Chapter 16

Dynamical Networks I: Modeling

16.1 Dynamical Network Models

There are several different classes of dynamical network models. In this chapter, we will discuss the following three, in this particular order:

Models for "dynamics *on* networks" These models are the most natural extension of traditional dynamical systems models. They consider how the states of components, or nodes, change over time through their interactions with other nodes that are connected to them. The connections are represented by links of a network, where the network topology is fixed throughout time. Cellular automata, Boolean networks, and artificial neural networks (without learning) all belong to this class.

Models for "dynamics *of* networks" These are the models that consider dynamical changes of network topology itself over time, for various purposes: to understand mechanisms that bring particular network topologies, to evaluate robustness and vulnerability of networks, to design procedures for improving certain properties of networks, etc. The dynamics *of* networks are a particularly hot topic in network science nowadays (as of 2015) because of the increasing availability of *temporal network* data [65].

Models for "adaptive networks" I must admit that I am not 100% objective when it comes to this class of models, because I am one of the researchers who have been actively promoting it [66]. Anyway, the adaptive network models are models that describe the *co-evolution* of dynamics *on* and *of* networks, where node states and network topologies dynamically change adaptively to each other. Adaptive network models try to unify different dynamical network models to provide a generalized modeling framework for complex systems, since many real-world systems show such adaptive network behaviors [67].

16.2 Simulating Dynamics *on* Networks

Because NetworkX adopts plain dictionaries as their main data structure, we can easily add states to nodes (and edges) and dynamically update those states iteratively. This is a simulation of dynamics *on* networks. This class of dynamical network models describes dynamic state changes taking place on a static network topology. Many real-world dynamical networks fall into this category, including:

- Regulatory relationships among genes and proteins within a cell, where nodes are genes and/or proteins and the node states are their expression levels.
- Ecological interactions among species in an ecosystem, where nodes are species and the node states are their populations.
- Disease infection on social networks, where nodes are individuals and the node states are their epidemiological states (e.g., susceptible, infected, recovered, immunized, etc.).
- Information/culture propagation on organizational/social networks, where nodes are individuals or communities and the node states are their informational/cultural states.

The implementation of simulation models for dynamics *on* networks is strikingly similar to that of CA. You may find it even easier on networks, because of the straightforward definition of "neighbors" on networks. Here, we will work on a simple local *majority rule* on a social network, with the following assumptions:

- Nodes represent individuals, and edges represent their symmetric connections for information sharing.
- Each individual takes either 0 or 1 as his or her state.
- Each individual changes his or her state to a majority choice within his or her local neighborhood (i.e., the individual him- or herself and the neighbors connected to him or her). This neighborhood is also called the *ego network* of the focal individual in social sciences.

- State updating takes place simultaneously on all individuals in the network.
- Individuals' states are initially random.

Let's continue to use pycxsimulator.py and implement the simulator code by defining three essential components—initialization, observation, and updating.

The initialization part is to create a model of social network and then assign random states to all the nodes. Here I propose to perform the simulation on our favorite Karate Club graph. Just like the CA simulation, we need to prepare two network objects, one for the current time step and the other for the next time step, to avoid conflicts during the state updating process. Here is an example of the initialization part:

Code 16.1:

```
def initialize():
    global g, nextg
    g = nx.karate_club_graph()
    g.pos = nx.spring_layout(g)
    for i in g.nodes_iter():
        g.node[i]['state'] = 1 if random() < .5 else 0
    nextg = g.copy()
```

Here, we pre-calculate node positions using `spring_layout` and store the results under `g` as an attribute `g.pos`. Python is so flexible that you can dynamically add any attribute to an object without declaring it beforehand; we will utilize this trick later in the agent-based modeling as well. `g.pos` will be used in the visualization so that nodes won't jump around every time you draw the network.

The `g.nodes_iter()` command used in the `for` loop above works essentially the same way as `g.nodes()` in this context, but it is much more memory efficient because it doesn't generate an actual fully spelled-out list, but instead it generates a much more compact representation called an *iterator*, an object that can return next values iteratively. There is also `g.edges_iter()` for similar purposes for edges. It is generally a good idea to use `nodes_iter()` and `edges_iter()` in loops, especially if your network is large. Finally, in the last line, the `copy` command is used to create a duplicate copy of the network.

The observation part is about drawing the network. We have already done this many times, but there is a new challenge here: We need to visualize the node states in addition to the network topology. We can use the `node_color` option for this purpose:

Code 16.2:

```
def observe():
    global g, nextg
    cla()
    nx.draw(g, cmap = cm.binary, vmin = 0, vmax = 1,
            node_color = [g.node[i]['state'] for i in g.nodes_iter()],
            pos = g.pos)
```

The `vmin`/`vmax` options are to use a fixed range of state values; otherwise matplotlib would automatically adjust the color mappings, which sometimes causes misleading visualization. Also note that we are using `g.pos` to keep the nodes in pre-calculated positions.

The updating part is pretty similar to what we did for CA. You just need to sweep all the nodes, and for each node, you sweep its neighbors, counting how many 1's you have in the local neighborhood. Here is a sample code:

Code 16.3:

```
def update():
    global g, nextg
    for i in g.nodes_iter():
        count = g.node[i]['state']
        for j in g.neighbors(i):
            count += g.node[j]['state']
        ratio = count / (g.degree(i) + 1.0)
        nextg.node[i]['state'] = 1 if ratio > .5 \
                                 else 0 if ratio < .5 \
                                 else 1 if random() < .5 else 0
    g, nextg = nextg, g
```

I believe this part deserves some in-depth explanation. The first `for` loop for `i` is to sweep the space, i.e., the whole network. For each node, `i`, the variable `count` is first initialized with node `i`'s own state. This is because, unlike in the previous CA implementations, the node `i` itself is not included in its neighbors, so we have to manually count it first. The second `for` loop for `j` is to sweep node `i`'s neighbors. NetworkX's `g.neighbors(i)` function gives a list of `i`'s neighbors, which is a lot simpler than the neighborhood sweep in CA; we don't have to write nested `for` loops for `dx` and `dy`, and we don't have to worry about boundary conditions at all. Neighbors are `neighbors`, period.

Once the local neighborhood sweep is done, the state ratio is calculated by dividing `count` by the number of nodes counted (= the degree of node `i` plus 1 for itself). If the state ratio is above 0.5, the local majority is 1, so the next state of node `i` will be 1. If it is below 0.5, the local majority is 0, so the next state will be 0. Moreover, since this dynamic is taking place on a network, there is a chance for a tie (which never occurs on CA with von Neumann or Moore neighborhoods). In the example above, I just included a tie-breaker coin toss, to be fair. And the last swap of `g` and `nextg` is something we are already familiar with.

The entire simulation code looks like this:

Code 16.4: net-majority.py

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

def initialize():
    global g, nextg
    g = nx.karate_club_graph()
    g.pos = nx.spring_layout(g)
    for i in g.nodes_iter():
        g.node[i]['state'] = 1 if random() < .5 else 0
    nextg = g.copy()

def observe():
    global g, nextg
    cla()
    nx.draw(g, cmap = cm.binary, vmin = 0, vmax = 1,
            node_color = [g.node[i]['state'] for i in g.nodes_iter()],
            pos = g.pos)

def update():
    global g, nextg
    for i in g.nodes_iter():
        count = g.node[i]['state']
        for j in g.neighbors(i):
            count += g.node[j]['state']
        ratio = count / (g.degree(i) + 1.0)
        nextg.node[i]['state'] = 1 if ratio > .5 \
                                 else 0 if ratio < .5 \
                                 else 1 if random() < .5 else 0
    g, nextg = nextg, g

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

Run this code and enjoy your first dynamical network simulation. You will notice that the network sometimes converges to a homogeneous state, while at other times it remains in a divided condition (Fig. 16.1). Can you identify the areas where the boundaries between the different states tend to form?

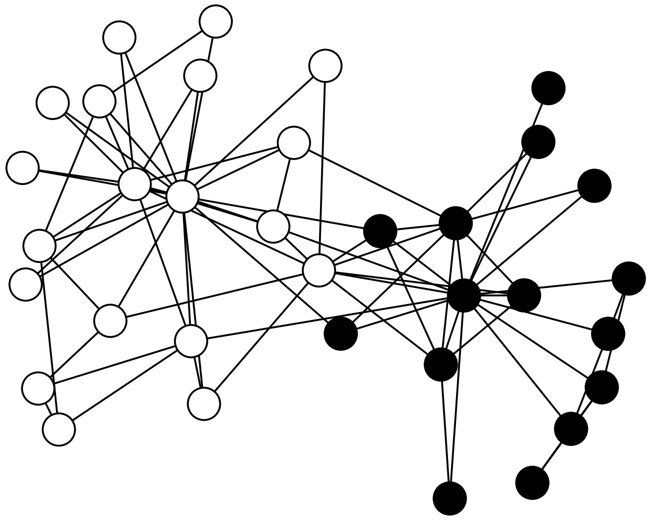


Figure 16.1: Visual output of Code 16.4. In this example, two groups of different states have formed.

Exercise 16.1 Revise the majority rule dynamical network model developed above so that each node stochastically flips its state with some probability. Then simulate the model and see how its behavior is affected.

In what follows, we will see some more examples of two categories of dynamics *on* networks models: discrete state/time models and continuous state/time models. We will discuss two exemplar models in each category.

Discrete state/time models (1): Voter model The first example is a revision of the majority rule dynamical network model developed above. A very similar model of abstract opinion dynamics has been studied in statistical physics, which is called the *voter model.* Just like in the previous model, each node ("voter") takes one of the finite discrete states (say, black and white, or red and blue—you could view it as a political opinion), but the updating rule is different. Instead of having all the nodes update their states simultaneously based on local majority choices, the voter model considers only one opinion transfer event at a time between a pair of connected nodes that are randomly chosen from the network (Fig. 16.2). In this sense, it is an *asynchronous* dynamical network model.

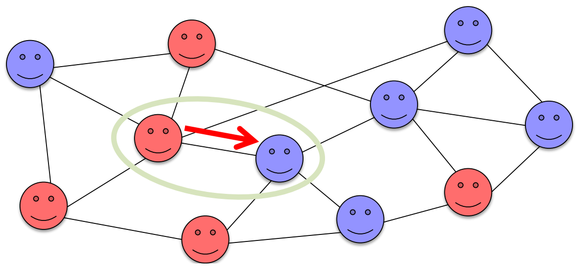


Figure 16.2: Schematic illustration of the voter model. Each time a pair of connected nodes are randomly selected (light green circle), the state of the speaker node (left) is copied to the listener node (right).

There are three minor variations of how those nodes are chosen:

Original ("pull") version: First, a "listener" node is randomly chosen from the network, and then a "speaker" node is randomly chosen from the listener's neighbors.

Reversed ("push") version: First, a "speaker" node is randomly chosen from the network, and then a "listener" node is randomly chosen from the speaker's neighbors.

Edge-based (symmetric) version: First, an edge is randomly chosen from the network, and then the two endpoints (nodes) of the edge are randomly assigned to be a "speaker" and a "listener."

In either case, the "listener" node copies the state of the "speaker" node. This is repeated a number of times until the entire network reaches a consensus with a homogenized state.

If you think through these model assumptions carefully, you may notice that the first two assumptions are not symmetric regarding the probability for a node to be chosen as a “speaker” or a “listener.” The probability actually depends on how popular a node is. Specifically, the original version tends to choose higher-degree nodes as a speaker more often, while the reversed version tends to choose higher-degree nodes as a listener more often. This is because of the famous *friendship paradox*, a counter-intuitive fact first reported by sociologist Scott Feld in the 1990s [68], which is that a randomly selected neighbor of a randomly selected node tends to have a larger-than-average degree. Therefore, it is expected that the original version of the voter model would promote homogenization of opinions as it gives more speaking time to the popular nodes, while the reversed version would give an equal chance of speech to everyone so the opinion homogenization would be much slower, and the edge-based version would be somewhere in between. We can check these predictions by computer simulations.

The good news is that the simulation code of the voter model is much simpler than that of the majority rule network model, because of the asynchrony of state updating. We no longer need to use two separate data structures; we can keep modifying just one `Graph` object directly. Here is a sample code for the original “pull” version of the voter model, again on the Karate Club graph:

Code 16.5: voter-model.py

```
import matplotlib
matplotlib.use('TkAgg')
from pylab import *
import networkx as nx
import random as rd

def initialize():
    global g
    g = nx.karate_club_graph()
    g.pos = nx.spring_layout(g)
    for i in g.nodes_iter():
        g.node[i]['state'] = 1 if random() < .5 else 0

def observe():
    global g
    cla()
    nx.draw(g, cmap = cm.binary, vmin = 0, vmax = 1,
            node_color = [g.node[i]['state'] for i in g.nodes_iter()],
            pos = g.pos)
```

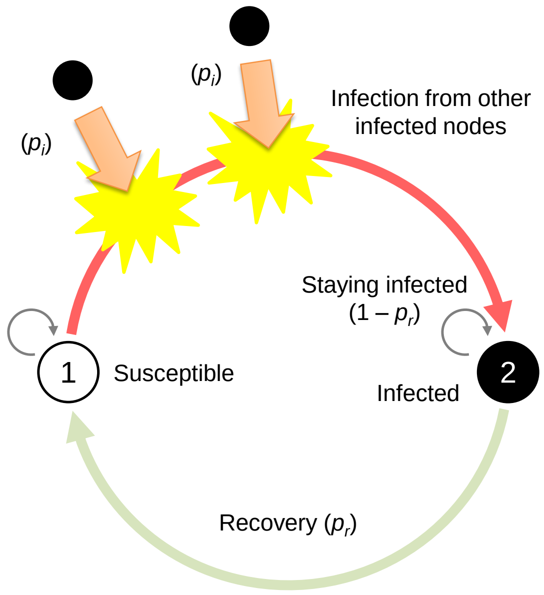
```
def update():
    global g
    listener = rd.choice(g.nodes())
    speaker = rd.choice(g.neighbors(listener))
    g.node[listener]['state'] = g.node[speaker]['state']

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

See how simple the `update` function is! This simulation takes a lot more steps for the system to reach a homogeneous state, because each step involves only two nodes. It is probably a good idea to set the step size to 50 under the "Settings" tab to speed up the simulations.

Exercise 16.2 Revise the code above so that you can measure how many steps it will take until the system reaches a consensus (i.e., homogenized state). Then run multiple simulations (Monte Carlo simulations) to calculate the average time length needed for consensus formation in the original voter model.

Exercise 16.3 Revise the code further to implement (1) the reversed and (2) the edge-based voter models. Then conduct Monte Carlo simulations to measure the average time length needed for consensus formation in each case. Compare the results among the three versions.

Discrete state/time models (2): Epidemic model The second example of discrete state/time dynamical network models is the epidemic model on a network. Here we consider the *Susceptible-Infected-Susceptible (SIS) model*, which is even simpler than the SIR model we discussed in Exercise 7.3. In the SIS model, there are only two states: Susceptible and Infected. A susceptible node can get infected from an infected neighbor node with infection probability p_i (per infected neighbor), while an infected node can recover back to a susceptible node (i.e., no immunity acquired) with recovery probability p_r (Fig. 16.3). This setting still puts everything in binary states like in the earlier examples, so we can recycle their codes developed above.



(p_i)
(p_i)
Infection from other infected nodes
Staying infected ($1 - p_r$)
1
Susceptible
Infected
2
Recovery (p_r)

Figure 16.3: Schematic illustration of the state-transition rule of the SIS model.

Regarding the updating scheme, I think you probably liked the simplicity of the asynchronous updating we used for the voter model, so let's adopt it for this SIS model too. We will first choose a node randomly, and if it is susceptible, then we will randomly choose one of its neighbors; this is similar to the original "pull" version of the voter model. All you need is to revise just the `updating` function as follows:

Code 16.6: SIS-model.py

```
p_i = 0.5 # infection probability
p_r = 0.5 # recovery probability

def update():
    global g
    a = rd.choice(g.nodes())
    if g.node[a]['state'] == 0: # if susceptible
        b = rd.choice(g.neighbors(a))
        if g.node[b]['state'] == 1: # if neighbor b is infected
            g.node[a]['state'] = 1 if random() < p_i else 0
    else: # if infected
```

```
        g.node[a]['state'] = 0 if random() < p_r else 1
```

Again, you should set the step size to 50 to speed up the simulations. With these parameter settings ($p_i = p_r = 0.5$), you will probably find that the disease (state 1 = black nodes) quickly dies off and disappears from the network, even though half of the initial population is infected at the beginning. This means that for these particular parameter values and network topology, the system can successfully suppress a pandemic without taking any special action.

> *Exercise 16.4* Conduct simulations of the SIS model with either p_i or p_r varied systematically, while the other one is fixed at 0.5. Determine the condition in which a pandemic occurs (i.e., the disease persists for an indefinitely long period of time). Is the transition gradual or sharp? Once you get the results, try varying the other parameter as well.

> *Exercise 16.5* Generate a much larger random network of your choice and conduct the same SIS model simulation on it. See how the dynamics are affected by the change of network size and/or topology. Will the disease continue to persist on the network?

Continuous state/time models (1): Diffusion model So far, we haven't seen any equation in this chapter, because all the models discussed above were described in algorithmic rules with stochastic factors involved. But if they are deterministic, dynamical network models are not fundamentally different from other conventional dynamical systems. In fact, any deterministic dynamical model of a network made of n nodes can be described in one of the following standard formulations using an n-dimensional vector state x,

$$x_t = F(x_{t-1}, t) \qquad \text{(for discrete-time models), or} \qquad (16.1)$$

$$\frac{dx}{dt} = F(x, t) \qquad \text{(for continuous-time models),} \qquad (16.2)$$

if function F correctly represents the interrelationships between the different components of x. For example, the local majority rule network model we discussed earlier in this chapter is fully deterministic, so its behavior can be captured entirely in a set of difference equations in the form of Eq. (16.1). In this and next subsections, we will discuss two illustrative examples of the differential equation version of dynamics *on* networks models, i.e., the diffusion model and the coupled oscillator model. They are both extensively studied in network science.

Diffusion on a network can be a generalization of spatial diffusion models into non-regular, non-homogeneous spatial topologies. Each node represents a local site where some "stuff" can be accumulated, and each symmetric edge represents a channel through which the stuff can be transported, one way or the other, driven by the gradient of its concentration. This can be a useful model of the migration of species between geographically semi-isolated habitats, flow of currency between cities across a nation, dissemination of organizational culture within a firm, and so on. The basic assumption is that the flow of the stuff is determined by the difference in its concentration across the edge:

$$\frac{dc_i}{dt} = \alpha \sum_{j \in N_i} (c_j - c_i) \tag{16.3}$$

Here c_i is the concentration of the stuff on node i, α is the diffusion constant, and N_i is the set of node i's neighbors. Inside the parentheses $(c_j - c_i)$ represents the difference in the concentration between node j and node i across the edge (i, j). If neighbor j has more stuff than node i, there is an influx from j to i, causing a positive effect on dc_i/dt. Or if neighbor j has less than node i, there is an outflux from i to j, causing a negative effect on dc_i/dt. This makes sense.

Note that the equation above is a linear dynamical system. So, if we represent the entire list of node states by a state vector $c = (c_1\, c_2\, \ldots\, c_n)^T$, Eq. (16.3) can be written as

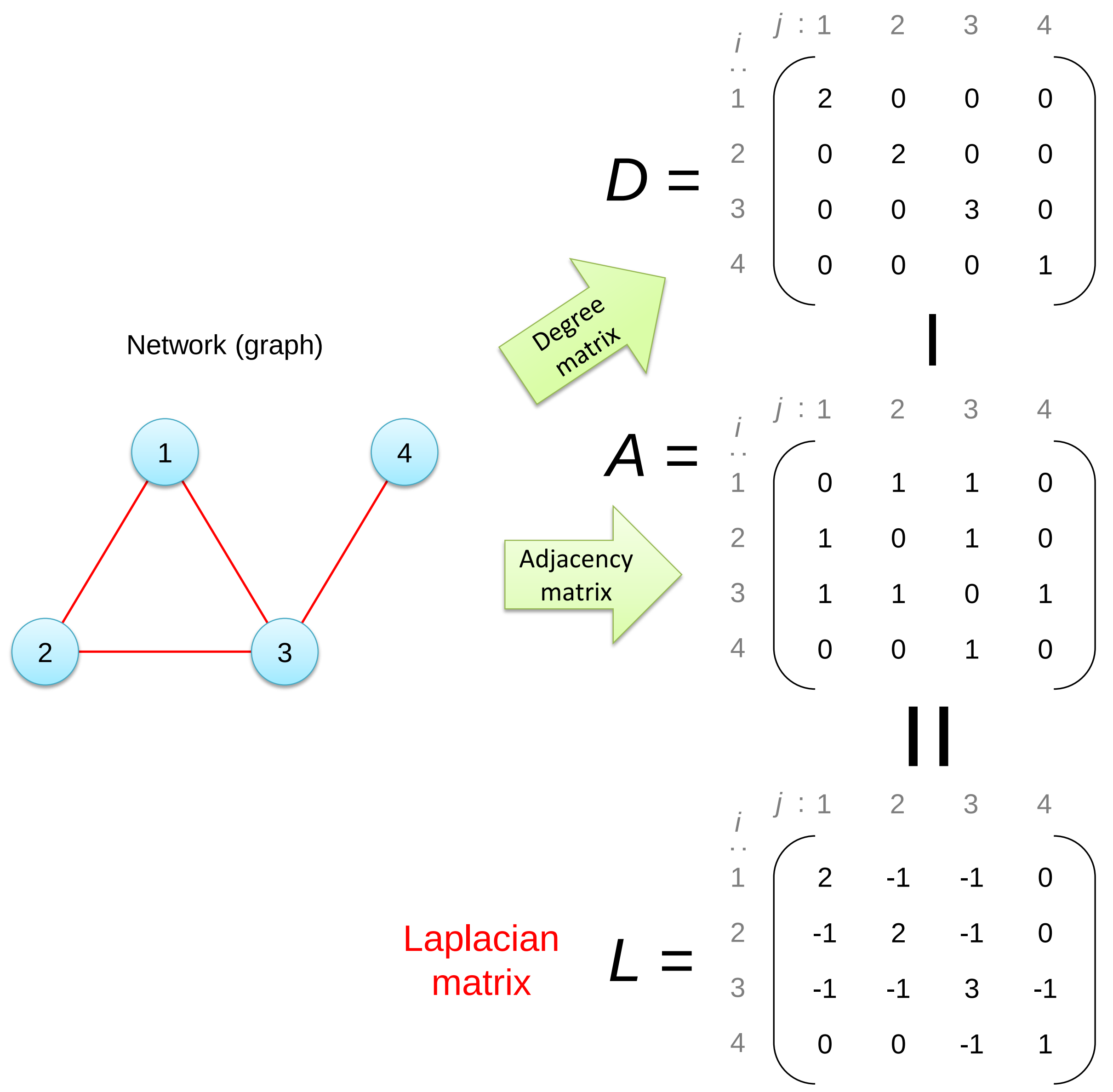
$$\frac{dc}{dt} = -\alpha L c, \tag{16.4}$$

where L is what is called a *Laplacian matrix* of the network, which is defined as

$$L = D - A, \tag{16.5}$$

where A is the adjacency matrix of the network, and D is the *degree matrix* of the network, i.e., a matrix whose i-th diagonal component is the degree of node i while all other components are 0. An example of those matrices is shown in Fig. 16.4.

Wait a minute. We already heard "Laplacian" many times when we discussed PDEs. The Laplacian operator (∇^2) appeared in spatial diffusion equations, while this new Laplacian matrix thing also appears in a network-based diffusion equation. Are they related? The answer is *yes.* In fact, they are (almost) identical linear operators in terms of their role. Remember that when we discretized Laplacian operators in PDEs to simulate using CA (Eq. (13.35)), we learned that a discrete version of a Laplacian can be calculated by the following principle:



$$D = \begin{array}{c|cccc} i \backslash j & 1 & 2 & 3 & 4 \\ \hline 1 & 2 & 0 & 0 & 0 \\ 2 & 0 & 2 & 0 & 0 \\ 3 & 0 & 0 & 3 & 0 \\ 4 & 0 & 0 & 0 & 1 \end{array}$$

$$-$$

$$A = \begin{array}{c|cccc} i \backslash j & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 1 & 1 & 0 \\ 2 & 1 & 0 & 1 & 0 \\ 3 & 1 & 1 & 0 & 1 \\ 4 & 0 & 0 & 1 & 0 \end{array}$$

$$=$$

$$L = \begin{array}{c|cccc} i \backslash j & 1 & 2 & 3 & 4 \\ \hline 1 & 2 & -1 & -1 & 0 \\ 2 & -1 & 2 & -1 & 0 \\ 3 & -1 & -1 & 3 & -1 \\ 4 & 0 & 0 & -1 & 1 \end{array}$$

Figure 16.4: Relationship among a degree matrix, an adjacent matrix, and a Laplacian matrix.

"Measure the difference between your neighbor and yourself, and then sum up all those differences."

Note that this is exactly what we did on a network in Eq. (16.3)! So, essentially, the Laplacian matrix of a graph is a discrete equivalent of the Laplacian operator for continuous space. The only difference, which is quite unfortunate in my opinion, is that they are defined with opposite signs for historical reasons; compare the first term of Eq. (13.17) and Eq.(16.4), and you will see that the Laplacian matrix did not absorb the minus sign inside it. So, conceptually,

$$\nabla^2 \Leftrightarrow -L. \tag{16.6}$$

I have always thought that this mismatch of signs was so confusing, but both of these "Laplacians" are already fully established in their respective fields. So, we just have to live with these inconsistent definitions of "Laplacians."

Now that we have the mathematical equations for diffusion on a network, we can discretize time to simulate their dynamics in Python. Eq. (16.3) becomes

$$\begin{aligned} c_i(t+\Delta t) &= c_i(t) + \left[\alpha \sum_{j \in N_i} \big(c_j(t) - c_i(t)\big)\right] \Delta t & (16.7) \\ &= c_i(t) + \alpha \left[\left(\sum_{j \in N_i} c_j(t)\right) - c_i(t)\deg(i)\right] \Delta t. & (16.8) \end{aligned}$$

Or, equivalently, we can also discretize time in Eq. (16.4), i.e.,

$$\begin{aligned} c(t+\Delta t) &= c(t) - \alpha L c(t) \Delta t & (16.9) \\ &= (I - \alpha L \Delta t)\, c(t), & (16.10) \end{aligned}$$

where c is now the state vector for the entire network, and I is the identity matrix. Eqs. (16.8) and (16.10) represent exactly the same diffusion dynamics, but we will use Eq. (16.8) for the simulation in the following, because it won't require matrix representation (which could be inefficient in terms of memory use if the network is large and sparse).

We can reuse Code 16.4 for this simulation. We just need to replace the `update` function with the following:

Code 16.7: net-diffusion.py

```
alpha = 1 # diffusion constant
Dt = 0.01 # Delta t
```

```
def update():
    global g, nextg
    for i in g.nodes_iter():
        ci = g.node[i]['state']
        nextg.node[i]['state'] = ci + alpha * ( \
            sum(g.node[j]['state'] for j in g.neighbors(i)) \
            - ci * g.degree(i)) * Dt
    g, nextg = nextg, g
```

And then we can simulate a nice smooth diffusion process on a network, as shown in Fig. 16.5, where continuous node states are represented by shades of gray. You can see that the diffusion makes the entire network converge to a homogeneous configuration with the average node state (around 0.5, or half gray) everywhere.

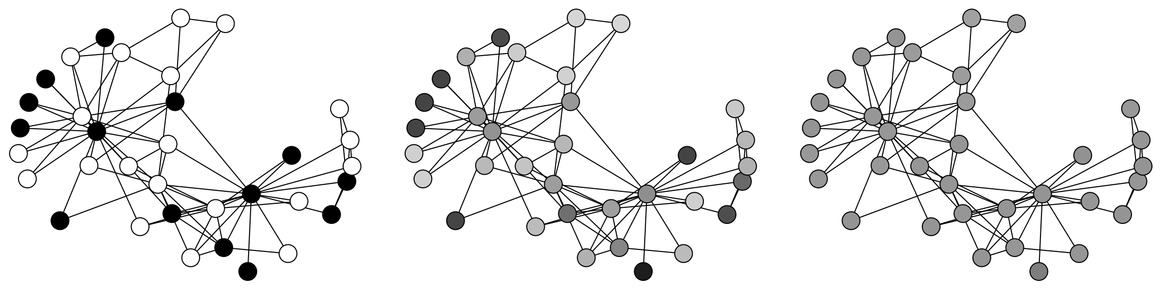


Figure 16.5: Visual output of Code 16.7. Time flows from left to right.

Exercise 16.6 This diffusion model conserves the sum of the node states. Confirm this by revising the code to measure the sum of the node states during the simulation.

Exercise 16.7 Simulate a diffusion process on each of the following network topologies, and then discuss how the network topology affects the diffusion on it. For example, does it make diffusion faster or slower?

- random graph

- barbell graph
- ring-shaped graph (i.e., degree-2 regular graph)

Continuous state/time models (2): Coupled oscillator model Now that we have a diffusion model on a network, we can naturally extend it to reaction-diffusion dynamics as well, just like we did with PDEs. Its mathematical formulation is quite straightforward; you just need to add a local reaction term to Eq. (16.3), to obtain

$$\frac{dc_i}{dt} = R_i(c_i) + \alpha \sum_{j \in N_i} (c_j - c_i). \tag{16.11}$$

You can throw in any local dynamics to the reaction term R_i. If each node takes vector-valued states (like PDE-based reaction-diffusion systems), then you may also have different diffusion constants (α_1, α_2, $\ldots$) that correspond to multiple dimensions of the state space. Some researchers even consider different network topologies for different dimensions of the state space. Such networks made of superposed network topologies are called *multiplex networks*, which is a very hot topic actively studied right now (as of 2015), but we don't cover it in this textbook.

For simplicity, here we limit our consideration to scalar-valued node states only. Even with scalar-valued states, there are some very interesting dynamical network models. A classic example is *coupled oscillators.* Assume you have a bunch of oscillators, each of which tends to oscillate at its own inherent frequency in isolation. This inherent frequency is slightly different from oscillator to oscillator. But when connected together in a certain network topology, the oscillators begin to influence each other's oscillation phases. A key question that can be answered using this model is: *When and how do these oscillators synchronize?* This problem about *synchronization* of the collective is an interesting problem that arises in many areas of complex systems [69]: firing patterns of spatially distributed fireflies, excitation patterns of neurons, and behavior of traders in financial markets. In each of those systems, individual behaviors naturally have some inherent variations. Yet if the connections among them are strong enough and meet certain conditions, those individuals begin to orchestrate their behaviors and may show a globally synchronized behavior (Fig.16.6), which may be good or bad, depending on the context. This problem can be studied using network models. In fact, addressing this problem was part of the original motivation for Duncan Watts' and Steven Strogatz's "small-world" paper published in the late 1990s [56], one of the landmark papers that helped create the field of network science.

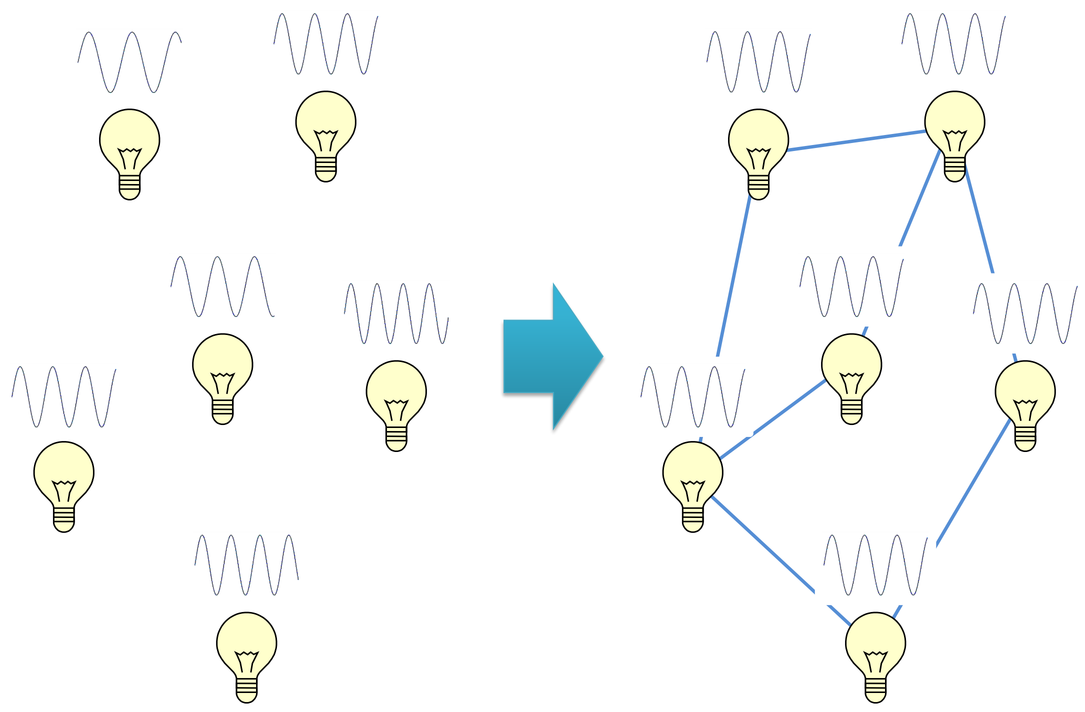


Figure 16.6: Schematic illustration of coupled oscillators. Light bulbs that change light intensity periodically at different frequencies (left) may synchronize globally if connected (right).

Representing oscillatory behavior naturally requires two or more dynamical variables, as we learned earlier. But purely harmonic oscillation can be reduced to a simple linear motion with constant velocity, if the behavior is interpreted as a projection of uniform circular motion and if the state is described in terms of *angle* θ. This turns the reaction term in Eq. (16.11) into just a constant angular velocity ω_i. In the meantime, this representation also affects the diffusion term, because the difference between two θ's is no longer computable by simple arithmetic subtraction like $\theta_j - \theta_i$, because there are infinitely many θ's that are equivalent (e.g., 0, 2π, -2π, 4π, -4π, etc.). So, the difference between the two states in the diffusion term has to be modified to focus only on the actual "phase difference" between the two nodes. You can imagine marching soldiers on a circular track as a visual example of this model (Fig. 16.7). Two soldiers who are far apart in θ may actually be close to each other in physical space (i.e., oscillation phase). The model should be set up in such a way that they try to come closer to each other in the physical space, not in the angular space.

To represent this type of phase-based interaction among coupled oscillators, a Japanese physicist Yoshiki Kuramoto proposed the following very simple, elegant mathematical model in the 1970s [70]:

$$\frac{d\theta_i}{dt} = \omega_i + \alpha \frac{\sum_{j \in N_i} \sin(\theta_j - \theta_i)}{|N_i|} \tag{16.12}$$

The angular difference part was replaced by a *sine function* of angular difference, which becomes positive if j is physically ahead of i, or negative if j is physically behind i on the circular track (because adding or subtracting 2π inside $\sin$ won't affect the result). These forces that soldier i receives from his or her neighbors will be averaged and used to determine the adjustment of his or her movement. The parameter α determines the strength of the couplings among the soldiers. Note that the original Kuramoto model used a fully connected network topology, but here we are considering the same dynamics on a network of a nontrivial topology.

Let's do some simulations to see if our networked soldiers can self-organize to march in sync. We can take the previous code for network diffusion (Code 16.7) and revise it for this Kuramoto model. There are several changes we need to implement. First, each node needs not only its dynamic state (θ_i) but also its static preferred velocity (ω_i), the latter of which should be initialized so that there are some variations among the nodes. Second, the visualization function should convert the states into some bounded values, since angular position θ_i continuously increases toward infinity. We can use $\sin(\theta_i)$ for this visualization purpose. Third, the updating rule needs to be revised, of course, to implement Eq. (16.12).

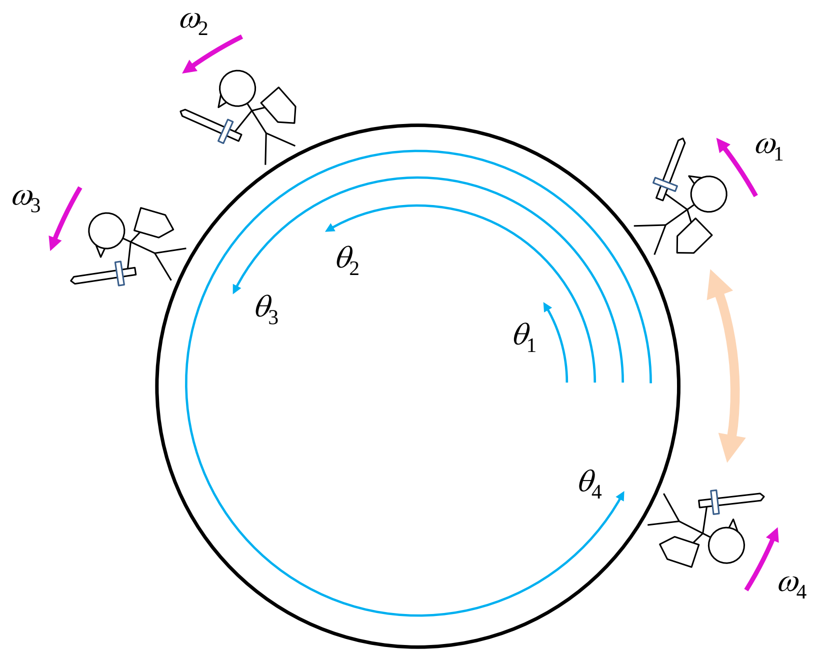


Figure 16.7: Marching soldiers on a circular track, as a visual representation of coupled oscillators described in angular space, where soldier 1 and his or her neighbors on the network (2, 3, 4) are illustrated. Each soldier has his or her own preferred speed (ω_i). The neighbors' physical proximity (not necessarily their angular proximity; see θ_1 and θ_4) should be used to determine in which direction soldier 1's angular velocity is adjusted.

Here is an example of the revised code, where I also changed the color map to `cm.hsv` so that the state can be visualized in a more cyclic manner:

Code 16.8: net-kuramoto.py

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

def initialize():
    global g, nextg
    g = nx.karate_club_graph()
    g.pos = nx.spring_layout(g)
    for i in g.nodes_iter():
        g.node[i]['theta'] = 2 * pi * random()
        g.node[i]['omega'] = 1. + uniform(-0.05, 0.05)
    nextg = g.copy()

def observe():
    global g, nextg
    cla()
    nx.draw(g, cmap = cm.hsv, vmin = -1, vmax = 1,
            node_color = [sin(g.node[i]['theta']) for i in g.nodes_iter()],
            pos = g.pos)

alpha = 1 # coupling strength
Dt = 0.01 # Delta t

def update():
    global g, nextg
    for i in g.nodes_iter():
        theta_i = g.node[i]['theta']
        nextg.node[i]['theta'] = theta_i + (g.node[i]['omega'] + alpha * ( \
            sum(sin(g.node[j]['theta'] - theta_i) for j in g.neighbors(i)) \
            / g.degree(i))) * Dt
    g, nextg = nextg, g
```

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

Figure 16.8 shows a typical simulation run. The dynamics of this simulation can be better visualized if you increase the step size to 50. Run it yourself, and see how the inherently diverse nodes get self-organized to start a synchronized marching. Do you see any spatial patterns there?

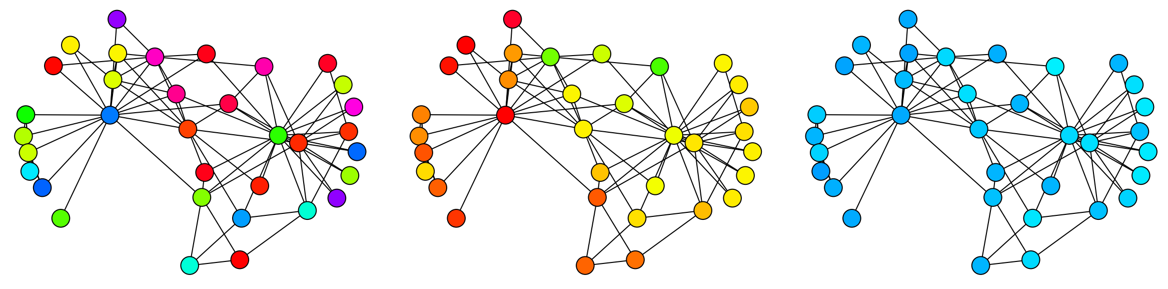


Figure 16.8: Visual output of Code 16.8. Time flows from left to right.

Exercise 16.8 It is known that the level of synchronization in the Kuramoto model (and many other coupled oscillator models) can be characterized by the following measurement (called *phase coherence*):

$$r = \left| \frac{1}{n} \sum_i e^{\mathbf{i}\theta_i} \right| \tag{16.13}$$

Here n is the number of nodes, θ_i is the state of node i, and $\mathbf{i}$ is the imaginary unit (to avoid confusion with node index i). This measurement becomes 1 if all the nodes are in perfect synchronization, or 0 if their phases are completely random and uniformly distributed in the angular space. Revise the simulation code so that you can measure and monitor how this phase coherence changes during the simulation.

Exercise 16.9 Simulate the Kuramoto model on each of the following network topologies:

- random graph
- barbell graph
- ring-shaped graph (i.e., degree-2 regular graph)

Discuss how the network topology affects the synchronization. Will it make synchronization easier or more difficult?

Exercise 16.10 Conduct simulations of the Kuramoto model by systematically increasing the amount of variations of ω_i (currently set to 0.05 in the `initialize` function) and see when/how the transition of the system behavior occurs.

Here are some more exercises of dynamics *on* networks models. Have fun!

Exercise 16.11 **Hopfield network** (a.k.a. *attractor network*) John Hopfield proposed a discrete state/time dynamical network model that can recover memorized arrangements of states from incomplete initial conditions [20, 21]. This was one of the pioneering works of artificial neural network research, and its basic principles are still actively used today in various computational intelligence applications.

Here are the typical assumptions made in the Hopfield network model:

- The nodes represent artificial neurons, which take either -1 or 1 as dynamic states.
- Their network is fully connected (i.e., a complete graph).
- The edges are weighted and symmetric.
- The node states will update synchronously in discrete time steps according to the following rule:

$$s_i(t+1) = \mathrm{sign}\left(\sum_j w_{ij} s_j(t)\right) \tag{16.14}$$

  Here, $s_i(t)$ is the state of node i at time t, w_{ij} is the edge weight between nodes i and j ($w_{ij} = w_{ji}$ because of symmetry), and $\mathrm{sign}(x)$ is a function that gives 1 if $x > 0$, -1 if $x < 0$, or 0 if $x = 0$.
- There are no self-loops ($w_{ii} = 0$ for all i).

What Hopfield showed is that one can "imprint" a finite number of pre-determined patterns into this network by carefully designing the edge weights using the following simple encoding formula:

$$w_{ij} = \sum_k s_{i,k} s_{j,k} \tag{16.15}$$

Here $s_{i,k}$ is the state of node i in the k-th pattern to be imprinted in the network. Implement a simulator of the Hopfield network model, and construct the edge weights w_{ij} from a few state patterns of your choice. Then simulate the dynamics of the network from a random initial condition and see how the network behaves. Once your model successfully demonstrates the recovery of imprinted patterns, try increasing the number of patterns to be imprinted, and see when/how the network loses the capability to memorize all of those patterns.

Exercise 16.12 **Cascading failure** This model is a continuous-state, discrete-time dynamical network model that represents how a functional failure of a component in an infrastructure network can trigger subsequent failures and cause a large-scale systemic failure of the whole network. It is often used as a stylized model of massive power blackouts, financial catastrophe, and other (undesirable) socio-technological and socio-economical events.

Here are the typical assumptions made in the cascading failure model:

- The nodes represent a component of an infrastructure network, such as power transmitters, or financial institutions. The nodes take non-negative real numbers as their dynamic states, which represent the amount of load or burden they are handling. The nodes can also take a specially designated "dead" state.
- Each node also has its own capacity as a static property.
- Their network can be in any topology.
- The edges can be directed or undirected.
- The node states will update either synchronously or asynchronously in discrete time steps, according to the following simple rules:
  - If the node is dead, nothing happens.
  - If the node is not dead but its load exceeds its capacity, it will turn to a dead state, and the load it was handling will be evenly distributed to its

neighbors that are still alive.

Implement a simulator of the cascading failure model, and simulate the dynamics of the network from an initial condition that is constructed with randomly assigned loads and capacities, with one initially overloaded node. Try increasing the average load level relative to the average capacity level, and see when/how the network begins to show a cascading failure. Investigate which node has the most significant impact when it is overloaded. Also try several different network topologies to see if topology affects the resilience of the networks.

Exercise 16.13 Create a continuous state/time network model of coupled oscillators where each oscillator tries to *desynchronize* from its neighbors. You can modify the Kuramoto model to develop such a model. Then simulate the network from an initial condition where the nodes' phases are almost the same, and demonstrate that the network can spontaneously diversify node phases. Discuss potential application areas for such a model.

16.3 Simulating Dynamics *of* Networks

Dynamics *of* networks models capture completely different kinds of network dynamics, i.e., changes in network topologies. This includes the addition and removal of nodes and edges over time. As discussed in the previous chapter, such dynamic changes of the system's topology itself are quite unusual from a traditional dynamical systems viewpoint, because they would make it impossible to assume a well-defined static phase space of the system. But permitting such topological changes opens up a whole new set of possibilities to model various natural and social phenomena that were hard to capture in conventional dynamical systems frameworks, such as:

- Evolutionary changes of gene regulatory and metabolic networks
- Self-organization and adaptation of food webs
- Social network formation and evolution
- Growth of infrastructure networks (e.g., traffic networks, power grids, the Internet, WWW)

- Growth of scientific citation networks

As seen above, growth and evolution of networks in society are particularly well studied using dynamics *of* network models. This is partly because their temporal changes take place much faster than other natural networks that change at ecological/evolutionary time scales, and thus researchers can compile a large amount of temporal network data relatively easily.

One iconic problem that has been discussed about social networks is this: *Why is our world so "small" even though there are millions of people in it?* This was called the *"small-world" problem* by social psychologist Stanley Milgram, who experimentally demonstrated this through his famous "six degrees of separation" experiment in the 1960s [71]. This empirical fact, that any pair of human individuals in our society is likely to be connected through a path made of only a small number of social ties, has been puzzling many people, including researchers. This problem doesn't sound trivial, because we usually have only the information about local social connections around ourselves, without knowing much about how each one of our acquaintances is connected to the rest of the world. But it is probably true that you are connected to, say, the President of the United States by fewer than ten social links.

This small-world problem already had a classic mathematical explanation. If the network is purely random, then the average distance between pairs of nodes are extremely small. In this sense, Erdős-Rényi random graph models were already able to explain the small-world problem beautifully. However, there is an issue with this explanation: *How come a person who has local information only can get connected to individuals randomly chosen from the entire society, who might be living on the opposite side of the globe?* This is a legitimate concern because, after all, most of our social connections are within a close circle of people around us. Most social connections are very local in both geographical and social senses, and there is no way we can create a truly random network in the real world. Therefore, we need different kinds of mechanisms by which social networks change their topologies to acquire the "small-world" property.

In the following, we will discuss two dynamics *of* networks models that can nicely explain the small-world problem in more realistic scenarios. Interestingly, these models were published at about the same time, in the late 1990s, and both contributed greatly to the establishment of the new field of network science.

Small-world networks by random edge rewiring In 1998, Duncan Watts and Steven Strogatz addressed this paradox, that social networks are "small" yet highly clustered locally, by introducing the *small-world network* model [56]. The *Watts-Strogatz model* is based on a random edge rewiring procedure to gradually modify network topology from a completely regular, local, clustered topology to a completely random, global, unclustered one. In the middle of this random rewiring process, they found networks that had the "small-world" property yet still maintained locally clustered topologies. They named these networks "small-world" networks. Note that having the "small-world" property is not a sufficient condition for a network to be called "small-world" in Watts-Strogatz sense. The network should also have high local clustering.

Watts and Strogatz didn't propose their random edge rewiring procedure as a dynamical process over time, but we can still simulate it as a dynamics *on* networks model. Here are their original model assumptions:

1. The initial network topology is a ring-shaped network made of n nodes. Each node is connected to k nearest neighbors (i.e., $k/2$ nearest neighbors clockwise and $k/2$ nearest neighbors counterclockwise) by undirected edges.

2. Each edge is subject to random rewiring with probability p. If selected for random rewiring, one of the two ends of the edge is reconnected to a node that is randomly chosen from the whole network. If this rewiring causes duplicate edges, the rewiring is canceled.

Watts and Strogatz's original model did the random edge rewiring (step 2 above) by sequentially visiting each node clockwise. But here, we can make a minor revision to the model so that the edge rewiring represents a more realistic social event, such as a random encounter at the airport of two individuals who live far apart from each other, etc. Here are the new, revised model assumptions:

1. The initial network topology is the same as described above.

2. In each edge rewiring event, a node is randomly selected from the whole network. The node drops one of its edges randomly, and then creates a new edge to a new node that is randomly chosen from the whole network (excluding those to which the node is already connected).

This model captures essentially the same procedure as Watts and Strogatz's, but the rewiring probability p is now represented implicitly by the length of the simulation. If the simulation continues indefinitely, then the network will eventually become completely random. Such a dynamical process can be interpreted as a model of social evolution in which an initially locally clustered society gradually accumulates more and more global connections that are caused by rare, random long-range encounters among people.

Let's simulate this model using Python. When you simulate the dynamics *of* networks, one practical challenge is how to visualize *topological changes* of the network. Unlike node states that can be easily visualized using colors, network topology is the shape of the network itself, and if the shape of the network is changing dynamically, we also need to simulate the physical movement of nodes in the visualization space as well. This is just for aesthetic visualization only, which has nothing to do with the real science going on in the network. But an effective visualization often helps us better understand what is going on in the simulation, so let's try some fancy animation here. Fortunately, NetworkX's `spring_layout` function can be used to update the current node positions slightly. For example:

Code 16.9:

```
g.pos = nx.spring_layout(g, pos = g.pos, iterations = 5)
```

This code calculates a new set of node positions by using `g.pos` as the initial positions and by applying the Fruchterman-Reingold force-directed algorithm to them for just five steps. This will effectively simulate a slight movement of the nodes from their current positions, which can be utilized to animate the topological changes smoothly.

Here is an example of the completed simulation code for the small-world network formation by random edge rewiring:

Code 16.10: small-world.py

```
import matplotlib
matplotlib.use('TkAgg')
from pylab import *
import networkx as nx
import random as rd

n = 30 # number of nodes
k = 4 # number of neighbors of each node

def initialize():
    global g
    g = nx.Graph()
    for i in xrange(n):
        for j in range(1, k/2 + 1):
            g.add_edge(i, (i + j) % n)
            g.add_edge(i, (i - j) % n)
    g.pos = nx.spring_layout(g)
    g.count = 0

def observe():
    global g
    cla()
    nx.draw(g, pos = g.pos)

def update():
    global g
    g.count += 1
    if g.count % 20 == 0: # rewiring once in every 20 steps
        nds = g.nodes()
        i = rd.choice(nds)
        if g.degree(i) > 0:
            g.remove_edge(i, rd.choice(g.neighbors(i)))
            nds.remove(i)
            for j in g.neighbors(i):
                nds.remove(j)
            g.add_edge(i, rd.choice(nds))

    # simulation of node movement
    g.pos = nx.spring_layout(g, pos = g.pos, iterations = 5)

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

The `initialize` function creates a ring-shaped network topology by connecting each node to its `k` nearest neighbors ($\mathtt{j} \in \{1, \ldots, k/2\}$). "`% n`" is used to confine the names of nodes to the domain $[0,\mathrm{n}-1]$ with periodic boundary conditions. An additional attribute `g.count` is also created in order to count time steps between events of topological changes. In the `update` function, `g.count` is incremented by one, and the random edge rewiring is simulated in every 20 steps as follows: First, node `i` is randomly chosen, and then one of its connections is removed. Then node `i` itself and its neighbors are removed from the candidate node list. A new edge is then created between `i` and a node randomly chosen from the remaining candidate nodes. Finally, regardless of whether a random edge rewiring occurred or not, node movement is simulated using the `spring_layout` function.

Running this code will give you an animated process of random edge rewiring, which gradually turns a regular, local, clustered network to a random, global, unclustered one (Fig. 16.9). Somewhere in this network dynamics, you will see the Watts-Strogatz small-world network arise, but just temporarily (which will be discussed further in the next chapter).

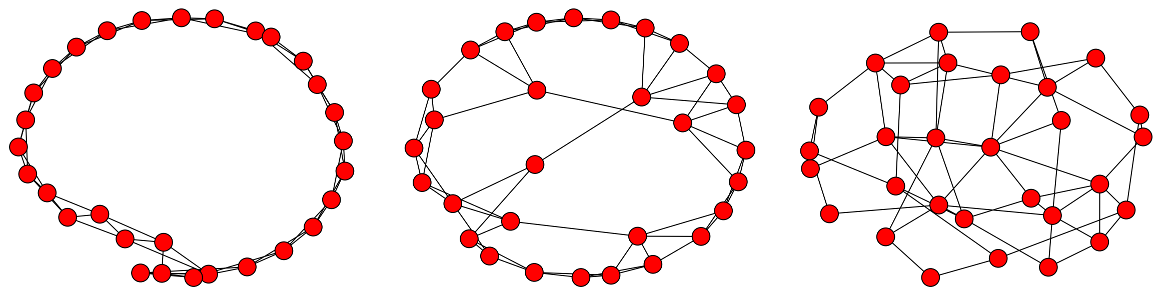


Figure 16.9: Visual output of Code 16.10. Time flows from left to right.

Exercise 16.14 Revise the small-world network formation model above so that the network is initially a two-dimensional grid in which each node is connected to its four neighbors (north, south, east, and west; except for those on the boundaries of the space). Then run the simulations, and see how random edge rewiring changes the topology of the network.

For your information, NetworkX already has a built-in graph generator function for Watts-Strogatz small-world networks, `watts_strogatz_graph(n, k, p)`. Here, `n` is the number of nodes, `k` is the degree of each node, and `p` is the edge rewiring probability. If you don't need to simulate the formation of small-world networks iteratively, you should just use this function instead.

Scale-free networks by preferential attachment The Watts-Strogatz small-world network model brought about a major breakthrough in explaining properties of real-world networks using abstract network models. But of course, it was not free from limitations. First, their model required a "Goldilocks" rewiring probability p (or such a simulation length in our revised model) in order to obtain a small-world network. This means that their model couldn't explain how such an appropriate p would be maintained in real social networks. Second, the small-world networks generated by their model didn't capture one important aspect of real-world networks: heterogeneity of connectivities. In every level of society, we often find a few very popular people as well as many other less-popular, "ordinary" people. In other words, there is always great variation in node degrees. However, since the Watts-Strogatz model modifies an initially regular graph by a series of random rewirings, the resulting networks are still quite close to regular, where each node has more or less the same number of connections. This is quite different from what we usually see in reality.

Just one year after the publication of Watts and Strogatz's paper, Albert-László Barabási and Réka Albert published another very influential paper [57] about a new model that could explain both the small-world property and large variations of node degrees in a network. The *Barabási-Albert model* described self-organization of networks over time caused by a series of *network growth* events with *preferential attachment*. Their model assumptions were as follows:

1. The initial network topology is an arbitrary graph made of m_0 nodes. There is no specific requirement for its shape, as long as each node has at least one connection (so its degree is positive).

2. In each network growth event, a newcomer node is attached to the network by m edges ($m \leq m_0$). The destination of each edge is selected from the network of existing nodes using the following selection probabilities:

$$p(i) = \frac{\deg(i)}{\sum_j \deg(j)} \tag{16.16}$$

   Here $p(i)$ is the probability for an existing node i to be connected by a newcomer node, which is proportional to its degree (preferential attachment).

This preferential attachment mechanism captures "the rich get richer" effect in the growth of systems, which is often seen in many socio-economical, ecological, and physical processes. Such cumulative advantage is also called the *"Matthew effect"* in sociology. What Barabási and Albert found was that the network growing with this simple dynamical rule will eventually form a certain type of topology in which the distribution of the node degrees follows a *power law*

$$P(k) \sim k^{-\gamma}, \tag{16.17}$$

where $P(k)$ is the probability for a node to have degree k, and $-\gamma$ is the scaling exponent ($\gamma = 3$ in the Barabási-Albert model). Since the exponent is negative, this distribution means that most nodes have very small k, while only a few nodes have large k (Fig. 16.10). But the key implication is that such super-popular *"hub"* nodes *do exist* in this distribution, which wouldn't be possible if the distribution was a normal distribution, for example. This is because a power law distribution has a *long tail* (also called a fat tail, heavy tail, etc.), in which the probability goes down much more slowly than in the tail of a normal distribution as k gets larger. Barabási and Albert called networks whose degree distributions follow a power law *scale-free networks*, because there is no characteristic "scale" that stands out in their degree distributions.

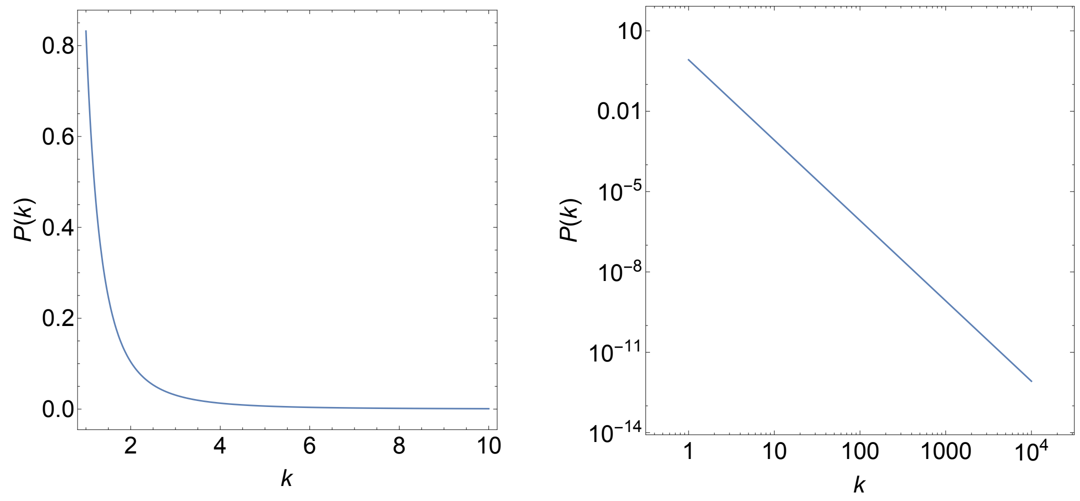


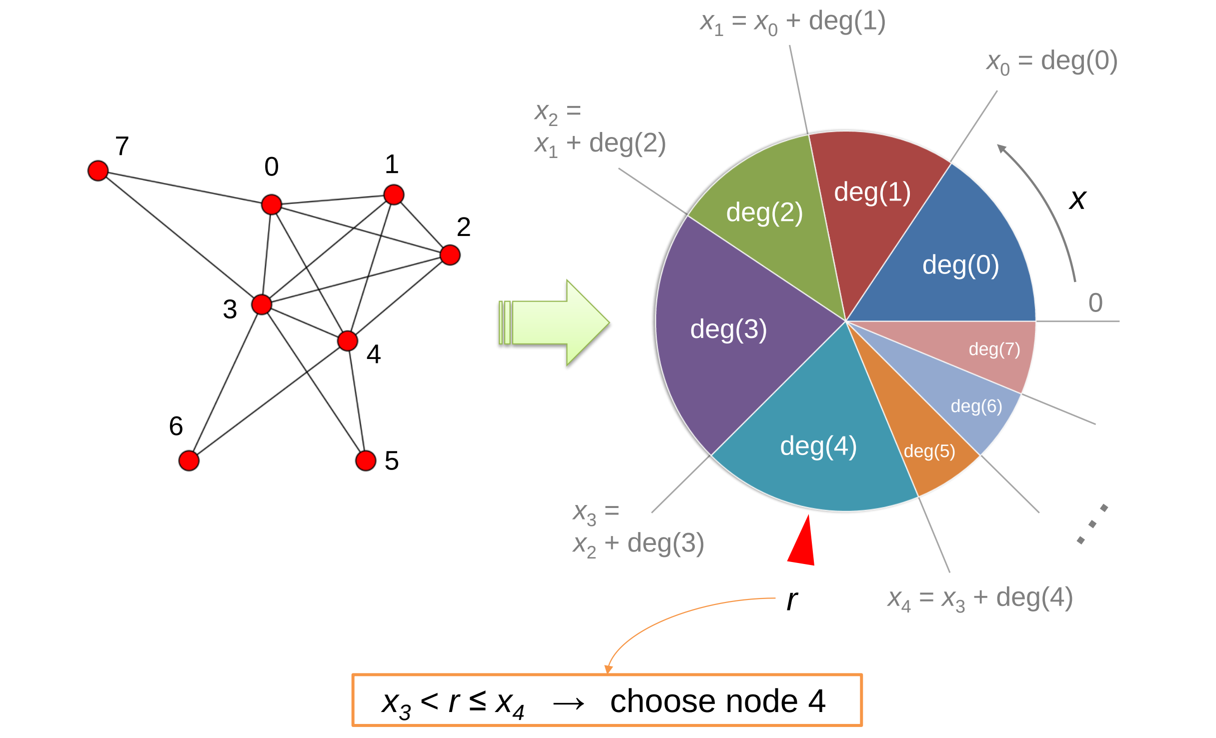
Figure 16.10: Plots of the power law distribution $P(k) \sim k^{-\gamma}$ with $\gamma = 3$. Left: In linear scales. Right: In log-log scales.

The network growth with preferential attachments is an interesting, fun process to simulate. We can reuse most of Code 16.10 for this purpose as well. One critical component we will need to newly define is the preferential node selection function that randomly chooses a node based on node degrees. This can be accomplished by *roulette selection*, i.e., creating a "roulette" from the node degrees and then deciding which bin in the roulette a randomly generated number falls into. In Python, this preferential node selection function can be written as follows:

Code 16.11:

```
def pref_select(nds):
    global g
    r = uniform(0, sum(g.degree(i) for i in nds))
    x = 0
    for i in nds:
        x += g.degree(i)
        if r <= x:
            return i
```

Here, `nds` is the list of node IDs, `r` is a randomly generated number, and `x` is the position of the boundary between the currently considered bin and the next one. The `for` loop moves the value of `x` from the beginning of the roulette (0) to each boundary of bins sequentially, and returns the node ID (`i`) as soon as it detects that the current bin contains `r` (i.e., `r <= x`). See Fig. 16.11 for a visual illustration of this process.



7
0
1
2
3
4
5
6
$x_1 = x_0 + \mathrm{deg}(1)$
$x_0 = \mathrm{deg}(0)$
$x_2 = x_1 + \mathrm{deg}(2)$
x
0
deg(0)
deg(1)
deg(2)
deg(3)
deg(4)
deg(5)
deg(6)
deg(7)
$x_3 = x_2 + \mathrm{deg}(3)$
r
$x_4 = x_3 + \mathrm{deg}(4)$
$x_3 < r \leq x_4 \;\rightarrow\;$ choose node 4

Figure 16.11: Schematic illustration of the roulette selection implemented in Code 16.11.

With this preferential node selection function, the completed simulation code looks like this:

Code 16.12: barabasi-albert.py

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

m0 = 5 # number of nodes in initial condition
m = 2 # number of edges per new node

def initialize():
    global g
    g = nx.complete_graph(m0)
    g.pos = nx.spring_layout(g)
    g.count = 0

def observe():
    global g
    cla()
    nx.draw(g, pos = g.pos)

def pref_select(nds):
    global g
    r = uniform(0, sum(g.degree(i) for i in nds))
    x = 0
    for i in nds:
        x += g.degree(i)
        if r <= x:
            return i

def update():
    global g
    g.count += 1
    if g.count % 20 == 0: # network growth once in every 20 steps
        nds = g.nodes()
        newcomer = max(nds) + 1
        for i in range(m):
            j = pref_select(nds)
            g.add_edge(newcomer, j)
            nds.remove(j)
        g.pos[newcomer] = (0, 0)

    # simulation of node movement
    g.pos = nx.spring_layout(g, pos = g.pos, iterations = 5)

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

In this sample code, the network growth starts with a five-node complete graph. Each time a newcomer node is added to the network, two new edges are established. Note that every time an edge is created between the newcomer and an existing node `j`, node `j` is removed from the candidate list `nds` so that no duplicate edges will be created. Finally, a new position `(0, 0)` for the newcomer is added to `g.pos`. Since `g.pos` is a Python dictionary, a new entry can be inserted just by using `newcomer` as a key.

Figure 16.12 shows a typical simulation result in which you see highly connected hub nodes spontaneously arise. When you continue this simulation long enough, the resulting distribution of node degrees comes closer to a power law distribution with $\gamma = 3$.

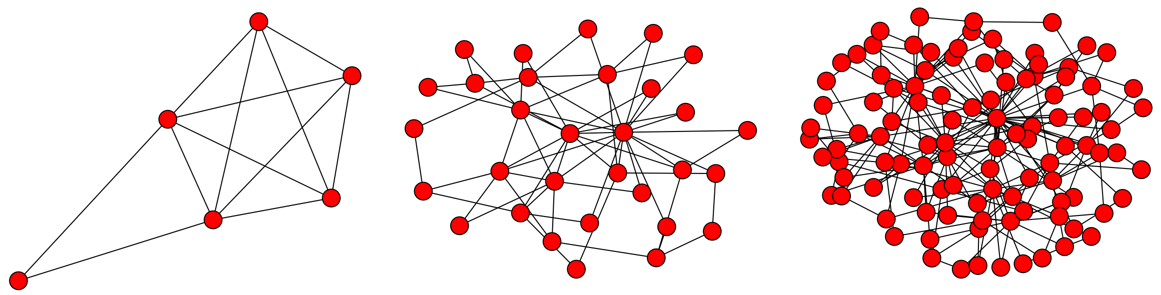


Figure 16.12: Visual output of Code 16.12. Time flows from left to right.

Exercise 16.15 Simulate the Barabási-Albert network growth model with $m = 1$, $m = 3$, and $m = 5$, and see how the growth process may be affected by the variation of this parameter.

Exercise 16.16 Modify the preferential node selection function so that the node selection probability $p(i)$ is each of the following:

- independent of the node degree (random attachment)
- proportional to the square of the node degree (strong preferential attachment)
- inversely proportional to the node degree (negative preferential attachment)

Conduct simulations for these cases and compare the resulting network topologies.

Note that NetworkX has a built-in graph generator function for the Barabási-Albert scale-free networks too, called `barabasi_albert_graph(n, m)`. Here, `n` is the number of nodes, and `m` the number of edges by which each newcomer node is connected to the network.

Here are some more exercises of dynamics *of* networks models, for your further exploration:

Exercise 16.17 **Closing triangles** This example is a somewhat reversed version of the Watts-Strogatz small-world network model. Instead of making a local, clustered network to a global, unclustered network, we can consider a dynamical process that makes an unclustered network more clustered over time. Here are the rules:

- The network is initially random, e.g., an Erdős-Rényi random graph.
- In each iteration, a two-edge path (A-B-C) is randomly selected from the network.
- If there is no edge between A and C, one of them loses one of its edges (but not the one that connects to B), and A and C are connected to each other instead.

This is an edge rewiring process that closes a triangle among A, B, and C, promoting what is called a *triadic closure* in sociology. Implement this edge rewiring model, conduct simulations, and see what kind of network topology results from this triadic closure rule. You can also combine this rule with the random edge rewiring used in the Watts-Strogatz model, to explore various balances between these two competing rules (globalization and localization) that shape the self-organization of the network topology.

Exercise 16.18 **Preferential attachment with node division** We can consider a modification of the Barabási-Albert model where each node has a capacity limit in terms of the number of connections it can hold. Assume that if a node's degree exceeds its predefined capacity, the node splits into two, and each node inherits about half of the connections the original node had. This kind of node division can be considered a representation of a split-up of a company or an organization, or the evolution of different specialized genes from a single gene that had many functions. Implement this modified network growth model, conduct simulations, and see how the node division influences the resulting network topology.

16.4 Simulating Adaptive Networks

The final class of dynamical network models is that of *adaptive networks.* It is a hybrid of dynamics *on* and *of* networks, where states and topologies "co-evolve," i.e., they interact with each other and keep changing, often over the same time scales. The word "adaptive" comes from the concept that states and topologies can adapt to each other in a co-evolutionary manner, although adaptation or co-evolution doesn't have to be biological in this context. Adaptive networks have been much less explored compared to the other two classes of models discussed above, but you can find many real-world examples of adaptive networks [67], such as:

- *Development of an organism.* The nodes are the cells and the edges are cell-cell adhesions and intercellular communications. The node states include intra-cellular gene regulatory and metabolic activities, which are coupled with topological changes caused by cell division, death, and migration.

- *Self-organization of ecological communities.* The nodes are the species and the edges are the ecological relationships (predation, symbiosis, etc.) among them. The node states include population levels and within-species genetic diversities, which are coupled with topological changes caused by invasion, extinction, adaptation, and speciation.

- *Epidemiological networks.* The nodes are the individuals and the edges are the physical contacts among them. The node states include their pathologic states (healthy or sick, infectious or not, etc.), which are coupled with topological changes caused by death, quarantine, and behavioral changes of those individuals.
- *Evolution of social communities.* The nodes are the individuals and the edges are social relationships, conversations, and/or collaborations. The node states include socio-cultural states, political opinions, wealth, and other social properties of those individuals, which are coupled with topological changes caused by the people's entry into or withdrawal from the community, as well as their behavioral changes.

In these adaptive networks, state transitions of each node and topological transformation of networks are deeply coupled with each other, which could produce characteristic behaviors that are not found in other forms of dynamical network models. As I mentioned earlier in this chapter, I am one of the proponents of this topic [66], so you should probably take what I say with a pinch of salt. But I firmly believe that modeling and predicting state-topology co-evolution of adaptive networks has become one of the major critical challenges in complex network research, especially because of the continual flood of temporal network data [65] that researchers are facing today.

The field of adaptive network research is still very young, and thus it is a bit challenging to select "well established" models as classic foundations of this research yet. So instead, I would like to show, with some concrete examples, how you can revise traditional dynamical network models into adaptive ones. You will probably find it very easy and straightforward to introduce adaptive network dynamics. There is nothing difficult in simulating adaptive network models, and yet the resulting behaviors may be quite unique and different from those of traditional models. This is definitely one of those unexplored research areas where your open-minded creativity in model development will yield you much fruit.

Adaptive epidemic model When you find out that one of your friends has caught a flu, you will probably not visit his or her room until he or she recovers from the illness. This means that human behavior, in response to the spread of illnesses, can change the topology of social ties, which in turn will influence the pathways of disease propagation. This is an illustrative example of adaptive network dynamics, which we can implement into the SIS model (Code 16.6) very easily. Here is the additional assumption we will add to the model:

- When a susceptible node finds its neighbor is infected by the disease, it will sever the edge to the infected node with severance probability p_s.

This new assumption is inspired by a pioneering adaptive network model of epidemic dynamics proposed by my collaborator Thilo Gross and his colleagues in 2006 [72]. They assumed that the edge from the susceptible to the infected nodes would be rewired to another susceptible node in order to keep the number of edges constant. But here, we assume that such edges can just be removed for simplicity. For your convenience, Code 16.6 for the SIS model is shown below again. Can you see where you can/should implement this new edge removal assumption in this code?

Code 16.13:

```
p_i = 0.5 # infection probability
p_r = 0.5 # recovery probability

def update():
    global g
    a = rd.choice(g.nodes())
    if g.node[a]['state'] == 0: # if susceptible
        b = rd.choice(g.neighbors(a))
        if g.node[b]['state'] == 1: # if neighbor b is infected
            g.node[a]['state'] = 1 if random() < p_i else 0
    else: # if infected
        g.node[a]['state'] = 0 if random() < p_r else 1
```

There are several different ways to implement adaptive edge removal in this code. An easy option would be just to insert another `if` statement to decide whether to cut the edge right before simulating the infection of the disease (third-to-last line). For example:

Code 16.14:

```
p_i = 0.5 # infection probability
p_r = 0.5 # recovery probability
p_s = 0.5 # severance probability

def update():
    global g
    a = rd.choice(g.nodes())
    if g.node[a]['state'] == 0: # if susceptible
        b = rd.choice(g.neighbors(a))
        if g.node[b]['state'] == 1: # if neighbor b is infected
            if random() < p_s:
                g.remove_edge(a, b)
            else:
                g.node[a]['state'] = 1 if random() < p_i else 0
    else: # if infected
        g.node[a]['state'] = 0 if random() < p_r else 1
```

Note that we now have one more probability, `p_s`, with which the susceptible node can sever the edge connecting itself to an infected neighbor. Since this simulation is implemented using an asynchronous state updating scheme, we can directly remove the edge `(a, b)` from the network as soon as node `a` decides to cut it. Such asynchronous updating is particularly suitable for simulating adaptive network dynamics in general.

By the way, there is a minor bug in the code above. Because there is a possibility for the edges to be removed from the network, some nodes may eventually lose all of their neighbors. If this happens, the `rd.choice(g.neighbors(a))` function causes an error. To avoid this problem, we need to check if node `a`'s degree is positive. Below is a slightly corrected code:

Code 16.15: SIS-model-adaptive.py

```
p_i = 0.5 # infection probability
p_r = 0.5 # recovery probability
p_s = 0.5 # severance probability

def update():
    global g
    a = rd.choice(g.nodes())
    if g.node[a]['state'] == 0: # if susceptible
        if g.degree(a) > 0:
            b = rd.choice(g.neighbors(a))
            if g.node[b]['state'] == 1: # if neighbor b is infected
                if random() < p_s:
                    g.remove_edge(a, b)
                else:
                    g.node[a]['state'] = 1 if random() < p_i else 0
    else: # if infected
        g.node[a]['state'] = 0 if random() < p_r else 1
```

If you run this code, the result will probably look quite similar to the original SIS model, because the parameter settings used in this code ($p_i = p_r = 0.5$) don't cause a pandemic.

But if you try different parameter settings that would cause a pandemic in the original SIS model (say, $p_i = 0.5$, $p_r = 0.2$), the effect of the adaptive edge removal is much more salient. Figure 16.13 shows a sample simulation result, where pandemic-causing parameter values ($p_i = 0.5$, $p_r = 0.2$) were used, but the edge severance probability p_s was also set to 0.5. Initially, the disease spreads throughout the network, but adaptive edge removal gradually removes edges from the network as an adaptive response to the pandemic, lowering the edge density and thus making it more and more difficult for the disease to spread. Eventually, the disease is eradicated when the edge density of the network hits a critical value below which the disease can no longer survive.

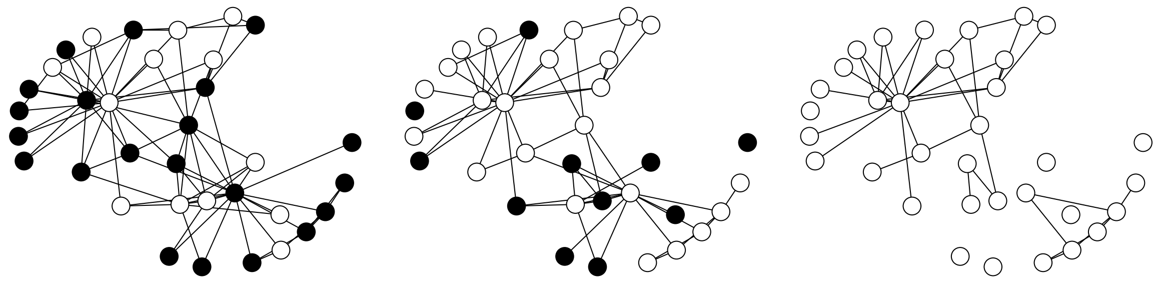


Figure 16.13: Visual output of Code 16.15, with varied parameter settings ($p_i = 0.5$, $p_r = 0.2$, $p_s = 0.5$). Time flows from left to right.

$\boxed{\textit{Exercise 16.19}}$ Conduct simulations of the adaptive SIS model on a random network of a larger size with $p_i = 0.5$ and $p_r = 0.2$, while varying p_s systematically. Determine the condition in which a pandemic will eventually be eradicated. Then try the same simulation on different network topologies (e.g, small-world, scale-free networks) and compare the results.

$\boxed{\textit{Exercise 16.20}}$ Implement a similar edge removal rule in the voter model so that an edge between two nodes that have opposite opinions can be removed probabilistically. Then conduct simulations with various edge severance probabilities to see how the consensus formation process is affected by the adaptive edge removal.

Adaptive diffusion model The final example in this chapter is the adaptive network version of the continuous state/time diffusion model, where the weight of a social tie can be strengthened or weakened according to the difference of the node states across the edge. This new assumption represents a simplified form of *homophily*, an empirically observed sociological fact that people tend to connect those who are similar to themselves. In contrast, the diffusion of node states can be considered a model of *social contagion*, another empirically observed sociological fact that people tend to become more similar to their social neighbors over time. There has been a lot of debate going on about which mechanism, homophily or social contagion, is more dominant in shaping the structure of social networks. This adaptive diffusion model attempts to combine these two mechanisms to investigate their subtle balance via computer simulations.

This example is actually inspired by an adaptive network model of a corporate merger that my collaborator Junichi Yamanoi and I presented recently [73]. We studied the conditions for two firms that recently underwent a merger to successfully assimilate and integrate their corporate cultures into a single, unified identity. The original model was a bit complex agent-based model that involved asymmetric edges, multi-dimensional state space, and stochastic decision making, so here we use a more simplified, deterministic, differential equation-based version. Here are the model assumptions:

- The network is initially made of two groups of nodes with two distinct cultural/ideological states.
- Each edge is undirected and has a weight, $w \in [0, 1]$, which represents the strength of the connection. Weights are initially set to 0.5 for all the edges.
- The diffusion of the node states occurs according to the following equation:

  $$\frac{dc_i}{dt} = \alpha \sum_{j \in N_i} (c_j - c_i) w_{ij} \tag{16.18}$$

  Here α is the diffusion constant and w_{ij} is the weight of the edge between node i and node j. The inclusion of w_{ij} signifies that diffusion takes place faster through edges with greater weights.
- In the meantime, each edge also changes its weight dynamically, according to the following equation:

  $$\frac{dw_{ij}}{dt} = \beta w_{ij}(1 - w_{ij})(1 - \gamma \, |c_i - c_j| \,) \tag{16.19}$$

Here β is the rate of adaptive edge weight change, and γ is a parameter that determines how intolerant, or "picky," each node is regarding cultural difference. For example, if $\gamma = 0$, w_{ij} always converges and stays at 1. But if γ is large (typically much larger than 1), the two nodes need to have very similar cultural states in order to maintain an edge between them, or otherwise the edge weight decreases. The inclusion of $w_{ij}(1-w_{ij})$ is to confine the weight values to the range $[0, 1]$ dynamically.

So, to simulate this model, we need a network made of two distinct groups. And this is the perfect moment to disclose a little more secrets about our favorite Karate Club graph (unless you have found it yourself already)! See the `node` attribute of the Karate Club graph, and you will find the following:

Code 16.16:

```
>>> nx.karate_club_graph().node
{0: {'club': 'Mr. Hi'}, 1: {'club': 'Mr. Hi'}, 2: {'club': 'Mr. Hi'},
  3: {'club': 'Mr. Hi'}, 4: {'club': 'Mr. Hi'}, 5: {'club': 'Mr. Hi'},
  6: {'club': 'Mr. Hi'}, 7: {'club': 'Mr. Hi'}, 8: {'club': 'Mr. Hi'},
  9: {'club': 'Officer'}, 10: {'club': 'Mr. Hi'}, 11: {'club': 'Mr. Hi'},
  12: {'club': 'Mr. Hi'}, 13: {'club': 'Mr. Hi'}, 14: {'club': 'Officer'},
  15: {'club': 'Officer'}, 16: {'club': 'Mr. Hi'}, 17: {'club': 'Mr. Hi'},
  18: {'club': 'Officer'}, 19: {'club': 'Mr. Hi'}, 20: {'club': 'Officer'},
  21: {'club': 'Mr. Hi'}, 22: {'club': 'Officer'}, 23: {'club': 'Officer'},
  24: {'club': 'Officer'}, 25: {'club': 'Officer'}, 26: {'club': 'Officer'},
  27: {'club': 'Officer'}, 28: {'club': 'Officer'}, 29: {'club': 'Officer'},
  30: {'club': 'Officer'}, 31: {'club': 'Officer'}, 32: {'club': 'Officer'},
  33: {'club': 'Officer'}}
```

Each node has an additional property, called `'club'`, and its values are either `'Mr. Hi'` or `'Officer'`! What are these?

The truth is, when Wayne Zachary studied this Karate Club, there was an intense political/ideological conflict between two factions. One was Mr. Hi, a part-time karate instructor hired by the club, and his supporters, while the other was the club president (Officer) John A., other officers, and their followers. They were in sharp conflict over the price of karate lessons. Mr. Hi wanted to raise the price, while the club president wanted to keep the price as it was. The conflict was so intense that the club eventually fired Mr. Hi, resulting in a fission of the club into two. The Karate Club graph is a snapshot of the club members' friendship network right before this fission, and therefore, each node comes with an attribute showing whether he or she was in Mr. Hi's camp or the Officer's camp. If you are interested in more details of this, you should read Zachary's original paper [59], which contains some more interesting stories.

This data looks perfect for our modeling purpose. We can set up the `initialize` function using this `'club'` property to assign binary states to the nodes. The implementation of dynamical equations are straightforward; we just need to discretize time and simulate them just as a discrete-time model, as we did before. Here is a completed code:

Code 16.17: net-diffusion-adaptive.py

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

def initialize():
    global g, nextg
    g = nx.karate_club_graph()
    for i, j in g.edges_iter():
        g.edge[i][j]['weight'] = 0.5
    g.pos = nx.spring_layout(g)
    for i in g.nodes_iter():
        g.node[i]['state'] = 1 if g.node[i]['club'] == 'Mr. Hi' else 0
    nextg = g.copy()

def observe():
    global g, nextg
    cla()
    nx.draw(g, cmap = cm.binary, vmin = 0, vmax = 1,
            node_color = [g.node[i]['state'] for i in g.nodes_iter()],
            edge_cmap = cm.binary, edge_vmin = 0, edge_vmax = 1,
            edge_color = [g.edge[i][j]['weight'] for i, j in g.edges_iter()],
            pos = g.pos)

alpha = 1 # diffusion constant
beta = 3 # rate of adaptive edge weight change
gamma = 3 # pickiness of nodes
Dt = 0.01 # Delta t
```

```
def update():
    global g, nextg
    for i in g.nodes_iter():
        ci = g.node[i]['state']
        nextg.node[i]['state'] = ci + alpha * ( \
            sum((g.node[j]['state'] - ci) * g.edge[i][j]['weight']
                for j in g.neighbors(i))) * Dt
    for i, j in g.edges_iter():
        wij = g.edge[i][j]['weight']
        nextg.edge[i][j]['weight'] = wij + beta * wij * (1 - wij) * ( \
            1 - gamma * abs(g.node[i]['state'] - g.node[j]['state'])
            ) * Dt
    nextg.pos = nx.spring_layout(nextg, pos = g.pos, iterations = 5)
    g, nextg = nextg, g

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

In the `initialize` function, the edge weights are all initialized as 0.5, while node states are set to 1 if the node belongs to Mr. Hi's faction, or 0 otherwise. There are also some modifications made to the `observe` function. Since the edges carry weights, we should color each edge in a different gray level. Therefore additional options (`edge_color`, `edge_cmap`, `edge_vmin`, and `edge_vmax`) were used to draw the edges in different colors. The `update` function is pretty straightforward, I hope.

Figure 16.14 shows the simulation result with $\alpha = 1$, $\beta = 3$, $\gamma = 3$. With this setting, unfortunately, the edge weights became weaker and weaker between the two political factions (while they became stronger within each), and the Karate Club was eventually split into two, which was what actually happened. In the meantime, we can also see that there were some cultural/ideological diffusion taking place across the two factions, because their colors became a little more gray-ish than at the beginning. So, there was apparently a competition not just between the two factions, but also between the two different dynamics: social assimilation and fragmentation. History has shown that the fragmentation won in the actual Karate Club case, but we can explore the parameter space of this model to see if there was any alternative future possible for the Karate Club!

Exercise 16.21 Examine, by simulations, the sensitivity of the Karate Club adaptive diffusion model on the initial node state assignments. Can a different node

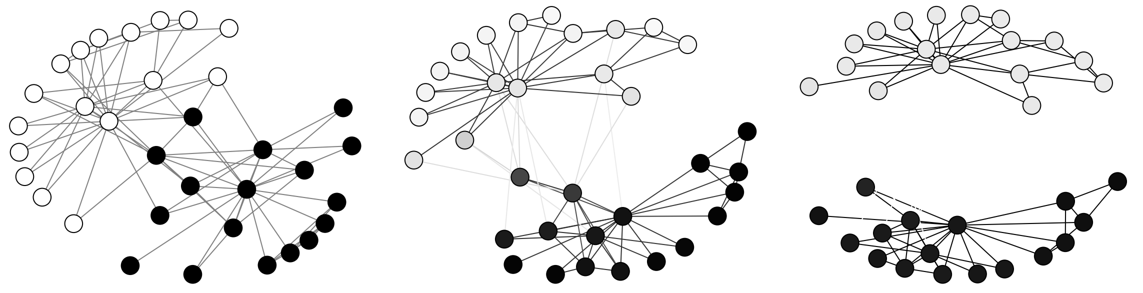


Figure 16.14: Visual output of Code 16.17. Time flows from left to right.

state assignment (with the same numbers of 0's and 1's) prevent the fission of the Club? If so, what are the effective strategies to assign the node states?

Exercise 16.22 Conduct simulations of the Karate Club adaptive diffusion model (with the original node state assignment) by systematically varying α, β, and γ, to find the parameter settings with which homogenization of the node states can be achieved. Then think about what kind of actions the members of the Karate Club could have taken to achieve those parameter values in a real-world setting, in order to avoid the fragmentation of the Club.

Chapter 17

Dynamical Networks II: Analysis of Network Topologies

17.1 Network Size, Density, and Percolation

Networks can be analyzed in several different ways. One way is to analyze their structural features, such as size, density, topology, and statistical properties.

Let me first begin with the most basic structural properties, i.e., the *size* and *density* of a network. These properties are conceptually similar to the mass and composition of matter—they just tell us how much stuff is in it, but they don't tell us anything about how the matter is organized internally. Nonetheless, they are still the most fundamental characteristics, which are particularly important when you want to compare multiple networks. You should compare properties of two networks of the same size and density, just like chemists who compare properties of gold and copper of the same mass.

The size of a network is characterized by the numbers of nodes and edges in it.

NetworkX's `Graph` objects have functions dedicated for measuring those properties:

Code 17.1:
```
>>> g = nx.karate_club_graph()
>>> g.number_of_nodes()
34
>>> g.number_of_edges()
78
```

The density of a network is the fraction between 0 and 1 that tells us what portion of all possible edges are actually realized in the network. For a network G made of n nodes and m edges, the density $\rho(G)$ is given by

$$\rho(G) = \frac{m}{\frac{n(n-1)}{2}} = \frac{2m}{n(n-1)} \tag{17.1}$$

for an undirected network, or

$$\rho(G) = \frac{m}{n(n-1)} \tag{17.2}$$

for a directed network.

NetworkX has a built-in function to calculate network density:

Code 17.2:

```
>>> g = nx.karate_club_graph()
>>> nx.density(g)
0.13903743315508021
```

Note that the size and density of a network don't specify much about the network's actual topology (i.e., shape). There are many networks with different topologies that have exactly the same size and density.

But there are some things the size and density can still predict about networks. One such example is *network percolation*, i.e., whether or not the nodes are sufficiently connected to each other so that they form a *giant component* that is visible at macroscopic scales. I can show you an example. In the code below, we generate Erdős-Rényi random graphs made of 100 nodes with different connection probabilities:

Code 17.3: net-percolation.py

```
from pylab import *
import networkx as nx

for i, p in [(1, 0.0001), (2, 0.001), (3, 0.01), (4, 0.1)]:
    subplot(1, 4, i)
    title('p = ' + str(p))
    g = nx.erdos_renyi_graph(100, p)
    nx.draw(g, node_size = 10)
```

```
show()
```

The result is shown in Fig. 17.1, where we can clearly see a transition from a completely disconnected set of nodes to a single giant network that includes all nodes.

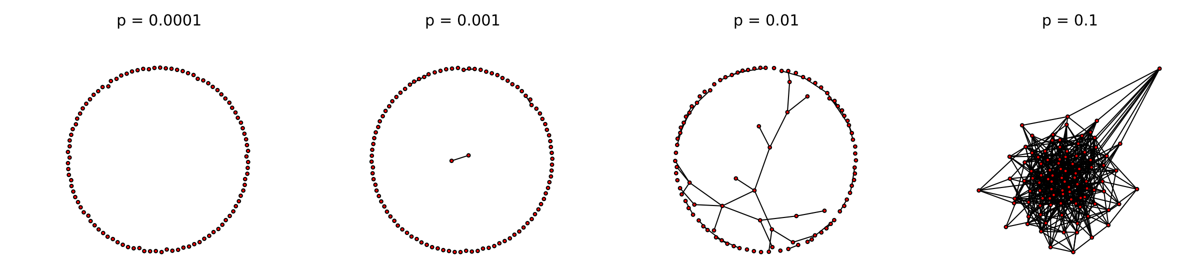


Figure 17.1: Visual output of Code 17.3.

The following code conducts a more systematic parameter sweep of the connection probability:

Code 17.4: net-percolation-plot.py

```
from pylab import *
import networkx as nx

p = 0.0001
pdata = []
gdata = []

while p < 0.1:
    pdata.append(p)
    g = nx.erdos_renyi_graph(100, p)
    ccs = nx.connected_components(g)
    gdata.append(max(len(cc) for cc in ccs))
    p *= 1.1

loglog(pdata, gdata)
xlabel('p')
ylabel('size of largest connected component')
show()
```

In this code, we measure the size of the largest connected component in an Erdős-Rényi graph with connection probability p, while geometrically increasing p. The result shown in Fig. 17.2 indicates that a percolation transition happened at around $p = 10^{-2}$.

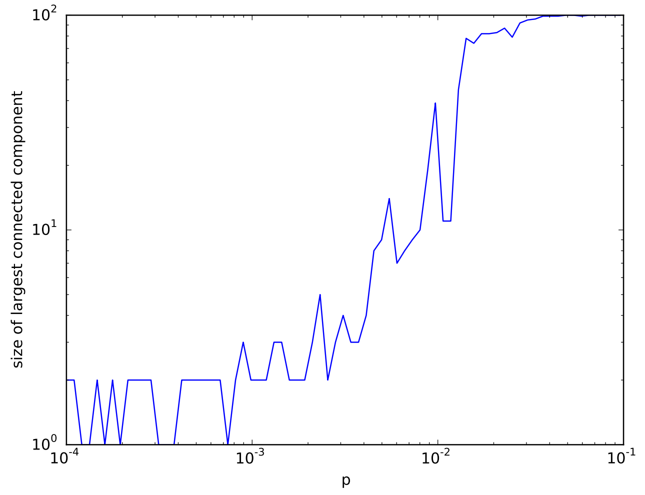


Figure 17.2: Visual output of Code 17.4.

Giant components are a special name given to the largest connected components that appear above this percolation threshold (they are seen in the third and fourth panels of Fig. 17.1), because their sizes are on the same order of magnitude as the size of the whole network. Mathematically speaking, they are defined as the connected components whose size $s(n)$ has the following property

$$\lim_{n \to \infty} \frac{s(n)}{n} = c > 0, \tag{17.3}$$

where n is the number of nodes. This limit would go to zero for all other non-giant components, so at macroscopic scales, we can only see giant components in large networks.

Exercise 17.1 Revise Code 17.4 so that you generate multiple random graphs for each p, and calculate the average size of the largest connected components. Then run the revised code for larger network sizes (say, 1,000) to obtain a smoother curve.

We can calculate the network percolation threshold for random graphs as follows. Let q be the probability for a randomly selected node to *not* belong to the largest connected component (LCC) of a network. If $q < 1$, then there is a giant component in the network. In order for a randomly selected node to be disconnected from the LCC, all of its neighbors must be disconnected from the LCC too. Therefore, somewhat tautologically

$$q = q^k, \tag{17.4}$$

where k is the degree of the node in question. But in general, degree k is not uniquely determined in a random network, so the right hand side should be rewritten as an expected value, as

$$q = \sum_{k=0}^{\infty} P(k) q^k, \tag{17.5}$$

where $P(k)$ is the probability for the node to have degree k (called the *degree distribution*; this will be discussed in more detail later). In an Erdős-Rényi random graph, this probability can be calculated by the following binomial distribution

$$P(k) = \begin{cases} \binom{n-1}{k} p^k (1-p)^{n-1-k} & \text{for } 0 \le k \le n-1, \\ 0 & \text{for } k \ge n, \end{cases} \tag{17.6}$$

because each node has $n-1$ potential neighbors and k out of $n-1$ neighbors need to be connected to the node (and the rest need to be disconnected from it) in order for it to have degree k. By plugging Eq. (17.6) into Eq. (17.5), we obtain

$$\begin{aligned} q &= \sum_{k=0}^{n-1} \binom{n-1}{k} p^k (1-p)^{n-1-k} q^k && (17.7) \\ &= \sum_{k=0}^{n-1} \binom{n-1}{k} (pq)^k (1-p)^{n-1-k} && (17.8) \\ &= (pq + 1 - p)^{n-1} && (17.9) \\ &= (1 + p(q-1))^{n-1}. && (17.10) \end{aligned}$$

We can rewrite this as

$$q = \left(1 + \frac{\langle k \rangle (q-1)}{n}\right)^{n-1}, \tag{17.11}$$

because the average degree $\langle k \rangle$ of an Erdős-Rényi graph for a large n is given by

$$\langle k \rangle = np. \tag{17.12}$$

Since $\lim_{n\to\infty}(1+x/n)^n = e^x$, Eq. (17.11) can be further simplified for large n to

$$q = e^{\langle k\rangle(q-1)}. \tag{17.13}$$

Apparently, $q = 1$ satisfies this equation. If this equation also has another solution in $0 < q < 1$, then that means a giant component is possible in the network. Figure 17.3 shows the plots of $y = q$ and $y = e^{\langle k\rangle(q-1)}$ for $0 < q < 1$ for several values of $\langle k\rangle$.

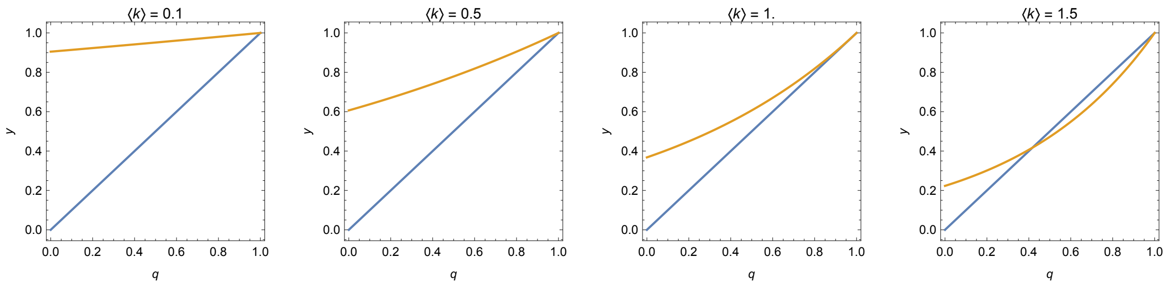


Figure 17.3: Plots of $y = q$ and $y = e^{\langle k\rangle(q-1)}$ for several values of $\langle k\rangle$.

These plots indicate that if the derivative of the right hand side of Eq. (17.13) (i.e., the slope of the curve) at $q = 1$ is greater than 1, then the equation has a solution in $q < 1$. Therefore,

$$\frac{d}{dq}e^{\langle k\rangle(q-1)}\bigg|_{q=1} = \langle k\rangle e^{\langle k\rangle(q-1)}|_{q=1} = \langle k\rangle > 1, \tag{17.14}$$

i.e., if the average degree is greater than 1, then network percolation occurs.

A *giant component* is a connected component whose size is on the same order of magnitude as the size of the whole network. *Network percolation* is the appearance of such a giant component in a random graph, which occurs when the average node degree is above 1.

Exercise 17.2 If Eq. (17.13) has a solution in $q < 1/n$, that means that all the nodes are essentially included in the giant component, and thus the network is made of a single connected component. Obtain the threshold of $\langle k\rangle$ above which this occurs.

17.2 Shortest Path Length

Network analysis can measure and characterize various features of network topologies that go beyond size and density. Many of the tools used here are actually borrowed from social network analysis developed and used in sociology [60].

The first measurement we are going to discuss is the *shortest path length* from one node to another node, also called *geodesic distance* in a graph. If the network is undirected, the distance between two nodes is the same, regardless of which node is the starting point and which is the end point. But if the network is directed, this is no longer the case in general.

The shortest path length is easily measurable using NetworkX:

Code 17.5:

```
>>> g = nx.karate_club_graph()
>>> nx.shortest_path_length(g, 16, 25)
4
```

The actual path can also be obtained as follows:

Code 17.6:

```
>>> nx.shortest_path(g, 16, 25)
[16, 5, 0, 31, 25]
```

The output above is a list of nodes on the shortest path from node 16 to node 25. This can be visualized using `draw_networkx_edges` as follows:

Code 17.7: shortest-path.py

```
from pylab import *
import networkx as nx

g = nx.karate_club_graph()
positions = nx.spring_layout(g)

path = nx.shortest_path(g, 16, 25)
edges = [(path[i], path[i+1]) for i in xrange(len(path) - 1)]

nx.draw_networkx_edges(g, positions, edgelist = edges,
                       edge_color = ’r’, width = 10)
```

```
nx.draw(g, positions, with_labels = True)
show()
```

The result is shown in Fig. 17.4.

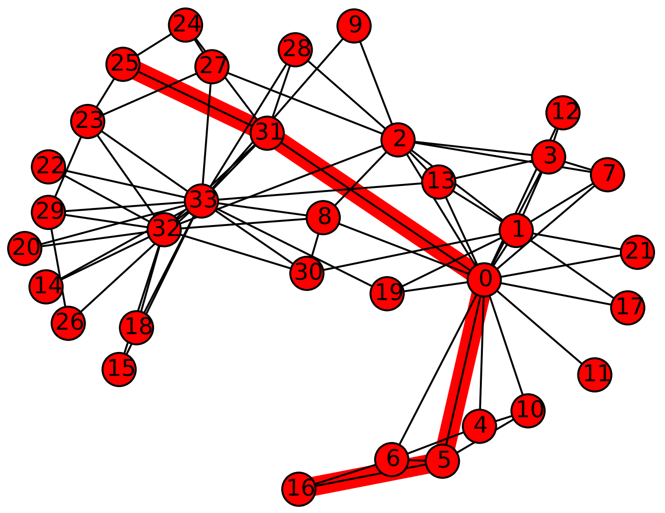


Figure 17.4: Visual output of Code 17.7.

We can use this shortest path length to define several useful metrics to characterize the network's topological properties. Let's denote a shortest path length from node i to node j as $d(i \to j)$. Clearly, $d(i \to i) = 0$ for all i. Then we can construct the following metrics:

Characteristic path length

$$L = \frac{\sum_{i,j} d(i \to j)}{n(n-1)} \tag{17.15}$$

where n is the number of nodes. This formula works for both undirected and directed networks. It calculates the average length of shortest paths for all possible node pairs in the network, giving an expected distance between two randomly chosen nodes. This is an intuitive characterization of how big (or small) the world represented by the network is.

Eccentricity

$$\varepsilon(i) = \max_j d(i \to j) \tag{17.16}$$

This metric is defined for each node and gives the maximal shortest path length a node can have with any other node in the network. This tells how far the node is to the farthest point in the network.

Diameter

$$D = \max_i \varepsilon(i) \tag{17.17}$$

This metric gives the maximal eccentricity in the network. Intuitively, it tells us how far any two nodes can get from one another within the network. Nodes whose eccentricity is D are called *peripheries.*

Radius

$$R = \min_i \varepsilon(i) \tag{17.18}$$

This metric gives the minimal eccentricity in the network. Intuitively, it tells us the smallest number of steps you will need to reach every node if you can choose an optimal node as a starting point. Nodes whose eccentricity is R are called *centers.*

In NetworkX, these metrics can be calculated as follows:

Code 17.8:

```
>>> import networkx as nx
>>> g = nx.karate_club_graph()
>>> nx.average_shortest_path_length(g)
2.408199643493761
>>> nx.eccentricity(g)
{0: 3, 1: 3, 2: 3, 3: 3, 4: 4, 5: 4, 6: 4, 7: 4, 8: 3, 9: 4, 10: 4,
11: 4, 12: 4, 13: 3, 14: 5, 15: 5, 16: 5, 17: 4, 18: 5, 19: 3, 20: 5,
21: 4, 22: 5, 23: 5, 24: 4, 25: 4, 26: 5, 27: 4, 28: 4, 29: 5, 30: 4,
31: 3, 32: 4, 33: 4}
>>> nx.diameter(g)
5
>>> nx.periphery(g)
```

```
[14, 15, 16, 18, 20, 22, 23, 26, 29]
>>> nx.radius(g)
3
>>> nx.center(g)
[0, 1, 2, 3, 8, 13, 19, 31]
```

Exercise 17.3 Visualize the Karate Club graph using the eccentricities of its nodes to color them.

Exercise 17.4 Randomize the topology of the Karate Club graph while keeping its size and density, and then measure the characteristic path length, diameter, and radius of the randomized graph. Repeat this many times to obtain distributions of those metrics for randomized graphs. Then compare the distributions with the metrics of the original Karate Club graph. Discuss which metric is more sensitive to topological variations.

Note: Randomized graphs may be disconnected. If so, all of the metrics discussed above will be infinite, and NetworkX will give you an error. In order to avoid this, you should check whether the randomized network is connected or not before calculating its metrics. NetworkX has a function `nx.is_connected` for this purpose.

17.3 Centralities and Coreness

The eccentricity of nodes discussed above can be used to detect which nodes are most central in a network. This can be useful because, for example, if you send out a message from one of the center nodes with minimal eccentricity, the message will reach every single node in the network in the shortest period of time.

In the meantime, there are several other more statistically based measurements of node centralities, each of which has some benefits, depending on the question you want to study. Here is a summary of some of those centrality measures:

Degree centrality

$$c_D(i) = \frac{\deg(i)}{n-1} \tag{17.19}$$

Degree centrality is simply a normalized node degree, i.e., the actual degree divided by the maximal degree possible $(n-1)$. For directed networks, you can define *in-degree centrality* and *out-degree centrality* separately.

Betweenness centrality

$$c_B(i) = \frac{1}{(n-1)(n-2)} \sum_{j \neq i, k \neq i, j \neq k} \frac{N_{\mathrm{sp}}(j \xrightarrow{i} k)}{N_{\mathrm{sp}}(j \to k)} \tag{17.20}$$

where $N_{\mathrm{sp}}(j \to k)$ is the number of shortest paths from node j to node k, and $N_{\mathrm{sp}}(j \xrightarrow{i} k)$ is the number of the shortest paths from node j to node k that go through node i. Betweenness centrality of a node is the probability for the shortest path between two randomly chosen nodes to go through that node. This metric can also be defined for edges in a similar way, which is called *edge betweenness*.

Closeness centrality

$$c_C(i) = \left(\frac{\sum_j d(i \to j)}{n-1} \right)^{-1} \tag{17.21}$$

This is an inverse of the average distance from node i to all other nodes. If $c_C(i) = 1$, that means you can reach any other node from node i in just one step. For directed networks, you can also define another closeness centrality by swapping i and j in the formula above to measure how accessible node i is *from* other nodes.

Eigenvector centrality

$$c_E(i) = v_i \quad \text{(i-th element of the dominant eigenvector v of the network's adjacency matrix)} \tag{17.22}$$

Eigenvector centrality measures the "importance" of each node by considering each incoming edge to the node an "endorsement" from its neighbor. This differs from degree centrality because, in the calculation of eigenvector centrality, endorsements coming from more important nodes count as more. Another completely different, but mathematically equivalent, interpretation of eigenvector centrality is that it counts the number of walks from any node in the network that reach node i in t steps, with t taken to infinity. Eigenvector v is usually chosen to be a non-negative unit vector ($v_i \geq 0$, $|v| = 1$).

PageRank

$$c_P(i) = v_i \quad \text{(i-th element of the dominant eigenvector v of the following transition probability matrix)} \tag{17.23}$$

$$T = \alpha A D^{-1} + (1-\alpha)\frac{J}{n} \tag{17.24}$$

where A is the adjacency matrix of the network, D^{-1} is a diagonal matrix whose i-th diagonal component is $1/\deg(i)$, J is an $n \times n$ all-one matrix, and α is the damping parameter ($\alpha = 0.85$ is commonly used by default).

PageRank is a variation of eigenvector centrality that was originally developed by Larry Page and Sergey Brin, the founders of Google, in the late 1990s [74, 75] to rank web pages. PageRank measures the asymptotic probability for a random walker on the network to be standing on node i, assuming that the walker moves to a randomly chosen neighbor with probability α, or jumps to any node in the network with probability $1-\alpha$, in each time step. Eigenvector v is usually chosen to be a probability distribution, i.e., $\sum_i v_i = 1$.

Note that all of these centrality measures give a normalized value between 0 and 1, where 1 means perfectly central while 0 means completely peripheral. In most cases, the values are somewhere in between. Functions to calculate these centrality measures are also available in NetworkX (outputs are omitted in the code below to save space):

Code 17.9:

```
>>> import networkx as nx
>>> g = nx.karate_club_graph()
>>> nx.degree_centrality(g)
>>> nx.betweenness_centrality(g)
>>> nx.closeness_centrality(g)
>>> nx.eigenvector_centrality(g)
>>> nx.pagerank(g)
```

While those centrality measures are often correlated, they capture different properties of each node. Which centrality should be used depends on the problem you are addressing. For example, if you just want to find the most popular person in a social network, you can just use degree centrality. But if you want to find the most efficient person to disseminate a rumor to the entire network, closeness centrality would probably be a more appropriate metric to use. Or, if you want to find the most effective person to monitor and manipulate information flowing within the network, betweenness centrality would be more appropriate, assuming that information travels through the shortest paths between people. Eigenvector centrality and PageRank are useful for generating a reasonable ranking of nodes in a complex network made of directed edges.

Exercise 17.5 Visualize the Karate Club graph using each of the above centrality measures to color the nodes, and then compare the visualizations to see how those centralities are correlated.

Exercise 17.6 Generate (1) an Erdős-Rényi random network, (2) a Watts-Strogatz small-world network, and (3) a Barabási-Albert scale-free network of comparable size and density, and obtain the distribution of node centralities of your choice for each network. Then compare those centrality distributions to find which one is most/least heterogeneous.

Exercise 17.7 Prove that PageRank with $\alpha = 1$ is essentially the same as the degree centrality for undirected networks.

A little different approach to characterize the centrality of nodes is to calculate their *coreness*. This can be achieved by the following simple algorithm:

1. Let $k = 0$.
2. Repeatedly delete all nodes whose degree is k or less, until no such nodes exist. Those removed nodes are given a coreness k.
3. If there are still nodes remaining in the network, increase k by 1, and then go back to the previous step.

Figure 17.5 shows an example of this calculation. At the very end of the process, we see a cluster of nodes whose degrees are at least the final value of k. This cluster is called a k-core, which can be considered the central part of the network.

In NetworkX, you can calculate the corenesses of nodes using the `core_number` function. Also, the k-core of the network can be obtained by using the `k_core` function. Here is an example:

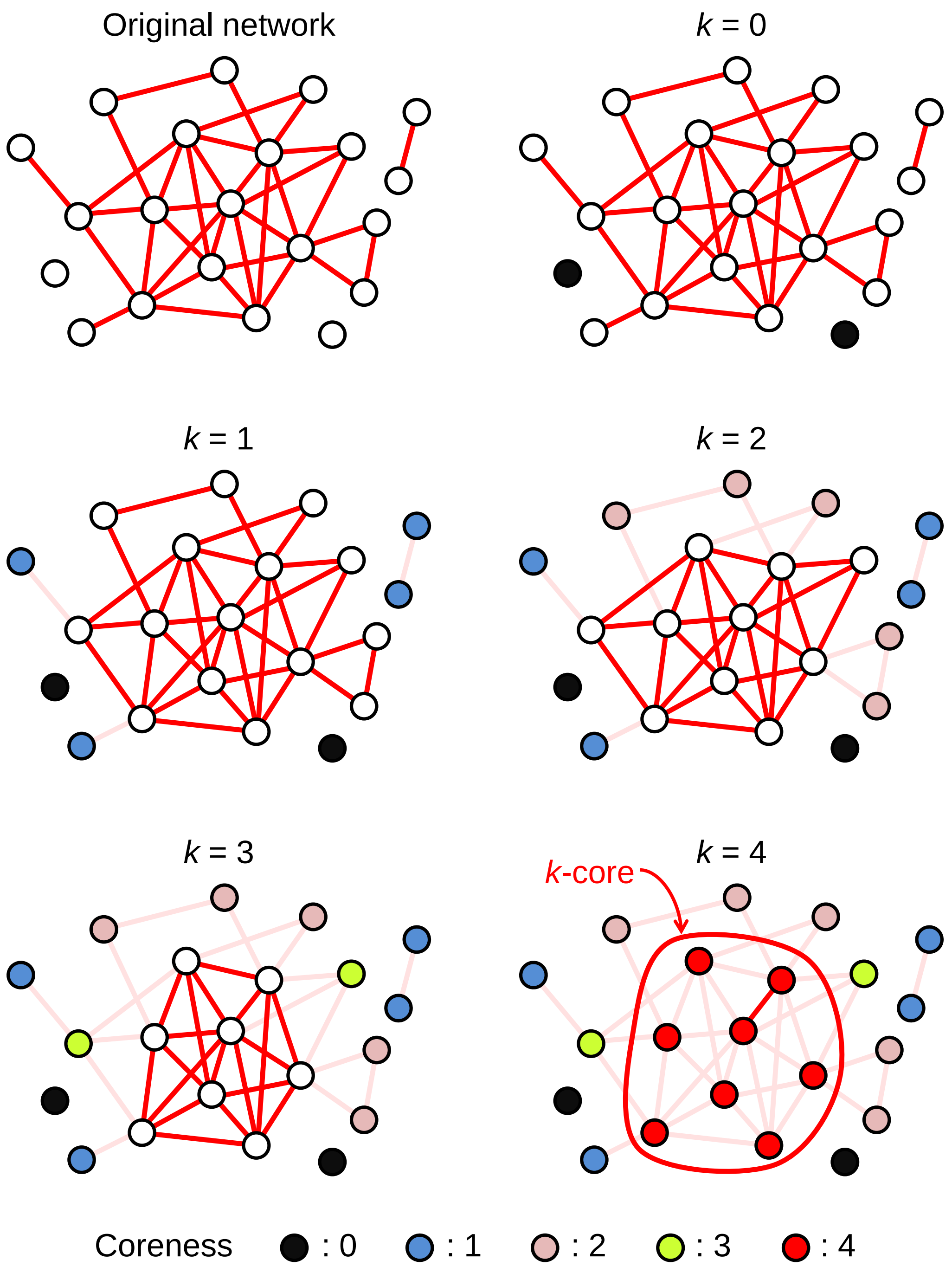


Figure 17.5: Example of how coreness is calculated.

Code 17.10:

```
>>> from pylab import *
>>> import networkx as nx
>>> g = nx.karate_club_graph()
>>> nx.core_number(g)
{0: 4, 1: 4, 2: 4, 3: 4, 4: 3, 5: 3, 6: 3, 7: 4, 8: 4, 9: 2, 10: 3,
  11: 1, 12: 2, 13: 4, 14: 2, 15: 2, 16: 2, 17: 2, 18: 2, 19: 3,
  20: 2, 21: 2, 22: 2, 23: 3, 24: 3, 25: 3, 26: 2, 27: 3, 28: 3,
  29: 3, 30: 4, 31: 3, 32: 4, 33: 4}
>>> nx.draw(nx.k_core(g), with_labels = True)
>>> show()
```

The resulting k-core of the Karate Club graph is shown in Fig. 17.6.

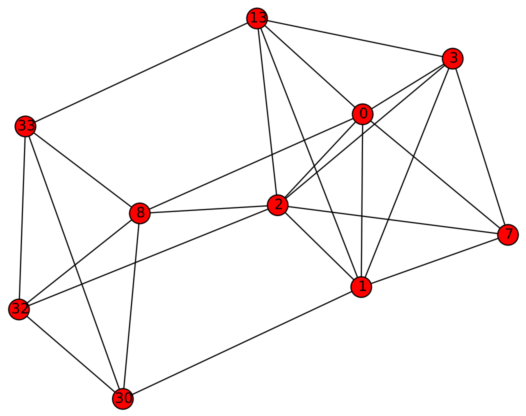


Figure 17.6: Visual output of Code 17.10, showing the k-core of the Karate Club graph, with $k = 4$.

One advantage of using coreness over other centrality measures is its scalability. Because the algorithm to compute it is so simple, the calculation of coreness of nodes in a very large network is much faster than other centrality measures (except for degree centrality, which is obviously very fast too).

Exercise 17.8 Import a large network data set of your choice from Mark Newman's Network Data website: `http://www-personal.umich.edu/~mejn/netdata/`

Calculate the coreness of all of its nodes and draw their histogram. Compare the speed of calculation with, say, the calculation of betweenness centrality. Also visualize the k-core of the network.

17.4 Clustering

Eccentricity, centralities, and coreness introduced above all depend on the whole network topology (except for degree centrality). In this sense, they capture some macroscopic aspects of the network, even though we are calculating those metrics for each node. In contrast, there are other kinds of metrics that only capture local topological properties. This includes metrics of *clustering*, i.e., how densely connected the nodes are to each other in a localized area in a network. There are two widely used metrics for this:

Clustering coefficient

$$C(i) = \frac{\left|\{\ \{j,k\} \mid d(i,j) = d(i,k) = d(j,k) = 1\ \}\right|}{\deg(i)(\deg(i)-1)/2} \tag{17.25}$$

The denominator is the total number of possible node pairs within node i's neighborhood, while the numerator is the number of actually connected node pairs among them. Therefore, the clustering coefficient of node i calculates the probability for its neighbors to be each other's neighbors as well. Note that this metric assumes that the network is undirected. The following *average clustering coefficient* is often used to measure the level of clustering in the entire network:

$$C = \frac{\sum_i C(i)}{n} \tag{17.26}$$

Transitivity

$$C_T = \frac{\left|\{\ (i,j,k) \mid d(i,j) = d(i,k) = d(j,k) = 1\ \}\right|}{\left|\{\ (i,j,k) \mid d(i,j) = d(i,k) = 1\ \}\right|} \tag{17.27}$$

This is very similar to clustering coefficients, but it is defined by counting connected node triplets over the entire network. The denominator is the number of connected node triplets (i.e., a node, i, and two of its neighbors, j and k), while the numerator is the number of such triplets where j is also connected to k. This essentially captures the same aspect of the network as the average clustering coefficient, i.e., how locally clustered the network is, but the transitivity can be calculated on directed networks too. It also treats each triangle more evenly, unlike the average clustering coefficient that tends to underestimate the contribution of triplets that involve highly connected nodes.

Again, calculating these clustering metrics is very easy in NetworkX:

Code 17.11:

```
>>> import networkx as nx
>>> g = nx.karate_club_graph()
>>> nx.clustering(g)
{0: 0.15, 1: 0.3333333333333333, 2: 0.24444444444444444, 3:
  0.6666666666666666, 4: 0.6666666666666666, 5: 0.5, 6: 0.5, 7: 1.0,
  8: 0.5, 9: 0.0, 10: 0.6666666666666666, 11: 0.0, 12: 1.0, 13: 0.6,
  14: 1.0, 15: 1.0, 16: 1.0, 17: 1.0, 18: 1.0, 19: 0.3333333333333333,
  20: 1.0, 21: 1.0, 22: 1.0, 23: 0.4, 24: 0.3333333333333333, 25:
  0.3333333333333333, 26: 1.0, 27: 0.16666666666666666, 28:
  0.3333333333333333, 29: 0.6666666666666666, 30: 0.5, 31: 0.2, 32:
  0.19696969696969696, 33: 0.11029411764705882}
>>> nx.average_clustering(g)
0.5706384782076823
>>> nx.transitivity(g)
0.2556818181818182
```

Exercise 17.9 Generate (1) an Erdős-Rényi random network, (2) a Watts-Strogatz small-world network, and (3) a Barabási-Albert scale-free network of comparable size and density, and compare them with regard to how locally clustered they are.

The clustering coefficient was first introduced by Watts and Strogatz [56], where they showed that their small-world networks tend to have very high clustering compared to their random counterparts. The following code replicates their computational experiment, varying the rewiring probability `p`:

Code 17.12: small-world-experiment.py

```
from pylab import *
import networkx as nx

pdata = []
Ldata = []
Cdata = []

g0 = nx.watts_strogatz_graph(1000, 10, 0)
L0 = nx.average_shortest_path_length(g0)
C0 = nx.average_clustering(g0)

p = 0.0001
while p < 1.0:
    g = nx.watts_strogatz_graph(1000, 10, p)
    pdata.append(p)
    Ldata.append(nx.average_shortest_path_length(g) / L0)
    Cdata.append(nx.average_clustering(g) / C0)
    p *= 1.5

semilogx(pdata, Ldata, label = 'L / L0')
semilogx(pdata, Cdata, label = 'C / C0')
xlabel('p')
legend()
show()
```

The result is shown in Fig. 17.7, where the characteristic path length (`L`) and the average clustering coefficient (`C`) are plotted as their fractions to the baseline values (`L0`, `C0`) obtained from a purely regular network `g0`. As you can see in the figure, the network becomes very small (low `L`) yet remains highly clustered (high `C`) at the intermediate value of `p` around 10^{-2}. This is the parameter regime where the Watts-Strogatz small-world networks arise.

Exercise 17.10 Revise Code 17.12 so that you generate multiple Watts-Strogatz networks for each p and calculate the averages of characteristic path lengths and average clustering coefficients. Then run the revised code to obtain a smoother curve.

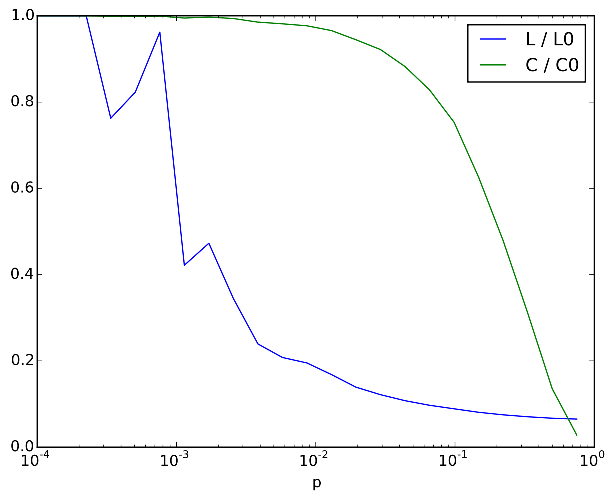


Figure 17.7: Visual output of Code 17.12.

17.5 Degree Distribution

Another local topological property that can be measured locally is, as we discussed already, the degree of a node. But if we collect them all for the whole network and represent them as a distribution, it will give us another important piece of information about how the network is structured:

A *degree distribution* of a network is a probability distribution

$$P(k) = \frac{\left|\{\, i \mid \deg(i) = k \}\right|}{n}, \tag{17.28}$$

i.e., the probability for a node to have degree k.

The degree distribution of a network can be obtained and visualized as follows:

Code 17.13:
```
>>> from pylab import *
>>> import networkx as nx
>>> g = nx.karate_club_graph()
>>> hist(g.degree().values(), bins = 20)
```

```
(array([ 1., 11., 6., 6., 0., 3., 2., 0., 0., 0., 1., 1., 0., 1., 0.,
  0., 0., 0., 1., 1.]), array([ 1.  , 1.8, 2.6, 3.4, 4.2, 5.  , 5.8,
  6.6, 7.4, 8.2, 9.  , 9.8, 10.6, 11.4, 12.2, 13.  , 13.8, 14.6, 15.4,
  16.2, 17. ]), <a list of 20 Patch objects>)
>>> show()
```

The result is shown in Fig. 17.8.

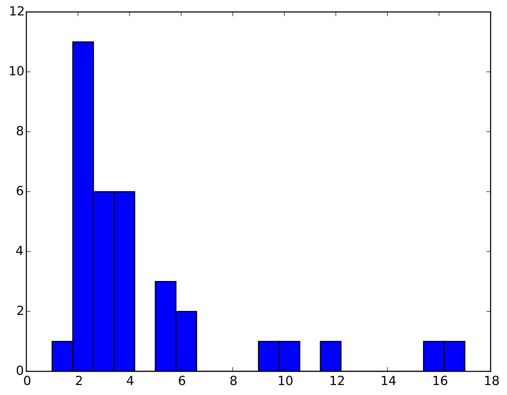


Figure 17.8: Visual output of Code 17.13.

You can also obtain the actual degree distribution $P(k)$ as follows:

Code 17.14:

```
>>> nx.degree_histogram(g)
[0, 1, 11, 6, 6, 3, 2, 0, 0, 1, 1, 0, 1, 0, 0, 0, 1, 1]
```

This list contains the value of (unnormalized) $P(k)$ for $k = 0, 1, \ldots, k_{\max}$, in this order. For larger networks, it is often more useful to plot a normalized degree histogram list in a log-log scale:

Code 17.15: degree-distributions-loglog.py

```
from pylab import *
import networkx as nx

n = 1000
```

```
er = nx.erdos_renyi_graph(n, 0.01)
ws = nx.watts_strogatz_graph(n, 10, 0.01)
ba = nx.barabasi_albert_graph(n, 5)

Pk = [float(x) / n for x in nx.degree_histogram(er)]
domain = range(len(Pk))
loglog(domain, Pk, '-', label = 'Erdos-Renyi')

Pk = [float(x) / n for x in nx.degree_histogram(ws)]
domain = range(len(Pk))
loglog(domain, Pk, '--', label = 'Watts-Strogatz')

Pk = [float(x) / n for x in nx.degree_histogram(ba)]
domain = range(len(Pk))
loglog(domain, Pk, ':', label = 'Barabasi-Albert')

xlabel('k')
ylabel('P(k)')
legend()
show()
```

The result is shown in Fig. 17.9, which clearly illustrates differences between the three network models used in this example. The Erdős-Rényi random network model has a bell-curved degree distribution, which appears as a skewed mountain in the log-log scale (blue solid line). The Watts-Strogatz model is nearly regular, and thus it has a very sharp peak at the average degree (green dashed line; $k = 10$ in this case). The Barabási-Albert model has a power-law degree distribution, which looks like a straight line with a negative slope in the log-log scale (red dotted line).

Moreover, it is often visually more meaningful to plot *not* the degree distribution itself but its *complementary cumulative distribution function (CCDF)*, defined as follows:

$$F(k) = \sum_{k'=k}^{\infty} P(k') \tag{17.29}$$

This is a probability for a node to have a degree k or higher. By definition, $F(0) = 1$ and $F(k_{\max} + 1) = 0$, and the function decreases monotonically along k. We can revise Code 17.15 to draw CCDFs:

Code 17.16: ccdfs-loglog.py

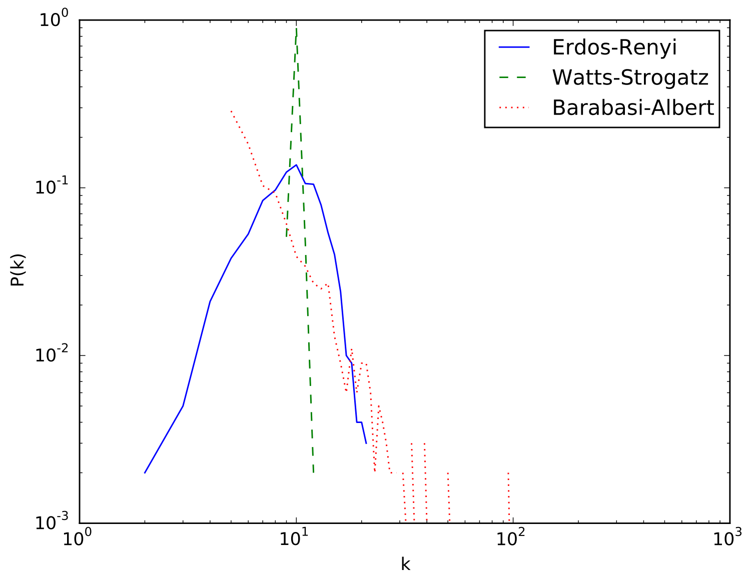


Figure 17.9: Visual output of Code 17.15.

```
from pylab import *
import networkx as nx

n = 1000
er = nx.erdos_renyi_graph(n, 0.01)
ws = nx.watts_strogatz_graph(n, 10, 0.01)
ba = nx.barabasi_albert_graph(n, 5)

Pk = [float(x) / n for x in nx.degree_histogram(er)]
domain = range(len(Pk))
ccdf = [sum(Pk[k:]) for k in domain]
loglog(domain, ccdf, '-', label = 'Erdos-Renyi')

Pk = [float(x) / n for x in nx.degree_histogram(ws)]
domain = range(len(Pk))
ccdf = [sum(Pk[k:]) for k in domain]
loglog(domain, ccdf, '--', label = 'Watts-Strogatz')
```

```
Pk = [float(x) / n for x in nx.degree_histogram(ba)]
domain = range(len(Pk))
ccdf = [sum(Pk[k:]) for k in domain]
loglog(domain, ccdf, ':', label = 'Barabasi-Albert')

xlabel('k')
ylabel('F(k)')
legend()
show()
```

In this code, we generate `ccdf`'s from `Pk` by calculating the sum of `Pk` after dropping its first `k` entries. The result is shown in Fig. 17.10.

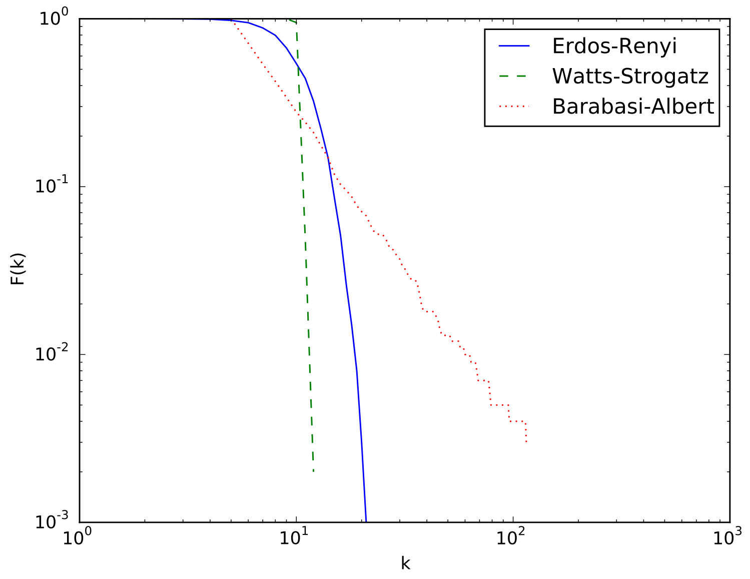


Figure 17.10: Visual output of Code 17.16.

As you can see in the figure, the power law degree distribution remains as a straight line in the CCDF plot too, because $F(k)$ will still be a power function of k, as shown below:

$$F(k) = \sum_{k'=k}^{\infty} P(k') = \sum_{k'=k}^{\infty} ak'^{-\gamma} \tag{17.30}$$

$$\approx \int_k^{\infty} ak'^{-\gamma} dk' = \left[\frac{ak'^{-\gamma+1}}{-\gamma+1}\right]_k^{\infty} = \frac{0 - ak^{-\gamma+1}}{-\gamma+1} \tag{17.31}$$

$$= \frac{a}{\gamma-1} k^{-(\gamma-1)} \tag{17.32}$$

This result shows that the scaling exponent of $F(k)$ for a power law degree distribution is less than that of the original distribution by 1, which can be visually seen by comparing their slopes between Figs. 17.9 and 17.10.

Exercise 17.11 Import a large network data set of your choice from Mark Newman's Network Data website: `http://www-personal.umich.edu/~mejn/netdata/` Plot the degree distribution of the network, as well as its CCDF. Determine whether the network is more similar to a random, a small-world, or a scale-free network model.

If the network's degree distribution shows a power law behavior, you can estimate its scaling exponent from the distribution by simple linear regression. You should use a CCDF of the degree distribution for this purpose, because CCDFs are less noisy than the original degree distributions. Here is an example of scaling exponent estimation applied to a Barabási-Albert network, where the `linregress` function in SciPy's `stats` module is used for linear regression:

Code 17.17: exponent-estimation.py

```
from pylab import *
import networkx as nx
from scipy import stats as st

n = 10000
ba = nx.barabasi_albert_graph(n, 5)
Pk = [float(x) / n for x in nx.degree_histogram(ba)]
domain = range(len(Pk))
ccdf = [sum(Pk[k:]) for k in domain]
```

```
logkdata = []
logFdata = []
prevF = ccdf[0]
for k in domain:
    F = ccdf[k]
    if F != prevF:
        logkdata.append(log(k))
        logFdata.append(log(F))
        prevF = F

a, b, r, p, err = st.linregress(logkdata, logFdata)
print 'Estimated CCDF: F(k) =', exp(b), '* k^', a
print 'r =', r
print 'p-value =', p

plot(logkdata, logFdata, 'o')
kmin, kmax = xlim()
plot([kmin, kmax],[a * kmin + b, a * kmax + b])
xlabel('log k')
ylabel('log F(k)')
show()
```

In the second code block, the `domain` and `ccdf` were converted to log scales for linear fitting. Also, note that the original `ccdf` contained values for all `k`'s, even for those for which $P(k) = 0$. This would cause unnecessary biases in the linear regression toward the higher `k` end where actual samples were very sparse. To avoid this, only the data points where the value of F changed (i.e., where there were actual nodes with degree `k`) are collected in the `logkdata` and `logFdata` lists.

The result is shown in Fig. 17.11, and also the following output comes out to the terminal, which indicates that this was a pretty good fit:

Code 17.18:

```
Estimated CCDF: F(k) = 27.7963518947 * k^ -1.97465957944
r = -0.997324657337
p-value = 8.16476416778e-127
```

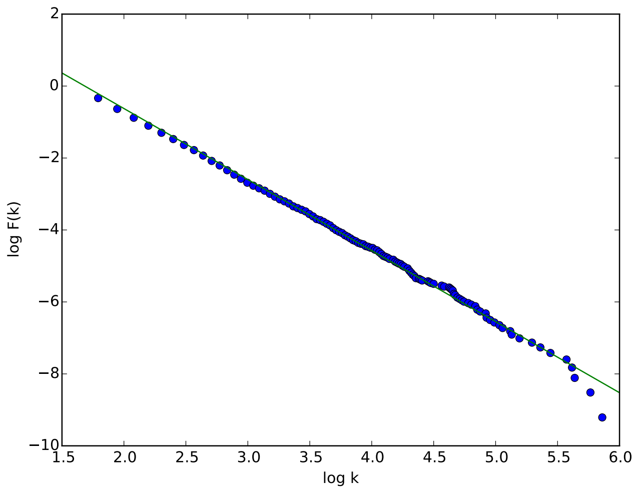


Figure 17.11: Visual output of Code 17.17.

According to this result, the CCDF had a negative exponent of about -1.97. Since this value corresponds to $-(\gamma-1)$, the actual scaling exponent γ is about 2.97, which is pretty close to its theoretical value, 3.

Exercise 17.12 Obtain a large network data set whose degree distribution appears to follow a power law, from any source (there are tons available online, including Mark Newman's that was introduced before). Then estimate its scaling exponent using linear regression.

17.6 Assortativity

Degrees are a metric measured on individual nodes. But when we focus on the edges, there are always two degrees associated with each edge, one for the node where the edge originates and the other for the node to where the edge points. So if we take the former for x and the latter for y from all the edges in the network, we can produce a scatter plot that visualizes a possible *degree correlation* between the nodes across the edges. Such correlations of node properties across edges can be generally described with the concept of *assortativity*:

Assortativity (positive assortativity) The tendency for nodes to connect to other nodes with similar properties within a network.

Disassortativity (negative assortativity) The tendency for nodes to connect to other nodes with *dissimilar* properties within a network.

Assortativity coefficient

$$r = \frac{\sum_{(i,j)\in E} (f(i) - \bar{f}_1)(f(j) - \bar{f}_2)}{\sqrt{\sum_{(i,j)\in E} (f(i) - \bar{f}_1)^2}\sqrt{\sum_{(i,j)\in E} (f(j) - \bar{f}_2)^2}} \tag{17.33}$$

where E is the set of directed edges (undirected edges should appear twice in E in two directions), and

$$\bar{f}_1 = \frac{\sum_{(i,j)\in E} f(i)}{|E|}, \qquad \bar{f}_2 = \frac{\sum_{(i,j)\in E} f(j)}{|E|}. \tag{17.34}$$

The assortativity coefficient is a Pearson correlation coefficient of some node property f between pairs of connected nodes. Positive coefficients imply assortativity, while negative ones imply disassortativity.

If the measured property is a node degree (i.e., $f = \mathrm{deg}$), this is called the *degree assortativity coefficient*. For directed networks, each of $\bar{f}_1$ and $\bar{f}_2$ can be either in-degree or out-degree, so there are four different degree assortativities you can measure: *in-in, in-out, out-in, and out-out.*

Let's look at some example. Here is how to draw a degree-degree scatter plot:

Code 17.19: degree-correlation.py

```
from pylab import *
import networkx as nx

n = 1000
ba = nx.barabasi_albert_graph(n, 5)

xdata = []
ydata = []
for i, j in ba.edges_iter():
    xdata.append(ba.degree(i)); ydata.append(ba.degree(j))
    xdata.append(ba.degree(j)); ydata.append(ba.degree(i))

plot(xdata, ydata, 'o', alpha = 0.05)
show()
```

In this example, we draw a degree-degree scatter plot for a Barabási-Albert network with 1,000 nodes. For each edge, the degrees of its two ends are stored in `xdata` and `ydata` twice in different orders, because an undirected edge can be counted in two directions. The markers in the plot are made transparent using the `alpha` option so that we can see the density variations in the plot.

The result is shown in Fig. 17.12, where each dot represents one directed edge in the network (so, an undirected edge is represented by two dots symmetrically placed across a diagonal mirror line). It can be seen that most edges connect low-degree nodes to each other, with some edges connecting low-degree and high-degree nodes, but it is quite rare that high-degree nodes are connected to each other. Therefore, there is a mild negative degree correlation in this case.

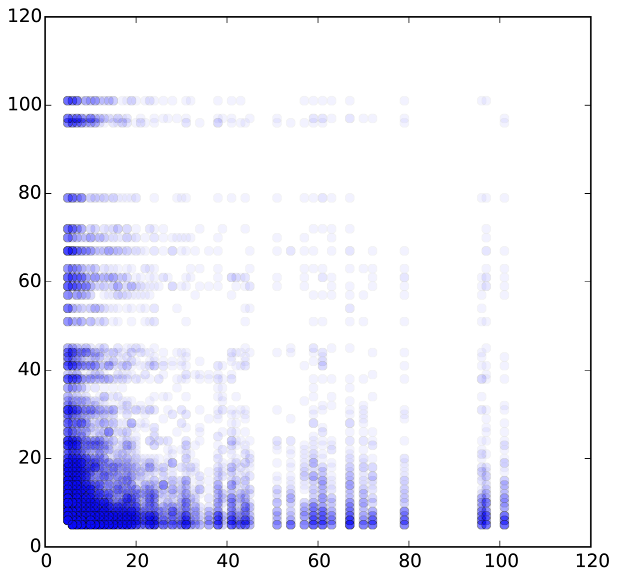


Figure 17.12: Visual output of Code 17.19.

We can confirm this observation by calculating the degree assortativity coefficient as follows:

Code 17.20:

```
>>> nx.degree_assortativity_coefficient(ba)
-0.058427968508962938
```

This function also has options `x` (for $\bar{f}_1$) and `y` (for $\bar{f}_2$) to specify which degrees you use in calculating coefficients for directed networks:

Code 17.21:

```
>>> import networkx as nx
>>> g = nx.DiGraph()
>>> g.add_edges_from([(0,1), (0,2), (0,3), (1,2), (2,3), (3,0)])
>>> nx.degree_assortativity_coefficient(g, x = 'in', y = 'in')
-0.250000000000001
>>> nx.degree_assortativity_coefficient(g, x = 'in', y = 'out')
0.63245553203367555
>>> nx.degree_assortativity_coefficient(g, x = 'out', y = 'in')
0.0
>>> nx.degree_assortativity_coefficient(g, x = 'out', y = 'out')
-0.44721359549995793
```

There are also other functions that can calculate assortativities of node properties other than degrees. Check out NetworkX's online documentation for more details.

Exercise 17.13 Measure the degree assortativity coefficient of the Karate Club graph. Explain the result in view of the actual topology of the graph.

It is known that real-world networks show a variety of assortativity. In general, social networks of human individuals, such as collaborative relationships among scientists or corporate directors, tend to show positive assortativity, while technological networks (power grid, the Internet, etc.) and biological networks (protein interactions, neural networks, food webs, etc.) tend to show negative assortativity [76].

In the meantime, it is also known that scale-free networks of a finite size (which many real-world networks are) naturally show negative disassortativity purely because of inherent structural limitations, which is called a *structural cutoff* [25]. Such disassortativity arises because there are simply not enough hub nodes available for themselves to connect to each other to maintain assortativity. This means that the positive assortativities found in human social networks indicate there is definitely some assortative mechanism driving their self-organization, while the negative assortativities found in technological and biological networks may be explained by this simple structural reason. In order to determine whether or not a network showing negative assortativity is fundamentally disassortative for non-structural reasons, you will need to conduct a control experiment by randomizing its topology and measuring assortativity while keeping the same degree distribution.

> *Exercise 17.14* Randomize the topology of the Karate Club graph while keeping its degree sequence, and then measure the degree assortativity coefficient of the randomized graph. Repeat this many times to obtain a distribution of the coefficients for randomized graphs. Then compare the distribution with the actual assortativity of the original Karate Club graph. Based on the result, determine whether or not the Karate Club graph is truly assortative or disassortative.

17.7 Community Structure and Modularity

The final topics of this chapter are the *community structure* and *modularity* of a network. These topics have been studied very actively in network science for the last several years. These are typical *mesoscopic properties* of a network; neither microscopic (e.g., degrees or clustering coefficients) nor macroscopic (e.g., density, characteristic path length) properties can tell us how a network is organized at spatial scales intermediate between those two extremes, and therefore, these concepts are highly relevant to the modeling and understanding of complex systems too.

Community A set of nodes that are connected more densely to each other than to the rest of the network. Communities may or may not overlap with each other, depending on their definitions.

Modularity The extent to which a network is organized into multiple communities.

Figure 17.13 shows an example of communities in a network.

There are literally dozens of different ways to define and detect communities in a network. But here, we will discuss just one method that is now widely used by network science researchers: the *Louvain method*, proposed by Vincent Blondel et al. in 2008 [77]. It is a very fast, efficient heuristic algorithm that maximizes the modularity of non-overlapping community structure through an iterative, hierarchical optimization process.

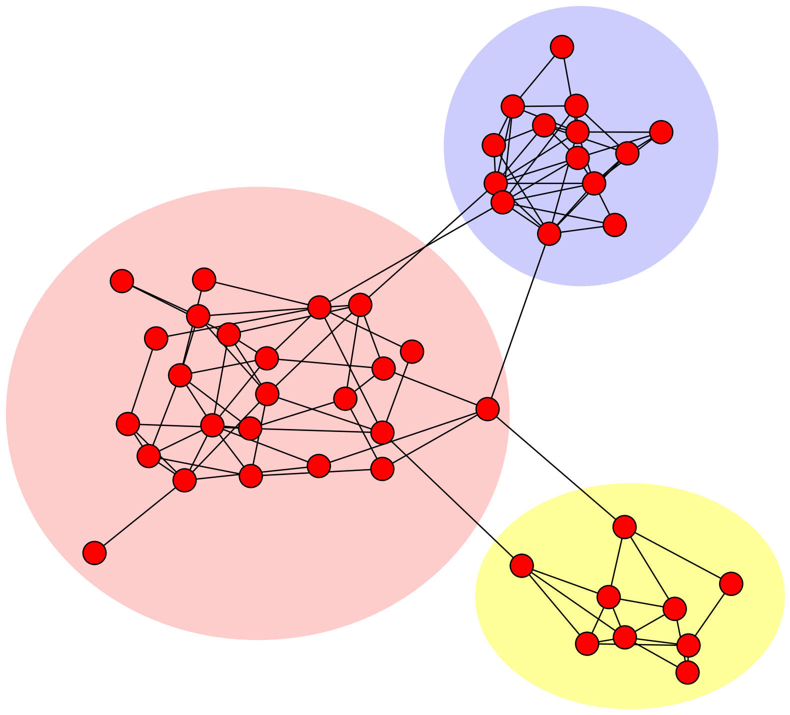


Figure 17.13: Example of communities in a network.

The modularity of a given set of communities in a network is defined as follows [78]:

$$Q = \frac{|E_{\mathrm{in}}| - \langle |E_{\mathrm{in}}| \rangle}{|E|} \tag{17.35}$$

Here, $|E|$ is the number of edges, $|E_{\mathrm{in}}|$ is the number of within-community edges (i.e., those that don't cross boundaries between communities), and $\langle |E_{\mathrm{in}}| \rangle$ is the expected number of within-community edges if the topology were purely random. The subtraction of $\langle |E_{\mathrm{in}}| \rangle$ on the numerator penalizes trivial community structure, such as considering the entire network a single community that would trivially maximize $|E_{\mathrm{in}}|$.

The Louvain method finds the optimal community structure that maximizes the modularity in the following steps:

1. Initially, each node is assigned to its own community where the node itself is the only community member. Therefore the number of initial communities equals the number of nodes.

2. Each node considers each of its neighbors and evaluates whether joining to the neighbor's community would increase the modularity of the community structure. After evaluating all the neighbors, it will join the community of the neighbor that achieves the maximal modularity increase (only if the change is positive; otherwise the node will remain in its own community). This will be repeatedly applied for all nodes until no more positive gain is achievable.

3. The result of Step 2 is converted to a new meta-network at a higher level, by aggregating nodes that belonged to a single community into a meta-node, representing edges that existed within each community as the weight of a self-loop attached to the meta-node, and representing edges that existed between communities as the weights of meta-edges that connect meta-nodes.

4. The above two steps are repeated until no more modularity improvement is possible.

One nice thing about this method is that it is parameter-free; you don't have to specify the number of communities or the criteria to stop the algorithm. All you need is to provide network topology data, and the algorithm heuristically finds the community structure that is close to optimal in achieving the highest modularity.

Unfortunately, NetworkX doesn't have this Louvain method as a built-in function, but its Python implementation has been developed and released freely by Thomas Aynaud, which is available from `http://perso.crans.org/aynaud/communities/`. Once you install it, a new `community` module becomes available in Python. Here is an example:

Code 17.22:

```
>>> import networkx as nx
>>> import community as comm
>>> g = nx.karate_club_graph()
>>> bp = comm.best_partition(g)
>>> bp
{0: 0, 1: 0, 2: 0, 3: 0, 4: 1, 5: 1, 6: 1, 7: 0, 8: 2, 9: 0, 10: 1,
  11: 0, 12: 0, 13: 0, 14: 2, 15: 2, 16: 1, 17: 0, 18: 2, 19: 0,
  20: 2, 21: 0, 22: 2, 23: 3, 24: 3, 25: 3, 26: 2, 27: 3, 28: 3,
  29: 2, 30: 2, 31: 3, 32: 2, 33: 2}
>>> comm.modularity(bp, g)
0.4188034188034188
```

Here, the two important functions in the `community` module are tested. The first one is `best_partition`, which generates community structure using the Louvain method. The result is given as a dictionary where keys and values are node IDs and community IDs, respectively. The second function shown above is `modularity`, which receives community structure and a network and returns the modularity value achieved by the given communities.

Exercise 17.15 Visualize the community structure in the Karate Club graph using the community IDs as the colors of the nodes.

Exercise 17.16 Import a large network data set of your choice from Mark Newman's Network Data website: `http://www-personal.umich.edu/~mejn/netdata/` Detect its community structure using the Louvain method and visualize it if possible.

Exercise 17.17 Do a quick online literature search for other community detection algorithms (e.g., Girvan-Newman method, k-clique percolation method, random walk method, etc.). Choose one of them and read the literature to learn how it works. If the software is available, try it yourself on the Karate Club graph or any other network and see how the result differs from that of the Louvain method.