Properties of Cellular Automata

John Tringham

Supervised by Andrzej Murawski

April 30, 2014

0.1 Abstract

This paper discusses the computational properties and behaviours of cellular automata, focusing on Wireworld, cyclic cellular automata and the design and implementation of a cellular automaton which portrays the workings of a classical Turing machine. Also discussed are Wolfram's elementary automata rules, a ruleset which solves mazes, and Smoothlife. This document also discusses the design and implementation of a cellular automata visualisation program (CAVP) made in Java by the author, which is used in order to gain insight into how these CA behave and evolve.

0.2 Keywords

Cellular automata, Wireworld, Turing machines, cyclic cellular automata, models

Contents

Part I

Introduction

Chapter 1

A brief introduction to Cellular Automata

Cellular Automata (CA) are discrete systems which run according to rules which govern the changes in states of each discrete cell in a regular grid. Different rules give different behaviours, with often the simplest of rules giving life to interesting and surprising behaviours [1, pp. 1-22]. A cellular automaton can formally be defined as a 4-tuple $\langle Z, S, N, f \rangle$ [2, p. 12][3], where

Z is the lattice of all cells, which may be infinite or finite. Each cell in this grid has exactly one value.

S is a non-empty finite set of values. These are also referred to as *states*.

N is the finite set of neighbours, which specifies cells whose positions are relative to a central cell. This very often includes the central cell itself. N is often referred to as the neighbourhood.

$f : S^{|N|} \to S$ is the transition function, which defines what value a cell will take in the next generation given the values of its neighbourhood. f is often referred to as the rule function, or more simply, *the rules*.

1.1 Neighbourhoods

The rules of cellular automata typically work only locally - they take in the current value of a specific cell c and all the cells in the *neighbourhood* of c, and define what value c should become in the next generation. The neighbourhood of a cell is a selection of cells that directly relate to that cell; it is typically (but not necessarily) the cells adjacent to the cell. The two main neighbourhoods that are used in cellular automata are the *Moore neighbourhood* and the *Von Neumann neighbourhood*, named after Edward E. Moore and John von Neumann respectively.

On a square grid lattice (which is where we'll stay during the course of this project), the Moore neighbourhood of a cell c are the 8 cells that share a corner with c; that is the cells that are directly adjacent to c and those that are on the diagonal. The Von Neumann neighbourhood is the neighbourhood of cells which are just directly adjacent in the 4 cardinal directions. Figure 1.1 shows the neighbourhoods diagrammatically.

0	1	2
3	c	4
5	6	7

	0	
1	c	2
	3	

Figure 1.1 – The Moore and Von Neumann neighbourhoods, left and right respectively. The numbers indicate how the cells in each neighbourhood have been indexed throughout this project. Both of these neighbourhoods technically also include c, although it has not been given an index as it is considered always accessable to the transition function.

While these are the neighbourhoods most cellular automata use, any arbitrary set of cells relative to c can be chosen to be the neighbourhood, and so can be tailored to the needs of the automata.

1.2 Grids

Cellular automata operate on any form of grid. Traditionally, this means an infinite 2D square lattice, but CA can operate on any regular graph[1], such as a triangular grid, hexagonal grid or higher dimensional lattices. It does not

necessarily have to be infinite either, it can be finite with different behaviours happening at the edges of the grid.

1.2.1 Torus-Based Grids

To emulate cellular automata on a computer, it becomes quickly apparent that simulating infinite grids will become a tricky thing to do, as most computers have only finite memory and processing power. This means that we will have to cut these grids down to a finite size. However, finite grids have edges, which are tricky to deal with as an edge in the traditional sense has nothing on the other side, and so these cells at the edge are going to have neighbours that do not exist. This causes a problem, as it means that these edge cells will have to act differently from the rest of the cells in the main body of the grid, and so will cause odd and unplanned behaviours.

One easy way to deal with this is to "bend" the grid. Imagine a piece of very flexible paper. If we take the left hand edge, and bend the paper and attach it to the right hand edge, we can create a tube of paper with no left hand edge and no right hand edge. Now if we take the top of this tube and bend round and attach it to the bottom of the tube, we have made a torus (a doughnut in layman's terms), which has no edges at all. Now every point on the original piece of paper still exists, but now all the points on the left hand side have a left hand neighbour, and similarly for all the other edges and corners. This is still finite, but have gotten rid of the edges.

We can take this method of gluing opposite edges together and use it for dealing with our grids. If we make the left edge neighbours those that are on the right edge, and vice versa, and similarly for the top and bottom edges, then we have turned our square grid into a torus where the edge cases work out in a natural way. This way, putting copies of the grid next to each other makes the automata tile, with cells flowing over onto the next copy seamlessly.

1.3 The Cellular Automata Visualisation Program

For this project, I have designed and created the Cellular Automata Visualisation Program (CAVP), a Java program which allows the user to simulate a wide

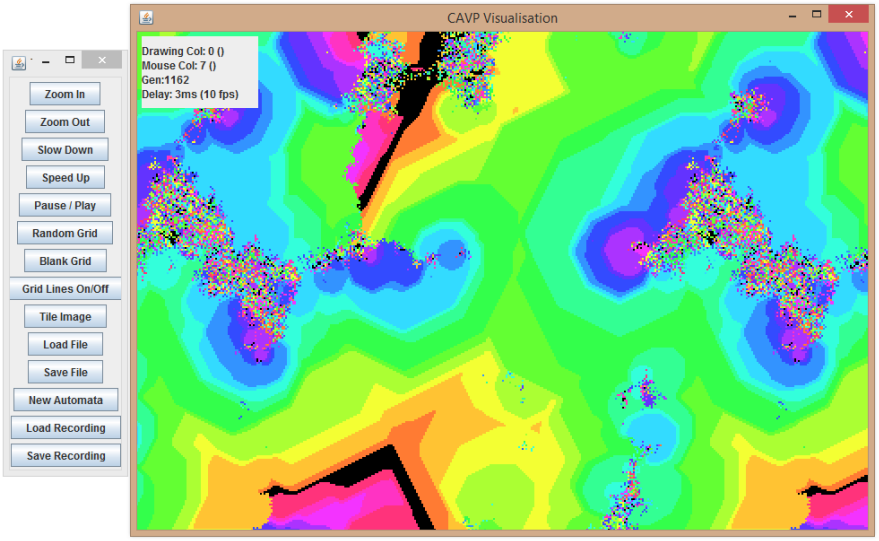

Figure 1.2 – The CAVP running an extended cyclic cellular automata after 690 generations starting from a random intial condition. To the right you can see the visualisation window, and to the left you can see the toolbox. In the top left of the visualisation window you can see the information pane.

variety of Cellular Automata. This program allows the user to save and load grid configurations, as well as allowing the ability to tweak each CA's parameters in order to aid exploring cellular automata properties and behaviours.

This program has been interesting to develop, and has been an incredibly useful tool in helping to discover properties and behaviours of cellular automata. **All images and videos of cellular automata included in this project have been rendered and created using this program**, in addition to a screen-capture program. The CAVP is available online for anyone to use, if you wish to, please read chapter 12, *Using The Program*. Videos of the CAVP are also available at `http://tringham.co.uk/zb/ca/`, which include demonstrations of using the program, as well as captures of some notable setups.

Chapter 2

About Cellular Automata

There are an infinite number of cellular automata rules. They can range from the seemingly incredibly simple to the vastly complex, with a sizeable proportion of producing significant behaviours.

Some cellular automata can be thought of as natural systems to be analysed: some rulesets are seemingly so simple that they often aren't thought of as discovered, rather they are thought to be merely investigated. Some simple rules occur in nature, such as the sea snail *Conus textile*, who's shell bares a remarkable resemblance to elementary rule 30 (see figure 2.1). Many biological, chemical and physical systems can be modelled by cellular automata, as they are thoroughly rooted in the idea of many discrete elements interacting with its immediate environment; from the way ants follow each other towards food,[4] to high level fluid dynamics[5], countless real life systems can be modelled using cellular automata.

However, cellular automata does not have to be limited to trying to imitate what already exists, it has the power to create and implement systems of its own, systems which do not occur elsewhere. Because its core definition is so simple and elegant and open, it has the power to create systems that are very rarely limited. Because of this, cellular automata become very important in the idea of theoretical computation, with them being incredibly useful tools in researching ideas about self-replication (in fact, this was the reason they were created [6, p. 1][7]), and have since broadened the horizons for many different academic areas, such as the study of fractals and cryptography[8][9][10].

This project aims to investigate the behaviours and the uses of some cellular automata, focusing on one 'natural' ruleset, one that is slightly more artificial, and one that is quite complex, those being cyclic cellular automata, Wireworld, and my own Turing machine automata respectively. Alongside these, this project briefly discusses several other cellular automata and inspects their own properties and behaviours.

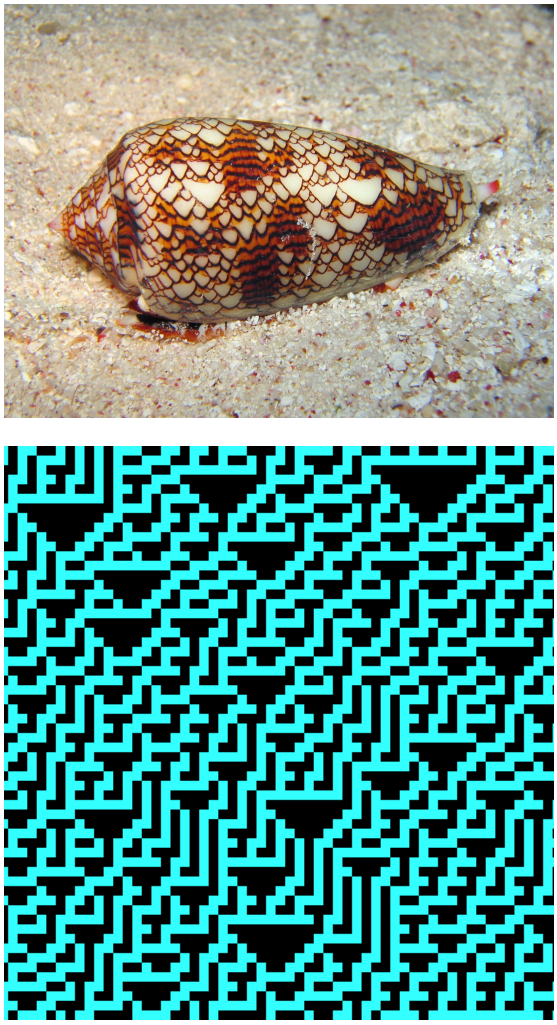

Figure 2.1 – Top: the shell of a *Conus textile*, bottom: rule 30. Image used with permission under a Creative Commons licence. Taken by Richard Ling.

Part II

The Rules

Chapter 3

Introduction

The rules of a cellular automata is the definition that gives it its behaviour. They are a function that takes in a cell and its neighbours and outputs the value of the cell that it will take in the next generation. These rules can be very complex or very simple, they can be designed with purpose or a complete lack of it, but regardless, they can create surprising behaviours.

We start off with the most basic of cellular automata: Wolfram's elementary cellular automata.

Chapter 4

Elementary Cellular Automata

4.1 Definition

Elementary cellular automata are systems whose inner workings are simple, but can create non-trivial and remarkable behaviours. They were first researched by Stephen Wolfram in the 1980's [1], and are notable theoretical systems.

One dimensional elementary cellular automata are very basic systems. First, we define the rule table, which is a function $\delta : \{0,1\}^3 \to \{0,1\}$, which defines the next value of a cell depending and its neighbours' values as well as its own. At every generation, these rule tables are applied to every cell.

These automata rules are very simple: there are only 256 different functions to chose from, and all we are doing is defining mapping and applying over and over. However, the interesting element here is how they evolve and what patterns they create.

Each rule is given a number based on its rule table. For instance, let's take rule 110. Rule 110's rule table is as follows:

111	110	101	100	011	010	001	000
0	1	1	0	1	1	1	0

If we always order the possible neighbourhoods in descending binary order (such as it is in the table) we can define any one dimensional elementary automata as just the bottom row of the table: in this case it is 01101110_2, which

when interpreted as a binary number is value 110 in base 10. This system allows us to express any elementary one dimensional automata as a unique number between 0 and 255.

These rules are very interesting as they caused a wave of new thinking about complex behaviour emerging from simple systems [1]. For example, figure 4.1 shows rule 110 creating a pattern starting from one single on cell surrounded by off cells, and figure 4.2 shows this after 500 generations. As you can see, a non-trivial pattern emerges, one that is not completely uniform, but not completely random either.

Figure 4.1 – Rule 110 after ten generations starting from one initial cell.

Figure 4.3 shows a selection of these rules and shows how they can vary quite dramatically, not only aesthetically, but in the way it interacts with itself. They are not merely patterns, they have interesting computational properties. Matthew Cook proved in 2004 that rule 110 is Turing complete[11], and so is able to compute anything computable. From this we can see that cellular automata itself is undecidable: as we know at least one CA is able to compute anything a Turing machine can compute, the question of whether a given cellular automata will enter a periodic repetition of states can be thought of as equivalent to the halting problem[11], which we know is undecidable. In comparison to rule 110, rule 30 creates chaotic aperiodic behaviour, which is so far from predictable that it is used as a random number generator in the popular software *Mathematica*[12].

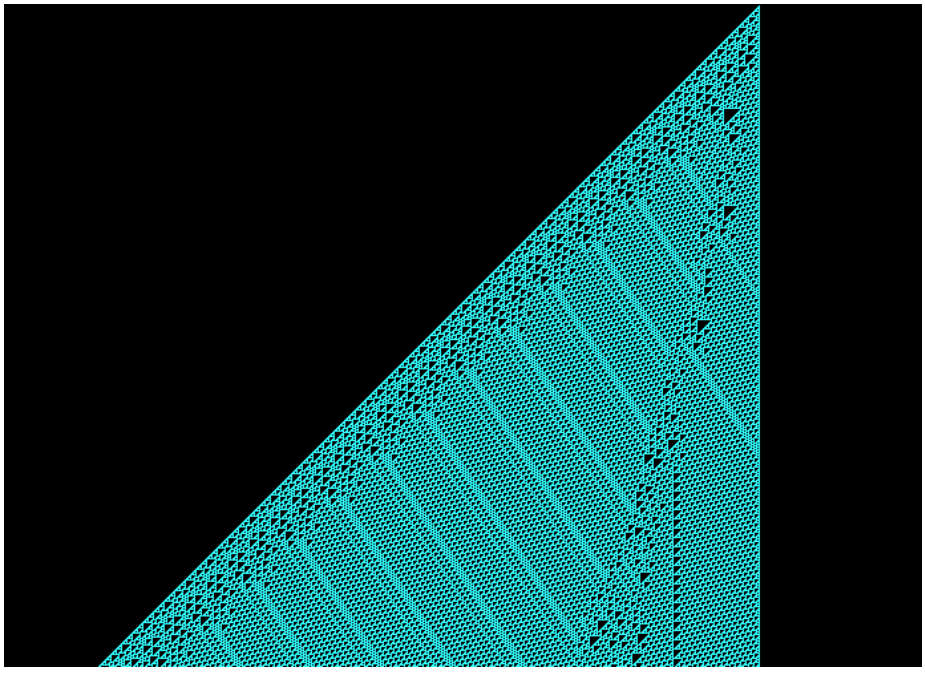

Figure 4.2 – Rule 110 after 500 generations starting from one initial cell. This diagram and all others of one dimension cellular automata displays each generation as a horizontal line of cells, with the successive generation appearing below it.

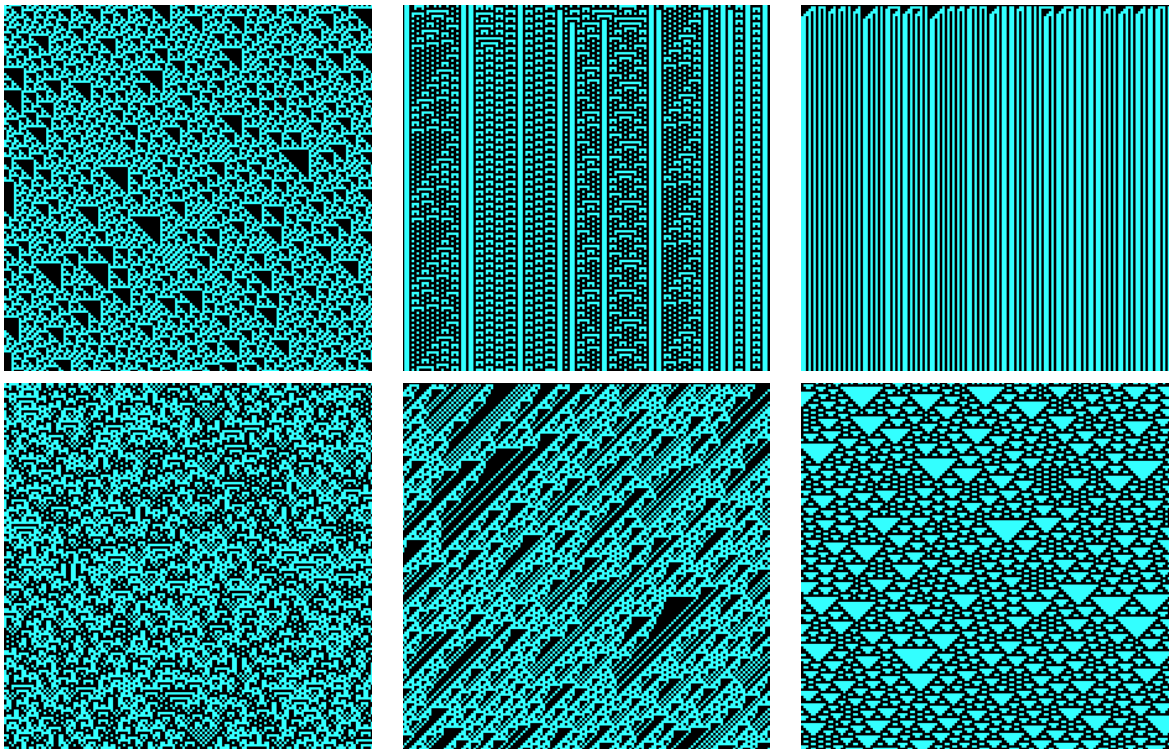

Figure 4.3 – A selection of different elementary rules, each starting from a random initial generation and lasting 150 generations. Clockwise from top-left, they are: rule 60, rule 73, rule 78, rule 129, rule 106 and rule 105.

Chapter 5

Cyclic Cellular Automata

One of the simplest cellular automata rules is the Cyclic ruleset. Given a cycle length n, each cell c of value v will become the value $v + 1 \pmod n$ if it has a neighbour of value $v+1 \pmod n$, and remains unchanged otherwise [13]. Figure 5.1 shows the this rule diagrammatically.

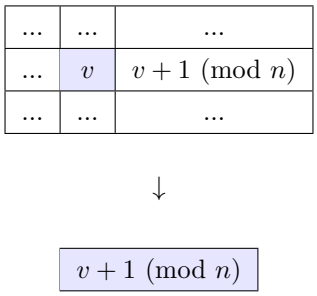

Figure 5.1 – A rule diagram for cyclic cellular automata for the Moore neighbourhood. This rule has a rotational symmetry of 8, meaning that the $v+1 \pmod n$ cell can be in any of the positions marked with an elipsis.

Cyclic automata, while simple, can create complex, and often quite aesthetically pleasing, behaviours.

The cyclic automata, starting from a random initial grid, goes through 4 distinct phases in order [14], which are:

Debris

The majority of cells are unchanged from the initial random condition,

as they are not neighbouring a cell of a value one higher than their own, and so sit unchanged. While small pockets of movement occur, the main body of the grid remains static and stays close to random state that they started with.

Droplets

The small pockets of movement that occurred in the debris stage start growing and moving, with gliding sections of colour moving over one another, taking over more of the area that before was static. This free flowing movement of cells eventually takes over the majority of the grid, with fewer and fewer cells remaining untouched by these changing areas of colour.

Spiral Formation

Eventually, it is likely that these droplets interact in such a way that for a cycle length n, there exist n adjacent cells that contain each of the possible values in ascending order that makes up a cycle (ie a path of neighbouring cells such that a cell with value 0 is next to a cell of value 1, which is next to a cell of value 2, and continuing until a cell of value $n-1$, which is neighbouring the original 0-valued cell). This causes each cell in this path to run through every possible value and start again every n steps. Because of this, this loop will cause a spiral to spread outwards, as every cell adjacent to a cell in this loop will become part of it (as it is guaranteed that a cell of one less value will neighbour it within at most n steps), and similarly every cell next to that. This causes spirals of increasing cells to spread out from the centre of these loops.

Demon Domination

These areas that are repeatedly changing their values from 0 to n and spreading out through the grid (refered to as 'demons' [14]) eventually take up the majority of the grid, and inevitably take over the entire area, causing the entire simulation to continuously cycle through every value in the 0-n range. The shapes that these demons take are highly dependent on what loop affected the cells first and the neighbourhood used. This final configuration repeats endlessly in period n.

5.1 Debris Stage

While these phases normally happen in order, depending on the cyclic length, the initial starting state and the neighbourhood that it is operating within, it may become stuck within one of the stages without reaching the next. For example, if the cyclic length n is large compared to the number of neighbours N, then it is likely that it will remain stuck in the debris state without significant droplets emerging. This is because the probability of a cell neighbouring its successor value is lower, and so the chance of a chain reaction diminishes, and so after possibly a small amount of initial movement the grid becomes static and repeating.

In the initial generation, the values of the cells are uniformly random over the range 0 to $(n-1)$. The initial probability of any one cell changing value is shown in (5.1).

$$\begin{aligned} &P(\textbf{cell } c \textbf{ of value } v \textbf{ has a neighbour of value } (v+1 \mod n)) \\ &= \\ &1 - P(c \textbf{ has no neighbour of value } (v+1 \mod n)) \\ &= \\ &1 - \left(\frac{n-1}{n}\right)^N \end{aligned} \tag{5.1}$$

It is quite trivial to see that this probability is highest when N is high and n is low. For typical values (which we will define now as cyclic length of around $n = 20$ and using the Moore neighbourhood, in which each cell has $N = 8$ non-self neighbours):

$$1 - \left(\frac{20-1}{20}\right)^8 \approx \frac{1}{3} \tag{5.2}$$

We can see that the initial change probability of any single cell is approximately $\frac{1}{3}$. Once two neighbouring cells have a difference in values of $\{-1, 0, 1\}$ mod n, this pair of cells can be called *bonded*, as it can be seen that once they have this property they will keep this property forever [14]. This means that, from a random start, if cell p changes due to a cell neighbouring cell q, these two bonded cells now act as a larger object where their values are now permanently

linked. These sets of bonded cells that alter each others values are referred to as *droplets*.

5.2 Droplet Stage

Droplets are sets of cells that for any given cells p, q in the droplet, either p and q are bonded, or there exists a chain of bonded cells between p and q. The values of cells in a droplet sweep across the droplet in waves, because of the very nature of bonding, and because of the fact that cell values can only change in one direction.

If we model droplets as polygons on the grid with all cells inside the polygon having only one value, we can determine how large these droplets need to be until we should expect a chain reaction, causing this droplet to expand endlessly, destroying all the debris until it either meets another droplet, is taken over by a demon, or engulfs the grid completely.

Given a droplet with perimeter C, the probability of every value appearing at least once on this perimeter is $P(n, C) =$[1]

$$\left(\frac{\sum_{k=0}^{n-1} (-1)^k \binom{n}{k} (n-k)^C}{n^C} \right) \tag{5.3}$$

As you can see in figure 5.2, for any given n, $P(n, C)$ is non-decreasing in C. That is, as the droplet size increases, unsurprisingly, the chance of it containing every single value increases, which means that once this state has been reached it is likely that this droplet will get larger and larger, taking over the grid (unless a demon occurs and destroys this growth). This property is important, as if it contains this, then eventually every cell on the perimeter will become bonded to another cell in the droplet, and so will become part of the droplet.

We can see this as if it contains this property then no matter what value the center of the droplet contains, then its successor state definitely neighbours it, and so the droplet colour changes to this successor, and then this repeats, with the next value controlling the central colour, and so on until the entire perimeter has been 'used up' by each in turn becoming the central colour. But at this point the droplet has expanded and made the new perimeter larger, meaning

[1] For the derivation of this formula see appendix A

it is now more likely to contain every single value than before. This series of events means that once a droplet has this property, it is likely to continue to expand outwards until it runs into something more dominant.

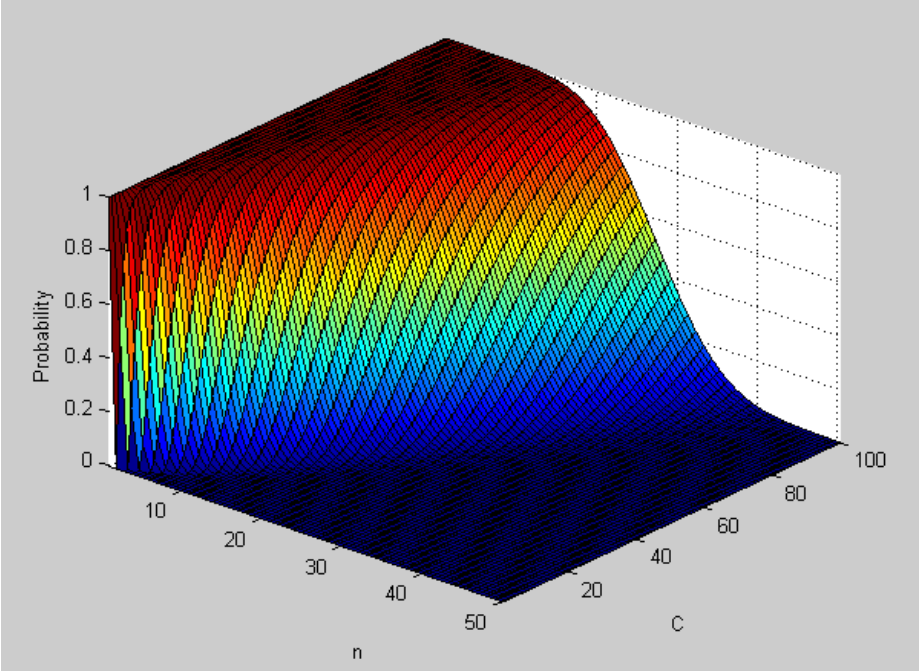

Figure 5.2 – A graph of (5.3) with values $1 < n < 50$ and $1 < C < 100$. Data generated using a Python program and plotted in Matlab.

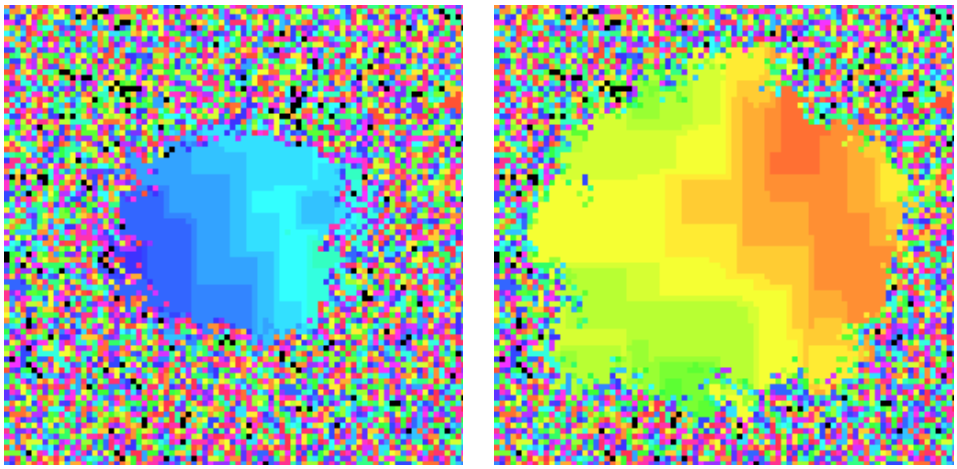

Figure 5.3 – Left: A naturally occurring droplet in a cyclic automata with cycle length 40 under the Moore neighbourhood after 475 generations. Right: the same droplet 245 generations later.

5.3 Spiral Formation

Once droplets have become the dominant bodies in the CA, it becomes increasingly likely that a spiral or 'demon' occurs. This happens when a cycle of kn (for some constant k) adjacent cells contain a loop of every possible value in ascending (mod n) order, and so in each generation each cell in this cycle becomes its successor immediately, causing an infinite loop of cells increasing.

If we go back to our definition of bonded cells, let's consider that inside a droplet D there exists a cell p such that there is a sequence $q_{(i)}$ of k cells, where $q_0 = q_k = p$ and $q_{i+1} - q_i = 1 \mod n$ for all i. This would trigger a spiral, as at every generation q_i would become the value of q_{i+1} for every i, and as it starts and ends at the same cell, they would loop.

This appearing inside a droplet that is simply expanding into debris is incredibly unlikely as it would require the debris to be in a very specific configuration. However, when two droplets collide with each other, this is more likely the case, as opposite 'waves' of values from each droplet hit each other and interact with each other in a way that causes these spirals, as can be seen in figure 5.4.

The creation of a spiral in this manor is as follows:

1. There are two droplets that now border each other, there is a line which

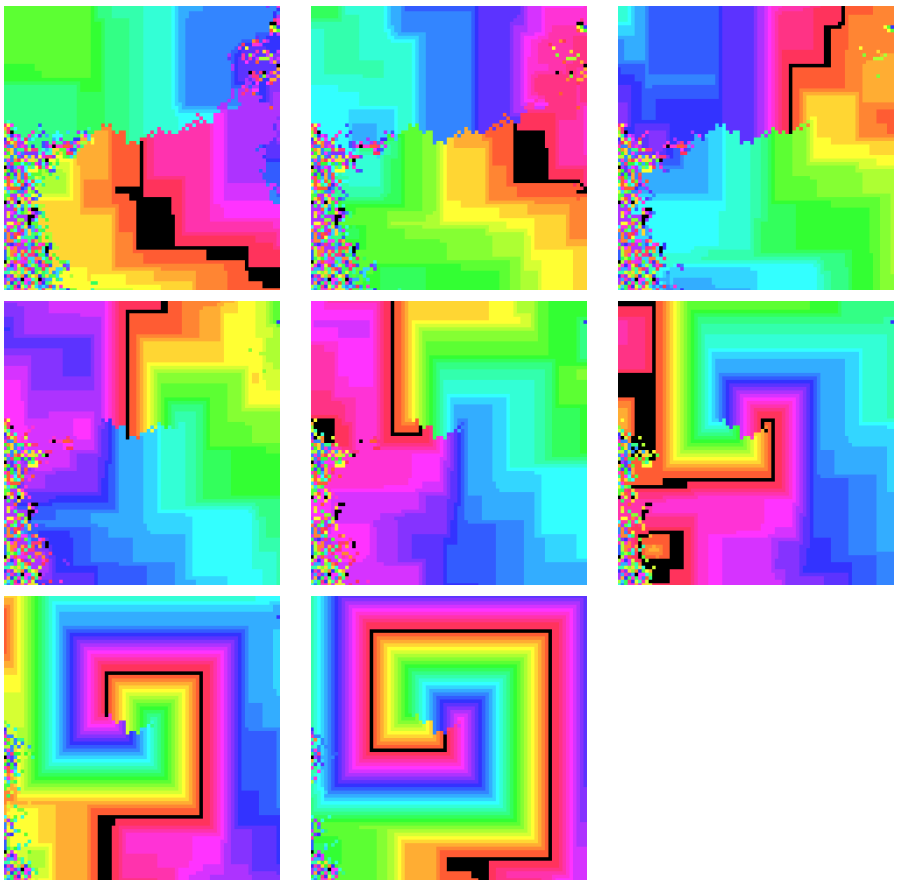

Figure 5.4 – Two droplets merging and creating a spiral.

separates these two droplets where no two cells on either side of the line are bonded. The waves of values in these droplets are going in opposite directions over this border (in figure 5.4 the top droplet's waves are happening right-to-left, while the bottom droplet is moving left-to-right). If they are both moving in the same direction along the border they are unlikely to make a spiral, and will more commonly simply merge to make a bigger droplet.

2. Eventually this border will break at some point where both droplets' waves meet with the same value. Because the droplets are moving in opposite directions, this will happen very soon after they first collide, and will very likely happen at two points, one where the first droplet joins the end of the second, and one where the second joins the end of the first. (In figure 5.4 this happens on the right edge of the border in the second image, and on the left edge in the third and fourth image).

3. Once this has happened, the breaks eat away at the border from both sides towards each other, until they meet in the middle, as shown in images 3, 4 and 5 in figure 5.4. Once they meet like this, they cannot destroy the border any more, and end up leaving this scar in the middle of the droplet, where there are cells that neighbour each other but yet aren't directly bonded.

4. Because waves cannot travel through this scar in the middle of the droplet, they have to go around it. Because the waves in the previous droplet were heading in opposite directions, they swirl around this scar. As this happens, the scar gets eaten away and eventually becomes as small as it can possibly be (n cells), while the spiralling increases speed, until it fits our formal spiral definition. This behaviour can be seen in image 5, 6, 7 and 8 in figure 5.4.

5.4 Demon Domination

Once a spiral is formed, they will either expand and take over the entire grid or until they collide with another spiral, either option ends with every cell being in the same droplet, with the entire grid repeating with period n. This can be seen in figure 5.5.

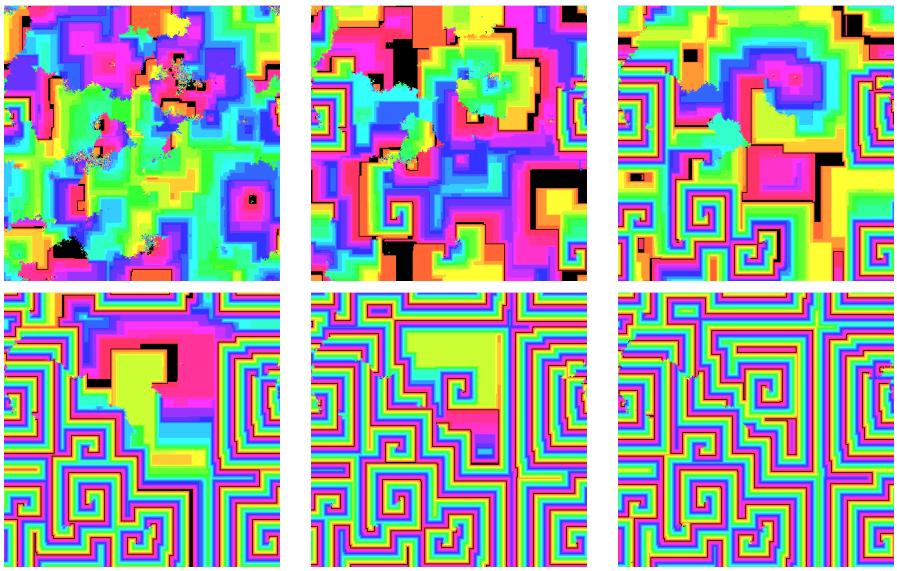

Figure 5.5 – Spirals expanding and filling the grid. The last image shows the final configuration, with it repeating with period n. 400 x 400, cycle length of 24.

5.5 Moving Neighbourhoods

While everything stated about cyclic automata so far is true for all neighbourhoods, we have so far been specifically examining cyclic automata in the Moore neighbourhood (the eight immediate neighbours in the four cardinal directions and the 4 diagonals), but other neighbourhoods create interesting views also. By deliberately restricting the neighbourhoods to fewer neighbours, we can create different patterns, and while the behaviours are all the same, they can end up looking drastically different, as you can see in figure 5.6 Although each of these run under the same set of rules and for the same amount of time, they end up in different stages, and even those stages themselves look different.

In figure 5.6 you can see that these all look quite different. (a), the Moore neighbourhood, is the one most often seen in cyclic automata, and because of the fact it has quite a large number of neighbours it quickly approached the demon domination stage, with spirals taking over and covering the grid and interacting with one another. (b), the Von Neumann neighbourhood has half the number of neighbours, and so is only at the droplet stage with remnants of debris still remaining static, dividing the droplets.

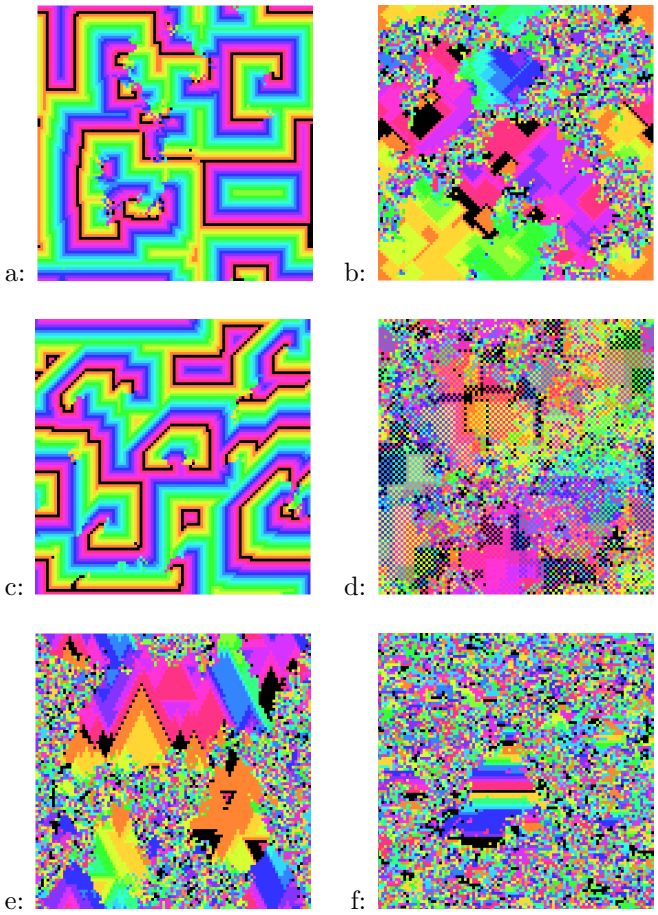

Figure 5.6 – Four different cyclic automata patterns of cycle length 15 after 150 generations, each on a 100x100 grid with torodial wrap around, the only difference between the automata is the neighbourhoods in which they operated. Reading left to right, top to bottom, the neighbourhoods used are (a): Moore, (b): Von Neumann, (c): 0123456 (all but bottom right), (d): 0257 (only the diagonals), (e): 0126 (north, north west, north east, and south), and (f): 01234 (all directions from Moore apart from south, south east and south west). These numberings indicate which cells in the Moore neighbourhood are active, using the cell indices from figure 1.1.

(c), using the neighbourhood 0123456, is very similar to the Moore neighbourhood, except for the fact it is missing the bottom right neighbour (cell 7), which causes its spirals to grow in an asymmetrical pattern, with it growing slower in the north west direction rather than the others.

(d), my personal favourite out of these 6, is over the neighbourhood 0257, which is only the diagonals. This is topologically similar to two Von Neumann neighbourhood cyclic automata running at the same time on top of each other. This is because every cell acts like a bishop in chess: those on a 'white square' (ie $x+y$ is odd) cannot ever interact with one on a 'black square' (ie $x+y$ is even). Because of this, there's effectively two different simulations running at the same time, which are completely unaware of each other, and cannot ever interact. So places where debris still exists for 'black squares' may be going through the final demon domination stage for 'white square'. Now if we rotate the entire gird by 45 degrees and examine only the 'white squares', we can see that this is indistinguishable from a Von Neumann-neighbourhood cyclic automata, and similarly for 'black squares'.

(e), which is the neighbourhood 0126, which is North West, North, North East, and South. This neighbourhood means that values spread out in the form of an isosceles triangle, with droplets forming triangularly. As there are more neighbours above the cell, their successor values are more likely to appear above than below, and so the waves move upwards. As the waves are moving upwards, the cells at the top of the droplets do not get to contribute their values to the rest of the droplet as easily as the bottom of the droplet, and so it experiences asymmetrical growth, with droplets expanding below faster than above, as you can see in figure 5.7.

(f) uses the neighbourhood 01234, which is all neighbours in the Moore neighbourhood apart from the bottom row (ie. it uses the W, E, NW, N and NE cells). This means that waves of values can only travel downwards (as each cell's successor will have to be from above), and so create 'veins' of movement flowing towards the bottom of the grid. This also means that, on an infinite grid, a spiral can never occur, as there's no way a series of neighbouring cells can reach back to its start, creating a cycle, as they can only go left, right and up, but never back down again. This means that the furthest it can go in its progression through the stages is the droplet stage, with droplets expanding

left and right but predominantly downwards. However, on a torus based grid[2], spirals can exist, by going off one side of the grid and appearing at the other side in order to create a cycle.

[2]See 1.2.1 for details about torus based cellular automata.

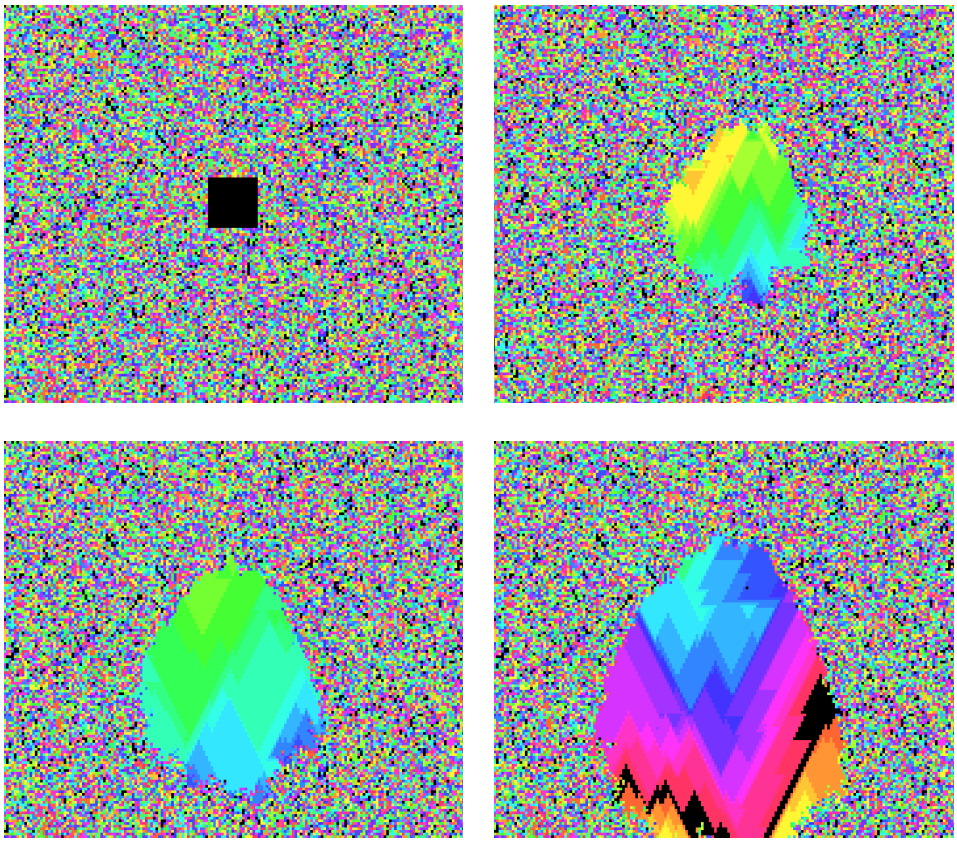

Figure 5.7 – The growth of cyclic automata with cycle length 25 over neighbourhood 0126. First image shows an artificial droplet (the black square) created admist an otherwise stable configuration. The next three images shows this setup after 700, 1000 and 1500 generations. You can see the droplet expanding faster downwards than upwards, and that the droplet as a whole is taking the rough shape of a triangle.

Chapter 6

Wireworld

Wireworld is a simplistic CA that is used to simulate electrons moving inside circuits. It has four different cell values, those being:

Blank / Empty

These cells are, as the name suggest, empty. They start empty and always remain empty for the entirety of the simulation, and never change values.

Wire

These cells are the cells in which the electrons flow. They are the pathways and routes for information to flow through the circuit.

Electron Head

Electron heads are the moving parts of the simulation, they flow through the wires and interact with one another, causing the logic and program flow to emerge.

Electron Tail

Electron tails immediately follow electron heads. They are there to give an electron signal a direction, stop an electron flowing backwards, and helps with letting the electrons interact with one another. An electron head followed by an electron tail will from now on be referred to as just an 'electron'.

Wireworld runs using the Moore neighbourhood, using the following rules: [15]

Blank	→	Blank	
Wire	→	Electron Head	if exactly one or two neighbours are electron heads
		Wire	Otherwise
Electron Head	→	Electron Tail	
Electron Tail	→	Wire	

These rules attempt to create a system which parallels electrical circuits and the way electrons flow through them. The system works remarkably well, as the system itself is Turing complete, and so can simulate any computer system given enough time and space.

6.1 Logic Gates

6.1.1 Clocks

To start off with, we can see that Wireworld is capable of basic logic gates. Our most basic tool is the clock (figure 6.1), a basic circuit that sends out an electron every n steps. It does this by having a loop of wire with electrons, which travels in circles around it until it reaches an exit, where it splits into two electrons, one continuing around the loop and the other travelling down the exit wire.

This allows our system to run regulated and at a beat, which allows us to create a system that is more reliable and predictable, and if we set our cycle to be an appropriate length we can stop complex, slower components interacting with past computations which would lead to unstable and unpredictable operations.

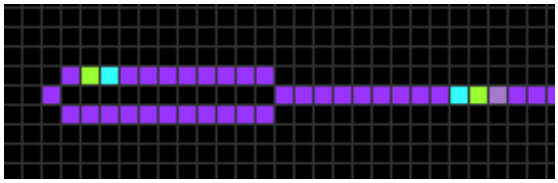

Figure 6.1 – A Wireworld clock; where the electron circles the loop anticlockwise until it reaches the exit point on the right, where two electrons emerge, one continuing the loop and the other heading right down the wire. This clock runs with a period of 24 generations. Here blank cells are black, wires are purple, electron heads are light green and electron tails are light blue.

6.1.2 OR Gates

One of the most basic logic gates is the OR gate, which takes two input wires and outputs an electron if either (or both) of them are sending in electrons. The OR gate consists of one main component, which is a 5 cell cross, where inputs are taken in from opposite sides. An input from either side travels through the cross and out perpendicularly to its input. The key to this component is that the electrons do not travel directly across the cross, because once the electron reaches the center of the cross the three central wire cells become electron heads, with the previous cell being an electron tail. Because there are three electron heads (rather than two or one) adjacent to the opposite input cell, it does not become an electron head, and the signal cannot travel that direction, and so cannot flow up the other input wire.

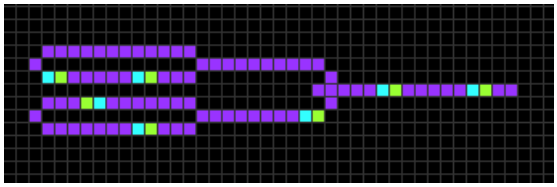

Figure 6.2 – A Wireworld OR gate, taking inputs from two clock running on the left side, and outputting on the wire to the right.

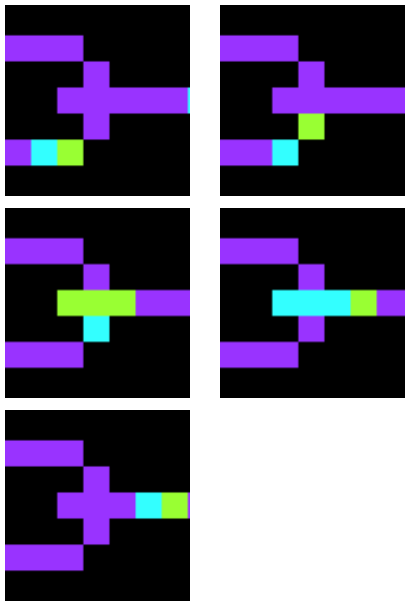

Figure 6.3 – The OR gate from figure 6.2 as it accepts an input from the lower input wire, and outputs to the right-side wire. It can be seen how it goes through its process of blocking access to the input wires by spreading out over 3 cells, causing the top cell in the cross to not become an electron head.

6.1.3 XOR Gates

The next gate is the exclusive-OR (XOR) gate, which takes in 2 inputs and outputs an electron if and only if exactly one of them is an electron. This works in a similar way to the the OR gate, stopping electrons flowing 'back' into the input wires by having three wire cells in a row that stop the electron travelling. The main part of the XOR gate is a hollow 3x4 rectangle of wire cells which take inputs from opposite sides. With only one input, the electron comes from one side, it splits into two, both going opposite directions around the rectangle. The electron that moves towards the output wire splits when it reaches it, one going down the output and the other continuing around the rectangle. There are now two electrons in the rectangle, moving in opposite directions towards each other. They hit, meaning that the an electron doesn't escape into the other input wire, again having three electron heads in a row.

If there are two electrons entering the XOR gate simultaneously, then they

both act in the same way, but they meet each other at the output wire and all three cells at the output side of the rectangle become electron heads, meaning that the output doesn't emit an electron.

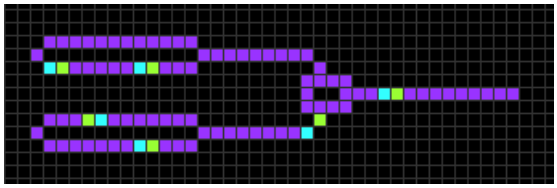

Figure 6.4 – A Wireworld XOR gate, taking inputs from two clocks running on the left side, and outputting on the wire to the right.

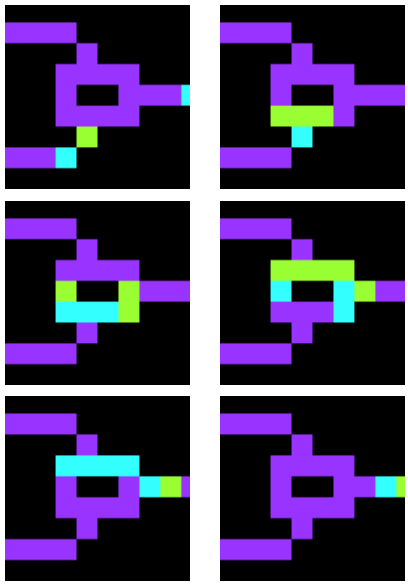

Figure 6.5 – The XOR gate from figure 6.4 as it accepts an input from the lower input wire, and outputs to the right-side wire.

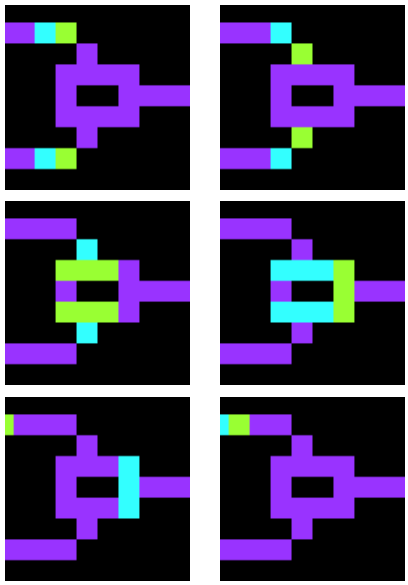

Figure 6.6 – The XOR gate from figure 6.4 as it accepts an input from both input wires at once, and therefore outputting nothing to the right-side wire.

6.1.4 AND Gates

Our third logic gate is the AND gate. This gate is more complicated than the previous gates, yet uses the same sort of system as before. It again uses the three-cell blocker technique, but is set up in a way such that when only one input contains an electron, this electron interferes with itself and cancels itself out, and the signal can only go through when electrons come from both sides simultaneously.

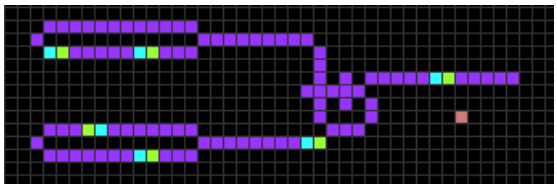

Figure 6.7 – A Wireworld AND gate taking inputs from two clocks and outputting to the wire on the right.

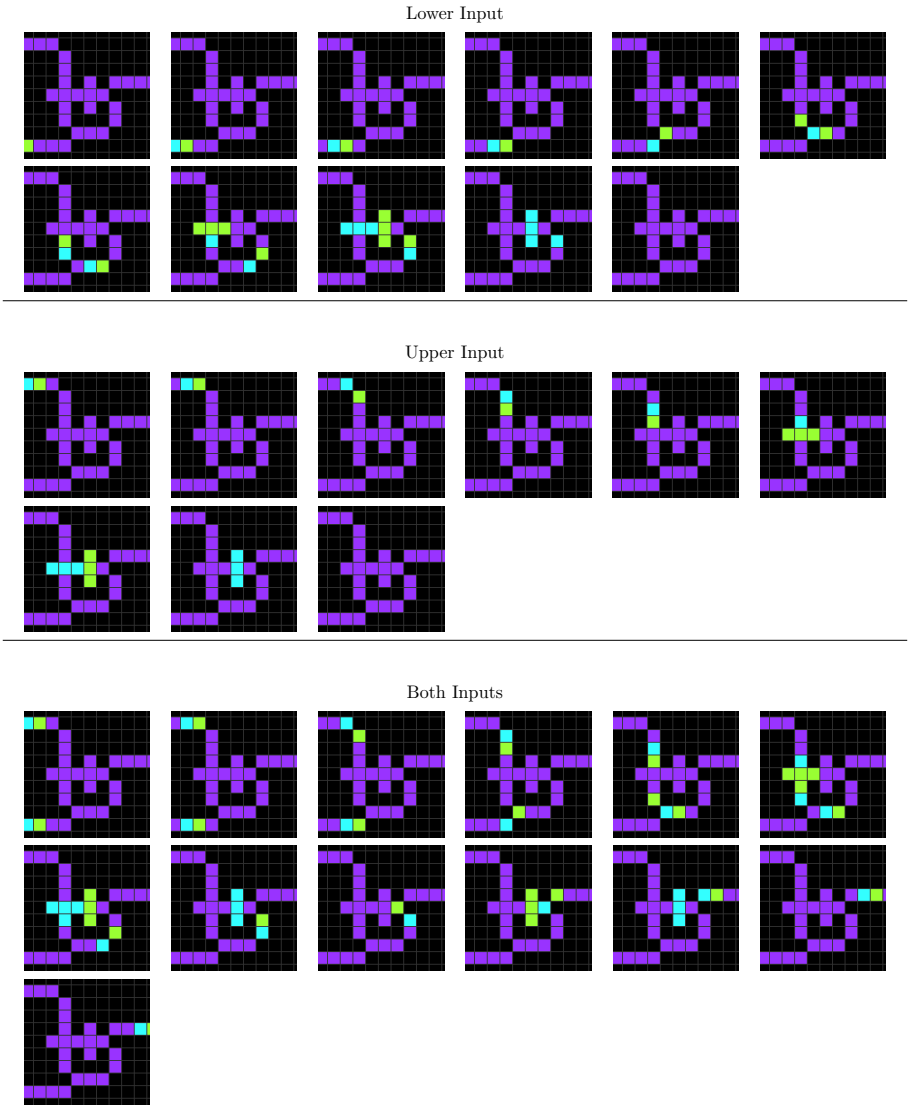

Figure 6.8 – The AND gate from figure 6.7. The top sequence shows what happens when it takes an electron from the lower wire; the middle sequence shows the upper wire input; and the bottom sequence shows what happens when both inputs happen simultaneously. As you can see, an electron is only released to the right output wire in the case where both inputs are both electrons, and does not release one otherwise.

6.1.5 NOT Gates

A NOT gate has a problem that we haven't encountered so far - given a non-electron input, it has to emit an electron. Previously the gates were just configurations of wire cells, and the underlying workings of them were powered by the electrons that they were processing.

However, with the NOT gate we wish for it to output an electron when no electron enters, and to not emit an electron when one is inputted. This raises some problems: given no input we can't constantly output electrons because they wouldn't travel as they need wire cells infront of them in order to move, and so the minimum we can do is output an electron every 3 generations (one to emit electron head, one for electron tail, and then we require a wire cell, and then we can repeat). Once we've settled on the frequency of the NOT gate, this gives us the problem of moving parts (ie. electrons which are part of the gate itself), which leads to specific windows of time when the input can enter the gate in order to get a correct output.

Our NOT gate works like a clock merged with an XOR gate - if the input is an electron and in time with the clock, then it outputs nothing, but if there's no electron inputted, we get an output from just the clock. This relies on timing - the input electrons need to be synchronised with the clock - for example if our clock is set to repeat every 6 generations (ie. it outputs an electron head, an electron tail, then four wire cells, and then repeats again, like the clock part in figure 6.9), then our inputs need to fit to this beat and enter the gate at the correct times; if they do not then our NOT gate will not work correctly.

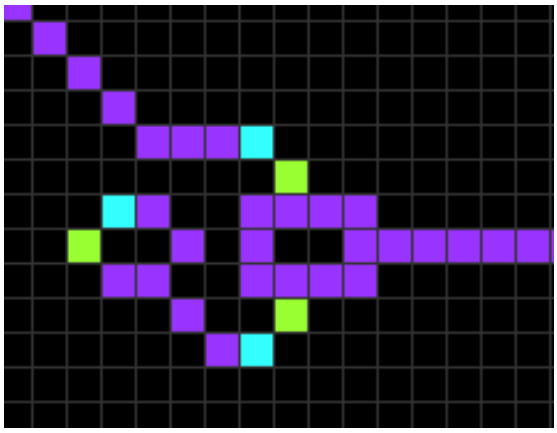

Figure 6.9 – A Wireworld NOT gate, taking inputs from the wire coming in from the top left. As you can see, it is made out of a small 6-generation clock (the loop on the left) connected to an XOR gate (the rectangle on the right). The electrons going into the XOR gate from above and below are coming in at the same time, and so the NOT gate will correctly output nothing.

6.2 Turing Completeness

As described in previous sections, we know some cellular automata are Turing complete, which means they are equivalent to a Turing machine in terms of computability, and theoretically can compute anything computable [16]. In this section I present my proof that Wireworld is Turing complete by showing that it can simulate elementary CA rule 110, which has been proven to be Turing complete by Mathew Cook [11]. If it can emulate all the behaviour of rule 110, then it has all the computing power of a Turing machine [17].

Rule 110's table is shown again in figure 6.10. If we interpret the left, center and right neighbour values as boolean values a, b and c respectively, we can rewrite the rule as the logical expression $XOR[OR[b, c], AND[a, b, c]]$. In the last few pages its been shown that Wireworld is capable of creating the necessary gates to evalulate this expression, but we still need to show that it can evaluate it in multiple 'cells' (cells in rule 110 that we are simulating), and can communicate adjacent 'cells'. In figure 6.11 we can see the computational circuit that takes in three inputs and calculates the associated rule 110 output.

Once we have the computational aspect of the 'cell', we now need to make the cell communicate with adjacent cells, specifically we need to send its own

111	110	101	100	011	010	001	000
0	1	1	0	1	1	1	0

Figure 6.10 – The rule table for elementary rule 110.

output back to itself to its b input, to the above cell as its c input and to the below cell's a input.

Figure 6.12 outlines our final component. It is designed to tessellate, and so any number of them can be placed vertically adjacent to one another, and the system works exactly as rule 110 works; as the component is 30 Wireworld cells tall and 120 cells wide (without lightbulbs), any rule 110 instance with n cells can be emulated by a $30n$ by 120 Wireworld circuit made of n of these circuits ontop of each other. This means that anything computable by rule 110 is also computable by Wireworld, and as rule 110 is Turing complete, then Wireworld is also Turing complete, and therefore can compute anything computable.

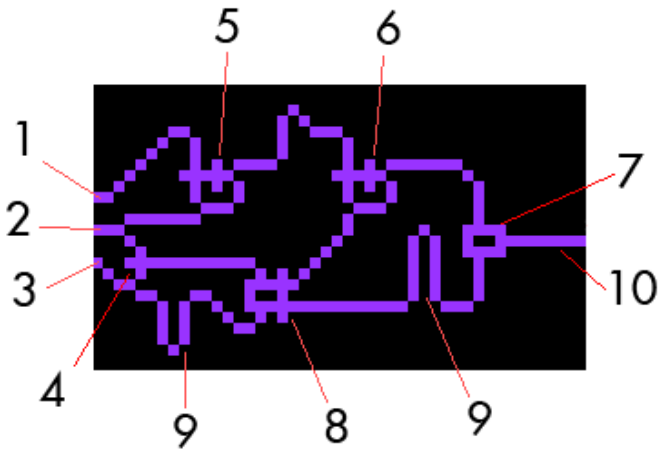

Figure 6.11 – The computational component of our 'cell', taking in 3 inputs (labelled 1, 2, and 3 for a, b and c respectively) and computing the rule 110 value to the output wire (labelled 10). (4) is an OR gate, taking values from b and c and outputting OR[b, c], while (5) computes AND[a, b]. (8) is a component yet not mentioned so far, which allows signals to cross over one another, so the resulting signal from OR[b, c] at (4) travels to the XOR gate at (7) via the bottom wire, while input c travels to (6), which computes AND[AND[a, b], c] = AND[a, b, c]. The result of this AND gate then travels to (7), which computes our final output XOR [OR[b, c], AND[a, b, c]]. The two winding paths labelled (9) are there to slow down the signals travelling to the next component in order to make sure the electrons reach them at the correct time and in sync with the other electrons needed for that gate.

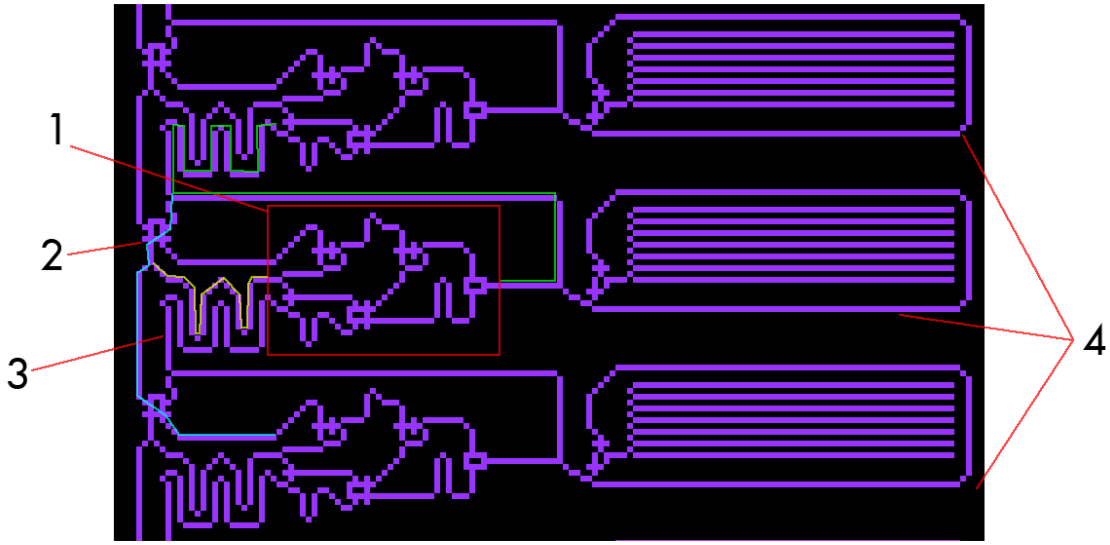

Figure 6.12 – Here are three identical completed components, with indications of how the output from the central component (labelled 1) travels to itself and the adjacent components (green, light blue and yellow lines). Again, we need the crossover segments (2) in order to allow the signal to transverse over another wire. As you can see, the path from the central 'cell's' output to the input of the lower 'cell' would be considerably longer than the path to the upper cell or to the central cell, which is why the snaking wire configuration (labelled 3) is set out for inputs b and c for each cell, but not for input a, this is to slow down the signals in order to let the a signal catch up and for all three inputs to enter the main body of the cell synchronously.

The large rectangular circuits on the right of each cell (labelled 4) are not necessary for computation or communication, but instead used to make the output easier for humans to read. These circuits (which we will call them 'lightbulbs') are triggered when the 'cell' emits an electron (ie. when the 'cell' is 'on'), which triggers a switch (the two crosses in the bottom left of each lightbulb) which continually shoot out electrons into the seven rows of wire, making the lightbulb light up. This continues for a set time, and turns off when the original electron travels around the outside and turns off the switch. This means whenever a 'cell' is on, this lightbulb glows for a short time and then turns off again, the length of time being slightly shorter than it takes for a rule 110 'generation' to occur. This gives us a visual output of which 'cells' are on each rule 110 'generation'.

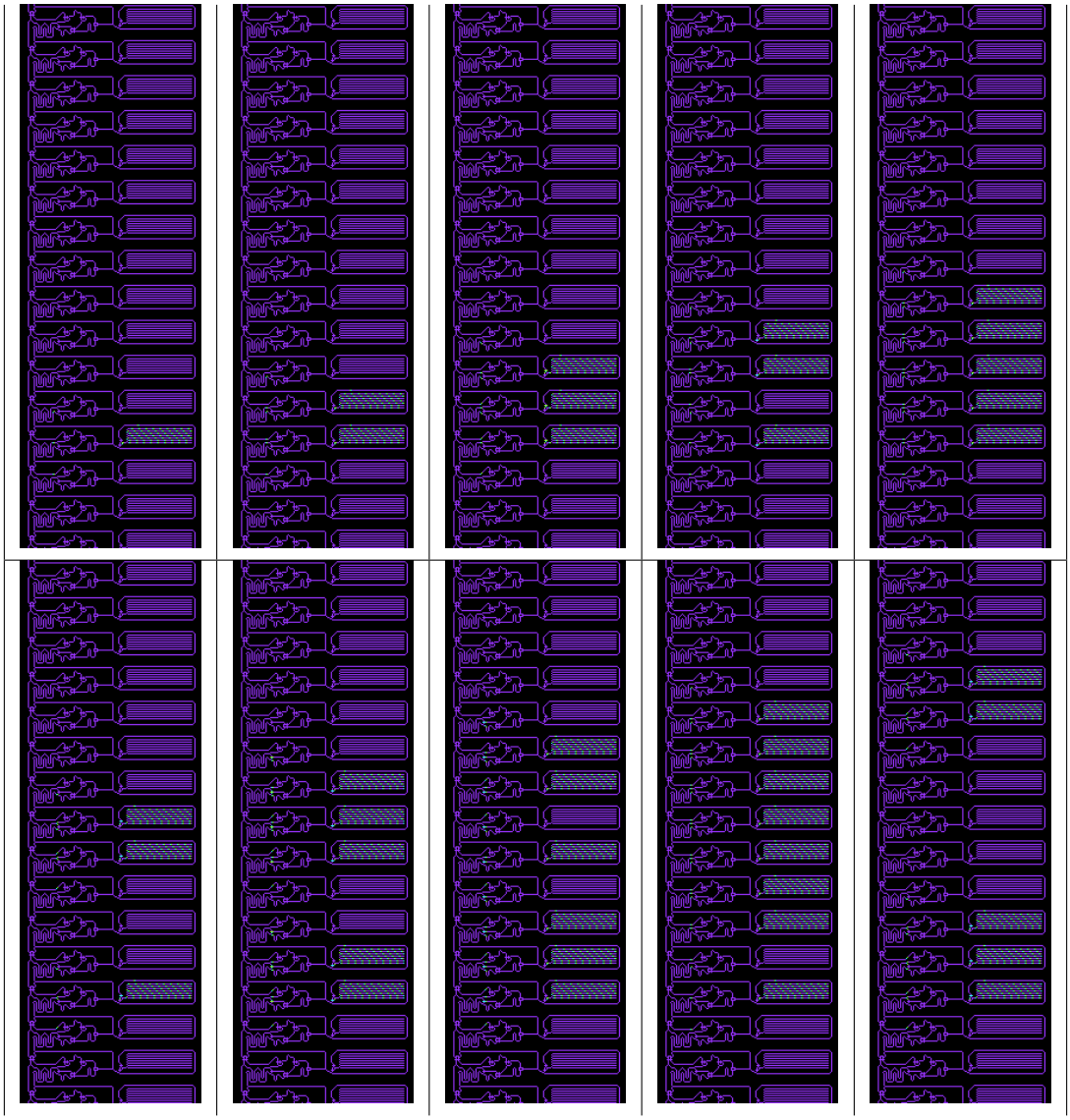

Figure 6.13 – This table shows the first 10 "supergenerations" of our rule 110 Wireworld simulation, starting from one alive 'cell'. At each supergeneration we can see which 'cells' are on or off by looking at each respective cell's lightbulb component. As you can see by comparing this to figure 6.14, the sequence of lit-up lightbulbs is the same as the first 10 generations of rule 110 from one initial on cell.

Figure 6.14 – The first 10 generations of the Rule 110 elementary cellular automata starting from one initial 'on' cell.

I can appreciate that it might be hard to visualise what is happening on paper, and so a video of this circuit and others is available online at:
`http://tringham.co.uk/zb/ca/`

Chapter 7

Turing Machine Cellular Automata

While many cellular automata have the property of being Turing complete [1][11][6][6], I designed this specific set of rules in order to create a Turing machine in its traditional sense, including a tape, a set of states, and a transition function, so it is easy to see what is actually happening and how the computations are computed. This, in a very direct way, shows that cellular automata are very powerful computational systems, and can be designed to fulfil tasks and problems. The emphasis for this was ease to program for, not simplicity of the rules. Because of this, the set of rules for this is quite complex.

7.1 Definition of Turing Machine

The classic definition of a Turing machine is a 7-Tuple [18] $M = \langle Q, \Gamma, b, \Sigma, \delta, q_0, F \rangle$, with:

Q being a finite non-empty set of states.

Γ being a finite non-empty tape alphabet.

$b \in \Gamma$ being the blank tape symbol.

$\Sigma \subseteq \Gamma \setminus \{b\}$ is the input alphabet.

$\delta : Q \setminus F \times \Gamma \to Q \times \Gamma \times \{Left, Right\}$ is the transition function, which takes in the input and the current state and outputs what to write to the tape, the next state and which direction to move the tape head.

$q_0 \in Q$ is the initial state.

$F \subseteq Q$ is the set of halting states.

While this definition is perfectly good, in the construction of our Turing machine we will take some small liberties with it, without any loss to the computability of the final product. Our final construction will also not be as general, and we will make some of these variables concrete and unchanging. The Turing machines made by this automata will have to satisfy these criteria:

$$\Gamma = \{0, 1, blank\}$$

$$b = blank$$

$$\Sigma = \{0, 1\}$$

$$\forall q \in Q \setminus F, \gamma \in \Gamma : \delta(q, \gamma) = \delta_s(q, \gamma) \times \delta_t(q, \gamma)$$

$$\delta_t : Q \setminus F \times \Gamma \to \Gamma \times \{Left, Right\}$$

$$\delta_s : Q \setminus F \times \Gamma \to Q$$

These specifics have been chosen to make the Turing machine automata easier to design and implement, while still allowing all the computational power a classical Turing machine has. The decision to split the transition function into two separate subfunctions does not technically matter in theory, as δ still fits the original definition, but in practice we will be interpreting them as two separate functions rather than one, which will be explained shortly.

7.2 Design

The type of Turing machine that this ruleset emulates is a relatively standard one. The tape alphabet consists of 0, 1 and BLANK, there is only one tape and only one tape read/write head, and this head can only move one item left/right at a time. Due to the restrictions of having only finite memory by running it on a computer, when running this ruleset in the CAVP, the tape has to be finite

size (although it can be arbitrarily large), but the set of rules governing it does not put a theoretical limit on the size of the tape.

7.2.1 Types of Cell

The design for this set of rules is based loosely on Wireworld, where 'electrons' travel through 'wires' in one direction. However, the difference here is that these electrons can come in multiple different types, and other cells can perform actions on them, albeit very basic ones.

There are 19 different types of cell, each of which fall into 9 different categories. For the rest of this chapter, to avoid confusion, when talking about specific cell values they will be in block capitals, while their categories will not; for instance, a TRANS0 cell is a transmitter.

Blank BLANK cells are empty cells, they do not perform any special action and remain blank (with a couple of very specific situations, explained later)

Wire WIRE cells are the highways for the signals, and allow the signals to travel through them.

Tape Cells There are 2 types of cell which were designed to act as elements on the tape, TAPE0 and TAPE1 for a 0 on the tape and a 1 on the tape respectively. The blank elements of the tape do not have their own type of cell, they are made up of BLANK cells.

Transmitters Transmitters are the equivalent of electron heads in Wireworld; they are the signals that travel around the wire sending data around the circuit. Each transmitter has a tail which comes directly behind it in order to give it direction, and to attach more data to the signal. There are three types of transmitter: TRANS0, TRANS1 and TRANSB, which represent a signal of 0, 1 and blank respectively.

Tails As said before, each transmitter is followed by a tail. There are three types of tail: TAILL, TAILR and TAIL, which represent a left signal, a right signal and no direction respectively.

Heads and Preheads In one cell directly below the tape is a HEAD cell, which is where the head is currently pointing on the tape. When it is next to a transmitter and tail pair it writes to the tape the value that

the transmitter carries (either a 0, 1 or a blank), and then "moves" in the direction that the tail indicates (either left, right or no movement). When it "moves", it turns from a head to a blank, and the adjacent cell in the direction indicated turns into a PREHEAD0, PREHEAD1 or a PREHEADB depending on whether the value in the tape is a 0, 1 or blank respectively. A transmitter is then shot out which contains the value that was read from the tape, and the prehead turns into a HEAD. The reason why we need a prehead cell is so that the tape head does not keep releasing copies of the tape elements, we only want to do this once, and the prehead cells work as this intermediary.

Floodgates and Dams There are situations where we want to control whether a signal travels down a wire or not, and want to have a 'switch' which opens or closes wires. To do this there are three types of cell, FLOODOPEN, FLOODCLOSED and DAM. These cells work together to allow only one transmitter/tail pair to travel through a wire, and then blocks that wire, stopping any transmitters to travel through until it is opened again.

Inverters INVERTER cells, when next to a transmitter cell change the type of transmitter. For instance, if a TRANS0 was next to a INVERTER, it would become a TRANS1, a TRANS1 would become a TRANSB, and a TRANSB would become a TRANS0. These cells have a very specific use, and work in conjunction with floodgates in order to perform different operations depending on what types of transmitter enters the state-circuit.

Crossovers In this way of creating the Turing machine, each state is a circuit made out of cells, and to create the transition function between states wires need to connect states together. However because of the amount of connections required to do this, wires need to cross over one another, and in order to do this there is a CROSSOVER cell, which sits in the center of a 4-way cross junction, and allows signals to travel in a straight line over one another without interfering with each other.

7.2.2 The Key Components

The general design of of the cellular automata Turing machine has several key components, the *tape*, the *states*, the *tape transition function* (δ_t) and the *state transition function* (δ_s).

The Tape The *tape* consists of a horizontal line of cells, which are either TAPE0, TAPE1 or BLANK cells. Underneath the tape is a HEAD cell, which reads any transmitter and tail cells that appear underneath it, and writes the value of the transmitter to the tape, and moves in the direction that is specified by the tail. Once it moves, it emits another transmitter cell which has the value of the tape under the new head position, and this transmitter cell travels to the current state for processing.

The States The *states* are small circuits which take in the current value of the tape, and apply the transition function. As only one state can be the current state, all but one state is blocked off using floodgates and dams, allowing no inputs to enter. The current state is the only one which is not blocked off, and allows the transmitter from the tape to enter, and it applies the transition function. As soon as a signal enters the current state, the current state is blocked off, so no more signals can enter it until it is the current state again.

The tape transition function The transition function of the Turing machine is split into two subfunctions, both taking in the input read from the tape. The first, the *tape transition function*, δ_t, deals with what to write to the tape and what direction to move the tape head. Each state contains 3 loops that contain a transmitter and a tail, the transmitter dictates what value to write to the tape, and the tail dictates what direction the head moves. There are three loops because there are three different tape symbols; the transmitter/tail signal from the first loop is sent to the tape if a TRANS1 enters the state circuit, the signal from the second loop is released if a TRANS0 enters the circuit, and the third if a TRANSB enters. The signal that is released makes its way back to the tape, and is written to the tape, and moves the head in the direction specified by the tail.

The state transition function The second subfunction, the *state transition function*, δ_s, dictates which state becomes the next current state. The process here is largely similar, there are three loops containing signals, and the signals from one of the loops is released, depending on what type of transmitter enters the circuit. However here each signal that is released is the same, a TRANS1 with a nondirectional tail, but the path that they each take differs. The path from each loop, instead of going back to the tape, goes to a different state, which

unblocks the entrance to it, and so when the next signal from the tape occurs it enters this new state.

7.3 Implementation

Figure 7.1 shows a two-state Turing machine that counts in binary onto the tape. The labels indicate the following:

1 The tape, made out of BLANK, TAPE0 and TAPE1 cells, which are black, grey, and white respectively. The non-blank part of the tape currently reads 0010, with the head of the tape (the light blue cell) on the rightmost element of the tape.

2 is a signal: a transmitter (green cell on the right) followed by a tail (the yellow cell on the left), heading towards the right. In this case the transmitter is a TRANS0 and the tail is a TAILR, and so when it reaches the tape head, it will write a TAPE0 to the tape and move the head right.

3 is the signal slowdown wire, which allows the signals to travel from the tape to the states and vice versa. The reason why it zig zags up and down is to slow down the signal and make sure the signal doesn't arrive at the states from the tape before the transition function is fully complete.

4 and 5 are the two states. There are 6 'signal loops' in each state, each of which are made out of 4 cells: 2 wire cells, a transmitter and a tail. These cells continually rotate until they are released as part of the transition function, where they copy themselves and continue down their wires. The three signal loops on the left are for the tape transition function, and so head back to the tape, while the right three signal loops each have their own path to different states as part of the state transition function. In the bottom left of 5 you can see that there is a red cell (a DAM cell), this cell stops any signals entering the state. However, this cell does not appear in 4, as this is the current state.

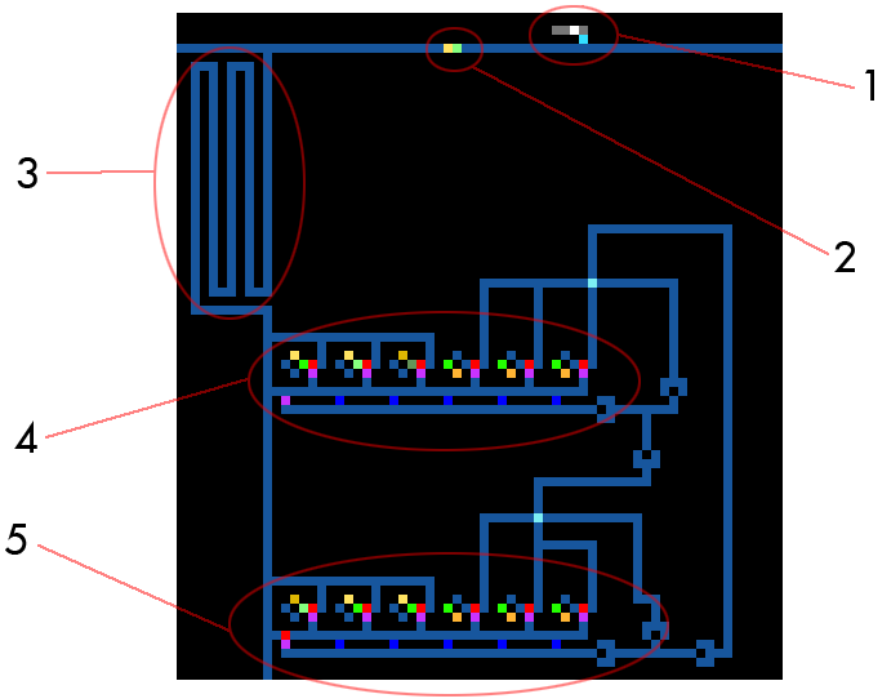

Figure 7.1 – A two-state Turing machine that counts in binary on the tape.

7.3.1 Floodgates and Dams

Floodgates and dams are one of the most useful sets of cells in the the Turing machine automata, as they allow control flow and the ability to redirect signals, as well as limiting the quantity of signals that come through a wire. Their purpose is to do one thing: allow only one transmitter to come through a wire, and then block any other transmitters from passing through this wire until it gets told to unblock it (in other words, to make sure signals only come through one at a time and only when wanted).

The way they work is relatively simple: once a FLOODCLOSED cell is placed under a WIRE cell, one transmitter can pass through this wire, but only one. Once the transmitter passes through the wire, the WIRE cell directly above the FLOODCLOSED does not turn back into a WIRE, it turns into a DAM, which transmitter cells cannot travel through. The DAM cell remains there, blocking any more transmitters that try to pass it.

These DAMs can only be gotten rid of by resetting the floodgates. To do this, a TRANS1 cell must be in the cell directly below the FLOODCLOSED cell. If this happens, the FLOODCLOSED becomes a FLOODOPEN cell, which converts the adjacent DAM cell back into a WIRE cell, so transmitters can flow through it again. The FLOODOPEN then immediately turns into a FLOOD-CLOSED, and this system begins again. Behaviour of floodgates and dams can be seen in figure 7.2.

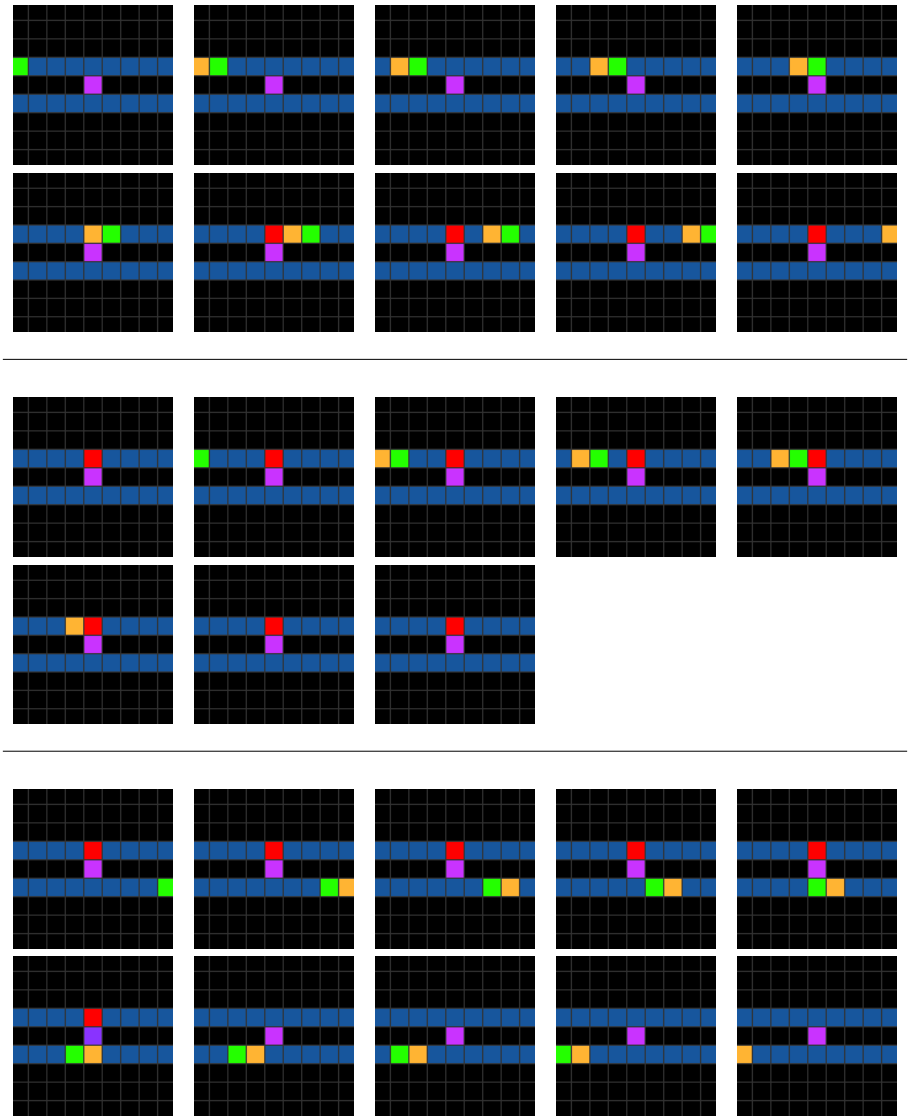

Figure 7.2 – Floodgates working. The top wire is the wire that is going to be blocked, while the wire underneath is the reset wire, while the purple cell inbetween is the floodgate. In the top sequence you can see a signal travelling across, and as it passes over the floodgate it leaves behind a DAM (red), which remains immobile, while the signal itself travels on unaffected. In the second sequence you can see another signal attempt to travel through the wire. As there is a DAM cell now in the way, it cannot pass through the wire. In the bottom you can see another transmitter coming through the bottom reset wire, which instead of causing a DAM, turns the FLOODCLOSED cell into a FLOODOPEN cell (the 6th image), and which removes all the DAM cells (7th image), and turns back into a FLOODCLOSED cell again. This resets the floodgate, and another signal can travel through the top wire once again. Resetting only works when a TRANS1 occurs under the FLOODCLOSED; a TRANS0 or TRANSB under the FLOODCLOSED cell will change nothing. The reason for this is explained in the Inverters section.

7.3.2 Inverters

Inverters have a special job of turning one type of transmitter into another. This allows us to create a system which identifies which type of transmitter has entered a state, and releases only the correct signal loops.

The inverter cell is placed directly underneath a wire cell. When a transmitter travels through this wire, it turns into another transmitter, and continues moving in the same direction. If we write this as a function, $Inv : Transmitter \to Transmitter$, it works as follows:

$$Inv(TRANS0) = TRANS1$$

$$Inv(TRANS1) = TRANSB$$

$$Inv(TRANSB) = TRANS0$$

We can see that this is a permutation, and we can apply Inv multiple times to change the transmitter to whatever we would like. For instance, we can identify what transmitter enters by how many times we need to apply Inv until it becomes a TRANS1:

$$TRANS1 = TRANS1$$

$$Inv(TRANS0) = TRANS1$$

$$Inv(Inv(TRANSB)) = TRANS1$$

$$Inv(Inv(Inv(TRANS1))) = TRANS1$$

This is the reason why floodgates only open when a TRANS1 appears below them rather than any other type of transmitter. If we have a series of floodgates with an inverter after each one (illustrated in figure 7.3), each floodgate will only open if a certain type of transmitter enters the wire. For example, if a TRANS1 entered, it would open the first, fourth, ... $(3n+1)$th floodgates. This is because once it reaches the $(3n+1)$th floodgate it would have passed through $3n$ inverters, and $Inv^{3n}(TRANS1) = TRANS1$. Similarly, a TRANS0 would open the second, fifth and $(3n+2)$th floodgates, because $Inv^{3n+1}(TRANS0) = TRANS1$; and a TRANSB would open the third, sixth and $(3n)$th floodgates because $Inv^{3n+2}(TRANSB) = TRANS1$.

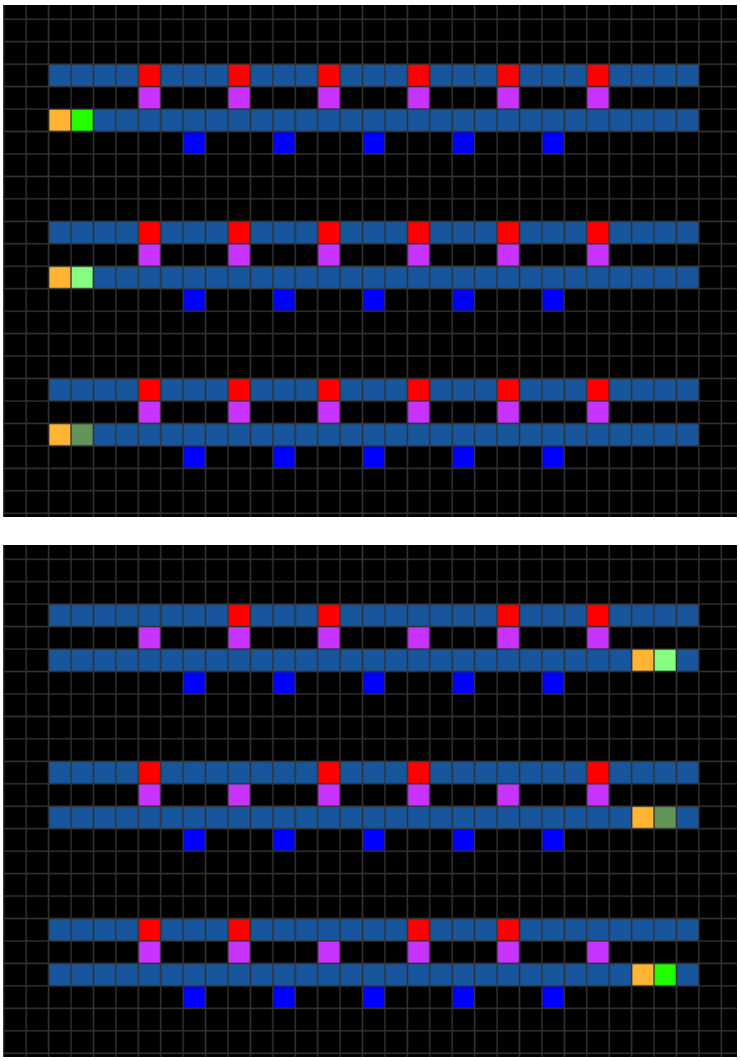

Figure 7.3 – A series of floodgates (purple) with an inverter (the dark blue cells under the wire) following each floodgate. The top image shows 3 identical circuits, where all the floodgates are closed, causing the DAMs (red cells). In the top circuit, a TRANS1 is about to flow through the circuit, in the middle a TRANS0, and the bottom a TRANSB. The lower image shows which gates have been opened by each transmitter once it has passed through the wire: TRANS1 has opened the first and fourth, TRANS0 has opened the second and fifth, and TRANSB has opened the third and sixth.

This basic design is the way the signal loops work, and the way the states only release the correct signal loop depending on the input. The floodgates for the signal loop are only opened if their respective transmitter has entered the state.

7.3.3 Demonstration

To demonstrate the Turing machine automata, let's examine the example depicted in figure 7.1, which counts in binary on the tape. The Turing machine's transition function is as follows:

State	Read from Tape	Next State	Write to tape	Direction
0	0	0	0	Right
0	1	0	1	Right
0	B	1	B	Left
1	0	0	1	Right
1	1	1	0	Left
1	B	0	1	Right

The process behind it is quite simple: state 0 simply takes the head back the far right of the non-blank part of the tape, and does not ever alter the values on the tape, it just keeps heading right until it finds a blank, at which point the head moves left (and so point the least significant bit) and switches to state 1. In state 1, the adding takes place: the head inspects the first bit, if it's a zero change it to a one and we're done, go back to state 0. However, if it's a one it turns it to a zero and needs to carry it over to the next bit, so head left and continue to add; this process repeats until the head finds a zero (or a blank) on the tape, at which point it flips it to a one, and goes to state 0 again to repeat the process.

This Turing machine is quite simple and fits our criteria perfectly, and so is a good example to demonstrate the Turing cellular automata.

Comparing this diagram to figure 7.1, the state transition function is quite clear to see; in each state, the three signal loops on the right are the signals used to open up the next state. Out of the three, the leftmost is release when a TRANS1 enters, the middle when a TRANS0 enters, and the right when a

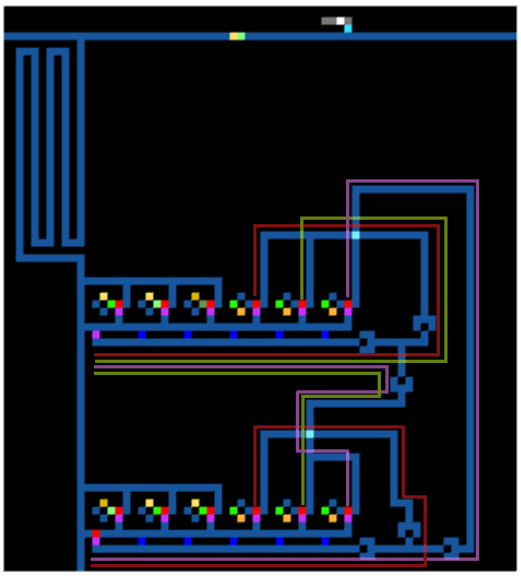

Figure 7.4 – The counting turing machine from previous diagrams. Here, the path the signals from the state transition function take are indicated, and you can see they correspond to the transition table.

TRANSB enters. Figure 7.4 shows the paths that these signals take. As you can see, the paths correspond to the transition function table, for example, in state 0, a TRANSB releases a signal from the rightmost loop, which follows the path indicated down to the reset wire of state 1, this corresponds to the third line of our transition table.

The tape transition function loops do not differ by path but by what type of signal is released, and so for each state and input, the signals inside the three tape transition function signal loops (the three loops on the left of each state) are also defined by our table. For example, the leftmost loop in state 1 is released when a TRANS1 enters the state, and so the signal inside must be a TRANS0 followed by a TAILL, as the table defines the next move to be writing a 0 to the tape, and moving the head to the left.

A video of this Turing machine is available at `http://tringham.co.uk/zb/ca`.

Chapter 8

Other CA Rules

The CAVP contains several other cellular automata rules, but this dissertation focuses on the four previously discussed. What follows is a brief discussion of some other rules included in the CAVP but not the main study of this project.

8.1 Maze-Solving Cellular Automata

This automata was created by adapting a general algorithm designed by Shahram and Meybodi [19]. There are four types of cell:

Start - This cell is the beginning of the path.

Target - This cell is where the path ends.

Free - These cells are blank cells where the path is free to travel.

Blocks - These cells are immovable obstacles which the path may not travel through.

Mark - These mark the wave that is travelling though the maze.

Path - These are the cells that are part of the final path.

Cells that are of type mark, path and target all have directions attached to them, either up, down, left, right or neutral. This automata runs under the von Neumann neighbourhood, and so each direction points to a neighbour.

The system works by creating a wave of mark cells that spread out and fill the free cells, exploring the maze, each mark noting the direction that it came from. The first mark cell to neighbour the target cell becomes a path cell pointing in the same direction as the mark cell it replaced. The cell that the path cell is pointing at becomes a path cell also, and this trail of path cells leads back to start cell, producing the shortest path between the start and target.

For the full set of rules and proof of correctness please see *A Maze Routing Algorithm Based on Two Dimensional Cellular Automata*, Shahram and Meybodi, 2006 [19]. A video of the CAVP demonstrating this CA can be seen at `http://tringham.co.uk/zb/ca`

8.2 Smoothlife

Smoothlife slightly bends the definition of cellular automata: rather than being discrete, it is continuous. It was designed by Stephan Rafler in order to generalise Conway's Game Of Life onto a continuous domain [20]. Conway's Game Of Life (GOL) has two possible states, alive or dead, with dead cells become alive if they are in the "birth" interval (which is just $\{3\}$) and remaining dead otherwise, and alive cells remaining alive if they are "survival" interval (which is $\{2, 3\}$), and becoming dead otherwise. GOL runs under the Moore neighbourhood, and so there are only 8 neighbouring cells. GOL, like other cellular automata, also runs under discrete time steps.

Smoothlife generalises this to a continuous domain by changing the way of thinking about GOL.

- Firstly, a Smoothlife cell can be thought of as a disc at a point on a plane. The value of the cell is not just on or off, it is the filling of the area inside this disc.
- Similiarly, the neighbourhood is a ring around this disc, and again is based on the filling of this ring.
- To make it even more smooth, instead of having solid boundaries where the cell becomes dead or alive, the filling values of the inner disc and outer ring are put through sigmoid functions.
- The death and survival intervals are now defined in $[0, 1] \times [0, 1]$ rather than sets of integers.

- Time steps are no longer discrete, cells are eased towards a value rather than instantly becoming that value in the next generation.

To implement this into the CAVP, the grid here is not an integer grid, but a `double` grid, allowing floating point numbers. Each cell on the grid then works out it's "value" by finding the integral of the values of the cells in its disc (using anti aliasing to make sure the edges of this disc aren't jagged), and divided by the total area, giving us the proportion of the disc that is filled. Similarly, the integral of the outer ring is computed. These outer value is put through the sigmoid functions, which "push" these values towards 0 or 1, depending on whether the inner disc is alive or dead and whether the outer values are between the respective death / survival intervals. The cells do not instantly become this computed value; in order to have continuous time steps, given our continuous time step $\delta_t \in (0, 1]$, current value c and newly computed value γ, each cell becomes $\delta_t\gamma + (1 - \delta_t)c$.

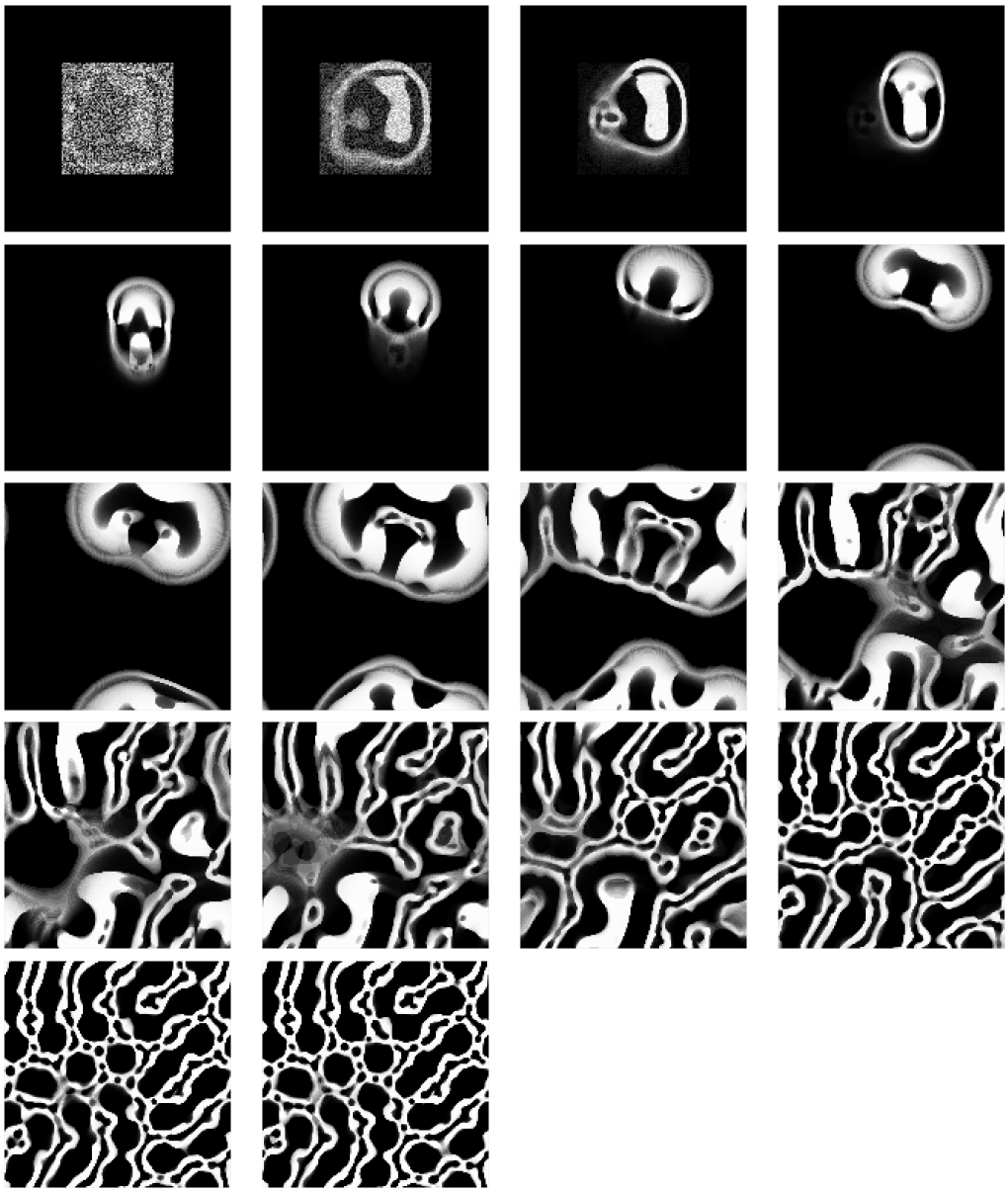

Figure 8.1 – An instance of Smoothlife on a toroidal grid rendered in the CAVP, starting from a grid which is blank, apart from the central square which is random. Here the birth interval was $[0.257, 0.336]$ and the death inteval was $[0.365, 0.549]$, with an outer radius of 20 and an inner radius of 6.66, with $\delta_t = 0.1$. These images were taken over 200 generations. As you can see, the random cells quickly form into a smooth shape that almost looks biological. This shape in images 2-7 moves a short distance upwards, and then "bursts", expanding rapidly and filling the grid, and eventually slowing to what appears to be close to a hexagonal tiling. This is only one specific setup of Smoothlife, the behaviours that can happen can vary wildly depending on the initial parameters.

Part III

The Cellular Automata Visualisation Program

Chapter 9

Aims

When designing the CAVP, there were a few aspects that I felt were of high priority for the success of the program. They were:

Visualisation

The program needed to very easily show the current generation of the CA that is being run, with a large area that shows a grid of the current cells in the CA at that generation.

Editability

An extremely valuable feature for investigating CA is the ability for creating conditions and specific set ups in order to see how they alter behaviours of different CA rule sets. This meant that the program needed the ways of letting the user change the specific values of the cells, as well as changing specific aspects of the rules that are being run.

A variety of rules

In order for the program to have a large reach and have flexible use, a decent sized repertoire of rules should be programmed into the CAVP in order for the user to view many different simulations and compare them to others.

Easy to use

The CAVP should be intuitively designed, so that users of the CAVP should be able to figure out how to use it quickly without being confused as to how it is used.

The final program, I feel, succeeds in all of these areas, due to careful designing and programming.

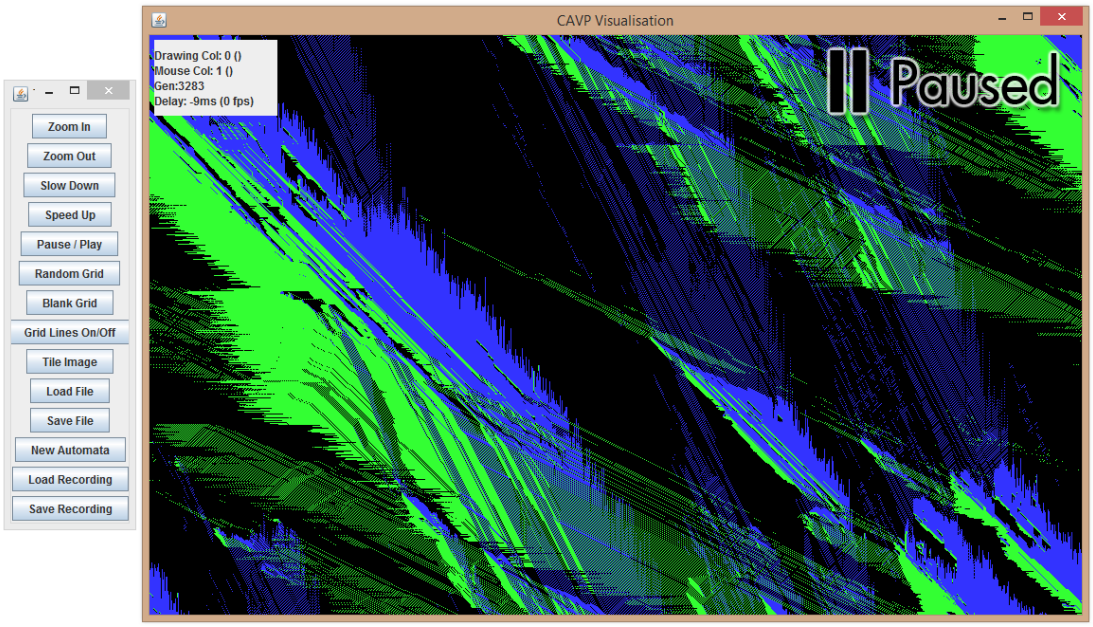

Figure 9.1 – A screenshot of the CAVP running the Biham-Middleton-Levine traffic model.

Chapter 10

Design and Implementation

The CAVP is built in Java, with modularity in mind at all times throughout the development. The program has 5 main parts:

Skeleton The Skeleton.Java is the backbone of the program, linking all these other parts together and making sure they communicate correctly. It is the class that runs first and creates instances of the others, and governs the flow of data between them.

Surface Surface.Java is the the graphical side of the program, which renders the CA visualisation and handles the mouse inputs. Left clicking draws cells of certain values onto the grid, while right clicking-and-dragging pans the visualisation around.

The Toolbox The toolbox is a separate window which allows the user to select options and settings that alter the display and behaviour of the program. These inputs are communicated to the skeleton, which are then sent to the appropriate modules for execution.

The Abstract CA Class CellularAutomata.Java is a Java abstract class which outlines the structure cellular automata rule classes, and has abstract functions that need to be defined by each CA rule.

The Rule Classes Each individual cellular automata rule has its own Java class, which is an implementation of CellularAutomata.Java class, and contains the definitions of the abstract functions which outline the behaviour of the CA.

10.1 Overview of Program Flow

The Skeleton is the first part of the program that loads. It firstly reads the initial inputs from the user, which specify the dimensions of the grid, and the specific rule that this being run first, and any parameters that the rule requires. Once these have been inputted, these values are used to set up the automata, creating the 2D array that stores the values of the grid, and creating the instance of the CA object. The Surface and Toolbox objects are then instantiated, creating the CAVP main window and the toolbox. The skeleton starts a timer, which causes the cellular automata object to update at regular intevals.

At each update, the 2D array of values is parsed to the CA rule object, which goes through each cell in the grid, inspects its neighbours and calculates what value it should have next, as governed by the rules of the CA. This new array of updated values is parsed back to the Skeleton, which sends it to the Surface object. The Surface object takes this 2D array and turns it into a bitmap, which is rendered on screen. The program then waits until the next update should occur, and begins again.

10.2 Skeleton

The skeleton of the program is where the main bulk of the program is. It handles several key jobs:

- Reading the inputs directly from the console
- Creating instances of the other elements of the program
- Handling the timing of the updates
- Communicating data and inputs/outputs between the other modules
- Controlling all other aspects of the program

10.3 Surface

The Surface class handles the graphics and visual output of the program, including:

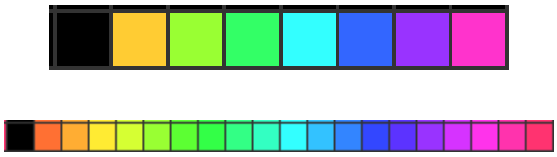

Figure 10.1 – Top: cells in ascending value order when there 8 possible cell values, bottom: when there are 20 values.

- Creating a function which linearly maps all unique cell values to pixel colours
- Creating a bitmap of the current grid
- Rendering this bitmap on screen at the right specified size and position
- Taking in mouse inputs that occur in the window and reacting to them

In order for the bitmap to show visually each unique value in the grid distinctly, a mapping must occur that takes the value of the cell and takes it to a unique colour, which is used for that cell's pixel colour. In order to do this, I used the Hue-Saturation-Brightness (HSB) colour space, mapping each value to a different colour, uniformly spread out throughout hue values with constant saturation and brightness values, with the exception of the value 0 which is always mapped to black. This way of spreading the colours out makes sense both programmatically and visually - the values ascend like they do in a rainbow, with adjacent values having "adjacent" colours.

As you can see in fig 10.1, each value is uniquely identifiable, and are easy to interpret. From this one can see that the light-blue and dark-blue cells are going to have similar values, while the pink and green are going to have distant values. It also means that in most cases these values are easily distinguished and identified, and so when interpreting the behaviours of the CA in the visualiser it is easy to see what types of cells are doing what.

This is not the first method that I used, and had to go through a few different methods before settling on this one. The first, and seemingly most obvious at the time, was to have the values range from black to white as the values increase. This method definitely allows the user to see instantly which cells are of a higher value compared to others, but however it became increasingly difficult to distinguish between similar shades of gray.

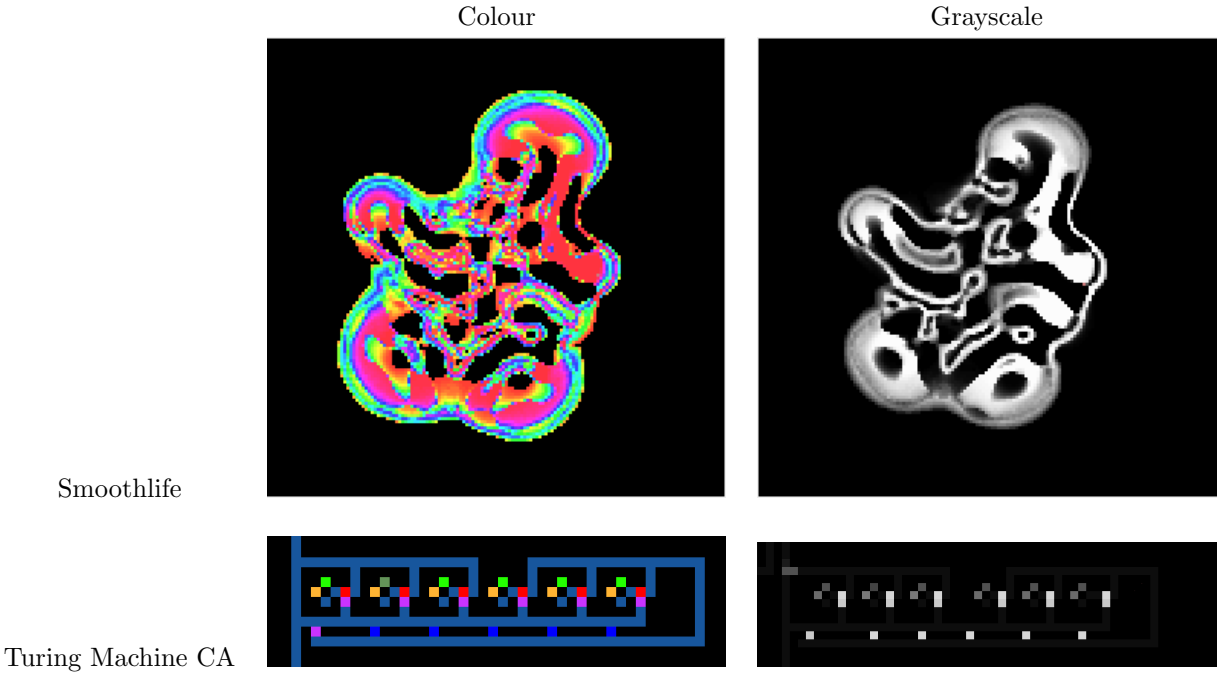

Figure 10.2 – A comparison of different colour schemes working on different automata.

What I learnt was that different colour schemes work for different systems. As you can see in figure 10.2, Smoothlife is very ugly and much harder to interpret using the HSB colourspace compared to grayscale, while the Turing machine cellular automata is much harder to see when in black and white. This led to a necessity for a system where each automata could define its own colour scheme, and if one was not specified it would default to the HSB model.

10.4 The Toolbox

The toolbox is a very simple part of the program, it is a small horizontal window that houses a list of buttons. These buttons affect the cellular automata being run and allow the user to control the system. There are 14 buttons listed, which include buttons to alter the CAVP's delay time (ie. how quickly the simulation runs, and pausing / playing), saving / loading CA configurations and recordings, changing the entire grid (to either a blank grid or to a randomly

generated grid), adding gridlines to the visualiser, and turning tiling on or off. When these buttons are pressed, they trigger their respective functions inside the Skeleton.

10.5 The Abstract Rule Class

The abstract rule class, CellularAutomata.Java, is a class which all rule classes in the CAVP need to implement. It contains a collection of abstract functions which need to be implemented in order for the rule class to be integrated easily with the rest of the program, and it also has several utility functions to allow the writing of rules to be quick and easy.

All ruleclasses need to implement these functions:

`int rules(int xpos, int ypos)`
: This function is the heart of the cellular automata - it takes in the current cell position (`xpos, ypos`), and returns what value the cell should become.

`int getNeighbour(int x, int y, int ind)`
: This is function indexes all the neighbours - given a cell at (`xpos, ypos`) this function returns the value of the (`ind`)th neighbour.

`int getMaxColours()`
: This function returns only a constant, which is the number of different cells that are needed for this automata. This function is used to work out how many different colours are needed to display the automata, and is used to pick them wisely.

The non-abstract functions in this class are available for all rules classes to use, without them having to write them out every time. Some of these utility functions are there to integrate with the rest of the program, and some are there to stop having to rewrite often-used functions in every rule class. Having these utility functions forces all rules to be written to the same standards. The utility functions are:

`int getCell(int x, int y)`
: This function returns the value of the cell in the `(x, y)` position.

`void setCell(int x, int y, int val)`
: This function sets the value of the cell in the `(x, y)` position to `val`.

`int[][] update()`
: This applies the rule function to every single cell, and therefore updates the automata by one generation.

`int getMooreNeighbour(int x, int y, int ind)` and
`int getVNNeighbour(int x, int y, ind ind)`
: These two functions return the value of the `(ind)`th neighbour in the Moore neighbourhood and Von Neumann neighbourhood respectively.

`int torusify (int x, int max)`
: This allows the creation of torus-based grids easily - if `x > max` or `x < 0` it returns the position of x if the grid were a torus, rather than having nothing over the edge.

`int[][] getGrid()` and `void setGrid(int[][] grid)`
: These functions return the entire grid, and set the entire grid respectively. This is useful when passing the grid to the graphics code, and when loading configurations.

There are a couple of other functions that can be overloaded by each rule class, but do not necessarily have to. For example, there is a function `String[] getCellNames`, which returns an array of strings, each of which is the name for the cell who's value is the strings index. For instance, in Wireworld, this function would return `{"Blank", "Electron", "Tail", "Conductor"}`, as the blank cells have value 0, the electron head has value 1, the tail has value 2, and the conductor has value 3. This method is not necessary for the working of the system, but the visualiser displays information about what cell the mouse is currently hovering over, including the cell's value and name, making it easier to see the current grid.

10.6 The Rule Classes

Each rule in the CAVP needs its own class. At the time of writing, there are 13 different rule classes. As mentioned before, all classes implement `CellularAutomata.Java`, and the one that is currently running is instantiated inside the Skeleton, with the `update` and `rules` functions running on a timer. These together compute the next generation's grid, which is passed to the Skeleton and then to the Surface for rendering.

Chapter 11

Development

The program was created using Java, using mainly the core Java functions and libraries, deliberately keeping third party libraries away from the project in order to be in complete control of development and strict quality control could be implemented. The program and documents were kept on a GitHub repository which was continually updated every time a change was made. This allowed me to work on several different machines in different places without having to worry about multiple copies that aren't in sync with each other. The GitHub repository also allowed backups of all files in case anything went drastically wrong, or if anything was lost. Luckily, a situation like that did not arise, but the GitHub repository would have kept a copy regardless.

The program was developed steadily over the course of the project, starting initial development in October 2013, with a final product finishing up around February 2014, with minor alterations and finalisations happening throughout the course of the theoretical research side of the project.

Because the CAVP was developed in Java, the program was highly cross-platform compatible, which allowed me to program on both my Windows laptop and the Warwick DCS computer systems, which are Linux based.

Chapter 12

Using the Program

12.1 Epilepsy Warning

While the majority of the visualisations made by the program are tame, a small proportion of the displays in this program can be quite intense, including flashing images and swirling, very "psychedelic" images. I am by no means fully educated in what does and doesn't cause seizures, but it is advised that people who have photosensitive epilepsy or history of seizures use the CAVP with caution, or possibly not use it at all. While I nor anyone involved in this project has epilepsy to my knowledge, it was a consideration that needed to be taken into account when working in public areas, such as the Warwick Computer Science department, and when displaying the program to others, such as in the presentation part of this project. This was taken seriously, and all public presentations limited any displays of flashing images, and all distributions of the program come with a readme file which contains a seizure warning.

Other than this, there are no other legal, social, ethical or professional issues related to this project.

12.2 Setup And Use

The program is very easy to download and install. Firstly, check that you have Java installed, and then download the source code from `https://github.com/johntringham/Cellular-Automata`. To compile, navigate to the source code's

folder (ie. `/Code/`), and then use the command `Javac Skeleton.Java`. Once compiled, to run the program, type `Java Skeleton`.

The first thing the program does is ask a couple of questions in the command line. You will need to first specify the width and height of the grid, and then you will be asked what type of cellular automata you would like to run. Each type of CA in the program has a keyword, and to run it you need to type this keyword. They are as follows:

Rule	**Keyword**
Conway's Game Of Life	`GOL`
Wireworld	`WW`
Cyclic Cellular Automata	`CYCLIC`
Biham-Middleton-Levine Traffic Model	`BML`
Shortest-Path Finding Automata	`MAZE`
One Dimensional Elementary CA	`1D`
Two Dimensional Elementary CA - Moore Neighbourhood	`2DM`
One Dimensional Elementary CA - Von Neumann Neighbourhood	`2DV`
Turing Machine Automata	`TURING`
Langton's Ant	`ANT`
Smoothlife	`SMOOTHLIFE`
View a pre-recorded CA	`PLAYBACK`

Once you enter a type keyword, the program might require more information specific to that rule type. For example, if you select Cyclic, then it will ask you to specify what cycle length to use, and which neighbours it should restrict, if any. If you pick BML, it will ask you for what percentage congestion to start with. A full list of parameters and what they accept is listed in appendix B.

Once the parameters are completed, the program asks if the simulation should be recorded. If yes, the user has the opportunity to save the recording and play it back later.

When these options are all specified, the graphical side of the program loads and appears on screen. Once this has happened, it is relatively straightforward in how to use the program. The toolbox buttons all have self-explanatory names, and so it is easy to see what each one does. In the top left hand corner of the

visualisation window there is a small grey box, which displays some information. It says the current drawing value, the value of the cell that the mouse is over, the current generation, the delay between generations, and the number of generations being processed per second.

To pan around the visualisation, hold down right click and drag the mouse. To draw the current drawing value onto the grid, either left click on the visualiser, or click-and-drag in order to colour multiple cells. To change the current drawing value, either use the mouse scroll wheel or the up and down arrow keys on the keyboard.

A video demonstration of how to use the program is available at `http://tringham.co.uk/zb/ca`.

Part IV

Conclusion

Chapter 13

Conclusion

From the making of this project, we have investigated properties of cellular automata, focusing on cyclic cellular automata, Wireworld, and a proprietary Turing machine cellular automata ruleset. We have investigated evolution of cyclic cellular automata and how it progresses through its different stages, and inspecting what causes the behaviours that arise from it, in both the traditional Moore and Von Neumann neighbourhoods, as well looking into how using less traditional neighbourhoods affects the overall behaviour. This project demonstrates how it is possible to build logical operators in Wireworld, as well as constructing larger, more intricate circuits, focusing on a circuit that models the Rule 110 elementary cellular automata, proving that Wireworld is Turing complete. The Turing machine cellular automata was designed, and a demonstration machine was described and displayed, as well as details on how to build other Turing machines.

This project also encompassed the design and creation of the Cellular Automata Visualisation Program, which allows the user to run a large variety of cellular automata rules. The program itself is efficient and easy to use, and made the inspection of the cellular automata included in this project easy and detailed, as well as allowing the production of images and videos of the cellular automata possible.

13.1 Improvements and Extensions

The CAVP currently has 11 different cellular automata built into the program, one obvious improvement would be to add more to the program. This would not be very difficult: because of the structure of the program, writing rules is quite easy, and would not take a long time to do.

However, other programs that perform similar purposes, such as Golly [21], have built file structures to define rules and specific grid arrangements. At the moment, the CAVP uses its own way of storing grids, and defines rules as individual Java class files. An extension that would seem worthwhile is to add a rule interpreter that was able to use these files, to allow use of the already sizeable library of rules and patterns that exist for these other programs. Allowing the program to have access to these would let the CAVP make use of these resources that are already available elsewhere.

The CAVP currently only allows one and two dimensional cellular automata that are based on a square grid, an interesting extension to the program would be to extend these to three dimensional grids, and have different shaped lattices. This would allow the user to examine more varieties of cellular automata, and increase the possible uses for the program.

On the theoretical side of this project, I would have liked to go into more detail about other cellular automata rules that were not mentioned in this document, but limited time and word-count restrictions prevented this.

13.2 Comparison to the Project Specification

The three main aims listed in the Project Specification were as follows:

What CA are and how to define one

How they can evolve into complex systems

Their use as computational devices

The final project, I feel, succeeds in all of these areas; the first is satisfied by Part One of this document, while the second and third aims are the driving force for the investigations into Cyclic CA, Wireworld, and the Turing Machine automata, and I feel this document answers these aims satisfactorily.

The Project Specification also listed several cellular automata which I wished to be included in the final project:

- Conways Game Of Life, as well as Life-Like automata
- Wireworld
- Rock Paper Scissors
- Langtons Ant
- Cyclic rules
- Elementary CA, especially rules 30 and rules 110, as well as others
- Some discussion of higher dimension (3 dimensions and possibly above) CA.
- Some proprietary CA rule sets to demonstrate their ability to compute, including but not necessarily limited to:
  - A programmable Turing machine
  - The shortest path problem

As you can see, this list is very similar to the final repertoire of CA rules included in the project, with a few changes. Rock Paper Scissors is a rule set which is very similar to cyclic CA, and so was not included as a separate entity in the final program. Game of Life and life-like automata were implemented into the program, but were not discussed in length in the project as they enjoy enough attention elsewhere, and I chose not to focus on them. Langton's Ant was also implemented into the program but not discussed at length in this document as I had chosen to focus on other rules, despite the fact that Langton's Ant and related rules possess very interesting properties and behaviours [22, 23], and I would have liked to examine them more closely. As well as these omissions, several rules that weren't on this list made the final cut into the program, such as Smoothlife and two-dimensional elementary cellular automata.

Despite these changes, the project still remains within the bounds of the Project Specification, as I had foreseen that I would change which rules to include and focus on, as it was easy to see that once research had begun it would

become clearer what rules were the most worthy of investigating. Because of this, the fact that the list wasn't definitive is not a big issue, and was predicted.

Overall, I feel the project was highly successful, and

13.3 Acknowledgements

I would like to thank Alex Wendland, a friend who helped code the Smoothlife rule class, when at first it seemed too complex to code efficiently; Nathanial Sear, who spent hours helping me proofread this document; and my supervisor, Andrzej Murawski.

I would also like to thank everyone who helped me with this project, and especially all of my friends and family who pretended to be interested while I kept talking about it.

Appendices

Appendix A

Derivation of formula (5.3)

What we wish to find: a formula that given an alphabet of size n and a string of size C, what is the probability that this string contains every letter in the alphabet?

We know that obviously there are a total of n^C different C length strings over the alphabet of size n. Then we know the probability is the number of strings that contain all letters divided by n^C. Let's investigate how many strings that are missing at least one letter, the compliment of the set we are looking for.

The set of all strings that are missing one chosen letter is $(n-1)^C$, and there are $\binom{n}{1} = n$ different ways of choosing this letter that is missing. However, just multiplying these two together $(n-1)^C \cdot n$ gives us a higher number than we want, as it blindly counts strings that are missing two or more letters multiple times, for example with the alphabet $\{a, b, c, d\}$, the string $aaaa$ would be counted three times (once when you have chosen b to be missing, once when c is missing, and once when d is missing). To get around this, we will continue on and look at when two or more letters are missing, when three or more letters are missing, and so on, and use the inclusion-exclusion principle to get around these dud results.

So when two letters are missing, there are $(n-2)^C$ strings and $\binom{n}{2}$ different ways of choosing these two letters. If we continue, we can see that if we are missing k letters then there $(n-k)^C$ strings and $\binom{n}{k}$ different ways of choosing these k letters. So, using the inclusion exclusion principle, we can see that the total number of strings is that *don't* have every letter is:

$$\binom{n}{1}(n-1)^C - \binom{n}{2}(n-2)^C + \binom{n}{3}(n-3)^C - \ldots + (-1)^n \binom{n}{n-1}1^C \quad \text{(A.1)}$$

So that number of strings that *do* contain every letter is

$$n^C - \binom{n}{1}(n-1)^C + \binom{n}{2}(n-2)^C - \binom{n}{3}(n-3)^C + \ldots + (-1)^{n-1}\binom{n}{n-1}1^C \quad \text{(A.2)}$$

We can compress this formula into sum notation:

$$\sum_{k=0}^{n-1}(-1)^k \binom{n}{k}(n-k)^C \quad \text{(A.3)}$$

And then finally, to get the probability we need to divide by n^C :

$$\left(\frac{\sum_{k=0}^{n-1}(-1)^k \binom{n}{k}(n-k)^C}{n^C}\right) \quad \text{(A.4)}$$

Which is what we wanted to find. Coincidentally, this is the same as $\frac{n! \cdot S_n^C}{n^C}$, where S_n^C is the Stirling number of the second kind with parameters C and n. As Stirling numbers are often unwieldy and tricky to deal with we will stick with the first form.

Appendix B

A List of All Parameters

When starting a new automata, the program may ask you to specify more information. Here is a full list of what parameters it asks for, and what the accept and what they are used for:

Conway's Game Of Life

`Blank Proportion?` - A number between 0 and 1. The grid is randomly initialised with this being the proportion of blank cells.

Cyclic

`Cycle Length?` - A positive integer. This defines the cycle length.

`Neighbours?` - A possibly empty string made out of the characters 0-7. The numbers used in this string refer to the cell that will be included in the neighbourhood, refering to the cells in the Moore neighbourhood as indexed according to 1.1. For example, the string 0257 will limit the neighbourhood to only the cells to the upper right, upper left, lower left and lower right. Leave blank to use all cells in the Moore neighbourhood.

One Dimensional Elementary

`Rule Number?` - A positive integer between 0 and 255. This is the rule number that will be used, as defined by Wolfram's rule-naming convention [1].

`Random Start?` - `Y` or `N`. Enter `Y` for the first generation to be random. If not, then only a single cell in the center will be on, the rest shall be off.

Two Dimensional Elementary, Moore

`Rule Number?` - A positive integer between 0 and $2^{18} - 1$. This is the rule number that will be used, as defined by Wolfram's rule-naming convention [1].

Two Dimensional Elementary, Von Neumann

`Rule Number?` - A positive integer between 0 and $2^{10} - 1$. This is the rule number that will be used, as defined by Wolfram's rule-naming conventions [1].

BML Traffic Model

`Blank Proportion?` - A number between 0 and 1. As the initial generation is random, this is the probability that any one cell is blank in the first generation.

Turing Machine CA

`States?` - A non-negative integer. If zero, the grid starts blank. Otherwise, the grid is initialised with many of the key components, and this number specifies the number of states to be made, getting rid of the need to redraw states multiple times.

Smoothlife

`Colour Number?` - A positive integer. The number of colours to split the continuous values into.

`Continuous Time?` - `Y` or `N`. This specifies whether to have continuous timesteps or not.

`Time Step?` - A number between 0 and 1. If continuous time is on, then this specifies the value of δt.

`Full Box?` - This specifies whether to start with the entirety of the grid being random, or whether only a smaller box in the center and surrounded by blank cells.

Langton's Ant and the shortest-path finding CA do not require any special parameters. The Playback automata, once selected, will bring up a file selection window to specify which recording to play.

Bibliography

[1] Stephen Wolfram. *A New Kind of Science*. English. Wolfram Media, 2002. ISBN: 1579550088. URL: http://www.wolframscience.com.

[2] Guillaume Hutzler. *Cellular Automata Notes*. https://www.ibisc.univ-evry.fr/~hutzler/Cours/mSSB/CellularAutomata.pdf. Mar. 2014.

[3] *Wikibooks: Cellular Automata*. http://en.wikibooks.org/wiki/Cellular_Automata/Mathematical_Model. Mar. 2014.

[4] Sibel Gokce and Ozhan Kayacan. "A cellular automata model for ant trails". In: *Pramana* 80.5 (2013), pp. 909–915. ISSN: 0304-4289. DOI: 10.1007/s12043-013-0533-4. URL: http://dx.doi.org/10.1007/s12043-013-0533-4.

[5] Dieter A. Wolf-Gladrow. *Lattice-Gas Cellular Automata and Lattice Boltzmann Models*. 2005.

[6] Christopher G. Langton. "Self-reproduction in cellular automata". In: *Physica D: Nonlinear Phenomena* 10.12 (1984), pp. 135 –144. ISSN: 0167-2789. DOI: http://dx.doi.org/10.1016/0167-2789(84)90256-2. URL: http://www.sciencedirect.com/science/article/pii/0167278984902562.

[7] John Von Neumann. *Theory of Self-Reproducing Automata*. Ed. by Arthur W. Burks. Champaign, IL, USA: University of Illinois Press, 1966.

[8] Stephen Wolfram. "Cryptography with Cellular Automata". In: CRYPTO '85 (1986), pp. 429–432. URL: http://dl.acm.org/citation.cfm?id=646751.704557.

[9] Stephen J. Willson. "Cellular automata can generate fractals". In: *Discrete Applied Mathematics* 8.1 (1984), pp. 91 –99. ISSN: 0166-218X. DOI: http://dx.doi.org/10.1016/0166-218X(84)90082-9. URL: http://www.sciencedirect.com/science/article/pii/0166218X84900829.

[10] Franciszek Seredynski, Pascal Bouvry, and Albert Y. Zomaya. "Cellular automata computations and secret key cryptography". In: *Parallel Computing* 30.56 (2004). Parallel and nature-inspired computational paradigms and applications, pp. 753 –766. ISSN: 0167-8191. DOI: http://dx.doi.org/10.1016/j.parco.2003.12.014. URL: http://www.sciencedirect.com/science/article/pii/S0167819104000419.

[11] Matthew Cook. "Universality in Elementary Cellular Automata". In: *Complex Systems* 15.1 (2004), pp. 1–40.

[12] *Mathematica Documentation*. http://reference.wolfram.com/mathematica/tutorial/RandomNumberGeneration.html.

[13] Robert Fisch. "Cyclic Cellular Automata and Related Processes". In: *Cellular Automata: Theory and Experiment*. Ed. by Howard Gutowitz. 1990, pp. 19–26.

[14] Clifford A. Reiter. *Chaos and Graphics: Medley of Spirals from Cyclic Cellular Automata*. Elmsford, NY, USA, Feb. 2010. DOI: 10.1016/j.cag.2009.04.008. URL: http://dx.doi.org/10.1016/j.cag.2009.04.008.

[15] Samuel Gulkis et al. "Scientific American: The Cellular Automata Programs that Create Wireworld, Rugworld and other Diversions". In: 262.1 (Jan. 1990). ISSN: 0036-8733 (print), 1946-7087 (electronic). DOI: http://dx.doi.org/10.1038/scientificamerican0190-132. URL: http://www.nature.com/scientificamerican/journal/v262/n1/pdf/scientificamerican0190-132.pdf.

[16] B. Jack Copeland. "The Church-Turing Thesis". In: *The Stanford Encyclopedia of Philosophy*. Ed. by Edward N. Zalta. Fall 2008. 2008.

[17] Dexter C. Kozen. "The Arithmetic Hierachy". In: *Theory of Computation*. 2006, pp. 236 –241.

[18] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006. ISBN: 0321455363.

[19] Shahram Golzari and MohammadReza Meybodi. "A Maze Routing Algorithm Based on Two Dimensional Cellular Automata". In: *Cellular Automata*. Ed. by Samira Yacoubi, Bastien Chopard, and Stefania Bandini. Vol. 4173. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2006, pp. 564–570. ISBN: 978-3-540-40929-8. DOI: 10.1007/11861201_65. URL: http://dx.doi.org/10.1007/11861201_65.

[20] S. Rafler. "Generalization of Conway's "Game of Life" to a continuous domain - SmoothLife". In: *ArXiv e-prints* (Nov. 2011). arXiv: 1111.1567 [nlin.CG].

[21] *Golly Documentation*. http://golly.sourceforge.net/Help/formats.html, Mar. 2014.

[22] A. Gajardo, A. Moreira, and E. Goles. "Complexity of Langton's ant". In: *Discrete Applied Mathematics* 117.13 (2002), pp. 41 –50. ISSN: 0166-218X. DOI: http://dx.doi.org/10.1016/S0166-218X(00)00334-6. URL: http://www.sciencedirect.com/science/article/pii/S0166218X00003346.

[23] Ian Stewart. "The Ultimate in Anty–Particles". In: *Scientific American* 271 (2008), pp. 104 –107. URL: http://dev.whydomath.org/Reading_Room_Material/ian_stewart/AntyParticles.pdf.