Versatile Mathematics

COMMON MATHEMATICAL APPLICATIONS

Josiah Hartley
Frederick Community College

Val Lochman
Frederick Community College

Erum Marfani
Frederick Community College

2nd Edition

2020

Attributions This book benefited tremendously from others who went before and freely shared their creative work. The following is a short list of those whom we have to thank for their work and their generosity in contributing to the free and open sharing of knowledge.

- David Lippman, author of *Math in Society.* This book uses sections derived from his chapters on Finance, Growth Models, and Statistics. He also administers MyOpenMath, the free online homework portal to which the problems in this text were added.
- The developers of onlinestatbook.com.
- OpenStax College (their book *Introductory Statistics* was used as a reference)
  OpenStax College, *Introductory Statistics.* OpenStax College. 19 September 2013. <http://cnx.org/content/col11562/latest/>
- The authors of OpenIntro Statistics, which was also used as a reference.
- The Saylor Foundation Statistics Textbook: http://www.saylor.org/site/textbooks/Introductory%20Statistics.pdf

Thanks The following is a short list of those whom we wish to thank for their help and support with this project.

- The President's office at Frederick Community College, for providing a grant to write the first chapters.
- Gary Hull, who in his tenure as department chair gave us his full support and gave us the impetus to start the project, and generously shared his notes for MA 103.
- The entire FCC math department, who provided untold support and encouragement, as well as aid in reviewing and editing the text.

Contents

Chapter

8

Graph Theory

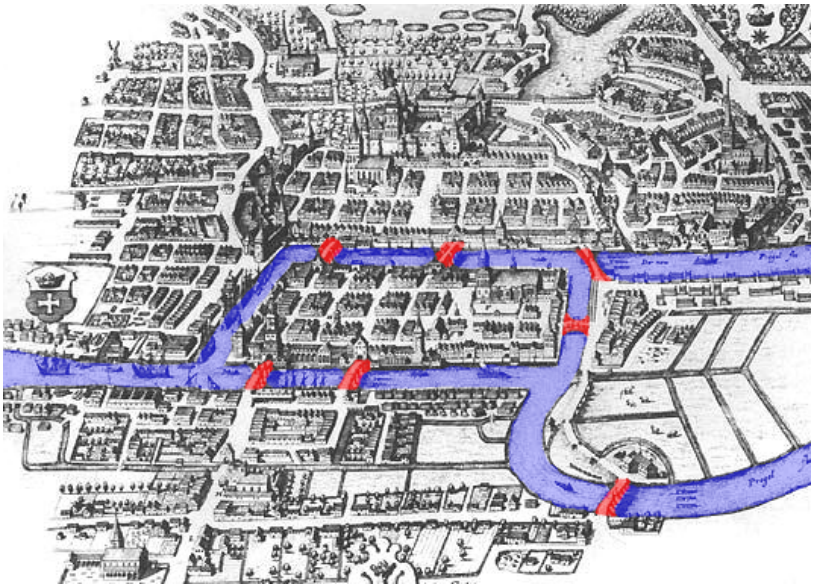
With billions of users, Facebook is probably the most successful social media company ever. Since their revenue is based on advertising, Facebook is completely focused on user engagement: attracting and keeping users, and getting users to sign in regularly and browse.

In order to accomplish this, they need to have a way to track relationships. New users are more likely to sign up if their friends have accounts, and current users are more likely to be engaged if their News Feed shows relevant content from the people they are closest to.

The way that social media companies track social networks is through the use of *graphs*, which resemble webs (it turns out that the Internet, the World Wide *Web*, is another example of a graph). By tracking connections between people, these companies can learn a tremendous amount of information. For instance, Facebook can suggest new friends for you, and they tend to indeed be people that you know, because Facebook can tell that you share many friends in common.

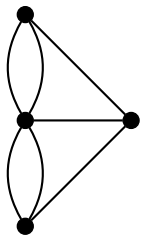
In this chapter, we will learn a few basic concepts related to these graphs, and we'll find how they can be used in a wide array of applications: any kind of network can be modeled in the same way as a friend group.

SECTION 8.1 Introduction to Graphs



In the early 1700s, the city of Königsberg in Prussia (now Kaliningrad, Russia), which was split by the Pregel River, had 7 bridges connecting the north and south banks of the city to two islands in the center of the river, as shown in the drawing above. A popular puzzle of the day challenged travelers to plan a walk through this part of the city in such a way that they would cross each bridge exactly once (you may want to pause and see if you can find such a path).

Leonhard Euler, in 1736, turned his attention to this problem. Although the problem doesn't seem to be an important one, what is interesting is how Euler solved it. He noticed that the path a person travels on a given land mass is irrelevant; all that matters is which bridges they cross. Thus, he decided to simplify the picture by simply drawing a dot to represent each land mass, and a line connecting two dots to represent a bridge. Here's the result:



That's the first example of a **graph**; while we use the word *graph* to mean several different things, in this chapter, a graph will refer to a diagram like the one above.

Graphs

A **graph**, in graph theory, consists of **nodes** (or vertices) and **edges**; each edge connects one node to another.

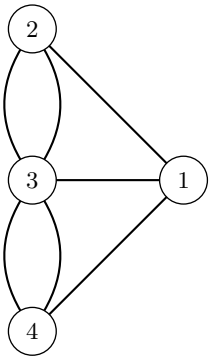
Now, notice that we've drawn the nodes in positions that roughly correspond to the orientation of the map; the one on the right represents the eastern island, for instance, while the two on the top and bottom of the left row represent the northern and southern banks.

Location is not important.

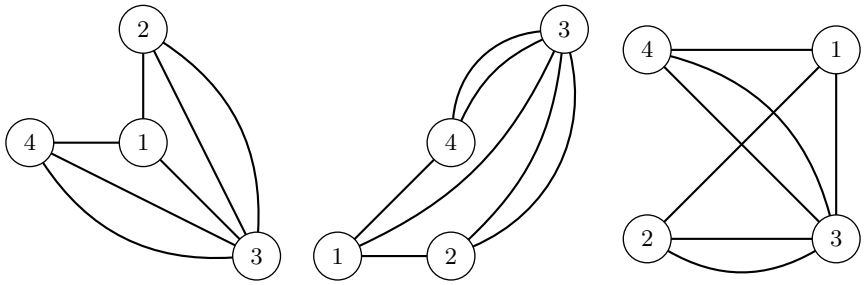
However, this is completely arbitrary; since we simplified the picture to land masses and connections, it turns out that the nodes can be rearranged at will, as long as the final result has the same pattern of connections, meaning that the same nodes are connected by edges.

Connections are what matters.

Before we illustrate this, let's label each node so that we can see them move:



Each of these graphs is equivalent, according to Euler's new theory, to the original one:



In fact, we could describe it in words, by saying something like

Of course, it's much more fun to draw pictures than to write something like this.

There are 4 nodes.
Node 1 is connected to nodes 2, 3, and 4.
Node 2 is connected to node 3 by two edges.
Node 3 is connected to node 4 by two edges.

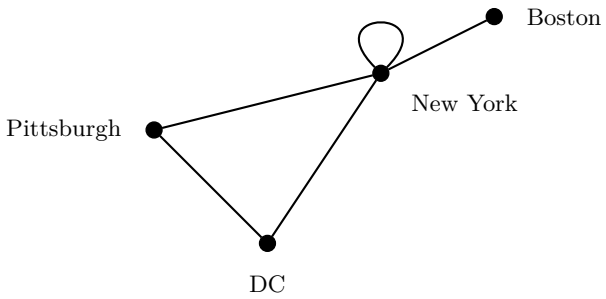
Any picture you could draw that fits this description would be equivalent to the first graph that we drew.

Examples and Definitions

Let's take a look at a few examples of graphs, and along the way, we'll encounter a few new terms that we can use to categorize and describe them.

Application 1

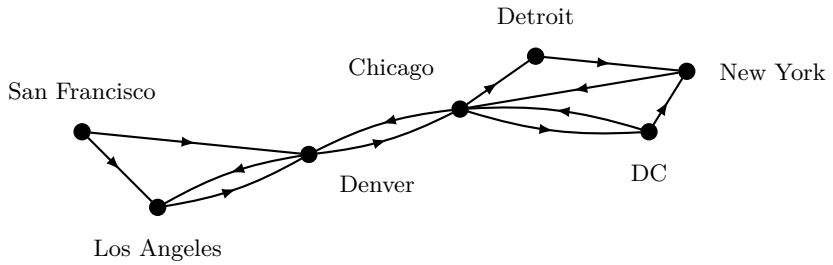
Transportation Network The graph below shows a simple train system; each edge indicates a line that runs between two cities.



Notice the **loop** at New York; this implies that there is a train line that returns to the same station from which it leaves. There could be, for instance, a sightseeing train that simply takes passengers on a short loop. We simply include it here to show the possibility of a loop, which is an edge that connects a node to itself.

Application 2

Communication Network This second example shows a fictional network, which could consist of connections between data centers.



Notice that the edges on this graph have arrows that indicate which direction the data can travel; for instance, data can flow from Detroit to New York, but to send data in the opposite direction, it would have to go from New York to Chicago, and then on to Detroit.

Also, this graph, unlike the one before it, has multiple edges between some pairs of nodes (we've seen this before, on the graph of Königsberg).

These two distinctions lead to our first set of definitions; the following terms can help to categorize graphs, and it is useful to think about new applications in terms of what category they fit into.

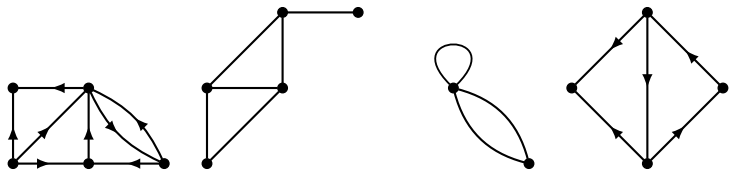
Graph Classifications

Simple Graphs and Multigraphs: a **simple graph** is one that has at most one edge between two nodes, and contains no loops; a **multigraph** can have multiple edges connecting the same pair of nodes, and may contain loops.

Undirected and Directed Graphs: an **undirected graph** has no notion of direction to the edges; in a **directed graph** (also called a **digraph**), each edge has an associated direction.

EXAMPLE 1 CLASSIFYING GRAPHS

For each of the following graphs, determine whether it is a simple graph or multigraph, and whether it is directed or undirected.



(a) (b) (c) (d)

Solution

(a) Since the edges have directions associated with them, and there is one pair of nodes that have multiple edges between them, this is a directed multigraph

(b) There are no multiple edges or loops, and no directions, so this is an undirected simple graph

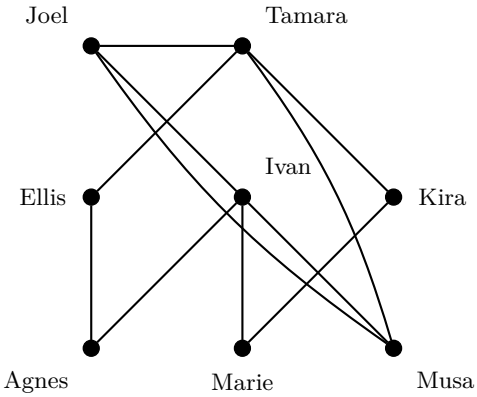
(c) There are no arrows, but because of the multiple edges and the loop, this is an undirected multigraph

(d) There are no multiple edges or loops, so this one is simple, but because of the arrows, it is a directed simple graph

Let's go back to examples of graph theory applications.

Application 3

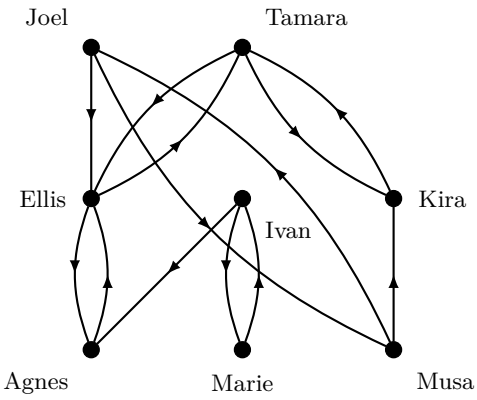
Social Network Suppose, for instance, that we selected a small group of people and checked whether they are connected on Facebook; if two people are friends, we draw an edge between them.



This is a simple graph, since it doesn't make sense to have more than one connection between two people; they are either friends or are not. Also, since Facebook friendship is a two-way relationship, this is an undirected graph.

If we wanted to make a directed multigraph for a social network, we could use Instagram, for instance, since one person can follow another without being followed back. The result might look like this:

Notice how complicated this is already, with only 8 people. Real social networks have incredibly complicated graphs; although we may not be able to draw them, computers can still analyze them to glean all sorts of information, like which users are the most influential.



Here, an edge indicates whether one person follows another, and the direction of the arrow goes from the follower to the one they follow.

Notice that one of the things we can glean from a graph like these is who in this social group is the most well-connected. The easiest way to do this is simply to count the number of connections that each person has to others, which corresponds to the number of edges that connect to that node. We have a special name for this count: we call it the **degree** of a node.

Degree

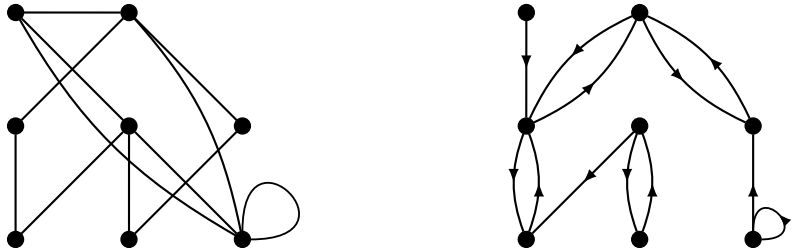
The **degree** of a node in an undirected graph is the number of edges that connect to it.

Note: a loop counts twice, since the loop connects to the node at both of its ends. This is mostly by convention, but it is important so that some concepts are consistent whether loops are included or not.

In a directed graph, each node has two degrees: the **in-degree** and **out-degree**. The in-degree counts how many edges come into that node; the out-degree counts how many edges leave from that node.

EXAMPLE 2 DEGREES OF NODES

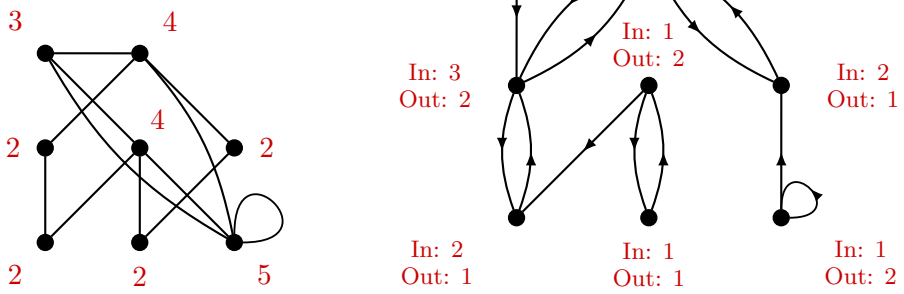
Find the degree of each node in the social network graphs shown here.



(a) (b)

Solution We simply need to count the number of edges that connect at each node. The only complication in part (a) is the loop on the lower-right node; remember that the loop adds a total of 2 the degree of that node. For part (b), pay attention to the direction of each edge; an edge adds one to the out-degree of its starting node and one to the in-degree of its ending node (note that a loop here contributes one each to the in-degree and out-degree of its node).

Here's the final result; each node is labeled with its degree(s):



(a) (b)

There are a couple of interesting observations we can make about node degrees.

Degrees: Observation 1

The sum of the degrees of all the nodes in an undirected graph is always even.

Similarly, the sum of all in-degrees and out-degrees of all the nodes in a directed graph is always even.

The reason for this is very simple: each edge contributes two degrees to the total for the graph, one degree at each end, so no matter how many edges there are, there will be twice as many degrees, and two times anything is always even. In a directed graph, each edge contributes one out-degree and one in-degree; thus, the total in-degrees and out-degrees will be equal, and when these are added, the result will also be even.

Degrees: Observation 2

An undirected graph has an even number of nodes with odd degree.

To verify this, we need to remember that the total number of degrees is even, and we need to know a principle of even numbers: **if you start with an even number, add something to it, and end up with an even number, the number you added must have been even**. In other words, if you add an even number and an odd number, the result will be odd.

Therefore, if we split the graph into even nodes and odd nodes, we can split up the total of all degrees (which we know is even) like this:

$$\text{degrees of even nodes} + \text{degrees of odd nodes} = \text{even}$$

Since the first part is even (if you add up a bunch of even degrees, the result is even), and the result is even, we know that the total number of degrees for all the odd nodes must be even also (that's the bolded principle in the last paragraph).

Think about adding together a bunch of odd numbers: when we add two of them, we get an even result, but adding a third makes the result odd. Adding a fourth makes it even again, and so on, so clearly there must be an even number of these nodes to make the result even.

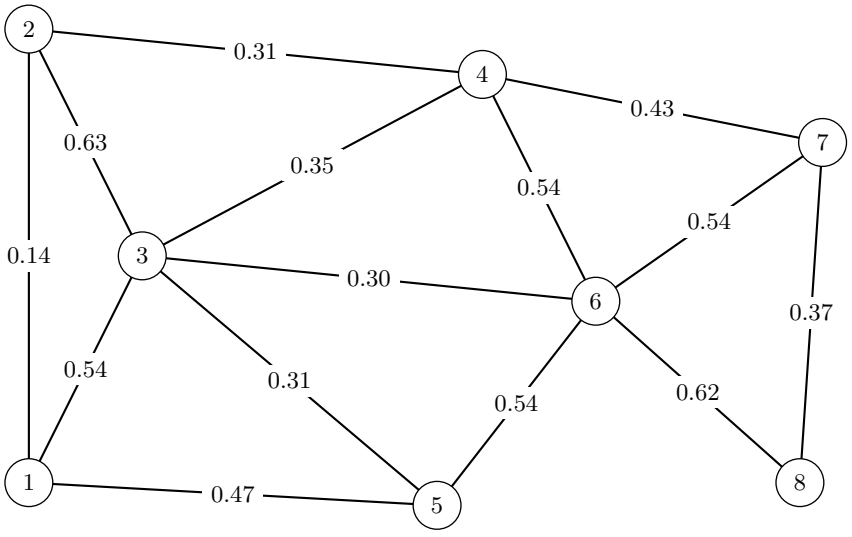
If that explanation wasn't clear to you, don't worry too much; this is simply an interesting principle that we can observe about graphs.

Back to examples!

Application 4

Street Map The graph below could represent a small segment of a street map, where each edge corresponds to a street, and each node corresponds to an intersection.

This is the way that Google Maps, for instance, stores mapping information. Notice the number written along each edge; these are called edge **weights**. These could represent the distance between each pair of nodes, or perhaps the time that it will take to cover that distance. Again, for services like Google Maps that track traffic information, these weights would be constantly updated to reflect current traffic data.



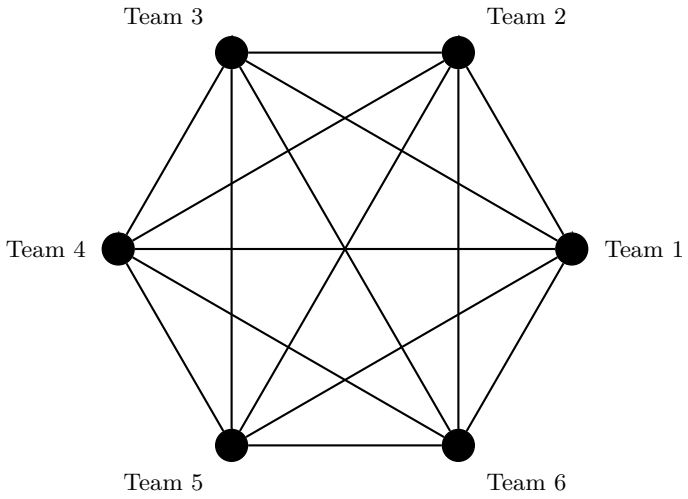
Weighted Graphs

A **weighted graph** has a weight associated with each edge; these weights can represent things like distance or cost.

Since the location of the nodes in a graph is not significant, weights can be used to encode distance information without having to worry about drawing a graph in such a way that it shows the distances between nodes.

Application 5

Round-Robin Tournament A tournament in which each team competes against every other team is called a *round-robin* tournament. If we draw a graph in which each node corresponds to a team and each edge represents a game played between two teams, it could look like the following.



This graph is interesting because every node is connected to every other node; we have a special name for graphs like this: we call them **complete graphs**.

Complete Graphs

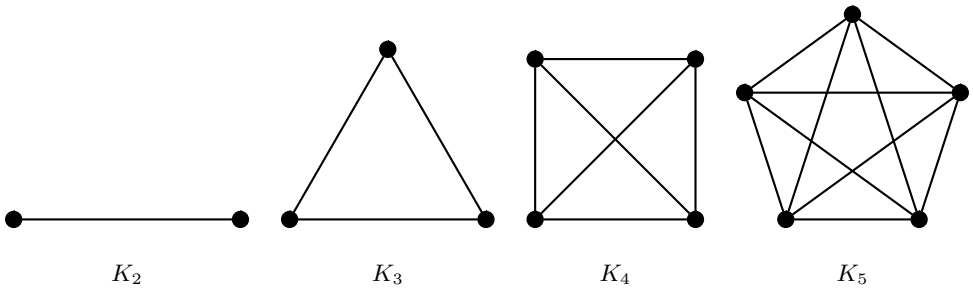
A **complete graph** is an undirected graph with an edge between every pair of nodes.

A complete graph with n nodes is often labeled K_n.

K_n has $\dfrac{n(n-1)}{2}$ edges.

Sidenote: there is an interesting probability question called the Birthday Problem, which deals with the probability that in a group of n people, at least two people will share the same birthday. It turns out that in a group of only 23 people, the probability is over 50%. This sounds surprising, because that seems like a very small group, but since the significant part is the *pairing* of people, we can think of this group as a complete graph with 23 nodes. By looking at the one above, you can imagine that K_{23} is much more complicated (with 253 edges). Now, recognizing that there are 253 pairings, the likelihood that one of these pairings will match birthdays is less surprising (there's more to the problem than this, but that's the core concept).

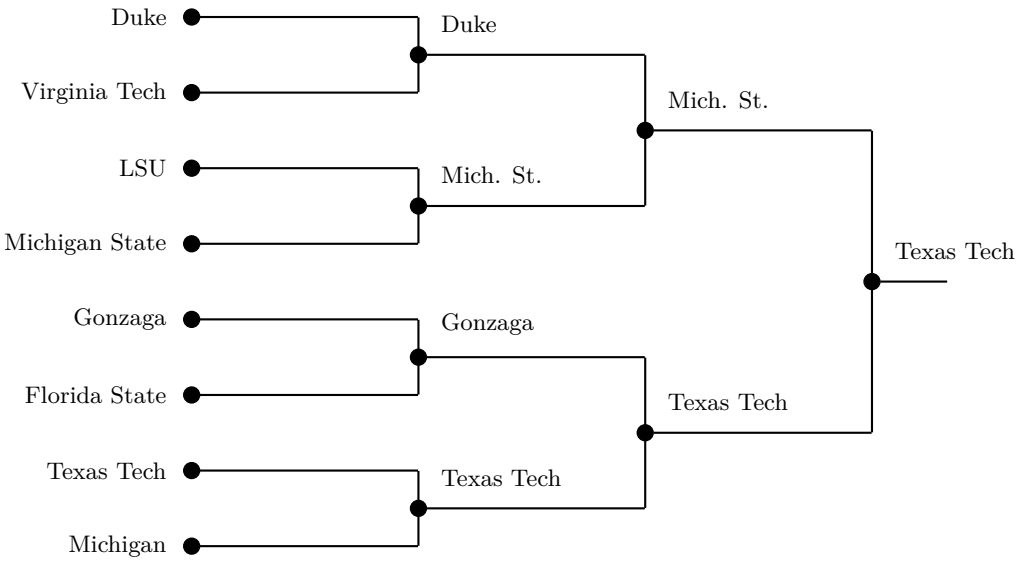
The first four complete graphs (K_2 through K_5) are shown below. For obvious reasons, K_1 is not interesting enough to show.



Application 6

Single-Elimination Tournament Another type of tournament, like most playoffs, is an **elimination** tournament, in which each team is paired up with another; the loser is knocked out, and the winner moves on to the next round to compete against the winner of another pairing. For simplicity, we can focus on single-elimination tournaments, where each competition consists of a single game (unlike the NBA playoffs, for instance, in which teams play a best-of-7-game series in each round).

The graph below shows a portion of the 2019 NCAA men's basketball tournament, from the Sweet Sixteen round onward in the West and East regions:



Notice that we could have drawn the full graph for the full bracket; instead, we drew only a *portion* of it. The graph above is a **subgraph** of the full tournament graph.

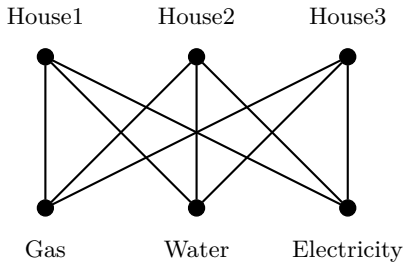
This graph also happens to be a *tree*, which we'll discuss more in the last section of this chapter.

Subgraphs

If you start with a graph, and remove some nodes and/or edges, the result is a **subgraph** of the original one (the original one is called a *supergraph* of the smaller one).

Application 7

Utility Connections Suppose we need to connect three houses to three utilities; the graph could look like the following.



It turns out that this is an example of a *bipartite graph*, which is one that can be split into two sets of nodes, where there are only connections between the two sets (none within each set). That's not what we'll focus on here, though.

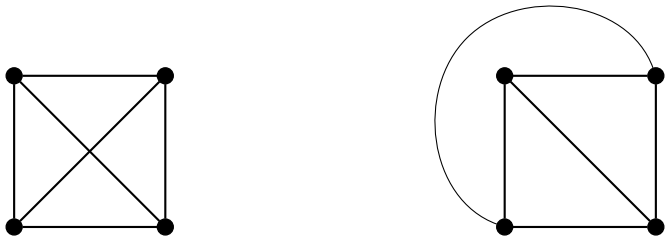
The question we'll think about is this one: is it possible to draw this graph in such a way that none of the edges cross? In practical terms, this would mean arranging the houses and utilities in such a way that burying the utility line would be simplified by not having to worry about the others while digging.

Planar Graphs

A **planar graph** is one that can be drawn without any edges crossing.

There are many applications of planar graphs; for instance, when designing a circuit board, engineers must lay out the connections in such a way that none of them cross, and highway engineers encounter a similar problem.

Let's take a look at a simpler example: we can show that K_4 is a planar graph by moving one of the edges to the outside. On the left below, we have the familiar representation of K_4, and on the right side, we have a planar representation for it:



Back to the example with the houses and utilities: it turns out that this example is *non-planar*. You can convince yourself of this by trying to draw its planar representation, but we won't show the full proof here for the sake of simplicity. In short, though, if you start with two houses and two utilities, that graph *is* planar, and you can add a third house without crossing any edges. Once you do, however, there's no region in which you can place the final utility in such a way that it can connect to all three houses without two edges intersecting.

We haven't exhausted the possibilities for applications, but hopefully the pattern is clear: whenever an application involves some sort of connection that can occur between two items, a graph can be used to model this situation. Once you start thinking about the world in this way, it becomes clear that the applications of graph theory are nearly unlimited.

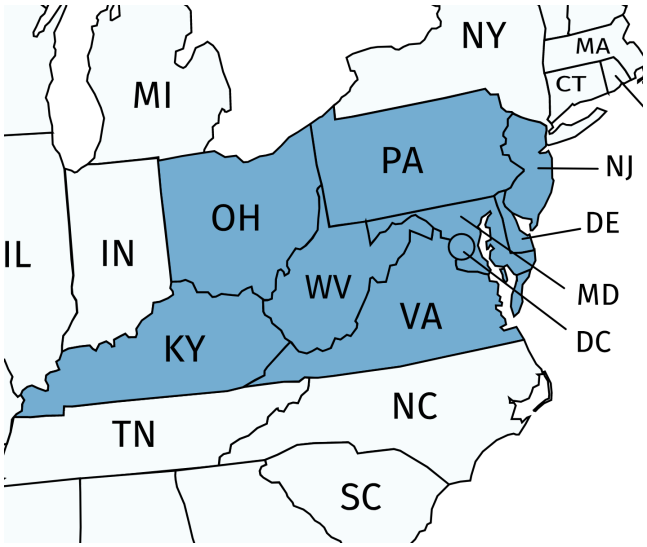
The terms and definitions in this section are ones that you should be familiar with, since we'll use them throughout the rest of the chapter. As long as you can connect each concept with the relevant examples, you'll be well prepared for the rest of our study of graph theory.

Exercises 8.1

1. There is a group of six people: Ana, Josh, Hope, Patricia, Jeff, and Carlos. Josh is friends with Patricia, Hope, and Ana; Jeff is friends with Patricia and Carlos; Patricia and Hope are friends. None of the other pairs of people are friends. Draw a graph to represent this group.

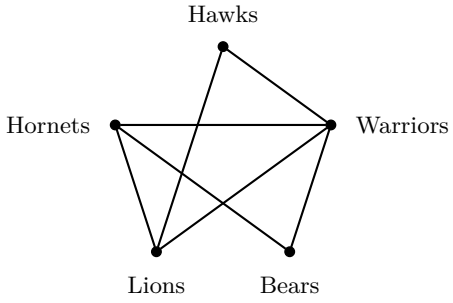
2. On campus, you travel between five buildings: Braddock Hall, Catoctin Hall, the library, the student center, and the arts building. There are walkways connecting Braddock Hall to Catoctin Hall and the library; the library is also connected to the arts building and the student center; there are also walkways connecting the student center to Catoctin Hall and the arts building. Draw a graph to represent this group of buildings.

3. The map below shows eight states and the District of Columbia highlighted in blue. Draw a graph to represent this map, where each node represents a state or district, and an edge represents a shared border between two regions.



4. The map below shows eight countries in Asia highlighted in blue. Draw a graph to represent this map, where each node represents a country, and an edge represents a shared border between two countries.

5. The graph below represents a tournament; each edge marks a game between two teams.



(a) Did the Hornets and the Hawks play each other?

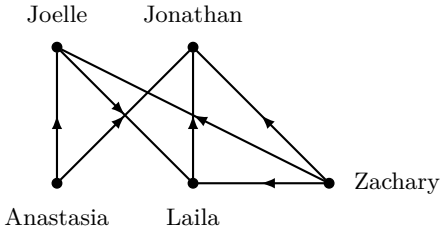
(b) How many games do the Warriors play?

(c) Which teams do the Bears play?

(d) Which team(s) played the most games?

(e) Which team(s) played the fewest games?

6. The graph below is an *influence graph*, where each edge represents the influence that one person has on another; the arrow goes from the influencer to the one they influence.



(a) Who does Laila influence?

(b) Does Jonathan influence Anastasia?

(c) How many people does Joelle influence?

(d) Who is the most influential (influences the most people)?

(e) Who is the most influenced (influenced by the most people)?

For problems 7–10, describe a graph that could be used to model the given application. Specifically, answer the following questions:

(a) Are loops allowed in this graph?

(b) Are multiple edges allowed between the same pair of nodes?

(c) Is this a simple graph or multigraph?

(d) Is this graph directed or undirected?

(e) Is this a complete graph (generally)?

7. Flights between major cities, if each node represents a city and each edge describes a flight from one city to another (or from a city to itself, if there is a sightseeing or training flight).

8. A party, where each node represents a person, and each edge represents whether one person knows the name of another.

9. The floorplan of a house, where each node represents a room or space (like a hallway), and each edge represents a doorway.

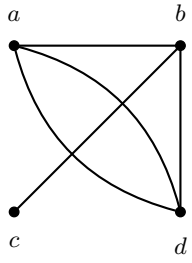
10. Courses offered at a college, where each node represents a course, and each edge represents a prerequisite requirement.

For problems 11–14, answer the following questions for the given graph:

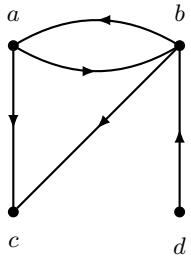
(a) Is it a simple graph or multigraph?

(b) Is it directed or undirected?

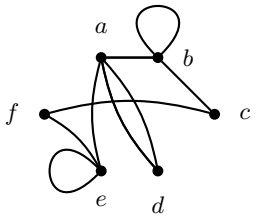
11.



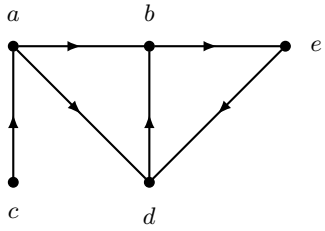
12.



13.

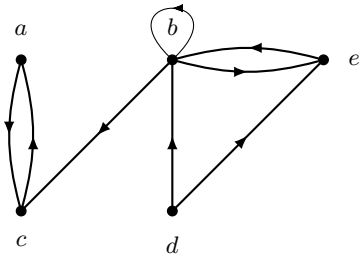


14.

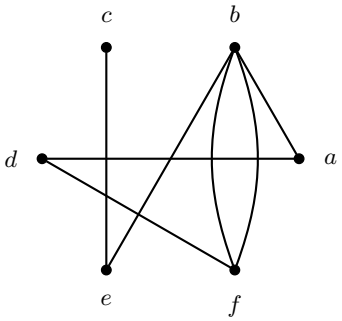


For problems 15–17, determine the degree of each node. For the directed graphs, determine both the in-degree and the out-degree of each node.

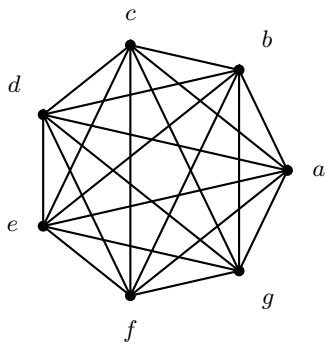
15.



16.



17.



SECTION 8.2 Euler and Hamilton Paths

CC BY-SA 2.0, Gage Skidmore

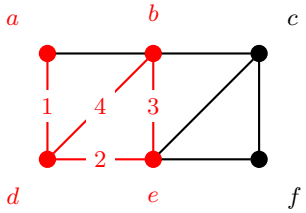
During a magazine interview in 1994, the actor Kevin Bacon made an offhand comment that he had "worked with everybody in Hollywood or someone who's worked with them." From this sprang the idea of the *Six Degrees of Kevin Bacon*, which sometimes appears as a game in which players must find a set of links between any given actor and Bacon. For instance, Emma Watson has a Bacon number of 2, meaning that it takes two steps to get from her to Bacon (Watson was in *The Circle* with Bill Paxton, who was in *Apollo 13* with Kevin Bacon).

It turns out that, as the name suggests, a surprising number and range of actors can be connected to Kevin Bacon (or really any prolific actor) with six steps or fewer. In fact, according to the website *oracleofbacon.org*, there are over 1.2 million actors with a Bacon number of 4 or less, and the average Bacon number is just over 3.

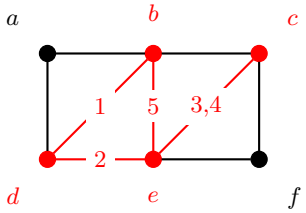
Paths and Circuits

What we've described above is the process of finding the **shortest path** between two nodes on a graph (a graph where the nodes represent actors, and two nodes are connected by an edge if those two actors have appeared together). We'll talk more about finding shortest paths in the next section, but for now, we can simply start with the definition of a **path**.

The definition of a path is quite intuitive: start at one node, then move along edges to another node, and you've taken a path. We use the term **circuit** to describe a path that ends at the same node where it started (so a *circuit* is a specific type of path; the term *path* is a general one that includes circuits).



Path: $a \to d \to e \to b \to d$



Circuit: $b \to d \to e \to c \to e \to b$

Note that in an undirected graph (like the two examples above), a path can travel in either direction along an edge, but in a directed graph, a path is only allowed to take an edge in its specified direction.

Paths and Circuits

A **path** in a graph is a sequence of edges; if the graph is simple (no multiple edges between nodes), the path can be described using the nodes that it passes through in order.

A **circuit** is a path that starts and ends at the same node.

Notice another distinction between the two graphs above: on the left, the path used four *distinct* edges to travel from a to d (although the path passed through d once before ending there), while the circuit on the right used the same edge (between c and e) twice. This brings us to another definition.

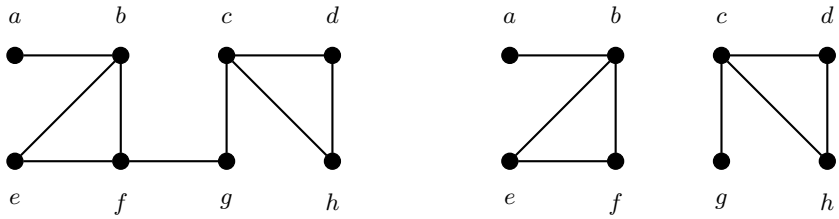
Simple Paths and Circuits

A **simple** path or circuit is one that does not contain any edge more than once.

In a moment, we'll return to the problem at the beginning of the last section: the Königsberg Bridge Problem. Remember that in that problem, we were hunting for a journey through the city that did not reuse any bridges; it turns out that we were actually looking for a *simple path*. Before we go back to that problem, though, we need to discuss one more concept: connectivity.

Connected Graphs

A connected graph is exactly what you would expect:



Connected

Disconnected (with two connected components)

Note: it's a bit more complicated for a directed graph; in that case, a graph is *strongly connected* if there is an allowable path between every pair of vertices, and *weakly connected* if we could find a path by ignoring directions.

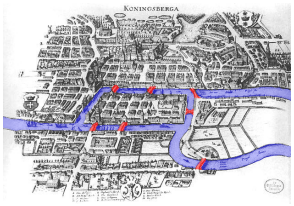
The definition is a simple one, and uses the idea of a path; a graph is connected if you can travel wherever you want from any starting place.

Connectivity

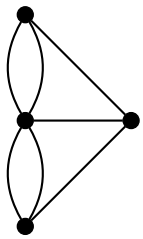
An undirected graph is **connected** if there is a path between every pair of nodes in the graph.

It turns out that we can even describe *how* connected a graph is, by seeing how hard it is to make it disconnected (in the example above, it was pretty easy to do, by simply removing one edge). But this simple description of connectivity is enough for us, because now we want to turn our attention back to the original problem: the bridges.

Euler Paths



Remember the setup: we want to find a way to travel through the city of Königsberg in such a way that we cross all of the seven bridges, but don't cross any more than once. In other words, we want to find a *simple path* through the following graph that uses every edge:



Just to clarify: a simple path means that we don't reuse edges, but we could leave some edges out. The additional requirement that we use *every* edge is what elevates this from a simple path to an *Euler path*.

Euler Paths

An **Euler path** in a graph is a path that goes through every edge of the graph exactly once.

Alternately, an Euler path is a simple path that contains every edge of the graph.

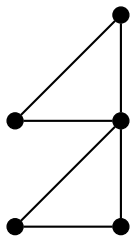
An *Euler circuit* is defined similarly: it is an Euler path that begins and ends at the same node.

Naturally, an Euler path is only possible in a connected graph.

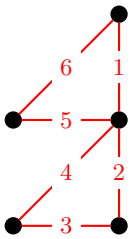
Now, the important question is, *how can we tell if an Euler path or circuit is possible?* If we know it's possible, of course, we'll also want to be able to find it.

It turns out that the answer to this question is surprisingly simple; there's something delightful about elegant answers like this one. Let's see what Euler discovered; we'll take a look at a few examples to see what we can learn.

Let's start with a simple one:



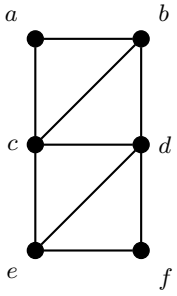
The goal is to trace over this graph, drawing every edge without lifting our virtual pen or retracing any lines. With a little trial and error, we can find an Euler circuit; for instance, if we start at the top node, we can trace every edge and end up back at the same place:



Notice that we can already draw one conclusion, in order to draw an Euler circuit, *we can't get stranded anywhere*. Now, what would it mean for us to get stranded? If we can find an edge leading to a node, but there aren't any unused edges that we can use to leave, we'll be stuck.

Observation 1:
to draw an Euler circuit, every time we arrive at a node, there must be an unused edge we can use to leave.

While you mull on that, here's another example:



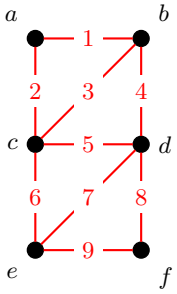
If we start at a, for instance, we'd have to also end there, because otherwise we'd have used up both edges that connect to it (one while leaving, and one while arriving) and have nowhere to go. In fact, this is true *whenever a node has an even degree*; leaving and returning uses up a pair of edges, so if you start at an even node, you have to end at the same node.

Observation 2:
if we start at an even node, we must end at the same node.

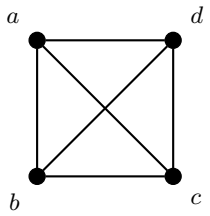
Now, notice that a, c, d, and f are all even nodes (only b and e are odd). If we started at any of these even nodes, we'd run into a problem, because we'd have to draw a circuit (see Observation 2). Look back at Observation 1: if we're drawing a circuit, we can't get stranded anywhere, which means we must be able to find an unused edge every time we arrive at a node. The first time we arrive at b, for instance, we'll have two unused edges to choose from, but when we return by the other one, we'd be stranded at b.

Okay, since we can't start at any of the even nodes, let's start at one of the odd ones (we'll pick b, but we could also use e). There are many options, but here's one possible route:

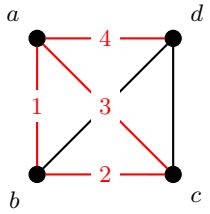
Observation 3:
if we start at an odd node, we must end at another odd node.



One final example (which happens to be K_4):



Since this one is symmetric, we can start at any node; there will be no difference if we pick another to start. Let's say we start at a:



Observation 4: if there are too many odd nodes, an Euler path is impossible.

After these first four edges, we arrive at d, and we're stuck; either way we go from here, we'll get stranded at either b or c. The problem is that there are too many odd nodes, so we end up running into one without anywhere to go.

Let's put together what we've observed from these three examples:

- The question hinges on whether nodes have even or odd degree.
- In the first example, all the nodes were even, and we could find an Euler circuit.
- In the second example, there were two odd nodes, and we could find an Euler path, but it had to start at one odd node and end at the other.
- In the third example, there were four odd nodes, and we couldn't find either an Euler path or an Euler circuit.

We can boil it down like this: for an Euler path/circuit, every time you arrive at a node, you need to be able to leave. The possible exceptions are the beginning and end, if you're only drawing a path, not a circuit. Arriving and leaving uses up two edges, so the nodes need to all be even for a circuit. For a path, there can be two odd nodes; one will be the starting point, and the other will be the ending point.

This sounds more complicated than it really is; we can write a simple rule to summarize everything we've observed.

Existence of Euler Paths and Circuits

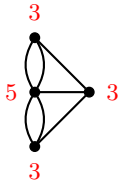
If all the nodes of an undirected graph are even, the graph has an Euler circuit (it can start and end at any node).

If there are two odd nodes, there is no Euler circuit, but there is an Euler path (starting at one odd node and ending at the other).

If there are any other number of odd nodes (other than zero or two), there is no Euler path or circuit.

Note: if there is no Euler path, we could still try the Chinese Postman Problem (named in honor of the Chinese mathematician Kwan Mei-Ko, who first studied it). The goal of this problem is to find the shortest circuit that visits every edge; in other words, the circuit that reuses as few edges as possible.

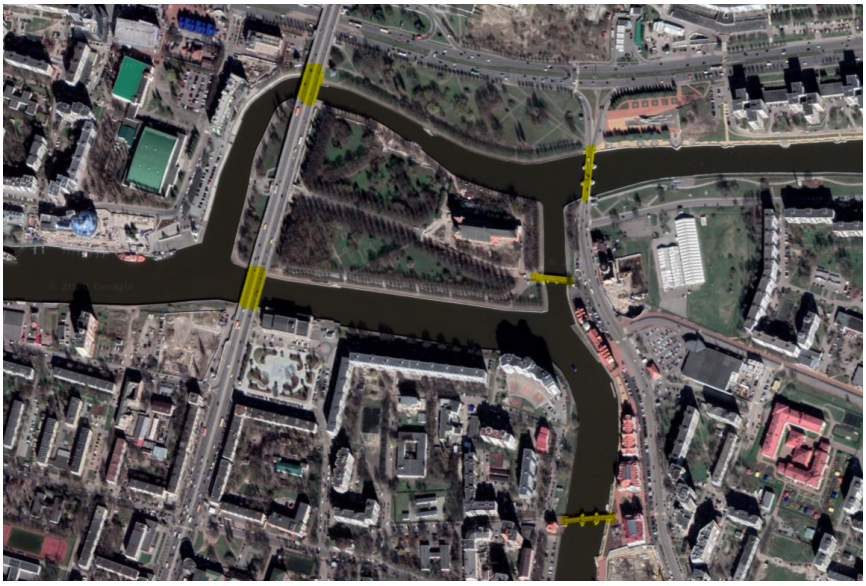
With that, let's go back to the bridges of Königsberg and see if there is an Euler path; let's label each node with its degree:



Since there are four odd nodes, there is no Euler path, so there's no way to travel through the city by crossing every bridge exactly once.

MODERN KÖNIGSBERG EXAMPLE 1

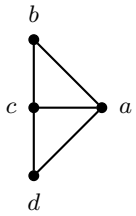
A modern image of part of Kaliningrad (which was once Königsberg) is shown below, with bridges highlighted.



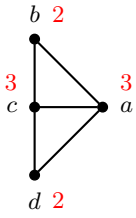
Is there an Euler path and/or circuit through this part of the city? If so, find one.

Solution

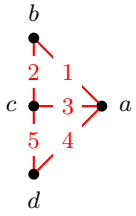
First, we need to abstract this map with a graph, keeping the nodes in the same positions as before, but redrawing the edges to match the current bridges:



Next, we can find the degree of each node:



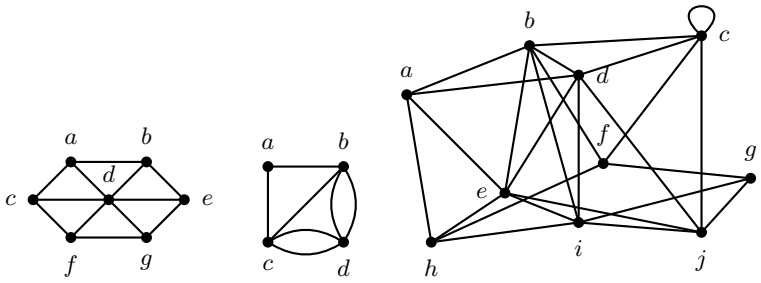
Since there are exactly two odd nodes, it is possible to find an Euler path through this graph, but not an Euler circuit. One Euler path is shown below:



This path can be written $\boxed{a \to b \to c \to a \to d \to c}$. Notice that it starts at one of the odd nodes and ends at the other.

EXAMPLE 2 EXISTENCE OF EULER PATHS

For each of the following graphs, determine if an Euler circuit exists. If not, determine whether there is an Euler path.



(a) (b) (c)

Solution

(a) Since all the nodes except d are odd, each with a degree of 3, there is no Euler path or circuit in this graph.

(b) All the nodes in this graph are even (a has degree 2, the rest have degree 4), there is an Euler circuit for this graph.

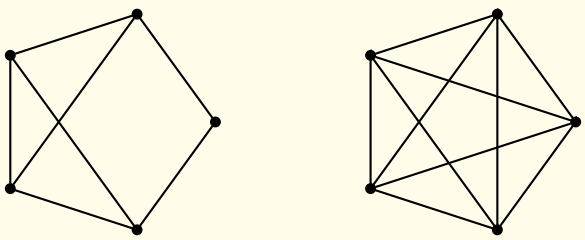
(c) Counting the degrees of each node for this graph is a bit more tedious, but the principle is the same. The results are below (remember that a loop contributes 2 to the degree of its node):

Node	Degree
a	4
b	6
c	6
d	6
e	6
f	4
g	3
h	4
i	6
j	5

Since there are exactly two nodes (g and j) with odd degree, there is an Euler path (but not an Euler circuit) for this graph.

TRY IT

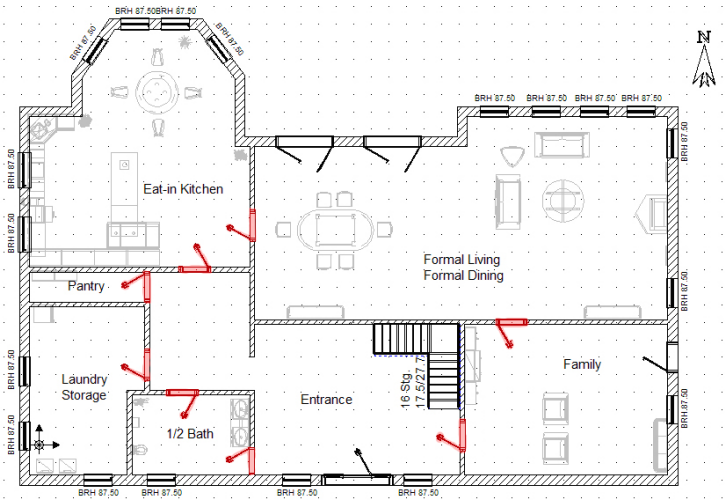
For each of the following graphs, determine if an Euler circuit exists. If not, determine whether there is an Euler path.



(a) (b)

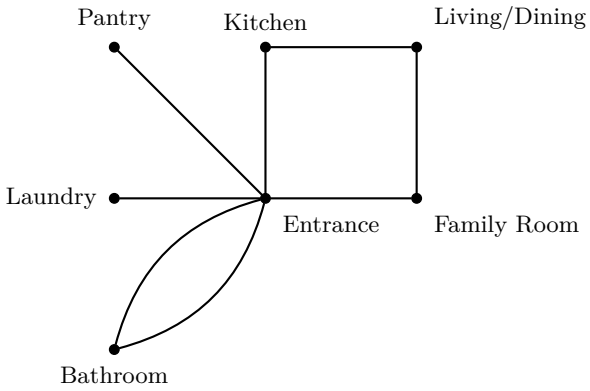
EULER PATH THROUGH HOUSE EXAMPLE 3

The floor plan below shows the first floor of a single-family home. Is there an Euler circuit/path through the interior of this level, using the highlighted doors (in other words, ignoring external doors and stairs)?



Solution

First, let's draw a graph to represent this home, with each node representing a room and each edge representing a door. Since the entrance is the most central location (in terms of connections to the rest of the home, we'll place that in the center, with edges out to the other rooms:



Now, we can count the degree of each node:

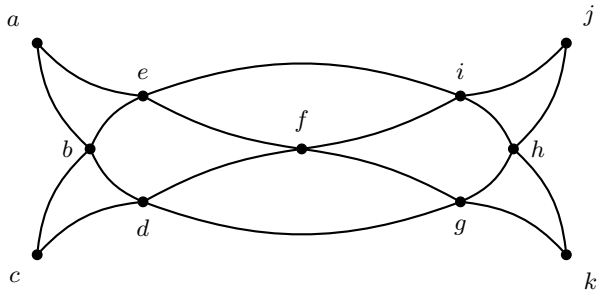
Room	Degree
Entrance	6
Family Room	2
Living/Dining Room	2
Kitchen	2
Pantry	1
Laundry	1
Bathroom	2

Since there are two rooms with an odd number of doors, there is an Euler path, but not an Euler circuit through this level. Any Euler path must start in either the pantry or laundry room and end in the other.

Finding an Euler Circuit

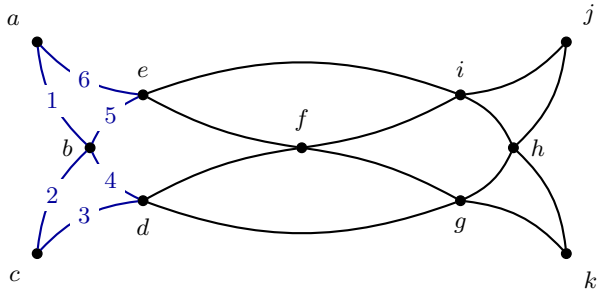
It's fairly simple to determine whether or not an Euler circuit exists, by simply identifying the degree of each node. Once we know that there *is* such a circuit, it is often simple enough to find what it is just by looking at the graph and tracing through it, as long as the graph is relatively small.

However, here we'll illustrate a more systematic process for finding an Euler circuit, in case it is useful. We'll use the graph below as an example.

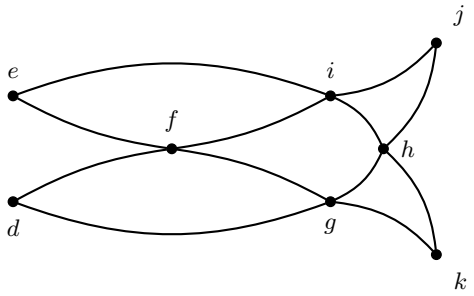


Notice that all the nodes are even, so it is possible to find an Euler circuit (or many, indeed). We can start at any point, and eventually come back and end at the same node.

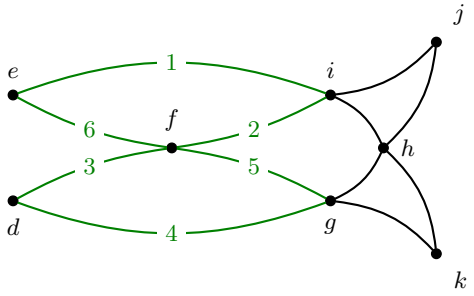
To begin, pick a starting point, and construct ***any*** circuit through the graph. Let's say we start with a; we could do something as simple as $a \to b \to e \to a$, but the longer our initial circuit is, the faster this process will go, so let's start with $a \to b \to c \to d \to b \to e \to a$:



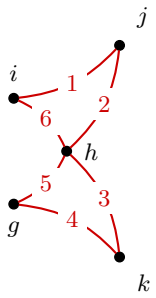
Now, delete those edges from the graph, and remove any points that are left isolated (so for instance, a will be deleted, but e will not, because there will be some remaining edges connected to e).



Then pick another point (we'll use e) and repeat this process. Say, for instance, we pick the next circuit to be $e \to i \to f \to d \to g \to f \to e$:



After removing those edges and isolated points, we can find one final circuit through the remaining points: $i \to j \to h \to k \to g \to h \to i$:



Having used all the edges now, we have three partial circuits:

$$a \to b \to c \to d \to b \to e \to a$$
$$e \to i \to f \to d \to g \to f \to e$$
$$i \to j \to h \to k \to g \to h \to i$$

To construct the final Euler circuit through the full graph, we simply stitch these three circuits together. Start from the end: notice that the last one begins and ends with i. Then, go to the one just before that: when that second circuit passes through i, we can pause and run through the third circuit before resuming. In practice, just replace i with the third circuit; instead of

$$e \to i \to f \to d \to g \to f \to e$$

we will now have

$$e \to i \to j \to h \to k \to g \to h \to i \to f \to d \to g \to f \to e$$

Finally, notice that this long string begins and ends at e, so we can substitute the whole thing in the place of e in the first circuit, and we'll have our full Euler circuit:

$$a \to b \to c \to d \to b \to e \to i \to j \to h \to k \to g \to h \to i \to f \to d \to g \to f \to e \to a$$

Again, in many cases there's no need to resort to a systematic process like this, but if you have trouble spotting an Euler circuit immediately, you can try it.

Euler Paths in Directed Graphs

The rule we described earlier (all nodes even → Euler circuit; two odd nodes → Euler path) applies to *undirected* graphs, but what about directed graphs? Now we don't have a single degree for each node; they each have an in-degree and an out-degree. Can we develop a similar rule, though?

With a bit of thought, we can adapt the earlier rule for directed graphs. Specifically, think about the condition for an Euler circuit, that all nodes be even. What does this actually mean? The reason for this is that we need to be able to leave every node that we arrive at, so really what the rule implies is that there are as many ways out as ways in. Thinking about it that way makes it clear that for a directed graph, we will look to see whether the in-degree and out-degree are equal.

Similarly, the condition for an Euler path is really about verifying that only the starting and ending node have one more exit and entrance, respectively, than the others, which all have equal entrances and exits. So for a directed graph, we can adapt this condition to say that all the nodes should have equal in-degree and out-degree, except for two: one of which has one more inlet than outlet, and the other has one more outlet than inlet.

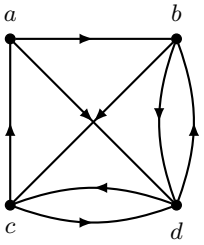
Euler Paths in Directed Graphs

If each node of a directed graph has equal in-degree and out-degree, the graph has an Euler circuit.

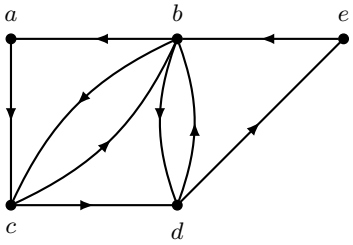
If one node has an in-degree that is larger than its out-degree by 1, and another node has an out-degree that is larger than its in-degree by 1, there is no Euler circuit, but there is an Euler path (starting at the node with larger out-degree and ending at the node with larger in-degree).

EXAMPLE 4 EULER PATHS IN DIRECTED GRAPHS

For each of the following graphs, determine if an Euler circuit exists. If not, determine whether there is an Euler path.



(a)



(b)

Solution

(a) Count the in-degree and out-degree of each node:

Node	In-Degree	Out-Degree
a	1	2
b	2	2
c	2	2
d	3	2

Notice that only two nodes do not have equal in- and out-degrees; one of them has exactly one more outlet than inlet, and the other has the reverse. Thus, while there is no Euler circuit, there is an Euler path for this graph, starting at a and ending at d.

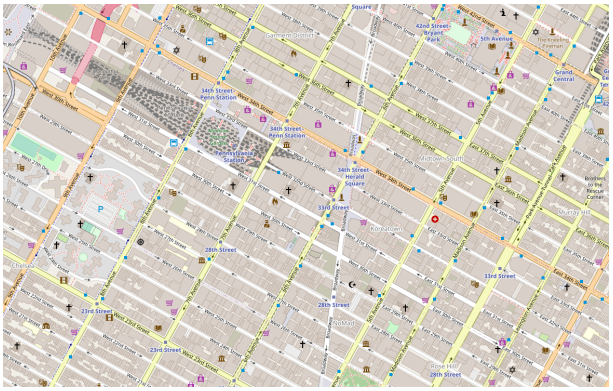
(b) Again, count the degrees:

Node	In-Degree	Out-Degree
a	1	1
b	3	3
c	2	2
d	2	2
e	1	1

This time, all the nodes have equal in-degree and out-degree, so there is an Euler circuit for this graph (we could start at any node).

Hamilton Paths

Take a look at the map below; what do you see?



By now, you can probably see a graph; if we designate each intersection as a node, the edges would be one-block segments of streets.

Now, suppose you work for the city's public works department, and your job is to schedule workers throughout the city as efficiently as possible.

After a snowstorm, your job is plan routes for the snowplows to get the streets cleared with a minimal amount of wasted driving. This is, in fact, an Euler path problem (or more likely a Chinese Postman problem, if an Euler path doesn't exist). Having read this chapter, you draw up a hyper-efficient plan and save the city thousands in fuel.

A few months later, the city decides to switch out all the traffic lights for a new, more efficient, more reliable model. Since you did such a great job with the snowplow scheduling, the department tasks you with planning the routes of the work crews through the city for this new contract. What's different about this problem?

In the snowplow problem, the goal was to *travel along every* ***edge*** exactly once. Now, with the traffic light problem, the edges are not the important part: there's no need to drive along every road, but simply to reach every intersection so that the crew can do their work there. Thus, the goal of the traffic light problem is to *travel to every* ***node*** exactly once. This is an entirely new kind of problem, and what we're looking for now is called a **Hamilton path**.

Hamilton Paths

A **Hamilton path** (or Hamiltonian path) in a graph is a path that passes through every node of the graph exactly once (and a Hamilton circuit is a Hamilton path that happens to be a circuit).

Now, having seen how easy it is to determine whether a graph has an Euler path or circuit, you may be expecting a similar rule for Hamilton paths. However, it turns out that there is no known rule that can tell us for certain whether or not any graph has a Hamilton path or circuit.

There are a few observations that we can make:

1. Every complete graph has a Hamilton path, and as long as there are at least 3 nodes, there is a Hamilton circuit. Since there are edges between every pair of nodes, we can visit all the nodes in any order we choose.

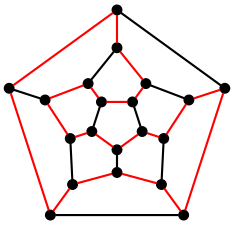
2. If a graph has a node with degree 1, it cannot have a Hamilton circuit, because there's no way to both arrive at and leave that node. It could, of course, have a Hamilton path.

3. When drawing a Hamilton path/circuit, once you have marked a node, you can eliminate all the other edges that connect to that node, which can simplify the picture.

There are other, more complicated theorems that give conditions under which Hamilton circuits exist (for instance, *Dirac's Theorem* says that a simple graph with n nodes, where $n \geq 3$, has a Hamilton circuit if the degree of every node is at least half of n), but for our purposes, we will simply check whether a graph has a Hamilton path or circuit by inspecting it and seeing if we can find one. Since we'll only deal with small graphs, this is good enough for us. For larger graphs, we could use a *brute force* approach, which means trying all the possible paths to see if any is Hamiltonian (as you can imagine, this is quite tedious).

Although William Rowan Hamilton was not the first one to study these paths, they are named for him because he invented a puzzle called the Icosian game, which starts with a 12-sided *dodecahedron*:

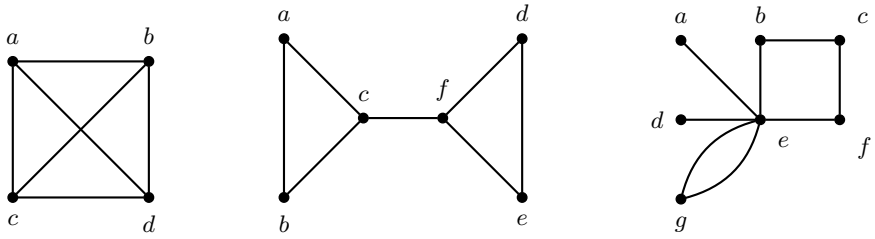
The goal is to find a Hamilton circuit, traveling along the edges and touching each corner exactly once. Hamilton sold this puzzle to a game dealer, and it was marketed throughout Europe. In the most popular form, the dodecahedron was modeled as a planar graph, like this one:



It was sold as a wooden board with pegs at the nodes of the graph, and players would wind a string along a path to solve it (one solution is shown in red).

EXAMPLE 5 HAMILTON PATHS

For each of the following graphs, determine whether a Hamilton circuit exists; if so, describe the circuit. If there is no Hamilton circuit, see if there is a Hamilton path.



Solution

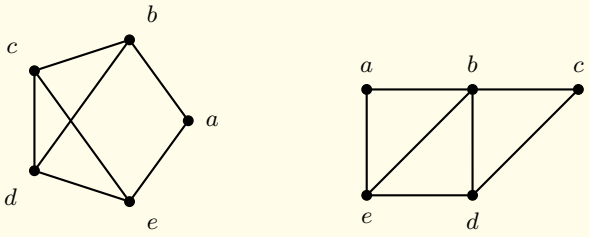
(a) Since this is a complete graph, K_4, we know that a Hamilton circuit exists. We could trace, for instance, the path $a \to d \to c \to b$.

(b) There is no Hamilton circuit, because c and f form bottlenecks; once you pass through one of them to go to the other side of the graph, there's no way to return. However, we can find a Hamilton path; $a \to b \to c \to d \to e \to f$, for instance.

(c) We know that there is no Hamilton circuit, because we have nodes with degree 1. Let's look for a Hamilton path; say we start at a. As soon as we travel to e, we're trapped, because no matter where we go, we'll cut off several points without any way to get to them without going back through e. Therefore, there is no Hamilton circuit or path for this graph.

TRY IT

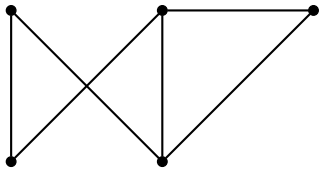
For each of the following graphs, determine whether a Hamilton circuit exists; if so, describe the circuit. If there is no Hamilton circuit, see if there is a Hamilton path.



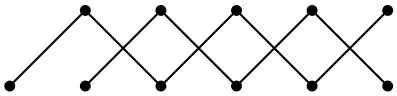
Exercises 8.2

In problems 1–3, determine whether the given graph is connected or disconnected.

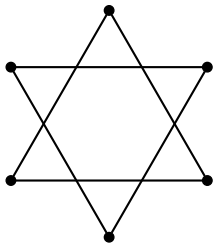
1.



2.

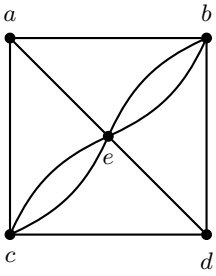


3.

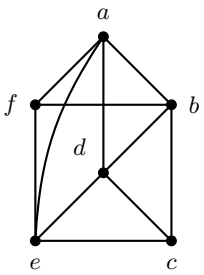


In problems 4–16, determine whether the given graph has an Euler circuit (and draw one if it exists). If not, determine whether it has an Euler path (and draw one if so).

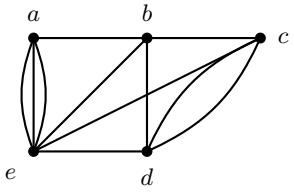
4.



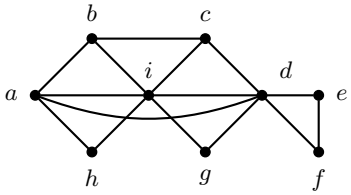
5.



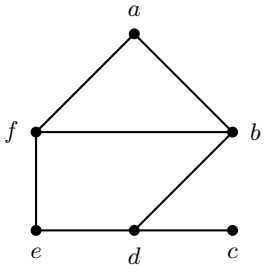
6.



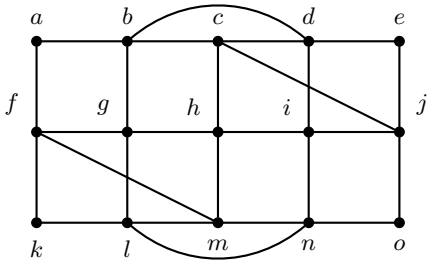
7.



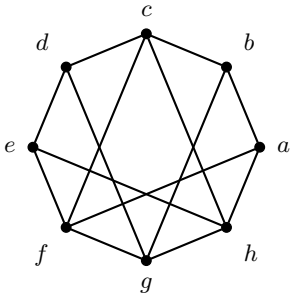
8.



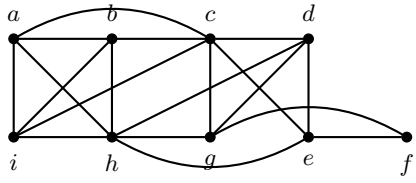
9.



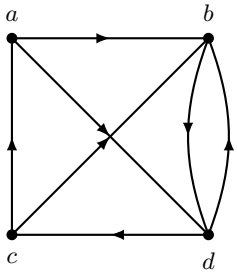
10.



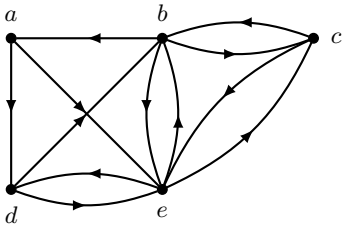
11.



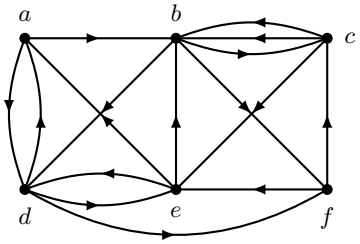
12.



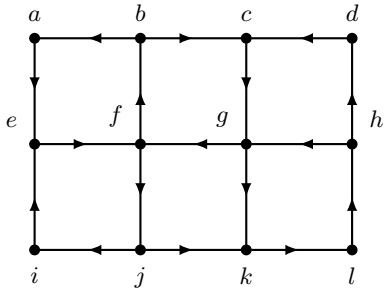
13.



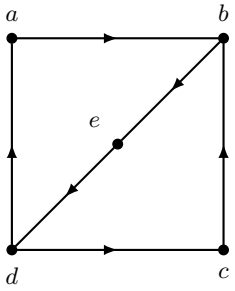
14.



15.

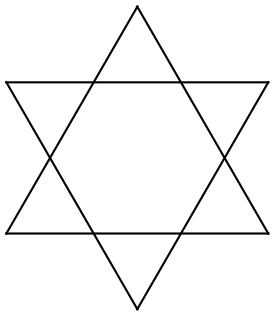


16.

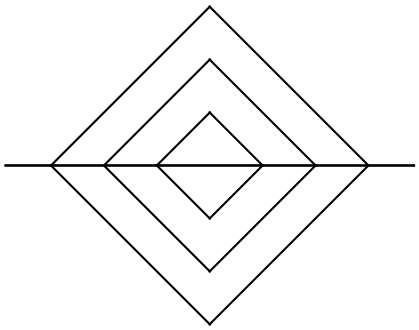


In problems 17–19, determine whether the picture shown could be drawn in one continuous motion without lifting the pencil or retracing part of the drawing.

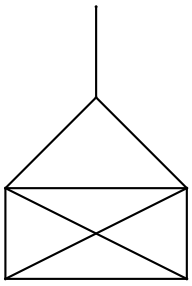
17.



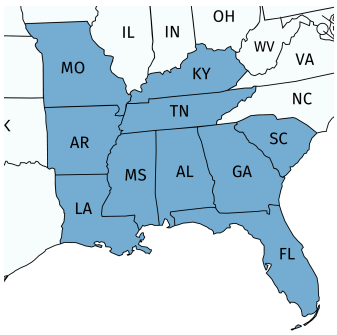
18.



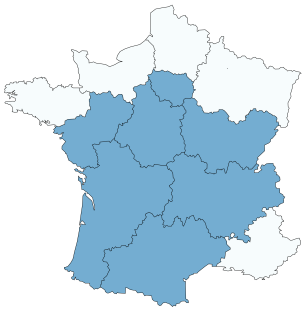
19.



20. The map below shows ten states highlighted in blue. Is there a path that a traveler could take through these states in such a way that they cross each border between two states exactly once?

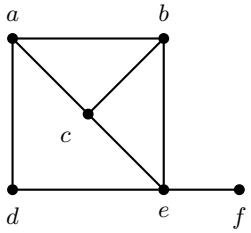


21. France is divided into 18 administrative regions, of which twelve are contiguous. The map below shows seven of these regions highlighted in blue. Is there a path that a traveler could take through these regions in such a way that they cross each border between two regions exactly once?

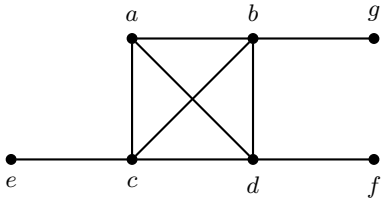


In problems 22–25, determine whether the given graph has a Hamilton circuit (and draw one if it does). If not, determine whether it has a Hamilton path (and draw one if so).

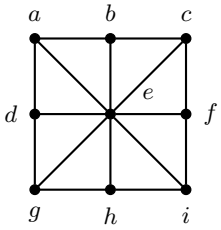
22.



23.



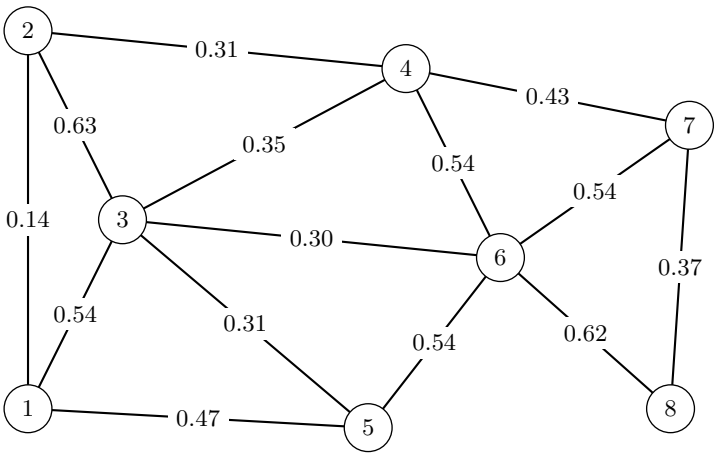
24.



SECTION 8.3 Shortest Paths

How does a navigation system find the best route to the destination? This is an incredibly complicated problem, searching through all possible routes, each with potentially dozens of twists and turns, to find the best one. In this section, we'll actually learn the basic process that underlies the way that Google Maps and similar programs do this.

To begin, remember that in the first section of this chapter, we discussed *weighted graphs*, like the one below, where each edge has a cost or weight associated with it. In the context of mapping software, each edge represents a road or street between two intersections, and the weight could represent either the distance between the intersections or the time required to travel along that edge.



In other applications, the cost could actually represent cost; for instance, we could draw a travel network where each edge represented a flight between two cities, and the edge weights could be the price of each airline ticket.

In this section, we'll address two different questions related to finding the *shortest* path through a graph, meaning the path with the lowest total weight. For instance, in the graph above, the path $3 \to 4 \to 6 \to 8$ would have a total weight of

$$0.35 + 0.54 + 0.62 = 1.51$$

We call this the **length** of this path. Notice that we aren't using length here to refer to the number of edges on the path, and certainly not to the actual drawn length of the lines, since those are arbitrary.

The two questions we'll consider are what we will call here the **Mail Delivery Problem** and the **Navigation Problem**.

In the Mail Delivery Problem, the goal is to **visit every node** in the graph using the shortest path possible. This is like a postal worker who must drop off mail in every mailbox along their route, and they would like to do so with as little driving as possible, to save both time and fuel.

In the Navigation Problem, the goal is to **get from one point to another** along the shortest route. In this problem, there's no need to visit every node; the only goal is to get to the destination, again with the lowest possible total cost.

The Mail Delivery Problem is also called the Traveling Salesperson Problem, and it is a famous problem in mathematics and computer science.

An **algorithm** is simply a process, or set of steps, used to accomplish a goal; an algorithm can be thought of as a recipe. For instance, long division is an algorithm used to divide two numbers. In most algorithms, there is some repetition or cycling; you can see this if you use long division.

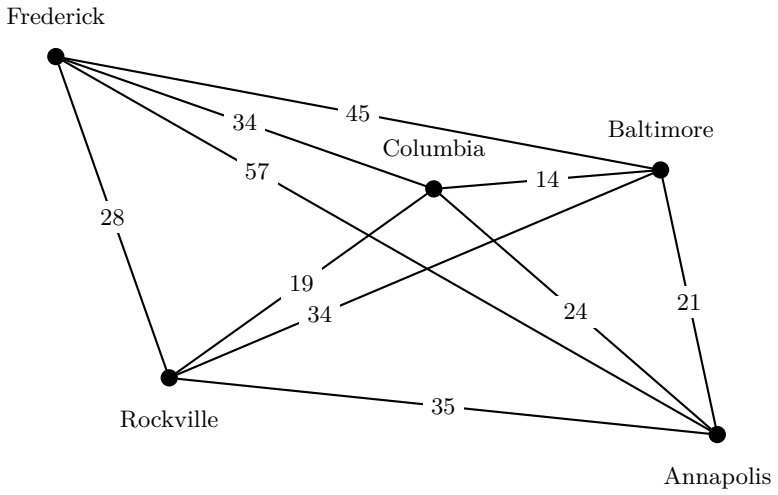
Algorithms are a common topic in computer science (which is one of the fields that uses graph theory the most); most programs can be called algorithms, in the sense that they describe a sequence of commands for the computer to follow.

To tackle these problems, we'll learn about two *algorithms*:

1. To solve the Mail Delivery Problem, we'll use the **Nearest Neighbor Algorithm**
2. For the Navigation Problem, we'll use **Dijkstra's Algorithm**, named for a Dutch computer scientist.

Mail Delivery - The Nearest Neighbor Algorithm

The most straightforward way to approach this problem of visiting every node with the shortest path is to check all the possibilities. Let's use the example of 5 cities in Maryland: Frederick, Baltimore, Annapolis, Columbia, and Rockville. For simplicity, we'll use the straight-line distance between them, shown on the graph below in miles.



Instead of this graph, we could also use a table like the following one to show the distances between all the cities:

	Frederick	Baltimore	Annapolis	Columbia	Rockville
Frederick	–	45	57	34	28
Baltimore	45	–	21	14	34
Annapolis	57	21	–	24	35
Columbia	34	14	24	–	19
Rockville	28	34	35	19	–

In order to read this table to find the distance between two cities, locate the column for one city and the row for the other city, and the distance is at the intersection (the dashes represent no distance). Notice the symmetry in the table: the distance between Frederick and Baltimore is the same as the distance between Baltimore and Frederick.

Now, notice that this is a complete graph, meaning that we can travel from any city to any other. This is realistic for a mail delivery van or salesperson, but it means that there are many possible paths. In fact, ignoring reverse paths (meaning that we'll treat Frederick → Baltimore → Columbia and Columbia → Baltimore → Frederick as the same route), there are a total of 12 ways to travel through this network from any given city.

Let's assume we start and end in Frederick; here are the 12 possible circuits, with the total distance listed for each (abbreviating each city with the first letter of its name):

Route	Total Distance (miles)
$F \to B \to A \to R \to C \to F$	154
$F \to B \to A \to C \to R \to F$	137
$F \to B \to C \to R \to A \to F$	170
$F \to B \to C \to A \to R \to F$	146
$F \to B \to R \to C \to A \to F$	179
$F \to B \to R \to A \to C \to F$	172
$F \to R \to B \to A \to C \to F$	141
$F \to R \to B \to C \to A \to F$	157
$F \to R \to C \to B \to A \to F$	139
$F \to R \to A \to B \to C \to F$	132
$F \to A \to R \to B \to C \to F$	174
$F \to A \to B \to R \to C \to F$	165

The shortest possible path is near the bottom of the list: traveling along the route Frederick → Rockville → Annapolis → Baltimore → Columbia → Frederick is the best option, for a total distance of 132 miles (47 miles shorter than the worst case).

Now, the obvious downside of this method is that it is incredibly tedious to do. The more serious problem is that as the number of cities increases, the number of possible paths explodes, making this brute-force approach impractical. For instance, if we used 25 stops (a small number for a local mail delivery), there would be so many possibilities that if we could check one path every nanosecond, it would take about ten million years to find the lengths of all of them. Since we don't have that kind of time, we need another approach, and for that, we turn to the **nearest neighbor algorithm**.

Nearest Neighbor Algorithm

At each step, go to the nearest *unvisited* node. Repeat until all nodes have been visited.

This does not generally give the absolute best result, but it is an easy way to get close.

The nearest neighbor algorithm is an example of a *greedy algorithm*, meaning that it takes the best option at each step without taking a larger view into account.

NEAREST NEIGHBOR ALGORITHM — EXAMPLE 1

Use the nearest neighbor algorithm to find a possible minimum circuit through the Maryland cities, starting and ending at Frederick.

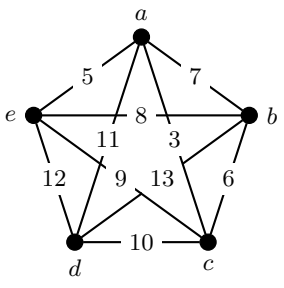
Solution

Starting at Frederick, the closest city is Rockville. Once we get to Rockville, the closest city (besides Frederick, since we've already been there), is Columbia, so we go there. From there, Baltimore is the closest remaining city, which just leaves Annapolis before we return to Frederick:

Circuit: $\boxed{F \to R \to C \to B \to A \to F}$

The total distance is 139 miles; notice that while this isn't the best possible path, it *is* the third best, only 7 miles longer than the optimal one. And the advantage is obvious; this process was much quicker and simpler than the brute force approach.

Using the graph shown in the margin, use the nearest neighbor algorithm to find a possible minimum circuit starting at a, then repeat this starting at b and starting at c.



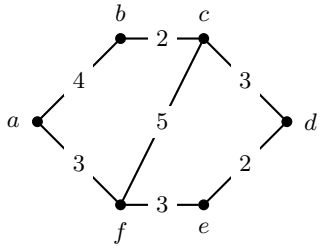
Navigation - Dijkstra's Algorithm

Edsger Dijkstra in 2002

Edsger Dijkstra (*pronounced DIKE-stra*) was a Dutch computer scientist before there was such a thing (before there were computer scientists, that is, not before there were Dutch people). In the 1950s, when he got married, he was required to put his profession on the application for a marriage license, and when he wrote "programmer," the authorities made him change it, since there was no such profession. Instead, he stated that he was a theoretical physicist, which was his first course of study.

Dijkstra was brilliant, and he went on to lay much of the foundation for modern computer science. One of the problems he studied was this problem of navigation, finding the shortest path between two points on a graph. His solution has been applied to many fields, including robotics (navigating a robot around obstacles), transportation, and even games (finding an optimal solution for a Rubik's cube, for instance).

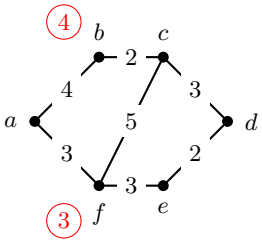
Dijkstra's algorithm is more complicated than the nearest neighbor algorithm, at least at first. The actual steps, though, are very simple mathematically. Before we give the full algorithm, let's do a simple example to illustrate how it works. Start with the graph below:



Now, suppose we want to travel from a to d. This graph is simple enough that we could probably spot the shortest path just by looking at it, but let's illustrate a more systematic process.

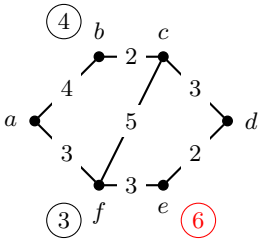
Essentially, what the process will do is find the minimum distance (and shortest path) from a to every other point, so once we're done, we just look at the result for d. As we go, we'll keep track of the shortest path to a particular point, and use that to move forward one more step. For instance, once we know the shortest path to c, we know that we can find the shortest path to d that passes through c by adding 3 for the path from c to d.

If we start at a, we have two options: go to b (4) or to f (3). We can keep track of these in a table like this:



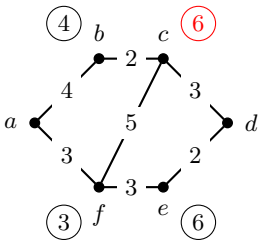
Node	Minimum Distance from a	Shortest Path from a
b	4	$a \to b$
f	3	$a \to f$

Now, if we take another step, we can get to c (through either b or f) and e (through f). Since there's only one option for e, let's start there: we can go through f (at a cost of 3) and on to e (another 3) for a total cost of 6. Let's add that to the table:



Node	Minimum Distance from a	Shortest Path from a
b	4	$a \to b$
e	**6**	$a \to f \to e$
f	3	$a \to f$

For c, we have two options: (1) go through b, with a total cost of 6 (the 4 to get to b, plus the 2 to go from b to c), or (2) go through f, with a total cost of 8. Of these, we choose the first option, since it has a lower cost; add that to the table:



Node	Minimum Distance from a	Shortest Path from a
b	4	$a \to b$
c	**6**	$a \to b \to c$
e	6	$a \to f \to e$
f	3	$a \to f$

Our last step ended at either c or e, so one more step from either of them will take us to d. Since both are at a minimum distance of 6 from a, the best path to d will be the one going through e, since that only requires adding 2 to get to d, instead of the 3 required from c. Thus, the optimal path from a to d is

$$\boxed{a \to f \to e \to d}$$

which covers a total distance of 8.

When we actually state Dijkstra's algorithm, it is written in a slightly different form, but the process is what we just outlined: by keeping track of the shortest path from the starting point to all the intermediate nodes, we can move through the graph in waves, so we don't have to evaluate all possible paths.

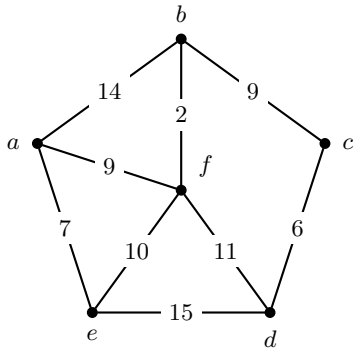
Dijkstra's Algorithm

To find the shortest path from the origin to the destination:

1. First, set the initial distances (from the origin) for every node:
   - The distance at the origin is 0.
   - All other distances are infinity (∞) to begin (so that when we find the first real distance, it will be smaller, and we can update this value).
2. Next, pick the node (from the ones that we haven't evaluated yet) with the smallest current distance:
   - Update the distances to all its neighbors; if the new distance is shorter than the previous one, replace the old with the new.
   - Check off that node.
3. Repeat step 2 until you reach the destination.

DIJKSTRA'S ALGORITHM EXAMPLE 2

Use Dijkstra's algorithm to find the shortest path between a and c in the graph shown below.

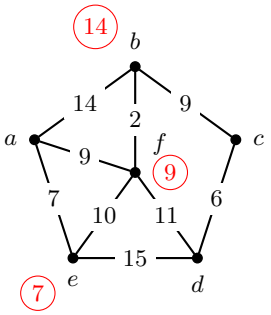


Solution

To keep track of the distances, we'll build a table, and update it at every step. First, we set the initial distances for each point (0 at a, infinity everywhere else).

Checked?	Node	Minimum Distance from a	Shortest Path from a
	a	0	
	b	∞	
	c	∞	
	d	∞	
	e	∞	
	f	∞	

Next, we select the point with the smallest current distance–that would be a–and update the distance to all its neighbors. Since a has b, e, and f as neighbors, we will update the distance and path for each of these three. The distance will be the current distance at a (0) plus the distance to the new point, and the path will be the path to a (nothing) with the new segment added on. Once we do all that, we can check off a.

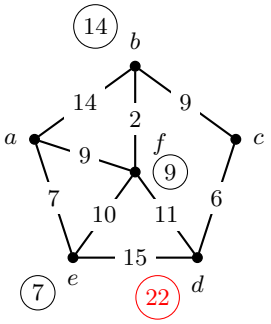


Checked?	Node	Min. Distance from a	Shortest Path from a
✓	a	0	
	b	~~∞~~ **14**	$a \to b$
	c	∞	
	d	∞	
	e	~~∞~~ **7**	$a \to e$
	f	~~∞~~ **9**	$a \to f$

Now we simply repeat this process: select the unchecked point with the smallest distance–which is now e–and update its neighbors, then check it off. By the time we get to c, we will be guaranteed to have found the shortest path from a to c.

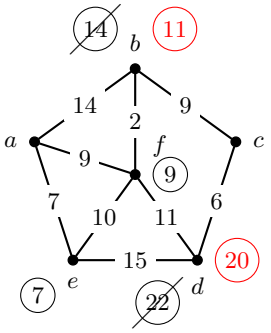
The (unchecked) neighbors of e are d and f, so for each of them, add the distance to that point from e together with the current distance at e (7). If that is smaller than the current distance at that point, update the distance and path; if not, do nothing there.

For d, this results in a distance of $7+15 = 22$, which is smaller than ∞. For f, however, this gives a distance of $7 + 10 = 17$, which is larger than the current distance there, so we'll leave the row for f unchanged.



Checked?	Node	Min. Distance from a	Shortest Path from a
✓	a	0	
	b	14	$a \to b$
	c	∞	
	d	~~∞~~ **22**	$a \to e \to d$
✓	e	7	$a \to e$
	f	9	$a \to f$

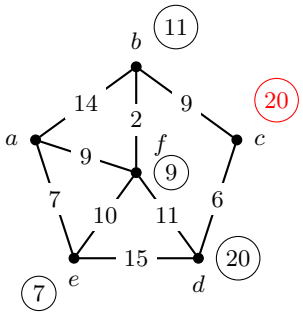
Next, update the neighbors of f (b and d, since a and e have already been checked): the new distance to b is 11, which is smaller than the 14 currently marked there, and the new distance to d is 20, which is also smaller than the current 22 for that point.



Checked?	Node	Min. Distance from a	Shortest Path from a
✓	a	0	
	b	~~14~~ **11**	$a \to f \to b$
	c	∞	
	d	~~22~~ **20**	$a \to f \to d$
✓	e	7	$a \to e$
✓	f	9	$a \to f$

After checking b, the table looks like this:

Checked?	Node	Min. Distance from a	Shortest Path from a
✓	a	0	
✓	b	11	$a \to f \to b$
	c	~~∞~~ **20**	$a \to f \to b \to c$
	d	20	$a \to f \to d$
✓	e	7	$a \to e$
✓	f	9	$a \to f$



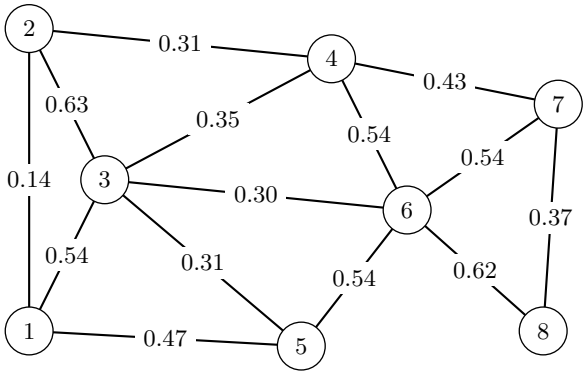
Finally, we'll check d (the new distance to c is 26, which is larger than its current distance):

Checked?	Node	Min. Distance from a	Shortest Path from a
✓	a	0	
✓	b	11	$a \to f \to b$
	c	20	$a \to f \to b \to c$
✓	d	20	$a \to f \to d$
✓	e	7	$a \to e$
✓	f	9	$a \to f$

The shortest path from a to c, then, is $\boxed{a \to f \to b \to c}$, with a total length of 20.

This process can look complicated at first, but with a little practice, it goes pretty quickly. Notice that the actual math we're doing is very simple; we just add two numbers at each step and compare them to a previous result.

For another (abbreviated) example, let's go back to the graph at the beginning of the section, and find the shortest path from (1) to (7) using Dijkstra's algorithm.



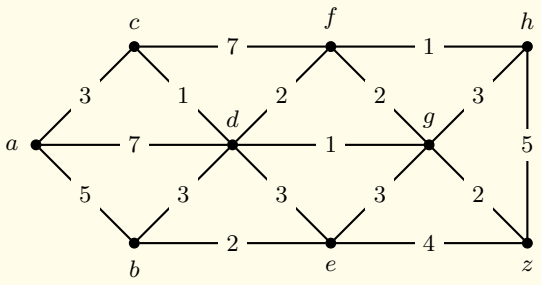
You can try this one on your own; when you do, you should end up with a table that looks like this:

Checked?	Node	Min. Distance from (1)	Shortest Path from (1)
✓	(1)	0	
✓	(2)	0.14	(1) → (2)
✓	(3)	0.54	(1) → (3)
✓	(4)	0.45	(1) → (2) → (4)
✓	(5)	0.47	(1) → (5)
✓	(6)	0.84	(1) → (3) → (6)
✓	(7)	0.88	(1) → (2) → (4) → (7)
✓	(8)	1.25	(1) → (2) → (4) → (7) → (8)

The shortest path from (1) to (7) is $\boxed{(1) \to (2) \to (4) \to (7)}$, which has a total length of 0.88.

TRY IT

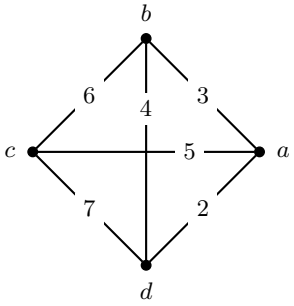
Use Dijkstra's algorithm to find the shortest path between a and z in the graph shown below, and the length of that path.



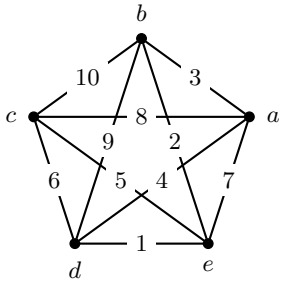
Exercises 8.3

In problems 1-3, use the nearest neighbor algorithm to find a minimum possible circuit through each graph starting at a.

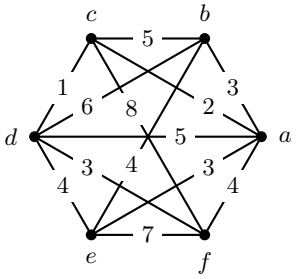
1.



2.

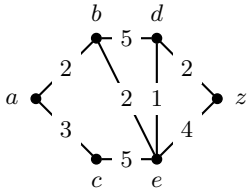


3.

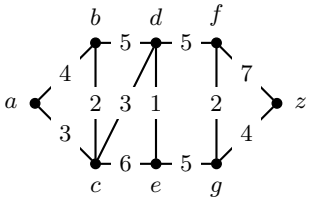


In problems 4–6, use Dijkstra's algorithm to find the shortest path through each graph between a and z.

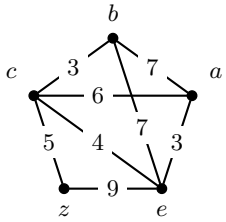
4.



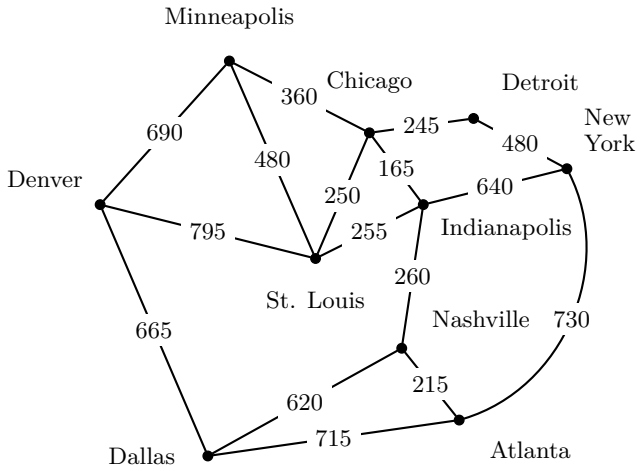
5.



6.



7. The graph below shows the distances between cities. Use the graph to answer the questions below.



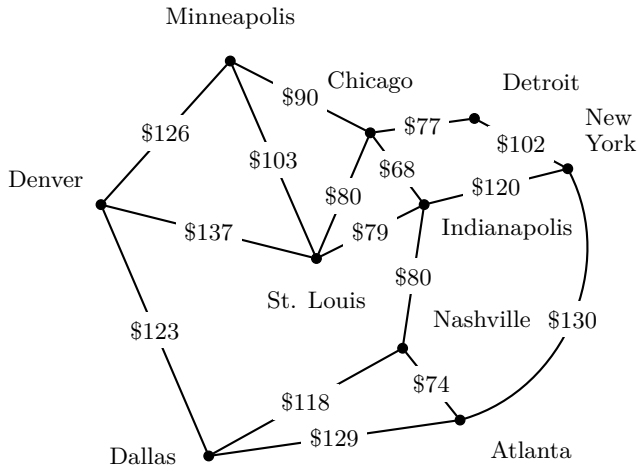
(a) Use the nearest neighbor algorithm to find a path that starts in Chicago and visits all the cities shown, while trying to minimize distance traveled.

(b) What is the total length of the path found in part (a)?

(c) Find the shortest path between Dallas and New York. What is the length of this path?

(d) Find the shortest path between Minneapolis and Atlanta. What is the length of this path?

8. The graph below shows the cost of flights between cities. Use the graph to answer the questions below.



(a) Use the nearest neighbor algorithm to find a path that starts in St. Louis and visits all the cities shown, while trying to minimize the cost of travel.

(b) What is the total cost of the path found in part (a)?

(c) Find the cheapest path between Nashville and Denver. What is the cost of this path?

(d) Find the cheapest path between Detroit and Denver. What is the cost of this path?

9. A salesperson has responsibility over four cities in Maryland and northern Virginia, and they compiled the distances between them; these distances are shown in the table below. If the salesperson needs to visit all four cities, and is currently in Ellicott City, use the nearest neighbor algorithm to plan their route. How far will they travel in total along this path?

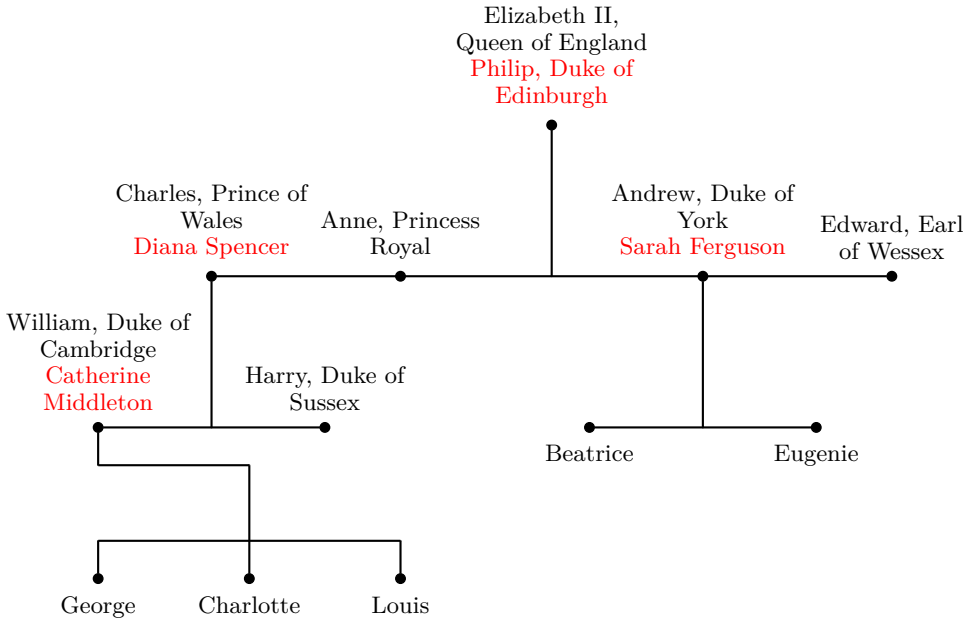
	Annapolis	Alexandria	Ellicott City	Reston
Annapolis	–	45	26	46
Alexandria	45	–	37	19
Ellicott City	26	37	–	35
Reston	46	19	35	–

10. An American tourist is traveling through Great Britain, and would like to visit five cities. The tourist has estimated the cost of a train ticket between pairs of these cities, and the results are shown in the table below. Plan a route that will take the tourist through all five cities with as little cost as possible, starting and ending in London. Use the nearest neighbor algorithm; what is the cost of this journey?

	London	Edinburgh	York	Cardiff	Chester
London	–	\$175	\$110	\$65	\$115
Edinburgh	\$175	–	\$95	\$195	\$105
York	\$110	\$95	–	\$145	\$60
Cardiff	\$65	\$195	\$145	–	\$85
Chester	\$115	\$105	\$60	\$85	–

SECTION 8.4 Trees

If you've ever studied your genealogy, one of the most basic ways to visualize it is by using a *family tree*, like the one shown below for a portion of the British royal family. In this tree, each node represents a member of the royal family by birth. If that member has children, the other parent of those children is shown in red.



The terminology of a tree and branches is clear, because the family tree above has the same structure as a physical tree, with branches that lead to smaller branches, and so on.

In graph theory, we borrow this term of a *tree* for a graph that has this same structure. Now, it can be hard to identify the defining feature of a tree branch, but notice that the main pattern is that none of the branches loop back to reconnect with the rest of the tree; once you head down a particular edge, there's no way to get back where you were without returning along that edge. This leads us to the definition of a tree (watch out; the definition is so terse that it can be hard to grasp at first).

Trees

A **tree** is a connected graph with no simple circuits.

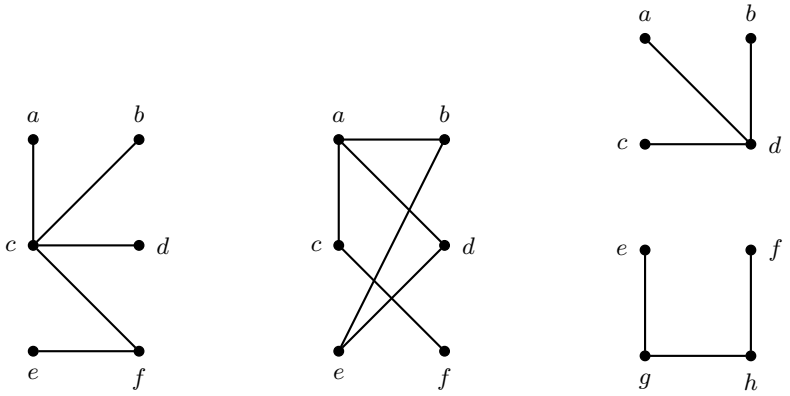
Let's make sure that makes sense: a tree is a graph where a path down a particular branch won't find itself looping back and reconnecting; that path is only connected to the rest of the graph at one point. Therefore, there can't be any simple circuits, because to return to a point that we left earlier, we'd have to retrace our steps.

Now, it turns out that there are a few equivalent ways to identify a tree. Each of the following is an equivalent definition for a tree: a graph G is a tree if

- There is a unique simple path between any two nodes (not particularly helpful for identification).
- G is connected, but barely; removing any edge would make it disconnected (think about cutting a branch from a tree–that branch doesn't hang there, connected at a another point). This can be easy to check visually: see if you can find any edges that you can remove and still leave the graph connected.
- G is connected, and there are $n - 1$ edges, where n is the number of nodes (this one is easy to check too, since it just involves counting).

IDENTIFYING TREES EXAMPLE 1

Which of the following are trees?



Solution

(a) This is a tree, because it *is* connected, but removing any edge would create disconnected components. We could also count the nodes and edges, and note that there is one fewer edge than the number of nodes.

(b) This is not a tree. Again, we could count the nodes and edges, or try removing edges to see if any leave it connected, but we can also note that there are simple circuits, like $a \to d \to e \to b \to a$ in this graph.

(c) This is not a tree, because it is not connected. However, since each of the components are trees themselves, this could be called a **forest**.

Our conclusion is that (a) is the only tree .

There are many, many applications of trees, but we will only discuss two of them: **binary search trees** and **spanning trees**.

for instance, trees can be used to describe games; chess programs do this, as one example

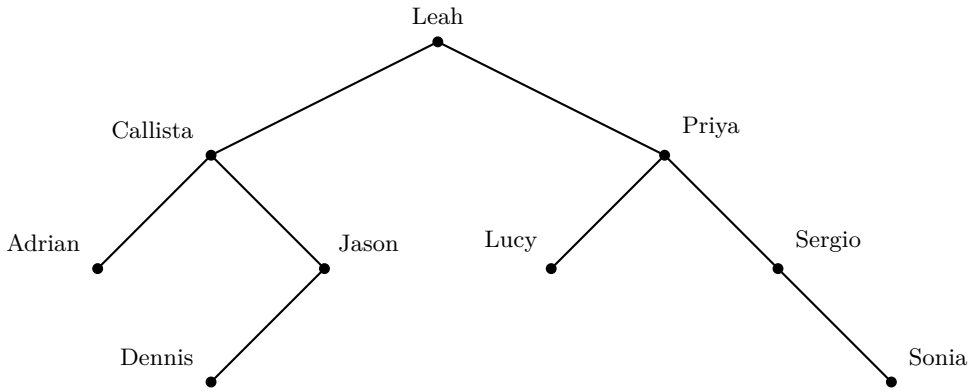
Binary Search Trees

Searching through a list is a hard problem. If you were handed a randomly sorted list of names and told to find a particular one, you'd have to simply read through every name, and you could expect on average to have to read half of the list before finding the name. Think of how often you use a search function in your life, and it becomes obvious that we need searching to be as efficient as possible.

One way to make searching more efficient is to sort the items. Say, for example, the list of names you were given was ordered alphabetically; would that make your job easier? Of course it would. Say, for instance, you were looking for the name Kayla: you would know to look near the middle of the list, and if your eye fell on the M's, you'd know to go backward. Similarly, if you started at the G's, you'd know to go forward.

This is the basic concept of a *binary search tree*; the word *binary* simply refers to two possibilities. In the alphabetically ordered list, you have two options: go forward or go backward, so in that case, you're really doing a binary search.

It turns out that there is an efficient way to use a tree to encode an ordered list like this, and there are advantages to this that have to do with the way that computers can read the tree. Below is an example of a tree that can be used to search for names in the following list: Leah, Callista, Priya, Lucy, Jason, Sergio, Sonia, Dennis, and Adrian.



Binary Tree: each node has one *parent* and at most two children

Notice the structure of this tree: first of all, it is called a **binary tree** because at each level, a node will have at most two branches coming out of it and going down to the next level (we can say that each node will have at most two *children*, if we view this like a family tree).

Next, the reason this tree can be used for searching is that it is **sorted**: notice that at each node, if you turn to the left and follow a branching path, all the names along that path will be *before* the original one, and if you go to the right, all the names will come *after* it, alphabetically. For instance, if you start with Priya, on the left-hand branch you'll find Lucy (before Priya alphabetically), and on the right-hand branch, you'll find Sergio and Sonia (both after Priya alphabetically). This is what makes it a binary *search* tree.

Say you are searching for the name Jason. Start at the top of the tree, and compare Jason to the name there: Leah. Since Jason comes before Leah alphabetically, we know Jason will be in the left branch, so head to the left. Now, pause here, and notice that we've immediately eliminated about half of the possible names in the list from consideration; we don't have to search through those. This is the major feature of binary search trees, since by eliminating about half of the remainder of the list at every step, we can hone in on our search term very quickly.

Back to the search: we're heading down the left branch, looking for Jason. The next name we encounter is Callista, and since Jason comes after that alphabetically, we take a turn to the right, which brings us to our result. Notice that this only took two steps, and we found the right name out of a list of 9. The efficiency of this method only gets more dramatic as the list gets bigger.

BUILDING A BINARY SEARCH TREE — EXAMPLE 2

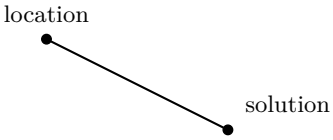
Build a binary search tree for the following list of words, starting with the first word at the top of the tree:

location, solution, promotion, decision, city, bread, enthusiasm, writer, signature, criticism

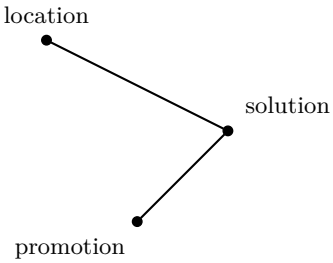
---

Solution

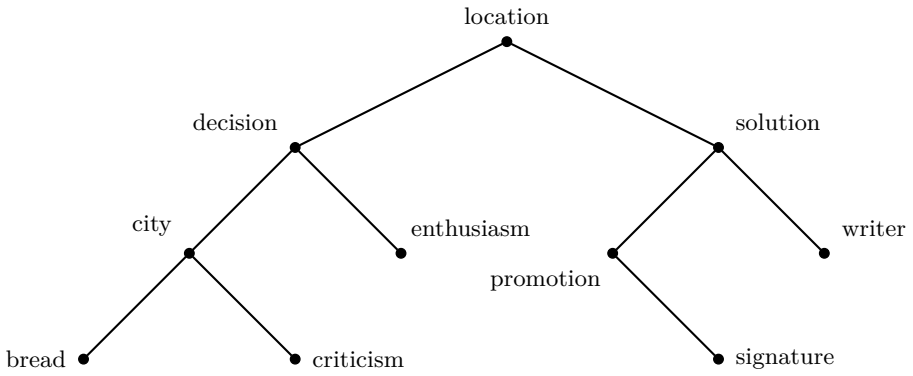
Start with the word *location* at the top. To add the word *solution*, note that it comes after *location* alphabetically, so create a node to the right.



For the next word, *promotion*, start again by comparing it to the top word, *location*. Since it comes after that, go to the right, and compare it to the word there, *solution*. This time, it comes before the comparison word, so we turn to the left, and since there's nothing there, we drop *promotion* in that spot.



If we continue this process, we'll eventually get the tree shown below (try it yourself and see if you can get the same tree).



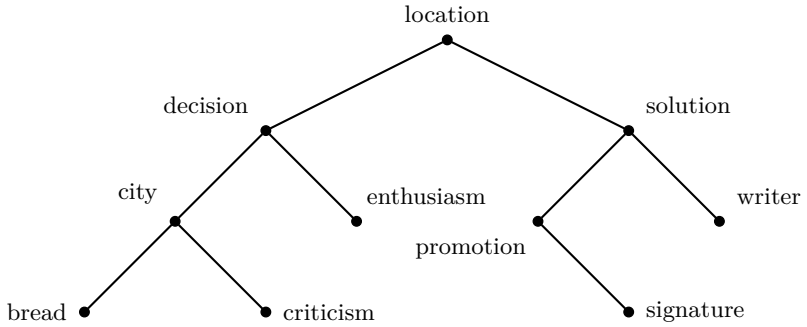
TRY IT

Build a binary search tree for the following list of words, starting with the first word:

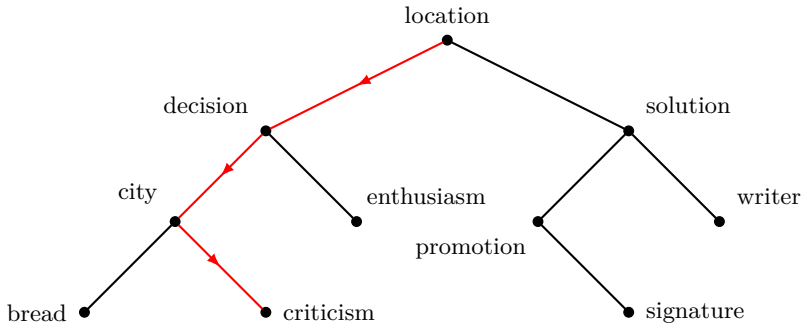
foster, dry, inspire, feign, courage, persist, reactor, inn, advisor, divorce

EXAMPLE 3 SEARCHING WITH A BINARY SEARCH TREE

Using the binary tree created in the last example, shown below, search for the word *criticism*. How many steps (comparisons) are needed to find it?



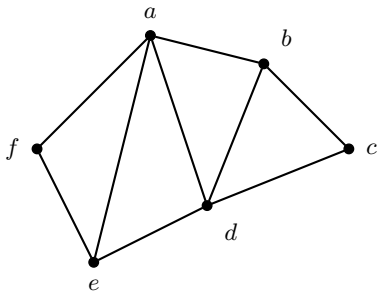
Solution First, compare *criticism* to the word at the top of the tree, *location*. Since our word comes before it, turn to the left. The next comparison is to *decision*; again, our word is before it, so we go to the left and find *city*. This time, our word comes after it, so we turn to the right and find *criticism*.



Notice that we used 3 comparisons : we compared our word to *location*, *decision*, and *city*.

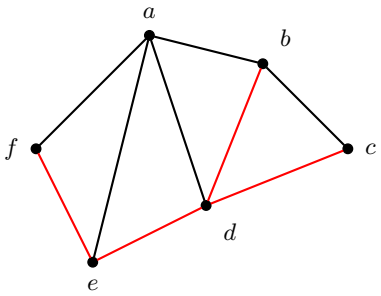
Spanning Trees

Suppose the graph below represents a network of roads, and your job is to plow the snow from these roads.

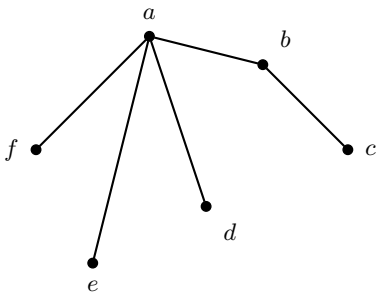


We've thought about a situation like this before, when we discussed Euler paths and looked for a way to plow *all* the roads as efficiently as possible. Now, however, let's ask a different question: if we have limited resources, what roads do we need to plow *at a minimum* so that no one gets stranded? In other words, what edges can we delete from this graph while leaving it connected?

If you notice, that brings us back to the definition of a tree: once we delete edges until we can't do so anymore without disconnecting the graph, the result will be a tree. There are many possibilities for how to do this; as an example, let's delete the edges marked below in red.



The result will be this tree:



This is an example of a **spanning tree**.

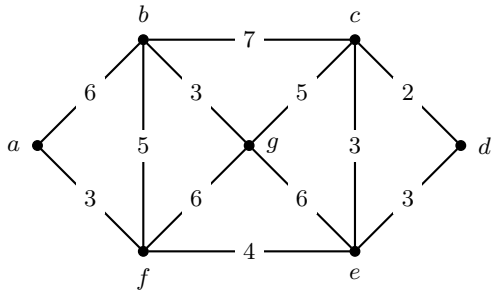
Spanning Trees

A **spanning tree** of an undirected graph is a tree that contains all the nodes of the graph.

as long as a graph is connected and undirected, it will have at least one spanning tree

Spanning trees are valuable when we want to be efficient in our use of connections, while still ensuring that all the nodes are connected. For instance, if we were building a communication network, we may want to lay as few cables as possible, in which case a spanning tree would come in handy. Every time you use the Internet, the routers and switches between you and whatever server you're connecting to use a spanning tree to avoid loops and make your connection as fast as possible.

Finding a Spanning Tree for a Weighted Graph: if we simply need to find *any* spanning tree, that's pretty simple: just start deleting edges until you can't delete any more without disconnecting the graph. However, if we add weights to the graph, the problem gets more interesting. For instance, consider the graph below.

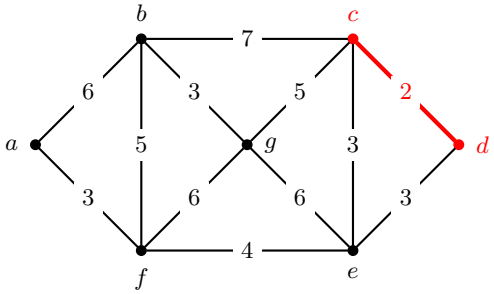


Let's say the weight of each edge is the cost of the cable between those two nodes. To reduce costs, we're going to build the minimum number of edges, so we're looking for a spanning tree. However, some spanning trees are going to be cheaper than others, and we want to find the least expensive spanning tree possible. There are a few ways to do this, but we'll use one called **Kruskal's algorithm**, which is pretty straightforward.

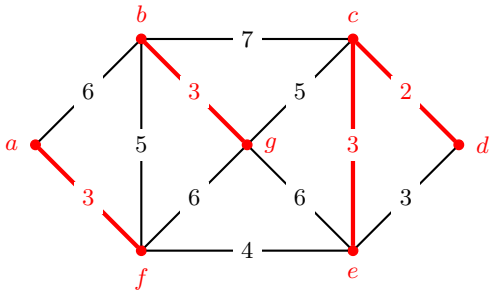
Kruskal's Algorithm

To build a **minimal spanning tree** for a graph, start with the edge with the lowest weight. Then choose the edge with the next lowest weight, and as long as adding that edge would not form a circuit, add it. Continue doing this until you have a spanning tree (stop as soon as all the nodes are included).

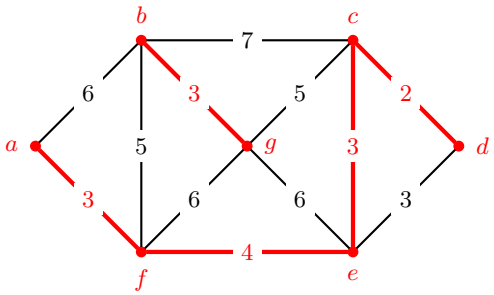
Let's see this in action, using the graph we just saw. The cheapest edge is the one connecting c and d, so we'll start with that one (we'll color it red to keep track of it).



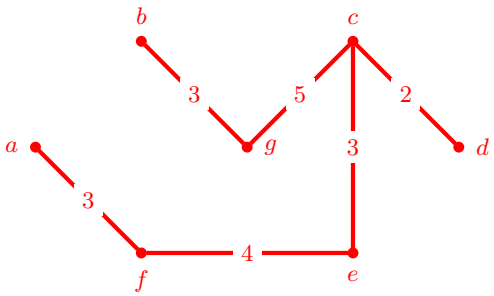
The next cheapest edge has a weight of 3; there are four such edges, but notice that if we add all of them, we'll form a circuit $c \to d \to e \to c$. Therefore, let's add three of them, leaving out the edge between d and e:



The next lowest weight is 4: add that edge (between e and f).

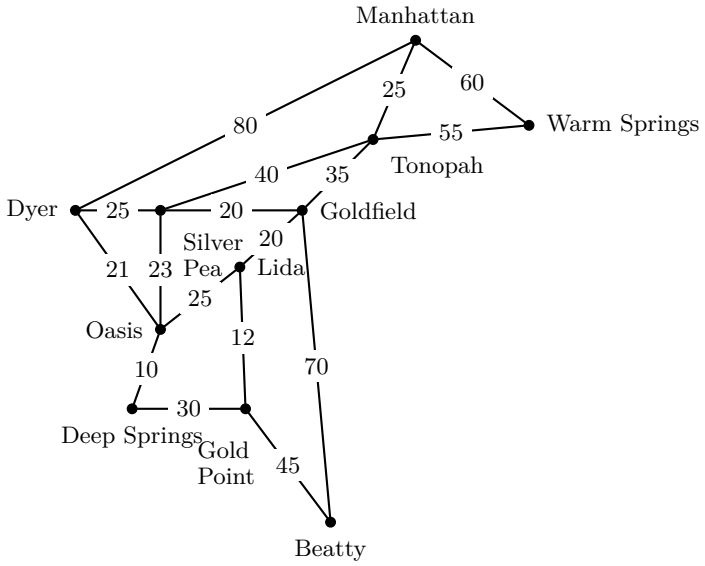


Finally, there are two edges with a weight of 5, but if we add both of them, circuits will be possible ($g \to b \to f \to e \to c \to g$, for example). Therefore, we'll just add one of them (it doesn't matter which we pick): we'll add the edge between c and g. The spanning tree looks like this:

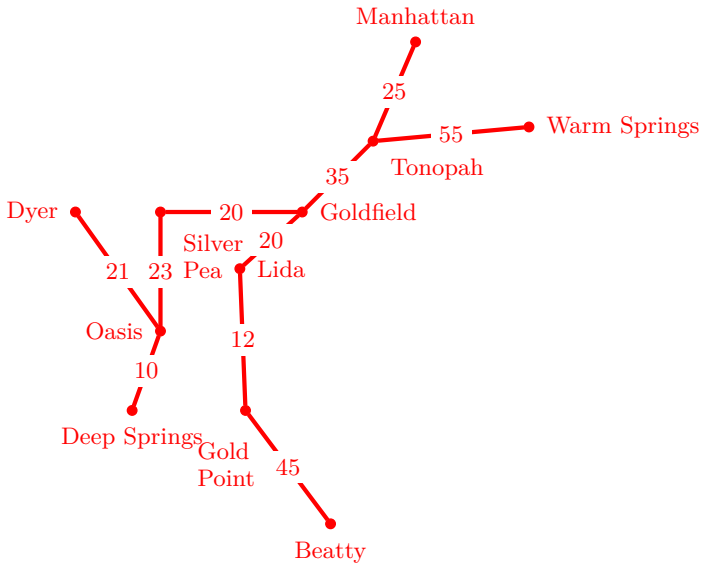


EXAMPLE 4 FINDING A MINIMAL SPANNING TREE

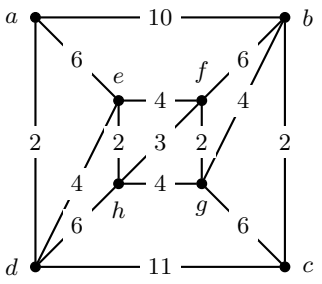
The graph below shows a network of roads between towns in Nevada. The roads shown on the graph are unpaved, and the weights represent the length of each road. Which roads should be paved so that there is a path of paved roads between every pair of towns, and the total length of paved road is as short as possible? In other words, find a minimal spanning tree for this graph.



We'll leave the details of the solution unwritten, but if you follow the process outlined above, one possible minimal spanning tree will look like this:



Find a minimal spanning tree for the graph shown in the margin.

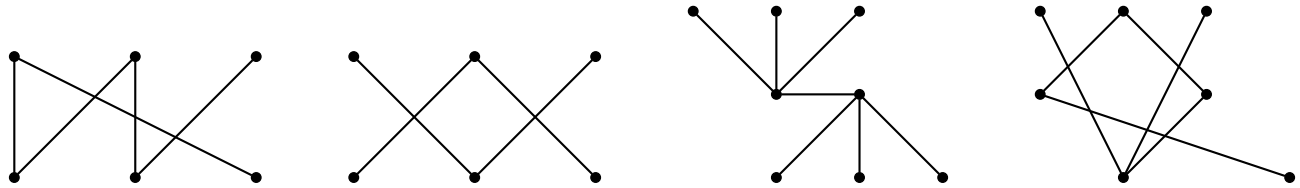


Exercises 8.4

1. Which of the following graphs are trees?

(a) (b) (c) (d)

2. Which of the following graphs are trees?



(a) (b) (c) (d)

3. Build a binary search tree for the following numbers, sorted by value: 15, 29, 9, 11, 2, 31, 18, 3, 14, and 6. Add numbers to this tree in the order in which they are listed.

(a) How many comparisons are needed to locate 11 in this tree, starting from the top?

(b) How many comparisons are needed to add 17 to this tree?

(c) What is the parent of the node labeled 2?

(d) List the children of the node labeled 29.

4. Build a binary search tree for the following words, sorted alphabetically: *gaffe, rebellion, fool, elaborate, spread, joke, freedom, stroke, guideline,* and *aware*. Add words to this tree in the order in which they are listed.

(a) How many comparisons are needed to locate *spread* in this tree, starting from the top?

(b) How many comparisons are needed to add the word *thorough* to this tree?

(c) What is the parent of the node labeled *elaborate*?

(d) List the children of the node labeled *fool*.

5. Build a binary search tree for the following list of countries, sorting them by population. Add countries to this tree in the order in which they are listed.

Country	Population (millions)
Philippines	108
Vietnam	96
Bangladesh	163
France	65
Mexico	128
Germany	84
Tanzania	58
Nigeria	201
Russia	146
Italy	61

(a) What is the parent node of Tanzania?

(b) How many children does the Mexico node have?

(c) How many comparisons are needed to locate Nigeria in this tree, starting from the top?

6. Build a binary search tree for the following list of Major League Baseball teams, sorting them by total payroll. Add teams to this tree in the order in which they are listed.

Team	Payroll (millions)
Cardinals	62
Cubs	70
Rangers	53
Reds	51
Phillies	63
Rockies	46
Red Sox	43
Brewers	37
Royals	32
Mariners	27

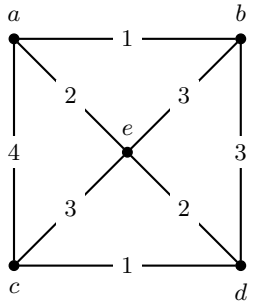
(a) What is the parent of the Phillies node?

(b) List the children of the Red Sox node.

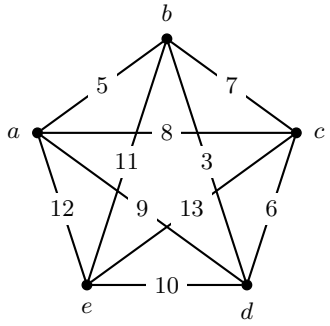
(c) How many comparisons are needed to locate the Royals in this tree, starting from the top?

In problems 7–9, use Kruskal's algorithm to find a minimum spanning tree for the given graph.

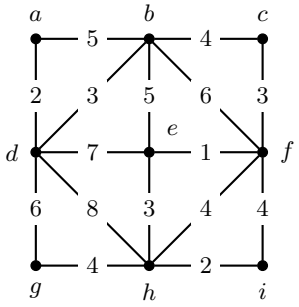
7.



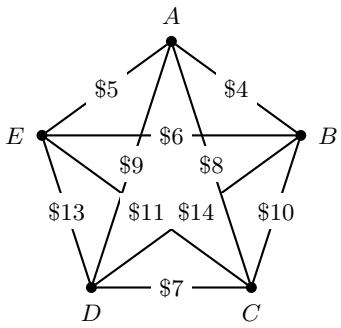
8.



9.

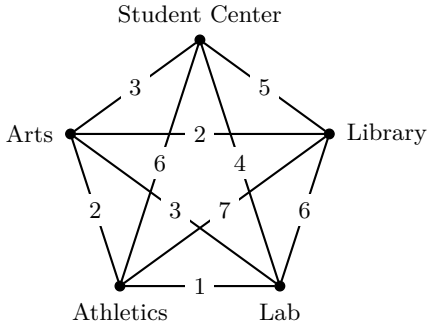


10. A company requires reliable intranet and phone connectivity between their five offices (labeled A through E), so they decide to lease dedicated lines from the phone company. The phone company will charge for each link made. The graph below shows the costs, in thousands of dollars per year, for each link.



In order to save on costs, design a network that will connect these five offices with the lowest possible cost.

11. A maintenance team is responsible for a group of five buildings on campus. These buildings are shown in the graph below, with the distance given between each pair of buildings. After a blizzard, the team is tasked with clearing the snow, but there is not enough time to clear all the walkways.



Which walkways should the maintenance team plow in order to connect all the buildings, while minimizing the time needed to do so (really, by minimizing the distance)?

12. A power company needs to lay updated distribution lines connecting eight cities in Virginia to the power grid. The distances between these cities are given in the table below. Design a network that will minimize the amount of new line.

	Purcellville	Leesburg	Middleburg	Chantilly	Sterling	McLean	Arlington	Annandale
Purcellville	–	8	11	23	19	32	37	35
Leesburg	8	–	14	17	10	24	29	27
Middleburg	11	14	–	18	16	30	34	31
Chantilly	23	17	18	–	8	13	18	13
Sterling	19	10	16	8	–	15	20	17
McLean	32	24	30	13	15	–	5	7
Arlington	37	29	34	18	20	5	–	6
Annandale	35	27	31	13	17	7	6	–

What is the total required length of line that must be laid?