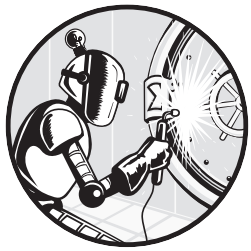
3

SECURING NETWORKS WITH GRAPH THEORY



Graph theory is a powerful but often overlooked tool in a security analyst's arsenal. A *graph* is a mathematical structure that shows relationships (called *edges* or *connections*) between things (called *nodes* or *vertices*), and graph theory provides a suite of algorithms for analyzing the structure and importance of these different, often interlinked, relationships. As technical a topic as security is, at its core it's about relationships: between computers and networks, users and systems, pieces of information, and so on. By modeling a computer network or social network as a graph, you can examine the composition of the relationships to determine, for example, which computers are integral to a business's communications, or which employees are most likely to forward a spam message, and to whom. Knowing which nodes (machines or employees) pose the greatest risk allows you to intelligently distribute your security resources.

This chapter starts by discussing the diverse applications of graph theory to information security, then goes over the theory itself. We'll cover types of graphs, how to create them efficiently in Python, and some interesting measurements you can perform on them. Chapters 4 through 6 then walk you through applying what you've learned here to analyze computer and social networks, the two types of network you'll face most often as a security engineer. We'll answer questions like which computers in a network received the most data, which members in a group are most influential, and how quickly information is likely to spread through a social network.

Graph Theory for Security Applications

Before we discuss how we can apply graph theory in practice, let's take a look at a simple travel graph example in Figure 3-1.

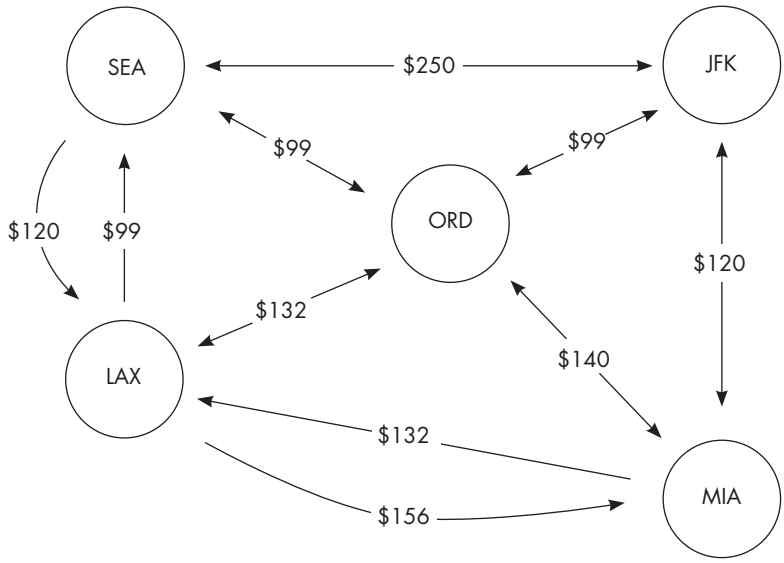


Figure 3-1: A travel graph

As mentioned previously, a graph is defined by nodes and edges. In this example, the nodes (the circles) represent airports in major cities, and the edges (the arrows) represent the cost of a plane ticket between two cities. A graph like this could save you money as you plan a trip. For example, if you wanted to travel from Seattle to New York, you could fly from SEA to LAX, then to MIA, and finally to JFK for a total cost of $396.00. You could also fly from SEA to ORD and then to JFK for $198.00, or directly from SEA to JFK for $250.00.

I don't know about you, but when I fly I don't think just about the cost; I also consider the travel time. In addition to the cost of each potential trip, you can also use this graph to determine the smallest number of stops between any two cities. Fewer stops means shorter trips. As you can see, even a simple graph can contain a lot of information.

When analyzing a computer network, the first step for both attackers and defenders is to get "the lay of the land." That means they can't start to attack or defend anything until they've built a graph of what's available around them. One way to create such a graph is to define computers as nodes and the network connections as edges; this is typical of most network maps. In one of the upcoming projects, we'll model a computer network from a raw packet capture. In this definition, the nodes will be the individual computers, and the edges will indicate when one machine sends packets to another.

Similarly, analyzing a social network can reveal key people and relationships, like the employees who will forward spam messages, or important members of a criminal organization. You can use that information to target or protect members of the network (depending on your job). For example, the FBI uses undercover agents to get information on organized crime families, then builds a social network graph, determines the key figures, and attempts to arrest them. Now, with the prevalence of social media, any amateur sleuth with a laptop can build an alarmingly accurate graph of an organization's (or individual's) social network and use that information to target key members for their own means.

Researchers also apply graph theory, using it to map technologies like cellular networks and cloud computing. For example, scholars have presented ways to apply shortest-path algorithms (similar to limiting the number of airport stops in Figure 3-1) to pick secure routes through graphs representing 5G cellular networks. The research analyzes how messages travel from point to point in the physical layer (OSI model) of the network.[1] We'll use a similar analytical model in Chapter 4 when we graph a computer network from packet data. Other modern research has focused on graphing the logical relationship of components hosted in the cloud. By mapping code usage and typical hypervisor loading activities, scientists have presented a formal way to describe cloud security concerns for virtualization platforms.[2]

Graph theory is also applied to *open source intelligence (OSINT)*, which, in short, collates publicly available information to gain intelligence about a target. An application named Maltego crawls the public web for related terms, email addresses, places, machines, and other details, and creates a graph of where they appear online, like in Figure 3-2. In 2017 at DEF CON, the annual information security convention, Andrew Hay gave an excellent introductory presentation on applied graph theory for OSINT.

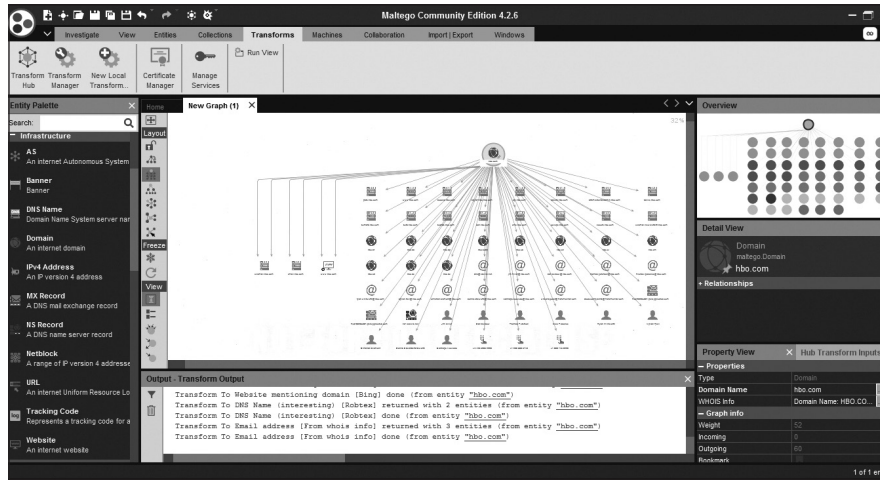


Figure 3-2: An intelligence-gathering application of graph theory

Applications like Maltego blend these logically different types of networks together in one graph, leading to very interesting insights. In one example, my team was able to locate a covert communication channel between two users of different forums. Although the forums were run by different companies, they resided on a shared hosting server. User A joined site X while user B joined site Y. Then, by manipulating the forum software, the two users were able to use local file reads and writes to pass messages on the underlying server. Had my team examined only the social network connections, we would have been stumped, but when we combined information about the social networks and the underlying machine networks, we realized that both accounts could access the same hardware. Of course, you don't need to rely on other people's tools; once you know the inner workings, you can produce your own OSINT-gathering tools, complete with pretty, yet functional, graph displays.

Graphs can also be used to describe how you can go from one condition to another by taking some action. For example, you can go from somewhat secure to completely unsecured by removing the locks from your doors. In this definition, secured and unsecured are called *states*, and removing the locks is the *action* that changes you from one state to another, known as a *transition*. Figure 3-3 shows such a graph, known as a *state machine graph*, that describes the potential for an attacker to move through an environment. Chapter 6 will cover state machines in detail.

You interpret the graph like so: if you're on the internet, and you want to take over an employee's system at your target organization, you can try phishing their customer service team. When you get a willing employee, you send them a remote-controlled malicious payload. You would then be on the employee system, but you might still need to perform some form of privilege escalation to completely take over the system. You can also see that this is just one path you could follow to achieve your objective.

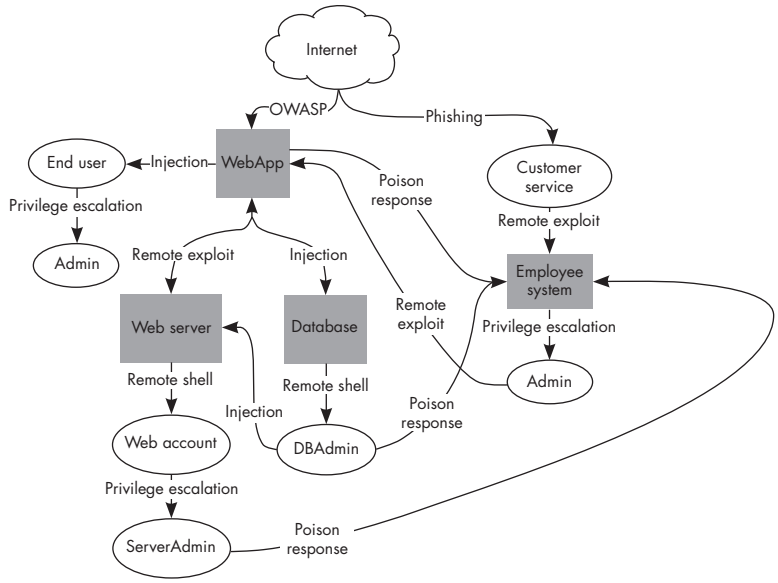


Figure 3-3: A state machine graph

Now that you have an idea of what to do with graph theory, let's discuss the math behind it.

Introduction to Graph Theory

A graph G comprises the set of nodes V and the set of edges E. Information travels between nodes along a set of nonrepeating edges that connect them, called a *path*. A node may forward information to any node it's directly connected to by an edge; the receiving node is the *neighbor* of the sending node. By convention I'll denote an edge as a tuple (u, v) containing an origin node u and a terminal node v, where both u and v are in V and are unique (not equivalent). We can write this in set notation as follows:

$$E \subseteq (u, v) \in V^2 \wedge u \neq v$$

Sometimes an edge in a graph points back to the same node (breaking my $u \neq v$ assumption); this is known as *self-looping*. For example, if you create a graph of function calls in a program that contains a recursive function, there will be an edge that leaves the recursive function and points directly back to it. Self-loops don't show up too often, but when they do they complicate the analysis and require specialized algorithms, so I recommend leaving them alone until you're very comfortable with the basics of graph theory.

Depending on the type of graph, edges may be bidirectional (*undirected graphs*) or directional (*directed graphs*). If the direction of communication is important to the question at hand, use a directed graph. Otherwise, use

an undirected graph. In practical implementation, undirected graphs are usually faster to work with since you assume $(u, v) = (v, u)$. A lot of problem descriptions require directed edges, though. In the travel graph from Figure 3-1, flying from LAX to MIA has a different cost than flying from MIA to LAX, so we need to use directed edges between those two nodes to capture the directional information.

An edge may contain *edge attributes*, additional pieces of information beyond the two nodes it connects. Nodes may also contain additional information called *node attributes*. When one of these attributes is used in ranking a node or edge, it's referred to as the node's or edge's *weight*, and a graph with weights is called a *weighted graph*. In some cases you may even need to add more than one edge connecting two nodes (called *edge multiplicity*) to account for different edge attributes or weights. I'll cover edge multiplicity in its own section later in the chapter, but for now we can extend the travel graph from Figure 3-1 for a simple example. Suppose we find out there are multiple flights leaving SEA for LAX. We may choose to add an edge for each additional flight, along with its cost as the weight. Adding these edges for all city pairs would give us a sense of which airports had the most travel options for our trip. We'll use multiple edges, edge attributes, and weighted graphs to inform our investigations in meaningful ways in the following chapters.

Simple graphs are unweighted, undirected graphs containing no self-looping or edge multiplicity. *Nonsimple* graphs (or, less commonly, *pseudographs*), those that do contain self-loops or multiple edges, comprise the vast majority of interesting graphs you'll encounter in practice.

A *cycle* of a graph G is a nonempty subset of E that forms a path such that the first node of the path corresponds to the last, and no other node is repeated along the path. This is a fancy way of saying a path that forms a closed loop. A self-loop is a special case of a graph cycle with a strict path length of 1. A *cyclic graph* is one that contains at least one graph cycle. A graph that is not cyclic (has no loops) is *acyclic*.

Before we go any further into the theory, let's walk through how to build one of these graph objects programmatically. In the next section we'll go over the current de facto standard library for Python graphs, NetworkX. Having access to the tools in this library will help you construct the examples in this book and play around with the theory at your own pace. The documentation for NetworkX also serves as a great reference to the theory that underlies each function.

Creating Graphs in NetworkX

You can use NetworkX (which contains implementations of most graph algorithms) and Pyplot (a part of the Matplotlib library) to generate and display an undirected graph. Listing 3-1 creates a graph with seven nodes and six weighted edges, then displays it.

```
❶ import networkx as nx
  from matplotlib import pyplot as plt

❷ G = nx.Graph()  # Create the default Graph object
❸ G.add_node('f') # Adds a node manually
  G.add_node('g') # Adds another node manually
❹ G.add_edge('a', 'b', weight=0.6) # Will add missing nodes
  G.add_edge('a', 'c', weight=0.2) # and connecting edges
  G.add_edge('c', 'd', weight=0.1) # Weight is one type of edge attribute
  G.add_edge('c', 'e', weight=0.7)
  G.add_edge('g', 'c', weight=0.8)
  G.add_edge('f', 'a', weight=0.5)
❺ pos = nx.layout.spring_layout(G, seed=42) # Try to optimize layout
  nx.draw(G, pos, with_labels=True, font_color='w')
  plt.show()
```

Listing 3-1: Creating a basic weighted undirected graph

First, we import the two libraries required to build and display the graph ❶. (Aliasing NetworkX as `nx` and Pyplot as `plt` is a common convention in examples online.) Then, we create a basic undirected graph with the NetworkX `Graph` constructor ❷. Defining a graph in this way returns an empty graph (with no nodes or edges).

To manually define the graph's structure, or *topography*, we can add either nodes or edges. To add a node to the graph, we use the `graph.add_node` function ❸ with an argument to use as an identifier (ID) for the node (during lookups, for example). In this case, the ID is the string literal `f`, but an ID can be any object that could act as a key to a Python dictionary (tuples, for instance). The `graph.add_edge` function, which takes two nodes and optional edge attributes as arguments, adds edges directly to the graph ❹. If either `a` or `b` (or both, as in this case) doesn't exist in the graph, NetworkX will helpfully add the missing node(s) before adding the edge. With directed graphs, the order in which you pass the nodes to `graph.add_edge` specifies the edge's direction: the edge starts at the first node and concludes at the second.

The real strength of graphs lies in their visual interpretations, as humans very often can detect patterns in information visually that they wouldn't have found otherwise. NetworkX supports several options for displaying graph information, including Matplotlib and Graphviz. For this example, we lay out the graph with one of NetworkX's built-in layout functions, `nx.layout.spring_layout` ❺, which uses a physics model of spring motion to position the nodes. The nodes' initial positions are randomly generated, but you can pass in the `seed` argument to make the image reproducible, which can be important if you want to share the conclusions from your research with others. The resulting node positions are stored in the dictionary `pos`, with structure {`node ID: (`*`x-coordinate, y-coordinate`*`)`}. The function `nx.draw` creates a plot object using these node positions, and Matplotlib displays the resulting figure. The additional parameters

to `nx.draw`, `labels` and `font_color`, control the look of the graph, shown in Figure 3-4.

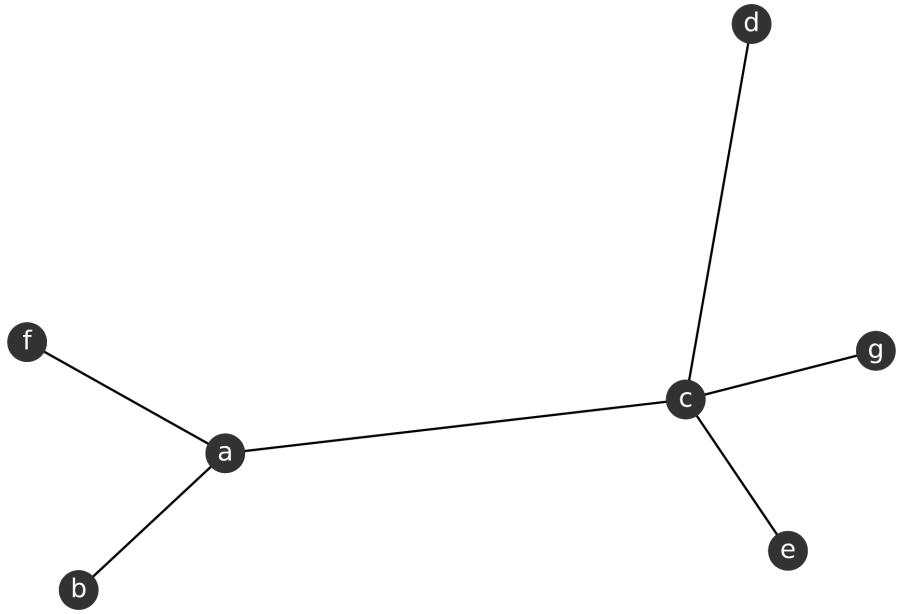


Figure 3-4: An undirected graph

If you remove the `seed` parameter and rerun the code your graph may look different, but it's guaranteed to be mathematically equivalent to the one in Figure 3-4.

Now that we have a way to codify and visualize graphs, let's look at some interesting measurements you can use in your analysis.

Discovering Relationships in Data

In this section, we'll examine a few of the most-used graph properties that can give us insight into the underlying relationships in our data. These properties are expressed as statistical relationships, such as the ratio of the number of possible paths between two nodes to the total number of paths in the graph. Typically we're interested in learning things like which nodes are isolated from other nodes, what are the shortest or longest possible paths between nodes, and how many different nodes can be reached from a particular starting node. There are dozens of possible graph properties to explore, but some are suitable only for certain types of graphs, while others are specialized use cases of these more general properties. The ones described here will give you everything you need to understand the projects in the next three chapters, but it's by no means a complete list.

Measuring Node Importance

A key concept in security is measuring the importance of different assets, be they human or machine, and the impact that compromising them may have on the operation of the organization as a whole. To do so, we need a way to measure which nodes are in critical positions. *Closeness* measures the connectivity of two nodes relative to the other connections in the graph.

NOTE *Closeness can have many interpretations. In Chapter 11 we'll look at how to use physical closeness (the distance between two objects in a physical space) to plan incident response. By selecting security personnel who are already in the vicinity of an incident, you can drastically reduce the reaction time in a lot of cases.*

When you apply closeness across all nodes in the graph, you're measuring some type of *centrality* (roughly, "importance") for each node. There are several types of centrality defined for graphs. The proper one to use depends on the behavior and structure of the network you're trying to analyze.[3] Sometimes you won't know in advance which measure of centrality makes the most sense for your problem. In these cases, start with simpler metrics (like closeness centrality) and move on to testing with other, more complex ones. We'll cover two types of centrality: betweenness centrality and degree centrality.

Finding Nodes That Facilitate Connections

Betweenness centrality considers nodes that connect other nodes together as more central to the graph. Consider a computer network like the one in Figure 3-5, where some systems act as proxies to connect users to databases.

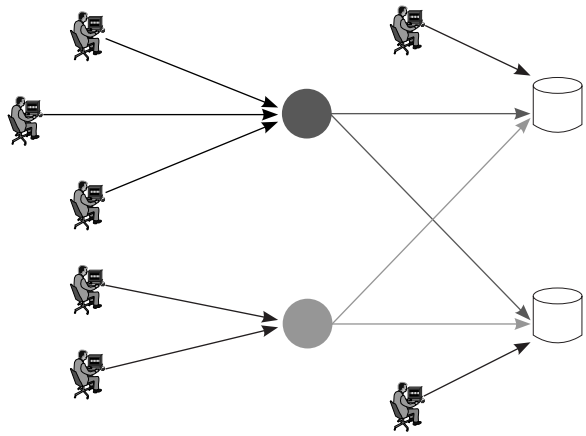


Figure 3-5: A simple proxy network

Betweenness centrality rates the gray proxy nodes much higher than any of the other nodes (the users and databases), since five of the seven users must connect to one of the two proxies to reach either database. The light gray circle at the top is between six paths (3 users × 2 databases = 6 paths)

and the dark gray circle at the bottom is between four paths (2 users × 2 databases = 4 paths). The centrality is further strengthened by the fact that these five users must pass through their respective proxy to reach the databases.

Formally speaking, betweenness centrality of a node u is the sum of ratios of all shortest paths from node s to node t, which pass through node u (noted as $\sigma_{(s,\ t)}(u)$), compared to the total number of shortest paths between node s and node t (noted as $\sigma_{(s,\ t)}$) for all paths where $s \neq u \neq t$. Putting it all together looks like this:

$$Betweenness\ (u) = \sum_{s \neq u \neq t} \frac{\sigma_{(s,t)}\ (u)}{\sigma_{(s,t)}}$$

The betweenness scores can be normalized to the number of nodes in G. The normalization function is $2\ /\ ((n-1)(n-2))$ for undirected graphs, and $1\ /\ ((n-1)(n-2))$ for directed graphs (where n is the number of nodes in the graph). The difference is due to the impact of directionality on the normalization. For undirected graphs, adding an edge between two nodes affects the betweenness score of both nodes and therefore carries twice as much influence as the same edge in a directed graph that impacts only one node (the source node). The normalization scores for both a directed and an undirected graph with five nodes each are calculated as follows:

$$undirected = \frac{2}{((4)(3))} = \frac{2}{12} = 0.166$$

$$directed = \frac{1}{((4)(3))} = \frac{1}{12} = 0.083$$

Unlike some other measures of centrality (such as closeness centrality), normalizing betweenness centrality is optional in NetworkX, and is specified by the Boolean keyword argument `normalized=True`. Listing 3-2 shows how we can retrieve the betweenness centrality scores for the map generated in Listing 3-1.

```
b_scores = nx.betweenness_centrality(G, normalized=True)
nx.set_node_attributes(G, name='between', values=b_scores)
print(G.nodes["c"]["between"])
```

Listing 3-2: Betweenness centrality for the graph created in Listing 3-1

The normalized result for the example graph should be approximately 0.8. There are a total of 15 shortest paths between all node pairs in Figure 3-4 (excluding pairs with c as the start or end node). Of those 15, 12 paths pass through c at some point ($12\ /\ 15 = 0.8$). The Jupyter notebook for this example shows how you can manually calculate the betweenness score by looping over the node pairs and counting the number of shortest paths that contain the target node.

NOTE *The mathematical definition of betweenness centrality just shown and the algorithm used by NetworkX to compute the betweenness centrality score are both from a paper by mathematician Ulrik Brandes, which has a lot of useful information on the theory of closeness and betweenness.*

Betweenness centrality has many applications within information security and network analysis because it represents the degree to which a node facilitates communication between other nodes. For example, a node in a computer network with high betweenness centrality would have more control over the network traffic, because more packets will eventually pass through it. For this reason, betweenness centrality can also be used to identify good places to perform inspections on network traffic. Another application is understanding critical points of failure in social networks, which we'll discuss more in Chapter 6.

Measuring the Number of Node Connections

Centrality can also be measured by how many neighbors a node has; this is known as *degree centrality*. Intuitively, degree centrality favors nodes that have a larger number of connections to other nodes in the graph. (Betweenness centrality can be seen as a specific measure of degree centrality.)

For undirected graphs, degree centrality is calculated as the fraction of all nodes that are connected directly to a node u. You'll commonly see the neighbors of a node u annotated as $\Gamma_{(u)}$.

$$Degree(u) = \frac{|\Gamma_{(u)}|}{|V|-1}$$

Remember from the math primer at the start of the book that the absolute value of a set of nodes ($|V|$, for example) is the same as the number of nodes in the set. We subtract 1 from the length of V to account for the fact that node c cannot be a neighbor of itself. Undirected degree centrality is calculated using `nx.degree_centrality`, as shown in Listing 3-3. The bold areas show the few changes required from Listing 3-2.

```
d_scores = nx.degree_centrality(G)
nx.set_node_attributes(G, name='degree', values=d_scores)
print(G.nodes["c"]["degree"])
```

Listing 3-3: Degree centrality with changes from Listing 3-2 in bold

The output of this code should be approximately 0.66, meaning node c is neighbors with two-thirds of the total number of nodes in the graph. In Figure 3-4 you can see that node c has four neighbors and, excluding c, there are a total of six nodes that could be neighbors of c. That gives us 4 / 6 = 2 / 3 = 0.66, which matches the `nx.degree_centrality` result.

For directed graphs, the degree centrality measure gets split in two pieces. The first deals with edges leading into a node, aptly named *in-degree centrality*. The second measure deals with edges leading out of a node, called *out-degree centrality*. The calculation for each is the same as for degree centrality, except that it considers only the subset of edges matching the specified direction. We denote these sets of edges as

$$(u \rightarrow) = E_{(u,)}$$

for out-degree and

$$(u \leftarrow) = E_{(,u)}$$

for in-degree. Listing 3-4 creates a directed version of the graph from Listing 3-1, then calculates in-degree and out-degree centrality for each node in the graph.

```
❶ G = nx.DiGraph() # Create the default Graph object
  G.add_edge('a', 'b', weight=0.6)
  G.add_edge('a', 'c', weight=0.2)
  G.add_edge('c', 'd', weight=0.1)
  G.add_edge('c', 'e', weight=0.7)
  G.add_edge('g', 'c', weight=0.8)
  G.add_edge('f', 'a', weight=0.5)
❷ i_scores = nx.in_degree_centrality(G)
❸ o_scores = nx.out_degree_centrality(G)
  nx.set_node_attributes(G, name='in-degree', values=i_scores)
  nx.set_node_attributes(G, name='out-degree', values=o_scores)
  print(G.nodes["c"]["in-degree"], G.nodes["c"]["out-degree"])
```

Listing 3-4: Creating a directed graph to measure in-degree and out-degree centrality

To make the graph directed, we replace the generator `nx.Graph` with `nx.DiGraph` ❶. Then, we use `nx.in_degree_centrality` ❷ and `nx.out_degree_centrality` ❸ to get their respective measures. The result of the code should be 0.33 for both values. If you examine the data, you'll see that node *c* has two incoming edges and two outgoing edges of the six total edges we defined. For each measure, then, the math works out to be 2 / 6 = 1 / 3 = 0.33. If you try running Listing 3-4 against an undirected graph, you'll get an error of the type `NetworkXError`, because in-degree and out-degree are specific to the `nx.DiGraph` and `nx.MultiDiGraph` objects.

NOTE *Several algorithms are implemented only for directed or undirected graphs in NetworkX, so if you use a statistical measurement that isn't supported for the type of edges in your graph, you'll need to implement the statistic for the other graph type yourself. We'll see an example of this in Chapter 6.*

The family of degree metrics allows us to specify the direction of information flow while calculating the scores, whereas both the closeness and betweenness measures make assumptions about the directionality. To understand why this matters, consider analyzing network traffic related to a *distributed denial-of-service (DDoS)* attack. A DDoS attack floods a network or specific target machine with more traffic than it is capable of processing, thereby blocking legitimate users' access. As packets travel from one system to the next, they create directed edges on the graph. A sudden increase of in-degree centrality would be seen at the target nodes, which could allow a script to automatically detect and respond to this threat. By including the direction of information flow, you can often provide more meaningful context to your graphs.

Analyzing Cliques to Track Associations

Investigators use *clique analysis* to track the associations of different groups who aren't kind enough to hand out membership lists. By collecting a list of who is talking to whom (and sometimes when), you can find interconnected clusters, or cliques. In theory, a clique β in graph G is any subset of V in which each node is adjacent to each other node in the set. Think of this as a group of friends who have all met each other, or a cluster of computers that are all connected. Some material may refer to these constructs as *complete subgraphs*. Figure 3-6 shows an undirected graph containing different cliques.

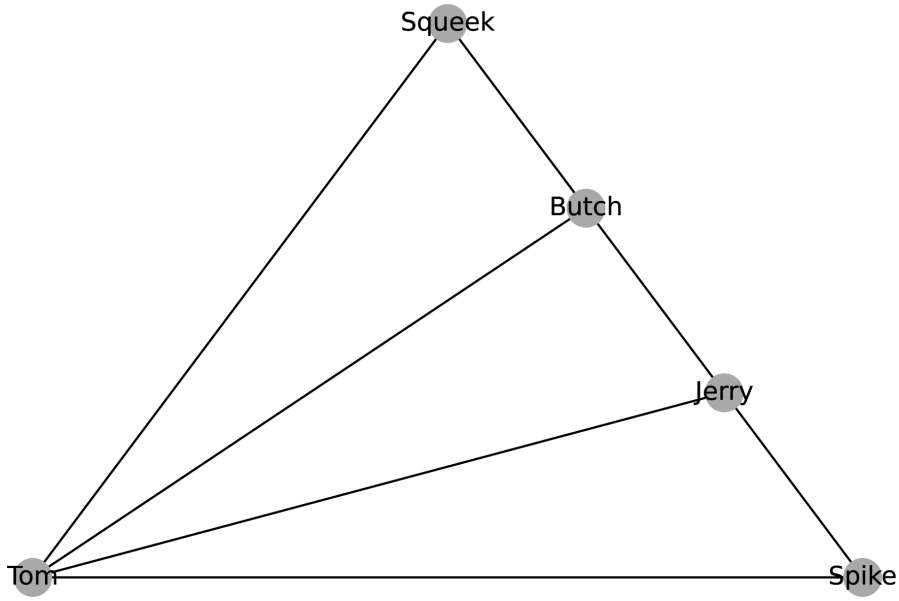


Figure 3-6: A cartoon character graph

A node may be in zero or more cliques. In the graph in Figure 3-6, for example, Tom is in three cliques: Tom, Spike, and Jerry; Tom, Butch, and Jerry; and Tom, Squeek, and Butch. Listing 3-5 creates the graph from Figure 3-6 and calculates a clique membership score for each node.

```
clique_graph = nx.Graph()
clique_graph.add_edges_from(
    [
        ("Tom", "Jerry"),("Butch", "Jerry"),("Spike", "Jerry"),
        ("Spike", "Tom"),("Tom", "Squeek"),("Tom", "Butch"),
        ("Squeek", "Butch")
    ]
)
clq = nx.algorithms.number_of_cliques(clique_graph)
tot = nx.algorithms.graph_number_of_cliques(clique_graph)
for m in clq:
    print(m, (clq[m]/tot))
```

Listing 3-5: Creating the cartoon clique graph in Figure 3-6

The call to `nx.algorithms.number_of_cliques` tallies the number of cliques each node belongs to, which you can use to easily find the node in the most cliques. To find the total number of cliques in the graph, we use `nx.graph_number_of_cliques`. We can then combine the number of cliques for each node and the total number of cliques to create a normalized score to determine which members of a network are key facilitators. The output from running this example code should be:

```
{'Tom': 1.0, 'Jerry': 0.66, 'Butch': 0.66, 'Spike': 0.33, 'Squeek': 0.33}
```

Tom is in every clique, Jerry and Butch are each in two-thirds of the possible cliques, and Spike and Squeek are only in one-third of the possible cliques. Clearly, Tom is the best-known member of this network. In social networks, like a company or an organized crime syndicate, the members in the most cliques are key to facilitating the operations. If we wanted to disrupt the activities of this organization, removing Tom would go a long way toward achieving that. You can also measure clique membership in your network to identify nodes that act as gateways between otherwise separate parts of the network.

The function `nx.algorithms.number_of_cliques` finds the number of *maximal* cliques to which each node belongs—that is, the largest group of nodes that are all connected to one another. In undirected graphs, any two adjacent nodes could be considered a clique, and, in any graph, cliques of four or more nodes contain cliques of three and two nodes, so working with maximal cliques takes those subcliques into account.

You can enumerate all the maximal cliques in a graph with the `nx.find_cliques` function, as shown in Listing 3-6.

```
cliques = list(nx.find_cliques(clique_graph))
print(cliques)
```

Listing 3-6: Creating a list of cliques from a directed graph

The result is a `Generator` object, which is a built-in object type in Python 3. You can either use it directly or cast it to a `list`. We'll see a practical application of finding cliques using `nx.find_cliques` in Chapter 5 when we build a social network graph from posts.

Determining the Connectedness of the Network

Graphs can be connected or disconnected. A *connected* graph is one where every node pair (u, v) has some connecting path (ρ). Therefore, a graph G is *disconnected* if any pair of nodes (u, v) doesn't have a connecting path (ρ) using any subset of edges in E. The only way to know whether or not a graph is connected is to check every pair of nodes to see if they're disconnected. We can write this out neatly using set notation and Boolean algebra:

$$Disconn(G) = \left(\sum_{(u,v)} \rho_{(u,v)} \not\subset E \right) > 0$$

A Boolean statement such as $\rho(u, v) \notin E$ returns `1` if it is true and `0` otherwise, so this equation technically counts all pairs of disconnected nodes. In practical implementations, we don't need to continue searching all remaining pairs of G because once we discover a missing edge, we know it's a disconnected graph. We can only say a graph is connected, however, once we've checked every pair of nodes and found no disconnected pairs. You can go through this exercise yourself with the graph from Figure 3-1 to determine if it's a connected or disconnected map.

Figure 3-7 shows the graph from Figure 3-6 extended to be a disconnected graph.

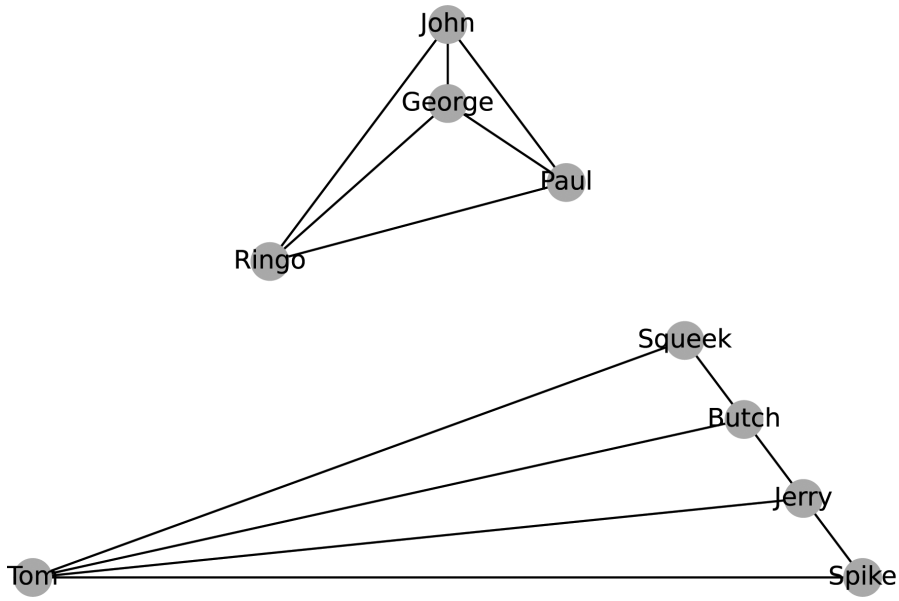


Figure 3-7: A disconnected graph

A disconnected graph is made up of two or more disparate sections called *connected components* (or just *components*). Formally speaking, a connected component of an undirected graph G ($\phi_i(G)$) is a subgraph in which every pair of nodes (u, v) is connected by a path $\rho(u, v) \in E$ (annotated $\rho(\phi_i, u, v)$ for a path in the ith component subgraph). Additionally, none of the nodes in ϕ may be connected to any additional nodes in the superset V. For example, the graph shown in Figure 3-7 has two connected components: one comprising cartoon characters, and the other comprising former bandmates.

NOTE *A clique can never extend beyond a single component, so you'll often care which component a clique is formed in, especially when performing social network analysis.*

Using Graph Edges to Capture Important Details

The final graph property we'll examine is one I mentioned earlier, edge multiplicity. This property is powerful once you know how to leverage the flexibility it brings to your analysis. In many practical instances, like the packet analysis project in the next chapter, there may be multiple edges between nodes that contain valuable information we want to keep for analysis.

For example, graphing the TCP handshake requires multiple directed edges between two nodes. The connecting machine (also called the *client*) sends a synchronization request to the target machine (a SYN packet), which creates one directed edge from u to v in the graph. The target machine then responds with an acknowledgment as well as a request of its own to synchronize (a SYN-ACK packet), which creates a directed edge from v back to u. (In an undirected graph, this response would count as a duplicate of the first edge.) Finally, the connecting machine sends the target machine its own acknowledgment packet (an ACK packet), which creates a second directed edge from u to v in the graph. Figure 3-8 shows two versions of the same graph data containing TCP handshakes between two different groups of systems.

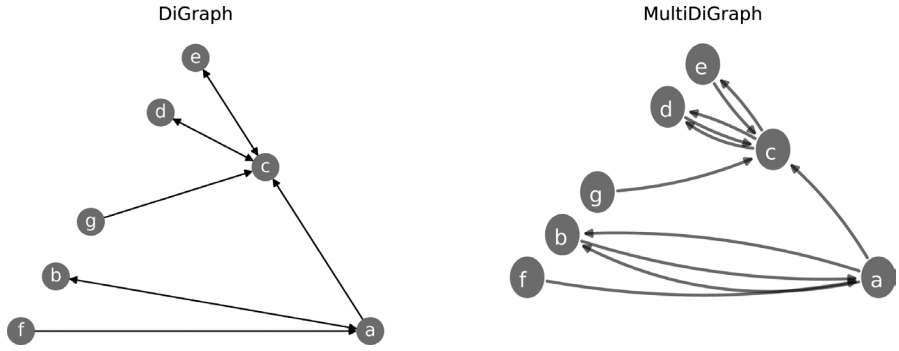


Figure 3-8: Comparing single- and multi-edge graphs

On the left is a standard `DiGraph` representation, which treats repeated communications between two nodes as a single directed edge. Examining this graph alone, you couldn't determine which nodes participated in a TCP handshake. On the right is a `MultiDiGraph` representation of the same data, which retains an edge for each occurrence of communication. Examining this graph, you can easily see that node c initiated a handshake-like exchange with node d. Node a also initiated a handshake with node b.

There are two schools of thought for dealing with edge multiplicity. The first school says you should summarize multiple edges based on their weight ω (and potentially other attributes) into a singular edge, like so:

$$\overline{(u \rightarrow v)} = \sum_{\forall (u \rightarrow v)} \omega$$

If the constituent edges are unweighted, the weight is the number of edges making up the composite:

$$\overline{(u \to v)}_{\omega} = |(u \to v)| \in E$$

This summation must take into account the directionality of edges when dealing with directed graphs. (If you're using complex values for edge attributes—such as ranges, which require specialized processing to summarize—you'll be better off implementing your own definition of edge summation in a function within your code.)

The second school of thought is to graph each edge individually and summarize edges only when it comes time to analyze them. Doing so allows you to retain more of the underlying structure. As an example, consider timestamp information on network packets. If you sum the edges into a single edge like the preceding example, you can't see the order in which the edges are created. Retaining each edge allows you to order their creation by timestamp and look for interesting patterns, like call and response pairs in edges.

There's no generally right or wrong way to handle edge multiplicity in a graph. The correct approach is often a little of both schools of thought, as we'll see in the next chapter.

Summary

The power of graph theory lies in the flexible interpretation of nodes and edges. Do nodes represent people, computers, cities, or something else entirely? Do edges measure physical distances or intangible relationships? The answer to all these questions is yes. Be warned, though: this freedom of perception is a double-edged sword. Because there are no strict definitions of what a node or edge represents, you can create a graph whose edges and analysis have no meaningful relationship to reality. An example would be using nodes that represent computers and edges that represent the physical distance between two cities where the computers are located. We typically don't think about how far a message travels on the internet in terms of physical distance, but rather in terms of the number of network "hops" it has to make before reaching its destination. Over the next three chapters, I'll explain the justification for different interpretations of information in more depth, because the meaning of a result, such as closeness centrality, relies on the meaning of an edge's weight and needs a bit of context to make sense.

There's a lot more useful theory than what I've covered here. The book *Introduction to Graph Theory* by Richard Trudeau (Dover, 2001) is an excellent resource.[4] If you're looking for a more advanced discussion, check out *Graph Theory and Complex Networks: An Introduction* by Maarten van Steen (author, 2010).[5] Both books make the topics easy to understand and the math easy to follow. For something more security focused, check out

the paper "Applied Graph Theory to Security: A Qualitative Placement of Security Solutions within IoT networks," published in the *Journal of Information Security and Applications* in December 2020, which uses graph theory to analyze the security of IoT network devices and determine suitable locations for monitoring device traffic.[6]

In the next two chapters, we'll put these theoretical concepts to work by examining both a computer network and a human social network to learn which nodes are important to the network, what information is being exchanged, and other important insights about the underlying graph structure. The final graph theory project, in Chapter 6, will give you the tools you need to simulate changes to a network over time. Once you understand the concepts and interpretations, the insights you can gain will make graph theory one of the most powerful and versatile weapons in your analytical arsenal.