Chapter 5

1-Dimensional Cellular Automata

In this chapter, we will shift course from studying graph theory and models of social networks and instead learn the basics of *cellular automata*. Cellular automata, or CAs for short, are important structures in mathematics and computer science which simulate how a collection of spatially-related objects can evolve over time. Perhaps an example may motivate the concept of CAs more effectively.

5.1 Diseased Cafeteria Lines: A Simple 1-D CA

Consider the challenge of modeling how a contagious disease could spread through a cafeteria line. We can think of this line as a spatially-related collection of people. Each person has two neighbors, one in front and one behind. Suppose the disease can only be spread between nearest neighbors. That is, in order for an individual to have the disease, either the guy in front or the guy behind must also have the disease. We'll consider the time it takes for the disease to move between consecutive people as discrete chunks, or timesteps. So if some individual in line has the disease at time t, that person's two neighbors will catch it (if they don't have it already) at time $t + 1$.

Consider the initial condition that a sick man joins the back of the line at time $t = 0$. Then the timestep progression of the disease will follow the diagram shown in Figure 5.1. Its easy to see that at time $t = 4$, all individuals in the line will have the disease.

This example illustrates a simple CA very well. The basic unit of a CA is the *cell*. A CA contains many cells, each of which has two abstract properties: a

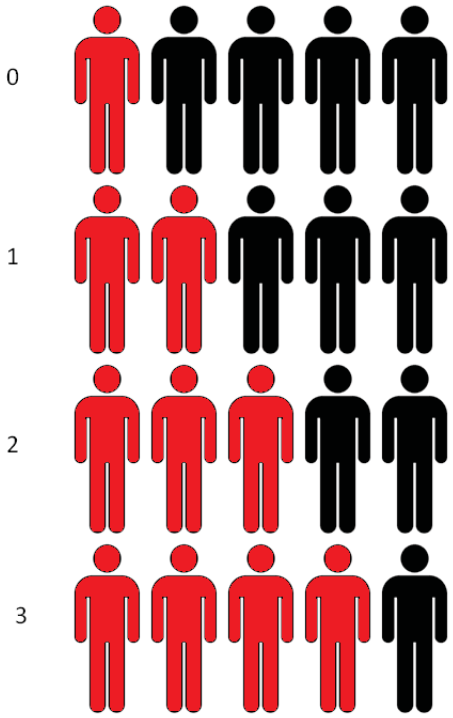

Figure 5.1: Time-based spread of infectious disease in cafeteria line. Red individuals have the disease, black do not.

state and a neighborhood. A cell's state describes its current condition. In our example, there were two possible states: sick and not sick. We often find it convenient to use integers to describe the state. That is, if a person is sick, we assign a state value of $s = 1$ to that cell, while a healthy cell will have state value $s = 0$. A cell's neighborhood is a finite set of other cells in the CA whose states influence the cell's state over time. In the cafeteria line, we decided that each person had two neighbors, one on either side. More formally, we could say that a person or cell at position i in the line had neighbors at positions $i - 1$ and $i + 1$. Of course, the people at the ends of the lines only have one neighbor instead of two, so we must account for these edge cases in our calculations.

In addition to holding a collection of spatially-related cells, a CA's purpose is to describe how the cells will evolve, or change their state, over time. For a CA, we think of time as a discrete number of steps, just like in our example. To this end, a CA will have a *rule*, which describes what a given cell's state should be at time $t + 1$ given its previous state at time t as well as the states of its neighbors at time t. Each member of the cafeteria line followed these basic rules:

- If I am sick currently, I stay sick for next time.

- If either of my neighbors are sick currently, I become sick next time.

All cells within a particular CA will follow the same rule. However, many different rule possibilities exist. Within the CA discipline, rules are often formalized in a particular way. This is largely thanks to Steven Wolfram, whose book *A New Kind of Science* helped make cellular automata famous for a number of reasons (which we'll cover later on in this and other chapters). A classic rule diagram is shown below in figure 5.1.

Within this figure, we see eight big boxes, one for each possible configuration of a line member's neighborhood. Within each big box, the particular input situation is shown as well as the rule's corresponding output. The top row of three boxes shows a possible current state of a person (middle) and her neighbors on either side. The next row shows the state of that person after one timestep. Black cells indicate sick people, while healthy line members are shown in white. We can easily check this against the known rules.

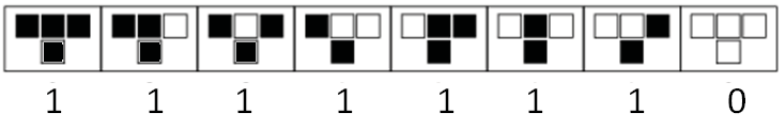

Figure 5.2: Rule diagram for the diseased cafeteria line model.

The rules for a particular CA are always listed in this order. That is, the input situation where everyone in the neighborhood has state "1" is always listed first, and so on. This allows a standardized way to describe the rule for a given CA as one particular binary number, as shown in the figure. Even better, we can convert this binary number to decimal and use it more easily in everyday language. This particular rule, as it turns out, is rule 254, since $0b11111110 = 0d254$.

5.2 The Sick Lunch Table: A 1-D Ring CA

Cellular automata are very abstract structures, so they can be extended easily to many different types of models and spatial-arrangements. To keep analysis easy, we'll constrain ourselves to one-dimensional CAs in this chapter.

1-D CAs can be surprisingly versatile. For example, suppose an individual from our cafeteria line, after catching our hypothetical disease, grabbed his lunch and joined a table of healthy classmates. How can we determine the spread of the disease in this situation?

Our old implementation doesn't quite cut it here. If the table is full (we'll assume our diseased subject is very popular), then everyone seated around the

table will have neighbors on either side. Our finite sequence model doesn't account for this ring arrangement.

So we must determine how to assign everyone at the table a numbered position while allowing everyone to have two neighbors. It turns out that modular arithmetic works very nicely for this application. For a brief primer on modular arithmetic, visit `http://en.wikipedia.org/wiki/Modular_arithmetic`.

For example, if there are n seats around the table in question, we can assign seats consecutive numbers $0, 1, \ldots n-1$. Then we can say the individual at position i has neighbors at $i-1 \bmod n$ and $i+1 \bmod n$.

Under this system, if our suspect sat down at seat 3 at the table, then we might observe the spreading behavior shown in Figure 5.2. We see that over time the disease gradually spreads from position 3 to those in seats 4, 5, 6 ... as well as those in seats 2, 1, 0, n-1, n-2, ... Even those all the way across the table won't be safe for long. According to our model, the disease should everyone in $n/2$ timesteps.

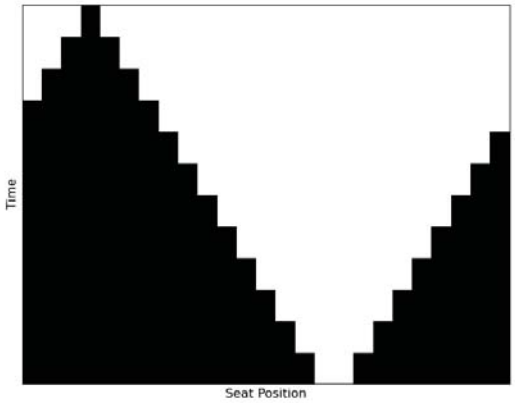

Figure 5.3: Time evolution of diseased cafeteria table.

Rule 254 is a very basic approximation of disease spread. However, the ultimate conditions it creates are not very interesting. Any initial arrangement of people with at least one sick individual will result in the entire group becoming diseased after enough time. Many other rules yield much more rich and interesting behavior.

5.3 Rule Classification

Steven Wolfram proposed that all CAs can be divided into four classes.

5.3.1 Class 1: Uniformly Predictable

Class 1 CAs represent rules that yield very predictable uniform results given almost any input condition. Rule 254 falls into this category.

5.3.2 Class 2: Intricate Patterns

Class 2 CAs yield behavior which, while exhibits intricate, predictable patterns. Rule 50, for example, is a Class 2 CA which yields the pattern shown in Figure 5.3.2. This resembles the Sierpinski Triangle, common example of a fractal shape.

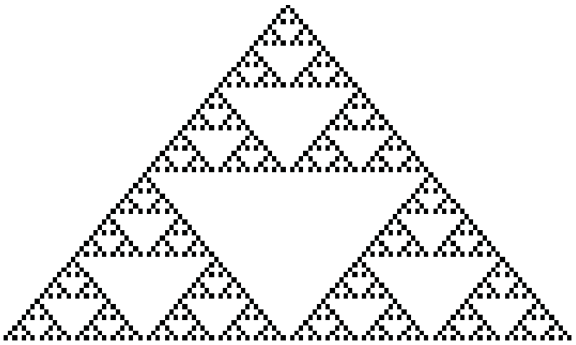

Figure 5.4: Time evolution of rule 18.

5.3.3 Class 3: Unpredictable Chaos

Class 3 CAs are known not for their patterns, but for their lack of patterns. Examine the behavior of rule 30 in the diagram below.

There does not seem to be an observable pattern in the bulk of the state table. We can see this more clearly if we look at a much larger grid over longer time period, as seen in Figure 5.3.3.

There is no visibly apparent pattern to the data shown here. But how random is it? Despite the very simple rule system here, Rule 30 has been frequently used as a pseudo-random number generator. For more details, see the next section.

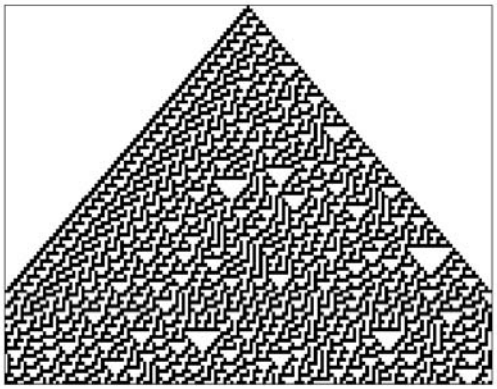

Figure 5.5: Time evolution of rule 30.

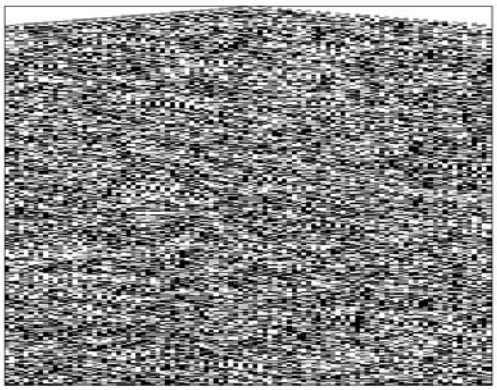

Figure 5.6: Time evolution of rule 30. Larger scale

5.3.4 Class 4: Complexity

Class 4 CAs exhibit the behavior of complex systems, meaning their time evolution results in complex structures that, while not predictable, are nevertheless not entirely random and without purpose.

Rule 110 is perhaps the most famous of this type of CA, thanks to a proof by Matthew Cook, a protege of Wolfram, that this rule can emulate a Turing machine and behave as a universal computer. Essentially, this means that applying the right initial condition, a clever and extremely patient programmer

could use a rule 110 CA to do any computation that is possible. Theoretically, at least, a rule 110 CA could add 2 + 2, determine the square root of 2 to 5 decimal places, or play minesweeper. In fact, we could even use a rule 110 CA to run calculations on other cellular automata! For more on this very interesting CA specimen, check out `http://mathworld.wolfram.com/Rule110.html` or `http://en.wikipedia.org/wiki/Rule_110`.

5.4 Randomness

Just how random is the rule 30 CA? Can it really compete with other pseudo-random numbers generators? We can compare the random bits generated by the middle column of the rule 30 CA with other sources, such as a similar length sequence using Python's `random` module, to assess if this could be a reliable PRNG.

Determining whether or not a sequence of numbers is truly random is an interesting and very challenging problem. Rigorous start-from-scratch testing requires deep understanding of statistics. Thankfully, however, many generous people have created open source analysis programs which provide many standard randomness tests. One of the most well documented and easy to use test batteries is provided by the Computer Security Division of the National Institute for Standards and Technology. See `http://csrc.nist.gov/groups/ST/toolkit/rng/documentation_software.html`.

While we cannot definitively prove that any given sequence is *random*, we can rule out sequences that are definitively non-random. That is, we can establish traits that are characteristic of random sequences and use statistical tests to verify that some given sequence of bits exhibits those traits. For our purposes, we'll focus on five rather simple but effective criteria for testing the randomness of a sequence with length n:

1. Frequency

   The basic model for randomness is that a bit occurs with uniform probability. That is, we expect each bit has a $1/2$ chance of being a zero and the same chance to be a one. This observation yields a simple test: do ones and zeros occur in a given sequence with approximately equal frequency? If we add the total number of ones, does that value reach $n/2$ within appropriate tolerances?

2. BlockFrequency

   Along with the expectation that the entire sequence have equal counts for zeros and ones, we expect that within any subsequence of length m, there would be approximately $m/2$ ones and $m/2$ zeros. We can test this block frequency observation using a block length default value of 9 digits.

3. Cumulative Sum

   If we compute a running total of the digits in a sequence of bits, there are particular values which we would and would not expect if the sequence was random. The cumulative sum test validates that the running total of normalized bits (-1 and 1 instead of 0 and 1) doesn't exceed expected tolerances.

4. Runs

   Given that the sequence is random, we expect that at various points we will observe runs (multiple consecutive zeros or ones) within the sequence. Given the uniform probability of a 0 or a 1 at each bit, we can determine exactly how likely it is that a run of length l occurs in a given sequence. The Runs test ensures that the runs observed in the given sequence to not wildly exceed the expected number of runs of each possible length.

5. Longest Run

   Finally, the longest run test examines the longest consecutive run of ones within a given sequence, and determines if this run is expected within tolerances.

This battery of tests should give us a crude method to eliminate a candidate sequence from those that are passably random. There are certainly others which are more sophisticated and comprehensive, but for the sake of understanding we'll avoid those for now.

For each of the above criteria, we test a given sequence and decide whether the sequence passes at the $\alpha = .01$ significance level. Roughly, this means we expect a truly random sequence to fail a test only 1/100 times, so the likelihood of a given sequence failing is quite low.

As a first comprehensive comparison, we obtained 50000 total bits from the middle column of a rule 30 CA and compared its test performance to the same amount generated by calling Python's `rand_int` method. Each total bit sequence was broken down into 50 sequences each 1000 bits long. The test results appear below.

Rule 30 results:

```
------------------------------------------------------------------------------
RESULTS FOR THE UNIFORMITY OF P-VALUES AND THE PROPORTION OF PASSING SEQUENCES
------------------------------------------------------------------------------
   generator: Rule 30 middle column
------------------------------------------------------------------------------
 C1  C2  C3  C4  C5  C6  C7  C8  C9 C10  P-VALUE  PROPORTION  STATISTICAL TEST
------------------------------------------------------------------------------
  3   5   5   6   3  10   5   1   9   3  0.122325   1.0000    Frequency
  1   3   4   4   4   5   8   6   6   9  0.350485   1.0000    BlockFrequency
  2   5   5   5   3   9   5   5   5   6  0.739918   1.0000    CumulativeSums
```

```
  2   5   7   6   2   3   3  10   5   7  0.213309   1.0000    CumulativeSums
  2   4   8   7   8   0   2   5   9   5  0.058984   1.0000    Runs
  4   3   4   5   8   6   5   4   7   4  0.883171   1.0000    LongestRun
```

Python `rand_int` results:

```
 ------------------------------------------------------------------------------
RESULTS FOR THE UNIFORMITY OF P-VALUES AND THE PROPORTION OF PASSING SEQUENCES
------------------------------------------------------------------------------
   generator: Python's rand_int
------------------------------------------------------------------------------
 C1  C2  C3  C4  C5  C6  C7  C8  C9 C10  P-VALUE  PROPORTION  STATISTICAL TEST
------------------------------------------------------------------------------
  3   6   6   7   2   7   3   4   6   6  0.739918   1.0000    Frequency
  1   6   2   4   4  10   4   5   6   8  0.171867   1.0000    BlockFrequency
  3   2   7   3   5   5   6   4   4  11  0.213309   1.0000    CumulativeSums
  3   6   6   3   4   4   3   5   7   9  0.616305   1.0000    CumulativeSums
  7   3   3   7   6   9   1   5   5   4  0.350485   1.0000    Runs
  8   4   5   6   7   2   4   9   3   2  0.289667   1.0000    LongestRun
```

We see that both tests had a 100% pass rate: not one of the 50 different 1000 bit sequences failed any of the tests. So at least at the 50000 bit level, the Rule 30 CA appears indistinguishable from a pseudo-random number generator. More testing is necessary to prove that both generators hold this status at larger sequence lengths. Unfortunately, generating very long sequences using CA can prove computationally intense. We'll leave more rigorous proof as an exercise to the reader.