Chapter 11

Cellular Automata I: Modeling

11.1 Definition of Cellular Automata

"Automaton" (plural: "automata") is a technical term used in computer science and mathematics for a theoretical machine that changes its internal state based on inputs and its previous state. The state set is usually defined as finite and discrete, which often causes nonlinearity in the system's dynamics.

Cellular automata (CA) [18] are a set of such automata arranged along a regular spatial grid, whose states are simultaneously updated by a uniformly applied *state-transition function* that refers to the states of their neighbors. Such simultaneous updating is also called *synchronous updating* (which could be loosened to be asynchronous; to be discussed later). The original idea of CA was invented in the 1940s and 1950s by John von Neumann and his collaborator Stanisław Ulam. They invented this modeling framework, which was among the very first to model complex systems, in order to describe self-reproductive and evolvable behavior of living systems [11].

Because CA are very powerful in describing highly nonlinear spatio-temporal dynamics in a simple, concise manner, they have been extensively utilized for modeling various phenomena, such as molecular dynamics, hydrodynamics, physical properties of materials, reaction-diffusion chemical processes, growth and morphogenesis of a living organism, ecological interaction and evolution of populations, propagation of traffic jams, social and economical dynamics, and so forth. They have also been utilized for computational applications such as image processing, random number generation, and cryptography.

There are some technical definitions and terminologies you need to know to discuss CA models, so here comes a barrage of definitions.

Mathematically speaking, CA are defined as a spatially distributed dynamical system

where both time and space are discrete. A CA model consists of identical automata (cells or sites) uniformly arranged on lattice points in a D-dimensional discrete space (usually $D = 1$, 2, or 3). Each automaton is a dynamical variable, and its temporal change is given by

$$s_{t+1}(x) = F\left(s_t(x + x_0), s_t(x + x_1), \ldots, s_t(x + x_{n-1})\right), \tag{11.1}$$

where $s_t(x)$ is the state of an automaton located at x at time t, F is the *state-transition function*, and $N = \{x_0, x_1, \ldots, x_{n-1}\}$ is the *neighborhood*. The idea that the same state-transition function and the same neighborhood apply uniformly to all spatial locations is the most characteristic assumption of CA. When von Neumann and Ulam developed this modeling framework, researchers generally didn't have explicit empirical data about how things were connected in real-world complex systems. Therefore, it was a reasonable first step to assume spatial regularity and homogeneity (which will be extended later in network models).

s_t is a function that maps spatial locations to states, which is called a *configuration* of the CA at time t. A configuration intuitively means the spatial pattern that the CA display at that time. These definitions are illustrated in Fig. 11.1.

Neighborhood N is usually set up so that it is centered around the focal cell being updated ($x_0 = 0$) and spatially localized ($|x_i - x_0| \leq r$ for $i = 1, 2, \ldots, n - 1$), where r is called a *radius* of N. In other words, a cell's next state is determined locally according to its own current state and its local neighbors' current states. A specific arrangement of states within the neighborhood is called a *situation* here. Figure 11.2 shows typical examples of neighborhoods often used for two-dimensional CA. In CA with von Neumann neighborhoods (Fig. 11.2, left), each cell changes its state according to the states of its upper, lower, right, and left neighbor cells as well as itself. With Moore neighborhoods (Fig. 11.2, right), four diagonal cells are added to the neighborhood.

The state-transition function is applied uniformly and simultaneously to all cells in the space. The function can be described in the form of a look-up table (as shown in Fig. 11.1), some mathematical formula, or a more high-level algorithmic language.

If a state-transition function always gives an identical state to all the situations that are identical to each other when rotated, then the CA model has *rotational symmetry*. Such symmetry is often employed in CA with an aim to model physical phenomena. Rotational symmetry is called *strong* if all the states of the CA are orientation-free and if the rotation of a situation doesn't involve any rotation of the states themselves (Fig. 11.3, left). Otherwise, it is called *weak* (Fig. 11.3, right). In CA with weak rotational symmetry, some states are oriented, and the rotation of a situation requires rotational adjustments of those states as well. Von Neumann's original CA model adopted weak rotational symmetry.

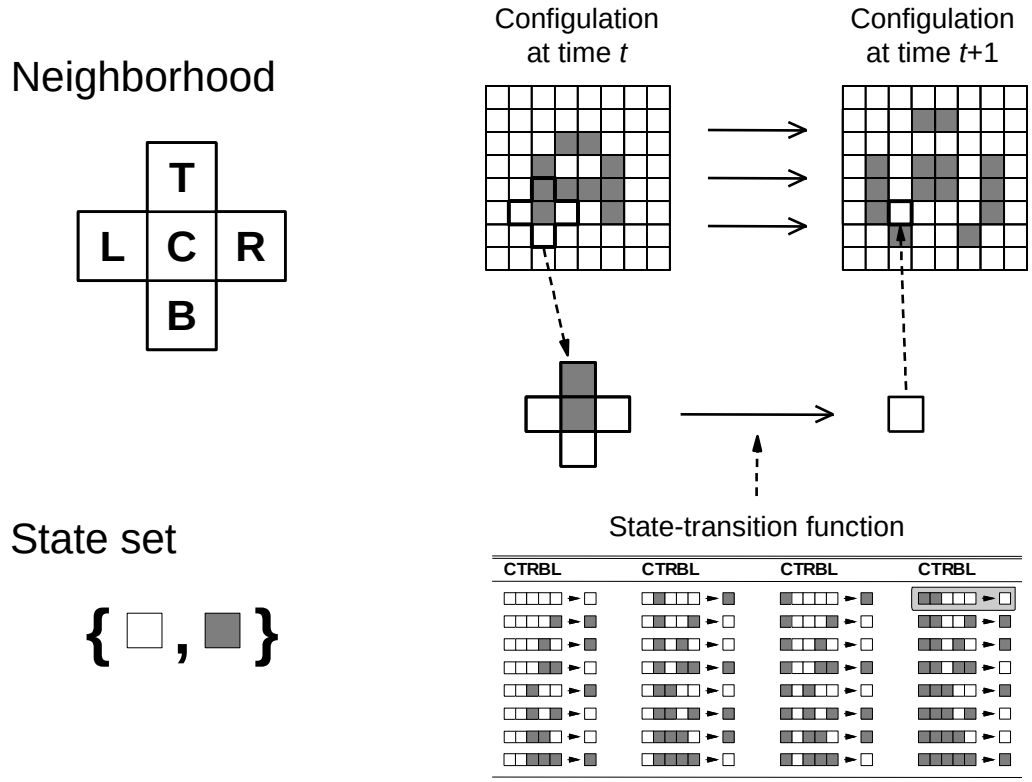


Figure 11.1: Schematic illustration of how cellular automata work.

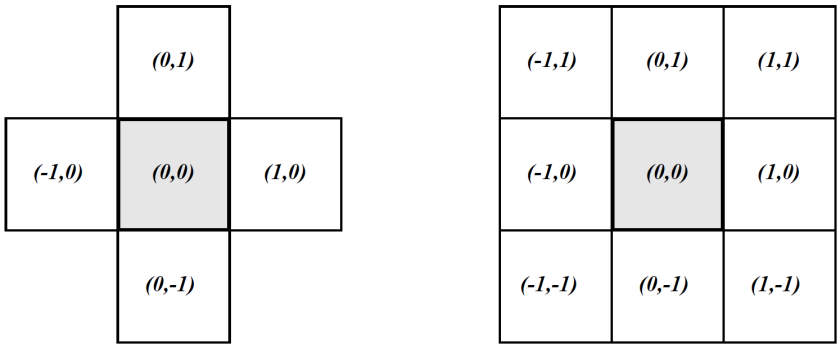


Figure 11.2: Examples of neighborhoods often used for two-dimensional CA. Left: von Neumann neighborhood. Right: Moore neighborhood.

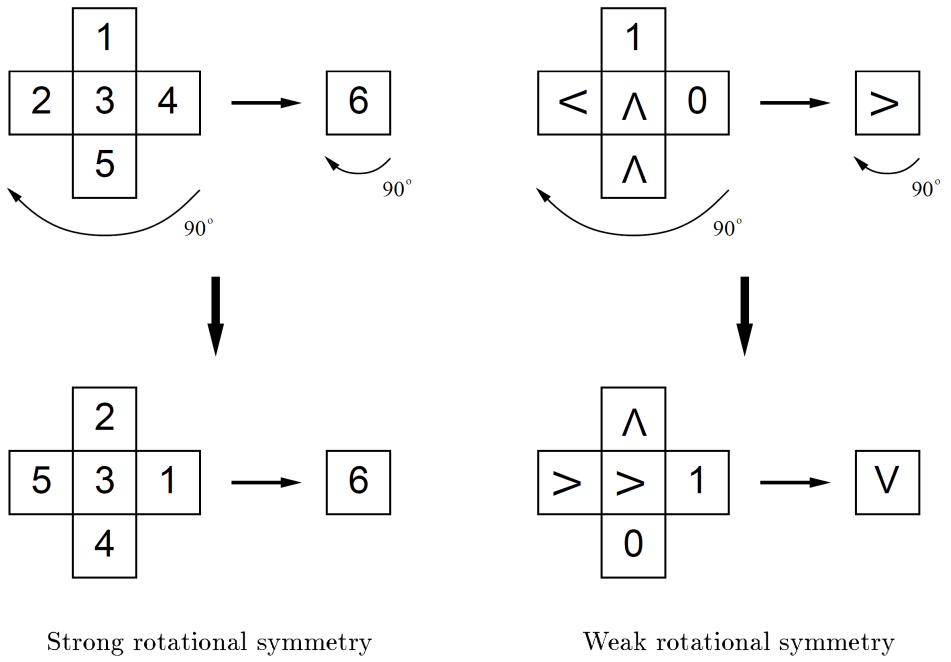


Figure 11.3: Schematic illustration of rotational symmetry in two-dimensional CA with von Neumann neighborhoods.

If a state-transition function depends only on the sum of the states of cells within the neighborhood, the CA is called *totalistic*. By definition, such state-transition functions are rotationally symmetric. The totalistic assumption makes the design of CA models much simpler, yet they can still produce a wide variety of complex dynamics [34].

The states of CA are usually categorized as either *quiescent* or *non-quiescent*. A cell in a quiescent state remains in the same state if all of its neighbors are also in the same quiescent state. Many CA models have at least one such quiescent state, often represented by either "0" or " " (blank). This state symbolizes a "vacuum" in the CA universe. Non-quiescent states are also called *active* states, because they can change dynamically and interact with nearby states. Such active states usually play a primary role in producing complex behaviors within CA.

Because CA models have a spatial extension as well as a temporal extension, you need to specify *spatial boundary conditions* in addition to initial conditions in order to study their behaviors. There are several commonly adopted boundary conditions:

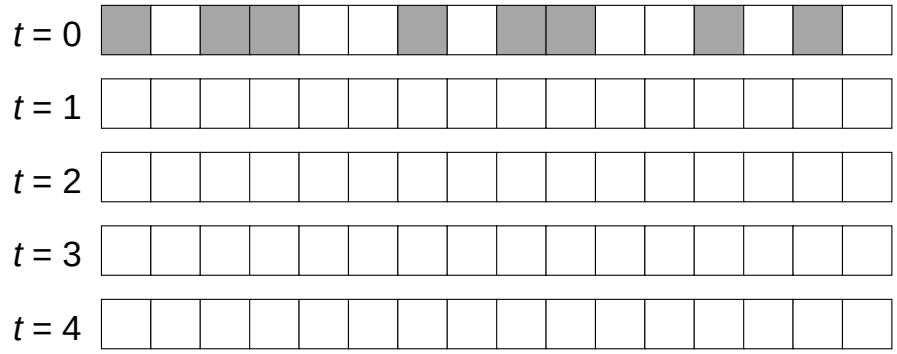
No boundaries This assumes that the space is infinite and completely filled with the quiescent state.

Periodic boundaries This assumes that the space is "wrapped around" each spatial axis, i.e., the cells at one edge of a finite space are connected to the cells at the opposite edge. Examples include a ring for 1-D CA and a torus for 2-D CA.

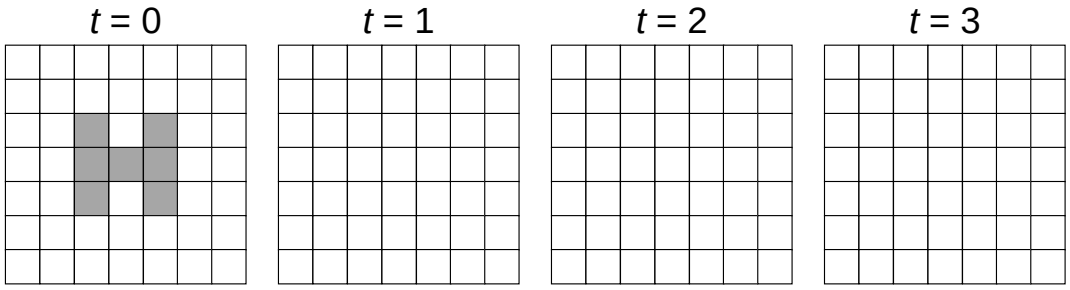
Cut-off boundaries This assumes that cells at the edge of a finite space don't have neighbors beyond the boundaries. This necessarily results in fewer numbers of neighbors, and therefore, the state-transition function must be designed so that it can handle such cases.

Fixed boundaries This assumes that cells at the edge of a finite space have fixed states that will never change. This is often the easiest choice when you implement a simulation code of a CA model.

Exercise 11.1 Shown below is an example of a one-dimensional totalistic CA model with radius 1 and a periodic boundary condition. White means 0, while gray means 1. Each cell switches to $\mathrm{round}(S/3)$ in every time step, where S is the local sum of states within its neighborhood. Complete the time evolution of this CA.



Exercise 11.2 Shown below is an example of a two-dimensional totalistic CA model with von Neumann neighborhoods and with no boundary (infinite space) conditions. White means 0 (= quiescent state), while gray means 1. Each cell switches to $\mathrm{round}(S/5)$ in every time step, where S is the local sum of the states within its neighborhood. Complete the time evolution of this CA.



11.2 Examples of Simple Binary Cellular Automata Rules

Majority rule The two exercises in the previous section were actually examples of CA with a state-transition function called the *majority rule* (a.k.a. *voting rule*). In this rule, each cell switches its state to a local majority choice within its neighborhood. This rule is so simple that it can be easily generalized to various settings, such as multi-dimensional space, multiple states, larger neighborhood size, etc. Note that all states are quiescent states in this CA. It is known that this CA model self-organizes into geographically separated patches made of distinct states, depending on initial conditions. Figure 11.4 shows an example of a majority rule-based 2-D CA with binary states (black and white), each of which is present at equal probability in the initial condition.

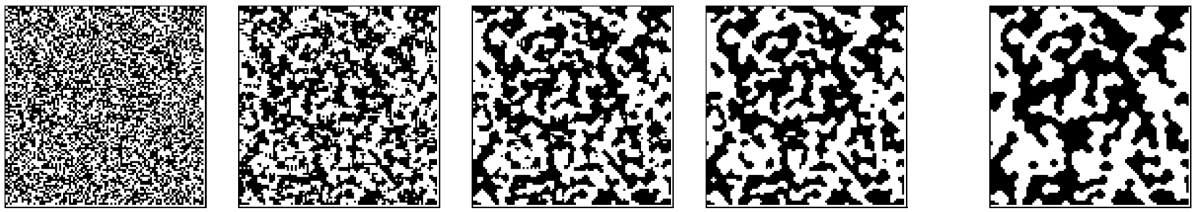


Figure 11.4: Typical behavior of binary CA with a Moore neighborhood governed by the majority rule.

Parity rule The *parity rule,* also called the *XOR (exclusive OR) rule,* is another well-known state-transition function for CA. Its state-transition function is defined mathematically as

$$s_{t+1}(x) = \sum_{i=0}^{n-1} s_t(x + x_i) \pmod{k}, \tag{11.2}$$

where k is the number of states ($k = 2$ for binary CA). It is known that, in this CA, any arbitrary pattern embedded in the initial configuration replicates itself and propagates over the space indefinitely. Figure 11.5 shows examples of such behaviors. In fact, this self-replicative feature is universal for all CA with parity rules, regardless of the numbers of states (k) or the neighborhood radius (r). This is because, under this rule, the growth pattern created from a single active cell (Fig. 11.5, top row) doesn't interact with other growth patterns that are created from other active cells (i.e., they are "independent" from each other). Any future pattern from any initial configuration can be predicted by a simple superposition of such growth patterns.

Game of Life The final example is the most popular 2-D binary CA, named the *"Game of Life,"* created by mathematician John Conway. Since its popularization by Martin Gardner in *Scientific American* in the early 1970s [35, 36], the Game of Life has attracted a lot of interest from researchers and mathematics/computer hobbyists all over the world, because of its fascinating, highly nontrivial behaviors. Its state-transition function uses the Moore neighborhood and is inspired by the birth and death of living organisms and their dependency on population density, as follows:

- A dead (quiescent) cell will turn into a living (active) cell if and only if it is surrounded by exactly three living cells.

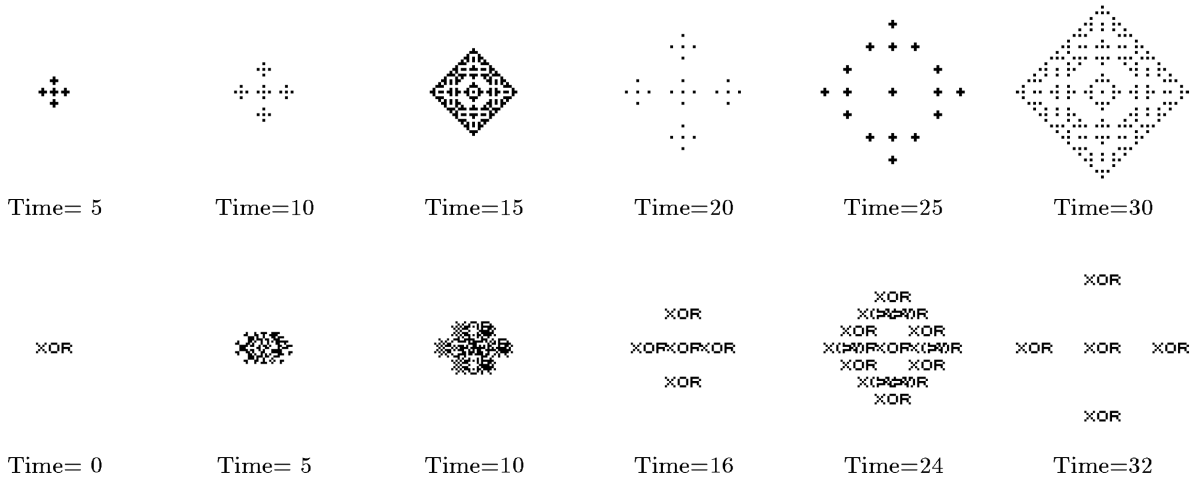


Figure 11.5: Typical behaviors of binary CA with a von Neumann neighborhood governed by the parity (XOR) rule.

- A living cell will remain alive if and only if it is surrounded by two or three other living cells. Otherwise it will die.

The Game of Life shows quite dynamic, almost life-like behaviors (Fig. 11.6). Many intriguing characteristics have been discovered about this game, including its statistical properties, computational universality, the possibility of the emergence of self-replicative creatures within it, and so on. It is often considered one of the historical roots of *Artificial Life*[1], an interdisciplinary research area that aims to synthesize living systems using nonliving materials. The artificial life community emerged in the 1980s and grew together with the complex systems community, and thus these two communities are closely related to each other. Cellular automata have been a popular modeling framework used by artificial life researchers to model self-replicative and evolutionary dynamics of artificial organisms [37, 38, 39, 40].

11.3 Simulating Cellular Automata

Despite their capability to represent various complex nonlinear phenomena, CA are relatively easy to implement and simulate because of their discreteness and homogeneity.

[1] `http://alife.org/`

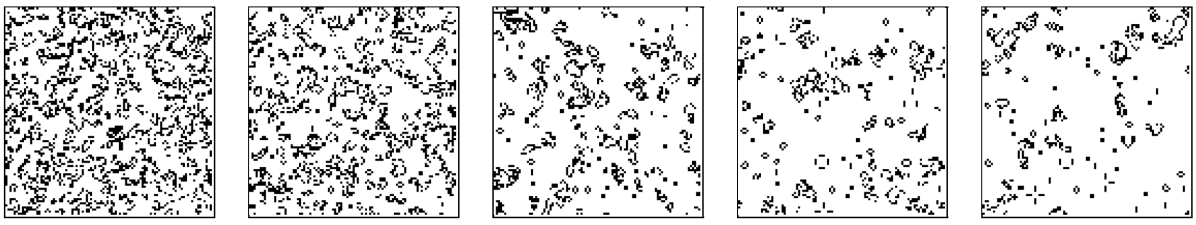


Figure 11.6: Typical behavior of the most well-known binary CA, the Game of Life.

There are existing software tools[2] and online interactive demonstrations[3] already available for cellular automata simulation, but it is nonetheless helpful to learn how to develop a CA simulator by yourself. Let's do so in Python, by working through the following example step by step.

The CA model we plan to implement here is a binary CA model with the *droplet rule* [4]. Its state-transition function can be understood as a model of panic propagation among individuals sitting in a gym after a fire alarm goes off. Here is the rule (which uses the Moore neighborhoods):

- A normal individual will get panicky if he or she is surrounded by four or more panicky individuals.
- A panicky individual will remain panicky if he or she is surrounded by three or more panicky individuals. Otherwise he or she will revert back to normal.

Note that this rule can be further simplified to the following single rule:

- If there are four or more panicky individuals within the neighborhood, the central cell will become panicky; otherwise it will become normal.

Here are other model assumptions:

- Space: 2-D, $n \times n$ ($n = 100$ for the time being)
- Boundary condition: periodic

[2]Most notable is Golly (`http://golly.sourceforge.net/`).

[3]For example, check out Wolfram Demonstrations Project (`http://demonstrations.wolfram.com/`) and Shodor.org's interactive activities (`http://www.shodor.org/interactivate/activities/`).

- Initial condition: Random (panicky individuals with probability p; normal ones with probability $1 - p$)

We will use pycxsimulator.py for dynamic CA simulations. Just as before, we need to design and implement the three essential components—initialization, observation, and updating.

To implement the initialization part, we have to decide how we are going to represent the states of the system. Since the configuration of this CA model is a regular two-dimensional grid, it is natural to use Python's `array` data structure. We can initialize it with randomly assigned states with probability p. Moreover, we should actually prepare two such arrays, one for the current time step and the other for the next time step, in order to avoid any unwanted conflict during the state updating process. So here is an example of how to implement the initialization part:

Code 11.1:

```
n = 100 # size of space: n x n
p = 0.1 # probability of initially panicky individuals

def initialize():
    global config, nextconfig
    config = zeros([n, n])
    for x in xrange(n):
        for y in xrange(n):
            config[x, y] = 1 if random() < p else 0
    nextconfig = zeros([n, n])
```

Here, `zeros([a, b])` is a function that generates an all-zero array with `a` rows and `b` columns. While `config` is initialized with randomly assigned states (1 for panicky individuals, 0 for normal), `nextconfig` is not, because it will be populated with the next states at the time of state updating.

The next thing is the observation. Fortunately, pylab has a built-in function `imshow` that is perfect for our purpose to visualize the content of an array:

Code 11.2:

```
def observe():
    global config, nextconfig
    cla()
    imshow(config, vmin = 0, vmax = 1, cmap = cm.binary)
```

Note that the `cmap` option in `imshow` is to specify the color scheme used in the plot. Without it, pylab uses dark blue and red for binary values by default, which are rather hard to see.

The updating part requires some algorithmic thinking. The state-transition function needs to count the number of panicky individuals within a neighborhood. How can we do this? The positions of the neighbor cells within the Moore neighborhood around a position (x, y) can be written as

$$\{(x', y') \mid x - 1 \leq x' \leq x + 1, y - 1 \leq y' \leq y + 1\}. \tag{11.3}$$

This suggests that the neighbor cells can be swept through using nested `for` loops for relative coordinate variables, say, dx and dy, each ranging from -1 to $+1$. Counting panicky individuals (1's) using this idea can be implemented in Python as follows:

Code 11.3:

```
count = 0
for dx in [-1, 0, 1]:
    for dy in [-1, 0, 1]:
        count += config[(x + dx) % n, (y + dy) % n]
```

Here, `dx` and `dy` are relative coordinates around (x, y), each varying from -1 to $+1$. They are added to `x` and `y`, respectively, to look up the current state of the cell located at $(x + dx, y + dy)$ in `config`. The expression `(...) % n` means that the value inside the parentheses is contained inside the $[0, n - 1]$ range by the mod operator (%). This is a useful coding technique to implement periodic boundary conditions in a very simple manner.

The counting code given above needs to be applied to all the cells in the space, so it should be included in another set of nested loops for `x` and `y` to sweep over the entire space. For each spatial location, the counting will be executed, and the next state of the cell will be determined based on the result. Here is the completed code for the updating:

Code 11.4:

```
def update():
    global config, nextconfig
    for x in xrange(n):
        for y in xrange(n):
            count = 0
            for dx in [-1, 0, 1]:
                for dy in [-1, 0, 1]:
                    count += config[(x + dx) % n, (y + dy) % n]
```

```
            nextconfig[x, y] = 1 if count >= 4 else 0
    config, nextconfig = nextconfig, config
```

Note the swapping of `config` and `nextconfig` at the end. This is precisely the same technique that we did for the simulation of multi-variable dynamical systems in the earlier chapters.

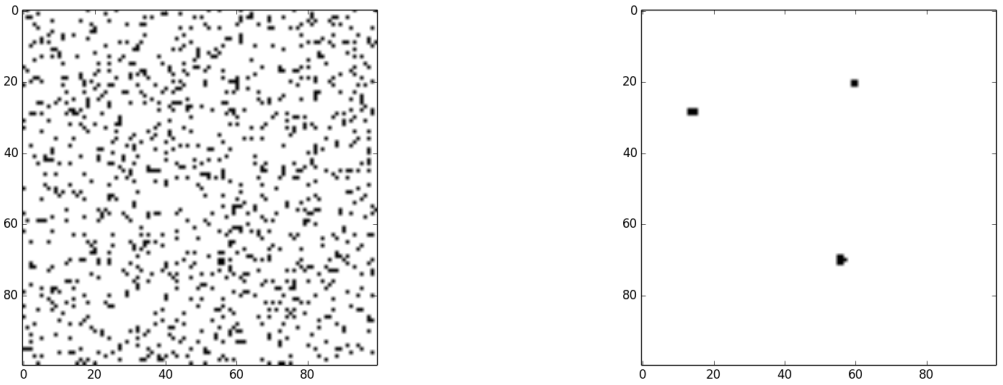
By putting all the above codes together, the completed simulator code in its entirety looks like this:

Code 11.5: panic-ca.py

```
import matplotlib
matplotlib.use('TkAgg')
from pylab import *

n = 100 # size of space: n x n
p = 0.1 # probability of initially panicky individuals

def initialize():
    global config, nextconfig
    config = zeros([n, n])
    for x in xrange(n):
        for y in xrange(n):
            config[x, y] = 1 if random() < p else 0
    nextconfig = zeros([n, n])

def observe():
    global config, nextconfig
    cla()
    imshow(config, vmin = 0, vmax = 1, cmap = cm.binary)

def update():
    global config, nextconfig
    for x in xrange(n):
        for y in xrange(n):
            count = 0
            for dx in [-1, 0, 1]:
                for dy in [-1, 0, 1]:
                    count += config[(x + dx) % n, (y + dy) % n]
```

```
            nextconfig[x, y] = 1 if count >= 4 else 0
    config, nextconfig = nextconfig, config

import pycxsimulator
pycxsimulator.GUI().start(func=[initialize, observe, update])
```

When you run this code, you see the results like those shown in Fig. 11.7. As you can see, with the initial configuration with $p = 0.1$, most of the panicky states disappear quickly, leaving only a few self-sustaining clusters of panicky people. They may look like condensed water droplets, hence the name of the model (droplet rule).



$t = 0$ $t = 20$

Figure 11.7: Visual output of Code 11.5. Left: Initial configuration with $p = 0.1$. Right: Final configuration after 20 time steps.

Exercise 11.3 Modify Code 11.5 to implement a simulator of the Game of Life CA. Simulate the dynamics from a random initial configuration. Measure the density of state 1's in the configuration at each time step, and plot how the density changes over time. This can be done by creating an empty list in the `initialize` function, and then making the measurement and appending the result to the list in the `observe` function. The results stored in the list can be plotted manually after the simulation, or they could be plotted next to the visualization using pylab's `subplot` function during the simulation.

Exercise 11.4 Modify Code 11.5 to implement a simulator of the majority rule CA in a two-dimensional space. Then answer the following questions by conducting simulations:

- What happens if you change the ratio of binary states in the initial condition?
- What happens if you increase the radius of the neighborhoods?
- What happens if you increase the number of states?

The second model revision in the previous exercise (increasing the radius of neighborhoods) for the majority rule CA produces quite interesting spatial dynamics. Specifically, the boundaries between the two states tend to straighten out, and the characteristic scale of spatial features continuously becomes larger and larger over time (Fig. 11.8). This behavior is called *coarsening* (to be more specific, non-conserved coarsening). It can be seen in many real-world contexts, such as geographical distributions of distinct cultural/political/linguistic states in human populations, or incompatible genetic types in animal/plant populations.

Figure 11.8: Coarsening behavior observed in a binary CA with the majority rule in a 100×100 2-D spatial grid with periodic boundary conditions. The radius of the neighborhoods was set to 5. Time flows from left to right.

What is most interesting about this coarsening behavior is that, once clear boundaries are established between domains of different states, the system's macroscopic behavior can be described using emergent properties of those boundaries, such as their surface tensions and direction of movement [41, 42]. The final fate of the system is determined *not* by the relative frequencies of the two states in the space, but by the topological features of the boundaries. For example, if a big "island" of white states is surrounded by a thin "ring" of black states, the latter minority will eventually dominate, no matter how small its initial proportion is (even though this is still driven by the majority rule!). This is an illustrative

example that shows how important it is to consider emergent macroscopic properties of complex systems, and how counter-intuitive their behaviors can be sometimes.

Here is one more interesting fact about CA dynamics. The droplet/panic model discussed above has an interesting property: When you increase the initial density of panicky people (e.g., $p = 0.3$), the result changes dramatically. As seen in Fig. 11.9, the initially formed clusters tend to attach to each other, which makes their growth unstoppable. The whole space will eventually be filled up with all panicky people, which could be a disaster if this was a real situation.

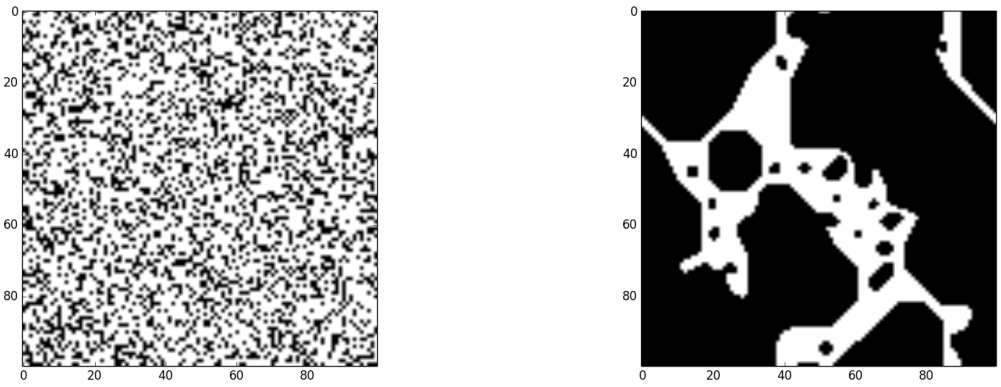


Figure 11.9: Another visual output of Code 11.5. Left: Initial configuration with $p = 0.3$. Right: Configuration after 40 time steps.

You can explore the value of p to find out that the transition between these two distinct behaviors takes place at a rather narrow range of p. This is an example of a *phase transition*, which is defined informally as follows:

A *phase transition* is a transition of macroscopic properties of a collective system that occurs when its environmental or internal conditions are varied.

A familiar example of phase transitions is the transition between different phases of matter, i.e., solid, liquid, and gas, which occur when temperature and/or pressure are varied. Phase transitions can be characterized by measuring what physicists call *order parame-*

ters[4] that represent how ordered a macroscopic state of the system is. A phase transition can be understood as a bifurcation observed in the macroscopic properties (i.e., order parameters) of a collective system.

> *Exercise 11.5* Implement an interactive parameter setter for p in Code 11.5. Then conduct systematic simulations with varying p, and identify its critical value below which isolated clusters are formed but above which the whole space is filled with panic.

11.4 Extensions of Cellular Automata

So far, we discussed CA models in their most conventional settings. But there are several ways to "break" the modeling conventions, which could make CA more useful and applicable to real-world phenomena. Here are some examples.

Stochastic cellular automata A state-transition function of CA doesn't have to be a rigorous mathematical function. It can be a computational process that produces the output probabilistically. CA with such probabilistic state-transition rules are called *stochastic CA*, which play an important role in mathematical modeling of various biological, social, and physical phenomena. A good example is a CA model of epidemiological processes where infection of a disease takes place stochastically (this will be discussed more in the following section).

Multi-layer cellular automata States of cells don't have to be scalar. Instead, each spatial location can be associated with several variables (i.e., vectors). Such vector-valued configurations can be considered a superposition of multiple layers, each having a conventional scalar-valued CA model. Multi-layer CA models are useful when multiple biological or chemical species are interacting with each other in a space-time. This is particularly related to reaction-diffusion systems that will be discussed in later chapters.

Asynchronous cellular automata Synchronous updating is a signature of CA models, but we can even break this convention to make the dynamics asynchronous. There are

[4] Note that the word "parameter" in this context means an outcome of a measurement, and not a condition or input as in "model parameters."

several asynchronous updating mechanisms possible, such as random updating (a randomly selected cell is updated at each time step), sequential updating (cells are updated in a predetermined sequential order), state-triggered updating (certain states trigger updating of nearby cells), etc. It is often argued that synchronous updating in conventional CA models is too artificial and fragile against slight perturbations in updating orders, and in this sense, the behaviors of asynchronous CA models are deemed more robust and applicable to real-world problems. Moreover, there is a procedure to create asynchronous CA that can robustly emulate the behavior of any synchronous CA [43].

11.5 Examples of Biological Cellular Automata Models

In this final section, I provide more examples of cellular automata models, with a particular emphasis on biological systems. Nearly all biological phenomena involve some kind of spatial extension, such as excitation patterns on neural or muscular tissue, cellular arrangements in an individual organism's body, and population distribution at ecological levels. If a system has a spatial extension, nonlinear local interactions among its components may cause *spontaneous pattern formation*, i.e., self-organization of static or dynamic spatial patterns from initially uniform conditions. Such self-organizing dynamics are quite counter-intuitive, yet they play essential roles in the structure and function of biological systems.

In each of the following examples, I provide basic ideas of the state-transition rules and what kind of patterns can arise if some conditions are met. I assume Moore neighborhoods in these examples (unless noted otherwise), but the shape of the neighborhoods is not so critically important. Completed Python simulator codes are available from `http://sourceforge.net/projects/pycx/files/`, but you should try implementing your own simulator codes first.

Turing patterns Animal skin patterns are a beautiful example of pattern formation in biological systems. To provide a theoretical basis of this intriguing phenomenon, British mathematician Alan Turing (who is best known for his fundamental work in theoretical computer science and for his code-breaking work during World War II) developed a family of models of spatio-temporal dynamics of chemical reaction and diffusion processes [44]. His original model was first written in a set of coupled ordinary differential equations on compartmentalized cellular structures, and then it was extended to *partial differential equations* (PDEs) in a continuous space. Later, a much simpler CA version of the same model was proposed by David Young [45]. We will discuss Young's simpler model here.

Assume a two-dimensional space made of cells where each cell can take either a passive (0) or active (1) state. A cell becomes activated if there are a sufficient number of active cells within its local neighborhood. However, other active cells outside this neighborhood try to suppress the activation of the focal cell with relatively weaker influences than those from the active cells in its close vicinity. These dynamics are called *short-range activation and long-range inhibition.* This model can be described mathematically as follows:

$$N_a = \left\{ x' \;\middle|\; |x'| \leq R_a \right\} \tag{11.4}$$

$$N_i = \left\{ x' \;\middle|\; |x'| \leq R_i \right\} \tag{11.5}$$

$$a_t(x) = w_a \sum_{x' \in N_a} s_t(x + x') - w_i \sum_{x' \in N_i} s_t(x + x') \tag{11.6}$$

$$s_{t+1}(x) = \begin{cases} 1 & \text{if } a_t(x) > 0, \\ 0 & \text{otherwise.} \end{cases} \tag{11.7}$$

Here, R_a and R_i are the radii of neighborhoods for activation (N_a) and inhibition (N_i), respectively ($R_a < R_i$), and w_a and w_i are the weights that represent their relative strengths. $a_t(x)$ is the result of two neighborhood countings, which tells you whether the short-range activation wins ($a_t(x) > 0$) or the long-range inhibition wins ($a_t(x) \leq 0$) at location x. Figure 11.10 shows the schematic illustration of this state-transition function, as well as a sample simulation result you can get if you implement this model successfully.

Exercise 11.6 Implement a CA model of the Turing pattern formation in Python. Then try the following:

- What happens if R_a and R_i are varied?
- What happens if w_a and w_i are varied?

Waves in excitable media Neural and muscle tissues made of animal cells can generate and propagate electrophysiological signals. These cells can get excited in response to external stimuli coming from nearby cells, and they can generate *action potential* across their cell membranes that will be transmitted as a stimulus to other nearby cells. Once excited, the cell goes through a *refractory period* during which it doesn't respond to any further stimuli. This causes the directionality of signal propagation and the formation of "traveling waves" on tissues.

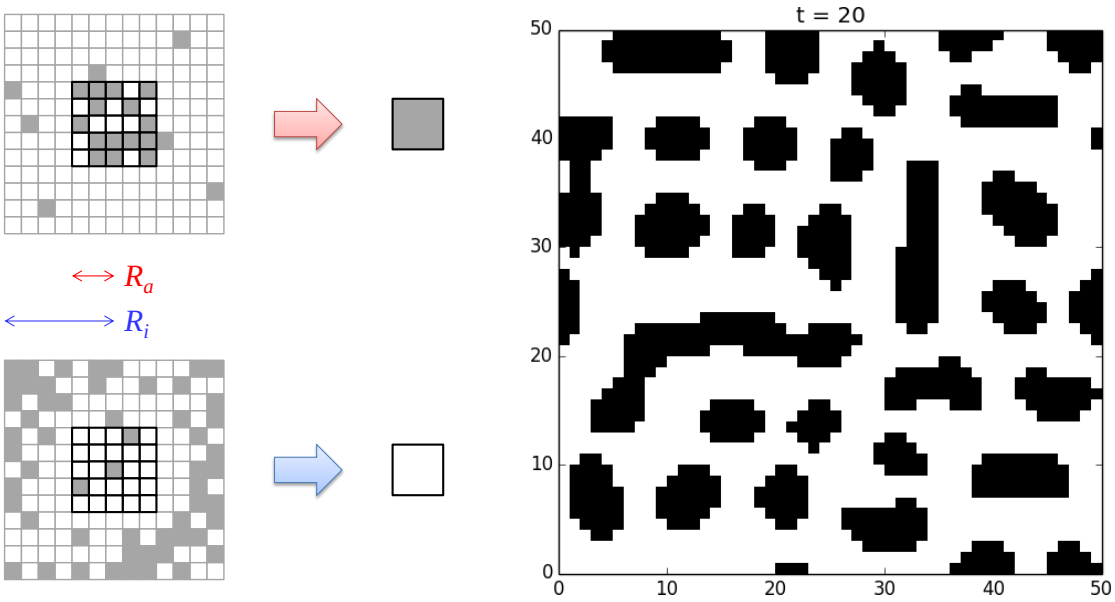


Figure 11.10: CA model of Turing patterns. Left: Schematic illustration of the model's state-transition function. Right: A sample simulation result.

This kind of spatial dynamics, driven by propagation of states between adjacent cells that are physically touching each other, are called *contact processes.* This model and the following two are all examples of contact processes.

A stylized CA model of this excitable media can be developed as follows. Assume a two-dimensional space made of cells where each cell takes either a normal (0; quiescent), excited (1), or refractory (2, 3, $\ldots k$) state. A normal cell becomes excited stochastically with a probability determined by a function of the number of excited cells in its neighborhood. An excited cell becomes refractory (2) immediately, while a refractory cell remains refractory for a while (but its state keeps counting up) and then it comes back to normal after it reaches k. Figure 11.11 shows the schematic illustration of this state-transition function, as well as a sample simulation result you can get if you implement this model successfully. These kinds of uncoordinated traveling waves of excitation (including spirals) are actually experimentally observable on the surface of a human heart under cardiac arrhythmia.

Exercise 11.7 Implement a CA model of excitable media in Python. Then try the following:

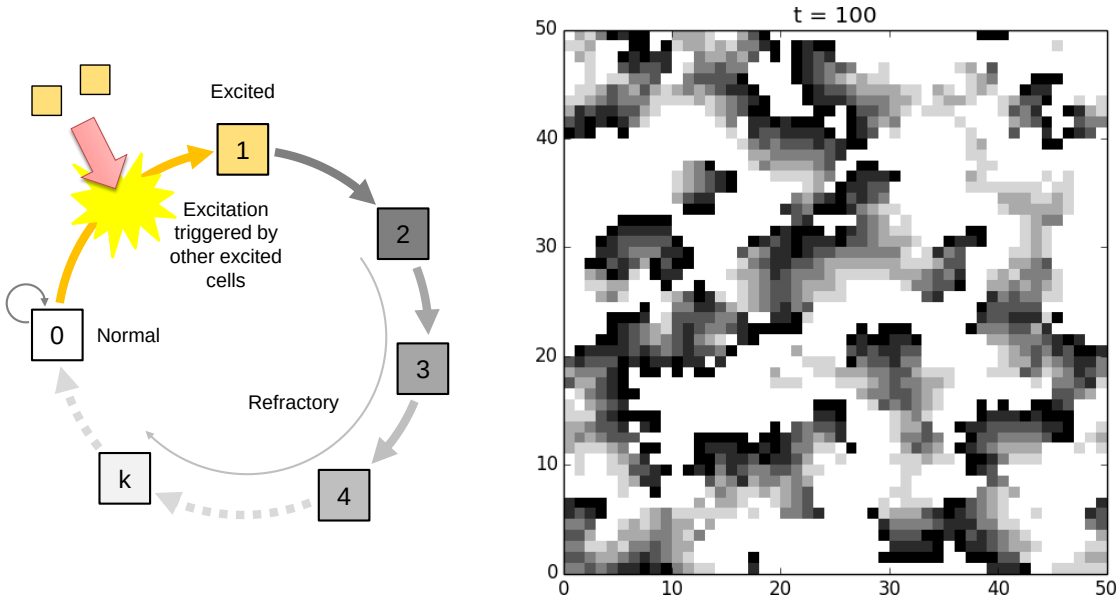


Figure 11.11: CA model of excitable media. Left: Schematic illustration of the model's state-transition function. Right: A sample simulation result.

- What happens if the excitation probability is changed?
- What happens if the length of the refractory period is changed?

Host-pathogen model A spatially extended host-pathogen model, studied in theoretical biology and epidemiology, is a nice example to demonstrate the subtle relationship between these two antagonistic players. This model can also be viewed as a spatially extended version of the Lotka-Volterra (predator-prey) model, where each cell at a particular location represents either an empty space or an individual organism, instead of population densities that the variables in the Lotka-Volterra model represented.

Assume a two-dimensional space filled with empty sites (0; quiescent) in which a small number of host organisms are initially populated. Some of them are "infected" by pathogens. A healthy host (1; also quiescent) without pathogens will grow into nearby empty sites stochastically. A healthy host may get infected by pathogens with a probability determined by a function of the number of infected hosts (2) in its neighborhood. An infected host will die immediately (or after some period of time). Figure 11.12 shows the schematic illustration of this state-transition function, as well as a sample simulation result

you could get if you implement this model successfully. Note that this model is similar to the excitable media model discussed above, with the difference that the healthy host state (which corresponds to the excited state in the previous example) is quiescent and thus can remain in the space indefinitely.

The spatial patterns formed in this model are quite different from those in the previously discussed models. Turing patterns and waves in excitable media have clear characteristic length scales (i.e., size of spots or width of waves), but the dynamic patterns formed in the host-pathogen model lacks such characteristic length scales. You will see a number of dynamically forming patches of various sizes, from tiny ones to very large continents.

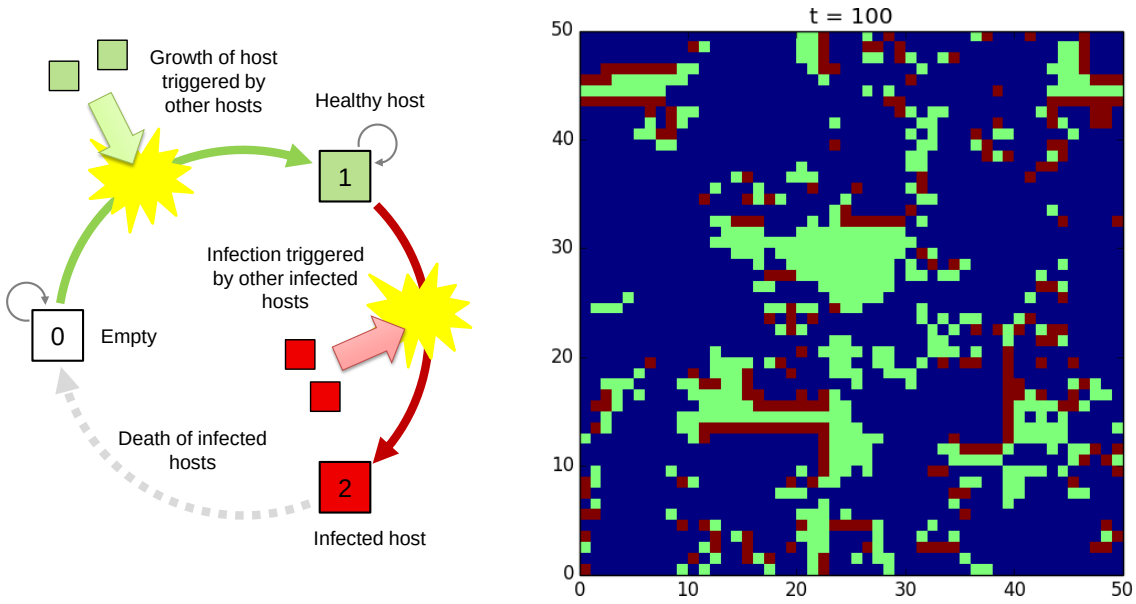


Figure 11.12: CA model of host-pathogen systems. Left: Schematic illustration of the model's state-transition function. Right: A sample simulation result.

Exercise 11.8 Implement a CA model of host-pathogen systems in Python. Then try the following:

- What happens if the infection probability is changed?
- In what conditions will both hosts and pathogens co-exist? In what conditions can hosts exterminate pathogens? In what conditions will both of them become extinct?

- Plot the populations of healthy/infected hosts over time, and see if there are any temporal patterns you can observe.

Epidemic/forest fire model The final example, the epidemic model, is also about contact processes similar to the previous two examples. One difference is that this model focuses more on static spatial distributions of organisms and their influence on the propagation of an infectious disease within a single epidemic event. This model is also known as the "forest fire" model, so let's use this analogy.

Assume there is a square-shaped geographical area, represented as a CA space, in which trees (1) are distributed with some given probability, p. That is, $p = 0$ means there are no trees in the space, whereas $p = 1$ means trees are everywhere with no open space left in the area. Then, you set fire (2) to one of the trees in this forest to see if the fire you started eventually destroys the entire forest (don't do this in real life!!). A tree will catch fire if there is at least one tree burning in its neighborhood, and the burning tree will be charred (3) completely after one time step.

Figure 11.13 shows the schematic illustration of this state-transition function, as well as a sample simulation result you could get if you implement this model successfully. Note that this model doesn't have cyclic local dynamics; possible state transitions are always one way from a tree (1) to a burning tree (2) to being charred (3), which is different from the previous two examples. So the whole system eventually falls into a static final configuration with no further changes possible. But the total area burned in the final configuration greatly depends on the density of trees p. If you start with a sufficiently large value of p, you will see that a significant portion of the forest will be burned down eventually. This phenomenon is called *percolation* in statistical physics, which intuitively means that something found a way to go through a large portion of material from one side to the other.

Exercise 11.9 Implement a CA model of epidemic/forest fire systems in Python. Then try the following:

- Compare the final results for different values of p.
- Plot the total burned area as a function of p.
- Plot the time until the fire stops spreading as a function of p.

When you work on the above tasks, make sure to carry out multiple independent simulation runs for each value of p and average the results to obtain statistically

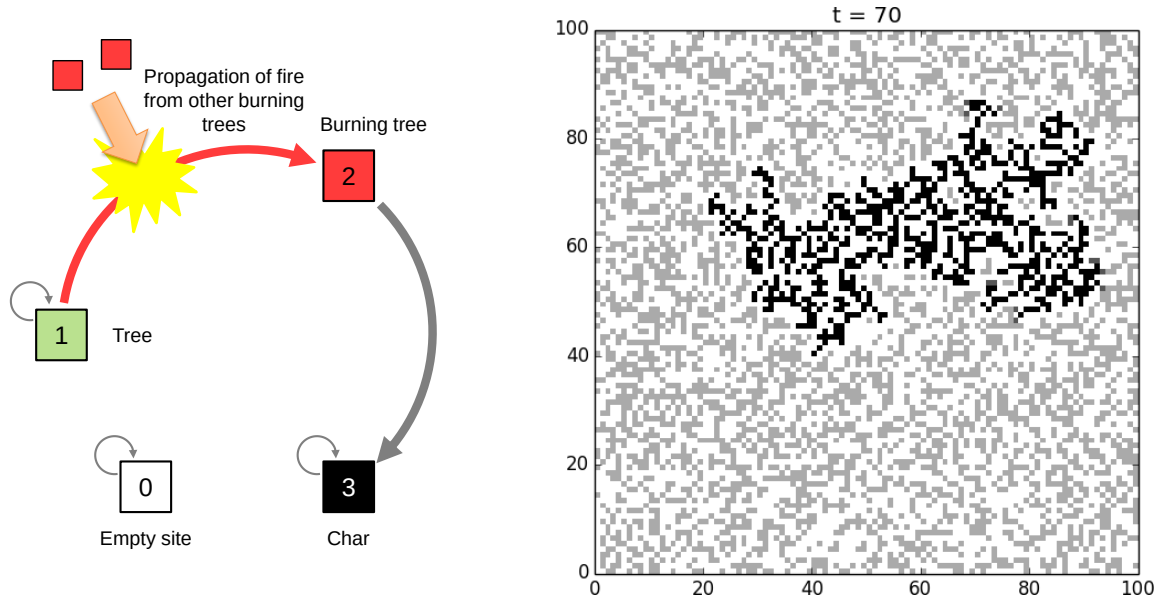


Figure 11.13: CA model of epidemic/forest fire systems. Left: Schematic illustration of the model's state-transition function. Right: A sample simulation result.

more reliable results. Such simulation methods based on random sampling are generally called *Monte Carlo simulations*. In Monte Carlo simulations, you conduct many replications of independent simulation runs for the same experimental settings, and measure the outcome variables from each run to obtain a statistical distribution of the measurements. Using this distribution (e.g., by calculating its average value) will help you enhance the accuracy and reliability of the experimental results.

If you have done the implementation and experiments correctly, you will probably see another case of phase transition in the exercise above. The system's response shows a sharp transition at a critical value of p, above which percolation occurs but below which it doesn't occur. Near this critical value, the system is very sensitive to minor perturbations, and a number of intriguing phenomena (such as the formation of self-similar *fractal* patterns) are found to take place at or near this transition point, which are called *critical behaviors.* Many complex systems, including biological and social ones, are considered to be utilizing such critical behaviors for their self-organizing and information processing purposes. For example, there is a conjecture that animal brains tend to dynamically maintain

critical states in their neural dynamics in order to maximize their information processing capabilities. Such *self-organized criticality* in natural systems has been a fundamental research topic in complex systems science.