Cellular Automata and Computational Universality

September 19, 2018

Thomas Archbold
University of Warwick
T.Archbold@warwick.ac.uk

Abstract

Cellular automata are discrete models with the ability to not only give rise to beautiful, intricate patterns, but also to be used as powerful tools of computation, with applications in cryptography, error-correction coding, and simulation of computer processors, to name a few. They also raise profound questions about the nature of our reality, asking whether our universe could be one such automaton. This paper provides a discussion into these automata, exploring the various power and limitations of a number of specific rule sets, covering John Conway's well-known "Game of Life" to the more obscure "Langton's ant" and "Wireworld". In particular, it explores the notion of computational universality, or Turing completeness, an automaton's ability to simulate any conceivable computation, and considers their potential in the context of solving two specific problems, the Firing Squad Synchronisation Problem and the Majority Problem. Existing solutions to these problems are explored, and their existing avenues for optimisation are discussed. In order to fully appreciate the complex structures that can arise from such simple beginnings, this project also presents software to visualise and probe further into the nature of the automata highlighted.

1 Introduction

A cellular automaton is a discrete model of computation which consists of a finite collection of "cells", each in one of a finite number of states. The state of each of these cells may evolve over the progression of time in discrete steps, and may do so according to a deterministic set of rules based on the states of neighbouring cells. Their inherently discrete nature allows for strong analogies to be made with digital computers, and gifts them the ability to simulate digital processes and the potential to solve problems in this area.

Consider the cellular automaton defined by the simple rule:

$$a_i^{t+1} = a_{i-1}^t + a_{i+1}^t \mod 2 \tag{1}$$

For any automaton, in each time step the rule is applied to all cells in the automaton instantaneously and simultaneously. In this case, our set of states is $0, 1$, and for each cell we look at the cells immediately preceding and succeeding it: if exactly one of them is in state 1, then this cell will be in state 1 in the next time step. Otherwise it will be in state 0.

1.1 Self-similarity

Figure 1 shows a visualisation of Eq. 1. A cell is filled white if it has the value 1, and black otherwise. Note that each new "line" of the visualisation represents the automaton in the next successive time-step from the one before; this automaton is one-dimensional, and so we may represent it in its entirety as a single row of cells. This pattern exhibits two important

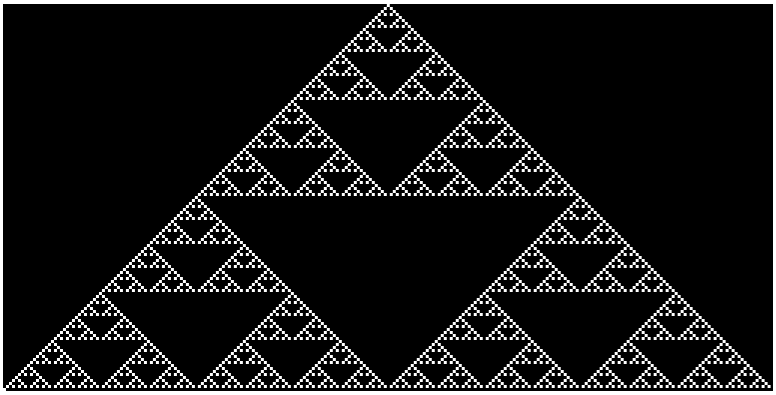


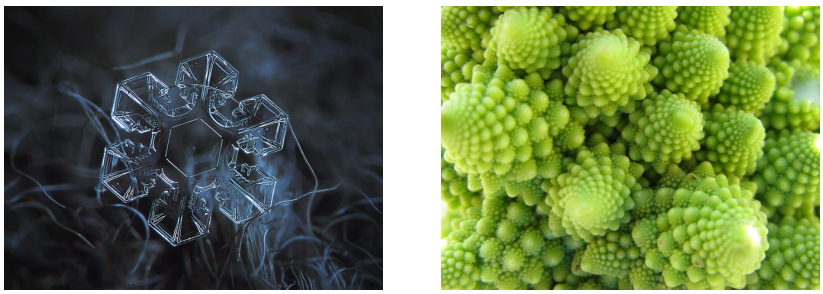
Figure 1: Pattern generated from applying Eq. 1 over 129 generations

characteristics, the first of which is "self-similarity", meaning that portions of the pattern generated are indistinguishable from the whole when magnified. As Wolfram states, in this way "the pattern is therefore invariant under rescaling... and may be characterised by a fractal dimension" [1]. One such definition of this is the Hausdorff-Besicovitch dimension [2].

As an example, suppose we have a cube with some mass, and we wish to find out how the mass scales when we try to make the same cube out of smaller copies of the original. Intuitively, one way to do this require eight smaller cubes, each of whose side length is half that of the original cube. So with the scaling factor of $\frac{1}{2}$, the mass of each smaller cube is $\frac{1}{8} = (\frac{1}{2})^3$. So we arrive at the equation $N = S^D$, where N is the number of smaller copies that can be stuck together to make the original, S is the scaling factor of these smaller copies, and D is the dimension. In this example, we compute that the dimension of a cube is 3, which is obviously correct. Back to the dimension of the Sierpinski Triangle, we can see that each larger triangle is made up of three smaller triangles, each of whose side lengths are half that of the larger triangle. So we calculate the dimension as:

$$\begin{aligned} \tfrac{1}{3} &= (\tfrac{1}{2})^D \\ \log \tfrac{1}{3} &= D \log \tfrac{1}{2} \\ D &= \tfrac{\log 3}{\log 2} \approx 1.585 \end{aligned} \tag{2}$$

Many naturally-occurring systems exhibit fractal structures, from snowflakes to pine cones to Romanesco broccoli, and raises the possibility that these are generated through the evolution of some natural cellular automata or similar processes.



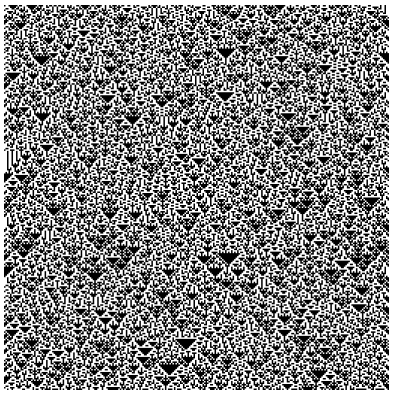
(a) Snowflake (b) Romanesco broccoli

Figure 2: Snowflakes and Romanesco broccoli exhibit fractal structures

1.2 Self-organisation

Second of the characteristics exhibited by this cellular automaton is "self-organisation", whereby from an initial disorderly or chaotic state there appear ordered structures seemingly spontaneously. This is illustrated in Fig. 3 below: one can clearly see the emergence of similar triangular structures in the cone snail's shell, appearing from a chaotic background. This starting chaos is modelled in the cellular automaton by randomly assigning each cell a value of 0 or 1, with each occurring with probability one half.

(a) Shell of a cone snail



(b) Rule 90 with random starting configuration

Figure 3: Natural processes can exhibit structures found in automata

2 Classifying automata

In order to classify automata, we can take into account both the size of the neighbourhood each cell considers going into the next generation, as well as the number of possible states in which a cell may be. Call these N and Q respectively; in the case of Eq. 1, $N = 3$ and $Q = 2$, and so the total number of different "transitions" from one state to the next is $N^Q = 8$. We may write out all of these transitions as follows:

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

Reading the bottom row of this transition table, we get the binary number $01011010_{\text{bin}} = 90_{\text{dec}}$, and so we name this rule set Rule 90. Using this convention, we may now construct our own automata based solely on the knowledge of its name (at least for elementary cellular automata). For example, take Rule 110, introduced by Wolfram 1983 [3]. We first convert 110_{dec} to binary, and then use this to fill in the table as above:

111	110	101	100	011	010	001	000	$\Rightarrow$	111	110	101	100	011	010	001	000
?	?	?	?	?	?	?	?		0	1	1	0	1	1	1	0

To convert this to a concrete set of rules to simulate in a computer programs (in a more elegant way than a collection of eight `if` statements), we can put the information we have gathered so far in a Karnaugh map to get a Boolean algebraic expression of the ruleset, as in the table below. We use A , B, and C to denote a_{i-1}, a_i, and a_{i+1}, at time t, respectively. From this we can reduce the ruleset to the Boolean expression in Eq. 3 by observation.

C \ AB	00	01	11	10
0	0	1	1	0
1	1	1	0	1

$$\begin{aligned} B^{t+1} &= \overline{A} \cdot C + B \cdot \overline{C} + \overline{B} \cdot C \\ &= \overline{A} \cdot C + B \oplus C \end{aligned} \tag{3}$$

The resulting visualisation in Fig. 4 exhibits similar characteristic of self-organisation, however it is hard to say whether it shows self-similarity as well.

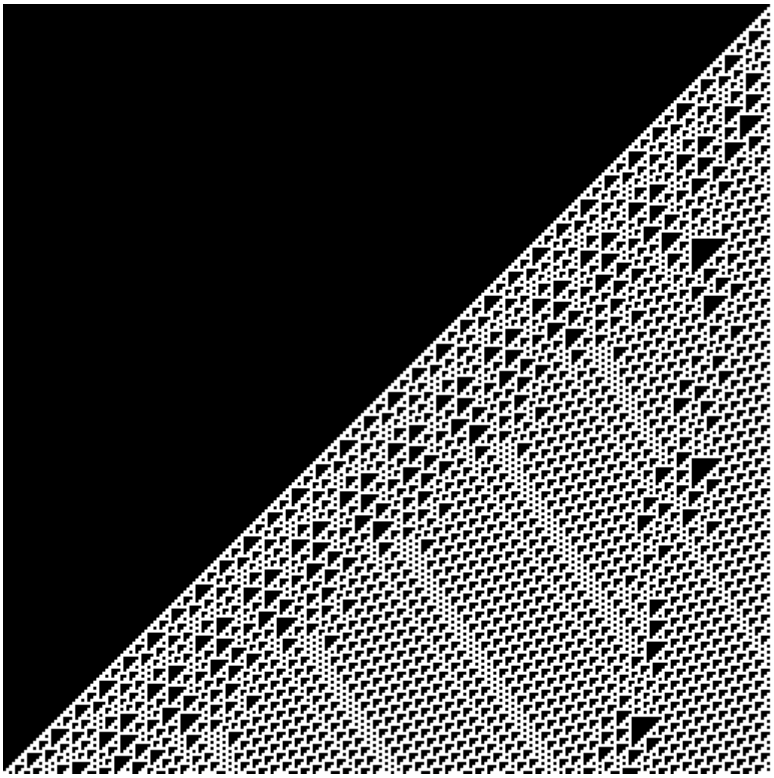


Figure 4: Rule 110 generated using Eq. 3

2.1 Elementary cellular automata

Elementary cellular automata are the simplest class of one-dimensional cellular automata; each cell may be in one of two states, 0 or 1, and rule sets only take into account the cell itself and its immediate neighbours (the nearest cell left and right of the cell being computed). Since there are a total of $2^3 = 8$ combinations of values for a neighbourhood of cells, and each of these neighbourhoods causes a cell to move to one of two states in the next generation, this means there are $2^8 = 256$ elementary cellular automata. Their evolution is often illustrated by each progressive "new line" showing the next generation of the automaton, as caused by applying the rule set to the current generation.

Two technically different automata may give rise to similar patterns, and indeed many automata are equivalent to each other thanks to simple transformations of their underlying geometry. The first of these transformations is reflection in the vertical axis; the result of

applying this is known as the mirror rule [4]. For example, take Rule 30. Each time some neighbourhood produces a live cell in Rule 30, let's take this neighbourhood and reflect it in the vertical plane, giving the table below:

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

The first line of the table shows the ruleset for Rule 30, while the line below gives that of our new ruleset, made by applying the mirror rule to Rule 30. Thus we get Rule 30's mirror rule, Rule 01010110_{bin} = Rule 86_{dec}. Rules which generate the same ruleset under mirroring are called amphichiral, and of the 256 elementary cellular automata, 64 are so.

The second of these transformations is to exchange the roles of 0s and 1s in the ruleset definition, and doing so is applying the complement rule [4]. Again applying this to Rule 30, we get the table below (note that the roles of 0s and 1s are exchanged in both the top and bottom rows):

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

So we arrive at Rule $10000111_{\text{bin}} = 135_{\text{dec}}$ as the complement of Rule 30. There are 16 rules which are the same as their complement. Both of these rules can be applied at once to a ruleset, and the result is Rule 149 when applied to Rule 30. There are 16 rules which are the same as their mirrored complementary rules. There are 88 rules which are inequivalent under these transformations [4].

2.2 Wolfram classes

Wolfram gives the following classes for categorising the behaviours of different automata:

- Class One: rapidly converge to a uniform state e.g. Rule 0, 232
- Class Two: rapidly converge to a stable or oscillating state e.g. Rule 250
- Class Three: appear to evolve in a chaotic fashion e.g. Rule 30, 150, 182
- Class Four: form areas of repetitive or stable states, but also structures which interact in complicated ways e.g. Rule 54

Two of these 256 rules are particularly special, namely Rule 54 and 110, both of which are Class 4 elementary cellular automata. Rule 110 has been proven to be universal, or Turing complete, meaning it is theorectically able to simulate any function. Meanwhile, it is so far unknown whether Rule 54 is capable of universal computation – interacting structures form, but it remains to been seen whether structures useful for computation arise. Computational universality is discussed more in-depth later on.

There have been attempts to classify automata more rigorously, and Culik and Yu proposed four well-defined classes for such a task; membership in such classes has, however, been shown to be undecidable [5].

2.3 Two dimensional cellular automata

Two dimensional cellular automata become a bit more interesting than their dimensionally-deficient brethren. Rather than having a single row of cells to consider at each generation, now a two-dimensional plane of cells is used to compute the evolution of the automaton. Whereas

elementary automata looked only at the neighbourhood of a_{i-1}, a_i, and a_{i+1} when applying the set of rules, two-dimensional automata typically have a larger neighbourhood to consider. Two of the most common types of neighbourhood are the Moore neighbourhood and the von Neumann neighbourhood [6]. The former includes the 8 cells directly surrounding the current cell being considered, while the latter only focuses on the four orthogonally ajacent cells. Occasionally, one may use the extended von Neumann neighbourhood, for which the next four closest orthogonally adjacent cells are also included.

Using a Moore neighbourhood and a set of states of size k, we can see that the total number of configurations for one particular neighbourhood is k^9, as each of the nine cells included in the neighbourhood may be in one of k states. For an automaton such as Conway's Game of Life, where each cell is either alive or dead and a moore neighbourhood is used, there are $2^9 = 512$ ways to configure one of these neighbourhoods alone. The general equation for the number of automata for a given neighbourhood and number of possible states is k^{k^n}, where k is the number of possible states and n is the size of the neighbourhood used when determining the automaton's evolution between time steps (including the cell being computed); in the case of a two-dimensional automaton using two states and a Moore neighbourhood (such as Game of Life), there are 2^{2^9} possible automata [7].

2.4 Game of Life and Life-like automata

At this point it makes sense to talk a bit more about Conway's Game of Life. It was invented in 1970 by the British mathematician John Conway, after developing an interest in them after their introduction by John von Neumann in the late 1940s, who aimed to discover a hypothetical machine that was capable of recreating itself [8]. Von Neumann was able to achieve this goal using a complex automaton, with 29 states and complicated rules to do so [9]. Von Neumann's automaton is discussed in more depth later. The Game of Life was thus Conway's way of simplifying von Neumann's ideas; in fact, not only can the Game of Life simulate the Game of Life, it is also capable of universal computation, a topic discussed more in-depth later. The rules for Game of Life are as follows:

- If the current cell $a_{i,j}^t$ is dead, then:
  - if the number of alive surrounding cells is 2 or 3, then $a_{i,j}$ will become alive in the next generation
  - else $a_{i,j}$ remains dead
- If the current cell $a_{i,j}^t$ is alive, then:
  - if the number of alive surrounding cells is less than 2, then $a_{i,j}$ dies as if through isolation
  - if the number of alive surrounding cells is greater than 3, then $a_{i,j}$ dies as if through overcrowding
  - else $a_{i,j}$ lives on to the next generation

There are many fascinating structures which arise out of these simple rules, and can be classed as follows:

- Still Life - patterns that do not change at all from one generation to the next
- Oscillators - patterns which generate some sort of cycle of patterns in a fixed place with a specific period
- Gliders and Spaceships - patterns which cycle through some finite set of patterns, but whose position is shifted each time, appeaing to move

- Guns - repeating patterns which produce a spaceship after a finite number of generations
- Puffers - moving patterns which leave a trail of stable or oscillating debris at regular intervals
- Rakes - moving patterns which emit spaceships at regular intervals as they move
- Breeders - oscillating patterns which deposit guns at regular intervals. Unline guns, puffers, and rakes, who each have a linear growth rate, breeders exhibit exponential growth rate

On 6^{th} March, 2018 the first knightship was discovered by Adam P. Goucher, named Sir Robin [10]. A knightship is a spaceship which moves two squares horizontally for every square it moves vertically, as opposed to regular spaceships, which move only vertically or horizontally, or gliders, which move exactly diagonally. Sir Robin became the first new spaceship movement pattern for an elementary spaceship to be discovered in forty eight years [11].

Conway originally believed that no pattern could grow infinitely, i.e. for the population to grow past any upper limit for some initial configuration with finite number of alive cells, and offered fifty dollars to the first person who could prove or disprove this conjecture before the end of 1970. In November of that year a team from the Massachusetts Institute of Technology, led by Bill Gosper, found a structure which produced gliders every 30 generations, and called it the Gosper Glider Gun. Later on, smaller patterns were found to produce infinite growth, one of which has been proven to be minimal, starting off with only ten alive cells [12]. More of what this means for the computational power of Game of Life, as well as other automata, will be discussed under Computational Universality.

A cellular automaton is Life-like (similar to Game of Life) if it meets the following criteria [13]:

- it is two-dimensional
- each cell of the automaton may be in one of two states, alive or dead (or on or off) at any point
- a cell's Moore neighbourhood is used to compute the cell's state in the next generation
- in each time step of the automaton, the next state of the cell can be expressed as a funciton of the number of adjacent cells that are in the alive state and of the cell's own state

As with elementary cellular automata, it is helpful to refer to specific rulesets in a more generalised way. Where this was using the rule's binary encoding for one-dimensional automata, one such convention for two-dimensional automat is using a string Bx/Sy, where x and y are a sequence of digits, 0 to 8, in ascending order. The presence of a digit d in x means that a dead cell with d alive neighbours will become alive in the next generation, while the presence of d in y signifies that an alive cell with d alive neighbours will continue to live on to the next generation. For example, Game of Life is denoted B3/S23: a dead cell is **B**orn if there are exactly 3 alive neighbours surrounding it, and and alive cell **S**urvives if there are two or three alive cells surrounding it.

2.5 Other types of automata

2.5.1 Reversible cellular automata

A cellular automaton is reversible if, for every configuration of the automaton, exactly one configuration that causes it (its preimage). If we consider the automaton as a function that maps configurations to configurations, then for a reversible automaton this function would be

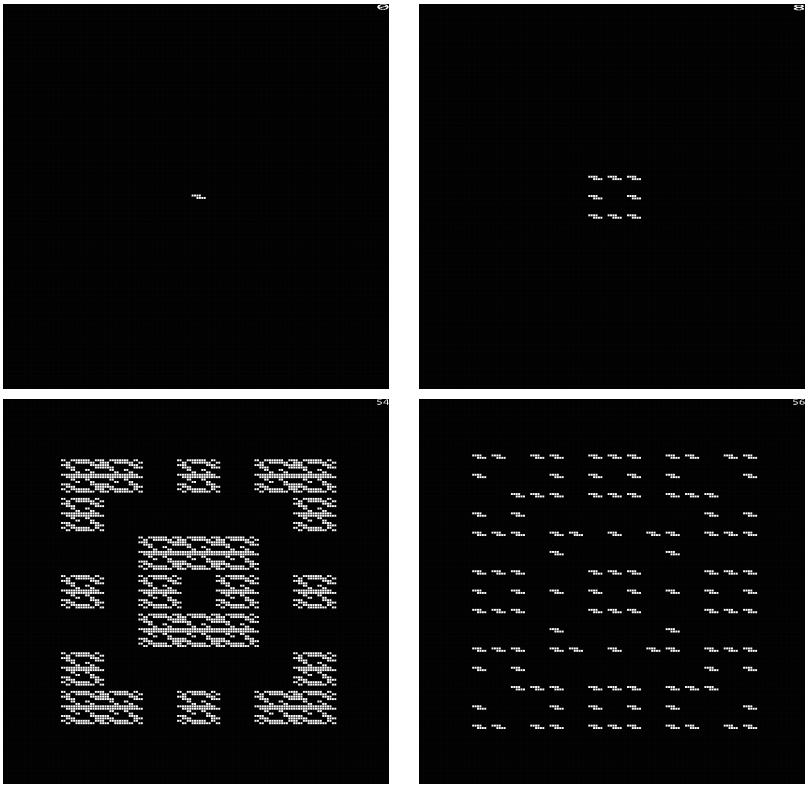


Figure 5: Replicator (B1357/S1357) replicating itself over the course of 56 generations

bijective. Where this is not the case, where not every configuration has a preimage, these configurations without preimages are called Garden of Eden patterns [14].

There are algorithms to compute whether or not a one-dimensional cellular automaton is reversible [15]; the same is not true for higher dimensions however, and computing reversibility for cellular automata in two-dimensions and up is undecidable [16].

2.5.2 Totalistic cellular automata

A cellular automaton is totalistic if the value of cell a at time t depends on the sum of the values of the cells in its neighbourhood at time $t-1$, with the values of cells being taken from some finite set [17]. An automaton is outer totalistic if it takes into account itself as well as the total of its neighbours at time $t-1$; Conway's Game of Life is an example of an outer totalistic cellular automaton where all cell values are 0 or 1, as are all other Life-like automata.

2.5.3 Probabilistic cellular automata

2.5.4 Cyclic cellular automata

3 Computational Universality

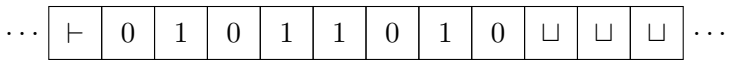
A system of rules, be they from a computer's instruction set, programming language, or cellular automaton, is called computationally universal, or Turing-complete, if it can be used to simulate a Turing machine. Before getting into the computational universality of cellular automata, we need to discuss the concept of Turing machines and what it means for a function to be computable.

3.1 Turing Machines

A Turing machine is a mathematical model of computation which was first proposed by Alan Turing in 1936 [18]. It consists of a infinite tape divided into cells each of which may contain one character from some finite alphabet, and a printer head capable of reading from and writing to the tape on the cell at which it is currently positioned. In a more formal sense, it is the seven-tuple $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$, where Q, Σ, Γ are all finite sets and

- Q is the set of states
- Σ is the input alphabet not containing the blank symbol, $\sqcup$
- Γ is the tape alphabet, where $\sqcup \in \Gamma$ and $\Sigma \subseteq \Gamma$
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R, S\}$ is the transition function, signifying whether the tape head should move left, right, or stay in place, respectively
- $q_0 \in Q$ is the start state
- $q_{\text{accept}} \in Q$ is the accept state, and
- $q_{\text{reject}} \in Q$ is the reject state, with $q_{\text{accept}} \neq q_{\text{reject}}$

It is conventional to use $\vdash$ to signify the start of the input string, and an infinite string of $\sqcup$ to show the end of the input string. For example, the string 01011010 we would expect to be represented on the tape as:



We call a set of strings Turing-recognisable [19] if we can construct some Turing machine to recognise it, that is, every input we feed it that is in the language will cause the Turing machine to eventually move to the accept state. Notice, however, that while there are two "destination" states, q_{accept} and q_{reject}, there is a third possible outcome: that our machine does not halt, but instead loops somewhere in its proceedings infinitely. Thus we call a set of strings decidable if our Turing machine always halts for its inputs, that is, for any input we feed it we are guaranteed to land in either q_{accept} or q_{reject}.

As an example of a Turing machine in action, suppose we want to construct a one to accept the language $L = \{w\#w | w \in \{0,1\}*\}$. This is not Context Free [**?**], meaning Turing Machines are at least more powerful than Push Down Automata. We construct the machine informally as follows:

$M =$ "on input w:

1. Find a decomposition of w into $w_1\#w_2$
2. Start at the first symbol of w_1, replace the symbol with a cross, and skip forward to the first symbol of w_2.
3. if this symbol is the same as the one you just crossed off, cross it off and rewind to the symbol on the right of the first symbol of w_1
4. else reject
5. Repeat this process, each time skipping forward and rewinding so that you are inspecting the first symbol to the right of a cross each time
6. If you end up inspecting a cell with the symbol #, and the symbols either side of it are crosses, accept"

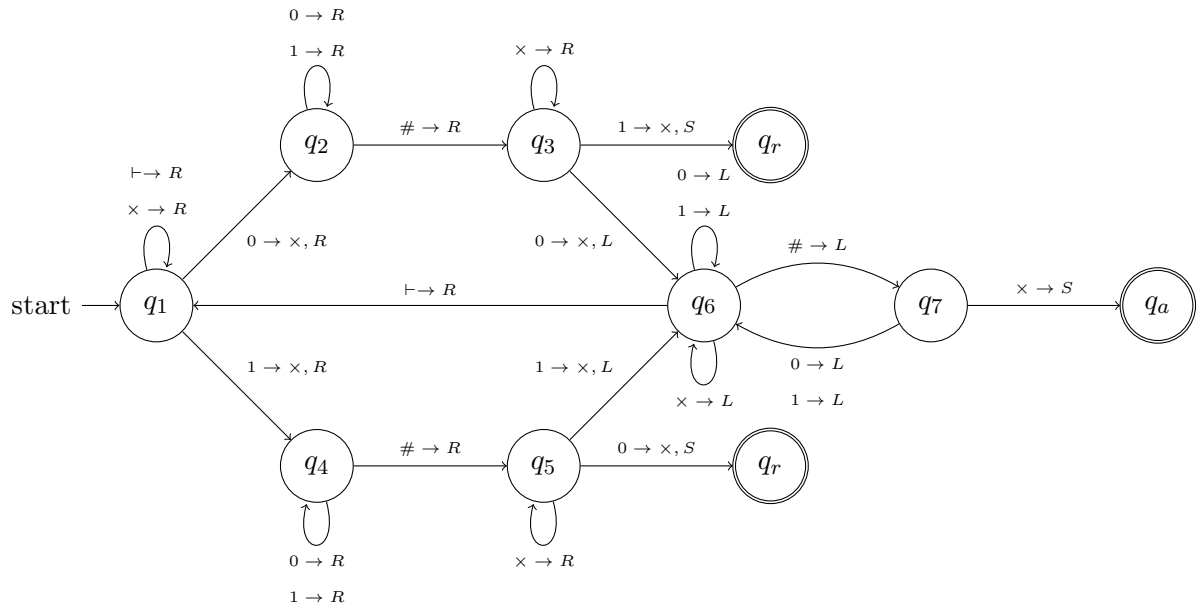


Figure 6: A possible Turing Machine to recognise L

The final thing to cover in order to understand how cellular automata such as the Game of Life can be used to simulate a Turing Machine is the concept of a Universal Turing Machine. A Universal Turing Machine is one which recognises the language

$$A_{TM} = \{\langle M, w\rangle \,|M \text{ is a Turing Machine and } M \text{ accepts } w\}$$

The $\langle M, w\rangle$ notation is used to denote the encoding of the Turing Machine M and its input w in a way that some Turing Machine would be able to understand and use as input. In essence, Universal Turing Machines are those which are capable of simulating any other Turing Machine just from its description and an input, and it is from this angle that we will look at the universality of certain cellular automata. In case this seems like a bit of a leap, we can simply describe the Universal Turing Machine U as follows:

$U =$ "on input $\langle M, w\rangle$, where M is a Turing Machine and w is a string:

1. Run M on input w

2. If M ever enters its accept state, accept; if M ever enters its reject state, reject"

Note that U will never halt if M never halts, and so while it recognises the language A_{TM}, it does not decide it. In fact, A_{TM} is called the Halting Problem and is undecidable: there is no way for an algorithm to determine whether some Turing Machine M may or may not halt on input w, without simulating it itself.

3.2 Universality in the Conway's Game of Life

A Turing Machine may compute a function by starting with the input to the function on the tape and halting with the output of the function on the tape. A function $f : \Sigma* \to \Sigma*$ is computable if some Turing Machine M halts, for every input w, with just the result $f(w)$ on the tape [20]. So for each computable function, there is a Turing Machine T such that running T on input x always halts and outputs the correct answer (i.e., ends with the correct answer on the tape). Thus, any model we can devise which could simulate the run of a Turing Machine on some input would be capable of computing all the functions that the Turing Machine could, that is, all computable functions, and hence such a model would be Turing complete.

Processor instruction sets may be fully implemented using just a NOT gate and one of either AND or OR in order to compute all binary logic. All that remains to obtain Turing Completeness is access to infinite, random access memory. Of course, given the finite nature of computers, this is impossible, but as long as the memory is as large as is required by the Turing Machine, this will suffice. Memory can be implemented in logic circuits using D-type flip flops, which can be done with four NAND gates. Thus, if this can be implemented using a cellular automaton, that automaton will be a universal computer.

The key to Game of Life's universal computation is its ability to create infinite growth, for example by using glider guns. Guns can then be positioned to shoot at blocks in specific locations, and doing so correctly can be used to move the block closer or further away. This sliding block memory can be used to simulate a counter [**?**]. Logical AND, OR, and NOT gates can also be constructed using gliders, with the presence of a glider signifying a 1, and 0 otherwise. These three Boolean operators are theoretically necessary and sufficient to build a Universal Turing Machine [21]. A fully functioning Turing Machine was designed to completion by Paul Rendell on 2nd April, 2000 [22], which doubled the number on the input tape. Further extensions led to the development of Universal Turing Machines in February 2010 and March 2011, both of which were capable of simulating the Turing Machine from 2000 [23, 24].

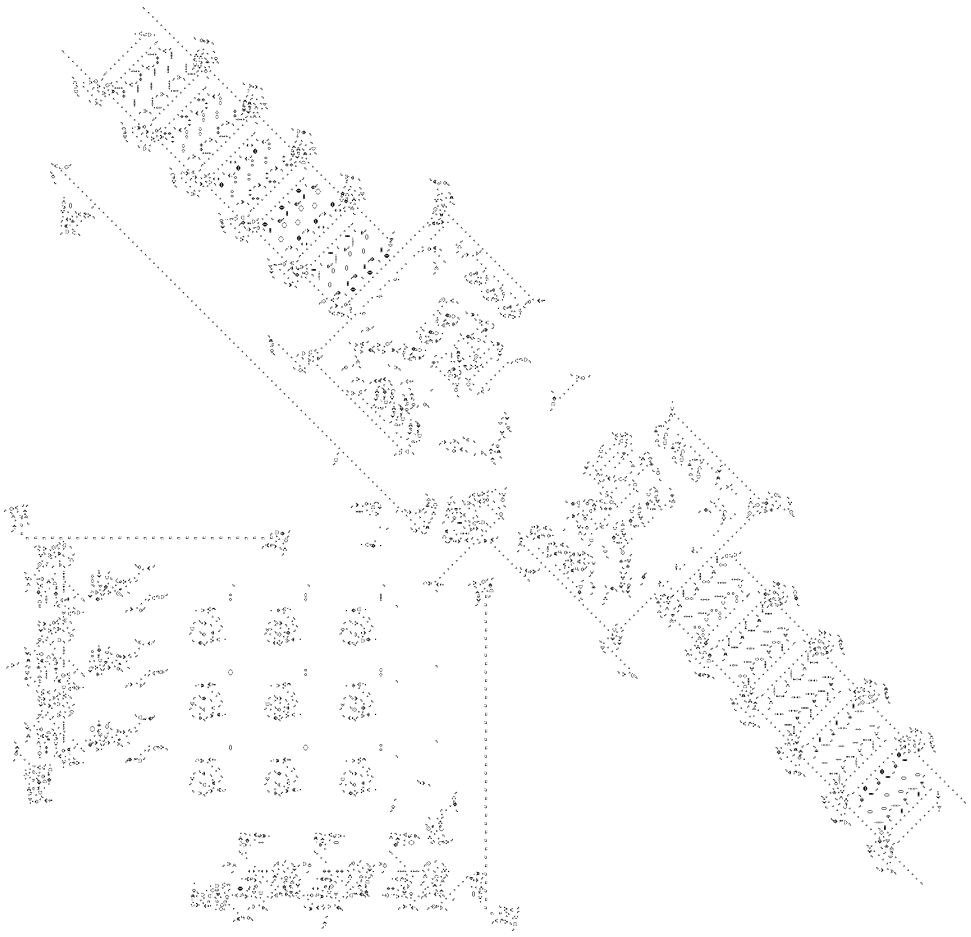


Figure 7: Section of a Turing Machine implemented in Conway's Game of Life

4 More cellular automata

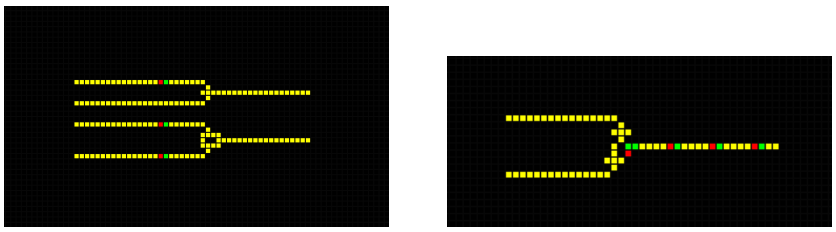
4.1 WireWorld

WireWorld is a two-dimensional cellular automaton first proposed by Brian Silverman in 1987. The ruleset is relatively simpleset, with four distinct states:

1. empty
2. conductor
3. electron head
4. electron tail

As the state names may betray, WireWorld is particularly good at simulating electronics. Like Game of Life, WireWorld uses a Moore neighbourhoodin its calculations, as opposed to the more restricted Von Neumann neighbourhood.The rules are as follows:

- *empty* → *empty*
- *head* → *tail*
- *tail* → *conductor*
- *conductor* → *head* if exactly 2 neighbours are *head*, else *conductor*



(a) OR and XOR gates (b) D-type flip flop

Figure 8: All the structures required for universal computation implemented in WireWorld

4.1.1 Computational Universality

As with Game of Life, WireWorld can be used to implement the logic gates required for Turing Completeness, and is inherently more suited to doing so. A full computer has been constructed in WireWorld, which calculates prime numbers [25]. Langton's Ant, a cellular automaton that will be discussed shortly, has also been implemented in WireWorld [26].

4.2 Rule 110

4.2.1 Computational Universality

The key to Rule 110's computational universality is its ability to simulate a "tag system", which is universal, using gliders inherent to the automaton [27]. The details are somewhat beyond the scope and level of this report, but Cook first proves that a tag system that removes two symbols at each stage is universal by compiling a two-state Turing Machine program. After this, he shows that such a tag system may be implemented using these gliders in Rule 110, and consequently shows that this cellular automaton is capable of universal computation. Some such gliders used by Cook in his proof of Rule 110's universality are shown in Fig. 9.

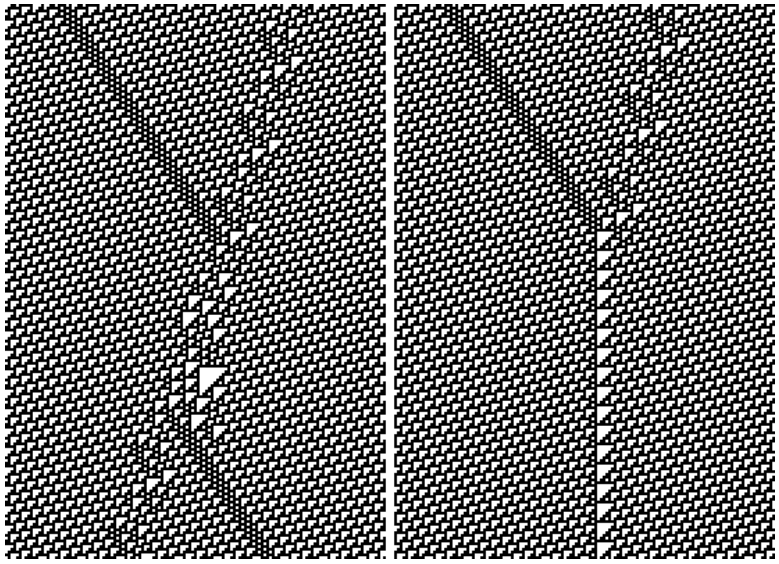


Figure 9: Gliders that interact by a) translation and b) forming a new structure

Due to Rule 110's simplicity, it suggests naturally occurring physical systems may also be capable of universality (systems similar to the way the shapes formed on the cone snail's shell, for example). Therefore, many of the properties of such systems would be undecidable, for which it is impossible to find closed-form mathematical solutions.

4.3 Von Neumann's automaton

Cellular automata can trace their roots back to the work done in the late 1940s by Stanislaw Ulam and John von Neumann. Von Neumann wanted to ultimately answer the question of whether a machine was capable of replicating itself. We are used to automata creating structures much simpler than themselves, for example a factory that produces bolts: presumably, one section of the machine would cast the metal, while another would make the thread. He was concerned whether there was some form of "extra-mechanical magic" to self-reproduction [28].

By 1952, von Neumann had completed his goal of engineering an automaton capable of recreating itself. It used a total of 29 states, and each cell's state in the next generation is calculated by considering its von Neumann neighbourhood in the current one. A summary of the states is as follows:

1. one ground state, U
2. eight transition states, $S, S_0, S_{00}, S_{000}, S_{01}, S_1, S_{10}, S_{11}$
3. four confluent states, $C_{00}, C_{01}, C_{10}, C11$
4. eight ordinary transmission states in directions North, East, South, and West
    - North-quiescent, North-excited
    - East-quiescent, East-excited
    - South-quiescent, South-excited
    - West-quiescent, West-excited
5. eight special transmission states in the same directions
    - North-quiescent, North-excited

- East-quiescent, East-excited
- South-quiescent, South-excited
- West-quiescent, West-excited

The ruleset can be broken down into four sections, concerning those for construction, destruction, transmission states, and confluent states. They are extensive and a full summary can be found at [29].

4.4 Fractran

4.5 Langton's Ant

5 Applications

5.1 Cryptography

6 Firing Squad Synchronisation Problem

6.1 The Problem

A firing squad consists of a group of soldiers, all of whom will fire a shot at the condemned party in order to end his or her life, at the command of a general. An important factor to this is that all the soldiers fire simultaneously; this means that no one soldier can be sure that they are the one to have administered the fatal shot, and can thus rest easy at night. If we model the line of soldiers as a one-dimensional cellular automaton, we want all of the cells in this automaton to be in a "firing" state simultaneously. More specifically, at each time step each soldier may change what he is doing (i.e. change his state) based solely upon what he and his two immediate neighbours are doing. Note that the soldiers at either end of the line only consider the actions of himself and his sole neighbour. Furthermore, we assume that each soldier starts in the same inactive state (also known as the quiescent state), and that the general is located at one end of the line, and only he may give the initial command to fire. He himself must also fire his weapon at the to-be recipient of the discharge of such weapons. Thus the question asked by the Firing Squad Synchronisation Problem is this: assuming there is a fixed number of states, and regardless of how many soldiers form the firing squad, how can the states and transitions be defined such that all soldiers are all, at some moment in time, firing together?

6.2 The Solution

Since the signal can only travel at most one cell to the left/right between each time step, it is clear that a minimal-time solution can not take shorter than $2(n-1)$ time (the signal must travel all the way down the line and all the way back, on the way down skipping the general and on the way back bouncing straight off the end soldier). The first minimal-time solution to the problem was Abraham Waksman's, published in 1966 [30]. This solution uses the following six basic states:

1. Q - the starting, quiescent state
2. T - the final, firing state
3. R - the trigger signal for the B state:
    - R_0 - propagates left
    - R_1 - propagates right

4. B - the state that generates the P state
    - B_0 - blocks the R state
    - B_1 - passes through the R
5. P - the state generating the A state (leads to T if both neighbours in P state)
    - P_0 - generates A_{0xx}
    - P_1 - generates A_{1xx}
6. A - the propagating state
    - $A_{000}, A_{001}, A_{010}, A_{011}, A_{100}, A_{101}, A_{110}, A_{111}$
    - A_{0xx} - generates the state R with no delay
    - A_{1xx} - generates the state R with one unit time delay
    - A_{x00}, A_{x01} - propagates to the left
    - A_{x10}, A_{x11} - propagates to the right

There are also two extra placeholder states:

- ϕ - external state, no neighbour to this side
- γ - neighbours note explicitly mentioned

The way Waksman's solution works can be summarized by the interactions between the states of his automaton which occur:

- Every A signal can be assigned a parity, even or odd, based on whether it is at an even or odd cell. The parity is relative to the cell who originated the signal
- Every cell with an even A signal generates a new R signal that travels in the opposite direction of the cell's A signal
- Each type of B signal will switch to the other type upon intersection with an R signal. One type allows the R signal to continue propagating, the other does not
- When A and B signals intersect, they set up a new generator, or P signal

Of course, time complexity is not the only metric by which to judge a solution's merit. The first solution to the problem, proposed by John McCarthy and Marvin Minsky, was relatively much more simple, involving propagating two waves down the line of soldiers, one moving at three times the speed of the other. The faster wave bounces off the edges and meet again in the center, creating two more waves for a total of four waves. This process repeats, until each cell is part of a wave simultaneously. At this point, the soldiers fire. This requires $3n$ units of time for n soldiers, and requires 15 states.

The current best solution to the problem was devised by Jacques Mazoyer [31], and uses six states to solve the problem in time $2(n-1)$.

In minimal-time solutions, the general sends signals $S_1, S_2, S_3, ..., S_i$ to the right (assuming he is the left most soldier) at speeds $1, \frac{1}{3}, \frac{1}{7}, ..., \frac{1}{2^{i-1}-1}$. S_1 reflects and meets signal S_i (for $i \geq 2$) at cell $\frac{n}{2^{i-1}}$. When S_1 reflects it also creates another general at the right end, and the process repeats. Fig. 10 shows Mazoyer's minimal-time and so far minimal-state solution to FSSP.

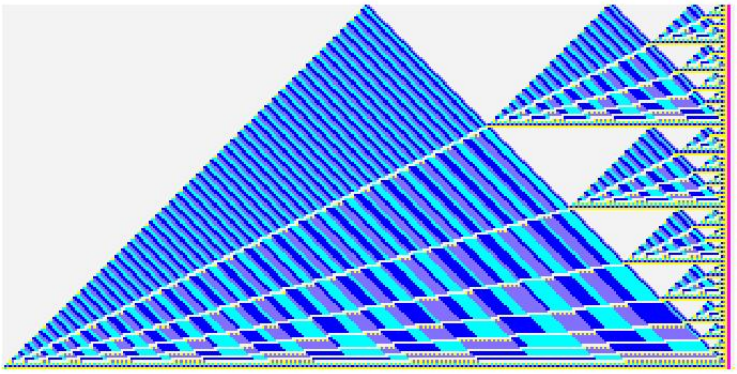


Figure 10: Mazoyer's minimal solution to FSSP. Time increases from left to right.

7 Majority Problem

7.1 The Problem

Suppose we have a one-dimensional cellular automaton, each of whose cells may be in one of two states. There are $i + j$ cells in total, i of which are in the zero state, the other j of which are in the one state. A correct solution to this problem must eventually set all cells to zero if $i > j$, or one if $i < j$. The desired final state of the automaton is unspecified if $i = j$.

7.2 The Solution

Gács, Kurdyumov, and Levin designed an automaton the solves the Majority Problem in most cases, but not all. In fact, with a random starting configuration, their solution is about 78% successful in correctly determining the majority [32]. They approached it by classifying the quality of a particular cellular automaton by the proportion of the 2^{i+j} possible starting configurations in which it correctly identified the majority.

An overview of their method is as follows: if a cell is 0, the next state is the majority value between itself, its neighbour immediately to its left, and its neighbour three spaces to the left. Similarly, if the cell is a 1, the value is taken as the majority of itself, its immediate right neighbour, and its neighbour three spaces to the right.

Capcarrere, Sipper, and Tomassini observed [33] the Majority Problem may be solved perfectly by altering the definition of having recognised the majority. For example, if Rule 184 is run on a finite universe with cyclic boundaries, each cell will eventually see two consecutive states of the majority value infinitely often, but will see two consecutive states of the minority value only finitely many times. Thus the Majority Problem cannot be solved perfectly if we require all cells to eventually stabilise to the majority state, but can if we relax this requirement and allow Rule 184 to run infinitely [34].

References

[1] S. Wolfram, "Cellular automata," *Wolfram on Cellular Automata and Complexity*, vol. 14, no. 3, p. 412, 1983.

[2] K. Clayton, "Basic concepts in nonlinear dynamics and chaos: Fractals and the fractal dimension." `https://www.vanderbilt.edu/AnS/psychology/cogsci/chaos/workshop/Workshop.html#Fractals`, 1996. From Society for Chaos Theory in Psychology and the Life Sciences annual meeting. Page accessed: 2018-08-03.

[3] S. Wolfram, *A New Kind of Science*, p. 676. Wolfram Media, 2002.

[4] "Elementary cellular automaton." https://en.wikipedia.org/wiki/Elementary_cellular_automaton. Page accessed: 2018-08-19.

[5] K. Culik II and S. Yu, "Undecidability of ca classification schemes," *Complex Systems*, vol. 2, pp. 177–190, 01 1988.

[6] L. B. Kier, P. G. Seybold, and C.-K. Cheng, "Modeling chemical systems using cellular automata," p. 15.

[7] I. Bialynicki-Birula and I. Bialynicka-Birula, *Modeling Reality: How Computers Mirror Life*. Oxford University Press.

[8] "Game of life: A brief history." https://web.stanford.edu/~cdebs/GameOfLife/#history. Page accessed: 2018-08-20.

[9] J. von Neumann, *Theory of Self-Reproducing Automata*. Champaign, IL, USA: University of Illinois Press, 1966.

[10] "Elementary knightship." http://www.conwaylife.com/forums/viewtopic.php?f=2&t=3303. Page accessed: 2018-09-12.

[11] "Conway's game of life." https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life#Notable_programs. Page accessed: 2018-09-12.

[12] "Eric weisstein's treasure trove of life: Infinite growth." http://www.ericweisstein.com/encyclopedias/life/InfiniteGrowth.html. Page accessed: 22-08-2018.

[13] "Life-like cellular automaton." https://en.wikipedia.org/wiki/Life-like_cellular_automaton. Page accessed: 2018-08-23.

[14] J. L. Schiff, "Cellular automata: A discrete view of the world," p. 40.

[15] K. Sutner, "De bruijn graphs and linear cellular automata," *Complex Systems*, vol. 5, pp. 19–30.

[16] J. Kari, "Reversibility of 2d cellular automata is undecidable," *Physica*, vol. 45, pp. 379–385.

[17] S. Wolfram, *A New Kind of Science*, p. 60. Wolfram Media, 2002.

[18] M. Sipser, *Introduction to the Theory of Computation*, ch. 3: The Church-Turing Thesis, p. 139. Course Technology, Cengage Learning, 2 ed., 2006.

[19] M. Sipser, *Introduction to the Theory of Computation*, ch. 3: The Church-Turing Thesis, p. 144. Course Technology, Cengage Learning, 2 ed., 2006.

[20] M. Sipser, *Introduction to the Theory of Computation*, ch. 5: Reducibility, p. 210. Course Technology, Cengage Learning, 2 ed., 2006.

[21] J.-C. Heudlin, *L'Évolution au bord du chaos*, p. 122. Hermès, Paris, 1998.

[22] "http://www.rendell-attic.org/gol/tm.htm." ThisisaTuringMachineimplementedinConway'sGameofLife. Page accessed: 2018-09-13.

[23] "This is a universal turing machine (utm) implemented in conway's game of life.." http://www.rendell-attic.org/gol/utm/index.htm. Page accessed: 2018-09-13.

[24] "This is a fully universal turing machine (utm) implemented in conway's game of life." http://www.rendell-attic.org/gol/fullutm/index.htm. Page accessed: 2018-09-13.

[25] "The wireworld computer." https://www.quinapalus.com/wi-index.html. Page accessed: 2018-09-17.

[26] "The ed pegg jr's langton's ant binary counter." https://web.archive.org/web/20110205092221/http://www.heise.ws:80/edpeggjrcounter.html. Page accessed: 2018-09-17.

[27] M. Cook, "Universality in elementary cellular automata," *Complex Systems*.

[28] "The origins of cellular automata." http://psoup.math.wisc.edu/491/CAorigins.htm. Page accessed: 2018-09-17.

[29] "Von neumann cellular automaton." https://en.wikipedia.org/wiki/Von_Neumann_cellular_automaton. Page accessed: 2018-09-18.

[30] A. Waksman, "An optimum solution to the firing-squad synchronization problem," *Information and Control*, vol. 9, pp. 66–78.

[31] J. Mazoyer, "A six-state minimal time solution to the firing squad synchronization problem," *Theoretical Computer Science*, pp. 183–238.

[32] P. Gács, G. L. Kurdyumov, and L. A. Levin, "One dimensional uniform arrays that wash out finite islands," *Problemy Peredachi Informatsii*, pp. 92–98.

[33] M. Capcarrere, M. Sipper, and M. Tomassini, "Two-state, r = 1 cellular automaton that classifies density," *Physical Review Letters 77*, p. 4969–4971.

[34] M. Land and R. Belew, "No perfect two-state cellular automata for density classification exists," *Physical Review Letters 74*, pp. 1548–1550.