

---

# Networks

---

---

Fred Jones of Peoria, sitting in a sidewalk cafe in Tunis and needing a light for his cigarette, asks the man at the next table for a match. They fall into conversation; the stranger is an Englishman who, it turns out, spent several months in Detroit studying the operation of an interchangeable-bottlecap factory. "I know it's a foolish question," says Jones, "but did you ever by any chance run into a fellow named Ben Arkadian? He's an old friend of mine, manages a chain of supermarkets in Detroit. . ."

"Arkadian, Arkadian," the Englishman mutters. "Why, upon my soul, I believe I do! Small chap, very energetic, raised merry hell with the factory over a shipment of defective bottlecaps."

"No kidding!" Jones exclaims in amazement.

"Good lord, it's a small world, isn't it?"

Stanley Milgram

*The Small-World Problem (1967)*

---

WHAT do Instagram, food webs, the banking system, and our brains all have in common? They are all *networks*: systems of interconnected units that exchange information over the links between them. There are networks all around us: social networks, road networks, protein interaction networks, electrical transmission networks, the Internet, networks of seismic faults, terrorist networks, networks of political influence, transportation networks, and semantic networks, to name a few. The continuous and dynamic local interactions in large networks such as these make them extraordinarily complex and hard to predict. Learning more about networks can help us combat disease, terrorism, and power outages. Realizations that some networks are *emergent* systems that develop global behaviors based on local interactions have improved our understanding of insect colonies, urban planning, and even our brains. Too little understanding of networks has had unfortunate

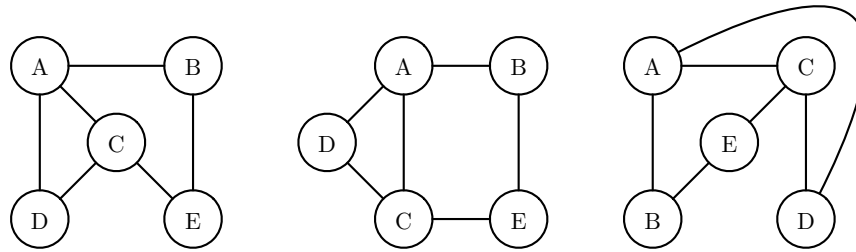


Figure 11.1 Three representations of the same graph.

consequences, such as when invasive species have been introduced into poorly understood ecological networks.

As with the other types of “big data,” we need computer algorithms to understand large and complex networks. In this chapter, we will begin by discussing how we can represent networks in algorithms so that we can analyze them. Then we will develop an algorithm to find the distance between any two nodes in a network. Recent discoveries have shown that many real networks exhibit a “small-world property,” meaning that the average distance between nodes is relatively small. In later sections, we will investigate the characteristics of small-world networks and their ramifications for solving real problems.

## 11.1 MODELING WITH GRAPHS

Networks are modeled with a mathematical structure called a *graph*. A graph consists of a set of *nodes* (or *vertices*) and a set of *links* (or *edges*) that connect pairs of nodes. If two nodes are connected by a link, we say they are *adjacent*. Nodes are usually drawn as circles (or another shape) and links are drawn as lines between them, but the placement of the nodes and links on the page is arbitrary. For example, all three of the graphs in Figure 11.1 are equivalent.

A social network (like Facebook or LinkedIn) can be represented by a graph in which the nodes are people and the links represent relationships (e.g., friends, connections, circles, followers). For example, in the social network in Figure 11.2, Caroline has three friends: Amelia, Lillian, and Nick. In a neural network, the nodes represent neurons and the links represent axons that transmit nerve impulses between the neurons. Figure 11.3 represents the interconnections between neurons in one of the simple neural networks that control digestion in the guts of arthropods. In a graph representing a power grid, like that in Figure 11.4, the nodes represent power stations and the links represent high-voltage transmission lines connecting them.

In an algorithm, a graph is usually represented in one of two ways. The first is called an *adjacency matrix*. An adjacency matrix contains a row and a column for every node in the network. A one in a matrix entry represents a link between the nodes in

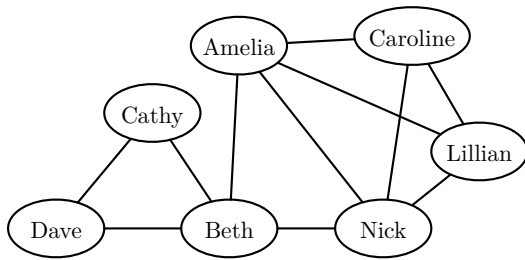


Figure 11.2 A small social network.

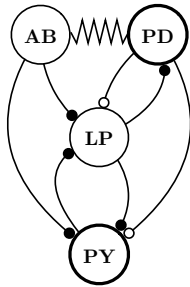


Figure 11.3 A model of the pyloric central pattern generator that controls stomach motion in lobsters.

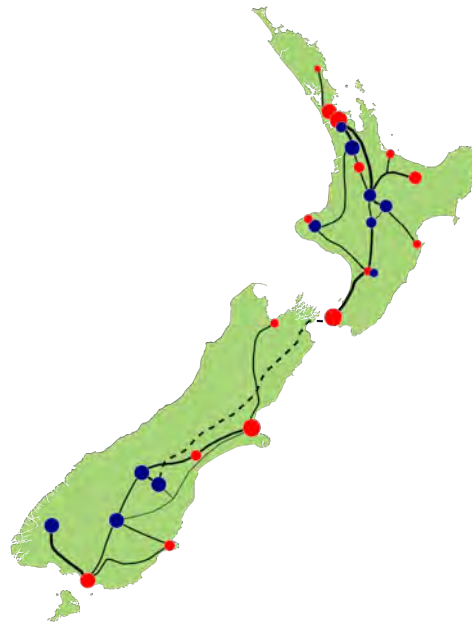


Figure 11.4 The electrical transmission network in New Zealand. [67]

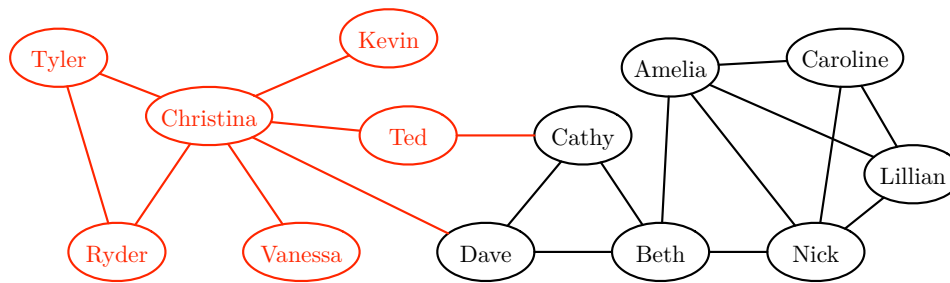
the corresponding row and column. A zero means that there is no link. The following table represents an adjacency matrix for the network in Figure 11.2.

	Amelia	Beth	Caroline	Cathy	Dave	Lillian	Nick
Amelia	0	1	1	0	0	1	1
Beth	1	0	0	1	1	0	1
Caroline	1	0	0	0	0	1	1
Cathy	0	1	0	0	1	0	0
Dave	0	1	0	1	0	0	0
Lillian	1	0	1	0	0	0	1
Nick	1	1	1	0	0	1	0

The first row indicates that Amelia is connected to Beth, Caroline, Lillian, and Nick. The second row shows that Beth is connected to Amelia, Cathy, Dave, and Nick. In Python, we would represent this matrix with the following nested list.

```
graph = [[0, 1, 1, 0, 0, 1, 1],
          [1, 0, 0, 1, 1, 0, 1],
          [1, 0, 0, 0, 0, 1, 1],
          [0, 1, 0, 0, 1, 0, 0],
          [0, 1, 0, 1, 0, 0, 0],
          [1, 0, 1, 0, 0, 0, 1],
          [1, 1, 1, 0, 0, 1, 0]]
```

Although the nodes' labels are not stored in the adjacency matrix itself, they could



**Figure 11.5** An expanded social network. The nodes and links in red are additions to the graph in Figure 11.2.

be stored separately as strings in a list. The index of each string in the list should equal the row/column of the corresponding node in the adjacency matrix.

**Reflection 11.1** Create an adjacency matrix for the graph in Figure 11.1. (Remember that all three pictures depict the same graph.)

The alternative representation, which we will use in this chapter, is an *adjacency list*. An adjacency list, which is actually a collection of lists, contains, for each node, a list of nodes to which it is connected. In Python, an adjacency list can be stored as a dictionary. The following dictionary represents the network in Figure 11.2.

```
graph = { 'Amelia': ['Beth', 'Caroline', 'Lillian', 'Nick'],
          'Beth':   ['Amelia', 'Cathy', 'Dave', 'Nick'],
          'Caroline': ['Amelia', 'Lillian', 'Nick'],
          'Cathy':   ['Beth', 'Dave'],
          'Dave':    ['Beth', 'Cathy'],
          'Lillian': ['Amelia', 'Caroline', 'Nick'],
          'Nick':    ['Amelia', 'Beth', 'Caroline', 'Lillian'] }
```

Each key in this dictionary, a string, represents a node, and each corresponding value is a list of strings representing the nodes to which the key node is connected. Notice that, if two nodes are connected, that information is stored in both nodes' lists. For example, there is a link connecting Amelia and Beth, so Beth is in Amelia's list and Amelia is in Beth's list.

**Reflection 11.2** Create an adjacency list for the graph in Figure 11.1.

### Making friends

Social networking sites often have an eerie ability to make good suggestions about who you should add to your list of “connections” or “friends.” One way they do this is by examining the connections of your connections (or “friends-of-friends”). For example, consider the expanded social network graph in Figure 11.5. Dave currently has only three friends. But his friends have an additional seven friends that an algorithm could suggest to Dave.

**Reflection 11.3** *Who are the seven friends-of-friends of Dave in Figure 11.5?*

In graph terminology, the connections of a node are called the node's *neighborhood*, and the size of a node's neighborhood is called its *degree*. In the graph in Figure 11.5, Dave's neighborhood contains Beth, Cathy and Christina, and therefore his degree is three.

**Reflection 11.4** *How can you compute the degree of a node from the graph's adjacency matrix? What about from the graph's adjacency list?*

Once we have a network in an adjacency list, writing an algorithm to collect new friend suggestions is relatively easy. The function below iterates over the neighbors of the node for which we would like suggestions and then, for each of these neighbors, iterates over the neighbors' neighbors.

---

```

1 def friendsOfFriends(network, node):
2     """Find new neighbors-of-neighbors of a node in a network.
3
4     Parameters:
5         network: a graph represented by a dictionary
6         node:    a node in the network
7
8     Return value: a list of new neighbors-of-neighbors of node
9     """
10    suggestions = [ ]
11    neighbors = network[node]
12    for neighbor in neighbors:           # neighbors of node
13        for neighbor2 in network[neighbor]: # neighbors of neighbors
14            if neighbor2 != node and \
15                neighbor2 not in neighbors and \
16                neighbor2 not in suggestions:
17                suggestions.append(neighbor2)
18    return suggestions

```

---

On line 9, `network[node]` is the list of nodes to which `node` is connected in the adjacency list named `network`. We assign this list to `neighbors`, and then iterate over it on line 10. On line 11, in the inner `for` loop, we then iterate over the list of each neighbors' neighbors. In the `if` statement, we choose `suggestions` to be a list of unique neighbors-of-neighbors that are not the `node` itself or neighbors of the `node`.

**Reflection 11.5** *Look carefully at the three-part `if` statement in the function above. How does each part contribute to the desired characteristics of `suggestions` listed above?*

**Reflection 11.6** *Insert the additional nodes and links from Figure 11.5 (in red) into the dictionary on page 446. Then call the `friendsOfFriends` function with this graph to find new friend suggestions for Dave.*

In the next section, we will design an algorithm to find paths to nodes that are

farther away. The ability to compute the distance between nodes will also allow us to better characterize and understand large networks.

### Exercises

11.1.1. Besides those presented in this section, describe three more examples of networks.

11.1.2. Draw the networks represented by the following adjacency matrices.

(a)\*

	A	B	C	D	E
A	0	1	0	1	0
B	1	0	1	1	1
C	0	1	0	1	0
D	1	1	1	0	1
E	0	1	0	1	0

(b)

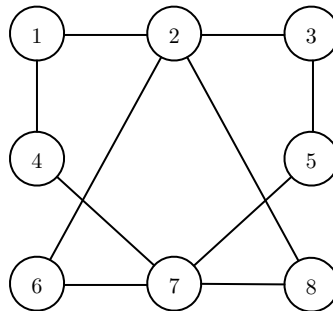
	A	B	C	D	E
A	0	1	1	0	0
B	1	0	0	1	1
C	1	0	0	0	1
D	0	1	0	0	1
E	0	1	1	1	0

11.1.3. Draw the networks represented by each of the following adjacency lists.

(a)\* `graph = {'A': ['C', 'D', 'F'],`  
`'B': ['C', 'E'],`  
`'C': ['A', 'B', 'D'],`  
`'D': ['A', 'C'],`  
`'E': ['B', 'F'],`  
`'F': ['A', 'E']}`

(b) `graph = {'A': ['C', 'D'],`  
`'B': ['C', 'D'],`  
`'C': ['A', 'B', 'D'],`  
`'D': ['A', 'B'],`  
`'E': ['F'],`  
`'F': ['E']}`

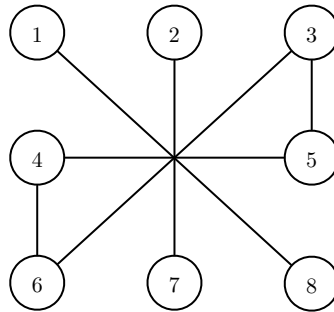
11.1.4\* Consider the following network.



(a) Show how to represent this network in Python as an adjacency matrix.

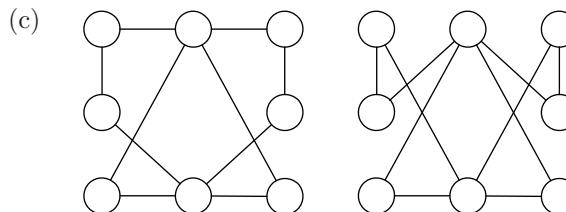
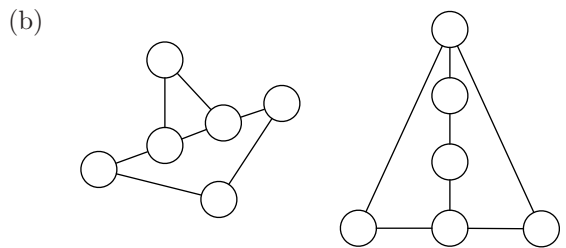
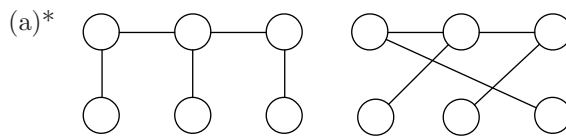
- (b) Show how to represent this network in Python as an adjacency list.
- (c) What is the neighborhood of each of the nodes in the network?
- (d) What is the degree of each node? Which node(s) have the maximum degree?

11.1.5. Consider the following network.



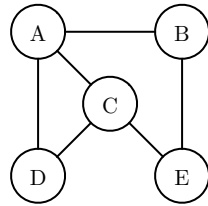
- (a) Show how to represent this network in Python as an adjacency matrix.
- (b) Show how to represent this network in Python as an adjacency list.
- (c) What is the neighborhood of each of the nodes in the network?
- (d) What is the degree of each node? Which node(s) have the maximum degree?

11.1.6. Are the networks in each of the following pairs the same or different? Why?



11.1.7\* A graph can be represented in a file by listing one link per line, with each link represented by a pair of nodes. For example, the graph below is represented

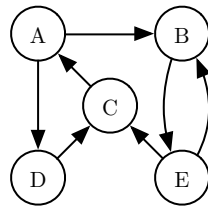
by the file on the right. Write a function that reads such a file and returns an adjacency list (as a dictionary) for the graph. Notice that, for each line A B in the file, your function will need to insert node B into the list of neighbors of A *and* insert node A into the list of neighbors of B.



graph.txt

```
A B
A C
A D
B E
C D
C E
```

- 11.1.8. In this chapter, we assumed that all graphs are *undirected*, meaning that each link represents a mutual relationship between two nodes. For example, if there is a link between nodes A and B, then this means that A is friends with B *and* B is friends with A, or that one can travel from city A to city B *and* from city B to city A. However, the relationships between nodes in some networks are not mutual or do not exist in both directions. Such a network is more accurately represented by a *directed graph* (or *digraph*), in which links are directed from one node to another. In a picture, the directions are indicated arrows. For example, in the directed graph below, one can go directly from node A to node B, but not vice versa. However, one can go in both directions between nodes B and E.



digraph.txt

```
A B
A D
B E
C A
D C
E B
E C
```

- Give three examples of networks that are better represented by a directed graph.
  - How would an adjacency list representation of a directed graph differ from that of an undirected graph?
  - Write a function that reads a file representing a directed graph (see the example above), and returns an adjacency list (as a dictionary) representing that directed graph.
- 11.1.9. Write a function that returns the maximum degree in a network represented by an adjacency list (dictionary).
- 11.1.10. Write a function that returns the average degree in a network represented by an adjacency list (dictionary).

## 11.2 SHORTEST PATHS

A sequence of links (or equivalently, linked nodes) between two nodes is called a *path*. A path with the minimum number of links is called a *shortest path*. For example, in Figure 11.5, a shortest path from Dave to Lillian starts with Dave, then visits Beth, then Nick, then Lillian. The *distance* between two nodes is the number of links on a shortest path between them. Because three links were crossed along the shortest path from Dave to Lillian, the distance between them is 3.

$$\text{Dave} \xrightarrow{1} \text{Beth} \xrightarrow{2} \text{Nick} \xrightarrow{3} \text{Lillian}$$

Computing the distance between two nodes is a fundamental problem in network analysis. In a transportation network, the distance between a source and destination gives the number of stops along the route. In an ecological network, the distance between two organisms may be a measure of how directly dependent one organism is upon the other. In a social network, the distance between two people is the number of introductions by friends that would be necessary for one person to meet the other.

**Reflection 11.7** *Are shortest paths always unique? Is there another shortest path between Dave and Lillian?*

Yes, there is:

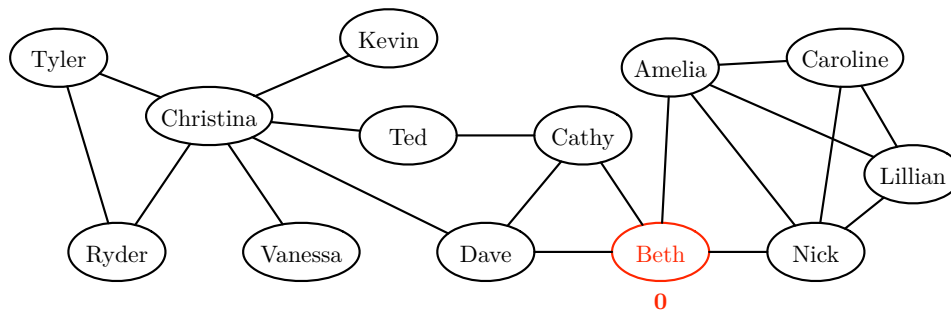
$$\text{Dave} \xrightarrow{1} \text{Beth} \xrightarrow{2} \text{Amelia} \xrightarrow{3} \text{Lillian}$$

There may be many shortest paths between two nodes in a network, but in most applications we are concerned with just finding one.

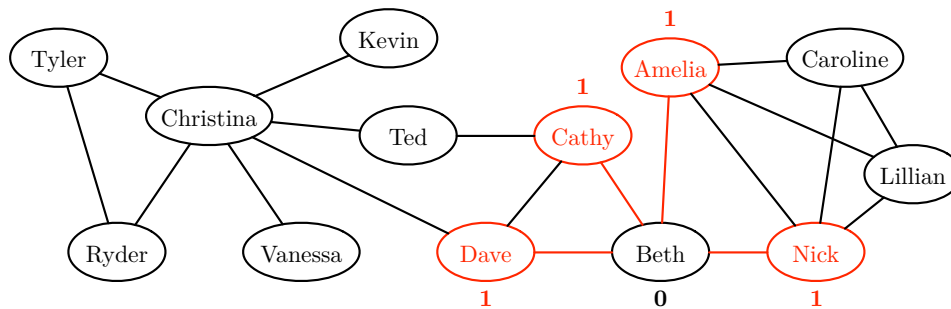
### Breadth-first search

Shortest paths can be computed using an algorithm called *breadth-first search* (BFS). A breadth-first search explores outward from a source node, first visiting all nodes with distance one from the source, then all nodes with distance two, etc., until it has visited every reachable node in the network. In other words, the BFS algorithm incrementally pushes its “frontier” of visited nodes outward from the source. When the algorithm finishes, it has computed the distances between the source node and every other node.

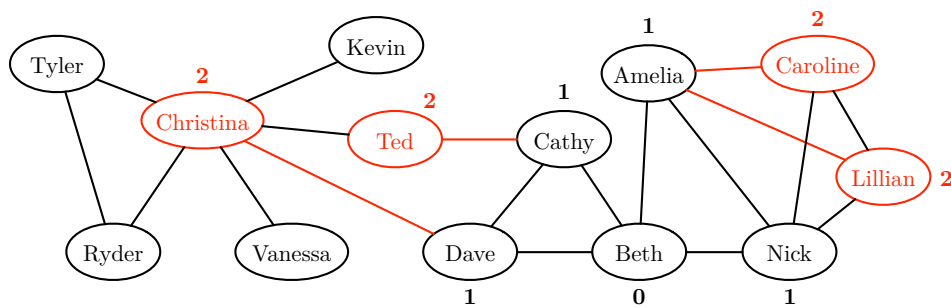
For example, suppose we wanted to discover the distance from Beth to every other person in the social network in Figure 11.5, reproduced below.



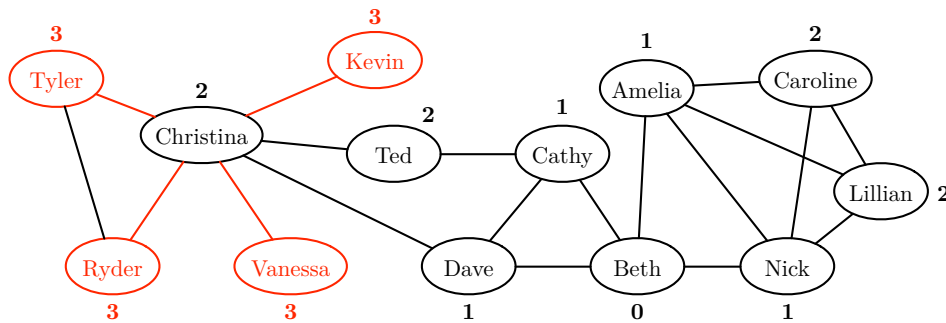
As indicated above, we begin by labeling Beth's node with distance 0, signifying that there are zero links between the node and itself. Then, in the first round of the algorithm, we explore all neighbors of Beth, colored red below.



Since these nodes are one hop away from the source, we label them with distance 1. These nodes now comprise the “frontier” being explored by the algorithm. In the next round, we explore all unvisited neighbors of the nodes on this frontier, as shown below.



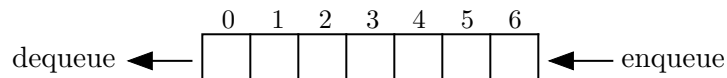
As indicated by the red links, Christina is visited from Dave, Ted is visited from Cathy, and both Caroline and Lillian are visited from Amelia. Notice that Caroline and Lillian could have been visited from Nick as well. The decision is arbitrary, depending, as we will see, on the order in which nodes are considered by the algorithm. Since all four of these nodes are neighbors of a node with distance 1, we label them with distance 2. Finally, in the third round, we visit all unvisited neighbors of the new frontier of nodes, as shown below.



Since these newly visited nodes are all neighbors of a node labeled with distance 2, we label all of them with distance 3. At this point, all of the nodes have been visited, and the final label of each node gives its distance from the source.

**Reflection 11.8** *If you also studied the depth-first search algorithm in Section 9.5, compare and contrast that approach with breadth-first search.*

In an algorithm, keeping track of the nodes on the current frontier could get complicated. The trick is to use a **queue**. A queue is a list in which items are always inserted at the end and deleted from the front. The insertion operation is called *enqueue* and the deletion operation is called *dequeue*.



**Reflection 11.9** *In Python, if we use a list named `queue` to implement a queue, how do we perform the enqueue and dequeue operations?*

An enqueue operation is simply an append:

```
queue.append(item) # enqueue an item
```

And then a dequeue can be implemented by “popping” the front item from the list:

```
item = queue.pop(0) # dequeue an item
```

In the breadth-first search algorithm, we use a queue to remember those nodes on the “frontier” that have been visited, but from which the algorithm has not yet visited new nodes. When we are ready to visit the unvisited neighbors of a node on the frontier, we dequeue that node, and then enqueue the newly visited neighbors so that we can remember to explore outward from them later.

**Reflection 11.10** *Why can we not explore outward from these newly visited neighbors right away? Why do they need to be stored in the queue for later?*

We need to wait because there may be nodes further ahead in the queue that have smaller distances from the source. For the algorithm to work correctly, we have to explore outward from these nodes first.

The Python function implements the breadth-first search algorithm.

---

```

1 import math

2 def bfs(network, source):
3     """Perform a breadth-first search on network, starting from source.

4     Parameters:
5         network: a graph represented by a dictionary
6         source: the node in network from which to start the BFS

7     Return value: a dictionary with distances from source to all nodes
8     """

9     visited = { }                # initialize all nodes
10    distance = { }               # to be unvisited
11    for node in network:
12        visited[node] = False
13        distance[node] = math.inf
14    visited[source] = True       # mark source visited
15    distance[source] = 0

16    queue = [source]            # start with the source
17    while queue != [ ]:
18        front = queue.pop(0)     # dequeue front node
19        for neighbor in network[front]: # visit every neighbor
20            if not visited[neighbor]:
21                visited[neighbor] = True
22                distance[neighbor] = distance[front] + 1
23                queue.append(neighbor) # enqueue visited node

24    return distance

```

---

The function maintains two dictionaries: `visited` keeps track of whether each node has been visited and `distance` keeps track of the distance from the `source` to each node. Lines 8–14 initialize the dictionaries. Every node, except the source, is marked as unvisited and assigned an initial distance of infinity ( $\infty$ ), represented by `math.inf`, because we do not yet know which nodes can be reached from the source. The source is marked as visited and assigned distance zero. On line 16, the queue is initialized to contain just the source node. Then, while the queue is not empty, the algorithm repeatedly dequeues the front node (line 18), and explores all neighbors of this node (lines 19–23). If a neighbor has not yet been visited (line 20), it is marked as visited (line 21), assigned a distance that is one greater than the node from which it is being visited (line 22), and then enqueued (line 23). Once the queue is empty, we know that all reachable nodes have been visited, so we return the `distance` dictionary, which now contains the distance to each node.

**Reflection 11.11** Call the `bfs` function with the `graph` that you created in the previous section to find the distances from Beth to all other nodes.

**Reflection 11.12** What does it mean if the `bfs` function returns a distance of  $\infty$  for a node?

If the final distance is  $\infty$ , then the node must not have been visited by the algorithm, which means that there is no path to it from the source.

**Reflection 11.13** If you just want the distance between two particular nodes, named `source` and `dest`, how can you use the `bfs` function to find it?

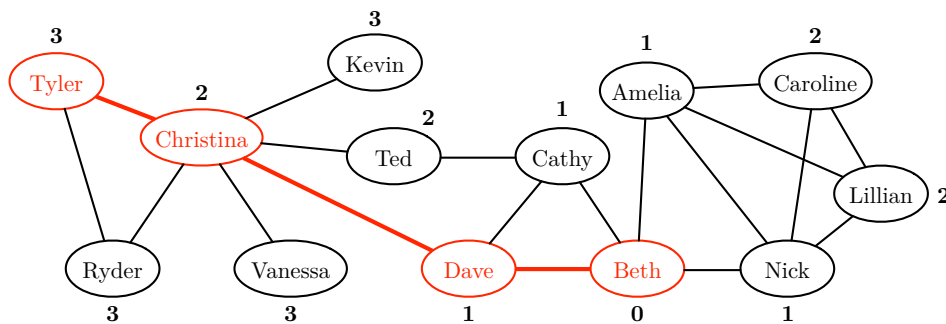
The `bfs` function finds the distance from a source node to every node, so you just need to call `bfs` and then pick out the particular distance you are interested in:

```
allDistances = bfs(graph, source)
distance = allDistances[dest]
```

### Finding the actual paths

In some applications, just finding the distance between two nodes is not enough; we actually need a path between the nodes. For example, just knowing that you are only three hops away from that potential employer in your social network is not very helpful. You want to know who to ask to introduce you! And in a road network, we want to know the actual directions, not just the distance.

Fortunately, the breadth-first search algorithm is already finding the shortest paths; we just need make some modifications to remember them. Consider how the distance from Beth to Tyler was computed in the previous example. As depicted below, from Beth, we visited Dave; from Dave, we visited Christina; and from Christina, we visited Tyler.



This sequence of nodes is the shortest path on which we based the distance to Tyler. Therefore, all we have to do is remember this order of nodes, as we visit them.

We implement this by adding another dictionary, named `predecessor`, to the `bfs` function. The `predecessor` dictionary remembers the node that comes before each node on the shortest path to it from the source. The dictionary needs to be initially assigned a value of `None` for every node in the `for` loop on lines 11–13. In the `while` loop, after line 22, when each `neighbor` is visited, we set `predecessor[neighbor]` to be `front`.

**Reflection 11.14** *Modify the `bfs` function to incorporate the predecessor dictionary. At the end of the function, return `predecessor` in addition to `distance`. Test the function with your graph dictionary by calling it with:*

```
distances, predecessors = bfs(graph, 'Beth')
```

*What are the predecessors of 'Beth', 'Dave', 'Christina', and 'Tyler'?*

**Reflection 11.15** *How can we use the final values in the predecessor dictionary to construct a shortest path between the source node and another node?*

To construct the path to any particular node, we need to follow the predecessors *backward* from the destination. As we follow them, we will insert each one into the *front* of a list so they are in the correct order when we are done. To find the shortest path from Beth to Tyler, we start at Tyler.

```
path = ['Tyler']
```

Tyler's predecessor was Christina, so we insert Christina into the front of the list:

```
path = ['Christina', 'Tyler']
```

Christina's predecessor was Dave, so we next insert Dave into the front of the list:

```
path = ['Dave', 'Christina', 'Tyler']
```

Finally, Dave's predecessor was Beth:

```
path = ['Beth', 'Dave', 'Christina', 'Tyler']
```

Since Beth was the source, we stop. The following function implements this algorithm.

---

```
def path(network, source, dest):
    """Find a shortest path in network from source to dest.

    Parameters:
        network: a graph represented by a dictionary
        source:  the source node in network
        dest:    the destination node in network

    Return value: a list containing a path from source to dest
    """

    allDistances, allPredecessors = bfs(network, source)

    path = [ ]
    current = dest
    while current != source:
        path.insert(0, current)
        current = allPredecessors[current]
    path.insert(0, source)

    return path
```

---

Starting with `current = dest`, in each iteration, the `while` loop moves `current`

one step closer to the source by assigning it to its predecessor. As this is happening, each value of `current` is inserted into the front of `path`. When `current` reaches the source, the loop ends and we insert the `source` as the first node in the `path`.

**Reflection 11.16** Find the shortest path between Beth and Tyler with `print(path(graph, 'Beth', 'Tyler'))`.

In the next section, we will use information about shortest paths to investigate a special kind of network called a *small-world* network.

### Exercises

- 11.2.1\* List the order in which nodes are visited by `bfs` when it is called to find the distance from Ted to every node in the graph in Figure 11.5. (There is more than one correct answer.)
- 11.2.2. List the order in which nodes are visited by `bfs` when it is called to find the distance between Caroline and every node in the graph in Figure 11.5. (There is more than one correct answer.)
- 11.2.3. By modifying one line, the `visited` dictionary can be completely removed from the `bfs` function. Show how.
- 11.2.4. Write a function that uses `bfs` to return the distance in a graph between two particular nodes. The function should take three parameters: the graph, the source node, and the destination node.
- 11.2.5. We say that a graph is *connected* if there is a path between any pair of nodes. Show how to modify `bfs` so that it returns a Boolean value indicating whether a graph is connected.
- 11.2.6. A depth-first search algorithm (see Section 9.5) can also be used to determine whether a graph is connected. Recall that a depth-first search recursively searches as far from the source as it can, and then backtracks when it reaches a dead end. Writing a depth-first search algorithm for a graph is actually much easier than writing the one in Section 9.5 because there are fewer base cases to deal with.

(a)\* Write a function

```
dfs(network, source, visited)
```

that performs a depth-first search on the given `network`, starting from the given `source` node. The third parameter, `visited`, is a list of nodes that have been visited by the depth-first search. The initial list argument passed in for `visited` should be empty, but when the function returns, `visited` should contain all of the visited nodes. In other words, you should call the function initially like this:

```
visited = []
dfs(network, source, visited)
```

(b) Write another function

```
connected(network)
```

that calls your `dfs` function to determine whether `network` is connected. (The source node can be any node in the network.)

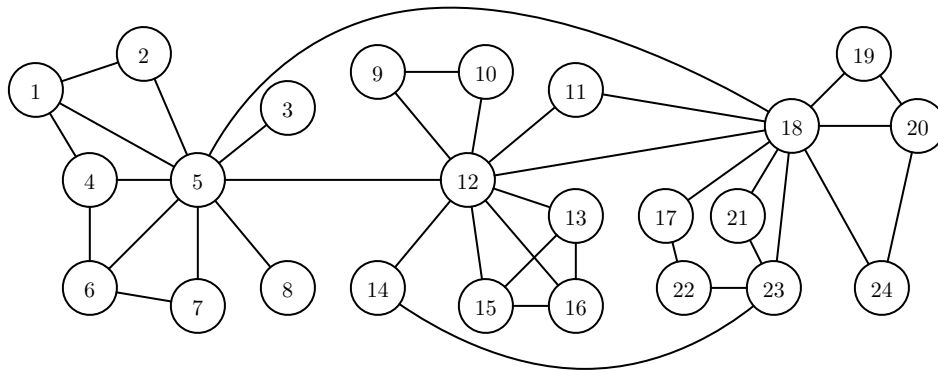


Figure 11.6 A very small small-world network.

### 11.3 IT'S A SMALL WORLD...

In a famous 1967 experiment, sociologist Stanley Milgram asked several individuals in the midwestern United States to forward a postcard to a particular person in Boston, Massachusetts. If they did not know this person on a first-name basis, they were asked instead to forward it to someone they thought might have a better chance of knowing the person. Each intermediate person was asked to follow the same instructions. Of the postcards that made it to the destination (many were simply not forwarded), the average number of hops was about six.

#### Small world networks

From this experiment later came the suggestion that there are only “six degrees of separation” between any two people on Earth, and that the human race must constitute a *small-world network*. In the late 1990s, using computers, researchers began to discover that networks representing a wide range of unrelated phenomena, from social networks to neural networks to the Internet, all exhibit the same small-world property: for most nodes in the network, there is a very short path connecting them. Put another way, in a small-world network, the average distance between any two nodes is small.

Intuitively, it seems as though a small-world network must have a lot of links to facilitate so many short paths. However, it has been shown that small-world networks can actually be quite *sparse*, meaning that the number of links is quite small relative to the number possible. The keys to a small-world network are a high degree of *clustering* and a few long-range shortcuts that facilitate short paths between clusters. A cluster is a set of nodes that are highly connected among themselves. In your social network, you probably participate in several clusters: family, friends at school, friends at home, co-workers, teammates, etc. Many of the members of each of these clusters are probably also connected to one another, but members of different clusters might be far apart if you did not act as a shortcut link between them.

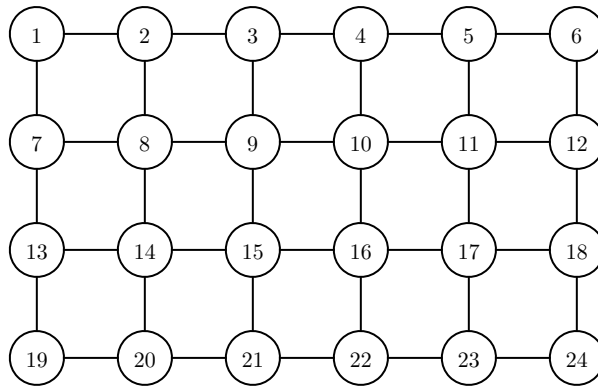
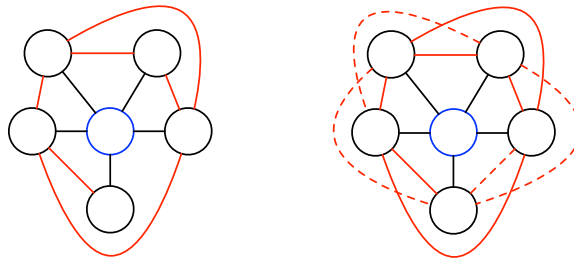


Figure 11.7 A grid network.

Although it is too small to really be called a small-world network, the network in Figure 11.6 illustrates these ideas. The graph contains three clusters of nodes, centered around nodes 5, 12 and 18, that are connected by a few shortcut links (e.g., the links between nodes 5 and 18 and between nodes 14 and 23). These two characteristics together give an average distance between nodes of about 2.42. On the other hand, the highly structured grid network in Figure 11.7 has an average node distance of about 3.33. Both of these graphs have 24 nodes and 38 links, so they are both sparse relative to their  $(24 \cdot 23)/2 = 276$  maximum possible links.

### Clustering coefficients

The extent to which the neighborhood of a node is clustered is measured by its *local clustering coefficient*. The local clustering coefficient of a node is the number of links between its neighbors, divided by the maximum number of possible links between its neighbors. For example, consider the cluster on the left below surrounding the blue node in the center.



The blue node has five neighbors, with six links between them (in red). Notice that each of these links, together with two black links, forms a closed cycle, called a *triangle*. So we can also think about the local clustering coefficient as counting these triangles. As shown on the right, there are four dashed links between neighbors of the blue node (i.e., four additional triangles) that are not present on the left, for a

total of ten possible links altogether. So the local clustering coefficient of the blue node is  $6/10 = 0.6$ . (The clustering coefficient will always be between 0 and 1.)

**Reflection 11.17** *In general, if a node has  $k$  neighbors, how many possible links are there between pairs of these neighbors?*

Each of the  $k$  neighbors could be connected to  $k - 1$  other neighbors, for a total of  $k(k - 1)$  links. However, this counts each link twice, so the total number of unique links is actually  $k(k - 1)/2$ . Therefore, the local clustering coefficient of a node with  $k$  neighbors is the number of pairs of neighbors that are connected to each other, divided by  $k(k - 1)/2$ . The clustering coefficient for a network is the average local clustering coefficient of its nodes. The highly structured grid or mesh graph in Figure 11.7 does not have any triangles at all, so its clustering coefficient is 0. On the other hand, the graph in Figure 11.6 has a clustering coefficient of about 0.59.

**Reflection 11.18** *If you had a small local clustering coefficient in your social network (i.e., if your friends are not friends with each other), what implications might this have?*

It has been suggested that situations like this breed instability. Imagine that, instead of a social network, we are talking about a network of nations and links represent the existence of diplomatic relations. A nation with diplomatic relations with many other nations that are enemies of each other is likely in a stressful situation. It might be helpful to detect such situations in advance to curtail potential conflicts.

To compute the local clustering coefficient for a node, we iterate over each of the node's neighbors and count the number of links between it and the other neighbors of the node. Then we divide this number by the maximum possible number of links between the node's neighbors. This is accomplished by the following function.

---

```
def clusteringCoefficient_Draft(network, node):
    """Compute the local clustering coefficient for a node.

    Parameters:
        network: a graph represented by a dictionary
        node:    a node in the network

    Return value: the local clustering coefficient of node
    """

    neighbors = network[node]
    numNeighbors = len(neighbors)
    if numNeighbors <= 1:
        return 0
    numLinks = 0
    for neighbor1 in neighbors:
        for neighbor2 in neighbors:
            if neighbor1 != neighbor2 and neighbor1 in network[neighbor2]:
                numLinks = numLinks + 1
    return numLinks / (numNeighbors * (numNeighbors - 1))
```

---

This function is relatively straightforward. The nested `for` loops iterate over every

possible pair of neighbors, and the `if` statement checks for a link between unique neighbors. However, this process effectively counts every link twice, so at the end we divide by `numNeighbors * (numNeighbors - 1)` (i.e.,  $k(k-1)$ ), which is twice what we discussed previously.

**Reflection 11.19** *Do you see why the function counts every link twice? How can we fix this?*

The function effectively counts every link twice because it checks whether each neighbor is in every other neighbor's list of adjacent nodes. Therefore, for any two connected neighbors, call them *A* and *B*, we are counting the link once when we see *A* in the list of adjacent nodes of *B* and again when we see *B* in the list of adjacent nodes of *A*.

To count each link just once, we can use the following trick. In the list of `neighbors`, we first check whether the node at index 0 is connected to nodes at indices 1, 2, ...,  $k-1$ . Then, to prevent counting a link twice, we never want to check whether any node is connected to node 0 again. So we next check whether node 1 is connected to nodes 2, 3, ...,  $k-1$ . Now, to prevent double counting, we never want to check whether any node is connected to nodes 0 or 1. So we next check whether node 2 is connected to nodes 3, 4, ...,  $k-1$ . Do you see the pattern? In general, we only want to check whether node *i* is connected to nodes  $i+1, i+2, \dots, k-1$ . (This is the same trick you may have seen in Exercise 7.5.6.) This is implemented in the following improved version of the function, with changes highlighted.

---

```
def clusteringCoefficient(network, node):
    """ (docstring omitted) """

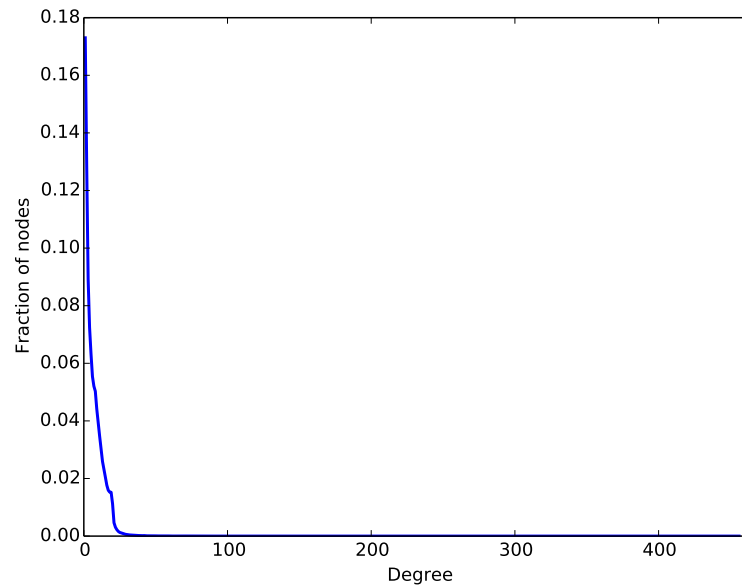
    neighbors = network[node]
    numNeighbors = len(neighbors)
    if numNeighbors <= 1:
        return 0
    numLinks = 0
    for index1 in range(len(neighbors) - 1):
        for index2 in range(index1 + 1, len(neighbors)):
            neighbor1 = neighbors[index1]
            neighbor2 = neighbors[index2]
            if neighbor1 != neighbor2 and neighbor1 in network[neighbor2]:
                numLinks = numLinks + 1
    return numLinks / (numNeighbors * (numNeighbors - 1) / 2)
```

---

Once we have this function, to compute the clustering coefficient for the network, we just have to call it for every node, and compute the average. We leave this, and writing a function to compute the average distance, as exercises.

### Scale-free networks

In addition to having short paths and high clustering, researchers soon discovered that most small-world networks also contain a few highly connected (i.e., high degree)



**Figure 11.8** The degree distribution of 875,713 nodes in the web network.

nodes called *hubs* that facilitate even shorter paths. In Figure 11.6, nodes 5, 12, and 18 are hubs because their degrees are large relative to the other nodes.

**Reflection 11.20** *How do connected hubs facilitate short paths?*

The existence of hubs in a large network can be seen by plotting, for each degree in the network, the fraction of the nodes that have that degree. This is called the *degree distribution* of the network. The degree distribution for a network with a few hubs will show the vast majority of nodes having relatively small degree and just a few nodes having very large degrees. For example, Figure 11.8 shows such a plot for a small portion of the (world wide) web. In the web graph, each node represents a web page and a directed link from one node to another represents a hyperlink from the first page to the second page.<sup>1</sup> In this network, 99% of the nodes have degree at most 25, while just a few have degrees that are much higher. (In fact, 98% of the nodes have degrees at most 20 and 90% have degrees at most 15.) These few hubs with high degree enable a small average distance and a clustering coefficient of about 0.37.

**Reflection 11.21** *In the web network from Figure 11.8, the degree of a node is the number of hyperlinks from that page. How do you think the degree distribution might change if we instead counted the number of hyperlinks to each page?*

Networks with this characteristic shape to their degree distributions are called *scale-free networks*. The name comes from the observation that the fraction of nodes

<sup>1</sup>Web network data obtained from <http://snap.stanford.edu/data/web-Google.html>



Figure 11.9 The North American routes of Delta Airlines. [12]

with degree  $d$  is roughly  $(1/d)^a$ , for some small value of  $a$ . Such functions are called “scale-free” because their plots have the same shape regardless of the scale at which you view them. A scale-free degree distribution is very different from the normal distribution that seems to describe most natural phenomena, which is why this discovery was so interesting.

**Reflection 11.22** How could recognizing that a network is scale-free and then identifying the hubs have practical importance?

The presence of hubs in a network is a double-edged sword. On the one hand, hubs enable efficient communication and transportation. For this reason, the Internet is structured in this way, as are airline networks (see Figure 11.9). Also, because so many of the nodes in a scale-free network are relatively unimportant, scale-free networks tend to be very robust when subjected to random attacks or damage. Some have speculated that, because some natural networks are scale-free, they may represent an evolutionary advantage. On the other hand, because the few hubs are so important, a directed attack on a hub can cause the network to fail. (Have you ever noticed the havoc that ensues when an airline hub is closed due to weather?) A directed attack on a hub can also be advantageous if we *want* the network to fail. For example, if we suspect that the network through which an epidemic is traveling is scale-free, we may have a better chance of stopping it if we vaccinate the hubs.

## Exercises

- 11.3.1\* Write a function that returns the average local clustering coefficient for a network. Test your function by calling it on some of the networks on the book website. You will need the function assigned in Exercise 11.1.7 to read these files.
- 11.3.2. Write a function that returns the average distance between every pair of nodes in a network. If two nodes are not connected by a path, assign their distance to be the number of nodes in the network (since this is longer than any possible path). Test your function by calling it on some of the networks on the book website. You will need the function assigned in Exercise 11.1.7 to read these files.
- 11.3.3\* The *closeness centrality* of a node is the total distance between it and all other nodes in the network. By this measure, the node with the smallest value is the most central (and perhaps most influential) node in the network. Write a function that computes the closeness centrality of a node. Your function should take two parameters: the network and a node. Test your function by calling it on some of the networks on the book website. You will need the function assigned in Exercise 11.1.7 to read these files.
- 11.3.4. Using the function you wrote in the previous exercise, write a function that returns the most central node (with minimum closeness centrality) in a network.
- 11.3.5. Write a function that plots the degree distribution of a network, producing a plot like that in Figure 11.8. Test your function on a small network first. Then call your function on the large Facebook network (with 4,039 nodes and 88,234 links) that is available on the book website. (You will need the function assigned in Exercise 11.1.7 to read these files.) Is the network scale-free?

## 11.4 RANDOM GRAPHS

---

Since small-world and scale-free networks seem to be so common, it is natural to ask whether such networks just happen randomly. To answer this question, we can compare the characteristics of these networks to a class of randomly generated graphs. In particular, we will look at the class of *uniform random graphs*, which are created by adding each possible edge with some probability  $p$ .

Creating a uniform random graph is straightforward. We first create an adjacency list with the desired number of nodes, then iterate over all possible pairs of nodes. For each pair of nodes, we link them with probability  $p$ , as shown below.

---

```
import random

def randomGraph(n, p):
    """Return a uniform random graph with n vertices.

    Parameters:
        n: the number of nodes
        p: the probability that two nodes are connected

    Return value: a random graph
    """
```

```

graph = { }
for node in range(n):           # label nodes 0, 1, ..., n-1
    graph[node] = [ ]          # graph has n nodes, 0 links

for node1 in range(n - 1):
    for node2 in range(node1 + 1, n):
        if random.random() < p:
            graph[node1].append(node2) # add edge between
            graph[node2].append(node1) # node1 and node2
return graph

```

Because we will get a different random graph every time we call this function, any characteristics that we want to measure will have to be averages over many random graphs with the same values of  $n$  and  $p$ . To illustrate, let's compute the average distance, clustering coefficient, and degree distribution for uniform random graphs with the same number of nodes and links as the graphs in Figures 11.6 and 11.7. Recall that those graphs had 24 nodes and 38 links.

**Reflection 11.23** *What parameters should we use to create a uniform random graph with 24 nodes and 38 links?*

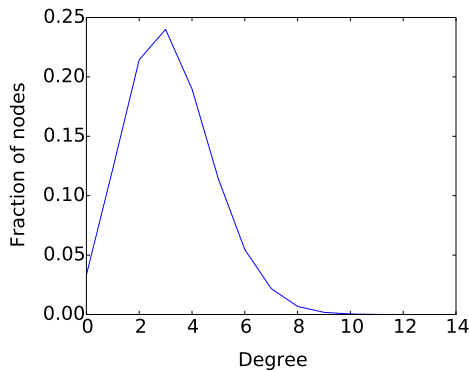
We cannot specify the number of links specifically in a uniform random graph, but we can set the probability so that we are likely to get a particular number, on average, over many trials. In particular, we want 38 out of a possible  $(24 \cdot 23)/2$  links, so we set

$$p = \frac{38}{(24 \cdot 23)/2} = \frac{38}{276} \approx 0.14.$$

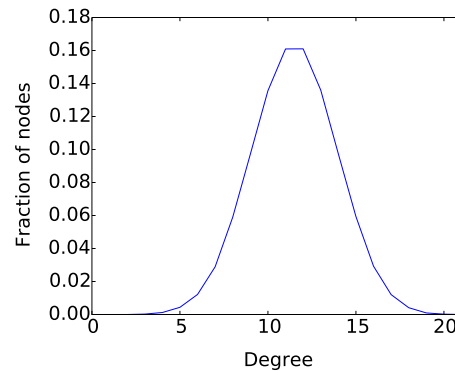
Averaging over 20,000 uniform random graphs, each generated by calling `randomGraph(24, 0.14)`, we find that the average distance between nodes is about 4.32 and the average clustering coefficient is about 0.12. The table below compares these results to what we computed previously for the other two graphs.

Graph	Average distance	Clustering coefficient
Figure 11.6 (clusters)	2.42	0.59
Figure 11.7 (grid)	3.33	0
Uniform random	4.32	0.12

The random graph with the same number of nodes and edges has a slightly longer average distance and a markedly smaller clustering coefficient than the graph in Figure 11.6 with the three clusters. Because these graphs are so small, these numbers alone, while suggestive, are not very strong evidence that random graphs do not have the small-world or scale-free properties. So let's also look at the average degree distribution of the random graphs, shown in Figure 11.10. The shape of the degree distribution is quite different from that of a scale-free network, and is much closer to a normal distribution. Because the probability of adding an edge was relatively low, the average degree was only about 3 and there were a number of nodes with degree



**Figure 11.10** The degree distribution of random graphs with  $n = 24$  and  $p = 38/276$ .



**Figure 11.11** The degree distribution of random graphs with  $n = 24$  and  $p = 1/2$ .

0, causing the plot to “run into” the  $y$ -axis. If we perform the same experiment with  $p = 0.5$ , as shown in Figure 11.11, we get a much clearer bell curve. These distributions show that random graphs do not have hubs; instead, the nodes all tend to have about the same degree. So there is definitely something non-random happening to generate scale-free networks.

**Reflection 11.24** *What kind of process do you think might create a scale-free network with a few high-degree nodes?*

The presumed process at play has been dubbed *preferential attachment* or, colloquially the “rich get richer” phenomenon. The idea is relatively intuitive: popular people, destinations, and web pages tend to get more popular over time as word of them moves through the network.

### Exercises

- 11.4.1\* Show how to call `randomGraph` to create a uniform random graph with 30 nodes and 50 links, on average.
- 11.4.2\* Exercise 11.3.1 asked you to write a function that returns the clustering coefficient for a graph. Use this function to write another function
 

```
avgCCRandom(n, p, trials)
```

 that returns the average clustering coefficient, over the given number of trials, of random graphs with the given values of  $n$  and  $p$ .
- 11.4.3. Exercise 11.3.2 asked you to write a function that returns the average distance between any two nodes in a graph. Use this function to write another function
 

```
avgDistanceRandom(n, p, trials)
```

 that returns the average of this value, over the given number of trials, for random graphs with the given values of  $n$  and  $p$ .
- 11.4.4. Exercise 11.3.5 asked you to write a function to plot the degree distribution of a graph and then call the function on the large Facebook network on the book

website. This network has 4,039 nodes and 88,234 links. To compare the degree distribution of this network to a random graph of the same size, write a function

```
degreeDistributionRandom(n, p, trials)
```

that plots the average degree distribution, over the given number of trials, of random graphs with the given values of  $n$  and  $p$ . Then use this function to plot the degree distribution of random graphs with 4,039 nodes and an average of 88,234 links. What do you notice?

- 11.4.5. We say that a graph is *connected* if there is a path between any pair of nodes. Random graphs that are generated with a low probability  $p$  are unlikely to be connected, while random graphs generated with a high probability  $p$  are very likely to be connected. But for what value of  $p$  does this transition between disconnected and connected graphs occur?

To determine whether a graph is connected, we can use either a breadth-first search (as in Exercise 11.2.5) or a depth-first search (as in Exercise 11.2.6). In either case, we start from any node in the network, and try to visit all of the other nodes. If the search is successful, then the graph must be connected. Otherwise, it must not be connected.

- (a) Write a function

```
connectedRandom(n, minp, maxp, stepp, trials)
```

that plots the fraction of random graphs with  $n$  nodes that are connected for values of  $p$  ranging from `minp` to `maxp`, in increments of `stepp`. To compute the fraction that are connected for each value of  $p$ , generate `trials` random graphs and count how many of those are connected using your `connected` function from either Exercise 11.2.5 or Exercise 11.2.6.

- (b) For  $n = 24$ , what do you find? For what value of  $p$  is there a 50% chance that the graph will be connected? Does the transition from disconnected graphs to connected graphs happen gradually or is change abrupt?

## 11.5 SUMMARY AND FURTHER DISCOVERY

---

In this chapter, we took a peek at one of the more exciting interdisciplinary areas in which computer scientists have become engaged. Networks are all around us, some obvious and some not so obvious. But they can all be described using the language of *graphs*. The shortest path and distance between any two nodes in a graph can be found with the *breadth-first search* algorithm. Graphs in which the distance between any two nodes is relatively short and the *clustering coefficient* is relatively high are called *small-world networks*. Networks that are also characterized by a few high-degree hubs are called *scale-free* networks. Scientists have discovered over the last two decades that virtually all large-scale natural and human-made networks are scale-free. Knowing this about a network can give a lot of information about how the network works and about its vulnerabilities.

### Notes for further discovery

This chapter's epigraph is from a 1967 article by Stanley Milgram titled, *The Small-World Problem* [40]. Dr. Milgram was an influential American social psychologist. Besides his small world experiment, he is best known for experiments in which he demonstrated that ordinary people are capable of disregarding their consciences when instructed to do so by those in authority.

There are several excellent books for a general audience about the emerging field of network science. Three of these are *Six Degrees* by Duncan Watts [66], *Sync* by Steven Strogatz [63], and *Linked* by Albert-László Barabási [3]. *Think Complexity* by Allen Downey [13] introduces slightly more advanced material on implementing algorithms on graphs, small-world networks, and scale-free networks in Python.

### \*11.6 PROJECTS

This section is available on the book website.