Think Complexity

Version 2.6.3

www.dbooks.org

Think Complexity

Version 2.6.3

Allen B. Downey

Green Tea Press

Needham, Massachusetts

www.dbooks.org

Copyright © 2016 Allen B. Downey.

Green Tea Press
9 Washburn Ave
Needham MA 02492

Permission is granted to copy, distribute, transmit and adapt this work under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License: http://thinkcomplex.com/license.

If you are interested in distributing a commercial version of this work, please contact the author.

The LaTeX source for this book is available from

```
https://github.com/AllenDowney/ThinkComplexity2
```

www.dbooks.org

Contents

Preface **xi**

0.1 Who is this book for? xii

0.2 Changes from the first edition xiii

0.3 Using the code xiii

1 Complexity Science **1**

1.1 The changing criteria of science 3

1.2 The axes of scientific models 4

1.3 Different models for different purposes 6

1.4 Complexity engineering 7

1.5 Complexity thinking 8

2 Graphs **11**

2.1 What is a graph? 11

2.2 NetworkX 13

2.3 Random graphs 16

2.4 Generating graphs 17

2.5 Connected graphs 18

2.6 Generating ER graphs 20

2.7 Probability of connectivity 22

2.8 Analysis of graph algorithms 24
2.9 Exercises 25

3 Small World Graphs **27**

3.1 Stanley Milgram 27
3.2 Watts and Strogatz 28
3.3 Ring lattice 30
3.4 WS graphs 32
3.5 Clustering 33
3.6 Shortest path lengths 35
3.7 The WS experiment 36
3.8 What kind of explanation is *that*? 38
3.9 Breadth-First Search 39
3.10 Dijkstra's algorithm 41
3.11 Exercises 43

4 Scale-free networks **47**

4.1 Social network data 47
4.2 WS Model 50
4.3 Degree 51
4.4 Heavy-tailed distributions 53
4.5 Barabási-Albert model 55
4.6 Generating BA graphs 57
4.7 Cumulative distributions 59
4.8 Explanatory models 62
4.9 Exercises 63

www.dbooks.org

5 Cellular Automatons 67

5.1 A simple CA 67
5.2 Wolfram's experiment 68
5.3 Classifying CAs 69
5.4 Randomness 71
5.5 Determinism 72
5.6 Spaceships 73
5.7 Universality 76
5.8 Falsifiability 77
5.9 What is this a model of? 78
5.10 Implementing CAs 80
5.11 Cross-correlation 82
5.12 CA tables 84
5.13 Exercises 85

6 Game of Life 89

6.1 Conway's GoL 89
6.2 Life patterns 92
6.3 Conway's conjecture 92
6.4 Realism 94
6.5 Instrumentalism 95
6.6 Implementing Life 97
6.7 Exercises 99

7 Physical modeling 103

7.1 Diffusion 103
7.2 Reaction-diffusion 105

7.3 Percolation . . . 109
7.4 Phase change . . . 110
7.5 Fractals . . . 113
7.6 Fractals and Percolation Models . . . 115
7.7 Exercises . . . 116

8 Self-organized criticality 119

8.1 Critical Systems . . . 119
8.2 Sand Piles . . . 120
8.3 Implementing the Sand Pile . . . 121
8.4 Heavy-tailed distributions . . . 125
8.5 Fractals . . . 127
8.6 Pink noise . . . 131
8.7 The sound of sand . . . 132
8.8 Reductionism and Holism . . . 134
8.9 SOC, causation, and prediction . . . 137
8.10 Exercises . . . 138

9 Agent-based models 141

9.1 Schelling's Model . . . 142
9.2 Implementation of Schelling's model . . . 143
9.3 Segregation . . . 145
9.4 Sugarscape . . . 147
9.5 Wealth inequality . . . 150
9.6 Implementing Sugarscape . . . 151
9.7 Migration and Wave Behavior . . . 154
9.8 Emergence . . . 155

www.dbooks.org

9.9 Exercises 157

10 Herds, Flocks, and Traffic Jams 159

10.1 Traffic jams 159

10.2 Random perturbation 163

10.3 Boids 164

10.4 The Boid algorithm 165

10.5 Arbitration 168

10.6 Emergence and free will 169

10.7 Exercises 171

11 Evolution 173

11.1 Simulating evolution 174

11.2 Fitness landscape 175

11.3 Agents 176

11.4 Simulation 177

11.5 No differentiation 178

11.6 Evidence of evolution 179

11.7 Differential survival 182

11.8 Mutation 183

11.9 Speciation 186

11.10 Summary 189

11.11 Exercises 190

12 Evolution of cooperation 191

12.1 Prisoner's Dilemma 192

12.2 The problem of nice 193

12.3 Prisoner's dilemma tournaments 195
12.4 Simulating evolution of cooperation 196
12.5 The Tournament 198
12.6 The Simulation 200
12.7 Results 202
12.8 Conclusions 205
12.9 Exercises 207

A Reading list **209**

www.dbooks.org

Preface

Complexity science is an interdisciplinary field — at the intersection of mathematics, computer science and natural science — that focuses on **complex systems**, which are systems with many interacting components.

One of the core tools of complexity science is discrete models, including networks and graphs, cellular automatons, and agent-based simulations. These tools are useful in the natural and social sciences, and sometimes in arts and humanities.

For an overview of complexity science, see http://thinkcomplex.com/complex.

Why should you learn about complexity science? Here are a few reasons:

- Complexity science is useful, especially for explaining why natural and social systems behave the way they do. Since Newton, math-based physics has focused on systems with small numbers of components and simple interactions. These models are effective for some applications, like celestial mechanics, and less useful for others, like economics. Complexity science provides a diverse and adaptable modeling toolkit.

- Many of the central results of complexity science are surprising; a recurring theme of this book is that simple models can produce complicated behavior, with the corollary that we can sometimes explain complicated behavior in the real world using simple models.

- As I explain in Chapter 1, complexity science is at the center of a slow shift in the practice of science and a change in what we consider science to be.

- Studying complexity science provides an opportunity to learn about diverse physical and social systems, to develop and apply programming skills, and to think about fundamental questions in the philosophy of science.

By reading this book and working on the exercises you will have a chance to explore topics and ideas you might not encounter otherwise, practice programming in Python, and learn more about data structures and algorithms.

Features of this book include:

Technical details Most books about complexity science are written for a popular audience. They leave out technical details, which is frustrating for people who can handle them. This book presents the code, the math, and the explanations you need to understand how the models work.

Further reading Throughout the book, I include pointers to further reading, including original papers (most of which are available electronically) and related articles from Wikipedia and other sources.

Jupyter notebooks For each chapter I provide a Jupyter notebook that includes the code from the chapter, additional examples, and animations that let you see the models in action.

Exercises and solutions At the end of each chapter I suggest exercises you might want to work on, with solutions.

For most of the links in this book I use URL redirection. This mechanism has the drawback of hiding the link destination, but it makes the URLs shorter and less obtrusive. Also, and more importantly, it allows me to update the links without updating the book. If you find a broken link, please let me know and I will change the redirection.

0.1 Who is this book for?

The examples and supporting code for this book are in Python. You should know core Python and be familiar with its object-oriented features, specifically using and defining classes.

www.dbooks.org

If you are not already familiar with Python, you might want to start with *Think Python*, which is appropriate for people who have never programmed before. If you have programming experience in another language, there are many good Python books to choose from, as well as online resources.

I use NumPy, SciPy, and NetworkX throughout the book. If you are familiar with these libraries already, that's great, but I will also explain them when they appear.

I assume that the reader knows some mathematics: I use logarithms in several places, and vectors in one example. But that's about it.

0.2 Changes from the first edition

For the second edition, I added two chapters, one on evolution, the other on the evolution of cooperation.

In the first edition, each chapter presented background on a topic and suggested experiments the reader could run. For the second edition, I have done those experiments. Each chapter presents the implementation and results as a worked example, then suggests additional experiments for the reader.

For the second edition, I replaced some of my own code with standard libraries like NumPy and NetworkX. The result is more concise and more efficient, and it gives readers a chance to learn these libraries.

Also, the Jupyter notebooks are new. For every chapter there are two notebooks: one contains the code from the chapter, explanatory text, and exercises; the other contains solutions to the exercises.

Finally, all supporting software has been updated to Python 3 (but most of it runs unmodified in Python 2).

0.3 Using the code

All code used in this book is available from a Git repository on GitHub: `http://thinkcomplex.com/repo`. If you are not familiar with Git, it is a version

control system that allows you to keep track of the files that make up a project. A collection of files under Git's control is called a "repository". GitHub is a hosting service that provides storage for Git repositories and a convenient web interface.

The GitHub homepage for my repository provides several ways to work with the code:

- You can create a copy of my repository by pressing the Fork button in the upper right. If you don't already have a GitHub account, you'll need to create one. After forking, you'll have your own repository on GitHub that you can use to keep track of code you write while working on this book. Then you can clone the repo, which means that you copy the files to your computer.
- Or you can clone my repository without forking; that is, you can make a copy of my repo on your computer. You don't need a GitHub account to do this, but you won't be able to write your changes back to GitHub.
- If you don't want to use Git at all, you can download the files in a Zip file using the green button that says "Clone or download".

I developed this book using Anaconda from Continuum Analytics, which is a free Python distribution that includes all the packages you'll need to run the code (and lots more). I found Anaconda easy to install. By default it does a user-level installation, not system-level, so you don't need administrative privileges. And it supports both Python 2 and Python 3. You can download Anaconda from https://continuum.io/downloads.

The repository includes both Python scripts and Jupyter notebooks. If you have not used Jupyter before, you can read about it at https://jupyter.org.

There are three ways you can work with the Jupyter notebooks:

Run Jupyter on your computer If you installed Anaconda, you can install Jupyter by running the following command in a terminal or Command Window:

```
$ conda install jupyter
```

www.dbooks.org

Before you launch Jupyter, you should `cd` into the directory that contains the code:

```
$ cd ThinkComplexity2/code
```

And then start the Jupyter server:

```
$ jupyter notebook
```

When you start the server, it should launch your default web browser or create a new tab in an open browser window. Then you can open and run the notebooks.

Run Jupyter on Binder Binder is a service that runs Jupyter in a virtual machine. If you follow this link, http://thinkcomplex.com/binder, you should get a Jupyter home page with the notebooks for this book and the supporting data and scripts.

You can run the scripts and modify them to run your own code, but the virtual machine you run them in is temporary. If you leave it idle, the virtual machine disappears along with any changes you made.

View notebooks on GitHub GitHub provides a view of the notebooks you can can use to read the notebooks and see the results I generated, but you won't be able to modify or run the code.

Good luck, and have fun!

Allen B. Downey
Professor of Computer Science
Olin College of Engineering
Needham, MA

Contributor List

If you have a suggestion or correction, please send email to `downey@allendowney.com`. If I make a change based on your feedback, I will add you to the contributor list (unless you ask to be omitted).

Let me know what version of the book you are working with, and what format. If you include at least part of the sentence the error appears in, that makes it easy for me to search. Page and section numbers are fine, too, but not quite as easy to work with. Thanks!

- John Harley, Jeff Stanton, Colden Rouleau and Keerthik Omanakuttan are Computational Modeling students who pointed out typos.
- Jose Oscar Mur-Miranda found several typos.
- Phillip Loh, Corey Dolphin, Noam Rubin and Julian Ceipek found typos and made helpful suggestions.
- Sebastian Schöner sent two pages of corrections!
- Philipp Marek sent a number of corrections.
- Jason Woodard co-taught Complexity Science with me at Olin College, introduced me to NK models, and made many helpful suggestions and corrections.
- Davi Post sent several corrections and suggestions.
- Graham Taylor sent a pull request on GitHub that fixed many typos.

I would especially like to thank the technical reviewers, Vincent Knight and Eric Ma, who made many helpful suggestions, and the copy editor, Charles Roumeliotis, who caught many errors and inconsistencies.

Other people who reported errors include Richard Hollands, Muhammad Najmi bin Ahmad Zabidi, Alex Hantman, and Jonathan Harford.

www.dbooks.org

Chapter 1

Complexity Science

Complexity science is relatively new; it became recognizable as a field, and was given a name, in the 1980s. But its newness is not because it applies the tools of science to a new subject, but because it uses different tools, allows different kinds of work, and ultimately changes what we mean by "science".

To demonstrate the difference, I'll start with an example of classical science: suppose someone asks you why planetary orbits are elliptical. You might invoke Newton's law of universal gravitation and use it to write a differential equation that describes planetary motion. Then you can solve the differential equation and show that the solution is an ellipse. QED!

Most people find this kind of explanation satisfying. It includes a mathematical derivation — so it has some of the rigor of a proof — and it explains a specific observation, elliptical orbits, by appealing to a general principle, gravitation.

Let me contrast that with a different kind of explanation. Suppose you move to a city like Detroit that is racially segregated, and you want to know why it's like that. If you do some research, you might find a paper by Thomas Schelling called "Dynamic Models of Segregation", which proposes a simple model of racial segregation:

Here is my description of the model, from Chapter 9:

> The Schelling model of the city is an array of cells where each cell represents a house. The houses are occupied by two kinds of

> "agents", labeled red and blue, in roughly equal numbers. About 10% of the houses are empty.
>
> At any point in time, an agent might be happy or unhappy, depending on the other agents in the neighborhood. In one version of the model, agents are happy if they have at least two neighbors like themselves, and unhappy if they have one or zero.
>
> The simulation proceeds by choosing an agent at random and checking to see whether it is happy. If so, nothing happens; if not, the agent chooses one of the unoccupied cells at random and moves.

If you start with a simulated city that is entirely unsegregated and run the model for a short time, clusters of similar agents appear. As time passes, the clusters grow and coalesce until there are a small number of large clusters and most agents live in homogeneous neighborhoods.

The degree of segregation in the model is surprising, and it suggests an explanation of segregation in real cities. Maybe Detroit is segregated because people prefer not to be greatly outnumbered and will move if the composition of their neighborhoods makes them unhappy.

Is this explanation satisfying in the same way as the explanation of planetary motion? Many people would say not, but why?

Most obviously, the Schelling model is highly abstract, which is to say not realistic. So you might be tempted to say that people are more complicated than planets. But that can't be right. After all, some planets have people on them, so they have to be more complicated than people.

Both systems are complicated, and both models are based on simplifications. For example, in the model of planetary motion we include forces between the planet and its sun, and ignore interactions between planets. In Schelling's model, we include individual decisions based on local information, and ignore every other aspect of human behavior.

But there are differences of degree. For planetary motion, we can defend the model by showing that the forces we ignore are smaller than the ones we include. And we can extend the model to include other interactions and

www.dbooks.org

show that the effect is small. For Schelling's model it is harder to justify the simplifications.

Another difference is that Schelling's model doesn't appeal to any physical laws, and it uses only simple computation, not mathematical derivation. Models like Schelling's don't look like classical science, and many people find them less compelling, at least at first. But as I will try to demonstrate, these models do useful work, including prediction, explanation, and design. One of the goals of this book is to explain how.

1.1 The changing criteria of science

Complexity science is not just a different set of models; it is also a gradual shift in the criteria models are judged by, and in the kinds of models that are considered acceptable.

For example, classical models tend to be law-based, expressed in the form of equations, and solved by mathematical derivation. Models that fall under the umbrella of complexity are often rule-based, expressed as computations, and simulated rather than analyzed.

Not everyone finds these models satisfactory. For example, in *Sync*, Steven Strogatz writes about his model of spontaneous synchronization in some species of fireflies. He presents a simulation that demonstrates the phenomenon, but then writes:

> I repeated the simulation dozens of times, for other random initial conditions and for other numbers of oscillators. Sync every time. [...] The challenge now was to prove it. Only an ironclad proof would demonstrate, in a way that no computer ever could, that sync was inevitable; and the best kind of proof would clarify *why* it was inevitable.

Strogatz is a mathematician, so his enthusiasm for proofs is understandable, but his proof doesn't address what is, to me, the most interesting part of the phenomenon. In order to prove that "sync was inevitable", Strogatz makes several simplifying assumptions, in particular that each firefly can see all the others.

In my opinion, it is more interesting to explain how an entire valley of fireflies can synchronize *despite the fact that they cannot all see each other*. How this kind of global behavior emerges from local interactions is the subject of Chapter 9. Explanations of these phenomena often use agent-based models, which explore (in ways that would be difficult or impossible with mathematical analysis) the conditions that allow or prevent synchronization.

I am a computer scientist, so my enthusiasm for computational models is probably no surprise. I don't mean to say that Strogatz is wrong, but rather that people have different opinions about what questions to ask and what tools to use to answer them. These opinions are based on value judgments, so there is no reason to expect agreement.

Nevertheless, there is rough consensus among scientists about which models are considered good science, and which others are fringe science, pseudoscience, or not science at all.

A central thesis of this book is that the criteria this consensus is based on change over time, and that the emergence of complexity science reflects a gradual shift in these criteria.

1.2 The axes of scientific models

I have described classical models as based on physical laws, expressed in the form of equations, and solved by mathematical analysis; conversely, models of complex systems are often based on simple rules and implemented as computations.

We can think of this trend as a shift over time along two axes:

Equation-based → simulation-based

Analysis → computation

Complexity science is different in several other ways. I present them here so you know what's coming, but some of them might not make sense until you have seen the examples later in the book.

www.dbooks.org

Continuous → discrete Classical models tend to be based on continuous mathematics, like calculus; models of complex systems are often based on discrete mathematics, including graphs and cellular automatons.

Linear → nonlinear Classical models are often linear, or use linear approximations to nonlinear systems; complexity science is more friendly to nonlinear models.

Deterministic → stochastic Classical models are usually deterministic, which may reflect underlying philosophical determinism, discussed in Chapter 5; complex models often include randomness.

Abstract → detailed In classical models, planets are point masses, planes are frictionless, and cows are spherical (see http://thinkcomplex.com/cow). Simplifications like these are often necessary for analysis, but computational models can be more realistic.

One, two → many Classical models are often limited to small numbers of components. For example, in celestial mechanics the two-body problem can be solved analytically; the three-body problem cannot. Complexity science often works with large numbers of components and larger number of interactions.

Homogeneous → heterogeneous In classical models, the components and interactions tend to be identical; complex models more often include heterogeneity.

These are generalizations, so we should not take them too seriously. And I don't mean to deprecate classical science. A more complicated model is not necessarily better; in fact, it is usually worse.

And I don't mean to say that these changes are abrupt or complete. Rather, there is a gradual migration in the frontier of what is considered acceptable, respectable work. Some tools that used to be regarded with suspicion are now common, and some models that were widely accepted are now regarded with scrutiny.

For example, when Appel and Haken proved the four-color theorem in 1976, they used a computer to enumerate 1,936 special cases that were, in some sense, lemmas of their proof. At the time, many mathematicians did not consider

the theorem truly proved. Now computer-assisted proofs are common and generally (but not universally) accepted.

Conversely, a substantial body of economic analysis is based on a model of human behavior called "Economic man", or, with tongue in cheek, *Homo economicus*. Research based on this model was highly regarded for several decades, especially if it involved mathematical virtuosity. More recently, this model is treated with skepticism, and models that include imperfect information and bounded rationality are hot topics.

1.3 Different models for different purposes

Complex models are often appropriate for different purposes and interpretations:

Predictive → explanatory Schelling's model of segregation might shed light on a complex social phenomenon, but it is not useful for prediction. On the other hand, a simple model of celestial mechanics can predict solar eclipses, down to the second, years in the future.

Realism → instrumentalism Classical models lend themselves to a realist interpretation; for example, most people accept that electrons are real things that exist. Instrumentalism is the view that models can be useful even if the entities they postulate don't exist. George Box wrote what might be the motto of instrumentalism: "All models are wrong, but some are useful."

Reductionism → holism Reductionism is the view that the behavior of a system can be explained by understanding its components. For example, the periodic table of the elements is a triumph of reductionism, because it explains the chemical behavior of elements with a model of electrons in atoms. Holism is the view that some phenomena that appear at the system level do not exist at the level of components, and cannot be explained in component-level terms.

We get back to explanatory models in Chapter 4, instrumentalism in Chapter 6, and holism in Chapter 8.

www.dbooks.org

1.4 Complexity engineering

I have been talking about complex systems in the context of science, but complexity is also a cause, and effect, of changes in engineering and the design of social systems:

Centralized → decentralized Centralized systems are conceptually simple and easier to analyze, but decentralized systems can be more robust. For example, in the World Wide Web clients send requests to centralized servers; if the servers are down, the service is unavailable. In peer-to-peer networks, every node is both a client and a server. To take down the service, you have to take down *every* node.

One-to-many → many-to-many In many communication systems, broadcast services are being augmented, and sometimes replaced, by services that allow users to communicate with each other and create, share, and modify content.

Top-down → bottom-up In social, political and economic systems, many activities that would normally be centrally organized now operate as grassroots movements. Even armies, which are the canonical example of hierarchical structure, are moving toward devolved command and control.

Analysis → computation In classical engineering, the space of feasible designs is limited by our capability for analysis. For example, designing the Eiffel Tower was possible because Gustave Eiffel developed novel analytic techniques, in particular for dealing with wind load. Now tools for computer-aided design and analysis make it possible to build almost anything that can be imagined. Frank Gehry's Guggenheim Museum Bilbao is my favorite example.

Isolation → interaction In classical engineering, the complexity of large systems is managed by isolating components and minimizing interactions. This is still an important engineering principle; nevertheless, the availability of computation makes it increasingly feasible to design systems with complex interactions between components.

Design → search Engineering is sometimes described as a search for solutions in a landscape of possible designs. Increasingly, the search process

can be automated. For example, genetic algorithms explore large design spaces and discover solutions human engineers would not imagine (or like). The ultimate genetic algorithm, evolution, notoriously generates designs that violate the rules of human engineering.

1.5 Complexity thinking

We are getting farther afield now, but the shifts I am postulating in the criteria of scientific modeling are related to 20th century developments in logic and epistemology.

Aristotelian logic → many-valued logic In traditional logic, any proposition is either true or false. This system lends itself to math-like proofs, but fails (in dramatic ways) for many real-world applications. Alternatives include many-valued logic, fuzzy logic, and other systems designed to handle indeterminacy, vagueness, and uncertainty. Bart Kosko discusses some of these systems in *Fuzzy Thinking*.

Frequentist probability → Bayesianism Bayesian probability has been around for centuries, but was not widely used until recently, facilitated by the availability of cheap computation and the reluctant acceptance of subjectivity in probabilistic claims. Sharon Bertsch McGrayne presents this history in *The Theory That Would Not Die*.

Objective → subjective The Enlightenment, and philosophic modernism, are based on belief in objective truth, that is, truths that are independent of the people that hold them. 20th century developments including quantum mechanics, Gödel's Incompleteness Theorem, and Kuhn's study of the history of science called attention to seemingly unavoidable subjectivity in even "hard sciences" and mathematics. Rebecca Goldstein presents the historical context of Gödel's proof in *Incompleteness*.

Physical law → theory → model Some people distinguish between laws, theories, and models. Calling something a "law" implies that it is objectively true and immutable; "theory" suggests that it is subject to revision; and "model" concedes that it is a subjective choice based on simplifications and approximations.

www.dbooks.org

I think they are all the same thing. Some concepts that are called laws are really definitions; others are, in effect, the assertion that a certain model predicts or explains the behavior of a system particularly well. We come back to the nature of physical laws in Section 4.8, Section 5.9 and Section 8.8.

Determinism → indeterminism Determinism is the view that all events are caused, inevitably, by prior events. Forms of indeterminism include randomness, probabilistic causation, and fundamental uncertainty. We come back to this topic in Section 5.5 and Section 10.6

These trends are not universal or complete, but the center of opinion is shifting along these axes. As evidence, consider the reaction to Thomas Kuhn's *The Structure of Scientific Revolutions*, which was reviled when it was published and is now considered almost uncontroversial.

These trends are both cause and effect of complexity science. For example, highly abstracted models are more acceptable now because of the diminished expectation that there should be a unique, correct model for every system. Conversely, developments in complex systems challenge determinism and the related concept of physical law.

This chapter is an overview of the themes coming up in the book, but not all of it will make sense before you see the examples. When you get to the end of the book, you might find it helpful to read this chapter again.

www.dbooks.org

Chapter 2

Graphs

The next three chapters are about systems made up of components and connections between components. For example, in a social network, the components are people and connections represent friendships, business relationships, etc. In an ecological food web, the components are species and the connections represent predator-prey relationships.

In this chapter, I introduce NetworkX, a Python package for building models of these systems. We start with the Erdős-Rényi model, which has interesting mathematical properties. In the next chapter we move on to models that are more useful for explaining real-world systems.

The code for this chapter is in `chap02.ipynb` in the repository for this book. More information about working with the code is in Section 0.3.

2.1 What is a graph?

To most people a "graph" is a visual representation of data, like a bar chart or a plot of stock prices over time. That's not what this chapter is about.

In this chapter, a **graph** is a representation of a system that contains discrete, interconnected elements. The elements are represented by **nodes** — also called **vertices** – and the interconnections are represented by **edges**.

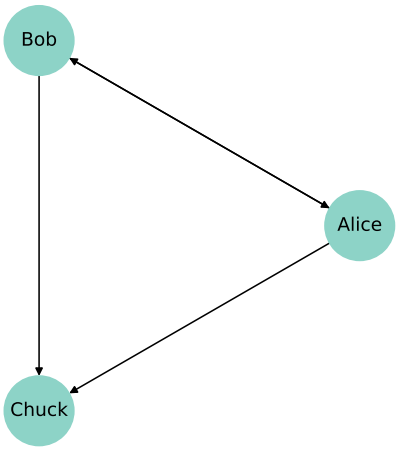


Figure 2.1: A directed graph that represents a social network.

For example, you could represent a road map with a node for each city and an edge for each road between cities. Or you could represent a social network using a node for each person, with an edge between two people if they are friends.

In some graphs, edges have attributes like length, cost, or weight. For example, in a road map, the length of an edge might represent distance between cities or travel time. In a social network there might be different kinds of edges to represent different kinds of relationships: friends, business associates, etc.

Edges may be **directed** or **undirected**, depending on whether the relationships they represent are asymmetric or symmetric. In a road map, you might represent a one-way street with a directed edge and a two-way street with an undirected edge. In some social networks, like Facebook, friendship is symmetric: if A is friends with B then B is friends with A. But on Twitter, for example, the "follows" relationship is not symmetric; if A follows B, that doesn't imply that B follows A. So you might use undirected edges to represent a Facebook network and directed edges for Twitter.

Graphs have interesting mathematical properties, and there is a branch of mathematics called **graph theory** that studies them.

Graphs are also useful, because there are many real world problems that can be solved using **graph algorithms**. For example, Dijkstra's shortest path algorithm is an efficient way to find the shortest path from a node to all other

www.dbooks.org

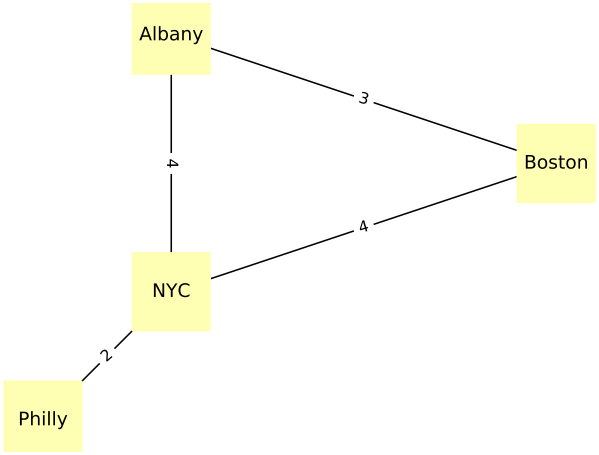


Figure 2.2: An undirected graph that represents driving time between cities.

nodes in a graph. A **path** is a sequence of nodes with an edge between each consecutive pair.

Graphs are usually drawn with squares or circles for nodes and lines for edges. For example, the directed graph in Figure 2.1 might represent three people who follow each other on Twitter. The arrow indicates the direction of the relationship. In this example, Alice and Bob follow each other, both follow Chuck, and Chuck follows no one.

The undirected graph in Figure 2.2 shows four cities in the northeast United States; the labels on the edges indicate driving time in hours. In this example the placement of the nodes corresponds roughly to the geography of the cities, but in general the layout of a graph is arbitrary.

2.2 NetworkX

To represent graphs, we'll use a package called NetworkX, which is the most commonly used network library in Python. You can read more about it at http://thinkcomplex.com/netx, but I'll explain it as we go along.

We can create a directed graph by importing NetworkX (usually imported as `nx`) and instantiating `nx.DiGraph`:

```
import networkx as nx
G = nx.DiGraph()
```

At this point, `G` is a `DiGraph` object that contains no nodes and no edges. We can add nodes using the `add_node` method:

```
G.add_node('Alice')
G.add_node('Bob')
G.add_node('Chuck')
```

Now we can use the `nodes` method to get a list of nodes:

```
>>> list(G.nodes())
NodeView(('Alice', 'Bob', 'Chuck'))
```

The `nodes` method returns a `NodeView`, which can be used in a for loop or, as in this example, used to make a list.

Adding edges works pretty much the same way:

```
G.add_edge('Alice', 'Bob')
G.add_edge('Alice', 'Chuck')
G.add_edge('Bob', 'Alice')
G.add_edge('Bob', 'Chuck')
```

And we can use `edges` to get the list of edges:

```
>>> list(G.edges())
[('Alice', 'Bob'), ('Alice', 'Chuck'),
 ('Bob', 'Alice'), ('Bob', 'Chuck')]
```

NetworkX provides several functions for drawing graphs; `draw_circular` arranges the nodes in a circle and connects them with edges:

```
nx.draw_circular(G,
                 node_color=COLORS[0],
                 node_size=2000,
                 with_labels=True)
```

That's the code I use to generate Figure 2.1. The option `with_labels` causes the nodes to be labeled; in the next example we'll see how to label the edges.

www.dbooks.org

To generate Figure 2.2, I start with a dictionary that maps from each city name to its approximate longitude and latitude:

```
positions = dict(Albany=(-74, 43),
                 Boston=(-71, 42),
                 NYC=(-74, 41),
                 Philly=(-75, 40))
```

Since this is an undirected graph, I instantiate `nx.Graph`:

```
G = nx.Graph()
```

Then I can use `add_nodes_from` to iterate the keys of `positions` and add them as nodes:

```
G.add_nodes_from(positions)
```

Next I'll make a dictionary that maps from each edge to the corresponding driving time:

```
drive_times = {('Albany', 'Boston'): 3,
               ('Albany', 'NYC'): 4,
               ('Boston', 'NYC'): 4,
               ('NYC', 'Philly'): 2}
```

Now I can use `add_edges_from`, which iterates the keys of `drive_times` and adds them as edges:

```
G.add_edges_from(drive_times)
```

Instead of `draw_circular`, which arranges the nodes in a circle, I'll use `draw`, which takes the position dictionary as the second parameter:

```
nx.draw(G, positions,
        node_color=COLORS[1],
        node_shape='s',
        node_size=2500,
        with_labels=True)
```

`draw` uses `positions` to determine the locations of the nodes.

To add the edge labels, we use `draw_networkx_edge_labels`:

```
nx.draw_networkx_edge_labels(G, positions,
                             edge_labels=drive_times)
```

The `edge_labels` parameter expects a dictionary that maps from each pair of nodes to a label; in this case, the labels are driving times between cities. And that's how I generated Figure 2.2.

In both of these examples, the nodes are strings, but in general they can be any hashable type.

2.3 Random graphs

A random graph is just what it sounds like: a graph with nodes and edges generated at random. Of course, there are many random processes that can generate graphs, so there are many kinds of random graphs.

One of the more interesting kinds is the Erdős-Rényi model, studied by Paul Erdős and Alfréd Rényi in the 1960s.

An Erdős-Rényi graph (ER graph) is characterized by two parameters: n is the number of nodes and p is the probability that there is an edge between any two nodes. See http://thinkcomplex.com/er.

Erdős and Rényi studied the properties of these random graphs; one of their surprising results is the existence of abrupt changes in the properties of random graphs as random edges are added.

One of the properties that displays this kind of transition is connectivity. An undirected graph is **connected** if there is a path from every node to every other node.

In an ER graph, the probability that the graph is connected is very low when p is small and nearly 1 when p is large. Between these two regimes, there is a rapid transition at a particular value of p, denoted p^*.

Erdős and Rényi showed that this critical value is $p^* = (\ln n)/n$, where n is the number of nodes. A random graph, $G(n, p)$, is unlikely to be connected if $p < p^*$ and very likely to be connected if $p > p^*$.

To test this claim, we'll develop algorithms to generate random graphs and check whether they are connected.

www.dbooks.org

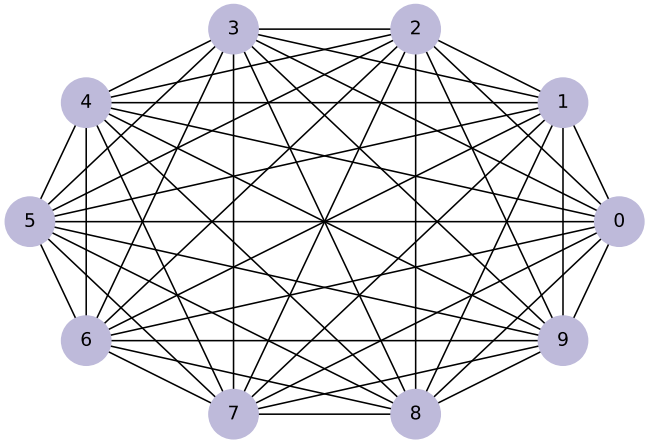


Figure 2.3: A complete graph with 10 nodes.

2.4 Generating graphs

I'll start by generating a **complete** graph, which is a graph where every node is connected to every other.

Here's a generator function that takes a list of nodes and enumerates all distinct pairs. If you are not familiar with generator functions, you can read about them at http://thinkcomplex.com/gen.

```
def all_pairs(nodes):
    for i, u in enumerate(nodes):
        for j, v in enumerate(nodes):
            if i>j:
                yield u, v
```

We can use `all_pairs` to construct a complete graph:

```
def make_complete_graph(n):
    G = nx.Graph()
    nodes = range(n)
    G.add_nodes_from(nodes)
    G.add_edges_from(all_pairs(nodes))
    return G
```

`make_complete_graph` takes the number of nodes, `n`, and returns a new `Graph` with `n` nodes and edges between all pairs of nodes.

The following code makes a complete graph with 10 nodes and draws it:

```
complete = make_complete_graph(10)
nx.draw_circular(complete,
                 node_color=COLORS[2],
                 node_size=1000,
                 with_labels=True)
```

Figure 2.3 shows the result. Soon we will modify this code to generate ER graphs, but first we'll develop functions to check whether a graph is connected.

2.5 Connected graphs

A graph is **connected** if there is a path from every node to every other node (see http://thinkcomplex.com/conn).

For many applications involving graphs, it is useful to check whether a graph is connected. Fortunately, there is a simple algorithm that does it.

You can start at any node and check whether you can reach all other nodes. If you can reach a node, v, you can reach any of the **neighbors** of v, which are the nodes connected to v by an edge.

The `Graph` class provides a method called `neighbors` that returns a list of neighbors for a given node. For example, in the complete graph we generated in the previous section:

```
>>> complete.neighbors(0)
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Suppose we start at node s. We can mark s as "seen" and mark its neighbors. Then we mark the neighbor's neighbors, and their neighbors, and so on, until we can't reach any more nodes. If all nodes are seen, the graph is connected.

Here's what that looks like in Python:

www.dbooks.org

```
def reachable_nodes(G, start):
    seen = set()
    stack = [start]
    while stack:
        node = stack.pop()
        if node not in seen:
            seen.add(node)
            stack.extend(G.neighbors(node))
    return seen
```

`reachable_nodes` takes a `Graph` and a starting node, `start`, and returns the set of nodes that can be reached from `start`.

Initially the set, `seen`, is empty, and we create a list called `stack` that keeps track of nodes we have discovered but not yet processed. Initially the stack contains a single node, `start`.

Now, each time through the loop, we:

1. Remove one node from the stack.
2. If the node is already in `seen`, we go back to Step 1.
3. Otherwise, we add the node to `seen` and add its neighbors to the stack.

When the stack is empty, we can't reach any more nodes, so we break out of the loop and return `seen`.

As an example, we can find all nodes in the complete graph that are reachable from node 0:

```
>>> reachable_nodes(complete, 0)
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

Initially, the stack contains node 0 and `seen` is empty. The first time through the loop, node 0 is added to `seen` and all the other nodes are added to the stack (since they are all neighbors of node 0).

The next time through the loop, `pop` returns the last element in the stack, which is node 9. So node 9 gets added to `seen` and its neighbors get added to the stack.

Notice that the same node can appear more than once in the stack; in fact, a node with k neighbors will be added to the stack k times. Later, we will look for ways to make this algorithm more efficient.

We can use `reachable_nodes` to write `is_connected`:

```
def is_connected(G):
    start = next(iter(G))
    reachable = reachable_nodes(G, start)
    return len(reachable) == len(G)
```

`is_connected` chooses a starting node by making a node iterator and choosing the first element. Then it uses `reachable` to get the set of nodes that can be reached from `start`. If the size of this set is the same as the size of the graph, that means we can reach all nodes, which means the graph is connected.

A complete graph is, not surprisingly, connected:

```
>>> is_connected(complete)
True
```

In the next section we will generate ER graphs and check whether they are connected.

2.6 Generating ER graphs

The ER graph $G(n, p)$ contains n nodes, and each pair of nodes is connected by an edge with probability p. Generating an ER graph is similar to generating a complete graph.

The following generator function enumerates all possible edges and chooses which ones should be added to the graph:

```
def random_pairs(nodes, p):
    for edge in all_pairs(nodes):
        if flip(p):
            yield edge
```

`random_pairs` uses `flip`:

www.dbooks.org

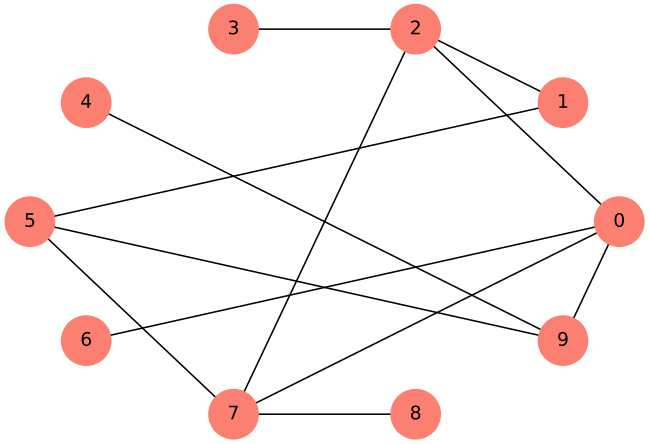


Figure 2.4: An ER graph with `n=10` and `p=0.3`.

```
def flip(p):
    return np.random.random() < p
```

This is the first example we're seen that uses NumPy. Following convention, I import `numpy` as `np`. NumPy provides a module named `random`, which provides a method named `random`, which returns a number between 0 and 1, uniformly distributed.

So `flip` returns `True` with the given probability, `p`, and `False` with the complementary probability, `1-p`.

Finally, `make_random_graph` generates and returns the ER graph $G(n, p)$:

```
def make_random_graph(n, p):
    G = nx.Graph()
    nodes = range(n)
    G.add_nodes_from(nodes)
    G.add_edges_from(random_pairs(nodes, p))
    return G
```

`make_random_graph` is almost identical to `make_complete_graph`; the only difference is that it uses `random_pairs` instead of `all_pairs`.

Here's an example with `p=0.3`:

```
random_graph = make_random_graph(10, 0.3)
```

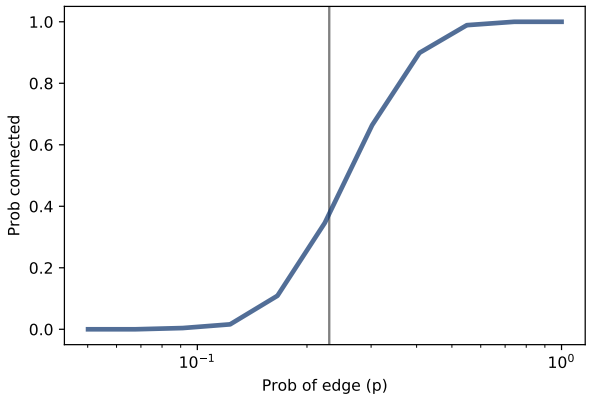


Figure 2.5: Probability of connectivity with $n = 10$ and a range of p. The vertical line shows the predicted critical value.

Figure 2.4 shows the result. This graph turns out to be connected; in fact, most ER graphs with $n = 10$ and $p = 0.3$ are connected. In the next section, we'll see how many.

2.7 Probability of connectivity

For given values of n and p, we would like to know the probability that $G(n, p)$ is connected. We can estimate it by generating a large number of random graphs and counting how many are connected. Here's how:

```
def prob_connected(n, p, iters=100):
    tf = [is_connected(make_random_graph(n, p))
          for i in range(iters)]
    return np.mean(bool)
```

The parameters `n` and `p` are passed along to `make_random_graph`; `iters` is the number of random graphs we generate.

This function uses a list comprehension; if you are not familiar with this feature, you can read about it at http://thinkcomplex.com/comp.

The result, `tf`, is a list of boolean values: `True` for each graph that's connected and `False` for each one that's not.

www.dbooks.org

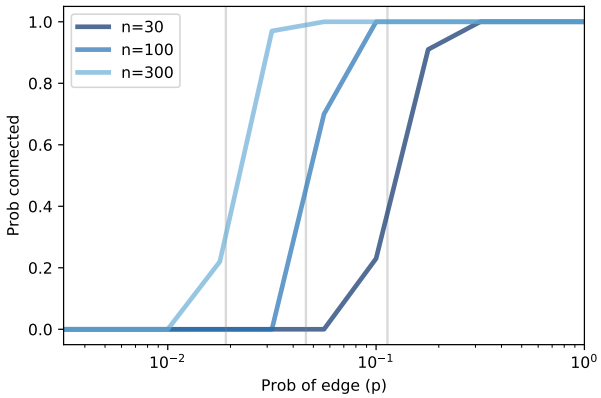


Figure 2.6: Probability of connectivity for several values of n and a range of p.

`np.mean` is a NumPy function that computes the mean of this list, treating `True` as 1 and `False` as 0. The result is the fraction of random graphs that are connected.

```
>>> prob_connected(10, 0.23, iters=10000)
0.33
```

I chose 0.23 because it is close to the critical value where the probability of connectivity goes from near 0 to near 1. According to Erdős and Rényi, $p^* = \ln n/n = 0.23$.

We can get a clearer view of the transition by estimating the probability of connectivity for a range of values of p:

```
n = 10
ps = np.logspace(-2.5, 0, 11)
ys = [prob_connected(n, p) for p in ps]
```

The NumPy function `logspace` returns an array of 11 values from $10^{-2.5}$ to $10^0 = 1$, equally spaced on a logarithmic scale.

For each value of `p` in the array, we compute the probability that a graph with parameter `p` is connected and store the results in `ys`.

Figure 2.5 shows the results, with a vertical line at the computed critical

value, $p^* = 0.23$. As expected, the transition from 0 to 1 occurs near the critical value.

Figure 2.6 shows similar results for larger values of n. As n increases, the critical value gets smaller and the transition gets more abrupt.

These experimental results are consistent with the analytic results Erdős and Rényi presented in their papers.

2.8 Analysis of graph algorithms

Earlier in this chapter I presented an algorithm for checking whether a graph is connected; in the next few chapters, we will see other graph algorithms. Along the way, we will analyze the performance of those algorithms, figuring out how their run times grow as the size of the graphs increases.

If you are not already familiar with analysis of algorithms, you might want to read Appendix B of *Think Python, 2nd Edition*, at http://thinkcomplex.com/tp2.

The order of growth for graph algorithms is usually expressed as a function of n, the number of vertices (nodes), and m, the number of edges.

As an example, let's analyze `reachable_nodes` from Section 2.5:

```
def reachable_nodes(G, start):
    seen = set()
    stack = [start]
    while stack:
        node = stack.pop()
        if node not in seen:
            seen.add(node)
            stack.extend(G.neighbors(node))
    return seen
```

Each time through the loop, we pop a node off the stack; by default, `pop` removes and returns the last element of a list, which is a constant time operation.

www.dbooks.org

Next we check whether the node is in `seen`, which is a set, so checking membership is constant time.

If the node is not already in `seen`, we add it, which is constant time, and then add the neighbors to the stack, which is linear in the number of neighbors.

To express the run time in terms of n and m, we can add up the total number of times each node is added to `seen` and `stack`.

Each node is only added to `seen` once, so the total number of additions is n.

But nodes might be added to `stack` many times, depending on how many neighbors they have. If a node has k neighbors, it is added to `stack` k times. Of course, if it has k neighbors, that means it is connected to k edges.

So the total number of additions to `stack` is the total number of edges, m, doubled because we consider every edge twice.

Therefore, the order of growth for this function is $O(n + m)$, which is a convenient way to say that the run time grows in proportion to either n or m, whichever is bigger.

If we know the relationship between n and m, we can simplify this expression. For example, in a complete graph the number of edges is $n(n-1)/2$, which is in $O(n^2)$. So for a complete graph, `reachable_nodes` is quadratic in n.

2.9 Exercises

The code for this chapter is in `chap02.ipynb`, which is a Jupyter notebook in the repository for this book. For more information about working with this code, see Section 0.3.

Exercise 2.1 Launch `chap02.ipynb` and run the code. There are a few short exercises embedded in the notebook that you might want to try.

Exercise 2.2 In Section 2.8 we analyzed the performance of `reachable_nodes` and classified it in $O(n + m)$, where n is the number of nodes and m is the number of edges. Continuing the analysis, what is the order of growth for `is_connected`?

```
def is_connected(G):
    start = list(G)[0]
    reachable = reachable_nodes(G, start)
    return len(reachable) == len(G)
```

Exercise 2.3 In my implementation of `reachable_nodes`, you might be bothered by the apparent inefficiency of adding *all* neighbors to the stack without checking whether they are already in `seen`. Write a version of this function that checks the neighbors before adding them to the stack. Does this "optimization" change the order of growth? Does it make the function faster?

Exercise 2.4 There are actually two kinds of ER graphs. The one we generated in this chapter, $G(n, p)$, is characterized by two parameters, the number of nodes and the probability of an edge between nodes.

An alternative definition, denoted $G(n, m)$, is also characterized by two parameters: the number of nodes, n, and the number of edges, m. Under this definition, the number of edges is fixed, but their location is random.

Repeat the experiments we did in this chapter using this alternative definition. Here are a few suggestions for how to proceed:

1. Write a function called `m_pairs` that takes a list of nodes and the number of edges, m, and returns a random selection of m edges. A simple way to do that is to generate a list of all possible edges and use `random.sample`.

2. Write a function called `make_m_graph` that takes n and m and returns a random graph with n nodes and m edges.

3. Make a version of `prob_connected` that uses `make_m_graph` instead of `make_random_graph`.

4. Compute the probability of connectivity for a range of values of m.

How do the results of this experiment compare to the results using the first type of ER graph?

www.dbooks.org

Chapter 3

Small World Graphs

Many networks in the real world, including social networks, have the "small world property", which is that the average distance between nodes, measured in number of edges on the shortest path, is much smaller than expected.

In this chapter, I present Stanley Milgram's famous Small World Experiment, which was the first demonstration of the small world property in a real social network. Then we'll consider Watts-Strogatz graphs, which are intended as a model of small world graphs. I'll replicate the experiment Watts and Strogatz performed and explain what it is intended to show.

Along the way, we'll see two new graph algorithms: breadth-first search (BFS) and Dijkstra's algorithm for computing the shortest path between nodes in a graph.

The code for this chapter is in `chap03.ipynb` in the repository for this book. More information about working with the code is in Section 0.3.

3.1 Stanley Milgram

Stanley Milgram was an American social psychologist who conducted two of the most famous experiments in social science, the Milgram experiment, which studied people's obedience to authority (http://thinkcomplex.com/

milgram) and the Small World Experiment, which studied the structure of social networks (http://thinkcomplex.com/small).

In the Small World Experiment, Milgram sent a package to several randomly-chosen people in Wichita, Kansas, with instructions asking them to forward an enclosed letter to a target person, identified by name and occupation, in Sharon, Massachusetts (which happens to be the town near Boston where I grew up). The subjects were told that they could mail the letter directly to the target person only if they knew him personally; otherwise they were instructed to send it, and the same instructions, to a relative or friend they thought would be more likely to know the target person.

Many of the letters were never delivered, but for the ones that were the average path length — the number of times the letters were forwarded — was about six. This result was taken to confirm previous observations (and speculations) that the typical distance between any two people in a social network is about "six degrees of separation".

This conclusion is surprising because most people expect social networks to be localized — people tend to live near their friends — and in a graph with local connections, path lengths tend to increase in proportion to geographical distance. For example, most of my friends live nearby, so I would guess that the average distance between nodes in a social network is about 50 miles. Wichita is about 1600 miles from Boston, so if Milgram's letters traversed typical links in the social network, they should have taken 32 hops, not 6.

3.2 Watts and Strogatz

In 1998 Duncan Watts and Steven Strogatz published a paper in *Nature*, "Collective dynamics of 'small-world' networks", that proposed an explanation for the small world phenomenon. You can download it from http://thinkcomplex.com/watts.

Watts and Strogatz start with two kinds of graph that were well understood: random graphs and regular graphs. In a random graph, nodes are connected at random. In a regular graph, every node has the same number of neighbors. They consider two properties of these graphs, clustering and path length:

www.dbooks.org

- Clustering is a measure of the "cliquishness" of the graph. In a graph, a **clique** is a subset of nodes that are all connected to each other; in a social network, a clique is a set of people who are all friends with each other. Watts and Strogatz defined a clustering coefficient that quantifies the likelihood that two nodes that are connected to the same node are also connected to each other.

- Path length is a measure of the average distance between two nodes, which corresponds to the degrees of separation in a social network.

Watts and Strogatz show that regular graphs have high clustering and high path lengths, whereas random graphs with the same size usually have low clustering and low path lengths. So neither of these is a good model of social networks, which combine high clustering with short path lengths.

Their goal was to create a **generative model** of a social network. A generative model tries to explain a phenomenon by modeling the process that builds or leads to the phenomenon. Watts and Strogatz proposed this process for building small-world graphs:

1. Start with a regular graph with n nodes and each node connected to k neighbors.

2. Choose a subset of the edges and "rewire" them by replacing them with random edges.

The probability that an edge is rewired is a parameter, p, that controls how random the graph is. With $p = 0$, the graph is regular; with $p = 1$ it is completely random.

Watts and Strogatz found that small values of p yield graphs with high clustering, like a regular graph, and low path lengths, like a random graph.

In this chapter I replicate the Watts and Strogatz experiment in the following steps:

1. We'll start by constructing a ring lattice, which is a kind of regular graph.

2. Then we'll rewire it as Watts and Strogatz did.

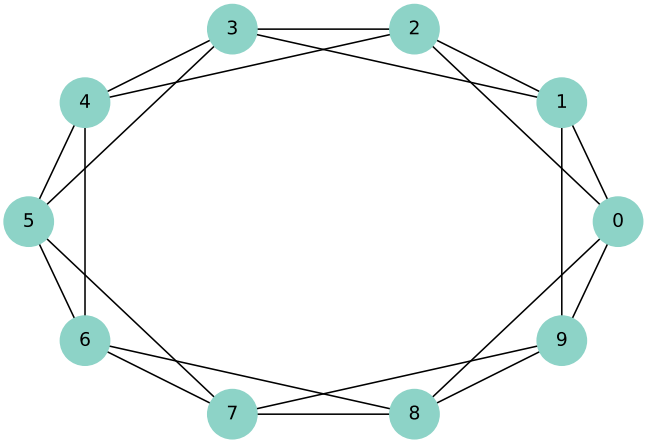


Figure 3.1: A ring lattice with $n = 10$ and $k = 4$.

3. We'll write a function to measure the degree of clustering and use a NetworkX function to compute path lengths.

4. Then we'll compute the degree of clustering and path length for a range of values of p.

5. Finally, I'll present Dijkstra's algorithm, which computes shortest paths efficiently.

3.3 Ring lattice

A **regular** graph is a graph where each node has the same number of neighbors; the number of neighbors is also called the **degree** of the node.

A ring lattice is a kind of regular graph, which Watts and Strogatz use as the basis of their model. In a ring lattice with n nodes, the nodes can be arranged in a circle with each node connected to the k nearest neighbors.

For example, a ring lattice with $n = 3$ and $k = 2$ would contain the following edges: $(0, 1)$, $(1, 2)$, and $(2, 0)$. Notice that the edges "wrap around" from the highest-numbered node back to 0.

More generally, we can enumerate the edges like this:

www.dbooks.org

```
def adjacent_edges(nodes, halfk):
    n = len(nodes)
    for i, u in enumerate(nodes):
        for j in range(i+1, i+halfk+1):
            v = nodes[j % n]
            yield u, v
```

`adjacent_edges` takes a list of nodes and a parameter, `halfk`, which is half of k. It is a generator function that yields one edge at a time. It uses the modulus operator, `%`, to wrap around from the highest-numbered node to the lowest.

We can test it like this:

```
>>> nodes = range(3)
>>> for edge in adjacent_edges(nodes, 1):
...     print(edge)
(0, 1)
(1, 2)
(2, 0)
```

Now we can use `adjacent_edges` to make a ring lattice:

```
def make_ring_lattice(n, k):
    G = nx.Graph()
    nodes = range(n)
    G.add_nodes_from(nodes)
    G.add_edges_from(adjacent_edges(nodes, k//2))
    return G
```

Notice that `make_ring_lattice` uses floor division to compute `halfk`, so it is only correct if `k` is even. If `k` is odd, floor division rounds down, so the result is a ring lattice with degree `k-1`. As one of the exercises at the end of the chapter, you will generate regular graphs with odd values of `k`.

We can test `make_ring_lattice` like this:

```
lattice = make_ring_lattice(10, 4)
```

Figure 3.1 shows the result.

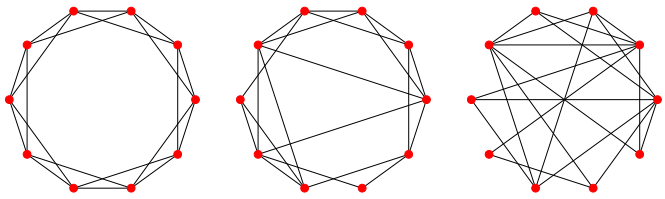


Figure 3.2: WS graphs with $n = 20$, $k = 4$, and $p = 0$ (left), $p = 0.2$ (middle), and $p = 1$ (right).

3.4 WS graphs

To make a Watts-Strogatz (WS) graph, we start with a ring lattice and "rewire" some of the edges. In their paper, Watts and Strogatz consider the edges in a particular order and rewire each one with probability p. If an edge is rewired, they leave the first node unchanged and choose the second node at random. They don't allow self loops or multiple edges; that is, you can't have a edge from a node to itself, and you can't have more than one edge between the same two nodes.

Here is my implementation of this process.

```
def rewire(G, p):
    nodes = set(G)
    for u, v in G.edges():
        if flip(p):
            choices = nodes - {u} - set(G[u])
            new_v = np.random.choice(list(choices))
            G.remove_edge(u, v)
            G.add_edge(u, new_v)
```

The parameter `p` is the probability of rewiring an edge. The `for` loop enumerates the edges and uses `flip` (defined in Section 2.6) to choose which ones get rewired.

If we are rewiring an edge from node `u` to node `v`, we have to choose a replacement for `v`, called `new_v`.

www.dbooks.org

1. To compute the possible choices, we start with `nodes`, which is a set, and subtract off `u` and its neighbors, which avoids self loops and multiple edges.

2. To choose `new_v`, we use the NumPy function `choice`, which is in the module `random`.

3. Then we remove the existing edge from `u` to `v`, and

4. Add a new edge from `u` to `new_v`.

As an aside, the expression `G[u]` returns a dictionary that contains the neighbors of `u` as keys. It is usually faster than using `G.neighbors` (see http://thinkcomplex.com/neigh).

This function does not consider the edges in the order specified by Watts and Strogatz, but that doesn't seem to affect the results.

Figure 3.2 shows WS graphs with $n = 20$, $k = 4$, and a range of values of p. When $p = 0$, the graph is a ring lattice. When $p = 1$, it is completely random. As we'll see, the interesting things happen in between.

3.5 Clustering

The next step is to compute the clustering coefficient, which quantifies the tendency for the nodes to form cliques. A **clique** is a set of nodes that are completely connected; that is, there are edges between all pairs of nodes in the set.

Suppose a particular node, u, has k neighbors. If all of the neighbors are connected to each other, there would be $k(k-1)/2$ edges among them. The fraction of those edges that actually exist is the local clustering coefficient for u, denoted C_u.

If we compute the average of C_u over all nodes, we get the "network average clustering coefficient", denoted $\bar{C}$.

Here is a function that computes it.

```
def node_clustering(G, u):
    neighbors = G[u]
    k = len(neighbors)
    if k < 2:
        return np.nan

    possible = k * (k-1) / 2
    exist = 0
    for v, w in all_pairs(neighbors):
        if G.has_edge(v, w):
            exist +=1
    return exist / possible
```

Again I use `G[u]`, which returns a dictionary with the neighbors of `u` as keys.

If a node has fewer than 2 neighbors, the clustering coefficient is undefined, so we return `np.nan`, which is a special value that indicates "Not a Number".

Otherwise we compute the number of possible edges among the neighbors, count the number of those edges that actually exist, and return the fraction that exist.

We can test the function like this:

```
>>> lattice = make_ring_lattice(10, 4)
>>> node_clustering(lattice, 1)
0.5
```

In a ring lattice with $k = 4$, the clustering coefficient for each node is 0.5 (if you are not convinced, take another look at Figure 3.1).

Now we can compute the network average clustering coefficient like this:

```
def clustering_coefficient(G):
    cu = [node_clustering(G, node) for node in G]
    return np.nanmean(cu)
```

The NumPy function `nanmean` computes the mean of the local clustering coefficients, ignoring any values that are `NaN`.

We can test `clustering_coefficient` like this:

www.dbooks.org

```
>>> clustering_coefficient(lattice)
0.5
```

In this graph, the local clustering coefficient for all nodes is 0.5, so the average across nodes is 0.5. Of course, we expect this value to be different for WS graphs.

3.6 Shortest path lengths

The next step is to compute the characteristic path length, L, which is the average length of the shortest path between each pair of nodes. To compute it, I'll start with a function provided by NetworkX, `shortest_path_length`. I'll use it to replicate the Watts and Strogatz experiment, then I'll explain how it works.

Here's a function that takes a graph and returns a list of shortest path lengths, one for each pair of nodes.

```
def path_lengths(G):
    length_map = nx.shortest_path_length(G)
    lengths = [length_map[u][v] for u, v in all_pairs(G)]
    return lengths
```

The return value from `nx.shortest_path_length` is a dictionary of dictionaries. The outer dictionary maps from each node, `u`, to a dictionary that maps from each node, `v`, to the length of the shortest path from `u` to `v`.

With the list of lengths from `path_lengths`, we can compute L like this:

```
def characteristic_path_length(G):
    return np.mean(path_lengths(G))
```

And we can test it with a small ring lattice:

```
>>> lattice = make_ring_lattice(3, 2)
>>> characteristic_path_length(lattice)
1.0
```

In this example, all 3 nodes are connected to each other, so the mean path length is 1.

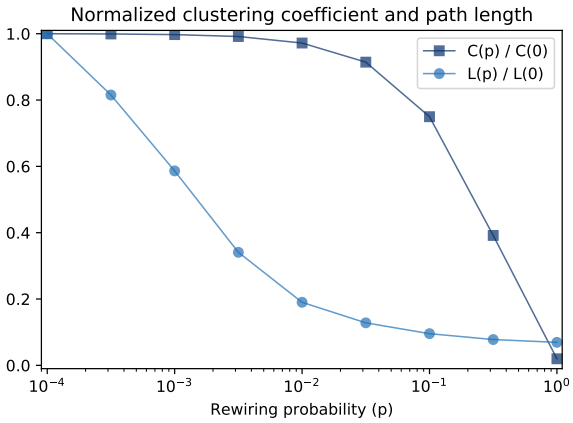


Figure 3.3: Clustering coefficient (C) and characteristic path length (L) for WS graphs with $n = 1000$, $k = 10$, and a range of p.

3.7 The WS experiment

Now we are ready to replicate the WS experiment, which shows that for a range of values of p, a WS graph has high clustering like a regular graph and short path lengths like a random graph.

I'll start with `run_one_graph`, which takes `n`, `k`, and `p`; it generates a WS graph with the given parameters and computes the mean path length, `mpl`, and clustering coefficient, `cc`:

```
def run_one_graph(n, k, p):
    ws = make_ws_graph(n, k, p)
    mpl = characteristic_path_length(ws)
    cc = clustering_coefficient(ws)
    return mpl, cc
```

Watts and Strogatz ran their experiment with `n=1000` and `k=10`. With these parameters, `run_one_graph` takes a few seconds on my computer; most of that time is spent computing the mean path length.

Now we need to compute these values for a range of `p`. I'll use the NumPy function `logspace` again to compute `ps`:

```
ps = np.logspace(-4, 0, 9)
```

www.dbooks.org

Here's the function that runs the experiment:

```
def run_experiment(ps, n=1000, k=10, iters=20):
    res = []
    for p in ps:
        t = [run_one_graph(n, k, p) for _ in range(iters)]
        means = np.array(t).mean(axis=0)
        res.append(means)
    return np.array(res)
```

For each value of `p`, we generate 20 random graphs and average the results. Since the return value from `run_one_graph` is a pair, `t` is a list of pairs. When we convert it to an array, we get one row for each iteration and columns for `L` and `C`. Calling `mean` with the option `axis=0` computes the mean of each column; the result is an array with one row and two columns.

When the loop exits, `means` is a list of pairs, which we convert to a NumPy array with one row for each value of `p` and columns for `L` and `C`.

We can extract the columns like this:

```
L, C = np.transpose(res)
```

In order to plot `L` and `C` on the same axes, we standardize them by dividing through by the first element:

```
L /= L[0]
C /= C[0]
```

Figure 3.3 shows the results. As p increases, the mean path length drops quickly, because even a small number of randomly rewired edges provide shortcuts between regions of the graph that are far apart in the lattice. On the other hand, removing local links decreases the clustering coefficient much more slowly.

As a result, there is a wide range of p where a WS graph has the properties of a small world graph, high clustering and low path lengths.

And that's why Watts and Strogatz propose WS graphs as a model for real-world networks that exhibit the small world phenomenon.

3.8 What kind of explanation is *that*?

If you ask me why planetary orbits are elliptical, I might start by modeling a planet and a star as point masses; I would look up the law of universal gravitation at http://thinkcomplex.com/grav and use it to write a differential equation for the motion of the planet. Then I would either derive the orbit equation or, more likely, look it up at http://thinkcomplex.com/orbit. With a little algebra, I could derive the conditions that yield an elliptical orbit. Then I would argue that the objects we consider planets satisfy these conditions.

People, or at least scientists, are generally satisfied with this kind of explanation. One of the reasons for its appeal is that the assumptions and approximations in the model seem reasonable. Planets and stars are not really point masses, but the distances between them are so big that their actual sizes are negligible. Planets in the same solar system can affect each other's orbits, but the effect is usually small. And we ignore relativistic effects, again on the assumption that they are small.

This explanation is also appealing because it is equation-based. We can express the orbit equation in a closed form, which means that we can compute orbits efficiently. It also means that we can derive general expressions for the orbital velocity, orbital period, and other quantities.

Finally, I think this kind of explanation is appealing because it has the form of a mathematical proof. It is important to remember that the proof pertains to the model and not the real world. That is, we can prove that an idealized model yields elliptical orbits, but we can't prove that real orbits are ellipses (in fact, they are not). Nevertheless, the resemblance to a proof is appealing.

By comparison, Watts and Strogatz's explanation of the small world phenomenon may seem less satisfying. First, the model is more abstract, which is to say less realistic. Second, the results are generated by simulation, not by mathematical analysis. Finally, the results seem less like a proof and more like an example.

Many of the models in this book are like the Watts and Strogatz model: abstract, simulation-based and (at least superficially) less formal than conventional mathematical models. One of the goals of this book is to consider the questions these models raise:

www.dbooks.org

- What kind of work can these models do: are they predictive, or explanatory, or both?
- Are the explanations these models offer less satisfying than explanations based on more traditional models? Why?
- How should we characterize the differences between these and more conventional models? Are they different in kind or only in degree?

Over the course of the book I will offer my answers to these questions, but they are tentative and sometimes speculative. I encourage you to consider them skeptically and reach your own conclusions.

3.9 Breadth-First Search

When we computed shortest paths, we used a function provided by NetworkX, but I have not explained how it works. To do that, I'll start with breadth-first search, which is the basis of Dijkstra's algorithm for computing shortest paths.

In Section 2.5 I presented `reachable_nodes`, which finds all the nodes that can be reached from a given starting node:

```
def reachable_nodes(G, start):
    seen = set()
    stack = [start]
    while stack:
        node = stack.pop()
        if node not in seen:
            seen.add(node)
            stack.extend(G.neighbors(node))
    return seen
```

I didn't say so at the time, but `reachable_nodes` performs a depth-first search (DFS). Now we'll modify it to perform breadth-first search (BFS).

To understand the difference, imagine you are exploring a castle. You start in a room with three doors marked A, B, and C. You open door C and discover another room, with doors marked D, E, and F.

Which door do you open next? If you are feeling adventurous, you might want to go deeper into the castle and choose D, E, or F. That would be a depth-first search.

But if you wanted to be more systematic, you might go back and explore A and B before D, E, and F. That would be a breadth-first search.

In `reachable_nodes`, we use the list method `pop` to choose the next node to "explore". By default, `pop` returns the last element of the list, which is the last one we added. In the example, that would be door F.

If we want to perform a BFS instead, the simplest solution is to pop the first element of the list:

```
        node = stack.pop(0)
```

That works, but it is slow. In Python, popping the last element of a list takes constant time, but popping the first element is linear in the length of the list. In the worst case, the length of the stack is $O(n)$, which makes this implementation of BFS $O(nm)$, which is much worse than what it should be, $O(n + m)$.

We can solve this problem with a double-ended queue, also known as a **deque**. The important feature of a deque is that you can add and remove elements from the beginning or end in constant time. To see how it is implemented, see http://thinkcomplex.com/deque.

Python provides a `deque` in the `collections` module, so we can import it like this:

```
from collections import deque
```

And we can use it to write an efficient BFS:

```
def reachable_nodes_bfs(G, start):
    seen = set()
    queue = deque([start])
    while queue:
        node = queue.popleft()
        if node not in seen:
            seen.add(node)
            queue.extend(G.neighbors(node))
    return seen
```

www.dbooks.org

The differences are:

- I replaced the list called `stack` with a deque called `queue`.
- I replaced `pop` with `popleft`, which removes and returns the leftmost element of the queue.

This version is back to being $O(n + m)$. Now we're ready to find shortest paths.

3.10 Dijkstra's algorithm

Edsger W. Dijkstra was a Dutch computer scientist who invented an efficient shortest-path algorithm (see http://thinkcomplex.com/dijk). He also invented the semaphore, which is a data structure used to coordinate programs that communicate with each other (see http://thinkcomplex.com/sem and Downey, *The Little Book of Semaphores*).

Dijkstra is famous (and notorious) as the author of a series of essays on computer science. Some, like "A Case against the GO TO Statement", had a profound effect on programming practice. Others, like "On the Cruelty of Really Teaching Computing Science", are entertaining in their cantankerousness, but less effective.

Dijkstra's algorithm solves the "single source shortest path problem", which means that it finds the minimum distance from a given "source" node to every other node in the graph (or at least every connected node).

I'll present a simplified version of the algorithm that considers all edges the same length. The more general version works with any non-negative edge lengths.

The simplified version is similar to the breadth-first search in the previous section except that we replace the set called `seen` with a dictionary called `dist`, which maps from each node to its distance from the source:

```
def shortest_path_dijkstra(G, source):
    dist = {source: 0}
    queue = deque([source])
    while queue:
        node = queue.popleft()
        new_dist = dist[node] + 1

        neighbors = set(G[node]).difference(dist)
        for n in neighbors:
            dist[n] = new_dist

        queue.extend(neighbors)
    return dist
```

Here's how it works:

- Initially, `queue` contains a single element, `source`, and `dist` maps from `source` to distance 0 (which is the distance from `source` to itself).
- Each time through the loop, we use `popleft` to select the next node in the queue.
- Next we find all neighbors of `node` that are not already in `dist`.
- Since the distance from `source` to `node` is `dist[node]`, the distance to any of the undiscovered neighbors is `dist[node]+1`.
- For each neighbor, we add an entry to `dist`, then we add the neighbors to the queue.

This algorithm only works if we use BFS, not DFS. To see why, consider this:

1. The first time through the loop `node` is `source`, and `new_dist` is 1. So the neighbors of `source` get distance 1 and they go in the queue.
2. When we process the neighbors of `source`, all of *their* neighbors get distance 2. We know that none of them can have distance 1, because if they did, we would have discovered them during the first iteration.

www.dbooks.org

3. Similarly, when we process the nodes with distance 2, we give their neighbors distance 3. We know that none of them can have distance 1 or 2, because if they did, we would have discovered them during a previous iteration.

And so on. If you are familiar with proof by induction, you can see where this is going.

But this argument only works if we process all nodes with distance 1 before we start processing nodes with distance 2, and so on. And that's exactly what BFS does.

In the exercises at the end of this chapter, you'll write a version of Dijkstra's algorithm using DFS, so you'll have a chance to see what goes wrong.

3.11 Exercises

Exercise 3.1 In a ring lattice, every node has the same number of neighbors. The number of neighbors is called the **degree** of the node, and a graph where all nodes have the same degree is called a **regular graph**.

All ring lattices are regular, but not all regular graphs are ring lattices. In particular, if `k` is odd, we can't construct a ring lattice, but we might be able to construct a regular graph.

Write a function called `make_regular_graph` that takes `n` and `k` and returns a regular graph that contains `n` nodes, where every node has `k` neighbors. If it's not possible to make a regular graph with the given values of `n` and `k`, the function should raise a `ValueError`.

Exercise 3.2 My implementation of `reachable_nodes_bfs` is efficient in the sense that it is in $O(n + m)$, but it incurs a lot of overhead adding nodes to the queue and removing them. NetworkX provides a simple, fast implementation of BFS, available from the NetworkX repository on GitHub at http://thinkcomplex.com/connx.

Here is a version I modified to return a set of nodes:

```
def plain_bfs(G, start):
    seen = set()
    nextlevel = {start}
    while nextlevel:
        thislevel = nextlevel
        nextlevel = set()
        for v in thislevel:
            if v not in seen:
                seen.add(v)
                nextlevel.update(G[v])
    return seen
```

Compare this function to `reachable_nodes_bfs` and see which is faster. Then see if you can modify this function to implement a faster version of `shortest_path_dijkstra`.

Exercise 3.3 The following implementation of BFS contains two performance errors. What are they? What is the actual order of growth for this algorithm?

```
def bfs(G, start):
    visited = set()
    queue = [start]
    while len(queue):
        curr_node = queue.pop(0)    # Dequeue
        visited.add(curr_node)

        # Enqueue non-visited and non-enqueued children
        queue.extend(c for c in G[curr_node]
                     if c not in visited and c not in queue)
    return visited
```

Exercise 3.4 In Section 3.10, I claimed that Dijkstra's algorithm does not work unless it uses BFS. Write a version of `shortest_path_dijkstra` that uses DFS and test it on a few examples to see what goes wrong.

Exercise 3.5 A natural question about the Watts and Strogatz paper is whether the small world phenomenon is specific to their generative model or whether other similar models yield the same qualitative result (high clustering and low path lengths).

www.dbooks.org

To answer this question, choose a variation of the Watts and Strogatz model and repeat the experiment. There are two kinds of variation you might consider:

- Instead of starting with a regular graph, start with another graph with high clustering. For example, you could put nodes at random locations in a 2-D space and connect each node to its nearest k neighbors.
- Experiment with different kinds of rewiring.

If a range of similar models yield similar behavior, we say that the results of the paper are **robust**.

Exercise 3.6 Dijkstra's algorithm solves the "single source shortest path" problem, but to compute the characteristic path length of a graph, we actually want to solve the "all pairs shortest path" problem.

Of course, one option is to run Dijkstra's algorithm n times, once for each starting node. And for some applications, that's probably good enough. But there are are more efficient alternatives.

Find an algorithm for the all-pairs shortest path problem and implement it. See http://thinkcomplex.com/short.

Compare the run time of your implementation with running Dijkstra's algorithm n times. Which algorithm is better in theory? Which is better in practice? Which one does NetworkX use?

www.dbooks.org

Chapter 4

Scale-free networks

In this chapter, we'll work with data from an online social network, and use a Watts-Strogatz graph to model it. The WS model has characteristics of a small world network, like the data, but it has low variability in the number of neighbors from node to node, unlike the data.

This discrepancy is the motivation for a network model developed by Barabási and Albert. The BA model captures the observed variability in the number of neighbors, and it has one of the small world properties, short path lengths, but it does not have the high clustering of a small world network.

The chapter ends with a discussion of WS and BA graphs as explanatory models for small world networks.

The code for this chapter is in `chap04.ipynb` in the respository for this book. More information about working with the code is in Section 0.3.

4.1 Social network data

Watts-Strogatz graphs are intended to model networks in the natural and social sciences. In their original paper, Watts and Strogatz looked at the network of film actors (connected if they have appeared in a movie together); the electrical power grid in the western United States; and the network of neurons in the brain of the roundworm *C. elegans*. They found that all of

these networks had the high connectivity and low path lengths characteristic of small world graphs.

In this section we'll perform the same analysis with a different dataset, a set of Facebook users and their friends. If you are not familiar with Facebook, users who are connected to each other are called "friends", regardless of the nature of their relationship in the real world.

I'll use data from the Stanford Network Analysis Project (SNAP), which shares large datasets from online social networks and other sources. Specifically, I'll use their Facebook data[1], which includes 4039 users and 88,234 friend relationships among them. This dataset is in the repository for this book, but it is also available from the SNAP website at http://thinkcomplex.com/snap.

The data file contains one line per edge, with users identified by integers from 0 to 4038. Here's the code that reads the file:

```
def read_graph(filename):
    G = nx.Graph()
    array = np.loadtxt(filename, dtype=int)
    G.add_edges_from(array)
    return G
```

NumPy provides a function called `loadtext` that reads the given file and returns the contents as a NumPy array. The parameter `dtype` indicates that the "data type" of the array is `int`.

Then we use `add_edges_from` to iterate the rows of the array and make edges. Here are the results:

```
>>> fb = read_graph('facebook_combined.txt.gz')
>>> n = len(fb)
>>> m = len(fb.edges())
>>> n, m
(4039, 88234)
```

The node and edge counts are consistent with the documentation of the dataset.

[1] J. McAuley and J. Leskovec. Learning to Discover Social Circles in Ego Networks. NIPS, 2012.

www.dbooks.org

Now we can check whether this dataset has the characteristics of a small world graph: high clustering and low path lengths.

In Section 3.5 we wrote a function to compute the network average clustering coefficient. NetworkX provides a function called `average_clustering`, which does the same thing a little faster.

But for larger graphs, they are both too slow, taking time proportional to nk^2, where n is the number of nodes and k is the number of neighbors each node is connected to.

Fortunately, NetworkX provides a function that estimates the clustering coefficient by random sampling. You can invoke it like this:

```
from networkx.algorithms.approximation import average_clustering
average_clustering(G, trials=1000)
```

The following function does something similar for path lengths.

```
def sample_path_lengths(G, nodes=None, trials=1000):
    if nodes is None:
        nodes = list(G)
    else:
        nodes = list(nodes)

    pairs = np.random.choice(nodes, (trials, 2))
    lengths = [nx.shortest_path_length(G, *pair)
               for pair in pairs]
    return lengths
```

`G` is a graph, `nodes` is the list of nodes to sample from, and `trials` is the number of random paths to sample. If `nodes` is `None`, we sample from the entire graph.

`pairs` is a NumPy array of randomly chosen nodes with one row for each trial and two columns.

The list comprehension enumerates the rows in the array and computes the shortest distance between each pair of nodes. The result is a list of path lengths.

`estimate_path_length` generates a list of random path lengths and returns their mean:

```
def estimate_path_length(G, nodes=None, trials=1000):
    return np.mean(sample_path_lengths(G, nodes, trials))
```

I'll use `average_clustering` to compute C:

```
C = average_clustering(fb)
```

And `estimate_path_lengths` to compute L:

```
L = estimate_path_lengths(fb)
```

The clustering coefficient is about 0.61, which is high, as we expect if this network has the small world property.

And the average path is 3.7, which is quite short in a network of more than 4000 users. It's a small world after all.

Now let's see if we can construct a WS graph that has the same characteristics as this network.

4.2 WS Model

In the Facebook dataset, the average number of edges per node is about 22. Since each edge is connected to two nodes, the average degree is twice the number of edges per node:

```
>>> k = int(round(2*m/n))
>>> k
44
```

We can make a WS graph with `n=4039` and `k=44`. When `p=0`, we get a ring lattice.

```
lattice = nx.watts_strogatz_graph(n, k, 0)
```

In this graph, clustering is high: `C` is 0.73, compared to 0.61 in the dataset. But `L` is 46, much higher than in the dataset!

www.dbooks.org

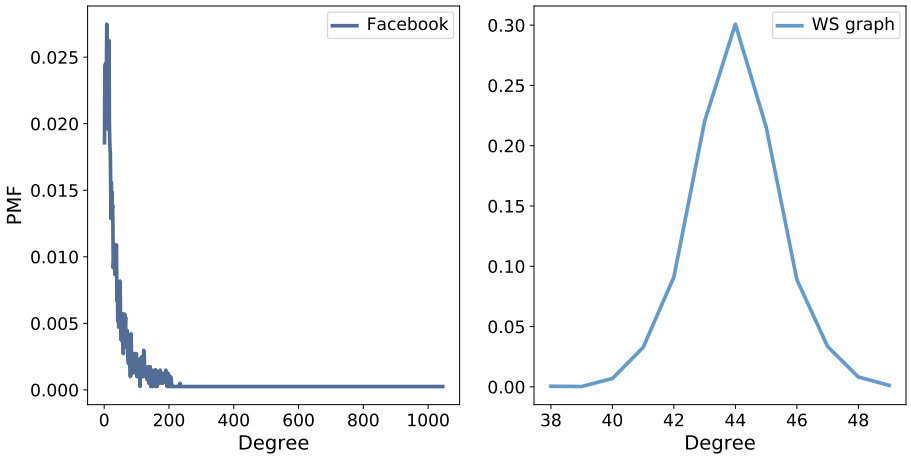


Figure 4.1: PMF of degree in the Facebook dataset and in the WS model.

With `p=1` we get a random graph:

```
random_graph = nx.watts_strogatz_graph(n, k, 1)
```

In the random graph, `L` is 2.6, even shorter than in the dataset (3.7), but `C` is only 0.011, so that's no good.

By trial and error, we find that when `p=0.05` we get a WS graph with high clustering and low path length:

```
ws = nx.watts_strogatz_graph(n, k, 0.05, seed=15)
```

In this graph `C` is 0.63, a bit higher than in the dataset, and `L` is 3.2, a bit lower than in the dataset. So this graph models the small world characteristics of the dataset well.

So far, so good.

4.3 Degree

If the WS graph is a good model for the Facebook network, it should have the same average degree across nodes, and ideally the same variance in degree.

This function returns a list of degrees in a graph, one for each node:

```
def degrees(G):
    return [G.degree(u) for u in G]
```

The mean degree in model is 44, which is close to the mean degree in the dataset, 43.7.

However, the standard deviation of degree in the model is 1.5, which is not close to the standard deviation in the dataset, 52.4. Oops.

What's the problem? To get a better view, we have to look at the **distribution** of degrees, not just the mean and standard deviation.

I'll represent the distribution of degrees with a `Pmf` object, which is defined in the `thinkstats2` module. `Pmf` stands for "probability mass function"; if you are not familiar with this concept, you might want to read Chapter 3 of *Think Stats, 2nd edition* at http://thinkcomplex.com/ts2.

Briefly, a `Pmf` maps from values to their probabilities. A `Pmf` of degrees is a mapping from each possible degree, d, to the fraction of nodes with degree d.

As an example, I'll construct a graph with nodes 1, 2, and 3 connected to a central node, 0:

```
G = nx.Graph()
G.add_edge(1, 0)
G.add_edge(2, 0)
G.add_edge(3, 0)
nx.draw(G)
```

Here's the list of degrees in this graph:

```
>>> degrees(G)
[3, 1, 1, 1]
```

Node 0 has degree 3, the others have degree 1. Now I can make a `Pmf` that represents this degree distribution:

```
>>> from thinkstats2 import Pmf
>>> Pmf(degrees(G))
Pmf({1: 0.75, 3: 0.25})
```

www.dbooks.org

The result is a `Pmf` object that maps from each degree to a fraction or probability. In this example, 75% of the nodes have degree 1 and 25% have degree 3.

Now we can make a `Pmf` that contains node degrees from the dataset, and compute the mean and standard deviation:

```
>>> from thinkstats2 import Pmf
>>> pmf_fb = Pmf(degrees(fb))
>>> pmf_fb.Mean(), pmf_fb.Std()
(43.691, 52.414)
```

And the same for the WS model:

```
>>> pmf_ws = Pmf(degrees(ws))
>>> pmf_ws.mean(), pmf_ws.std()
(44.000, 1.465)
```

We can use the `thinkplot` module to plot the results:

```
thinkplot.Pdf(pmf_fb, label='Facebook')
thinkplot.Pdf(pmf_ws, label='WS graph')
```

Figure 4.1 shows the two distributions. They are very different.

In the WS model, most users have about 44 friends; the minimum is 38 and the maximum is 50. That's not much variation. In the dataset, there are many users with only 1 or 2 friends, but one has more than 1000!

Distributions like this, with many small values and a few very large values, are called **heavy-tailed**.

4.4 Heavy-tailed distributions

Heavy-tailed distributions are a common feature in many areas of complexity science and they will be a recurring theme of this book.

We can get a clearer picture of a heavy-tailed distribution by plotting it on a log-log axis, as shown in Figure 4.2. This transformation emphasizes the tail of the distribution; that is, the probabilities of large values.

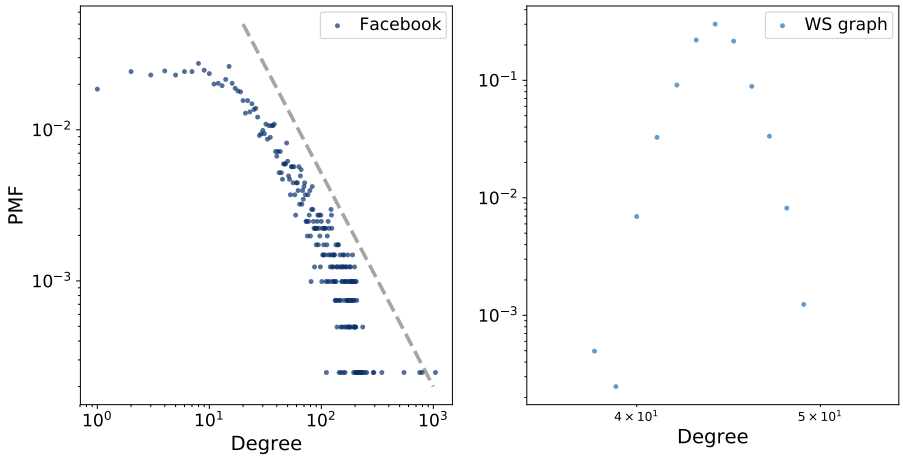


Figure 4.2: PMF of degree in the Facebook dataset and in the WS model, on a log-log scale.

Under this transformation, the data fall approximately on a straight line, which suggests that there is a **power law** relationship between the largest values in the distribution and their probabilities. Mathematically, a distribution obeys a power law if

$$\text{PMF}(k) \sim k^{-\alpha}$$

where $\text{PMF}(k)$ is the fraction of nodes with degree k, α is a parameter, and the symbol $\sim$ indicates that the PMF is asymptotic to $k^{-\alpha}$ as k increases.

If we take the log of both sides, we get

$$\log \text{PMF}(k) \sim -\alpha \log k$$

So if a distribution follows a power law and we plot $\text{PMF}(k)$ versus k on a log-log scale, we expect a straight line with slope $-\alpha$, at least for large values of k.

All power law distributions are heavy-tailed, but there are other heavy-tailed distributions that don't follow a power law. We will see more examples soon.

But first, we have a problem: the WS model has the high clustering and low path length we see in the data, but the degree distribution doesn't resemble

www.dbooks.org

the data at all. This discrepancy is the motivation for our next topic, the Barabási-Albert model.

4.5 Barabási-Albert model

In 1999 Barabási and Albert published a paper, "Emergence of Scaling in Random Networks", that characterizes the structure of several real-world networks, including graphs that represent the interconnectivity of movie actors, web pages, and elements in the electrical power grid in the western United States. You can download the paper from http://thinkcomplex.com/barabasi.

They measure the degree of each node and compute PMF(k), the probability that a vertex has degree k. Then they plot PMF(k) versus k on a log-log scale. The plots fit a straight line, at least for large values of k, so Barabási and Albert conclude that these distributions are heavy-tailed.

They also propose a model that generates graphs with the same property. The essential features of the model, which distinguish it from the WS model, are:

Growth: Instead of starting with a fixed number of vertices, the BA model starts with a small graph and adds vertices one at a time.

Preferential attachment: When a new edge is created, it is more likely to connect to a vertex that already has a large number of edges. This "rich get richer" effect is characteristic of the growth patterns of some real-world networks.

Finally, they show that graphs generated by the Barabási-Albert (BA) model have a degree distribution that obeys a power law.

Graphs with this property are sometimes called **scale-free networks**, for reasons I won't explain; if you are curious, you can read more at http://thinkcomplex.com/scale.

NetworkX provides a function that generates BA graphs. We will use it first; then I'll show you how it works.

```
ba = nx.barabasi_albert_graph(n=4039, k=22)
```

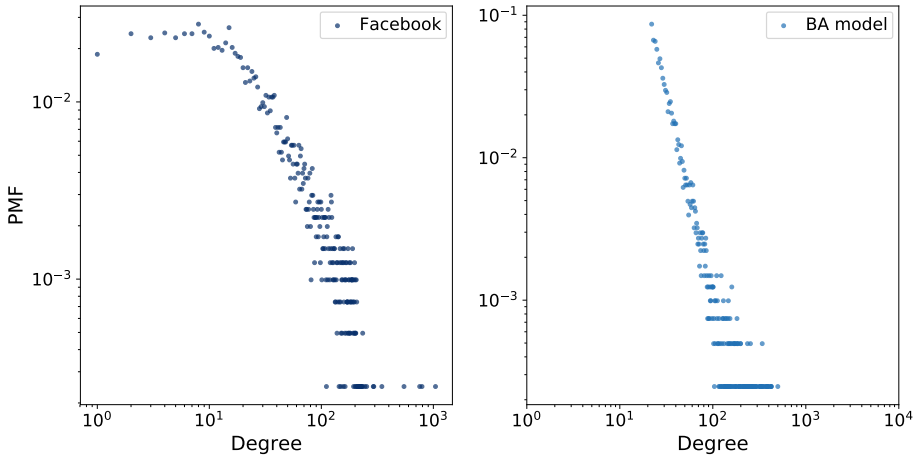


Figure 4.3: PMF of degree in the Facebook dataset and in the BA model, on a log-log scale.

The parameters are `n`, the number of nodes to generate, and `k`, the number of edges each node starts with when it is added to the graph. I chose `k=22` because that is the average number of edges per node in the dataset.

The resulting graph has 4039 nodes and 21.9 edges per node. Since every edge is connected to two nodes, the average degree is 43.8, very close to the average degree in the dataset, 43.7.

And the standard deviation of degree is 40.9, which is a bit less than in the dataset, 52.4, but it is much better than what we got from the WS graph, 1.5.

Figure 4.3 shows the degree distributions for the Facebook dataset and the BA model on a log-log scale. The model is not perfect; in particular, it deviates from the data when `k` is less than 10. But the tail looks like a straight line, which suggests that this process generates degree distributions that follow a power law.

So the BA model is better than the WS model at reproducing the degree distribution. But does it have the small world property?

In this example, the average path length, L, is 2.5, which is even more "small world" than the actual network, which has $L = 3.69$. So that's good, although maybe too good.

www.dbooks.org

	Facebook	WS model	BA model
C	0.61	0.63	0.037
L	3.69	3.23	2.51
Mean degree	43.7	44	43.7
Std degree	52.4	1.5	40.1
Power law?	maybe	no	yes

Table 4.1: Characteristics of the Facebook dataset compared to two models.

On the other hand, the clustering coefficient, C, is 0.037, not even close to the value in the dataset, 0.61. So that's a problem.

Table 4.1 summarizes these results. The WS model captures the small world characteristics, but not the degree distribution. The BA model captures the degree distribution, at least approximately, and the average path length, but not the clustering coefficient.

In the exercises at the end of this chapter, you can explore other models intended to capture all of these characteristics.

4.6 Generating BA graphs

In the previous sections we used a NetworkX function to generate BA graphs; now let's see how it works. Here is a version of `barabasi_albert_graph`, with some changes I made to make it easier to read:

```
def barabasi_albert_graph(n, k):

    G = nx.empty_graph(k)
    targets = list(range(k))
    repeated_nodes = []

    for source in range(k, n):
        G.add_edges_from(zip([source]*k, targets))

        repeated_nodes.extend(targets)
        repeated_nodes.extend([source] * k)

        targets = _random_subset(repeated_nodes, k)

    return G
```

The parameters are `n`, the number of nodes we want, and `k`, the number of edges each new node gets (which will turn out to be the average number of edges per node).

We start with a graph that has `k` nodes and no edges. Then we initialize two variables:

> The list of `k` nodes that will be connected to the next node. Initially `targets` contains the original `k` nodes; later it will contain a random subset of nodes.
>
> A list of existing nodes where each node appears once for every edge it is connected to. When we select from `repeated_nodes`, the probability of selecting any node is proportional to the number of edges it has.

Each time through the loop, we add edges from the source to each node in `targets`. Then we update `repeated_nodes` by adding each target once and the new node `k` times.

Finally, we choose a subset of the nodes to be targets for the next iteration. Here's the definition of `_random_subset`:

www.dbooks.org

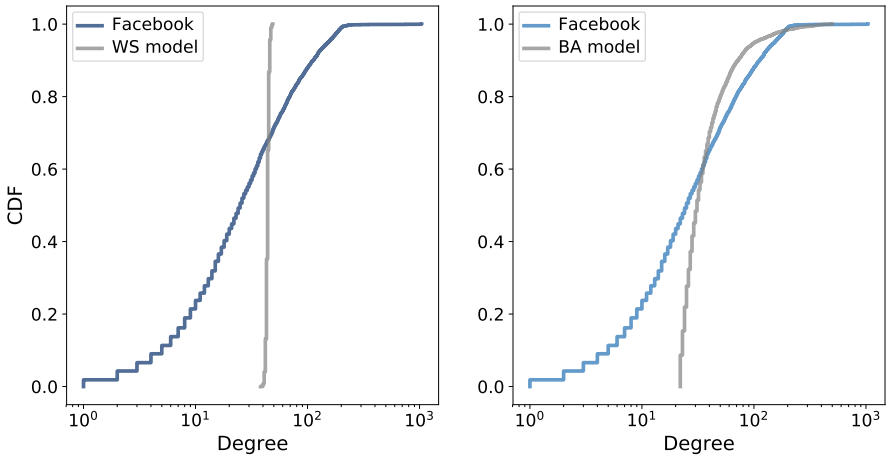


Figure 4.4: CDF of degree in the Facebook dataset along with the WS model (left) and the BA model (right), on a log-x scale.

```
def _random_subset(repeated_nodes, k):
    targets = set()
    while len(targets) < k:
        x = random.choice(repeated_nodes)
        targets.add(x)
    return targets
```

Each time through the loop, `_random_subset` chooses from `repeated_nodes` and adds the chosen node to `targets`. Because `targets` is a set, it automatically discards duplicates, so the loop only exits when we have selected `k` different nodes.

4.7 Cumulative distributions

Figure 4.3 represents the degree distribution by plotting the probability mass function (PMF) on a log-log scale. That's how Barabási and Albert present their results and it is the representation used most often in articles about power law distributions. But it is not the best way to look at data like this.

A better alternative is a **cumulative distribution function** (CDF), which maps from a value, x, to the fraction of values less than or equal to x.

Given a `Pmf`, the simplest way to compute a cumulative probability is to add up the probabilities for values up to and including x:

```
def cumulative_prob(pmf, x):
    ps = [pmf[value] for value in pmf if value<=x]
    return np.sum(ps)
```

For example, given the degree distribution in the dataset, `pmf_fb`, we can compute the fraction of users with 25 or fewer friends:

```
>>> cumulative_prob(pmf_fb, 25)
0.506
```

The result is close to 0.5, which means that the median number of friends is about 25.

CDFs are better for visualization because they are less noisy than PMFs. Once you get used to interpreting CDFs, they provide a clearer picture of the shape of a distribution than PMFs.

The `thinkstats` module provides a class called `Cdf` that represents a cumulative distribution function. We can use it to compute the CDF of degree in the dataset.

```
from thinkstats2 import Cdf
cdf_fb = Cdf(degrees(fb), label='Facebook')
```

And `thinkplot` provides a function called `Cdf` that plots cumulative distribution functions.

```
thinkplot.Cdf(cdf_fb)
```

Figure 4.4 shows the degree CDF for the Facebook dataset along with the WS model (left) and the BA model (right). The x-axis is on a log scale.

Clearly the CDF for the WS model is very different from the CDF from the data. The BA model is better, but still not very good, especially for small values.

In the tail of the distribution (values greater than 100) it looks like the BA model matches the dataset well enough, but it is hard to see. We can get

www.dbooks.org

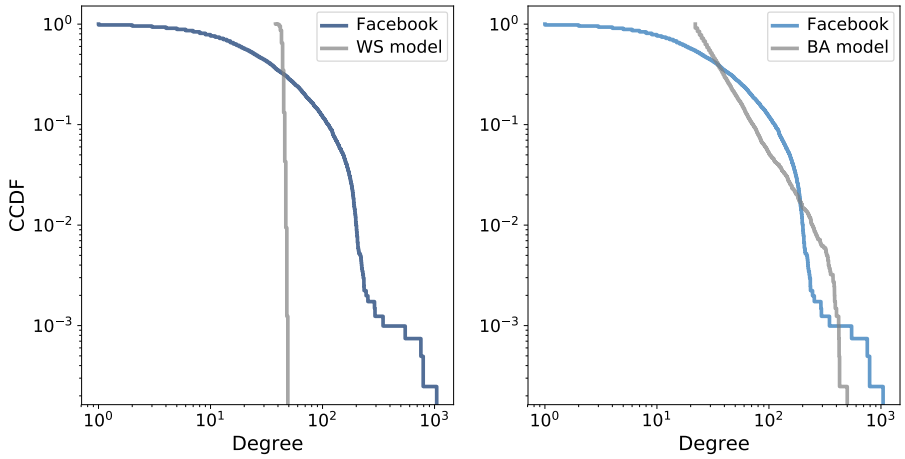


Figure 4.5: Complementary CDF of degree in the Facebook dataset along with the WS model (left) and the BA model (right), on a log-log scale.

a clearer view with one other view of the data: plotting the complementary CDF on a log-log scale.

The **complementary CDF** (CCDF) is defined

$$\text{CCDF}(x) \equiv 1 - \text{CDF}(x)$$

This definition is useful because if the PMF follows a power law, the CCDF also follows a power law:

$$\text{CCDF}(x) \sim \left(\frac{x}{x_m}\right)^{-\alpha}$$

where x_m is the minimum possible value and α is a parameter that determines the shape of the distribution.

Taking the log of both sides yields:

$$\log \text{CCDF}(x) \sim -\alpha(\log x - \log x_m)$$

So if the distribution obeys a power law, we expect the CCDF on a log-log scale to be a straight line with slope $-\alpha$.

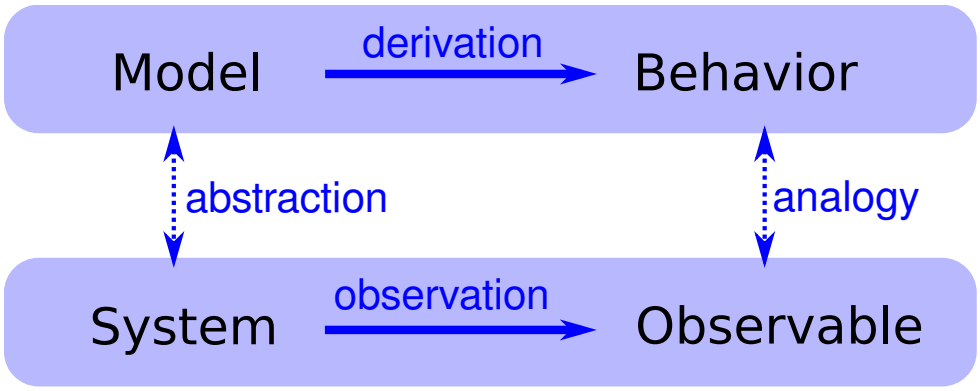


Figure 4.6: The logical structure of an explanatory model.

Figure 4.5 shows the CCDF of degree for the Facebook data, along with the WS model (left) and the BA model (right), on a log-log scale.

With this way of looking at the data, we can see that the BA model matches the tail of the distribution (values above 20) reasonably well. The WS model does not.

4.8 Explanatory models

We started the discussion of networks with Milgram's Small World Experiment, which shows that path lengths in social networks are surprisingly small; hence, "six degrees of separation".

When we see something surprising, it is natural to ask "Why?" but sometimes it's not clear what kind of answer we are looking for. One kind of answer is an **explanatory model** (see Figure 4.6). The logical structure of an explanatory model is:

1. We see something in the world, called the target system, S, we see something observable, O, that warrants explanation.

2. We construct a model, M, that is analogous to the system; that is, there is a correspondence between the elements of the model and the elements of the system.

www.dbooks.org

3. By simulation or mathematical derivation, we show that the model exhibits a behavior, B, that is analogous to O.

4. We conclude that S exhibits O *because* S is similar to M, M exhibits B, and B is similar to O.

At its core, this is an argument by analogy, which says that if two things are similar in some ways, they are likely to be similar in other ways.

Argument by analogy can be useful, and explanatory models can be satisfying, but they do not constitute a proof in the mathematical sense of the word.

Remember that all models leave out, or "abstract away", details that we think are unimportant. For any system there are many possible models that include or ignore different features. And there might be models that exhibit different behaviors that are similar to O in different ways. In that case, which model explains O?

The small world phenomenon is an example: the Watts-Strogatz (WS) model and the Barabási-Albert (BA) model both exhibit elements of small world behavior, but they offer different explanations:

- The WS model suggests that social networks are "small" because they include both strongly-connected clusters and "weak ties" that connect clusters (see http://thinkcomplex.com/weak).

- The BA model suggests that social networks are small because they include nodes with high degree that act as hubs, and that hubs grow, over time, due to preferential attachment.

As is often the case in young areas of science, the problem is not that we have no explanations, but too many.

4.9 Exercises

Exercise 4.1 In Section 4.8 we discussed two explanations for the small world phenomenon, "weak ties" and "hubs". Are these explanations compatible; that is, can they both be right? Which do you find more satisfying as an explanation, and why?

Is there data you could collect, or experiments you could perform, that would provide evidence in favor of one model over the other?

Choosing among competing models is the topic of Thomas Kuhn's essay, "Objectivity, Value Judgment, and Theory Choice", which you can read at http://thinkcomplex.com/kuhn.

What criteria does Kuhn propose for choosing among competing models? Do these criteria influence your opinion about the WS and BA models? Are there other criteria you think should be considered?

Exercise 4.2 NetworkX provides a function called `powerlaw_cluster_graph` that implements the "Holme and Kim algorithm for growing graphs with powerlaw degree distribution and approximate average clustering". Read the documentation of this function (http://thinkcomplex.com/hk) and see if you can use it to generate a graph that has the same number of nodes as the Facebook dataset, the same average degree, and the same clustering coefficient. How does the degree distribution in the model compare to the actual distribution?

Exercise 4.3 Data files from the Barabási and Albert paper are available from http://thinkcomplex.com/netdata. Their actor collaboration data is included in the repository for this book in a file named `actor.dat.gz`. The following function reads the file and builds the graph.

```
import gzip

def read_actor_network(filename, n=None):
    G = nx.Graph()
    with gzip.open(filename) as f:
        for i, line in enumerate(f):
            nodes = [int(x) for x in line.split()]
            G.add_edges_from(thinkcomplexity.all_pairs(nodes))
            if n and i >= n:
                break
    return G
```

Compute the number of actors in the graph and the average degree. Plot the PMF of degree on a log-log scale. Also plot the CDF of degree on a log-x

www.dbooks.org

scale, to see the general shape of the distribution, and on a log-log scale, to see whether the tail follows a power law.

Note: The actor network is not connected, so you might want to use `nx.connected_component_s` to find connected subsets of the nodes.

www.dbooks.org

Chapter 5

Cellular Automatons

A **cellular automaton** (CA) is a model of a world with very simple physics. "Cellular" means that the world is divided into discrete chunks, called cells. An "automaton" is a machine that performs computations — it could be a real machine, but more often the "machine" is a mathematical abstraction or a computer simulation.

This chapter presents experiments Stephen Wolfram performed in the 1980s, showing that some cellular automatons display surprisingly complicated behavior, including the ability to perform arbitrary computations.

I discuss implications of these results, and at the end of the chapter I suggest methods for implementing CAs efficiently in Python.

The code for this chapter is in `chap05.ipynb` in the repository for this book. More information about working with the code is in Section 0.3.

5.1 A simple CA

Cellular automatons[1] are governed by rules that determine how the state of the cells changes over time.

[1] You might also see the plural "automata".

As a trivial example, consider a cellular automaton (CA) with a single cell. The state of the cell during time step i is an integer, x_i. As an initial condition, suppose $x_0 = 0$.

Now all we need is a rule. Arbitrarily, I'll pick $x_{i+1} = x_i + 1$, which says that during each time step, the state of the CA gets incremented by 1. So this CA performs a simple calculation: it counts.

But this CA is atypical; normally the number of possible states is finite. As an example, suppose a cell can only have one of two states, 0 or 1. For a 2-state CA, we could write a rule like $x_{i+1} = (x_i + 1)\%2$, where % is the remainder (or modulus) operator.

The behavior of this CA is simple: it blinks. That is, the state of the cell switches between 0 and 1 during each time step.

Most CAs are **deterministic**, which means that rules do not have any random elements; given the same initial state, they always produce the same result. But some CAs are nondeterministic; we will see examples later.

The CA in this section has only one cell, so we can think of it as zero-dimensional. In the rest of this chapter, we explore one-dimensional (1-D) CAs; in the next chapter we explore two-dimensional CAs.

5.2 Wolfram's experiment

In the early 1980s Stephen Wolfram published a series of papers presenting a systematic study of 1-D CAs. He identified four categories of behavior, each more interesting than the last. You can read one of these papers, "Statistical mechanics of cellular automata," at http://thinkcomplex.com/ca.

In Wolfram's experiments, the cells are arranged in a lattice (which you might remember from Section 3.2) where each cell is connected to two neighbors. The lattice can be finite, infinite, or arranged in a ring.

The rules that determine how the system evolves in time are based on the notion of a "neighborhood", which is the set of cells that determines the next state of a given cell. Wolfram's experiments use a 3-cell neighborhood: the cell itself and its two neighbors.

www.dbooks.org

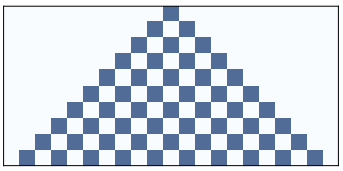


Figure 5.1: Rule 50 after 10 time steps.

In these experiments, the cells have two states, denoted 0 and 1 or "off" and "on". A rule can be summarized by a table that maps from the state of the neighborhood (a tuple of three states) to the next state of the center cell. The following table shows an example:

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

The first row shows the eight states a neighborhood can be in. The second row shows the state of the center cell during the next time step. As a concise encoding of this table, Wolfram suggested reading the bottom row as a binary number; because 00110010 in binary is 50 in decimal, Wolfram calls this CA "Rule 50".

Figure 5.1 shows the effect of Rule 50 over 10 time steps. The first row shows the state of the system during the first time step; it starts with one cell "on" and the rest "off". The second row shows the state of the system during the next time step, and so on.

The triangular shape in the figure is typical of these CAs; is it a consequence of the shape of the neighborhood. In one time step, each cell influences the state of one neighbor in either direction. During the next time step, that influence can propagate one more cell in each direction. So each cell in the past has a "triangle of influence" that includes all of the cells that can be affected by it.

5.3 Classifying CAs

How many of these CAs are there?

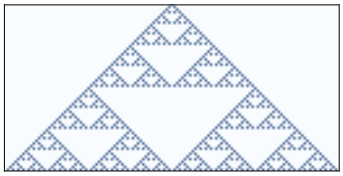


Figure 5.2: Rule 18 after 64 steps.

Since each cell is either on or off, we can specify the state of a cell with a single bit. In a neighborhood with three cells, there are 8 possible configurations, so there are 8 entries in the rule tables. And since each entry contains a single bit, we can specify a table using 8 bits. With 8 bits, we can specify 256 different rules.

One of Wolfram's first experiments with CAs was to test all 256 possibilities and classify them.

Examining the results visually, he proposed that the behavior of CAs can be grouped into four classes. Class 1 contains the simplest (and least interesting) CAs, the ones that evolve from almost any starting condition to the same uniform pattern. As a trivial example, Rule 0 always generates an empty pattern after one time step.

Rule 50 is an example of Class 2. It generates a simple pattern with nested structure, that is, a pattern that contains many smaller versions of itself. Rule 18 makes the nested structure is even clearer; Figure 5.2 shows what it looks like after 64 steps.

This pattern resembles the Sierpiński triangle, which you can read about at http://thinkcomplex.com/sier.

Some Class 2 CAs generate patterns that are intricate and pretty, but compared to Classes 3 and 4, they are relatively simple.

www.dbooks.org

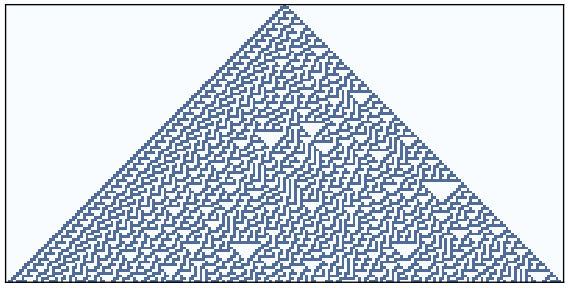


Figure 5.3: Rule 30 after 100 time steps.

5.4 Randomness

Class 3 contains CAs that generate randomness. Rule 30 is an example; Figure 5.3 shows what it looks like after 100 time steps.

Along the left side there is an apparent pattern, and on the right side there are triangles in various sizes, but the center seems quite random. In fact, if you take the center column and treat it as a sequence of bits, it is hard to distinguish from a truly random sequence. It passes many of the statistical tests people use to test whether a sequence of bits is random.

Programs that produce random-seeming numbers are called **pseudo-random number generators** (PRNGs). They are not considered truly random because:

- Many of them produce sequences with regularities that can be detected statistically. For example, the original implementation of `rand` in the C library used a linear congruential generator that yielded sequences with easily detectable serial correlations.

- Any PRNG that uses a finite amount of state (that is, storage) will eventually repeat itself. One of the characteristics of a generator is the **period** of this repetition.

- The underlying process is fundamentally deterministic, unlike some physical processes, like radioactive decay and thermal noise, that are considered to be fundamentally random.

Modern PRNGs produce sequences that are statistically indistinguishable from random, and they can be implemented with periods so long that the universe will collapse before they repeat. The existence of these generators raises the question of whether there is any real difference between a good quality pseudo-random sequence and a sequence generated by a "truly" random process. In *A New Kind of Science*, Wolfram argues that there is not (pages 315–326).

5.5 Determinism

The existence of Class 3 CAs is surprising. To explain how surprising, let me start with philosophical **determinism** (see http://thinkcomplex.com/deter). Many philosophical stances are hard to define precisely because they come in a variety of flavors. I often find it useful to define them with a list of statements ordered from weak to strong:

D1: Deterministic models can make accurate predictions for some physical systems.

D2: Many physical systems can be modeled by deterministic processes, but some are intrinsically random.

D3: All events are caused by prior events, but many physical systems are nevertheless fundamentally unpredictable.

D4: All events are caused by prior events, and can (at least in principle) be predicted.

My goal in constructing this range is to make D1 so weak that virtually everyone would accept it, D4 so strong that almost no one would accept it, with intermediate statements that some people accept.

The center of mass of world opinion swings along this range in response to historical developments and scientific discoveries. Prior to the scientific revolution, many people regarded the working of the universe as fundamentally

www.dbooks.org

unpredictable or controlled by supernatural forces. After the triumphs of Newtonian mechanics, some optimists came to believe something like D4; for example, in 1814 Pierre-Simon Laplace wrote:

> We may regard the present state of the universe as the effect of its past and the cause of its future. An intellect which at a certain moment would know all forces that set nature in motion, and all positions of all items of which nature is composed, if this intellect were also vast enough to submit these data to analysis, it would embrace in a single formula the movements of the greatest bodies of the universe and those of the tiniest atom; for such an intellect nothing would be uncertain and the future just like the past would be present before its eyes.

This "intellect" is now called "Laplace's Demon". See http://thinkcomplex.com/demon. The word "demon" in this context has the sense of "spirit", with no implication of evil.

Discoveries in the 19th and 20th centuries gradually dismantled Laplace's hope. Thermodynamics, radioactivity, and quantum mechanics posed successive challenges to strong forms of determinism.

In the 1960s chaos theory showed that in some deterministic systems prediction is only possible over short time scales, limited by precision in the measurement of initial conditions.

Most of these systems are continuous in space (if not time) and nonlinear, so the complexity of their behavior is not entirely surprising. Wolfram's demonstration of complex behavior in simple cellular automatons is more surprising — and disturbing, at least to a deterministic world view.

So far I have focused on scientific challenges to determinism, but the longest-standing objection is the apparent conflict between determinism and human free will. Complexity science provides a possible resolution of this conflict; I'll come back to this topic in Section 10.6.

5.6 Spaceships

The behavior of Class 4 CAs is even more surprising. Several 1-D CAs, most notably Rule 110, are **Turing complete**, which means that they can compute

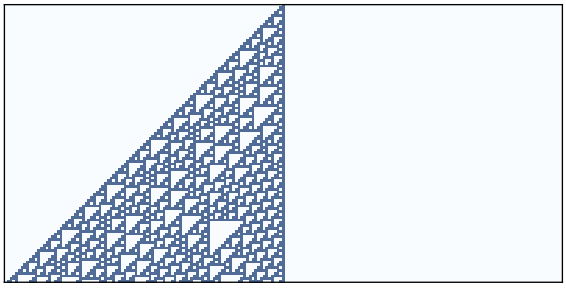


Figure 5.4: Rule 110 after 100 time steps.

any computable function. This property, also called **universality**, was proved by Matthew Cook in 1998. See http://thinkcomplex.com/r110.

Figure 5.4 shows what Rule 110 looks like with an initial condition of a single cell and 100 time steps. At this time scale it is not apparent that anything special is going on. There are some regular patterns but also some features that are hard to characterize.

Figure 5.5 shows a bigger picture, starting with a random initial condition and 600 time steps:

After about 100 steps the background settles into a simple repeating pattern, but there are a number of persistent structures that appear as disturbances in the background. Some of these structures are stable, so they appear as vertical lines. Others translate in space, appearing as diagonals with different slopes, depending on how many time steps they take to shift by one column. These structures are called **spaceships**.

Collisions between spaceships yield different results depending on the types of the spaceships and the phase they are in when they collide. Some collisions annihilate both ships; others leave one ship unchanged; still others yield one or more ships of different types.

These collisions are the basis of computation in a Rule 110 CA. If you think of spaceships as signals that propagate through space, and collisions as gates

www.dbooks.org

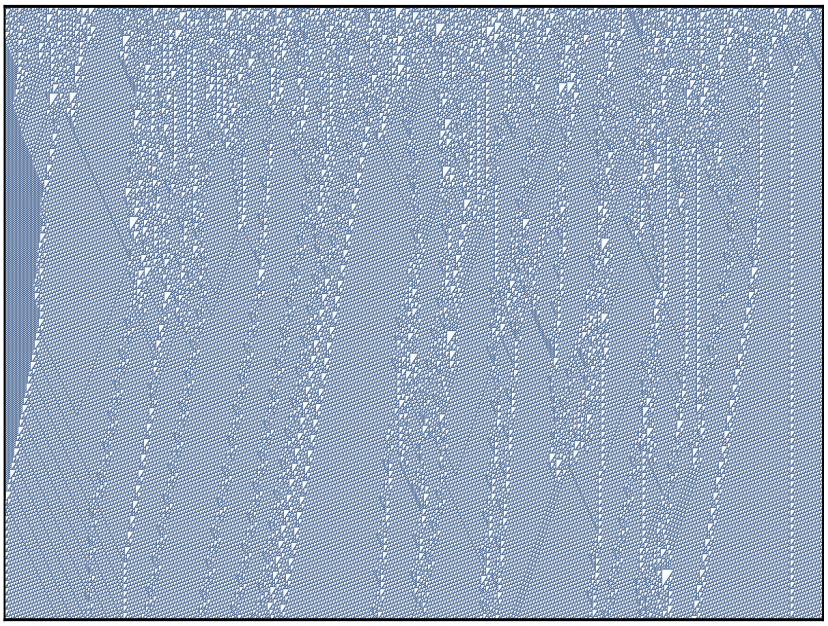


Figure 5.5: Rule 110 with random initial conditions and 600 time steps.

that compute logical operations like AND and OR, you can see what it means for a CA to perform a computation.

5.7 Universality

To understand universality, we have to understand computability theory, which is about models of computation and what they compute.

One of the most general models of computation is the Turing machine, which is an abstract computer proposed by Alan Turing in 1936. A Turing machine is a 1-D CA, infinite in both directions, augmented with a read-write head. At any time, the head is positioned over a single cell. It can read the state of that cell (usually there are only two states) and it can write a new value into the cell.

In addition, the machine has a register, which records the state of the machine (one of a finite number of states), and a table of rules. For each machine state and cell state, the table specifies an action. Actions include modifying the cell the head is over and moving one cell to the left or right.

A Turing machine is not a practical design for a computer, but it models common computer architectures. For a given program running on a real computer, it is possible (at least in principle) to construct a Turing machine that performs an equivalent computation.

The Turing machine is useful because it is possible to characterize the set of functions that can be computed by a Turing machine, which is what Turing did. Functions in this set are called "Turing computable".

To say that a Turing machine can compute any Turing-computable function is a tautology: it is true by definition. But Turing-computability is more interesting than that.

It turns out that just about every reasonable model of computation anyone has come up with is "Turing complete"; that is, it can compute exactly the same set of functions as the Turing machine. Some of these models, like lamdba calculus, are very different from a Turing machine, so their equivalence is surprising.

www.dbooks.org

This observation led to the Church-Turing Thesis, which is the claim that these definitions of computability capture something essential that is independent of any particular model of computation.

The Rule 110 CA is yet another model of computation, and remarkable for its simplicity. That it, too, turns out to be Turing complete lends support to the Church-Turing Thesis.

In *A New Kind of Science*, Wolfram states a variation of this thesis, which he calls the "principle of computational equivalence" (see http://thinkcomplex.com/equiv):

> Almost all processes that are not obviously simple can be viewed as computations of equivalent sophistication.
>
> More specifically, the principle of computational equivalence says that systems found in the natural world can perform computations up to a maximal ("universal") level of computational power, and that most systems do in fact attain this maximal level of computational power. Consequently, most systems are computationally equivalent.

Applying these definitions to CAs, Classes 1 and 2 are "obviously simple". It may be less obvious that Class 3 is simple, but in a way perfect randomness is as simple as perfect order; complexity happens in between. So Wolfram's claim is that Class 4 behavior is common in the natural world, and that almost all systems that manifest it are computationally equivalent.

5.8 Falsifiability

Wolfram holds that his principle is a stronger claim than the Church-Turing thesis because it is about the natural world rather than abstract models of computation. But saying that natural processes "can be viewed as computations" strikes me as a statement about theory choice more than a hypothesis about the natural world.

Also, with qualifications like "almost" and undefined terms like "obviously simple", his hypothesis may be **unfalsifiable**. Falsifiability is an idea from

the philosophy of science, proposed by Karl Popper as a demarcation between scientific hypotheses and pseudoscience. A hypothesis is falsifiable if there is an experiment, at least in the realm of practicality, that would contradict the hypothesis if it were false.

For example, the claim that all life on earth is descended from a common ancestor is falsifiable because it makes specific predictions about similarities in the genetics of modern species (among other things). If we discovered a new species whose DNA was almost entirely different from ours, that would contradict (or at least bring into question) the theory of universal common descent.

On the other hand, "special creation", the claim that all species were created in their current form by a supernatural agent, is unfalsifiable because there is nothing that we could observe about the natural world that would contradict it. Any outcome of any experiment could be attributed to the will of the creator.

Unfalsifiable hypotheses can be appealing because they are impossible to refute. If your goal is never to be proved wrong, you should choose hypotheses that are as unfalsifiable as possible.

But if your goal is to make reliable predictions about the world — and this is at least one of the goals of science — unfalsifiable hypotheses are useless. The problem is that they have no consequences (if they had consequences, they would be falsifiable).

For example, if the theory of special creation were true, what good would it do me to know it? It wouldn't tell me anything about the creator except that he has an "inordinate fondness for beetles" (attributed to J. B. S. Haldane). And unlike the theory of common descent, which informs many areas of science and bioengineering, it would be of no use for understanding the world or acting in it.

5.9 What is this a model of?

Some cellular automatons are primarily mathematical artifacts. They are interesting because they are surprising, or useful, or pretty, or because they provide tools for creating new mathematics (like the Church-Turing thesis).

www.dbooks.org

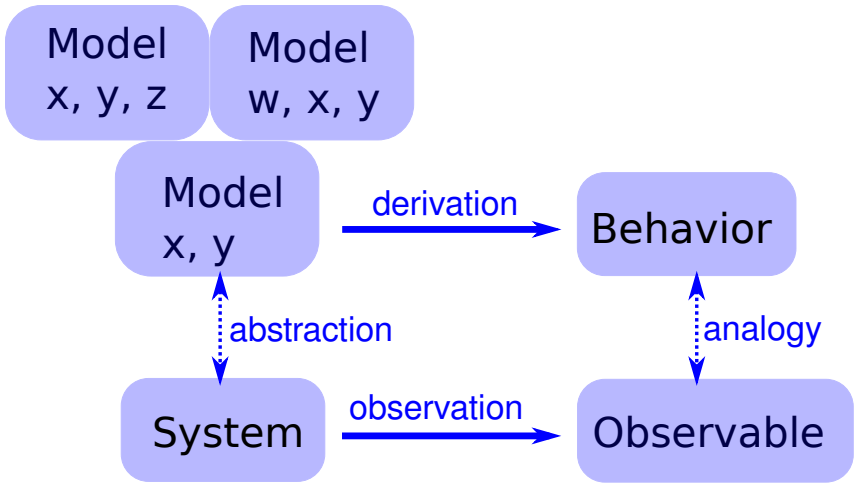


Figure 5.6: The logical structure of a simple physical model.

But it is not clear that they are models of physical systems. And if they are, they are highly abstracted, which is to say that they are not very detailed or realistic.

For example, some species of cone snail produce a pattern on their shells that resembles the patterns generated by cellular automatons (see http://thinkcomplex.com/cone). So it is natural to suppose that a CA is a model of the mechanism that produces patterns on shells as they grow. But, at least initially, it is not clear how the elements of the model (so-called cells, communication between neighbors, rules) correspond to the elements of a growing snail (real cells, chemical signals, protein interaction networks).

For conventional physical models, being realistic is a virtue. If the elements of a model correspond to the elements of a physical system, there is an obvious analogy between the model and the system. In general, we expect a model that is more realistic to make better predictions and to provide more believable explanations.

Of course, this is only true up to a point. Models that are more detailed are harder to work with, and usually less amenable to analysis. At some point, a model becomes so complex that it is easier to experiment with the system.

At the other extreme, simple models can be compelling exactly because they are simple.

Simple models offer a different kind of explanation than detailed models. With a detailed model, the argument goes something like this: "We are interested in physical system S, so we construct a detailed model, M, and show by analysis and simulation that M exhibits a behavior, B, that is similar (qualitatively or quantitatively) to an observation of the real system, O. So why does O happen? Because S is similar to M, and B is similar to O, and we can prove that M leads to B."

With simple models we can't claim that S is similar to M, because it isn't. Instead, the argument goes like this: "There is a set of models that share a common set of features. Any model that has these features exhibits behavior B. If we make an observation, O, that resembles B, one way to explain it is to show that the system, S, has the set of features sufficient to produce B."

For this kind of argument, adding more features doesn't help. Making the model more realistic doesn't make the model more reliable; it only obscures the difference between the essential features that cause B and the incidental features that are particular to S.

Figure 5.6 shows the logical structure of this kind of model. The features x and y are sufficient to produce the behavior. Adding more detail, like features w and z, might make the model more realistic, but that realism adds no explanatory power.

5.10 Implementing CAs

To generate the figures in this chapter, I wrote a Python class called `Cell1D` that represents a 1-D cellular automaton, and a class called `Cell1DViewer` that plots the results. Both are defined in `Cell1D.py` in the repository for this book.

To store the state of the CA, I use a NumPy array with one column for each cell and one row for each time step.

To explain how my implementation works, I'll start with a CA that computes the parity of the cells in each neighborhood. The "parity" of a number is 0 if the number is even and 1 if it is odd.

www.dbooks.org

I use the NumPy function `zeros` to create an array of zeros, then put a 1 in the middle of the first row.

```
rows = 5
cols = 11
array = np.zeros((rows, cols), dtype=np.uint8)
array[0, 5] = 1
print(array)
```

```
[[ 0.  0.  0.  0.  0.  1.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]]
```

The data type `uint8` indicates that the elements of `array` are unsigned 8-bit integers.

`plot_ca` displays the elements of an `array` graphically:

```
import matplotlib.pyplot as plt

def plot_ca(array, rows, cols):
    cmap = plt.get_cmap('Blues')
    plt.imshow(array, cmap=cmap, interpolation='none')
```

I import `pyplot` with the abbreviated name `plt`, which is conventional. The function `get_cmap` returns a colormap, which maps from the values in the array to colors. The colormap `'Blues'` draws the "on" cells in dark blue and the "off" cells in light blue.

`imshow` displays the array as an "image"; that is, it draws a colored square for each element of the array. Setting `interpolation` to `none` indicates that `imshow` should not interpolate between on and off cells.

To compute the state of the CA during time step `i`, we have to add up consecutive elements of `array` and compute the parity of the sum. We can due that using a slice operator to select the elements and the modulus operator to compute parity:

```
def step(array, i):
    rows, cols = array.shape
    row = array[i-1]
    for j in range(1, cols):
        elts = row[j-1:j+2]
        array[i, j] = sum(elts) % 2
```

`rows` and `cols` are the dimensions of the array. `row` is the previous row of the array.

Each time through the loop, we select three elements from `row`, add them up, compute the parity, and store the result in row `i`.

In this example, the lattice is finite, so the first and last cells have only one neighbor. To handle this special case, I don't update the first and last column; they are always 0.

5.11 Cross-correlation

The operation in the previous section — selecting elements from an array and adding them up — is an example of an operation that is so useful, in so many domains, that it has a name: **cross-correlation**. And NumPy provides a function, called `correlate`, that computes it. In this section I'll show how we can use NumPy to write a simpler, faster version of `step`.

The NumPy `correlate` function takes an array, a, and a "window", w, with length N and computes a new array, c, where element `k` is the following summation:

$$c_k = \sum_{n=0}^{N-1} a_{n+k} \cdot w_n$$

We can write this operation in Python like this:

```
def c_k(a, w, k):
    N = len(w)
    return sum(a[k:k+N] * w)
```

This function computes element `k` of the correlation between `a` and `w`. To show how it works, I'll create an array of integers:

www.dbooks.org

```
N = 10
row = np.arange(N, dtype=np.uint8)
print(row)

[0 1 2 3 4 5 6 7 8 9]
```

And a window:

```
window = [1, 1, 1]

print(window)
```

With this window, each element, `c_k`, is the sum of consecutive elements from `a`:

```
c_k(row, window, 0)
3

c_k(row, window, 1)
6
```

We can use `c_k` to write `correlate`, which computes the elements of `c` for all values of `k` where the window and the array overlap.

```
def correlate(row, window):
    cols = len(row)
    N = len(window)
    c = [c_k(row, window, k) for k in range(cols-N+1)]
    return np.array(c)
```

Here's the result:

```
c = correlate(row, window)
print(c)

[ 3  6  9 12 15 18 21 24]
```

The NumPy function `correlate` does the same thing:

```
c = np.correlate(row, window, mode='valid')
print(c)

[ 3  6  9 12 15 18 21 24]
```

The argument `mode='valid'` means that the result contains only the elements where the window and array overlap, which are considered valid.

The drawback of this mode is that the result is not the same size as `array`. We can fix that with `mode='same'`, which adds zeros to the beginning and end of `array`:

```
c = np.correlate(row, window, mode='same')
print(c)

[ 1  3  6  9 12 15 18 21 24 17]
```

Now the result is the same size as `array`. As an exercise at the end of this chapter, you'll have a chance to write a version of `correlate` that does the same thing.

We can use NumPy's implementation of `correlate` to write a simple, faster version of `step`:

```
def step2(array, i, window=[1,1,1]):
    row = array[i-1]
    c = np.correlate(row, window, mode='same')
    array[i] = c % 2
```

In the notebook for this chapter, you'll see that `step2` yields the same results as `step`.

5.12 CA tables

The function we have so far works if the CA is "totalitic", which means that the rules only depend on the sum of the neighbors. But most rules also depend on which neighbors are on and off. For example, `100` and `001` have the same sum, but for many CAs, they would yield different results.

We can make `step2` more general using a window with elements `[4, 2, 1]`, which interprets the neighborhood as a binary number. For example, the neighborhood `100` yields 4; `010` yields 2, and `001` yields 1. Then we can take these results and look them up in the rule table.

www.dbooks.org

Here's the more general version of `step2`:

```
def step3(array, i, window=[4,2,1]):
    row = array[i-1]
    c = np.correlate(row, window, mode='same')
    array[i] = table[c]
```

The first two lines are the same. Then the last line looks up each element from `c` in `table` and assigns the result to `array[i]`.

Here's the function that computes the table:

```
def make_table(rule):
    rule = np.array([rule], dtype=np.uint8)
    table = np.unpackbits(rule)[::-1]
    return table
```

The parameter, `rule`, is an integer between 0 and 255. The first line puts `rule` into an array with a single element so we can use `unpackbits`, which converts the rule number to its binary representation. For example, here's the table for Rule 150:

```
>>> table = make_table(150)
>>> print(table)
[0 1 1 0 1 0 0 1]
```

The code in this section is encapsulated in the `Cell1D` class, defined in `Cell1D.py` in the repository for this book.

5.13 Exercises

The code for this chapter is in the Jupyter notebook `chap05.ipynb` in the repository for this book. Open this notebook, read the code, and run the cells. You can use this notebook to work on the exercises in this chapter. My solutions are in `chap05soln.ipynb`.

Exercise 5.1 Write a version of `correlate` that returns the same result as `np.correlate` with `mode='same'`. Hint: use the NumPy function `pad`.

Exercise 5.2 This exercise asks you to experiment with Rule 110 and some of its spaceships.

1. Read the Wikipedia page about Rule 110, which describes its background pattern and spaceships: http://thinkcomplex.com/r110.

2. Create a Rule 110 CA with an initial condition that yields the stable background pattern.

   Note that the `Cell1D` class provides `start_string`, which allows you to initialize the state of the array using a string of `1`s and `0`s.

3. Modify the initial condition by adding different patterns in the center of the row and see which ones yield spaceships. You might want to enumerate all possible patterns of n bits, for some reasonable value of n. For each spaceship, can you find the period and rate of translation? What is the biggest spaceship you can find?

4. What happens when spaceships collide?

Exercise 5.3 The goal of this exercise is to implement a Turing machine.

1. Read about Turing machines at http://thinkcomplex.com/tm.

2. Write a class called `Turing` that implements a Turing machine. For the action table, use the rules for a 3-state busy beaver.

3. Write a class named `TuringViewer` that generates an image that represents the state of the tape and the position and state of the head. For one example of what that might look like, see http://thinkcomplex.com/turing.

Exercise 5.4 This exercise asks you to implement and test several PRNGs. For testing, you will need to install `DieHarder`, which you can download from http://thinkcomplex.com/dh, or it might be available as a package for your operating system.

1. Write a program that implements one of the linear congruential generators described at http://thinkcomplex.com/lcg. Test it using `DieHarder`.

2. Read the documentation of Python's `random` module. What PRNG does it use? Test it.

www.dbooks.org

3. Implement a Rule 30 CA with a few hundred cells, run it for as many time steps as you can in a reasonable amount of time, and output the center column as a sequence of bits. Test it.

Exercise 5.5 Falsifiability is an appealing and useful idea, but among philosophers of science it is not generally accepted as a solution to the demarcation problem, as Popper claimed.

Read http://thinkcomplex.com/false and answer the following questions.

1. What is the demarcation problem?

2. How, according to Popper, does falsifiability solve the demarcation problem?

3. Give an example of two theories, one considered scientific and one considered unscientific, that are successfully distinguished by the criterion of falsifiability.

4. Can you summarize one or more of the objections that philosophers and historians of science have raised to Popper's claim?

5. Do you get the sense that practicing philosophers think highly of Popper's work?

www.dbooks.org

Chapter 6

Game of Life

In this chapter we consider two-dimensional cellular automatons, especially John Conway's Game of Life (GoL). Like some of the 1-D CAs in the previous chapter, GoL follows simple rules and produces surprisingly complicated behavior. And like Wolfram's Rule 110, GoL turns out to be universal; that is, it can compute any computable function, at least in theory.

Complex behavior in GoL raises issues in the philosophy of science, particularly related to scientific realism and instrumentalism. I discuss these issues and suggest additional reading.

At the end of the chapter, I demonstrate ways to implement GoL efficiently in Python.

The code for this chapter is in `chap06.ipynb` in the repository for this book. More information about working with the code is in Section 0.3.

6.1 Conway's GoL

One of the first cellular automatons to be studied, and probably the most popular of all time, is a 2-D CA called "The Game of Life", or GoL for short. It was developed by John H. Conway and popularized in 1970 in Martin Gardner's column in *Scientific American*. See http://thinkcomplex.com/gol.

The cells in GoL are arranged in a 2-D **grid**, that is, an array of rows and columns. Usually the grid is considered to be infinite, but in practice it is often "wrapped"; that is, the right edge is connected to the left, and the top edge to the bottom.

Each cell in the grid has two states — live and dead — and 8 neighbors — north, south, east, west, and the four diagonals. This set of neighbors is sometimes called a "Moore neighborhood".

Like the 1-D CAs in the previous chapters, GoL evolves over time according to rules, which are like simple laws of physics.

In GoL, the next state of each cell depends on its current state and its number of live neighbors. If a cell is alive, it stays alive if it has 2 or 3 neighbors, and dies otherwise. If a cell is dead, it stays dead unless it has exactly 3 neighbors.

This behavior is loosely analogous to real cell growth: cells that are isolated or overcrowded die; at moderate densities they flourish.

GoL is popular because:

- There are simple initial conditions that yield surprisingly complex behavior.
- There are many interesting stable patterns: some oscillate (with various periods) and some move like the spaceships in Wolfram's Rule 110 CA.
- And like Rule 110, GoL is Turing complete.
- Another factor that generated interest was Conway's conjecture — that there is no initial condition that yields unbounded growth in the number of live cells — and the $50 bounty he offered to anyone who could prove or disprove it.
- Finally, the increasing availability of computers made it possible to automate the computation and display the results graphically.

www.dbooks.org

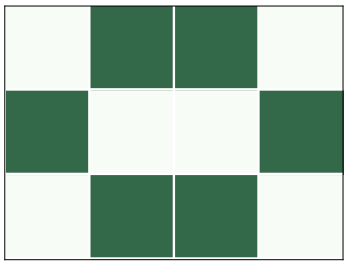


Figure 6.1: A stable pattern called a beehive.

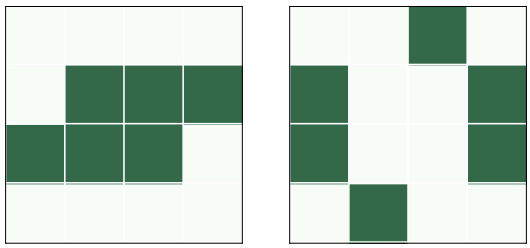


Figure 6.2: An oscillator called a toad.

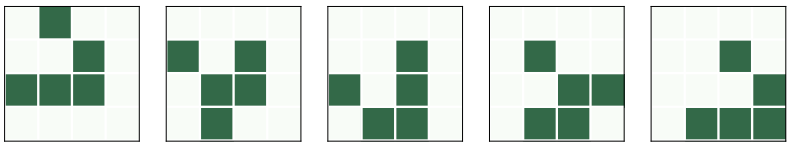


Figure 6.3: A spaceship called a glider.

6.2 Life patterns

If you run GoL from a random starting state, a number of stable patterns are likely to appear. Over time, people have identified these patterns and given them names.

For example, Figure 6.1 shows a stable pattern called a "beehive". Every cell in the beehive has 2 or 3 neighbors, so they all survive, and none of the dead cells adjacent to the beehive has 3 neighbors, so no new cells are born.

Other patterns "oscillate"; that is, they change over time but eventually return to their starting configuration (provided they don't collide with another pattern). For example, Figure 6.2 shows a pattern called a "toad", which is an oscillator that alternates between two states. The "period" of this oscillator is 2.

Finally, some patterns oscillate and return to the starting configuration, but shifted in space. Because these patterns seem to move, they are called "spaceships".

Figure 6.3 shows a spaceship called a "glider". After a period of 4 steps, the glider is back in the starting configuration, shifted one unit down and to the right.

Depending on the starting orientation, gliders can move along any of the four diagonals. There are other spaceships that move horizontally and vertically.

People have spent embarrassing amounts of time finding and naming these patterns. If you search the web, you will find many collections.

6.3 Conway's conjecture

From most initial conditions, GoL quickly reaches a stable state where the number of live cells is nearly constant (possibly with some oscillation).

But there are some simple starting conditions that yield a surprising number of live cells, and take a long time to settle down. Because these patterns are so long-lived, they are called "Methuselahs".

www.dbooks.org

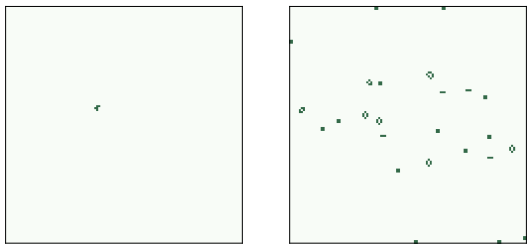


Figure 6.4: Starting and final configurations of the r-pentomino.

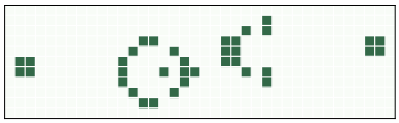


Figure 6.5: Gosper's glider gun, which produces a stream of gliders.

One of the simplest Methuselahs is the r-pentomino, which has only five cells, roughly in the shape of the letter "r". Figure 6.4 shows the initial configuration of the r-pentomino and the final configuration after 1103 steps.

This configuration is "final" in the sense that all remaining patterns are either stable, oscillators, or gliders that will never collide with another pattern. In total, the r-pentomino yields 6 gliders, 8 blocks, 4 blinkers, 4 beehives, 1 boat, 1 ship, and 1 loaf.

The existence of long-lived patterns prompted Conway to wonder if there are initial patterns that never stabilize. He conjectured that there were not, but he described two kinds of pattern that would prove him wrong, a "gun" and a "puffer train". A gun is a stable pattern that periodically produces a spaceship — as the stream of spaceships moves out from the source, the number of live cells grows indefinitely. A puffer train is a translating pattern that leaves live cells in its wake.

It turns out that both of these patterns exist. A team led by Bill Gosper discovered the first, a glider gun now called Gosper's Gun, which is shown in Figure 6.5. Gosper also discovered the first puffer train.

There are many patterns of both types, but they are not easy to design or find. That is not a coincidence. Conway chose the rules of GoL so that his conjecture would not be obviously true or false. Of all possible rules for a 2-D CA, most yield simple behavior: most initial conditions stabilize quickly or grow unboundedly. By avoiding uninteresting CAs, Conway was also avoiding Wolfram's Class 1 and Class 2 behavior, and probably Class 3 as well.

If we believe Wolfram's Principle of Computational Equivalence, we expect GoL to be in Class 4, and it is. The Game of Life was proved Turing complete in 1982 (and again, independently, in 1983). Since then, several people have constructed GoL patterns that implement a Turing machine or another machine known to be Turing complete.

6.4 Realism

Stable patterns in GoL are hard not to notice, especially the ones that move. It is natural to think of them as persistent entities, but remember that a CA is made of cells; there is no such thing as a toad or a loaf. Gliders and other spaceships are even less real because they are not even made up of the same cells over time. So these patterns are like constellations of stars. We perceive them because we are good at seeing patterns, or because we have active imaginations, but they are not real.

Right?

Well, not so fast. Many entities that we consider "real" are also persistent patterns of entities at a smaller scale. Hurricanes are just patterns of air flow, but we give them personal names. And people, like gliders, are not made up of the same cells over time.

This is not a new observation — about 2500 years ago Heraclitus pointed out that you can't step in the same river twice — but the entities that appear in the Game of Life are a useful test case for thinking about scientific realism.

www.dbooks.org

Scientific realism pertains to scientific theories and the entities they postulate. A theory postulates an entity if it is expressed in terms of the properties and behavior of the entity. For example, theories about electromagnetism are expressed in terms of electrical and magnetic fields. Some theories about economics are expressed in terms of supply, demand, and market forces. And theories about biology are expressed in terms of genes.

But are these entities real? That is, do they exist in the world independent of us and our theories?

Again, I find it useful to state philosophical positions in a range of strengths; here are four statements of scientific realism with increasing strength:

SR1: Scientific theories are true or false to the degree that they approximate reality, but no theory is exactly true. Some postulated entities may be real, but there is no principled way to say which ones.

SR2: As science advances, our theories become better approximations of reality. At least some postulated entities are known to be real.

SR3: Some theories are exactly true; others are approximately true. Entities postulated by true theories, and some entities in approximate theories, are real.

SR4: A theory is true if it describes reality correctly, and false otherwise. The entities postulated by true theories are real; others are not.

SR4 is so strong that it is probably untenable; by such a strict criterion, almost all current theories are known to be false. Most realists would accept something in the range between SR1 and SR3.

6.5 Instrumentalism

But SR1 is so weak that it verges on **instrumentalism**, which is the view that theories are instruments that we use for our purposes: a theory is useful, or not, to the degree that it is fit for its purpose, but we can't say whether it is true or false.

To see whether you are comfortable with instrumentalism, I made up the following test. Read the following statements and give yourself a point for each one you agree with. If you score 4 or more, you might be an instrumentalist!

> "Entities in the Game of Life aren't real; they are just patterns of cells that people have given cute names."
>
> "A hurricane is just a pattern of air flow, but it is a useful description because it allows us to make predictions and communicate about the weather."
>
> "Freudian entities like the Id and the Superego aren't real, but they are useful tools for thinking and communicating about psychology (or at least some people think so)."
>
> "Electric and magnetic fields are postulated entities in our best theory of electromagnetism, but they aren't real. We could construct other theories, without postulating fields, that would be just as useful."
>
> "Many of the things in the world that we identify as objects are arbitrary collections like constellations. For example, a mushroom is just the fruiting body of a fungus, most of which grows underground as a barely-contiguous network of cells. We focus on mushrooms for practical reasons like visibility and edibility."
>
> "Some objects have sharp boundaries, but many are fuzzy. For example, which molecules are part of your body: Air in your lungs? Food in your stomach? Nutrients in your blood? Nutrients in a cell? Water in a cell? Structural parts of a cell? Hair? Dead skin? Dirt? Bacteria on your skin? Bacteria in your gut? Mitochondria? How many of those molecules do you include when you weigh yourself? Conceiving the world in terms of discrete objects is useful, but the entities we identify are not real."

If you are more comfortable with some of these statements than others, ask yourself why. What are the differences in these scenarios that influence your reaction? Can you make a principled distinction between them?

For more on instrumentalism, see http://thinkcomplex.com/instr.

www.dbooks.org

6.6 Implementing Life

The exercises at the end of this chapter ask you to experiment with and modify the Game of Life, and implement other 2-D cellular automatons. This section explains my implementation of GoL, which you can use as a starting place for your experiments.

To represent the state of the cells, I use a NumPy array of 8-bit unsigned integers. As an example, the following line creates a 10 by 10 array initialized with random values of 0 and 1.

```
a = np.random.randint(2, size=(10, 10), dtype=np.uint8)
```

There are a few ways we can compute the GoL rules. The simplest is to use `for` loops to iterate through the rows and columns of the array:

```
b = np.zeros_like(a)
rows, cols = a.shape
for i in range(1, rows-1):
    for j in range(1, cols-1):
        state = a[i, j]
        neighbors = a[i-1:i+2, j-1:j+2]
        k = np.sum(neighbors) - state
        if state:
            if k==2 or k==3:
                b[i, j] = 1
        else:
            if k == 3:
                b[i, j] = 1
```

Initially, `b` is an array of zeros with the same size as `a`. Each time through the loop, `state` is the condition of the center cell and `neighbors` is the 3x3 neighborhood. `k` is the number of live neighbors (not including the center cell). The nested `if` statements evaluate the GoL rules and turn on cells in `b` accordingly.

This implementation is a straightforward translation of the rules, but it is verbose and slow. We can do better using cross-correlation, as we saw in Section 5.11. There, we used `np.correlate` to compute a 1-D correlation.

Now, to perform 2-D correlation, we'll use `correlate2d` from `scipy.signal`, a SciPy module that provides functions related to signal processing:

```
from scipy.signal import correlate2d

kernel = np.array([[1, 1, 1],
                   [1, 0, 1],
                   [1, 1, 1]])

c = correlate2d(a, kernel, mode='same')
```

What we called a "window" in the context of 1-D correlation is called a "kernel" in the context of 2-D correlation, but the idea is the same: `correlate2d` multiplies the kernel and the array to select a neighborhood, then adds up the result. This kernel selects the 8 neighbors that surround the center cell.

`correlate2d` applies the kernel to each location in the array. With `mode='same'`, the result has the same size as `a`.

Now we can use logical operators to compute the rules:

```
b = (c==3) | (c==2) & a
b = b.astype(np.uint8)
```

The first line computes a boolean array with `True` where there should be a live cell and `False` elsewhere. Then `astype` converts the boolean array to an array of integers.

This version is faster, and probably good enough, but we can simplify it slightly by modifying the kernel:

```
kernel = np.array([[1, 1, 1],
                   [1,10, 1],
                   [1, 1, 1]])

c = correlate2d(a, kernel, mode='same')
b = (c==3) | (c==12) | (c==13)
b = b.astype(np.uint8)
```

This version of the kernel includes the center cell and gives it a weight of 10. If the center cell is 0, the result is between 0 and 8; if the center cell is 1,

www.dbooks.org

the result is between 10 and 18. Using this kernel, we can simplify the logical operations, selecting only cells with the values 3, 12, and 13.

That might not seem like a big improvement, but it allows one more simplification: with this kernel, we can use a table to look up cell values, as we did in Section 5.12.

```
table = np.zeros(20, dtype=np.uint8)
table[[3, 12, 13]] = 1
c = correlate2d(a, kernel, mode='same')
b = table[c]
```

`table` has zeros everywhere except locations 3, 12, and 13. When we use `c` as an index into `table`, NumPy performs element-wise lookup; that is, it takes each value from `c`, looks it up in `table`, and puts the result into `b`.

This version is faster and more concise than the others; the only drawback is that it takes more explaining.

`Life.py`, which is included in the repository for this book, provides a `Life` class that encapsulates this implementation of the rules. If you run `Life.py`, you should see an animation of a "puffer train", which is a spaceship that leaves a trail of detritus in its wake.

6.7 Exercises

The code for this chapter is in the Jupyter notebook `chap06.ipynb` in the repository for this book. Open this notebook, read the code, and run the cells. You can use this notebook to work on the following exercises. My solutions are in `chap06soln.ipynb`.

Exercise 6.1 Start GoL in a random state and run it until it stabilizes. What stable patterns can you identify?

Exercise 6.2 Many named patterns are available in portable file formats. Modify `Life.py` to parse one of these formats and initialize the grid.

Exercise 6.3 One of the longest-lived small patterns is "rabbits", which starts with 9 live cells and takes 17,331 steps to stabilize. You can get the initial

configuration in various formats from http://thinkcomplex.com/rabbits. Load this configuration and run it.

Exercise 6.4 In my implementation, the `Life` class is based on a parent class called `Cell2D`, and the `LifeViewer` class is based on `Cell2DViewer`. You can use these base classes to implement other 2-D cellular automatons.

For example, one variation of GoL, called "Highlife", has the same rules as GoL, plus one additional rule: a dead cell with 6 neighbors comes to life.

Write a class named `Highlife` that inherits from `Cell2D` and implements this version of the rules. Also write a class named `HighlifeViewer` that inherits from `Cell2DViewer` and try different ways to visualize the results. As a simple example, use a different colormap.

One of the more interesting patterns in Highlife is the replicator (see http://thinkcomplex.com/repl). Use `add_cells` to initialize Highlife with a replicator and see what it does.

Exercise 6.5 If you generalize the Turing machine to two dimensions, or add a read-write head to a 2-D CA, the result is a cellular automaton called a Turmite. It is named after a termite because of the way the read-write head moves, but spelled wrong as an homage to Alan Turing.

The most famous Turmite is Langton's Ant, discovered by Chris Langton in 1986. See http://thinkcomplex.com/langton.

The ant is a read-write head with four states, which you can think of as facing north, south, east or west. The cells have two states, black and white.

The rules are simple. During each time step, the ant checks the color of the cell it is on. If black, the ant turns to the right, changes the cell to white, and moves forward one space. If the cell is white, the ant turns left, changes the cell to black, and moves forward.

Given a simple world, a simple set of rules, and only one moving part, you might expect to see simple behavior — but you should know better by now. Starting with all white cells, Langton's ant moves in a seemingly random pattern for more than 10,000 steps before it enters a cycle with a period of 104 steps. After each cycle, the ant is translated diagonally, so it leaves a trail called the "highway".

www.dbooks.org

Write an implementation of Langton's Ant.

www.dbooks.org

Chapter 7

Physical modeling

The cellular automatons we have seen so far are not physical models; that is, they are not intended to describe systems in the real world. But some CAs are intended as physical models.

In this chapter we consider a CA that models chemicals that diffuse (spread out) and react with each other, which is a process Alan Turing proposed to explain how some animal patterns develop.

And we'll experiment with a CA that models percolation of liquid through porous material, like water through coffee grounds. This model is the first of several models that exhibit **phase change** behavior and **fractal geometry**, and I'll explain what both of those mean.

The code for this chapter is in `chap07.ipynb` in the repository for this book. More information about working with the code is in Section 0.3.

7.1 Diffusion

In 1952 Alan Turing published a paper called "The chemical basis of morphogenesis", which describes the behavior of systems involving two chemicals that diffuse in space and react with each other. He showed that these systems produce a wide range of patterns, depending on the diffusion and reaction rates, and conjectured that systems like this might be an important mechanism in

biological growth processes, particularly the development of animal coloration patterns.

Turing's model is based on differential equations, but it can be implemented using a cellular automaton.

Before we get to Turing's model, we'll start with something simpler: a diffusion system with just one chemical. We'll use a 2-D CA where the state of each cell is a continuous quantity (usually between 0 and 1) that represents the concentration of the chemical.

We'll model the diffusion process by comparing each cell with the average of its neighbors. If the concentration of the center cell exceeds the neighborhood average, the chemical flows from the center to the neighbors. If the concentration of the center cell is lower, the chemical flows the other way.

The following kernel computes the difference between each cell and the average of its neighbors:

```
    kernel = np.array([[0, 1, 0],
                       [1,-4, 1],
                       [0, 1, 0]])
```

Using `np.correlate2d`, we can apply this kernel to each cell in an array:

```
    c = correlate2d(array, kernel, mode='same')
```

We'll use a diffusion constant, `r`, that relates the difference in concentration to the rate of flow:

```
    array += r * c
```

Figure 7.1 shows results for a CA with size `n=9`, diffusion constant `r=0.1`, and initial concentration 0 everywhere except for an "island" in the middle. The figure shows the starting configuration and the state of the CA after 5 and 10 steps. The chemical spreads from the center outward, continuing until the concentration is the same everywhere.

www.dbooks.org

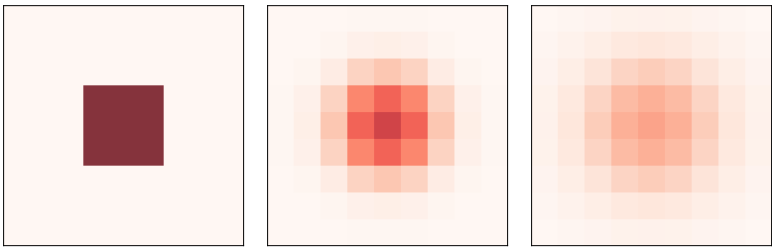


Figure 7.1: A simple diffusion model after 0, 5, and 10 steps.

7.2 Reaction-diffusion

Now let's add a second chemical. I'll define a new object, `ReactionDiffusion`, that contains two arrays, one for each chemical:

```
class ReactionDiffusion(Cell2D):

    def __init__(self, n, m, params, noise=0.1):
        self.params = params
        self.array = np.ones((n, m), dtype=float)
        self.array2 = noise * np.random.random((n, m))
        add_island(self.array2)
```

`n` and `m` are the number of rows and columns in the array. `params` is a tuple of parameters, which I explain below.

`array` represents the concentration of the first chemical, `A`; the NumPy function `ones` initializes it to 1 everywhere. The data type `float` indicates that the elements of `A` are floating-point values.

`array2` represents the concentration of `B`, which is initialized with random values between 0 and `noise`, which is 0.1 by default. Then `add_island` adds an island of higher concentration in the middle:

```
def add_island(a, height=0.1):
    n, m = a.shape
    radius = min(n, m) // 20
    i = n//2
    j = m//2
    a[i-radius:i+radius, j-radius:j+radius] += height
```

The radius of the island is one twentieth of `n` or `m`, whichever is smaller. The height of the island is `height`, with the default value 0.1.

Here is the `step` function that updates the arrays:

```
    def step(self):
        A = self.array
        B = self.array2
        ra, rb, f, k = self.params

        cA = correlate2d(A, self.kernel, **self.options)
        cB = correlate2d(B, self.kernel, **self.options)

        reaction = A * B**2
        self.array += ra * cA - reaction + f * (1-A)
        self.array2 += rb * cB + reaction - (f+k) * B
```

The parameters are

> The diffusion rate of `A` (analogous to `r` in the previous section).
>
> The diffusion rate of `B`. In most versions of this model, `rb` is about half of `ra`.
>
> The "feed" rate, which controls how quickly `A` is added to the system.
>
> The "kill" rate, which controls how quickly `B` is removed from the system.

Now let's look more closely at the update statements:

```
        reaction = A * B**2
        self.array += ra * cA - reaction + f * (1-A)
        self.array2 += rb * cB + reaction - (f+k) * B
```

www.dbooks.org

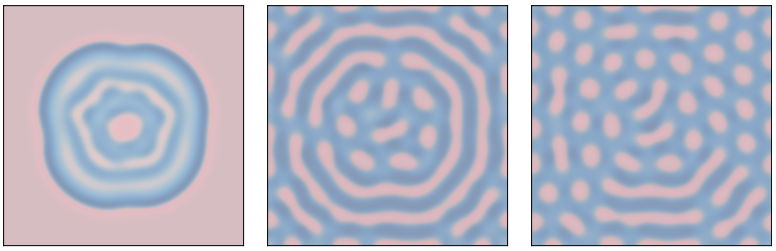


Figure 7.2: Reaction-diffusion model with parameters `f=0.035` and `k=0.057` after 1000, 2000, and 4000 steps.

The arrays `cA` and `cB` are the result of applying a diffusion kernel to `A` and `B`. Multiplying by `ra` and `rb` yields the rate of diffusion into or out of each cell.

The term `A * B**2` represents the rate that `A` and `B` react with each other. Assuming that the reaction consumes `A` and produces `B`, we subtract this term in the first equation and add it in the second.

The term `f * (1-A)` determines the rate that `A` is added to the system. Where `A` is near 0, the maximum feed rate is `f`. Where `A` approaches 1, the feed rate drops off to zero.

Finally, the term `(f+k) * B` determines the rate that `B` is removed from the system. As `B` approaches 0, this rate goes to zero.

As long as the rate parameters are not too high, the values of `A` and `B` usually stay between 0 and 1.

With different parameters, this model can produce patterns similar to the stripes and spots on a variety of animals. In some cases, the similarity is striking, especially when the feed and kill parameters vary in space.

For all simulations in this section, `ra=0.5` and `rb=0.25`.

Figure 7.2 shows results with `f=0.035` and `k=0.057`, with the concentration of `B` shown in darker colors. With these parameters, the system evolves toward a stable configuration with light spots of `A` on a dark background of `B`.

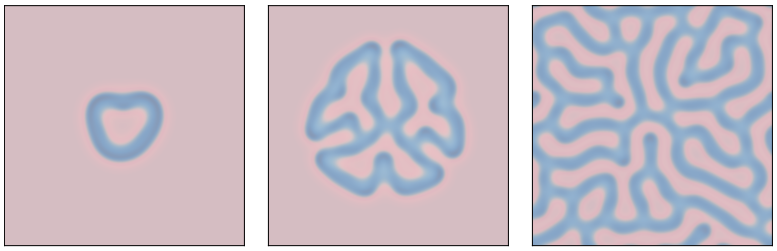


Figure 7.3: Reaction-diffusion model with parameters `f=0.055` and `k=0.062` after 1000, 2000, and 4000 steps.

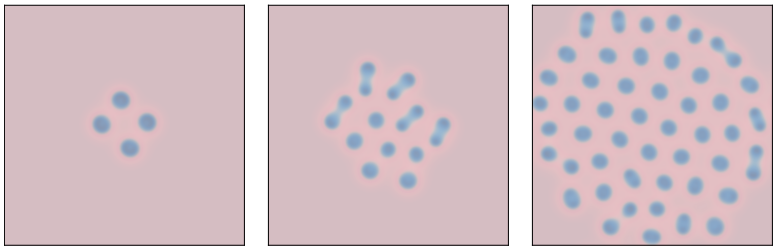


Figure 7.4: A reaction-diffusion model with parameters `f=0.039` and `k=0.065` after 1000, 2000, and 4000 steps.

Figure 7.3 shows results with `f=0.055` and `k=0.062`, which yields a coral-like pattern of `B` on a background of `A`.

Figure 7.4 shows results with `f=0.039` and `k=0.065`. These parameters produce spots of `B` that grow and divide in a process that resembles mitosis, ending with a stable pattern of equally-spaced spots.

Since 1952, observations and experiments have provided some support for Turing's conjecture. At this point it seems likely, but not yet proven, that many animal patterns are actually formed by reaction-diffusion processes of some kind.

www.dbooks.org

7.3 Percolation

Percolation is a process in which a fluid flows through a semi-porous material. Examples include oil in rock formations, water in paper, and hydrogen gas in micropores. Percolation models are also used to study systems that are not literally percolation, including epidemics and networks of electrical resistors. See http://thinkcomplex.com/perc.

Percolation models are often represented using random graphs like the ones we saw in Chapter 2, but they can also be represented using cellular automatons. In the next few sections we'll explore a 2-D CA that simulates percolation.

In this model:

- Initially, each cell is either "porous" with probability `q` or "non-porous" with probability `1-q`.
- When the simulation begins, all cells are considered "dry" except the top row, which is "wet".
- During each time step, if a porous cell has at least one wet neighbor, it becomes wet. Non-porous cells stay dry.
- The simulation runs until it reaches a "fixed point" where no more cells change state.

If there is a path of wet cells from the top to the bottom row, we say that the CA has a "percolating cluster".

Two questions of interest regarding percolation are (1) the probability that a random array contains a percolating cluster, and (2) how that probability depends on `q`. These questions might remind you of Section 2.3, where we considered the probability that a random Erdős-Rényi graph is connected. We will see several connections between that model and this one.

I define a new class to represent a percolation model:

```
class Percolation(Cell2D):

    def __init__(self, n, q):
        self.q = q
        self.array = np.random.choice([1, 0], (n, n), p=[q, 1-q])
        self.array[0] = 5
```

`n` and `m` are the number of rows and columns in the CA.

The state of the CA is stored in `array`, which is initialized using `np.random.choice` to choose 1 (porous) with probability `q`, and 0 (non-porous) with probability `1-q`.

The state of the top row is set to 5, which represents a wet cell. Using 5, rather than the more obvious 2, makes it possible to use `correlate2d` to check whether any porous cell has a wet neighbor. Here is the kernel:

```
    kernel = np.array([[0, 1, 0],
                       [1, 0, 1],
                       [0, 1, 0]])
```

This kernel defines a 4-cell "von Neumann" neighborhood; unlike the Moore neighborhood we saw in Section 6.1, it does not include the diagonals.

This kernel adds up the states of the neighbors. If any of them are wet, the result will exceed 5. Otherwise the maximum result is 4 (if all neighbors happen to be porous).

We can use this logic to write a simple, fast `step` function:

```
    def step(self):
        a = self.array
        c = correlate2d(a, self.kernel, mode='same')
        self.array[(a==1) & (c>=5)] = 5
```

This function identifies porous cells, where `a==1`, that have at least one wet neighbor, where `c>=5`, and sets their state to 5, which indicates that they are wet.

Figure 7.5 shows the first few steps of a percolation model with `n=10` and `p=0.7`. Non-porous cells are white, porous cells are lightly shaded, and wet cells are dark.

7.4 Phase change

Now let's test whether a random array contains a percolating cluster:

www.dbooks.org

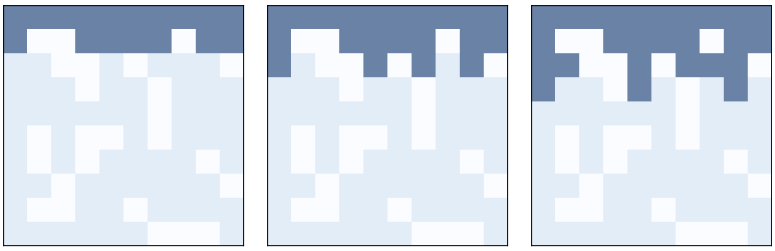


Figure 7.5: The first three steps of a percolation model with `n=10` and `p=0.7`.

```
def test_perc(perc):
    num_wet = perc.num_wet()

    while True:
        perc.step()

        if perc.bottom_row_wet():
            return True

        new_num_wet = perc.num_wet()
        if new_num_wet == num_wet:
            return False

        num_wet = new_num_wet
```

`test_perc` takes a `Percolation` object as a parameter. Each time through the loop, it advances the CA one time step. It checks the bottom row to see if any cells are wet; if so, it returns `True`, to indicate that there is a percolating cluster.

During each time step, it also computes the number of wet cells and checks whether the number increased since the last step. If not, we have reached a fixed point without finding a percolating cluster, so `test_perc` returns `False`.

To estimate the probability of a percolating cluster, we generate many random arrays and test them:

```
def estimate_prob_percolating(n=100, q=0.5, iters=100):
    t = [test_perc(Percolation(n, q)) for i in range(iters)]
    return np.mean(t)
```

`estimate_prob_percolating` makes 100 `Percolation` objects with the given values of `n` and `q` and calls `test_perc` to see how many of them have a percolating cluster. The return value is the fraction that do.

When `p=0.55`, the probability of a percolating cluster is near 0. At `p=0.60`, it is about 70%, and at `p=0.65` it is near 1. This rapid transition suggests that there is a critical value of `p` near 0.6.

We can estimate the critical value more precisely using a **random walk**. Starting from an initial value of `q`, we construct a `Percolation` object and check whether it has a percolating cluster. If so, `q` is probably too high, so we decrease it. If not, `q` is probably too low, so we increase it.

Here's the code:

```
def find_critical(n=100, q=0.6, iters=100):
    qs = [q]
    for i in range(iters):
        perc = Percolation(n, q)
        if test_perc(perc):
            q -= 0.005
        else:
            q += 0.005
        qs.append(q)
    return qs
```

The result is a list of values for `q`. We can estimate the critical value, `q_crit`, by computing the mean of this list. With `n=100` the mean of `qs` is about 0.59; this value does not seem to depend on `n`.

The rapid change in behavior near the critical value is called a **phase change** by analogy with phase changes in physical systems, like the way water changes from liquid to solid at its freezing point.

A wide variety of systems display a common set of behaviors and characteristics when they are at or near a critical point. These behaviors are known collectively as **critical phenomena**. In the next section, we explore one of them: fractal geometry.

www.dbooks.org

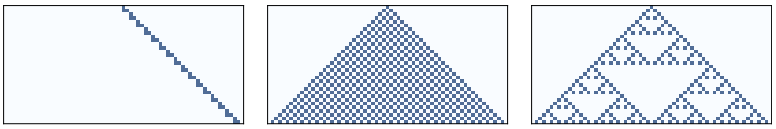


Figure 7.6: One-dimensional CAs with rules 20, 50, and 18, after 32 time steps.

7.5 Fractals

To understand fractals, we have to start with dimensions.

For simple geometric objects, dimension is defined in terms of scaling behavior. For example, if the side of a square has length l, its area is l^2. The exponent, 2, indicates that a square is two-dimensional. Similarly, if the side of a cube has length l, its volume is l^3, which indicates that a cube is three-dimensional.

More generally, we can estimate the dimension of an object by measuring some kind of size (like area or volume) as a function of some kind of linear measure (like the length of a side).

As an example, I'll estimate the dimension of a 1-D cellular automaton by measuring its area (total number of "on" cells) as a function of the number of rows.

Figure 7.6 shows three 1-D CAs like the ones we saw in Section 5.2. Rule 20 (left) generates a set of cells that seems like a line, so we expect it to be one-dimensional. Rule 50 (center) produces something like a triangle, so we expect it to be 2-D. Rule 18 (right) also produces something like a triangle, but the density is not uniform, so its scaling behavior is not obvious.

I'll estimate the dimension of these CAs with the following function, which counts the number of on cells after each time step. It returns a list of tuples, where each tuple contains i, i^2, and the total number of cells.

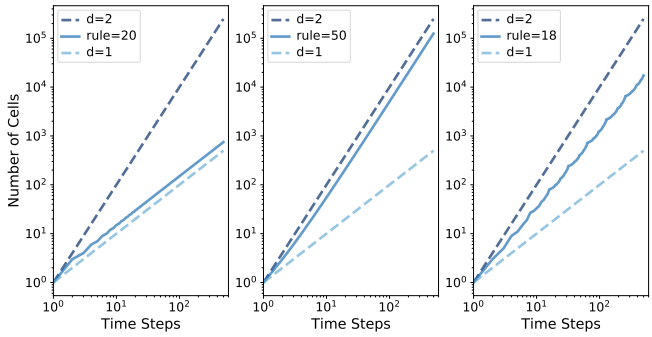


Figure 7.7: Number of "on" cells versus number of time steps for rules 20, 50, and 18.

```
def count_cells(rule, n=500):
    ca = Cell1D(rule, n)
    ca.start_single()

    res = []
    for i in range(1, n):
        cells = np.sum(ca.array)
        res.append((i, i**2, cells))
        ca.step()

    return res
```

Figure 7.7 shows the results plotted on a log-log scale.

In each figure, the top dashed line shows $y = i^2$. Taking the log of both sides, we have $\log y = 2 \log i$. Since the figure is on a log-log scale, the slope of this line is 2.

Similarly, the bottom dashed line shows $y = i$. On a log-log scale, the slope of this line is 1.

Rule 20 (left) produces 3 cells every 2 time steps, so the total number of cells after i steps is $y = 1.5i$. Taking the log of both sides, we have $\log y = \log 1.5 + \log i$, so on a log-log scale, we expect a line with slope 1. In fact, the estimated slope of the line is 1.01.

www.dbooks.org

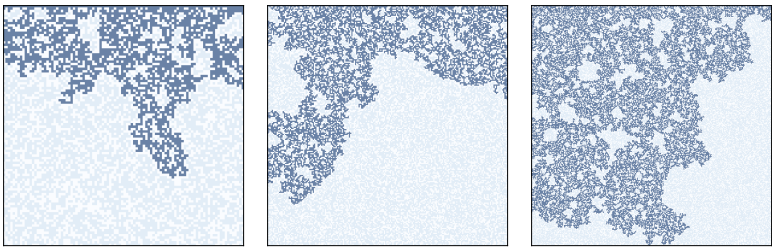


Figure 7.8: Percolation models with `q=0.6` and `n=100`, `200`, and `300`.

Rule 50 (center) produces $i+1$ new cells during the ith time step, so the total number of cells after i steps is $y = i^2 + i$. If we ignore the second term and take the log of both sides, we have $\log y \sim 2 \log i$, so as i gets large, we expect to see a line with slope 2. In fact, the estimated slope is 1.97.

Finally, for Rule 18 (right), the estimated slope is about 1.57, which is clearly not 1, 2, or any other integer. This suggests that the pattern generated by Rule 18 has a "fractional dimension"; that is, it is a fractal.

This way of estimating a fractal dimension is called **box-counting**. For more about it, see http://thinkcomplex.com/box.

7.6 Fractals and Percolation Models

Now let's get back to percolation models. Figure 7.8 shows clusters of wet cells in percolation simulations with `p=0.6` and `n=100`, `200`, and `300`. Informally, they resemble fractal patterns seen in nature and in mathematical models.

To estimate their fractal dimension, we can run CAs with a range of sizes, count the number of wet cells in each percolating cluster, and then see how the cell counts scale as we increase the size of the array.

The following loop runs the simulations:

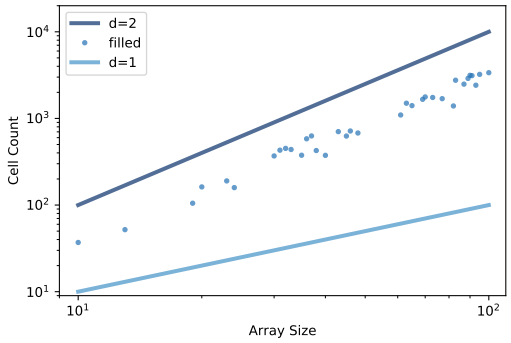


Figure 7.9: Number of cells in the percolating cluster versus CA size.

```
res = []
for size in sizes:
    perc = Percolation(size, q)
    if test_perc(perc):
        num_filled = perc.num_wet() - size
        res.append((size, size**2, num_filled))
```

The result is a list of tuples where each tuple contains `size`, `size**2`, and the number of cells in the percolating cluster (not including the initial wet cells in the top row).

Figure 7.9 shows the results for a range of sizes from 10 to 100. The dots show the number of cells in each percolating cluster. The slope of a line fitted to these dots is often near 1.85, which suggests that the percolating cluster is, in fact, fractal when `q` is near the critical value.

When `q` is larger than the critical value, nearly every porous cell gets filled, so the number of wet cells is close to `q * size^2`, which has dimension 2.

When `q` is substantially smaller than the critical value, the number of wet cells is proportional to the linear size of the array, so it has dimension 1.

7.7 Exercises

Exercise 7.1 In Section 7.6 we showed that the Rule 18 CA produces a fractal. Can you find other 1-D CAs that produce fractals?

www.dbooks.org

Note: the `Cell1D` object does not wrap around from the left edge to the right, which creates artifacts at the boundaries for some rules. You might want to use `Wrap1D`, which is a child class of `Cell1D` that wraps around. It is defined in `Cell1D.py` in the repository for this book.

Exercise 7.2 In 1990 Bak, Chen and Tang proposed a cellular automaton that is an abstract model of a forest fire. Each cell is in one of three states: empty, occupied by a tree, or on fire.

The rules of the CA are:

1. An empty cell becomes occupied with probability p.
2. A cell with a tree burns if any of its neighbors is on fire.
3. A cell with a tree spontaneously burns, with probability f, even if none of its neighbors is on fire.
4. A cell with a burning tree becomes an empty cell in the next time step.

Write a program that implements this model. You might want to inherit from `Cell2D`. Typical values for the parameters are $p = 0.01$ and $f = 0.001$, but you might want to experiment with other values.

Starting from a random initial condition, run the model until it reaches a steady state where the number of trees no longer increases or decreases consistently.

In steady state, is the geometry of the forest fractal? What is its fractal dimension?

www.dbooks.org

Chapter 8

Self-organized criticality

In the previous chapter we saw an example of a system with a critical point and we explored one of the common properties of critical systems, fractal geometry.

In this chapter, we explore two other properties of critical systems: heavy-tailed distributions, which we saw in Chapter 4.4 and pink noise, which I'll explain in this chapter.

These properties are interesting in part because they appear frequently in nature; that is, many natural systems produce fractal-like geometry, heavy-tailed distributions, and pink noise.

This observation raises a natural question: why do so many natural systems have properties of critical systems? A possible answer is **self-organized criticality** (SOC), which is the tendency of some systems to evolve toward, and stay in, a critical state.

In this chapter I'll present a **sand pile model** that was the first system shown to exhibit SOC.

The code for this chapter is in `chap08.ipynb` in the repository for this book. More information about working with the code is in Section 0.3.

8.1 Critical Systems

Many critical systems demonstrate common behaviors:

- Fractal geometry: For example, freezing water tends to form fractal patterns, including snowflakes and other crystal structures. Fractals are characterized by self-similarity; that is, parts of the pattern are similar to scaled copies of the whole.
- Heavy-tailed distributions of some physical quantities: For example, in freezing water the distribution of crystal sizes is characterized by a power law.
- Variations in time that exhibit **pink noise**: Complex signals can be decomposed into their frequency components. In pink noise, low-frequency components have more power than high-frequency components. Specifically, the power at frequency f is proportional to $1/f$.

Critical systems are usually unstable. For example, to keep water in a partially frozen state requires active control of the temperature. If the system is near the critical temperature, a small deviation tends to move the system into one phase or the other.

Many natural systems exhibit characteristic behaviors of criticality, but if critical points are unstable, they should not be common in nature. This is the puzzle Bak, Tang and Wiesenfeld address. Their solution is called self-organized criticality (SOC), where "self-organized" means that from any initial condition, the system moves toward a critical state, and stays there, without external control.

8.2 Sand Piles

The sand pile model was proposed by Bak, Tang and Wiesenfeld in 1987. It is not meant to be a realistic model of a sand pile, but rather an abstraction that models physical systems with a large number of elements that interact with their neighbors.

The sand pile model is a 2-D cellular automaton where the state of each cell represents the slope of a part of a sand pile. During each time step, each cell is checked to see whether it exceeds a critical value, K, which is usually 3. If so, it "topples" and transfers sand to four neighboring cells; that is, the slope of the cell is decreased by 4, and each of the neighbors is increased by 1. At

www.dbooks.org

the perimeter of the grid, all cells are kept at slope 0, so the excess spills over the edge.

Bak, Tang and Wiesenfeld initialize all cells at a level greater than `K` and run the model until it stabilizes. Then they observe the effect of small perturbations: they choose a cell at random, increment its value by 1, and run the model again until it stabilizes.

For each perturbation, they measure `T`, the number of time steps the pile takes to stabilize, and `S`, the total number of cells that topple[1].

Most of the time, dropping a single grain causes no cells to topple, so `T=1` and `S=0`. But occasionally a single grain can cause an **avalanche** that affects a substantial fraction of the grid. The distributions of `T` and `S` turn out to be heavy-tailed, which supports the claim that the system is in a critical state.

They conclude that the sand pile model exhibits "self-organized criticality", which means that it evolves toward a critical state without the need for external control or what they call "fine tuning" of any parameters. And the model stays in a critical state as more grains are added.

In the next few sections I replicate their experiments and interpret the results.

8.3 Implementing the Sand Pile

To implement the sand pile model, I define a class called `SandPile` that inherits from `Cell2D`, which is defined in `Cell2D.py` in the repository for this book.

```
class SandPile(Cell2D):

    def __init__(self, n, m, level=9):
        self.array = np.ones((n, m)) * level
```

All values in the array are initialized to `level`, which is generally greater than the toppling threshold, `K`.

Here's the `step` method that finds all cells above `K` and topples them:

[1]The original paper uses a different definition of `S`, but most later work uses this definition.

```
    kernel = np.array([[0, 1, 0],
                       [1,-4, 1],
                       [0, 1, 0]])

    def step(self, K=3):
        toppling = self.array > K
        num_toppled = np.sum(toppling)
        c = correlate2d(toppling, self.kernel, mode='same')
        self.array += c
        return num_toppled
```

To show how `step` works, I'll start with a small pile that has two cells ready to topple:

```
pile = SandPile(n=3, m=5, level=0)
pile.array[1, 1] = 4
pile.array[1, 3] = 4
```

Initially, `pile.array` looks like this:

```
[[0 0 0 0 0]
 [0 4 0 4 0]
 [0 0 0 0 0]]
```

Now we can select the cells that are above the toppling threshold:

```
toppling = pile.array > K
```

The result is a boolean array, but we can use it as if it were an array of integers like this:

```
[[0 0 0 0 0]
 [0 1 0 1 0]
 [0 0 0 0 0]]
```

If we correlate this array with the kernel, it makes copies of the kernel at each location where `toppling` is 1.

```
c = correlate2d(toppling, kernel, mode='same')
```

And here's the result:

www.dbooks.org

```
[[ 0  1  0  1  0]
 [ 1 -4  2 -4  1]
 [ 0  1  0  1  0]]
```

Notice that where the copies of the kernel overlap, they add up.

This array contains the change for each cell, which we use to update the original array:

```
pile.array += c
```

And here's the result.

```
[[0 1 0 1 0]
 [1 0 2 0 1]
 [0 1 0 1 0]]
```

So that's how `step` works.

With `mode='same'`, `correlate2d` considers the boundary of the array to be fixed at zero, so any grains of sand that go over the edge disappear.

`SandPile` also provides `run`, which calls `step` until no more cells topple:

```
    def run(self):
        total = 0
        for i in itertools.count(1):
            num_toppled = self.step()
            total += num_toppled
            if num_toppled == 0:
                return i, total
```

The return value is a tuple that contains the number of time steps and the total number of cells that toppled.

If you are not familiar with `itertools.count`, it is an infinite generator that counts up from the given initial value, so the `for` loop runs until `step` returns 0. You can read about the `itertools` module at http://thinkcomplex.com/iter.

Finally, the `drop` method chooses a random cell and adds a grain of sand:

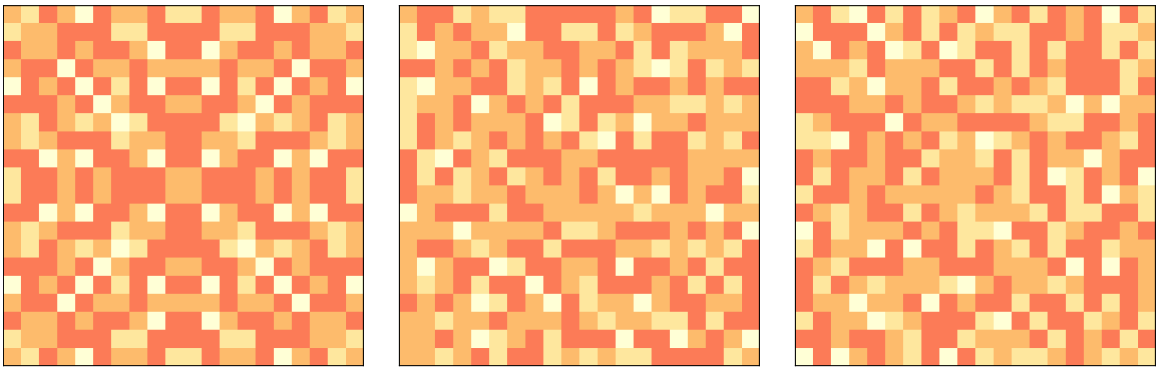


Figure 8.1: Sand pile model initial state (left), after 200 steps (middle), and 400 steps (right).

```
    def drop(self):
        a = self.array
        n, m = a.shape
        index = np.random.randint(n), np.random.randint(m)
        a[index] += 1
```

Let's look at a bigger example, with `n=20`:

```
pile = SandPile(n=20, level=10)
pile.run()
```

With an initial level of `10`, this sand pile takes 332 time steps to reach equilibrium, with a total of 53,336 topplings. Figure 8.1 (left) shows the configuration after this initial run. Notice that it has the repeating elements that are characteristic of fractals. We'll come back to that soon.

Figure 8.1 (middle) shows the configuration of the sand pile after dropping 200 grains onto random cells, each time running until the pile reaches equilibrium. The symmetry of the initial configuration has been broken; the configuration looks random.

www.dbooks.org

Finally Figure 8.1 (right) shows the configuration after 400 drops. It looks similar to the configuration after 200 drops. In fact, the pile is now in a steady state where its statistical properties don't change over time. I'll explain some of those statistical properties in the next section.

8.4 Heavy-tailed distributions

If the sand pile model is in a critical state, we expect to find heavy-tailed distributions for quantities like the duration and size of avalanches. So let's take a look.

I'll make a larger sand pile, with `n=50` and an initial level of `30`, and run until equilibrium:

```
pile2 = SandPile(n=50, level=30)
pile2.run()
```

Next, I'll run 100,000 random drops

```
iters = 100000
res = [pile2.drop_and_run() for _ in range(iters)]
```

As the name suggests, `drop_and_run` calls `drop` and `run` and returns the duration of the avalanche and total number of cells that toppled.

So `res` is a list of `(T, S)` tuples, where `T` is duration, in time steps, and `S` is cells toppled. We can use `np.transpose` to unpack `res` into two NumPy arrays:

```
T, S = np.transpose(res)
```

A large majority of drops have duration 1 and no toppled cells; if we filter them out before plotting, we get a clearer view of the rest of the distribution.

```
T = T[T>1]
S = S[S>0]
```

The distributions of `T` and `S` have many small values and a few very large ones. I'll use the `Pmf` class from `thinkstats2` to make a PMF of the values, that is, a map from each value to its probability of occurring (see Section 4.3).

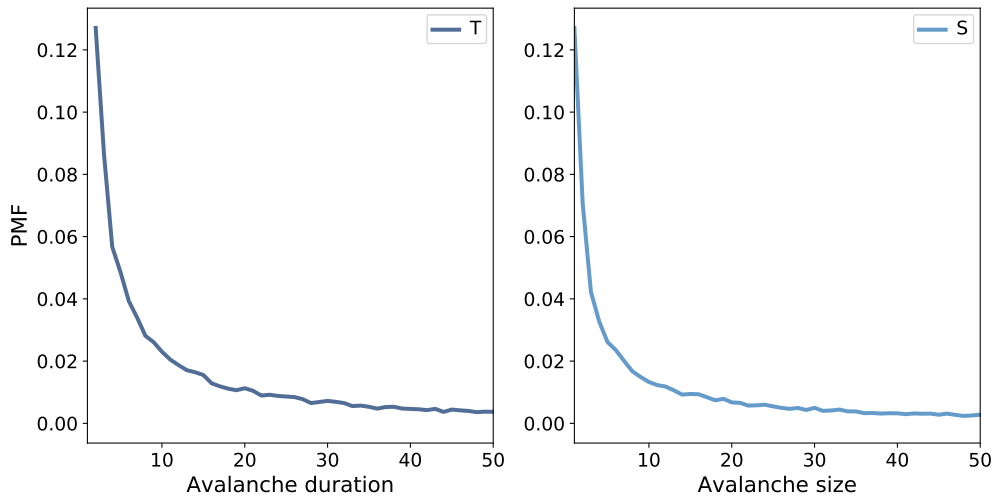


Figure 8.2: Distribution of avalanche duration (left) and size (right), linear scale.

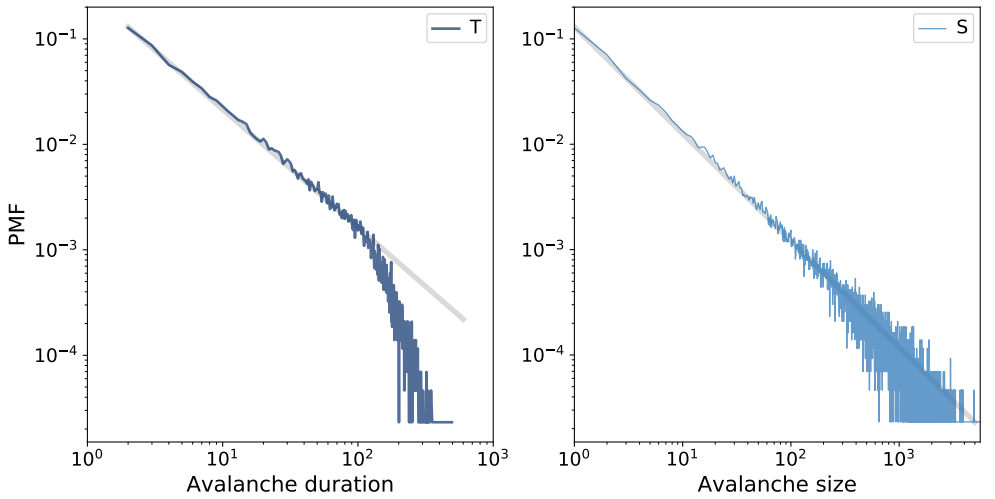


Figure 8.3: Distribution of avalanche duration (left) and size (right), log-log scale.

www.dbooks.org

```
pmfT = Pmf(T)
pmfS = Pmf(S)
```

Figure 8.2 shows the results for values less than 50.

As we saw in Section 4.4, we can get a clearer picture of these distributions by plotting them on a log-log scale, as shown in Figure 8.3.

For values between 1 and 100, the distributions are nearly straight on a log-log scale, which is characteristic of a heavy tail. The gray lines in the figure have slopes near -1, which suggests that these distributions follow a power law with parameters near $\alpha = 1$.

For values greater than 100, the distributions fall away more quickly than the power law model, which means there are fewer very large values than the model predicts. One possibility is that this effect is due to the finite size of the sand pile; if so, we might expect larger piles to fit the power law better.

Another possibility, which you can explore in one of the exercises at the end of this chapter, is that these distributions do not strictly obey a power law. But even if they are not power-law distributions, they might still be heavy-tailed.

8.5 Fractals

Another property of critical systems is fractal geometry. The initial configuration in Figure 8.1 (left) resembles a fractal, but you can't always tell by looking. A more reliable way to identify a fractal is to estimate its fractal dimension, as we saw in Section 7.5 and Section 7.6.

I'll start by making a bigger sand pile, with `n=131` and initial level `22`.

```
pile3 = SandPile(n=131, level=22)
pile3.run()
```

It takes 28,379 steps for this pile to reach equilibrium, with more than 200 million cells toppled.

To see the resulting pattern more clearly, I select cells with levels 0, 1, 2, and 3, and plot them separately:

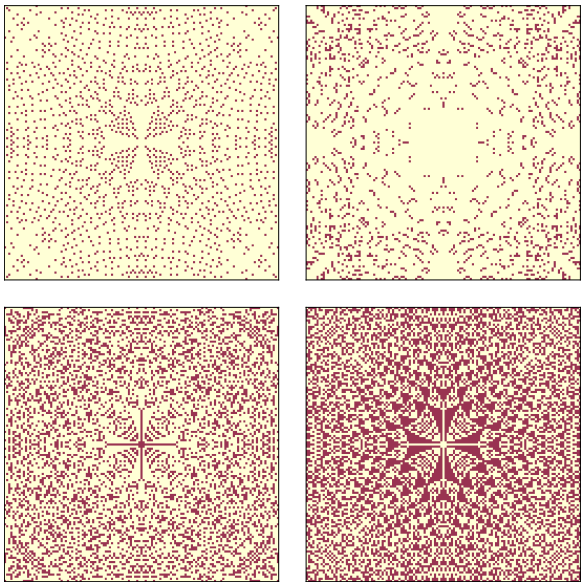


Figure 8.4: Sand pile model in equilibrium, selecting cells with levels 0, 1, 2, and 3, left to right, top to bottom.

```
def draw_four(viewer, levels=range(4)):
    thinkplot.preplot(rows=2, cols=2)
    a = viewer.viewee.array

    for i, level in enumerate(levels):
        thinkplot.subplot(i+1)
        viewer.draw_array(a==level, vmax=1)
```

`draw_four` takes a `SandPileViewer` object, which is defined in `Sand.py` in the repository for this book. The parameter `levels` is the list of levels we want to plot; the default is the range 0 through 3. If the sand pile has run until equilibrium, these are the only levels that should exist.

Inside the loop, it uses `a==level` to make a boolean array that's `True` where the array is `level` and `False` otherwise. `draw_array` treats these booleans as 1s and 0s.

Figure 8.4 shows the results for `pile3`. Visually, these patterns resemble fractals, but looks can be deceiving. To be more confident, we can estimate the fractal dimension for each pattern using **box-counting**, as we saw in Section 7.5.

www.dbooks.org

We'll count the number of cells in a small box at the center of the pile, then see how the number of cells increases as the box gets bigger. Here's my implementation:

```
def count_cells(a):
    n, m = a.shape
    end = min(n, m)

    res = []
    for i in range(1, end, 2):
        top = (n-i) // 2
        left = (m-i) // 2
        box = a[top:top+i, left:left+i]
        total = np.sum(box)
        res.append((i, i**2, total))

    return np.transpose(res)
```

The parameter, `a`, is a boolean array. The size of the box is initially 1. Each time through the loop, it increases by 2 until it reaches `end`, which is the smaller of `n` and `m`.

Each time through the loop, `box` is a set of cells with width and height `i`, centered in the array. `total` is the number of "on" cells in the box.

The result is a list of tuples, where each tuple contains `i`, `i**2`, and the number of cells in the box. When we pass this result to `transpose`, NumPy converts it to an array with three columns, and then **transposes** it; that is, it makes the columns into rows and the rows into columns. The result is an array with 3 rows: `i`, `i**2`, and `total`.

Here's how we use `count_cells`:

```
res = count_cells(pile.array==level)
steps, steps2, cells = res
```

The first line creates a boolean array that contains `True` where the array equals `level`, calls `count_cells`, and gets an array with three rows.

The second line unpacks the rows and assigns them to `steps`, `steps2`, and `cells`, which we can plot like this:

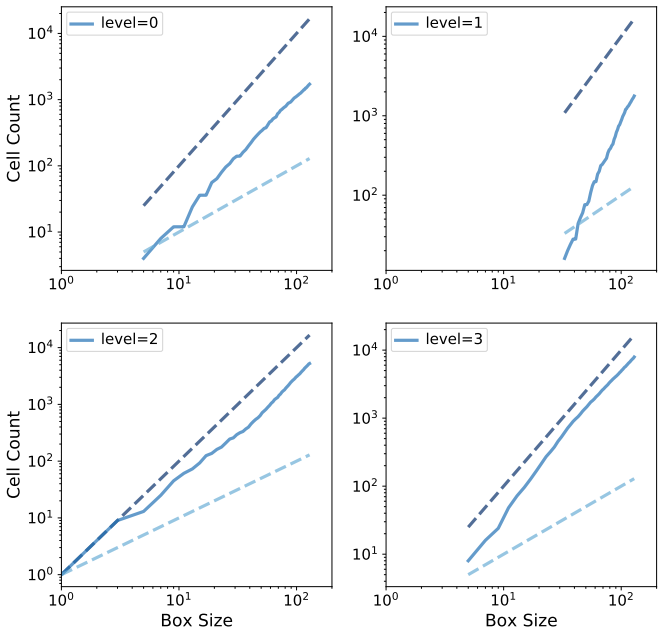


Figure 8.5: Box counts for cells with levels 0, 1, 2, and 3, compared to dashed lines with slopes 1 and 2.

```
    thinkplot.plot(steps, steps2, linestyle='dashed')
    thinkplot.plot(steps, cells)
    thinkplot.plot(steps, steps, linestyle='dashed')
```

Figure 8.5 shows the results. On a log-log scale, the cell counts form nearly straight lines, which indicates that we are measuring fractal dimension over a valid range of box sizes.

To estimate the slopes of these lines, we can use the SciPy function `linregress`, which fits a line to the data by linear regression (see http://thinkcomplex.com/regress).

```
from scipy.stats import linregress

params = linregress(np.log(steps), np.log(cells))
slope = params[0]
```

www.dbooks.org

The estimated fractal dimensions are:

```
0  1.871
1  3.502
2  1.781
3  2.084
```

The fractal dimension for levels 0, 1, and 2 seems to be clearly non-integer, which indicates that the image is fractal.

The estimate for level 3 is indistinguishable from 2, but given the results for the other values, the apparent curvature of the line, and the appearance of the pattern, it seems likely that it is also fractal.

One of the exercises in the notebook for this chapter asks you to run this analysis again with different values of `n` and the initial `level` to see if the estimated dimensions are consistent.

8.6 Pink noise

The title of the original paper that presented the sand pile model is "Self-Organized Criticality: An Explanation of $1/f$ Noise". You can read it at http://thinkcomplex.com/bak.

As the subtitle suggests, Bak, Tang and Wiesenfeld were trying to explain why many natural and engineered systems exhibit $1/f$ noise, which is also known as "flicker noise" and "pink noise".

To understand pink noise, we have to take a detour to understand signals, power spectrums, and noise.

Signal: A **signal** is any quantity that varies in time. One example is sound, which is variation in air density. In the sand pile model, the signals we'll consider are avalanche durations and sizes as they vary over time.

Power spectrum: Any signal can be decomposed into a set of frequency components with different levels of **power**, which is related to amplitude or volume. The **power spectrum** of a signal is a function that shows the power of each frequency component.

Noise: In common use, **noise** is usually an unwanted sound, but in the context of signal processing, it is a signal that contains many frequency components.

There are many kinds of noise. For example, "white noise" is a signal that has components with equal power over a wide range of frequencies.

Other kinds of noise have different relationships between frequency and power. In "red noise", the power at frequency f is $1/f^2$, which we can write like this:

$$P(f) = 1/f^2$$

We can generalize this equation by replacing the exponent 2 with a parameter β:

$$P(f) = 1/f^\beta$$

When $\beta = 0$, this equation describes white noise; when $\beta = 2$ it describes red noise. When the parameter is near 1, the result is called $1/f$ noise. More generally, noise with any value between 0 and 2 is called "pink", because it's between white and red.

We can use this relationship to derive a test for pink noise. Taking the log of both sides yields

$$\log P(f) = -\beta \log f$$

So if we plot $P(f)$ versus f on a log-log scale, we expect a straight line with slope $-\beta$.

What does this have to do with the sand pile model? Suppose that every time a cell topples, it makes a sound. If we record a sand pile model while its running, what would it sound like? In the next section, we'll simulate the sound of the sand pile model and see if it is pink noise.

8.7 The sound of sand

As my implementation of `SandPile` runs, it records the number of cells that topple during each time step, accumulating the results in a list called `toppled_seq`. After running the model in Section 8.4, we can extract the resulting signal:

www.dbooks.org

```
signal = pile2.toppled_seq
```

To compute the power spectrum of this signal we can use the SciPy function `welch`:

```
from scipy.signal import welch

nperseg = 2048
freqs, spectrum = welch(signal, nperseg=nperseg, fs=nperseg)
```

This function uses Welch's method, which splits the signal into segments and computes the power spectrum of each segment. The result is typically noisy, so Welch's method averages across segments to estimate the average power at each frequency. For more about Welch's method, see http://thinkcomplex.com/welch.

The parameter `nperseg` specifies the number of time steps per segment. With longer segments, we can estimate the power for more frequencies. With shorter segments, we get better estimates for each frequency. The value I chose, 2048, balances these tradeoffs.

The parameter `fs` is the "sampling frequency", which is the number of data points in the signal per unit of time. By setting `fs=nperseg`, we get a range of frequencies from 0 to `nperseg/2`. This range is convenient, but because the units of time in the model are arbitrary, it doesn't mean much.

The return values, `freqs` and `powers`, are NumPy arrays containing the frequencies of the components and their corresponding powers, which we can plot. Figure 8.6 shows the result.

For frequencies between 10 and 1000 (in arbitrary units), the spectrum falls on a straight line, which is what we expect for pink or red noise.

The gray line in the figure has slope -1.58, which indicates that

$$\log P(f) \sim -\beta \log f$$

with parameter $\beta = 1.58$, which is the same parameter reported by Bak, Tang, and Wiesenfeld. This result confirms that the sand pile model generates pink noise.

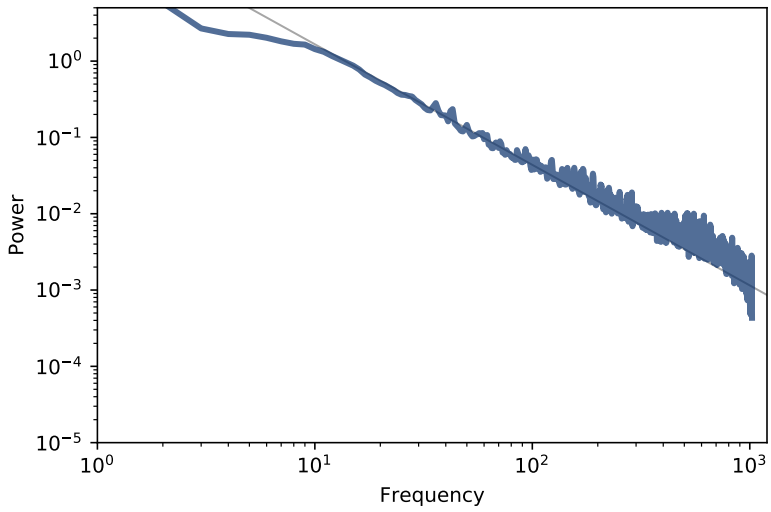


Figure 8.6: Power spectrum of the number of toppled cells over time, log-log scale.

8.8 Reductionism and Holism

The original paper by Bak, Tang and Wiesenfeld is one of the most frequently-cited papers in the last few decades. Some subsequent papers have reported other systems that are apparently self-organized critical (SOC). Others have studied the sand pile model in more detail.

As it turns out, the sand pile model is not a good model of a sand pile. Sand is dense and not very sticky, so momentum has a non-negligible effect on the behavior of avalanches. As a result, there are fewer very large and very small avalanches than the model predicts, and the distribution might not be heavy-tailed.

Bak has suggested that this observation misses the point. The sand pile model is not meant to be a realistic model of a sand pile; it is meant to be a simple example of a broad category of models.

To understand this point, it is useful to think about two kinds of models, **reductionist** and **holistic**. A reductionist model describes a system by describing its parts and their interactions. When a reductionist model is used

www.dbooks.org

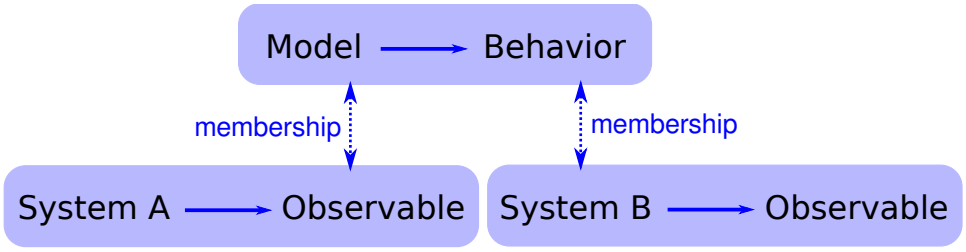


Figure 8.7: The logical structure of a holistic model.

as an explanation, it depends on an analogy between the components of the model and the components of the system.

For example, to explain why the ideal gas law holds, we can model the molecules that make up a gas with point masses and model their interactions as elastic collisions. If you simulate or analyze this model, you find that it obeys the ideal gas law. This model is satisfactory to the degree that molecules in a gas behave like molecules in the model. The analogy is between the parts of the system and the parts of the model.

Holistic models are more focused on similarities between systems and less interested in analogous parts. A holistic approach to modeling consists of these steps:

- Observe a behavior that appears in a variety of systems.
- Find a simple model that demonstrates that behavior.
- Identify the elements of the model that are necessary and sufficient to produce the behavior.

For example, in *The Selfish Gene*, Richard Dawkins suggests that genetic evolution is just one example of an evolutionary system. He identifies the essential elements of the category — discrete replicators, variability, and differential reproduction — and proposes that any system with these elements will show evidence of evolution.

As another example of an evolutionary system, he proposes "memes", which are thoughts or behaviors that are replicated by transmission from person to

person[2]. As memes compete for the resource of human attention, they evolve in ways that are similar to genetic evolution.

Critics of the meme model have pointed out that memes are a poor analogy for genes; they differ from genes in many obvious ways. Dawkins has argued that these differences are beside the point because memes are not *supposed* to be analogous to genes. Rather, memes and genes are examples of the same category: evolutionary systems. The differences between them emphasize the real point, which is that evolution is a general model that applies to many seemingly disparate systems. The logical structure of this argument is shown in Figure 8.7.

Bak has made a similar argument that self-organized criticality is a general model for a broad category of systems:

> Since these phenomena appear everywhere, they cannot depend on any specific detail whatsoever... If the physics of a large class of problems is the same, this gives [the theorist] the option of selecting the *simplest* possible [model] belonging to that class for detailed study.[3]

Many natural systems demonstrate behaviors characteristic of critical systems. Bak's explanation for this prevalence is that these systems are examples of the broad category of self-organized criticality. There are two ways to support this argument. One is to build a realistic model of a particular system and show that the model exhibits SOC. The second is to show that SOC is a feature of many diverse models, and to identify the essential characteristics those models have in common.

The first approach, which I characterize as reductionist, can explain the behavior of a particular system. The second approach, which I am calling holistic, can explain the prevalence of criticality in natural systems. They are different models with different purposes.

For reductionist models, realism is the primary virtue, and simplicity is secondary. For holistic models, it is the other way around.

[2]This use of "meme" is original to Dawkins, and predates the distantly-related use of the word on the Internet by about 20 years.

[3]Bak, *How Nature Works*, Springer-Verlag 1996, page 43.

www.dbooks.org

8.9 SOC, causation, and prediction

If a stock market index drops by a fraction of a percent in a day, there is no need for an explanation. But if it drops 10%, people want to know why. Pundits on television are willing to offer explanations, but the real answer may be that there is no explanation.

Day-to-day variability in the stock market shows evidence of criticality: the distribution of value changes is heavy-tailed and the time series exhibits pink noise. If the stock market is a critical system, we should expect occasional large changes as part of the ordinary behavior of the market.

The distribution of earthquake sizes is also heavy-tailed, and there are simple models of the dynamics of geological faults that might explain this behavior. If these models are right, they imply that large earthquakes are not exceptional; that is, they do not require explanation any more than small earthquakes do.

Similarly, Charles Perrow has suggested that failures in large engineered systems, like nuclear power plants, are like avalanches in the sand pile model. Most failures are small, isolated, and harmless, but occasionally a coincidence of bad fortune yields a catastrophe. When big accidents occur, investigators go looking for the cause, but if Perrow's "normal accident theory" is correct, there may be no special cause of large failures.

These conclusions are not comforting. Among other things, they imply that large earthquakes and some kinds of accidents are fundamentally unpredictable. It is impossible to look at the state of a critical system and say whether a large avalanche is "due". If the system is in a critical state, then a large avalanche is always possible. It just depends on the next grain of sand.

In a sand pile model, what is the cause of a large avalanche? Philosophers sometimes distinguish the **proximate** cause, which is most immediately responsible, from the **ultimate** cause, which is considered some deeper kind of explanation (see http://thinkcomplex.com/cause).

In the sand pile model, the proximate cause of an avalanche is a grain of sand, but the grain that causes a large avalanche is identical to every other grain, so it offers no special explanation. The ultimate cause of a large avalanche is the structure and dynamics of the systems as a whole: large avalanches occur because they are a property of the system.

Many social phenomena, including wars, revolutions, epidemics, inventions, and terrorist attacks, are characterized by heavy-tailed distributions. If these distributions are prevalent because social systems are SOC, major historical events may be fundamentally unpredictable and unexplainable.

8.10 Exercises

The code for this chapter is in the Jupyter notebook `chap08.ipynb` in the repository for this book. Open this notebook, read the code, and run the cells. You can use this notebook to work on the following exercises. My solutions are in `chap08soln.ipynb`.

Exercise 8.1 To test whether the distributions of `T` and `S` are heavy-tailed, we plotted their PMFs on a log-log scale, which is what Bak, Tang and Wiesenfeld show in their paper. But as we saw in Section 4.7, this visualization can obscure the shape of the distribution. Using the same data, make a plot that shows the cumulative distributions (CDFs) of `S` and `T`. What can you say about their shape? Do they follow a power law? Are they heavy-tailed?

You might find it helpful to plot the CDFs on a log-x scale and on a log-log scale.

Exercise 8.2 In Section 8.5 we showed that the initial configuration of the sand pile model produces fractal patterns. But after we drop a large number of random grains, the patterns look more random.

Starting with the example in Section 8.5, run the sand pile model for a while and then compute fractal dimensions for each of the 4 levels. Is the sand pile model fractal in steady state?

Exercise 8.3 Another version of the sand pile model, called the "single source" model, starts from a different initial condition: instead of all cells at the same level, all cells are set to 0 except the center cell, which is set to a large value. Write a function that creates a `SandPile` object, sets up the single source initial condition, and runs until the pile reaches equilibrium. Does the result appear to be fractal?

You can read more about this version of the sand pile model at http://thinkcomplex.com/sand.

www.dbooks.org

Exercise 8.4 In their 1989 paper, Bak, Chen and Creutz suggest that the Game of Life is a self-organized critical system (see http://thinkcomplex.com/bak89).

To replicate their tests, start with a random configuration and run the GoL CA until it stabilizes. Then choose a random cell and flip it. Run the CA until it stabilizes again, keeping track of `T`, the number of time steps it takes, and `S`, the number of cells affected. Repeat for a large number of trials and plot the distributions of `T` and `S`. Also, estimate the power spectrums of `T` and `S` as signals in time, and see if they are consistent with pink noise.

Exercise 8.5 In *The Fractal Geometry of Nature*, Benoit Mandelbrot proposes what he calls a "heretical" explanation for the prevalence of heavy-tailed distributions in natural systems. It may not be, as Bak suggests, that many systems can generate this behavior in isolation. Instead there may be only a few, but interactions between systems might cause the behavior to propagate.

To support this argument, Mandelbrot points out:

- The distribution of observed data is often "the joint effect of a fixed underlying *true distribution* and a highly variable *filter*".
- Heavy-tailed distributions are robust to filtering; that is, "a wide variety of filters leave their asymptotic behavior unchanged".

What do you think of this argument? Would you characterize it as reductionist or holist?

Exercise 8.6 Read about the "Great Man" theory of history at http://thinkcomplex.com/great. What implication does self-organized criticality have for this theory?

www.dbooks.org

Chapter 9

Agent-based models

The models we have seen so far might be characterized as "rule-based" in the sense that they involve systems governed by simple rules. In this and the following chapters, we explore **agent-based models**.

Agent-based models include **agents** that are intended to model people and other entities that gather information about the world, make decisions, and take actions.

The agents are usually situated in space or in a network, and interact with each other locally. They usually have imperfect or incomplete information about the world.

Often there are differences among agents, unlike previous models where all components are identical. And agent-based models often include randomness, either among the agents or in the world.

Since the 1970s, agent-based modeling has become an important tool in economics, other social sciences, and some natural sciences.

Agent-based models are useful for modeling the dynamics of systems that are not in equilibrium (although they are also used to study equilibrium). And they are particularly useful for understanding relationships between individual decisions and system behavior.

The code for this chapter is in `chap09.ipynb`, which is a Jupyter notebook in the repository for this book. For more information about working with this code, see Section 0.3.

9.1 Schelling's Model

In 1969 Thomas Schelling published "Models of Segregation", which proposed a simple model of racial segregation. You can read it at http://thinkcomplex.com/schell.

The Schelling model of the world is a grid where each cell represents a house. The houses are occupied by two kinds of agents, labeled red and blue, in roughly equal numbers. About 10% of the houses are empty.

At any point in time, an agent might be happy or unhappy, depending on the other agents in the neighborhood, where the "neighborhood" of each house is the set of eight adjacent cells. In one version of the model, agents are happy if they have at least two neighbors like themselves, and unhappy if they have one or zero.

The simulation proceeds by choosing an agent at random and checking to see whether they are happy. If so, nothing happens; if not, the agent chooses one of the unoccupied cells at random and moves.

You will not be surprised to hear that this model leads to some segregation, but you might be surprised by the degree. From a random starting point, clusters of similar agents form almost immediately. The clusters grow and coalesce over time until there are a small number of large clusters and most agents live in homogeneous neighborhoods.

If you did not know the process and only saw the result, you might assume that the agents were racist, but in fact all of them would be perfectly happy in a mixed neighborhood. Since they prefer not to be greatly outnumbered, they might be considered mildly xenophobic. Of course, these agents are a wild simplification of real people, so it may not be appropriate to apply these descriptions at all.

Racism is a complex human problem; it is hard to imagine that such a simple model could shed light on it. But in fact it provides a strong argument about the relationship between a system and its parts: if you observe segregation in a real city, you cannot conclude that individual racism is the immediate cause, or even that the people in the city are racists.

www.dbooks.org

Of course, we have to keep in mind the limitations of this argument: Schelling's model demonstrates a possible cause of segregation, but says nothing about actual causes.

9.2 Implementation of Schelling's model

To implement Schelling's model, I wrote yet another class that inherits from `Cell2D`:

```
class Schelling(Cell2D):

    def __init__(self, n, p):
        self.p = p
        choices = [0, 1, 2]
        probs = [0.1, 0.45, 0.45]
        self.array = np.random.choice(choices, (n, n), p=probs)
```

`n` is the size of the grid, and `p` is the threshold on the fraction of similar neighbors. For example, if `p=0.3`, an agent will be unhappy if fewer than 30% of their neighbors are the same color.

`array` is a NumPy array where each cell is 0 if empty, 1 if occupied by a red agent, and 2 if occupied by a blue agent. Initially 10% of the cells are empty, 45% red, and 45% blue.

The `step` function for Schelling's model is substantially more complicated than previous examples. If you are not interested in the details, you can skip to the next section. But if you stick around, you might pick up some NumPy tips.

First, I make boolean arrays that indicate which cells are red, blue, and empty:

```
        a = self.array
        red = a==1
        blue = a==2
        empty = a==0
```

Then I use `correlate2d` to count, for each location, the number of neighboring cells that are red, blue, and non-empty. We saw `correlate2d` in Section 6.6.

```
    options = dict(mode='same', boundary='wrap')

    kernel = np.array([[1, 1, 1],
                       [1, 0, 1],
                       [1, 1, 1]], dtype=np.int8)

    num_red = correlate2d(red, kernel, **options)
    num_blue = correlate2d(blue, kernel, **options)
    num_neighbors = num_red + num_blue
```

`options` is a dictionary that contains the options we pass to `correlate2d`. With `mode='same'`, the result is the same size as the input. With `boundary='wrap'`, the top edge is wrapped to meet the bottom, and the left edge is wrapped to meet the right.

`kernel` indicates that we want to consider the eight neighbors that surround each cell.

After computing `num_red` and `num_blue`, we can compute the fraction of neighbors, for each location, that are red and blue.

```
    frac_red = num_red / num_neighbors
    frac_blue = num_blue / num_neighbors
```

Then, we can compute the fraction of neighbors, for each agent, that are the same color as the agent. I use `np.where`, which is like an element-wise `if` expression. The first parameter is a condition that selects elements from the second or third parameter.

```
    frac_same = np.where(red, frac_red, frac_blue)
    frac_same[empty] = np.nan
```

In this case, wherever `red` is `True`, `frac_same` gets the corresponding element of `frac_red`. Where `red` is `False`, `frac_same` gets the corresponding element of `frac_blue`. Finally, where `empty` indicates that a cell is empty, `frac_same` is set to `np.nan`, which is a special value that indicates "Not a Number".

Now we can identify the locations of the unhappy agents:

```
    unhappy = frac_same < self.p
    unhappy_locs = locs_where(unhappy)
```

www.dbooks.org

`locs_where` is a wrapper function for `np.nonzero`:

```
def locs_where(condition):
    return list(zip(*np.nonzero(condition)))
```

`np.nonzero` takes an array and returns the coordinates of all non-zero cells; the result is a tuple of arrays, one for each dimension. Then `locs_where` uses `list` and `zip` to convert this result to a list of coordinate pairs.

Similarly, `empty_locs` is an array that contains the coordinates of the empty cells:

```
    empty_locs = locs_where(empty)
```

Now we get to the core of the simulation. We loop through the unhappy agents and move them:

```
    num_empty = np.sum(empty)
    for source in unhappy_locs:
        i = np.random.randint(num_empty)
        dest = empty_locs[i]

        a[dest] = a[source]
        a[source] = 0
        empty_locs[i] = source
```

`i` is the index of a random empty cell; `dest` is a tuple containing the coordinates of the empty cell.

In order to move an agent, we copy its value (1 or 2) from `source` to `dest`, and then set the value of `source` to 0 (since it is now empty).

Finally, we replace the entry in `empty_locs` with `source`, so the cell that just became empty can be chosen by the next agent.

9.3 Segregation

Now let's see what happens when we run the model. I'll start with `n=100` and `p=0.3`, and run for 10 steps.

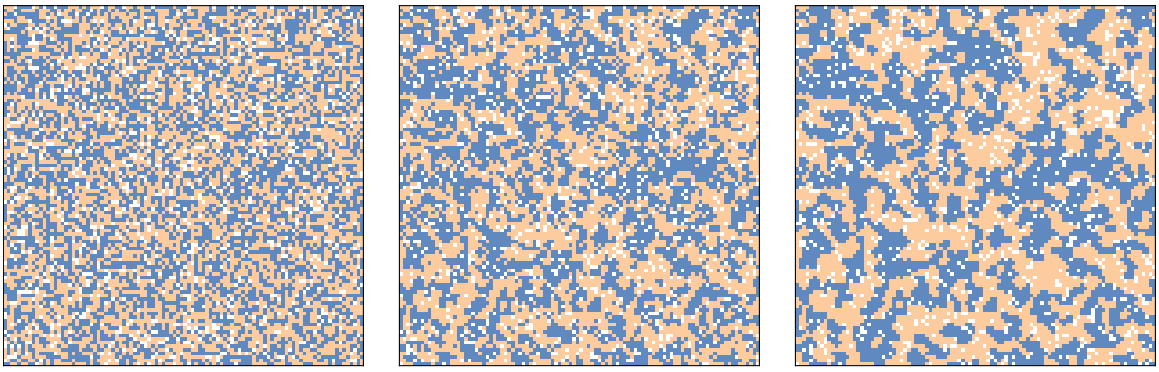


Figure 9.1: Schelling's segregation model with `n=100`, initial condition (left), after 2 steps (middle), and after 10 steps (right).

```
grid = Schelling(n=100, p=0.3)
for i in range(10):
    grid.step()
```

Figure 9.1 shows the initial configuration (left), the state of the simulation after 2 steps (middle), and the state after 10 steps (right).

Clusters form almost immediately and grow quickly, until most agents live in highly-segregated neighborhoods.

As the simulation runs, we can compute the degree of segregation, which is the average, across agents, of the fraction of neighbors who are the same color as the agent:

```
    np.nanmean(frac_same)
```

In Figure 9.1, the average fraction of similar neighbors is 50% in the initial configuration, 65% after two steps, and 76% after 10 steps!

Remember that when `p=0.3` the agents would be happy if 3 of 8 neighbors were their own color, but they end up living in neighborhoods where 6 or 7 of their neighbors are their own color, typically.

www.dbooks.org

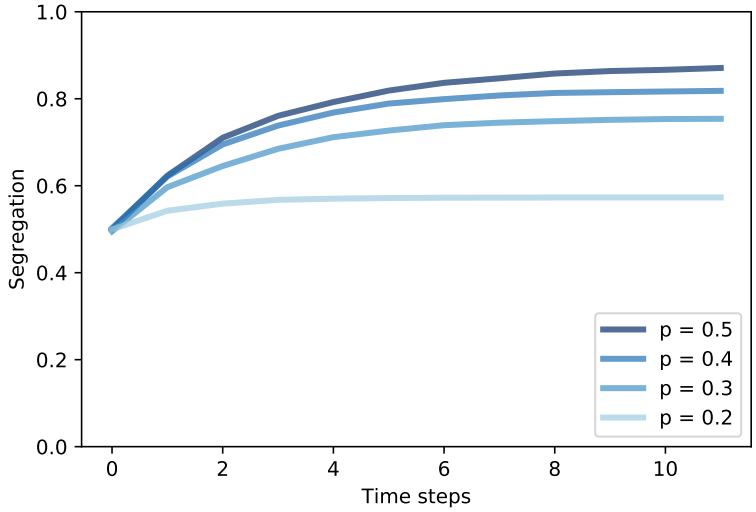


Figure 9.2: Degree of segregation in Schelling's model, over time, for a range of `p`.

Figure 9.2 shows how the degree of segregation increases and where it levels off for several values of `p`. When `p=0.4`, the degree of segregation in steady state is about 82%, and a majority of agents have no neighbors with a different color.

These results are surprising to many people, and they make a striking example of the unpredictable relationship between individual decisions and system behavior.

9.4 Sugarscape

In 1996 Joshua Epstein and Robert Axtell proposed Sugarscape, an agent-based model of an "artificial society" intended to support experiments related to economics and other social sciences.

Sugarscape is a versatile model that has been adapted for a wide variety of topics. As examples, I will replicate the first few experiments from Epstein and Axtell's book, *Growing Artificial Societies*.

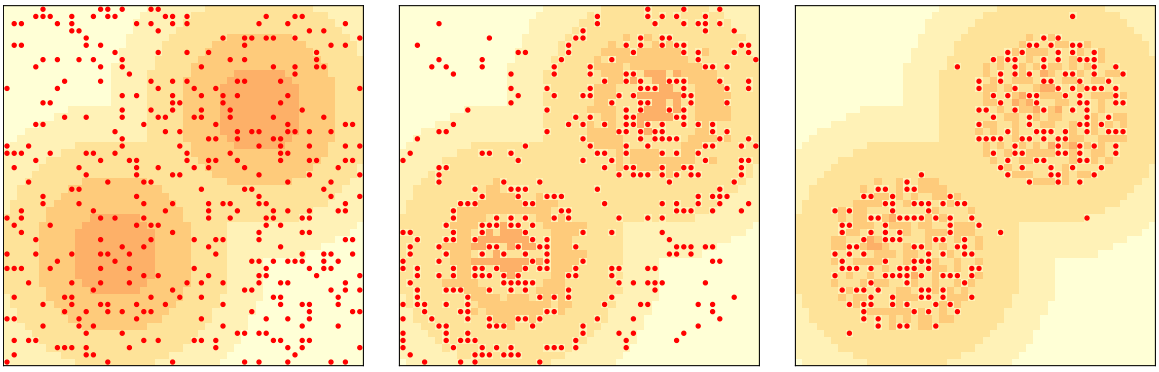


Figure 9.3: Replication of the original Sugarscape model: initial configuration (left), after 2 steps (middle) and after 100 steps (right).

In its simplest form, Sugarscape is a model of a simple economy where agents move around on a 2-D grid, harvesting and accumulating "sugar", which represents economic wealth. Some parts of the grid produce more sugar than others, and some agents are better at finding it than others.

This version of Sugarscape is often used to explore and explain the distribution of wealth, in particular the tendency toward inequality.

In the Sugarscape grid, each cell has a capacity, which is the maximum amount of sugar it can hold. In the original configuration, there are two high-sugar regions, with capacity 4, surrounded by concentric rings with capacities 3, 2, and 1.

Figure 9.3 (left) shows the initial configuration, with the darker areas indicating cells with higher capacity, and small dots representing the agents.

Initially there are 400 agents placed at random locations. Each agent has three randomly-chosen attributes:

Sugar: Each agent starts with an endowment of sugar chosen from a uniform distribution between 5 and 25 units.

www.dbooks.org

Metabolism: Each agent has some amount of sugar they must consume per time step, chosen uniformly between 1 and 4.

Vision: Each agent can "see" the amount of sugar in nearby cells and move to the cell with the most, but some agents can see and move farther than others. The distance agents see is chosen uniformly between 1 and 6.

During each time step, agents move one at a time in a random order. Each agent follows these rules:

- The agent surveys `k` cells in each of the 4 compass directions, where `k` is the range of the agent's vision.
- It chooses the unoccupied cell with the most sugar. In case of a tie, it chooses the closer cell; among cells at the same distance, it chooses randomly.
- The agent moves to the selected cell and harvests the sugar, adding the harvest to its accumulated wealth and leaving the cell empty.
- The agent consumes some part of its wealth, depending on its metabolism. If the resulting total is negative, the agent "starves" and is removed.

After all agents have executed these steps, the cells grow back some sugar, typically 1 unit, but the total sugar in each cell is bounded by its capacity.

Figure 9.3 (middle) shows the state of the model after two steps. Most agents are moving toward the areas with the most sugar. Agents with high vision move the fastest; agents with low vision tend to get stuck on the plateaus, wandering randomly until they get close enough to see the next level.

Agents born in the areas with the least sugar are likely to starve unless they have a high initial endowment and high vision.

Within the high-sugar areas, agents compete with each other to find and harvest sugar as it grows back. Agents with high metabolism or low vision are the most likely to starve.

When sugar grows back at 1 unit per time step, there is not enough sugar to sustain the 400 agents we started with. The population drops quickly at first, then more slowly, and levels off around 250.

Figure 9.3 (right) shows the state of the model after 100 time steps, with about 250 agents. The agents who survive tend to be the lucky ones, born with high vision and/or low metabolism. Having survived to this point, they are likely to survive forever, accumulating unbounded stockpiles of sugar.

9.5 Wealth inequality

In its current form, Sugarscape models a simple ecology, and could be used to explore the relationship between the parameters of the model, like the growth rate and the attributes of the agents, and the carrying capacity of the system (the number of agents that survive in steady state). And it models a form of natural selection, where agents with higher "fitness" are more likely to survive.

The model also demonstrates a kind of wealth inequality, with some agents accumulating sugar faster than others. But it would be hard to say anything specific about the distribution of wealth because it is not "stationary"; that is, the distribution changes over time and does not reach a steady state.

However, if we give the agents finite lifespans, the model produces a stationary distribution of wealth. Then we can run experiments to see what effect the parameters and rules have on this distribution.

In this version of the model, agents have an age that gets incremented each time step, and a random lifespan chosen from a uniform distribution between 60 to 100. If an agent's age exceeds its lifespan, it dies.

When an agent dies, from starvation or old age, it is replaced by a new agent with random attributes, so the number of agents is constant.

Starting with 250 agents (which is close to carrying capacity) I run the model for 500 steps. After each 100 steps, I plot the cumulative distribution function (CDF) of sugar accumulated by the agents. We saw CDFs in Section 4.7. Figure 9.4 shows the results on a linear scale (left) and a log-x scale (right).

After about 200 steps (which is twice the longest lifespan) the distribution doesn't change much. And it is skewed to the right.

Most agents have little accumulated wealth: the 25th percentile is about 10 and the median is about 20. But a few agents have accumulated much more: the 75th percentile is about 40, and the highest value is more than 150.

www.dbooks.org

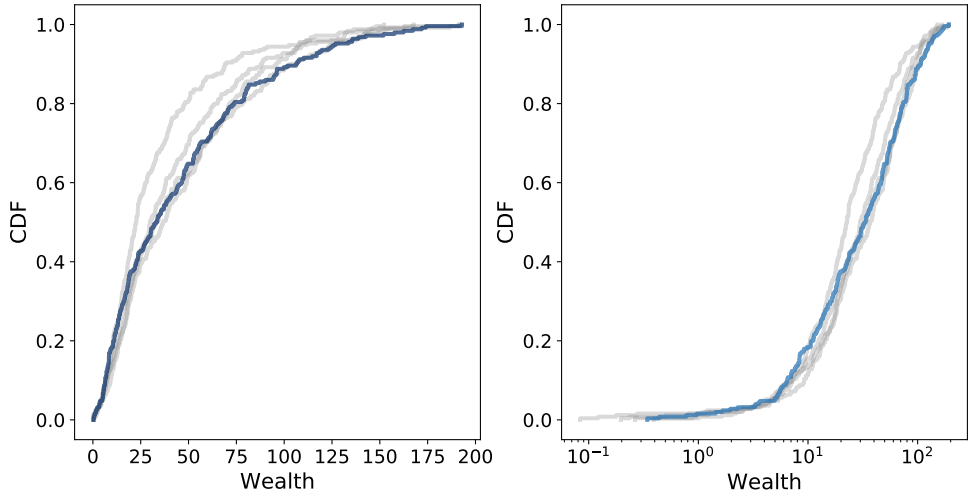


Figure 9.4: Distribution of sugar (wealth) after 100, 200, 300, and 400 steps (gray lines) and 500 steps (dark line). Linear scale (left) and log-x scale (right).

On a log scale the shape of the distribution resembles a Gaussian or normal distribution, although the right tail is truncated. If it were actually normal on a log scale, the distribution would be lognormal, which is a heavy-tailed distribution. And in fact, the distribution of wealth in practically every country, and in the world, is a heavy-tailed distribution.

It would be too much to claim that Sugarscape explains why wealth distributions are heavy-tailed, but the prevalence of inequality in variations of Sugarscape suggests that inequality is characteristic of many economies, even very simple ones. And experiments with rules that model taxation and other income transfers suggest that it is not easy to avoid or mitigate.

9.6 Implementing Sugarscape

Sugarscape is more complicated than the previous models, so I won't present the entire implementation here. I will outline the structure of the code and you can see the details in the Jupyter notebook for this chapter, `chap09.ipynb`, which is in the repository for this book. If you are not interested in the details, you can skip this section.

During each step, the agent moves, harvests sugar, and ages. Here is the `Agent` class and its `step` method:

```
class Agent:

    def step(self, env):
        self.loc = env.look_and_move(self.loc, self.vision)
        self.sugar += env.harvest(self.loc) - self.metabolism
        self.age += 1
```

The parameter `env` is a reference to the environment, which is a `Sugarscape` object. It provides methods `look_and_move` and `harvest`:

- `look_and_move` takes the location of the agent, which is a tuple of coordinates, and the range of the agent's vision, which is an integer. It returns the agent's new location, which is the visible cell with the most sugar.

- `harvest` takes the (new) location of the agent, and removes and returns the sugar at that location.

`Sugarscape` inherits from `Cell2D`, so it is similar to the other grid-based models we've seen.

The attributes include `agents`, which is a list of `Agent` objects, and `occupied`, which is a set of tuples, where each tuple contains the coordinates of a cell occupied by an agent.

Here is the `Sugarscape` class and its `step` method:

www.dbooks.org

```
class Sugarscape(Cell2D):

    def step(self):

        # loop through the agents in random order
        random_order = np.random.permutation(self.agents)
        for agent in random_order:

            # mark the current cell unoccupied
            self.occupied.remove(agent.loc)

            # execute one step
            agent.step(self)

            # if the agent is dead, remove from the list
            if agent.is_starving():
                self.agents.remove(agent)
            else:
                # otherwise mark its cell occupied
                self.occupied.add(agent.loc)

        # grow back some sugar
        self.grow()
        return len(self.agents)
```

During each step, the `Sugarscape` uses the NumPy function `permutation` so it loops through the agents in random order. It invokes `step` on each agent and then checks whether it is dead. After all agents have moved, some of the sugar grows back. The return value is the number of agents still alive.

I won't show more details here; you can see them in the notebook for this chapter. If you want to learn more about NumPy, you might want to look at these functions in particular:

- `make_visible_locs`, which builds the array of locations an agent can see, depending on its vision. The locations are sorted by distance, with locations at the same distance appearing in random order. This function uses `np.random.shuffle` and `np.vstack`.

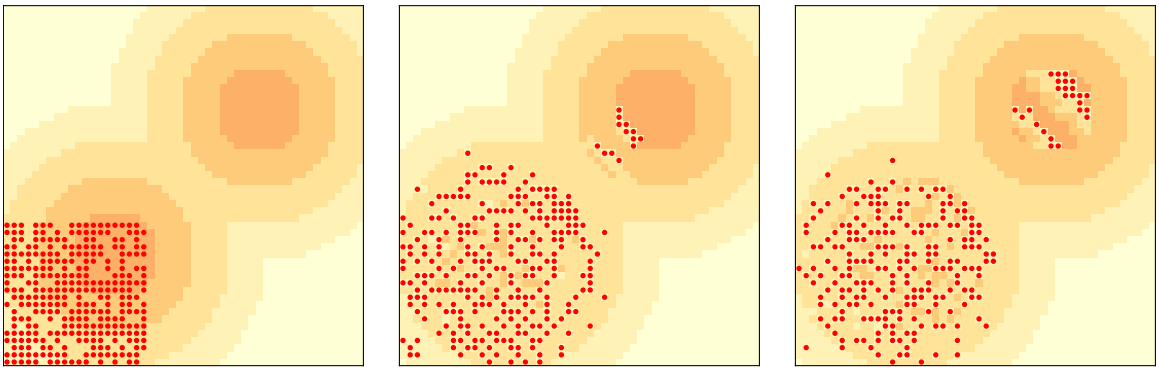


Figure 9.5: Wave behavior in Sugarscape: initial configuration (left), after 6 steps (middle) and after 12 steps (right).

- `make_capacity`, which initializes the capacity of the cells using NumPy functions `indices`, `hypot`, `minimum`, and `digitize`.
- `look_and_move`, which uses `argmax`.

9.7 Migration and Wave Behavior

Although the purpose of Sugarscape is not primarily to explore the movement of agents in space, Epstein and Axtell observed some interesting patterns when agents migrate.

If we start with all agents in the lower-left corner, they quickly move toward the closest "peak" of high-capacity cells. But if there are more agents than a single peak can support, they quickly exhaust the sugar and agents are forced to move into lower-capacity areas.

The ones with the longest vision cross the valley between the peaks and propagate toward the northeast in a pattern that resembles a wave front. Because

www.dbooks.org

they leave a stripe of empty cells behind them, other agents don't follow until the sugar grows back.

The result is a series of discrete waves of migration, where each wave resembles a coherent object, like the spaceships we saw in the Rule 110 CA and Game of Life (see Section 5.6 and Section 6.2).

Figure 9.5 shows the initial condition (left) and the state of the model after 6 steps (middle) and 12 steps (right). You can see the first two waves reaching and moving through the second peak, leaving a stripe of empty cells behind. You can see an animated version of this model, where the wave patterns are more clearly visible, in the notebook for this chapter.

These waves move diagonally, which is surprising because the agents themselves only move north or east, never northeast. Outcomes like this — groups or "aggregates" with properties and behaviors that the agents don't have — are common in agent-based models. We will see more examples in the next chapter.

9.8 Emergence

The examples in this chapter demonstrate one of the most important ideas in complexity science: emergence. An **emergent property** is a characteristic of a system that results from the interaction of its components, not from their properties.

To clarify what emergence is, it helps to consider what it isn't. For example, a brick wall is hard because bricks and mortar are hard, so that's not an emergent property. As another example, some rigid structures are built from flexible components, so that seems like a kind of emergence. But it is at best a weak kind, because structural properties follow from well understood laws of mechanics.

In contrast, the segregation we see in Schelling's model is an emergent property because it is not caused by racist agents. Even when the agents are only mildly xenophobic, the outcome of the system is substantially different from the intention of the agent's decisions.

The distribution of wealth in Sugarscape might be an emergent property, but it is a weak example because we could reasonably predict it based on the distributions of vision, metabolism, and lifespan. The wave behavior we saw in the last example might be a stronger example, since the wave displays a capability — diagonal movement — that the agents do not have.

Emergent properties are surprising: it is hard to predict the behavior of the system even if we know all the rules. That difficulty is not an accident; in fact, it may be the defining characteristic of emergence.

As Wolfram discusses in *A New Kind of Science*, conventional science is based on the axiom that if you know the rules that govern a system, you can predict its behavior. What we call "laws" are often computational shortcuts that allow us to predict the outcome of a system without building or observing it.

But many cellular automatons are **computationally irreducible**, which means that there are no shortcuts. The only way to get the outcome is to implement the system.

The same may be true of complex systems in general. For physical systems with more than a few components, there is usually no model that yields an analytic solution. Numerical methods provide a kind of computational shortcut, but there is still a qualitative difference.

Analytic solutions often provide a constant-time algorithm for prediction; that is, the run time of the computation does not depend on t, the time scale of prediction. But numerical methods, simulation, analog computation, and similar methods take time proportional to t. And for many systems, there is a bound on t beyond which we can't compute reliable predictions at all.

These observations suggest that emergent properties are fundamentally unpredictable, and that for complex systems we should not expect to find natural laws in the form of computational shortcuts.

To some people, "emergence" is another name for ignorance; by this reckoning, a property is emergent if we don't have a reductionist explanation for it, but if we come to understand it better in the future, it would no longer be emergent.

The status of emergent properties is a topic of debate, so it is appropriate to be skeptical. When we see an apparently emergent property, we should not

www.dbooks.org

assume that there can never be a reductionist explanation. But neither should we assume that there has to be one.

The examples in this book and the principle of computational equivalence give good reasons to believe that at least some emergent properties can never be "explained" by a classical reductionist model.

You can read more about emergence at http://thinkcomplex.com/emerge.

9.9 Exercises

The code for this chapter is in the Jupyter notebook `chap09.ipynb` in the repository for this book. Open this notebook, read the code, and run the cells. You can use this notebook to work on the following exercises. My solutions are in `chap09soln.ipynb`.

Exercise 9.1 Bill Bishop, author of *The Big Sort*, argues that American society is increasingly segregated by political opinion, as people choose to live among like-minded neighbors.

The mechanism Bishop hypothesizes is not that people, like the agents in Schelling's model, are more likely to move if they are isolated, but that when they move for any reason, they are likely to choose a neighborhood with people like themselves.

Modify your implementation of Schelling's model to simulate this kind of behavior and see if it yields similar degrees of segregation.

There are several ways you can model Bishop's hypothesis. In my implementation, a random selection of agents moves during each step. Each agent considers `k` randomly-chosen empty locations and chooses the one with the highest fraction of similar neighbors. How does the degree of segregation depend on `k`?

Exercise 9.2 In the first version of SugarScape, we never add agents, so once the population falls, it never recovers. In the second version, we only replace agents when they die, so the population is constant. Now let's see what happens if we add some "population pressure".

Write a version of SugarScape that adds a new agent at the end of every step. Add code to compute the average vision and the average metabolism of the agents at the end of each step. Run the model for a few hundred steps and plot the population over time, as well as the average vision and average metabolism.

You should be able to implement this model by inheriting from `SugarScape` and overriding `__init__` and `step`.

Exercise 9.3 Among people who study philosophy of mind, "Strong AI" is the theory that an appropriately-programmed computer could have a mind in the same sense that humans have minds.

John Searle presented a thought experiment called "The Chinese Room", intended to show that Strong AI is false. You can read about it at http://thinkcomplex.com/searle.

What is the **system reply** to the Chinese Room argument? How does what you have learned about emergence influence your reaction to the system response?

www.dbooks.org

Chapter 10

Herds, Flocks, and Traffic Jams

The agent-based models in the previous chapter are based on grids: the agents occupy discrete locations in two-dimensional space. In this chapter we consider agents that move is continuous space, including simulated cars on a one-dimensional highway and simulated birds in three-dimensional space.

The code for this chapter is in `chap10.ipynb`, which is a Jupyter notebook in the repository for this book. For more information about working with this code, see Section 0.3.

10.1 Traffic jams

What causes traffic jams? Sometimes there is an obvious cause, like an accident, a speed trap, or something else that disturbs the flow of traffic. But other times traffic jams appear for no apparent reason.

Agent-based models can help explain spontaneous traffic jams. As an example, I implement a highway simulation based on a model in Mitchell Resnick's book, *Turtles, Termites and Traffic Jams*.

Here's the class that represents the "highway":

```
class Highway:

    def __init__(self, n=10, length=1000, eps=0):
        self.length = length
        self.eps = eps

        # create the drivers
        locs = np.linspace(0, length, n, endpoint=False)
        self.drivers = [Driver(loc) for loc in locs]

        # and link them up
        for i in range(n):
            j = (i+1) % n
            self.drivers[i].next = self.drivers[j]
```

`n` is the number of cars, `length` is the length of the highway, and `eps` is the amount of random noise we'll add to the system.

`locs` contains the locations of the drivers; the NumPy function `linspace` creates an array of `n` locations equally spaced between `0` and `length`.

The `drivers` attribute is a list of `Driver` objects. The `for` loop links them so each `Driver` contains a reference to the next. The highway is circular, so the last `Driver` contains a reference to the first.

During each time step, the `Highway` moves each of the drivers:

```
# Highway

    def step(self):
        for driver in self.drivers:
            self.move(driver)
```

The `move` method lets the `Driver` choose its acceleration. Then `move` computes the updated speed and position. Here's the implementation:

www.dbooks.org

```
# Highway

    def move(self, driver):
        dist = self.distance(driver)

        # let the driver choose acceleration
        acc = driver.choose_acceleration(dist)
        acc = min(acc, self.max_acc)
        acc = max(acc, self.min_acc)
        speed = driver.speed + acc

        # add random noise to speed
        speed *= np.random.uniform(1-self.eps, 1+self.eps)

        # keep it nonnegative and under the speed limit
        speed = max(speed, 0)
        speed = min(speed, self.speed_limit)

        # if current speed would collide, stop
        if speed > dist:
            speed = 0

        # update speed and loc
        driver.speed = speed
        driver.loc += speed
```

`dist` is the distance between `driver` and the next driver ahead. This distance is passed to `choose_acceleration`, which specifies the behavior of the driver. This is the only decision the driver gets to make; everything else is determined by the "physics" of the simulation.

- `acc` is acceleration, which is bounded by `min_acc` and `max_acc`. In my implementation, cars can accelerate with `max_acc=1` and brake with `min_acc=-10`.
- `speed` is the old speed plus the requested acceleration, but then we make some adjustments. First, we add random noise to `speed`, because the world is not perfect. `eps` determines the magnitude of the relative error; for example, if `eps` is 0.02, `speed` is multiplied by a random factor between 0.98 and 1.02.

- Speed is bounded between 0 and `speed_limit`, which is 40 in my implementation, so cars are not allowed to go backward or speed.
- If the requested speed would cause a collision with the next car, `speed` is set to 0.
- Finally, we update the `speed` and `loc` attributes of `driver`.

Here's the definition for the `Driver` class:

```
class Driver:

    def __init__(self, loc, speed=0):
        self.loc = loc
        self.speed = speed

    def choose_acceleration(self, dist):
        return 1
```

The attributes `loc` and `speed` are the location and speed of the driver.

This implementation of `choose_acceleration` is simple: it always accelerates at the maximum rate.

Since the cars start out equally spaced, we expect them all to accelerate until they reach the speed limit, or until their speed exceeds the space between them. At that point, at least one "collision" will occur, causing some cars to stop.

Figure 10.1 shows a few steps in this process, starting with 30 cars and `eps=0.02`. On the left is the configuration after 16 time steps, with the highway mapped to a circle. Because of random noise, some cars are going faster than others, and the spacing has become uneven.

During the next time step (middle) there are two collisions, indicated by the triangles.

During the next time step (right) two cars collide with the stopped cars, and we can see the initial formation of a traffic jam. Once a jam forms, it tends to persist, with additional cars approaching from behind and colliding, and with cars in the front accelerating away.

Under some conditions, the jam itself propagates backwards, as you can see if you watch the animations in the notebook for this chapter.

www.dbooks.org

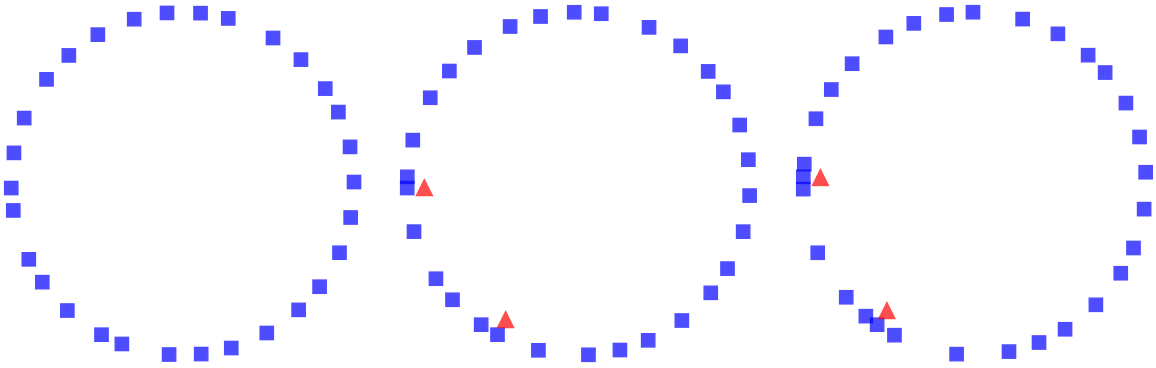


Figure 10.1: Simulation of drivers on a circular highway at three points in time. Squares indicate the position of the drivers; triangles indicate places where one driver has to brake to avoid another.

10.2 Random perturbation

As the number of cars increases, traffic jams become more severe. Figure 10.2 shows the average speed cars are able to achieve, as a function of the number of cars.

The top line shows results with `eps=0`, that is, with no random variation in speed. With 25 or fewer cars, the spacing between cars is greater than 40, which allows cars to reach and maintain the maximum speed, which is 40. With more than 25 cars, traffic jams form and the average speed drops quickly.

This effect is a direct result of the physics of the simulation, so it should not be surprising. If the length of the road is 1000, the spacing between `n` cars is `1000/n`. And since cars can't move faster than the space in front of them, the highest average speed we expect is `1000/n` or 40, whichever is less.

But that's the best case scenario. With just a small amount of randomness, things get much worse.

Figure 10.2 also shows results with `eps=0.001` and `eps=0.01`, which correspond to errors in speed of 0.1% and 1%.

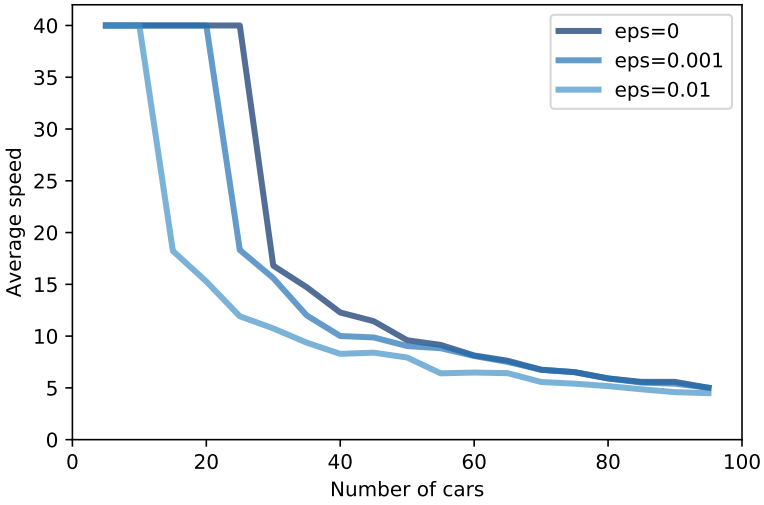


Figure 10.2: Average speed as a function of the number of cars, for three magnitudes of added random noise.

With 0.1% errors, the capacity of the highway drops from 25 to 20 (by "capacity" I mean the maximum number of cars that can reach and sustain the speed limit). And with 1% errors, the capacity drops to 10. Ugh.

As one of the exercises at the end of this chapter, you'll have a chance to design a better driver; that is, you will experiment with different strategies in `choose_acceleration` and see if you can find driver behaviors that improve average speed.

10.3 Boids

In 1987 Craig Reynolds published "Flocks, herds and schools: A distributed behavioral model", which describes an agent-based model of herd behavior. You can download his paper from http://thinkcomplex.com/boid.

Agents in this model are called "Boids", which is both a contraction of "bird-oid" and an accented pronunciation of "bird" (although Boids are also used to model fish and herding land animals).

Each agent simulates three behaviors:

www.dbooks.org

Flock centering: Move toward the center of the flock.

Collision avoidance: Avoid obstacles, including other Boids.

Velocity matching: Align velocity (speed and direction) with neighboring Boids.

Boids make decisions based on local information only; each Boid only sees (or pays attention to) other Boids in its field of vision.

In the repository for this book, you will find `Boids7.py`, which contains my implementation of Boids, based in part on the description in Gary William Flake's book, *The Computational Beauty of Nature*.

My implementation uses VPython, which is a library that provides 3-D graphics. VPython provides a `Vector` object, which I use to represent the position and velocity of Boids in three dimensions. You can read about them at http://thinkcomplex.com/vector.

10.4 The Boid algorithm

`Boids7.py` defines two classes: `Boid`, which implements the Boid behaviors, and `World`, which contains a list of Boids and a "carrot" the Boids are attracted to.

The `Boid` class defines the following methods:

- `center`: Finds other Boids within range and computes a vector toward their centroid.
- `avoid`: Finds objects, including other Boids, within a given range, and computes a vector that points away from their centroid.
- `align`: Finds other Boids within range and computes the average of their headings.
- `love`: Computes a vector that points toward the carrot.

Here's the implementation of `center`:

```
    def center(self, boids, radius=1, angle=1):
        neighbors = self.get_neighbors(boids, radius, angle)
        vecs = [boid.pos for boid in neighbors]
        return self.vector_toward_center(vecs)
```

The parameters `radius` and `angle` are the radius and angle of the field of view, which determines which other Boids are taken into consideration. `radius` is in arbitrary units of length; `angle` is in radians.

`center` uses `get_neighbors` to get a list of `Boid` objects that are in the field of view. `vecs` is a list of `Vector` objects that represent their positions.

Finally, `vector_toward_center` computes a `Vector` that points from `self` to the centroid of `neighbors`.

Here's how `get_neighbors` works:

```
    def get_neighbors(self, boids, radius, angle):
        neighbors = []
        for boid in boids:
            if boid is self:
                continue

            # if not in range, skip it
            offset = boid.pos - self.pos
            if offset.mag > radius:
                continue

            # if not within viewing angle, skip it
            if self.vel.diff_angle(offset) > angle:
                continue

            # otherwise add it to the list
            neighbors.append(boid)

        return neighbors
```

For each other Boid, `get_neighbors` uses vector subtraction to compute the

www.dbooks.org

vector from `self` to `boid`. The magnitude of this vector is the distance between them; if this magnitude exceeds `radius`, we ignore `boid`.

`diff_angle` computes the angle between the velocity of `self`, which points in the direction the Boid is heading, and the position of `boid`. If this angle exceeds `angle`, we ignore `boid`.

Otherwise `boid` is in view, so we add it to `neighbors`.

Now here's the implementation of `vector_toward_center`, which computes a vector from `self` to the centroid of its neighbors.

```
    def vector_toward_center(self, vecs):
        if vecs:
            center = np.mean(vecs)
            toward = vector(center - self.pos)
            return limit_vector(toward)
        else:
            return null_vector
```

VPython vectors work with NumPy, so `np.mean` computes the mean of `vecs`, which is a sequence of vectors. `limit_vector` limits the magnitude of the result to 1; `null_vector` has magnitude 0.

We use the same helper methods to implement `avoid`:

```
    def avoid(self, boids, carrot, radius=0.3, angle=np.pi):
        objects = boids + [carrot]
        neighbors = self.get_neighbors(objects, radius, angle)
        vecs = [boid.pos for boid in neighbors]
        return -self.vector_toward_center(vecs)
```

`avoid` is similar to `center`, but it takes into account the carrot as well as the other Boids. Also, the parameters are different: `radius` is smaller, so Boids only avoid objects that are too close, and `angle` is wider, so Boids avoid objects from all directions. Finally, the result from `vector_toward_center` is negated, so it points *away* from the centroid of any objects that are too close.

Here's the implementation of `align`:

```
    def align(self, boids, radius=0.5, angle=1):
        neighbors = self.get_neighbors(boids, radius, angle)
        vecs = [boid.vel for boid in neighbors]
        return self.vector_toward_center(vecs)
```

`align` is also similar to `center`; the big difference is that it computes the average of the neighbors' velocities, rather than their positions. If the neighbors point in a particular direction, the Boid tends to steer toward that direction.

Finally, `love` computes a vector that points in the direction of the carrot.

```
    def love(self, carrot):
        toward = carrot.pos - self.pos
        return limit_vector(toward)
```

The results from `center`, `avoid`, `align`, and `love` are what Reynolds calls "acceleration requests", where each request is intended to achieve a different goal.

10.5 Arbitration

To arbitrate among these possibly conflicting goals, we compute a weighted sum of the four requests:

```
    def set_goal(self, boids, carrot):
        w_avoid = 10
        w_center = 3
        w_align = 1
        w_love = 10

        self.goal = (w_center * self.center(boids) +
                     w_avoid * self.avoid(boids, carrot) +
                     w_align * self.align(boids) +
                     w_love * self.love(carrot))
        self.goal.mag = 1
```

`w_center`, `w_avoid`, and the other weights determine the importance of the acceleration requests. Usually `w_avoid` is relatively high and `w_align` is relatively low.

www.dbooks.org

After computing a goal for each Boid, we update their velocity and position:

```
    def move(self, mu=0.1, dt=0.1):
        self.vel = (1-mu) * self.vel + mu * self.goal
        self.vel.mag = 1
        self.pos += dt * self.vel
        self.axis = self.length * self.vel
```

The new velocity is the weighted sum of the old velocity and the goal. The parameter `mu` determines how quickly the birds can change speed and direction. Then we normalize velocity so its magnitude is 1, and orient the axis of the Boid to point in the direction it is moving.

To update position, we multiply velocity by the time step, `dt`, to get the change in position. Finally, we update `axis` so the orientation of the Boid when it is drawn is aligned with its velocity.

Many parameters influence flock behavior, including the radius, angle and weight for each behavior, as well as maneuverability, `mu`. These parameters determine the ability of the Boids to form and maintain a flock, and the patterns of motion and organization within the flock. For some settings, the Boids resemble a flock of birds; other settings resemble a school of fish or a cloud of flying insects.

As one of the exercises at the end of this chapter, you can modify these parameters and see how they affect Boid behavior.

10.6 Emergence and free will

Many complex systems have properties, as a whole, that their components do not:

- The Rule 30 cellular automaton is deterministic, and the rules that govern its evolution are completely known. Nevertheless, it generates a sequence that is statistically indistinguishable from random.
- The agents in Schelling's model are not racist, but the outcome of their interactions is a high degree of segregation.

- Agents in Sugarscape form waves that move diagonally even though the agents cannot.

- Traffic jams move backward even though the cars in them are moving forward.

- Flocks and herds behave as if they are centrally organized even though the animals in them are making individual decisions based on local information.

These examples suggest an approach to several old and challenging questions, including the problems of consciousness and free will.

Free will is the ability to make choices, but if our bodies and brains are governed by deterministic physical laws, our choices are completely determined.

Philosophers and scientists have proposed many possible resolutions to this apparent conflict; for example:

- William James proposed a two-stage model in which possible actions are generated by a random process and then selected by a deterministic process. In that case our actions are fundamentally unpredictable because the process that generates them includes a random element.

- David Hume suggested that our perception of making choices is an illusion; in that case, our actions are deterministic because the system that produces them is deterministic.

These arguments reconcile the conflict in opposite ways, but they agree that there is a conflict: the system cannot have free will if the parts are deterministic.

The complex systems in this book suggest the alternative that free will, at the level of options and decisions, is compatible with determinism at the level of neurons (or some lower level). In the same way that a traffic jam moves backward while the cars move forward, a person can have free will even though neurons don't.

www.dbooks.org

10.7 Exercises

The code for the traffic jam simulation is in the Jupyter notebook `chap09.ipynb` in the repository for this book. Open this notebook, read the code, and run the cells. You can use this notebook to work on the following exercise. My solutions are in `chap09soln.ipynb`.

Exercise 10.1 In the traffic jam simulation, define a class, `BetterDriver`, that inherits from `Driver` and overrides `choose_acceleration`. See if you can define driving rules that do better than the basic implementation in `Driver`. You might try to achieve higher average speed, or a lower number of collisions.

Exercise 10.2 The code for my Boid implementation is in `Boids7.py` in the repository for this book. To run it, you will need VPython, a library for 3-D graphics and animation. If you use Anaconda (as I recommend in Section 0.3), you can run the following in a terminal or Command Window:

```
conda install -c vpython vpython
```

Then run `Boids7.py`. It should either launch a browser or create a window in a running browser, and create an animated display showing Boids, as white cones, circling a red sphere, which is the carrot. If you click and move the mouse, you can move the carrot and see how the Boids react.

Read the code to see how the parameters control Boid behaviors. Experiment with different parameters. What happens if you "turn off" one of the behaviors by setting its weight to 0?

To generate more bird-like behavior, Flake suggests adding a behavior to maintain a clear line of sight; in other words, if there is another bird directly ahead, the Boid should move away laterally. What effect do you expect this rule to have on the behavior of the flock? Implement it and see.

Exercise 10.3 Read more about free will at http://thinkcomplex.com/will. The view that free will is compatible with determinism is called **compatibilism**. One of the strongest challenges to compatibilism is the "consequence argument". What is the consequence argument? What response can you give to the consequence argument based on what you have read in this book?

www.dbooks.org

Chapter 11

Evolution

The most important idea in biology, and possibly all of science, is the **theory of evolution by natural selection**, which claims that *new species are created and existing species change due to natural selection*. Natural selection is a process in which inherited variations between individuals cause differences in survival and reproduction.

Among people who know something about biology, the theory of evolution is widely regarded as a fact, which is to say that it is consistent with all current observations; it is highly unlikely to be contradicted by future observations; and, if it is revised in the future, the changes will almost certainly leave the central ideas substantially intact.

Nevertheless, many people do not believe in evolution. In a survey run by the Pew Research Center, survey respondents were asked which of the following claims is closer to their view:

1. Humans and other living things have evolved over time.

2. Humans and other living things have existed in their present form since the beginning of time.

About 34% of Americans chose the second (see http://thinkcomplex.com/arda).

Even among the ones who believe that living things have evolved, barely more than half believe that the cause of evolution is natural selection. In other words, only a third of Americans believe that the theory of evolution is true.

How is this possible? In my opinion, contributing factors include:

- Some people think that there is a conflict between evolution and their religious beliefs. Feeling like they have to reject one, they reject evolution.
- Others have been actively misinformed, often by members of the first group, so that much of what they know about evolution is misleading or false. For example, many people think that evolution means humans evolved from monkeys. It doesn't, and we didn't.
- And many people simply don't know anything about evolution.

There's probably not much I can do about the first group, but I think I can help the others. Empirically, the theory of evolution is hard for people to understand. At the same time, it is profoundly simple: for many people, once they understand it, it seems both obvious and irrefutable.

To help people make this transition from confusion to clarity, the most powerful tool I have found is computation. Ideas that are hard to understand in theory can be easy to understand when we see them happening in simulation. That is the goal of this chapter.

The code for this chapter is in `chap11.ipynb`, which is a Jupyter notebook in the repository for this book. For more information about working with this code, see Section 0.3.

11.1 Simulating evolution

I start with a simple model that demonstrates a basic form of evolution. According to the theory, the following features are sufficient to produce evolution:

- Replicators: We need a population of agents that can reproduce in some way. We'll start with replicators that make perfect copies of themselves. Later we'll add imperfect copying, that is, mutation.

www.dbooks.org

- Variation: We need variability in the population, that is, differences between individuals.
- Differential survival or reproduction: The differences between individuals have to affect their ability to survive or reproduce.

To simulate these features, we'll define a population of agents that represent individual organisms. Each agent has genetic information, called its **genotype**, which is the information that gets copied when the agent replicates. In our model[1], a genotype is represented by a sequence of `N` binary digits (zeros and ones), where `N` is a parameter we choose.

To generate variation, we create a population with a variety of genotypes; later we will explore mechanisms that create or increase variation.

Finally, to generate differential survival and reproduction, we define a function that maps from each genotype to a **fitness**, where fitness is a quantity related to the ability of an agent to survive or reproduce.

11.2 Fitness landscape

The function that maps from genotype to fitness is called a **fitness landscape**. In the landscape metaphor, each genotype corresponds to a location in an `N`-dimensional space, and fitness corresponds to the "height" of the landscape at that location. For visualizations that might clarify this metaphor, see http://thinkcomplex.com/fit.

In biological terms, the fitness landscape represents information about how the genotype of an organism is related to its physical form and capabilities, called its **phenotype**, and how the phenotype interacts with its **environment**.

In the real world, fitness landscapes are complicated, but we don't need to build a realistic model. To induce evolution, we need *some* relationship between genotype and fitness, but it turns out that it can be *any* relationship. To demonstrate this point, we'll use a totally random fitness landscape.

[1] This model is a variant of the NK model developed primarily by Stuart Kauffman (see http://thinkcomplex.com/nk).

Here is the definition for a class that represents a fitness landscape:

```
class FitnessLandscape:

    def __init__(self, N):
        self.N = N
        self.one_values = np.random.random(N)
        self.zero_values = np.random.random(N)

    def fitness(self, loc):
        fs = np.where(loc, self.one_values,
                           self.zero_values)
        return fs.mean()
```

The genotype of an agent, which corresponds to its location in the fitness landscape, is represented by a NumPy array of zeros and ones called `loc`. The fitness of a given genotype is the mean of `N` **fitness contributions**, one for each element of `loc`.

To compute the fitness of a genotype, `FitnessLandscape` uses two arrays: `one_values`, which contains the fitness contributions of having a `1` in each element of `loc`, and `zero_values`, which contains the fitness contributions of having a `0`.

The `fitness` method uses `np.where` to select a value from `one_values` where `loc` has a `1`, and a value from `zero_values` where `loc` has a `0`.

As an example, suppose `N=3` and

```
one_values =  [0.1, 0.2, 0.3]
zero_values = [0.4, 0.7, 0.9]
```

In that case, the fitness of `loc = [0, 1, 0]` would be the mean of `[0.4, 0.2, 0.9]`, which is `0.5`.

11.3 Agents

Next we need agents. Here's the class definition:

www.dbooks.org

```
class Agent:

    def __init__(self, loc, fit_land):
        self.loc = loc
        self.fit_land = fit_land
        self.fitness = fit_land.fitness(self.loc)

    def copy(self):
        return Agent(self.loc, self.fit_land)
```

The attributes of an `Agent` are:

- `loc`: The location of the `Agent` in the fitness landscape.
- `fit_land`: A reference to a `FitnessLandscape` object.
- `fitness`: The fitness of this `Agent` in the `FitnessLandscape`, represented as a number between 0 and 1.

`Agent` provides `copy`, which copies the genotype exactly. Later, we will see a version that copies with mutation, but mutation is not necessary for evolution.

11.4 Simulation

Now that we have agents and a fitness landscape, I'll define a class called `Simulation` that simulates the creation, reproduction, and death of the agents. To avoid getting bogged down, I'll present a simplified version of the code here; you can see the details in the notebook for this chapter.

Here's the definition of `Simulation`:

```
class Simulation:

    def __init__(self, fit_land, agents):
        self.fit_land = fit_land
        self.agents = agents
```

The attributes of a `Simulation` are:

- `fit_land`: A reference to a `FitnessLandscape` object.
- `agents`: An array of `Agent` objects.

The most important function in `Simulation` is `step`, which simulates one time step:

```
# class Simulation:

    def step(self):
        n = len(self.agents)
        fits = self.get_fitnesses()

        # see who dies
        index_dead = self.choose_dead(fits)
        num_dead = len(index_dead)

        # replace the dead with copies of the living
        replacements = self.choose_replacements(num_dead, fits)
        self.agents[index_dead] = replacements
```

`step` uses three other methods:

- `get_fitnesses` returns an array containing the fitness of each agent.
- `choose_dead` decides which agents die during this time step, and returns an array that contains the indices of the dead agents.
- `choose_replacements` decides which agents reproduce during this time step, invokes `copy` on each one, and returns an array of new `Agent` objects.

In this version of the simulation, the number of new agents during each time step equals the number of dead agents, so the number of live agents is constant.

11.5 No differentiation

Before we run the simulation, we have to specify the behavior of `choose_dead` and `choose_replacements`. We'll start with simple versions of these functions that don't depend on fitness:

www.dbooks.org

```
# class Simulation

    def choose_dead(self, fits):
        n = len(self.agents)
        is_dead = np.random.random(n) < 0.1
        index_dead = np.nonzero(is_dead)[0]
        return index_dead
```

In `choose_dead`, `n` is the number of agents and `is_dead` is a boolean array that contains `True` for the agents who die during this time step. In this version, every agent has the same probability of dying: 0.1. `choose_dead` uses `np.nonzero` to find the indices of the non-zero elements of `is_dead` (`True` is considered non-zero).

```
# class Simulation

    def choose_replacements(self, n, fits):
        agents = np.random.choice(self.agents, size=n, replace=True)
        replacements = [agent.copy() for agent in agents]
        return replacements
```

In `choose_replacements`, `n` is the number of agents who reproduce during this time step. It uses `np.random.choice` to choose `n` agents with replacement. Then it invokes `copy` on each one and returns a list of new `Agent` objects.

These methods don't depend on fitness, so this simulation does not have differential survival or reproduction. As a result, we should not expect to see evolution. But how can we tell?

11.6 Evidence of evolution

The most inclusive definition of evolution is a change in the distribution of genotypes in a population. Evolution is an aggregate effect: in other words, individuals don't evolve; populations do.

In this simulation, genotypes are locations in a high-dimensional space, so it is hard to visualize changes in their distribution. However, if the genotypes change, we expect their fitness to change as well. So we will use *changes in*

the distribution of fitness as evidence of evolution. In particular, we'll look at the mean and standard deviation of fitness over time.

Before we run the simulation, we have to add an `Instrument`, which is an object that gets updated after each time step, computes a statistic of interest, or "metric", and stores the result in a sequence we can plot later.

Here is the parent class for all instruments:

```
class Instrument:
    def __init__(self):
        self.metrics = []
```

And here's the definition for `MeanFitness`, an instrument that computes the mean fitness of the population at each time step:

```
class MeanFitness(Instrument):
    def update(self, sim):
        mean = np.nanmean(sim.get_fitnesses())
        self.metrics.append(mean)
```

Now we're ready to run the simulation. To avoid the effect of random changes in the starting population, we start every simulation with the same set of agents. And to make sure we explore the entire fitness landscape, we start with one agent at every location. Here's the code that creates the `Simulation`:

```
N = 8
fit_land = FitnessLandscape(N)
agents = make_all_agents(fit_land, Agent)
sim = Simulation(fit_land, agents)
```

`make_all_agents` creates one `Agent` for every location; the implementation is in the notebook for this chapter.

Now we can create and add a `MeanFitness` instrument, run the simulation, and plot the results:

```
instrument = MeanFitness()
sim.add_instrument(instrument)
sim.run()
```

www.dbooks.org

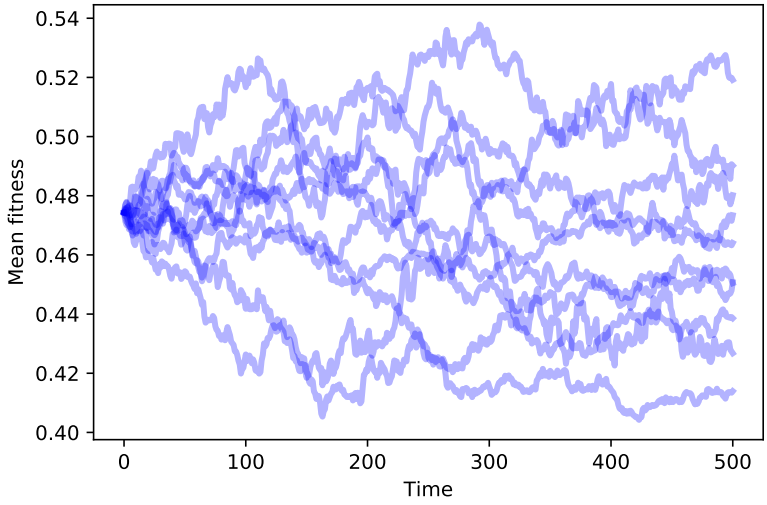


Figure 11.1: Mean fitness over time for 10 simulations with no differential survival or reproduction.

`Simulation` keeps a list of `Instrument` objects. After each time step it invokes `update` on each `Instrument` in the list.

Figure 11.1 shows the result of running this simulation 10 times. The mean fitness of the population drifts up or down at random. Since the distribution of fitness changes over time, we infer that the distribution of phenotypes is also changing. By the most inclusive definition, this **random walk** is a kind of evolution. But it is not a particularly interesting kind.

In particular, this kind of evolution does not explain how biological species change over time, or how new species appear. The theory of evolution is powerful because it explains phenomena we see in the natural world that seem inexplicable:

- Adaptation: Species interact with their environments in ways that seem too complex, too intricate, and too clever to happen by chance. Many features of natural systems seem as if they were designed.

- Increasing diversity: Over time the number of species on earth has generally increased (despite several periods of mass extinction).

- Increasing complexity: The history of life on earth starts with relatively simple life forms, with more complex organisms appearing later in the geological record.

These are the phenomena we want to explain. So far, our model doesn't do the job.

11.7 Differential survival

Let's add one more ingredient, differential survival. Here's a class that extends `Simulation` and overrides `choose_dead`:

```
class SimWithDiffSurvival(Simulation):

    def choose_dead(self, fits):
        n = len(self.agents)
        is_dead = np.random.random(n) > fits
        index_dead = np.nonzero(is_dead)[0]
        return index_dead
```

Now the probability of survival depends on fitness; in fact, in this version, the probability that an agent survives each time step *is* its fitness.

Since agents with low fitness are more likely to die, agents with high fitness are more likely to survive long enough to reproduce. Over time we expect the number of low-fitness agents to decrease, and the number of high-fitness agents to increase.

Figure 11.2 shows mean fitness over time for 10 simulations with differential survival. Mean fitness increases quickly at first, but then levels off.

You can probably figure out why it levels off: if there is only one agent at a particular location and it dies, it leaves that location unoccupied. Without mutation, there is no way for it to be occupied again.

With `N=8`, this simulation starts with 256 agents occupying all possible locations. Over time, the number of occupied locations decreases; if the simulation runs long enough, eventually all agents will occupy the same location.

www.dbooks.org

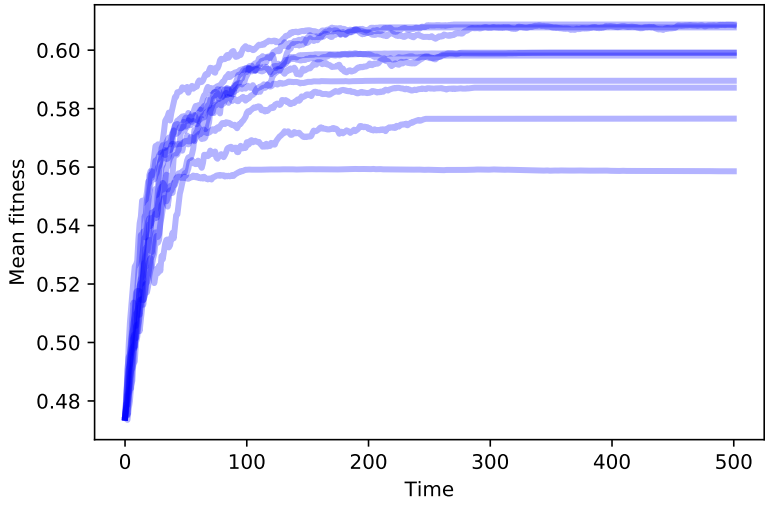


Figure 11.2: Mean fitness over time for 10 simulations with differential survival.

So this simulation starts to explain adaptation: increasing fitness means that the species is getting better at surviving in its environment. But the number of occupied locations decreases over time, so this model does not explain increasing diversity at all.

In the notebook for this chapter, you will see the effect of differential reproduction. As you might expect, differential reproduction also increases mean fitness. But without mutation, we still don't see increasing diversity.

11.8 Mutation

In the simulations so far, we start with the maximum possible diversity — one agent at every location in the landscape — and end with the minimum possible diversity, all agents at one location.

That's almost the opposite of what happened in the natural world, which apparently began with a single species that branched, over time, into the millions, or possibly billions, of species on Earth today (see http://thinkcomplex.com/bio).

With perfect copying in our model, we never see increasing diversity. But if we add mutation, along with differential survival and reproduction, we get a step closer to understanding evolution in nature.

Here is a class definition that extends `Agent` and overrides `copy`:

```
class Mutant(Agent):

    def copy(self, prob_mutate=0.05)::
        if np.random.random() > prob_mutate:
            loc = self.loc.copy()
        else:
            direction = np.random.randint(self.fit_land.N)
            loc = self.mutate(direction)
        return Mutant(loc, self.fit_land)
```

In this model of mutation, every time we call `copy`, there is a 5% chance of mutation. In case of mutation, we choose a random direction from the current location — that is, a random bit in the genotype — and flip it. Here's `mutate`:

```
    def mutate(self, direction):
        new_loc = self.loc.copy()
        new_loc[direction] ^= 1
        return new_loc
```

The operator `^=` computes "exclusive OR"; with the operand 1, it has the effect of flipping a bit (see http://thinkcomplex.com/xor).

Now that we have mutation, we don't have to start with an agent at every location. Instead, we can start with the minimum variability: all agents at the same location.

Figure 11.3 shows the results of 10 simulations with mutation and differential survival and reproduction. In every case, the population evolves toward the location with maximum fitness.

To measure diversity in the population, we can plot the number of occupied locations after each time step. Figure 11.4 shows the results. We start with 100 agents at the same location. As mutations occur, the number of occupied locations increases quickly.

www.dbooks.org

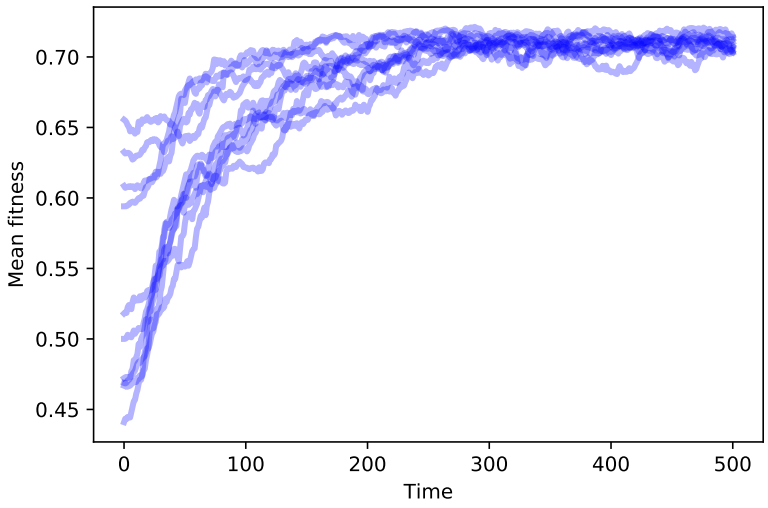


Figure 11.3: Mean fitness over time for 10 simulations with mutation and differential survival and reproduction.

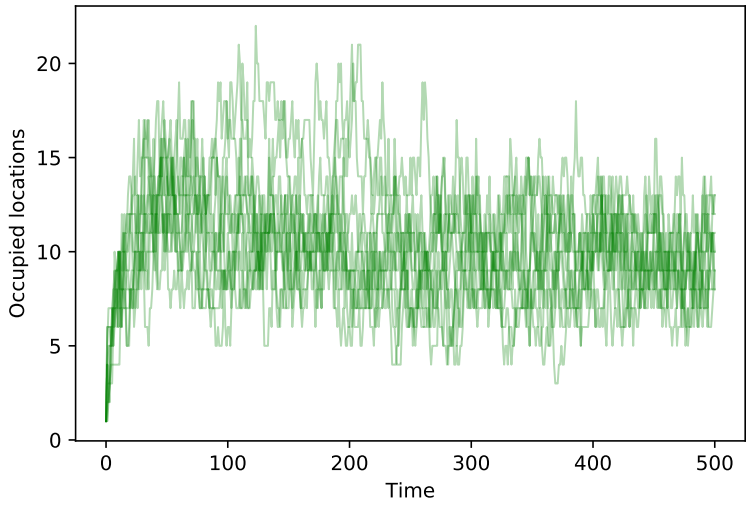


Figure 11.4: Number of occupied locations over time for 10 simulations with mutation and differential survival and reproduction.

When an agent discovers a high-fitness location, it is more likely to survive and reproduce. Agents at lower-fitness locations eventually die out. Over time, the population migrates through the landscape until most agents are at the location with the highest fitness.

At that point, the system reaches an equilibrium where mutation occupies new locations at the same rate that differential survival causes lower-fitness locations to be left empty.

The number of occupied locations in equilibrium depends on the mutation rate and the degree of differential survival. In these simulations the number of unique occupied locations at any point is typically 5–15.

It is important to remember that the agents in this model don't move, just as the genotype of an organism doesn't change. When an agent dies, it can leave a location unoccupied. And when a mutation occurs, it can occupy a new location. As agents disappear from some locations and appear in others, the population migrates across the landscape, like a glider in Game of Life. But organisms don't evolve; populations do.

11.9 Speciation

The theory of evolution says that natural selection changes existing species and creates new ones. In our model, we have seen changes, but we have not seen a new species. It's not even clear, in the model, what a new species would look like.

Among species that reproduce sexually, two organisms are considered the same species if they can breed and produce fertile offspring. But the agents in the model don't reproduce sexually, so this definition doesn't apply.

Among organisms that reproduce asexually, like bacteria, the definition of species is not as clear-cut. Generally, a population is considered a species if their genotypes form a cluster, that is, if the genetic differences within the population are small compared to the differences between populations.

Before we can model new species, we need the ability to identify clusters of agents in the landscape, which means we need a definition of **distance** between

www.dbooks.org

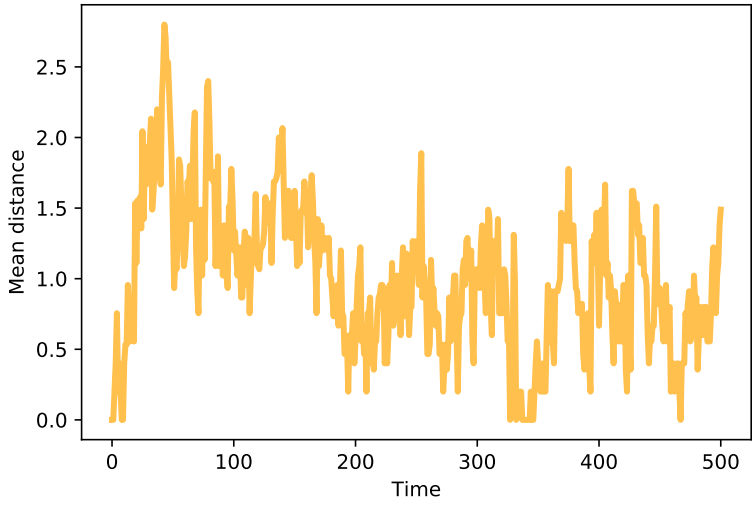


Figure 11.5: Mean distance between agents over time.

locations. Since locations are represented with arrays of bits, we'll define distance as the number of bits that differ between locations. `FitnessLandscape` provides a `distance` method:

```
# class FitnessLandscape

    def distance(self, loc1, loc2):
        return np.sum(np.logical_xor(loc1, loc2))
```

The `logical_xor` function computes "exclusive OR", which is `True` for bits that differ, and `False` for the bits that are the same.

To quantify the dispersion of a population, we can compute the mean of the distances between pairs of agents. In the notebook for this chapter, you'll see the `MeanDistance` instrument, which computes this metric after each time step.

Figure 11.5 shows mean distance between agents over time. Because we start with identical mutants, the initial distances are 0. As mutations occur, mean distance increases, reaching a maximum while the population migrates across the landscape.

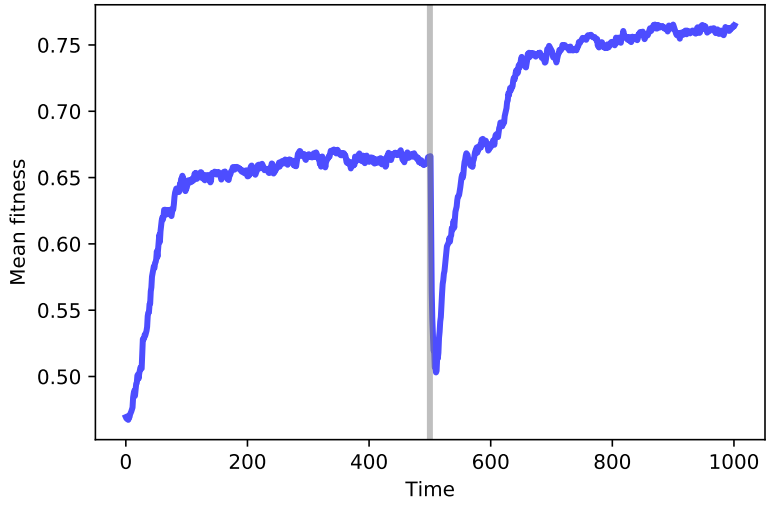


Figure 11.6: Mean fitness over time. After 500 time steps, we change the fitness landscape.

Once the agents discover the optimal location, mean distance decreases until the population reaches an equilibrium where increasing distance due to mutation is balanced by decreasing distance as agents far from the optimal location disappear. In these simulations, the mean distance in equilibrium is near 1; that is, most agents are only one mutation away from optimal.

Now we are ready to look for new species. To model a simple kind of speciation, suppose a population evolves in an unchanging environment until it reaches steady state (like some species we find in nature that seem to have changed very little over long periods of time).

Now suppose we either change the environment or transport the population to a new environment. Some features that increased fitness in the old environment might decrease it in the new environment, and vice versa.

We can model these scenarios by running a simulation until the population reaches steady state, then changing the fitness landscape, and then resuming the simulation until the population reaches steady state again.

Figure 11.6 shows results from a simulation like that. We start with 100 identical mutants at a random location, and run the simulation for 500 time

www.dbooks.org

steps. At that point, many agents are at the optimal location, which has fitness near 0.65 in this example. The genotypes of the agents form a cluster, with the mean distance between agents near 1.

After 500 steps, we run `FitnessLandscape.set_values`, which changes the fitness landscape; then we resume the simulation. Mean fitness is lower, as we expect because the optimal location in the old landscape is no better than a random location in the new landscape.

After the change, mean fitness increases again as the population migrates across the new landscape, eventually finding the new optimum, which has fitness near 0.75 (which happens to be higher in this example, but needn't be).

Once the population reaches steady state, it forms a new cluster, with mean distance between agents near 1 again.

Now if we compute the distance between the agents' locations before and after the change, they differ by more than 6, on average. The distances between clusters are much bigger than the distances between agents in each cluster, so we can interpret these clusters as distinct species.

11.10 Summary

We have seen that mutation, along with differential survival and reproduction, is sufficient to cause increasing fitness, increasing diversity, and a simple form of speciation. This model is not meant to be realistic; evolution in natural systems is much more complicated than this. Rather, it is meant to be a "sufficiency theorem"; that is, a demonstration that the features of the model are sufficient to produce the behavior we are trying to explain (see http://thinkcomplex.com/suff).

Logically, this "theorem" doesn't prove that evolution in nature is caused by these mechanisms alone. But since these mechanisms do appear, in many forms, in biological systems, it is reasonable to think that they at least contribute to natural evolution.

Likewise, the model does not prove that these mechanisms always cause evolution. But the results we see here turn out to be robust: in almost any

model that includes these features — imperfect replicators, variability, and differential reproduction — evolution happens.

I hope this observation helps to demystify evolution. When we look at natural systems, evolution seems complicated. And because we primarily see the results of evolution, with only glimpses of the process, it can be hard to imagine and hard to believe.

But in simulation, we see the whole process, not just the results. And by including the minimal set of features to produce evolution — temporarily ignoring the vast complexity of biological life — we can see evolution as the surprisingly simple, inevitable idea that it is.

11.11 Exercises

The code for this chapter is in the Jupyter notebook `chap11.ipynb` in the repository for this book. Open the notebook, read the code, and run the cells. You can use the notebook to work on the following exercises. My solutions are in `chap11soln.ipynb`.

Exercise 11.1 The notebook shows the effects of differential reproductions and survival separately. What if you have both? Write a class called `SimWithBoth` that uses the version of `choose_dead` from `SimWithDiffSurvival` and the version of `choose_replacements` from `SimWithDiffReproduction`. Does mean fitness increase more quickly?

As a Python challenge, can you write this class without copying code?

Exercise 11.2 When we change the landscape as in Section 11.9, the number of occupied locations and the mean distance usually increase, but the effect is not always big enough to be obvious. Try out some different random seeds to see how general the effect is.

www.dbooks.org

Chapter 12

Evolution of cooperation

In this final chapter, I take on two questions, one from biology and one from philosophy:

- In biology, the "problem of altruism" is the apparent conflict between natural selection, which suggests that animals live in a state of constant competition, and altruism, which is the tendency of many animals to help other animals, even to their own detriment. See http://thinkcomplex.com/altruism.

- In moral philosophy, the question of human nature asks whether humans are fundamentally good, or evil, or blank states shaped by their environment. See http://thinkcomplex.com/nature.

The tools I use to address these questions are agent-based simulation (again) and game theory, which is a set of abstract models meant to describe ways agents interact. Specifically, the game we will consider is the Prisoner's Dilemma.

The code for this chapter is in `chap12.ipynb`, which is a Jupyter notebook in the repository for this book. For more information about working with this code, see Section 0.3.

12.1 Prisoner's Dilemma

The Prisoner's Dilemma is a topic in game theory, but it's not the fun kind of game. Instead, it is the kind of game that sheds light on human motivation and behavior. Here is the presentation of the dilemma from Wikipedia (http://thinkcomplex.com/pd):

> Two members of a criminal gang are arrested and imprisoned. Each prisoner is in solitary confinement with no means of communicating with the other. The prosecutors lack sufficient evidence to convict the pair on the principal charge, but they have enough to convict both on a lesser charge. Simultaneously, the prosecutors offer each prisoner a bargain. Each prisoner is given the opportunity to either: (1) betray the other by testifying that the other committed the crime, or (2) cooperate with the other by remaining silent. The offer is:
>
> - If A and B each betray the other, each of them serves 2 years in prison.
> - If A betrays B but B remains silent, A will be set free and B will serve 3 years in prison (and vice versa).
> - If A and B both remain silent, both of them will only serve 1 year in prison (on the lesser charge).

Obviously, this scenario is contrived, but it is meant to represent a variety of interactions where agents have to choose whether to "cooperate" with each other or "defect", and where the reward (or punishment) for each agent depends on what the other chooses.

With this set of punishments, it is tempting to say that the players should cooperate, that is, that both should remain silent. But neither agent knows what the other will do, so each has to consider two possible outcomes. First, looking at it from A's point of view:

- If B remains silent, A is better off defecting; she would go free rather than serve 1 year.
- If B defects, A is still better off defecting; she would serve only 2 years rather than 3.

www.dbooks.org

No matter what B does, A is better off defecting. And because the game is symmetric, this analysis is the same from B's point of view: no matter what A does, B is better off defecting.

In the simplest version of this game, we assume that A and B have no other considerations to take into account. They can't communicate with each other, so they can't negotiate, make promises, or threaten each other. And they consider only the immediate goal of minimizing their sentences; they don't take into account any other factors.

Under those assumptions, the rational choice for both agents is to defect. That might be a good thing, at least for purposes of criminal justice. But for the prisoners, it is frustrating because there is, apparently, nothing they can do to achieve the outcome they both want. And this model applies to other scenarios in real life where cooperation would be better for the greater good as well as for the players.

Studying these scenarios, and ways to escape from the dilemma, is the focus of people who study game theory, but it is not the focus of this chapter. We are headed in a different direction.

12.2 The problem of nice

Since the Prisoner's Dilemma was first discussed in the 1950s, it has been a popular topic of study in social psychology. Based on the analysis in the previous section, we can say what a perfectly rational agent *should* do; it is harder to predict what real people actually do. Fortunately, the experiment has been done[1].

If we assume that people are smart enough to do the analysis (or understand it when explained), and that they generally act in their own interest, we would expect them to defect pretty much all the time. But they don't. In most

[1]Here's a recent report with references to previous experiments: Barreda-Tarrazona, Jaramillo-Gutiérrez, Pavan, and Sabater-Grande, "Individual Characteristics vs. Experience: An Experimental Study on Cooperation in Prisoner's Dilemma", Frontiers in Psychology, 2017; 8: 596. http://thinkcomplex.com/pdexp.

experiments, subjects cooperate much more than the rational agent model predicts[2].

The most obvious explanation of this result is that people are not rational agents, which should not be a surprise to anyone. But why not? Is it because they are not smart enough to understand the scenario or because they are knowingly acting contrary to their own interest?

Based on experimental results, it seems that at least part of the explanation is plain altruism: many people are willing to incur a cost to themselves in order to benefit another person. Now, before you nominate that conclusion for publication in the *Journal of Obvious Results*, let's keep asking why:

- Why do people help other people, even at a cost to themselves? At least part of the reason is that they want to; it makes them feel good about themselves and the world.

- And why does being nice make people feel good? It might be tempting to say that they were raised right, or more generally trained by society to want to do good things. But there is little doubt that some part of altruism is innate; a proclivity for altruism is the result of normal brain development.

- Well, why is that? The innate parts of brain development, and the personal characteristics that follow, are the result of genetic information. Of course, the relationship between genes and altruism is complicated; there are probably many genes that interact with each other and with environmental factors to cause people to be more or less altruistic in different circumstances. Nevertheless, there are almost certainly genes that tend to make people altruistic.

- Finally, why is that? If, under natural selection, animals are in constant competition with each other to survive and reproduce, it seems obvious that altruism would be counterproductive. In a population where some people help others, even to their own detriment, and others are purely selfish, it seems like the selfish ones would benefit, the altruistic ones would suffer, and the genes for altruism would be driven to extinction.

[2]For an excellent video summarizing what we have discussed so far, see http://thinkcomplex.com/pdvid1.

www.dbooks.org

This apparent contradiction is the "problem of altruism": *why haven't the genes for altruism died out*?

Among biologists, there are many possible explanations, including reciprocal altruism, sexual selection, kin selection, and group selection. Among non-scientists, there are even more explanations. I leave it to you to explore the alternatives; for now I want to focus on just one explanation, arguably the simplest one: maybe altruism is adaptive. In other words, maybe genes for altruism make people more likely to survive and reproduce.

It turns out that the Prisoner's Dilemma, which raises the problem of altruism, might also help resolve it.

12.3 Prisoner's dilemma tournaments

In the late 1970s Robert Axelrod, a political scientist at the University of Michigan, organized a tournament to compare strategies for playing Prisoner's Dilemma (PD).

He invited participants to submit strategies in the form of computer programs, then played the programs against each other and kept score. Specifically, they played the iterated version of PD, in which the agents play multiple rounds against the same opponent, so their decisions can be based on history.

In Axelrod's tournaments, a simple strategy that did surprisingly well was called "tit for tat", or TFT. TFT always cooperates during the first round of an iterated match; after that, it copies whatever the opponent did during the previous round. If the opponent keeps cooperating, TFT keeps cooperating. If the opponent defects at any point, TFT defects in the next round. But if the opponent goes back to cooperating, so does TFT.

For more information about these tournaments, and an explanation of why TFT does so well, see this video: http://thinkcomplex.com/pdvid2.

Looking at the strategies that did well in these tournaments, Alexrod identified the characteristics they tended to share:

- Nice: The strategies that do well cooperate during the first round, and generally cooperate as often as they defect in subsequent rounds.

- Retaliating: Strategies that cooperate all the time did not do as well as strategies that retaliate if the opponent defects.
- Forgiving: But strategies that were too vindictive tended to punish themselves as well as their opponents.
- Non-envious: Some of the most successful strategies seldom outscore their opponents; they are successful because they do *well enough* against a wide variety of opponents.

TFT has all of these properties.

Axelrod's tournaments offer a partial, possible answer to the problem of altruism: maybe the genes for altruism are prevalent because they are adaptive. To the degree that many social interactions can be modeled as variations on the Prisoner's Dilemma, a brain that is wired to be nice, tempered by a balance of retaliation and forgiveness, will tend to do well in a wide variety of circumstances.

But the strategies in Axelrod's tournaments were designed by people; they didn't evolve. We need to consider whether it is credible that genes for niceness, retribution, and forgiveness could appear by mutation, successfully invade a population of other strategies, and resist being invaded by subsequent mutations.

12.4 Simulating evolution of cooperation

Evolution of Cooperation is the title of the first book where Axelrod presented results from Prisoner's Dilemma tournaments and discussed the implications for the problem of altruism. Since then, he and other researchers have explored the evolutionary dynamics of PD tournaments, that is, how the distribution of strategies changes over time in a population of PD contestants. In the rest of this chapter, I run a version of those experiments and present the results.

First, we'll need a way to encode a PD strategy as a genotype. For this experiment, I consider strategies where the agent's choice in each round depends only on the opponent's choice in the previous two rounds. I represent a strategy using a dictionary that maps from the opponent's previous two choices to the agent's next choice.

www.dbooks.org

Here is the class definition for these agents:

```
class Agent:

    keys = [(None, None),
            (None, 'C'),
            (None, 'D'),
            ('C', 'C'),
            ('C', 'D'),
            ('D', 'C'),
            ('D', 'D')]

    def __init__(self, values, fitness=np.nan):
        self.values = values
        self.responses = dict(zip(self.keys, values))
        self.fitness = fitness
```

`keys` is the sequence of keys in each agent's dictionary, where the tuple `('C', 'C')` means that the opponent cooperated in the previous two rounds; `(None, 'C')` means that only one round has been played and the opponent cooperated; and `(None, None)` means that no rounds have been played.

In the `__init__` method, `values` is a sequence of choices, either `'C'` or `'D'`, that correspond to `keys`. So if the first element of `values` is `'C'`, that means that this agent will cooperate in the first round. If the last element of `values` is `'D'`, this agent will defect if the opponent defected in the previous two rounds.

In this implementation, the genotype of an agent who always defects is `'DDDDDDD'`; the genotype of an agent who always cooperates is `'CCCCCCC'`, and the genotype for TFT is `'CCDCDCD'`.

The `Agent` class provides `copy`, which makes another agent with the same genotype, but with some probability of mutation:

```
    def copy(self, prob_mutate=0.05):
        if np.random.random() > prob_mutate:
            values = self.values
        else:
            values = self.mutate()
        return Agent(values, self.fitness)
```

Mutation works by choosing a random value in the genotype and flipping from 'C' to 'D', or vice versa:

```
    def mutate(self):
        values = list(self.values)
        index = np.random.choice(len(values))
        values[index] = 'C' if values[index] == 'D' else 'D'
        return values
```

Now that we have agents, we need a tournament.

12.5 The Tournament

The `Tournament` class encapsulates the details of the PD competition:

```
    payoffs = {('C', 'C'): (3, 3),
               ('C', 'D'): (0, 5),
               ('D', 'C'): (5, 0),
               ('D', 'D'): (1, 1)}

    num_rounds = 6

    def play(self, agent1, agent2):
        agent1.reset()
        agent2.reset()

        for i in range(self.num_rounds):
            resp1 = agent1.respond(agent2)
            resp2 = agent2.respond(agent1)

            pay1, pay2 = self.payoffs[resp1, resp2]

            agent1.append(resp1, pay1)
            agent2.append(resp2, pay2)

        return agent1.score, agent2.score
```

`payoffs` is a dictionary that maps from the agents' choices to their rewards. For example, if both agents cooperate, they each get 3 points. If one defects

www.dbooks.org

and the other cooperates, the defector gets 5 and the cooperator gets 0. If they both defect, each gets 1. These are the payoffs Axelrod used in his tournaments.

The `play` method runs several rounds of the PD game. It uses the following methods from the `Agent` class:

- `reset`: Initializes the agents before the first round, resetting their scores and the history of their responses.

- `respond`: Asks each agent for their response, given the opponent's previous responses.

- `append`: Updates each agent by storing the choices and adding up the scores from successive rounds.

After the given number of rounds, `play` returns the total score for each agent. I chose `num_rounds=6` so that each element of the genotype is accessed with roughly the same frequency. The first element is only accessed during the first round, or one sixth of the time. The next two elements are only accessed during the second round, or one twelfth each. The last four elements are accessed four of six times, or one sixth each, on average.

`Tournament` provides a second method, `melee`, that determines which agents compete against each other:

```
    def melee(self, agents, randomize=True):
        if randomize:
            agents = np.random.permutation(agents)

        n = len(agents)
        i_row = np.arange(n)
        j_row = (i_row + 1) % n

        totals = np.zeros(n)

        for i, j in zip(i_row, j_row):
            agent1, agent2 = agents[i], agents[j]
            score1, score2 = self.play(agent1, agent2)
            totals[i] += score1
            totals[j] += score2

        for i in i_row:
            agents[i].fitness = totals[i] / self.num_rounds / 2
```

`melee` takes a list of agents and a boolean, `randomize`, that determines whether each agent fights the same neighbors every time, or whether the pairings are randomized.

`i_row` and `j_row` contain the indices of the pairings. `totals` contains the total score of each agent.

Inside the loop, we select two agents, invoke `play`, and update `totals`. At the end, we compute the average number of points each agent got, per round and per opponent, and store the results in the `fitness` attribute of each agent.

12.6 The Simulation

The `Simulation` class for this chapter is based on the one in Section 11.4; the only differences are in `__init__` and `step`.

Here's the `__init__` method:

www.dbooks.org

```
class PDSimulation(Simulation):

    def __init__(self, tournament, agents):
        self.tournament = tournament
        self.agents = np.asarray(agents)
        self.instruments = []
```

A `Simulation` object contains a `Tournament` object, a sequence of agents, and a sequence of `Instrument` objects (as in Section 11.6).

Here's the `step` method:

```
    def step(self):
        self.tournament.melee(self.agents)
        Simulation.step(self)
```

This version of `step` uses `Tournament.melee`, which sets the `fitness` attribute for each agent; then it calls the `step` method from the `Simulation` class, reproduced here:

```
# class Simulation

    def step(self):
        n = len(self.agents)
        fits = self.get_fitnesses()

        # see who dies
        index_dead = self.choose_dead(fits)
        num_dead = len(index_dead)

        # replace the dead with copies of the living
        replacements = self.choose_replacements(num_dead, fits)
        self.agents[index_dead] = replacements

        # update any instruments
        self.update_instruments()
```

`Simulation.step` collects the agents' fitnesses in an array; then it calls `choose_dead` to decide which agents die, and `choose_replacements` to decide which agents reproduce.

My simulation includes differential survival, as in Section 11.7, but not differential reproduction. You can see the details in the notebook for this chapter. As one of the exercises, you will have a chance to explore the effect of differential reproduction.

12.7 Results

Suppose we start with a population of three agents: one always cooperates, one always defects, and one plays the TFT strategy. If we run `Tournament.melee` with this population, the cooperator gets 1.5 points per round, the TFT agent gets 1.9, and the defector gets 3.33. This result suggests that "always defect" should quickly become the dominant strategy.

But "always defect" contains the seeds of its own destruction. If nicer strategies are driven to extinction, the defectors have no one to take advantage of. Their fitness drops, and they become vulnerable to invasion by cooperators.

Based on this analysis, it is not easy to predict how the system will behave: will it find a stable equilibrium, or oscillate between various points in the genotype landscape? Let's run the simulation and find out!

I start with 100 identical agents who always defect, and run the simulation for 5000 steps:

```
tour = Tournament()
agents = make_identical_agents(100, list('DDDDDDD'))
sim = PDSimulation(tour, agents)
```

Figure 12.1 shows mean fitness over time (using the `MeanFitness` instrument from Section 11.6). Initially mean fitness is 1, because when defectors face each other, they get only 1 point each per round.

After about 500 time steps, mean fitness increases to nearly 3, which is what cooperators get when they face each other. However, as we suspected, this situation in unstable. Over the next 500 steps, mean fitness drops below 2, climbs back toward 3, and continues to oscillate.

The rest of the simulation is highly variable, but with the exception of one big drop, mean fitness is usually between 2 and 3, with the long-term mean close to 2.5.

www.dbooks.org

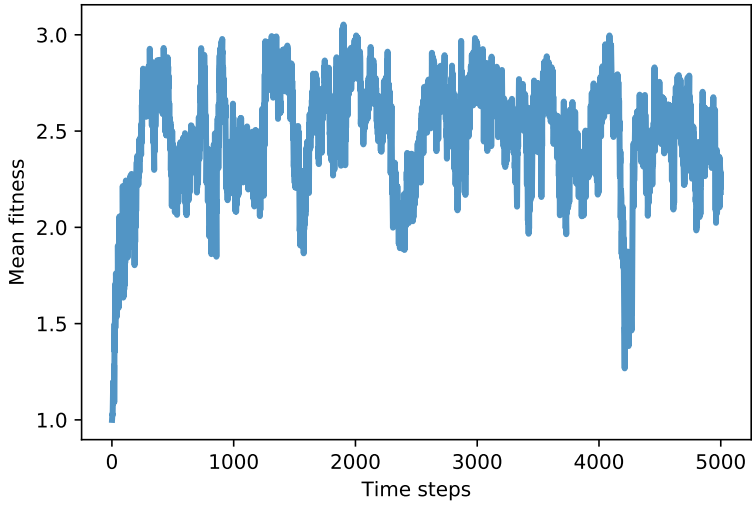


Figure 12.1: Average fitness (points scored per round of Prisoner's Dilemma).

And that's not bad! It's not quite a utopia of cooperation, which would average 3 points per round, but it's a long way from the dystopia of perpetual defection. And it's a lot better than what we might expect from the natural selection of self-interested agents.

To get some insight into this level of fitness, let's look at a few more instruments. `Niceness` measures the fraction of cooperation in the genotypes of the agents after each time step:

```
class Niceness(Instrument):

    def update(self, sim):
        responses = np.array([agent.values
                              for agent in sim.agents])
        metric = np.mean(responses == 'C')
        self.metrics.append(metric)
```

`responses` is an array with one row for each agent and one column for each element of the genome. `metric` is the fraction of elements that are `'C'`, averaged across agents.

Figure 12.2 (left) shows the results: starting from 0, average niceness increases

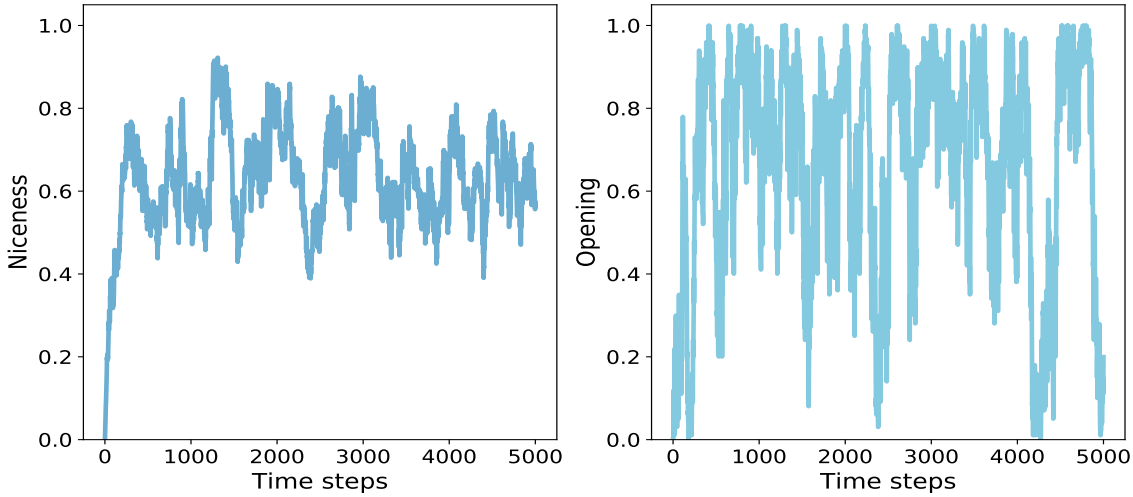


Figure 12.2: Average niceness across all genomes in the population (left), and fraction of population that cooperates in the first round (right).

quickly to 0.75, then oscillates between 0.4 and 0.85, with a long-term mean near 0.65. Again, that's a lot of niceness!

Looking specifically at the opening move, we can track the fraction of agents that cooperate in the first round. Here's the instrument:

```
class Opening(Instrument):

    def update(self, sim):
        responses = np.array([agent.values[0]
                              for agent in sim.agents])
        metric = np.mean(responses == 'C')
        self.metrics.append(metric)
```

Figure 12.2 (right) shows the results, which are highly variable. The fraction of agents who cooperate in the first round is often near 1, and occasionally near 0. The long-term average is close to 0.65, similar to overall niceness. These results are consistent with Axelrod's tournaments; in general, nice strategies do well.

The other characteristics Axelrod identifies in successful strategies are retali-

www.dbooks.org

ation and forgiveness. To measure retaliation, I define this instrument:

```
class Retaliating(Instrument):

    def update(self, sim):
        after_d = np.array([agent.values[2::2]
                            for agent in sim.agents])
        after_c = np.array([agent.values[1::2]
                            for agent in sim.agents])
        metric = np.mean(after_d=='D') - np.mean(after_c=='D')
        self.metrics.append(metric)
```

`Retaliating` compares the number of elements in all genomes where an agent defects after the opponent defects (elements 2, 4, and 6) with the number of places where an agents defects after the opponent cooperates. As you might expect by now, the results vary substantially (you can see the graph in the notebook). On average the difference between these fractions is less than 0.1, so if agents defect 30% of the time after the opponent cooperates, they might defect 40% of the time after a defection.

This result provides weak support for the claim that successful strategies retaliate. But maybe it's not necessary for all agents, or even many, to be retaliatory; if there is at least some tendency toward retaliation in the population as a whole, that might be enough to prevent high-defection strategies from gaining ground.

To measure forgiveness, I define one more instrument to see whether agents might be more likely to cooperate after D-C in the previous two rounds, compared to C-D. In my simulations, there is no evidence for this particular kind of forgiveness. On the other hand, the strategies in these simulations are necessarily forgiving because they consider only the previous two rounds of history. In this context, forgetting is a kind of forgiving.

12.8 Conclusions

Axelrod's tournaments suggest a possible resolution to the problem of altruism: maybe being nice, but not *too* nice, is adaptive. But the strategies in the

original tournaments were designed by people, not evolution, and the distribution of strategies did not change over the course of the tournaments.

So that raises a question: strategies like TFT might do well in a fixed population of human-designed strategies, but can they evolve? In other words, can they appear in a population through mutation, compete successfully with their ancestors, and resist invasion by their descendants?

The simulations in this chapter suggest:

- Populations of defectors are vulnerable to invasion by nicer strategies.
- Populations that are too nice are vulnerable to invasion by defectors.
- As a result, the average level of niceness oscillates, but the average amount of niceness is generally high, and the average level of fitness is generally closer to a utopia of cooperation than to a dystopia of defection.
- TFT, which was a successful strategy in Alexrod's tournaments, does not seem to be a specially optimal strategy in an evolving population. In fact, there is probably no stable optimal strategy.
- Some degree of retaliation may be adaptive, but it might not be necessary for all agents to retaliate. If there is enough retaliation in the population as a whole, that might be enough to prevent invasion by defectors[3].

Obviously, the agents in these simulations are simple, and the Prisoner's Dilemma is a highly abstract model of a limited range of social interactions. Nevertheless, the results in this chapter provide some insight into human nature. Maybe our inclinations toward cooperation, retaliation, and forgiveness are innate, at least in part. These characteristics are a result of how our brains are wired, which is controlled by our genes, at least in part. And maybe our genes build our brains that way because over the history of human evolution, genes for less altruistic brains were less likely to propagate.

Maybe that's why selfish genes build altruistic brains.

[3]And that introduces a whole new topic in game theory, the free-rider problem (see http://thinkcomplex.com/rider)

www.dbooks.org

12.9 Exercises

The code for this chapter is in the Jupyter notebook `chap12.ipynb` in the repository for this book. Open the notebook, read the code, and run the cells. You can use this notebook to work on the following exercises. My solutions are in `chap12soln.ipynb`.

Exercise 12.1 The simulations in this chapter depend on conditions and parameters I chose arbitrarily. As an exercise, I encourage you to explore other conditions to see what effect they have on the results. Here are some suggestions:

1. Vary the initial conditions: instead of starting with all defectors, see what happens if you start with all cooperators, all TFT, or random agents.

2. In `Tournament.melee`, I shuffle the agents at the beginning of each time step, so each agent plays against two randomly-chosen agents. What happens if you don't shuffle? In that case, each agent plays against the same neighbors repeatedly. That might make it easier for a minority strategy to invade a majority, by taking advantage of locality.

3. Since each agent only plays against two other agents, the outcome of each round is highly variable: an agent that would do well against most other agents might get unlucky during any given round, or the other way around. What happens if you increase the number of opponents each agent plays against during each round? Or what if an agent's fitness at the end of each step is the average of its current score and its fitness at the end of the previous round?

4. The function I chose for `prob_survival` varies from 0.7 to 0.9, so the least fit agent, with `p=0.7`, lives for 3.33 time steps on average, and the most fit agent lives for 10 time steps. What happens if you make the degree of differential survival more or less "aggressive"?

5. I chose `num_rounds=6` so that each element of the genome has roughly the same impact on the outcome of a match. But that is substantially shorter than what Alexrod used in his tournaments. What happens if you increase `num_rounds`? Note: if you explore the effect of this parameter, you might want to modify `Niceness` to measure the niceness of the last

4 elements of the genome, which will be under more selective pressure as `num_rounds` increases.

6. My implementation has differential survival but not differential reproduction. What happens if you add differential reproduction?

Exercise 12.2 In my simulations, the population never converges to a state where a majority share the same, presumably optimal, genotype. There are two possible explanations for this outcome: one is that there is no optimal strategy, because whenever the population is dominated by a majority genotype, that condition creates an opportunity for a minority to invade; the other possibility is that the mutation rate is high enough to maintain a diversity of genotypes.

To distinguish between these explanations, try lowering the mutation rate to see what happens. Alternatively, start with a random population and run without mutation until only one genotype survives. Or run with mutation until the system reaches something like a steady state; then turn off mutation and run until there is only one surviving genotype. What are the characteristics of the genotypes that prevail in these conditions?

Exercise 12.3 The agents in my experiment are "reactive" in the sense that their choice during each round depends only on what the opponent did during previous rounds. Explore strategies that also take into account the agent's past choices. These strategies can distinguish an opponent who retaliates from an opponent who defects without provocation.

www.dbooks.org

Appendix A

Reading list

The following are selected books related to topics in this book. Most are written for a non-technical audience.

- Axelrod, Robert, *Complexity of Cooperation*, Princeton University Press, 1997.
- Axelrod, Robert *The Evolution of Cooperation*, Basic Books, 2006.
- Bak, Per *How Nature Works*, Copernicus (Springer), 1996.
- Barabási, Albert-László, *Linked*, Perseus Books Group, 2002.
- Buchanan, Mark, *Nexus*, W. W. Norton & Company, 2002.
- Dawkins, Richard, *The Selfish Gene*, Oxford University Press, 2016.
- Epstein, Joshua and Axtell, Robert, *Growing Artificial Societies*, Brookings Institution Press & MIT Press, 1996.
- Fisher, Len, *The Perfect Swarm*, Basic Books, 2009.
- Flake, Gary William, *The Computational Beauty of Nature*, MIT Press 2000.
- Goldstein, Rebecca, *Incompleteness*, W. W. Norton & Company, 2005.
- Goodwin, Brian *How the Leopard Changed Its Spots*, Princeton University Press, 2001.

- Holland, John, *Hidden Order*, Basic Books, 1995.
- Johnson, Steven, *Emergence*, Scribner, 2001.
- Kelly, Kevin, *Out of Control*, Basic Books, 2002.
- Kosko, Bart, *Fuzzy Thinking*, Hyperion, 1993.
- Levy, Steven *Artificial Life*, Pantheon, 1992.
- Mandelbrot, Benoit, *Fractal Geometry of Nature*, Times Books, 1982.
- McGrayne, Sharon Bertsch, *The Theory That Would Not Die*, Yale University Press, 2011.
- Mitchell, Melanie, *Complexity: A Guided Tour*. Oxford University Press, 2009.
- Waldrop, M. Mitchell *Complexity: The Emerging Science at the Edge of Order and Chaos*, Simon & Schuster, 1992.
- Resnick, Mitchell, *Turtles, Termites, and Traffic Jams*, Bradford, 1997.
- Rucker, Rudy, *The Lifebox, the Seashell, and the Soul*, Thunder's Mouth Press, 2005.
- Sawyer, R. Keith, *Social Emergence: Societies as Complex Systems*, Cambridge University Press, 2005.
- Schelling, Thomas, *Micromotives and Macrobehaviors*, W. W. Norton & Company, 2006.
- Strogatz, Steven, *Sync*, Hachette Books, 2003.
- Watts, Duncan, *Six Degrees*, W. W. Norton & Company, 2003.
- Wolfram, Stephen, *A New Kind Of Science*, Wolfram Media, 2002.

www.dbooks.org