data for the graphs in figure 1.32 on page 42 are generated by `lattice-compare.py`.
- The subfigures of figure Figures 2.9 on page 61 and the data for the graph in figure 2.10(a) is generated by `wator-hoca.py`.
- The data for the graph in figure 2.10(b) is generated by `hoca-lattice-compare.py`.

A.8 HOCA

Folder: `hoca`

This program is a modified version of the WATOR-World program, and works in the same way, except that all the small python scripts below use the `interface.py` library to display the CAs and the `hoca.py` library to generate them.

- Figure 2.2 on page 49 is generated by `hoca-255-28.py`.
- The subfigures of figure 2.3 on page 50 is generated by `hoca-example2.py`.
- Figure 2.4 on page 51 is generated by `hoca-114-148.py`.
- The subfigures of figure 2.5 on page 52 is generated by `hoca-241-94.py`.
- Figure 2.6 on page 53 is generated by `hoca-example3.py`.

Appendix B

Radius 1 binary 1-CA

This appendix lists the 256 wolfram rules grouped as explained in section 1.4.1. They are sorted according to their wolfram number, and any rule producing an isomorphic pattern is named in the label, with a letter in parenthesis describing what transformation to apply to the displayed rule. There are three transformations: m, i and M, which mean mirror, inverse and mirror-inverse, respectively.

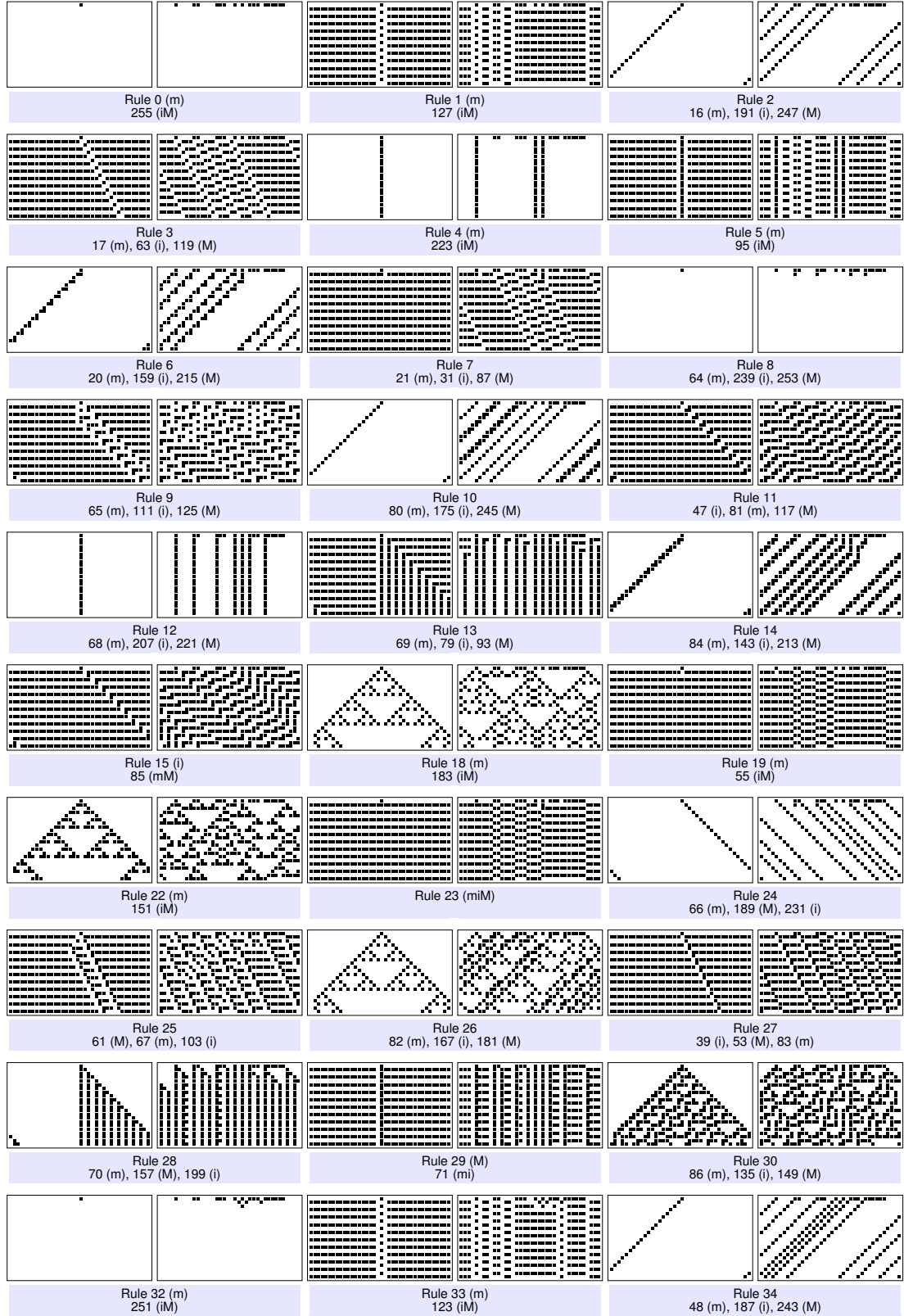


Figure B.1: The 1D cellular automatas, page 1

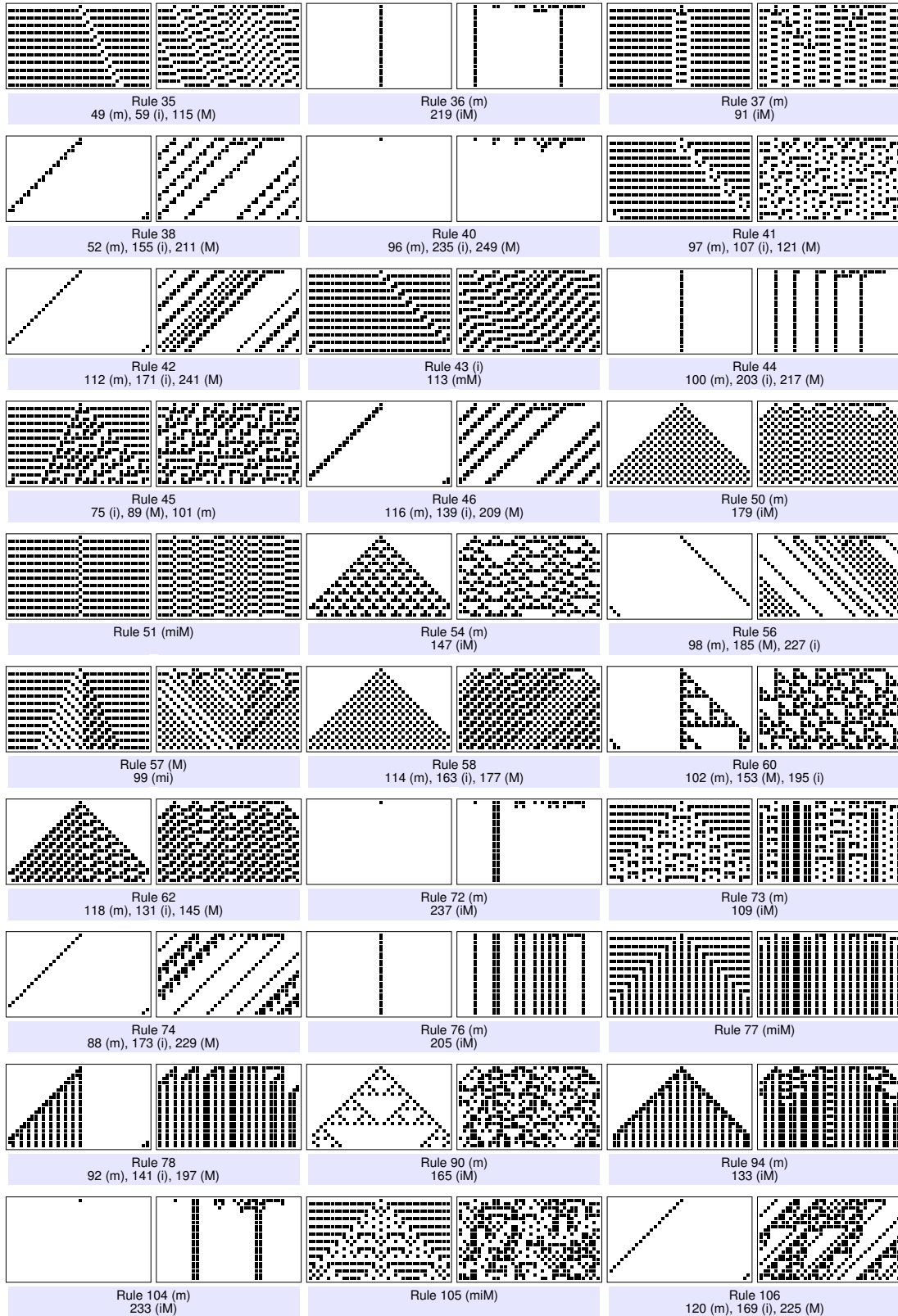


Figure B.2: The 1D cellular automatas, page 2

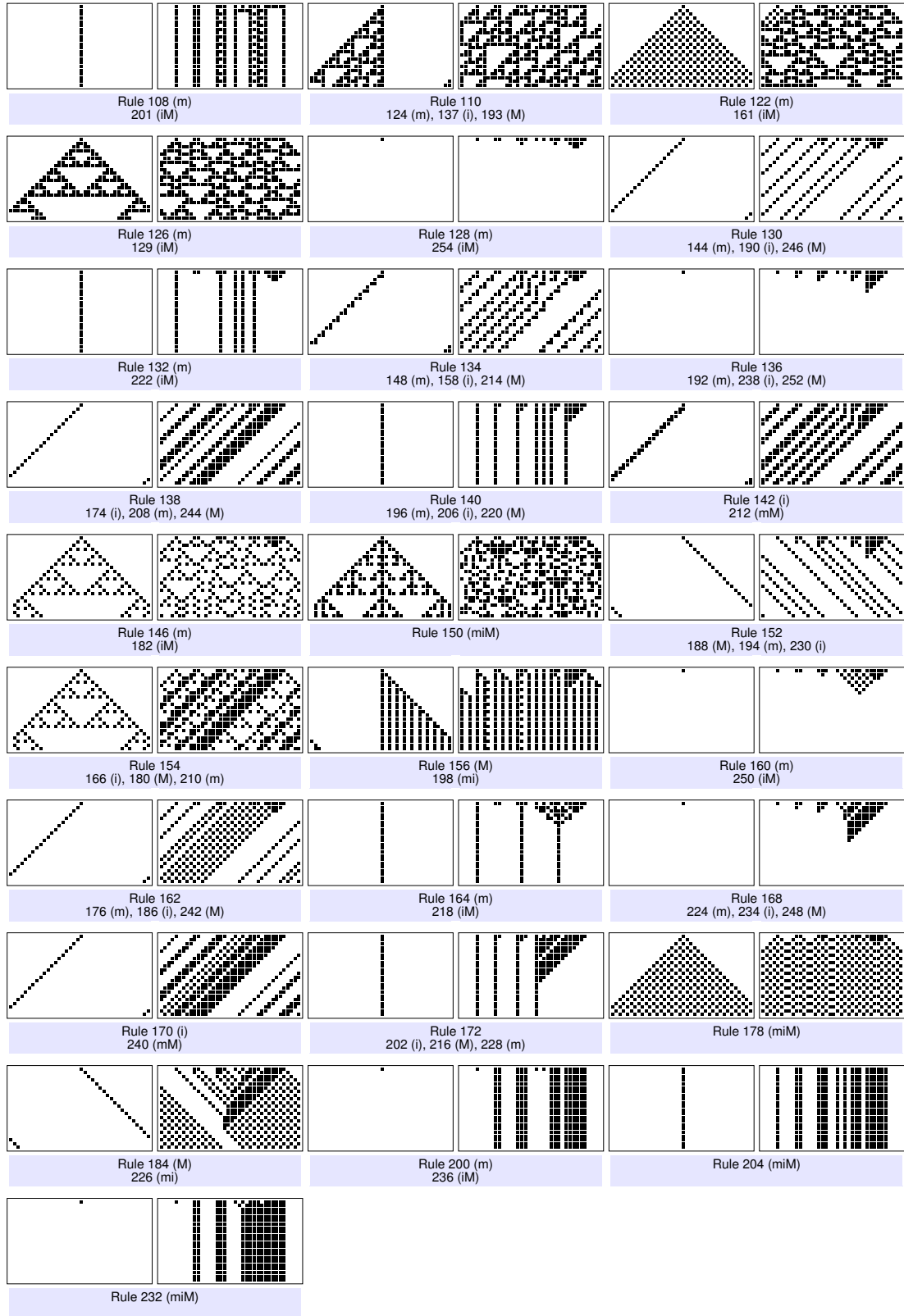


Figure B.3: The 1D cellular automatas, page 3

Index

- 2-cells, 45
- 2-neighbourhood, 44
- 2-rule, 44
- 2-transition map, 45

- \mathcal{A} , 12
- \mathbf{a}^t , 12
- A New Kind of Science, 11
- abbreviations, 1
- action, 13
- \mathbf{a}_i vs. \mathbf{a}^i , 13
- alphabet, 7, 12
- atto-fox problem, 37

- Baas, Nils A., 43
- behaviour, 13
- Belousov-Zhabotinsky, 33
- blinker, 31
- Buckingham, David J., 31
- Burks, Arthur Walter, 10

- C , 13
- CA, 1, 10
- canonical lookup table, 15
- canonicalizing, 48
- caterpillar, 31
- cell, 8, 12
 - alive, 30
 - dead, 30
 - healthy, 33
 - ill, 33
 - infected, 33
- cellular automata, 5–10, 12
 - 1-dimensional, 18
 - 3-state, 16
 - chaotic, 20
 - class I, 19
 - class II, 19
 - class III, 20
 - class IV, 21
 - classification system, 19
 - drawing by hand, 6
 - famous, 30
 - injective, 13
 - isomorphic, 18
 - labeling, 14
 - numbering, 14
 - outer totalistic, 17
 - programming, 11
 - recreational, 11
 - repeating, 19
 - reversible, 13
 - surjective, 13
 - survey of, 11
 - totalistic, 17
- cellular automaton, 1
 - higher order, 1
- Codd, Edgar Frank, 10
- code, 17
- codes
 - code 1599, 25
- computer algorithm, 12
- configuration, 8, 13
 - finite, 13
 - initial, 10
 - random, 34
- Conway, John, 10
- Cook, Matthew, 21
- cyclic space, 9

- d -CA, 12
- d_1 , 14

- Dewdney, Alexander Keewatin, 37
- differential equations, 37
- d_∞ , 14
- evolution, 13
- f , 12
- fish, 37
- follows, 13
- forest fire, 9
- formula, 12
- function, 8
- G , 13
- g , 17
- Game of Life, 9, 10, 30
- Garden-of-Eden configurations, 13
- Gardner, Martin, 10
- Gerhardt, Martin, 33
- glider, 30, 31
- global function, 13
 - bijective, 13
- global state, 8, 13, 19
 - homogenous, 19
 - initial, 13
- gnuplot, 68
- Gosper, William, 48
- h , 17
- hashing, 48
- Helvik, Torbjørn, 43
- higher order cellular automaton, 1
- HOCA, 1
- hodgepodge machine, 9
- IBM, 37
- inhibitor, 27
- instructions, 6
- Kusch, Ingo, 27
- \mathcal{L} , 12
- lattice, 7, 12
 - finite, 19
 - infinite, 19
- Life, *see* Game of Life
- limit set, 17
- local transition function, *see* LTF
- Lotka-Volterra, 37, 60
- LTF, 12
 - canonical lookup table, 15
 - explicit form, 15, 19
- manhattan norm, 1, 14
- map, 1
- Markus, Mario, 27
- Mathematica, 11
- maximum norm, 1, 14
- memoization, 48
- Moore neighbourhood, 16
- Moore, Edward F., 13
- morphological level, 44
- Myhill, John, 13
- N , 12
- N_M , 14
- N_r , 14
- N_{vN} , 14
- neighbourhood, 8, 12, 14
 - configuration, 8
 - first neighbours neighbourhood, 14
 - Moore neighbourhood, 14
 - radius > 1 , 23
 - radius neighbourhood, 8, 14
 - state, 8, 15, 16
 - sum of states in, 17
 - von Neumann neighbourhood, 14
- neighbourhood configuration, 44
- neighbourhood state, 8
- notation, 1
- organs, 45
- oscillators, 31
- \mathcal{P} , 1
- Palladium Oxidation, 33
- patterns
 - stationary, 21
- predator, 37
- prey, 37

- quadtree, 48
- quiescent state, 13
- quiescent states, 12
- rule, 8
- rules
 - inverted, 18
 - mirror, 18
 - mirror-inverse, 18
 - rule 28, 49
 - rule 0, 19
 - rule 110, 21
 - rule 12, 20
 - rule 135, 18
 - rule 149, 18
 - rule 150, 20
 - rule 160, 19
 - rule 24, 20
 - rule 255, 49
 - rule 30, 6, 18, 20
 - rule 32, 19
 - rule 4, 20
 - rule 40, 19
 - rule 8, 19
 - rule 86, 18
 - rule 90, 20
 - symmetric, 18
 - total number of, 16
- S , 12
- Schuster, Heike, 33
- Scientific American, 10, 37
- seashell patterns, 27
- second level alphabet, 45
- self-replicating automatic factories, 10
- set, 1
 - of all integers, 1
 - of non-negative numbers, 1
- sets of sets, 1
- sharks, 37
- simple programs, 11
- spaceship, 30, 31
- speed of light, 31
- state, 8, 10, 12
- states
 - more than 2, 23
- still life, 31
- support, 13
- traffic CA, 9
- Ulam, Stanisław, 10
- universal computation, 21
- vector, 1
- von Neumann, John, 10
- Wolfram|Alpha, 11
- Wolfram, Stephen, 11, 15, 17, 19
- $[x]$, 1
- $[x]$, 1
- \mathbb{Z} , 1
- \vec{z} , 1
- z_i , 1
- \mathbb{Z}_n , 1
- Z^+ , 1