# Network Models and Algorithms

## Introduction

A *network* is a way to represent relationships or connections between pairs of objects or people. *Network science* is a field that applies the mathematical results of graph theory to study network models of real world problems. Examples of networks you may be familiar with are social networks, road networks, brain (neuronal) networks, food chains, power grids, and the inter**net**. Networks are all around us! This packet serves as an introduction to the fundamentals of graph theory and focuses on one of the most important concepts in network science: path finding.

Network science has largely become a computationally- and data-driven field, with the rise in scientific computing and "big data". In Section 3, we cover two network algorithms (sets of instructions used to solve a problem, typically performed by a computer) used to find shortest paths in a network. We also provide a brief overview of programming in Python, but you are encouraged to dive deep into learning about Python and computer programming if you have time!

Some problems are marked as Challenge! problems, which focus on using the concepts you've learned and critical thinking to explain why some mathematical results must be true. These problems are optional, but everyone should at least give them a try.

Finally, learning new concepts takes time and sometimes a little help. You may need to re-read some sections multiple times before you finally "get" it. Don't be afraid to ask plenty of questions! If something doesn't make sense, then your team and team leader are there to help find the missing links to connect the dots.

# Contents

# 1 Graph theory basics

*Graph theory* is the study of mathematical structures called graphs, also known as networks. In the following section, you will be introduced to the fundamental properties of networks, as well as the basics of constructing a network model.

## 1.1 Definitions

A *network* (or *graph*) is a mathematical structure used to represent pairwise relationships between objects. Networks are composed of two features: *nodes* (or *vertices*), which represent objects; and *edges* (or *links*), which represent relationships. Visually, nodes are drawn as points or circles and edges as lines connecting the nodes. If two nodes are connected by an edge, they are said to be *adjacent* (or *neighbors*), and the number of edges connected to a node is its *degree*.

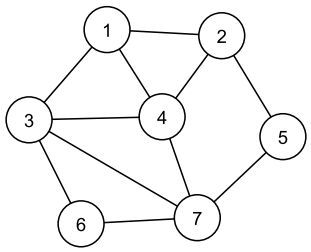



Figure 1: A network of 7 nodes and 11 edges. Node 1 has three neighbors: nodes 2, 3, and 4, each of which is connected to node 1 by an edge. The degree of node 1 is therefore 3.

For our purposes, we will focus on *simple networks*, which are networks where any two distinct nodes can only share at most one edge and no edge connects a node to itself.

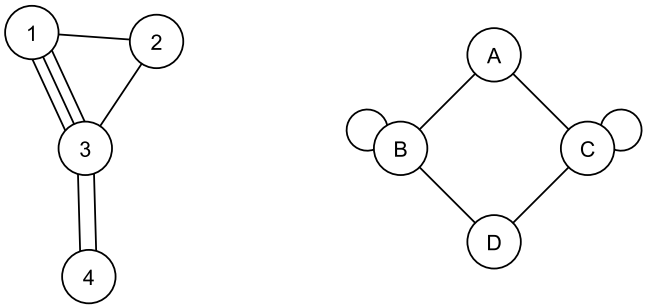



Figure 2: Two cases of **non-simple** networks. The network on the left has nodes sharing multiple edges, while the network on the right has "self-loops" (edges from a node to itself). The network on the left is called a *multigraph*.

Networks can be used to model all kinds of systems that we interact with daily. For example, in a **social network**, each person is represented by a node with edges representing friendships. Social networks can be used in a range of applications from sociology (are there distinct cliques in a high school friendship network?) to epidemiology (how easily will the flu spread through a town?) and many others.

Another application probably everyone is familiar with is **road networks**. In particular, we want to navigate from location A to location B along the quickest route possible. In road networks, typically each node represents an intersection, and two intersections are joined by an edge if there is a road directly connecting them. A visual representation of a road network then ends up looking just like a simplified map, where the edges are the roads!

When creating a network model, we may need to consider more than just what each node and edge represents. Even two networks modeling the same system can be different depending on what questions we want to ask about it.

### 1.1.1 Undirected vs. directed

While edges represent pairwise relationships within a network, not all relationships go both ways. For example, on Instagram it is possible to follow others who do not follow you. This one-way relationship would distinguish the Instagram social network from, say, the Facebook social network, in which being "Facebook friends" requires both people to be friends of each other.

Networks in which the direction of relationships matter are called *directed* networks. Networks where the direction doesn't matter or where the relationships always go both ways are called *undirected* networks.

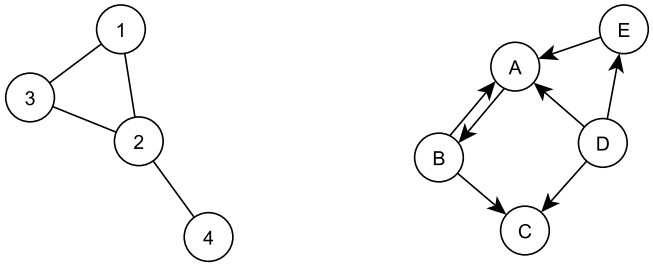



Figure 3: (Left) An undirected network of 4 nodes and 4 edges. (Right) A directed network of 5 nodes and 7 edges. Notice that each edge of the directed network has an associated direction. If two nodes in a directed network have a two-way relationship, it requires one edge in each direction (such as between nodes $A$ and $B$).

For directed networks, we distinguish between the *in-degree* and *out-degree* of a node, which is the number of incoming and outgoing edges, respectively. In the above figure, node $A$ has an in-degree of 3 (for the three edges pointing into $A$) and an out-degree of 1 (for the one edge coming out of $A$).

### 1.1.2 Unweighted vs. weighted

In our road map example, we want to find the shortest route to our destination. This requires us to track the distance we would need to travel between each intersection along our roads (the edges). Each edge can then be assigned a numerical value corresponding to this distance. This number assigned to an edge is called its *weight* (or its *length* when the weight represents a distance).

A network where each edge has a weight is said called a *weighted* network (or more precisely, an edge-weighted network). Not all network models require edge weights, such as our social network examples. Networks without weights are called *unweighted* networks.

**Exercise 1: Network models**

For each of the following scenarios, create a network model that would help us study the problem. In particular, be sure to describe:

- What do nodes and edges represent?
- Should the network be undirected or directed, and why?
- Should the network be unweighted or weighted, and why? If weighted, what do the edge weights represent?

Keep in mind that there may be more than one correct answer!

(1) Wikipedia pages are able to refer to each other through a direct hyperlink. We want to determine the most "important" Wikipedia page, based on the number of other pages that link to that page.

(2) A flu outbreak has occurred in Maryland. We want to predict how fast the illness will spread throughout the population and test various countermeasures through a mathematical model.

(3) We need to determine the fastest way to send a message through the internet. Here we use "internet" to refer to the physical system of computers, routers, servers, satellites, and other devices joined by wired and wireless connections.

## 1.2 Network representations

One way to represent a network, as we have seen, is through drawings. While drawings can help us visualize a network, they are not as useful if we want to mathematically analyze a network. In the following section, we discuss two alternative ways to represent a network: using an adjacency matrix and using an adjacency list.

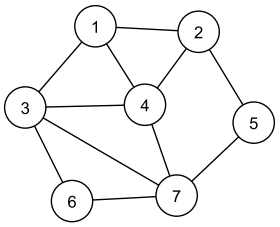
### 1.2.1 Adjacency matrix

A *matrix* is a mathematical object that can be thought of as a table of numbers. For example, consider the matrix $A$ given by

$$A = \begin{bmatrix} 2 & -1 & 0 \\ 5 & 2 & -7 \end{bmatrix}.$$

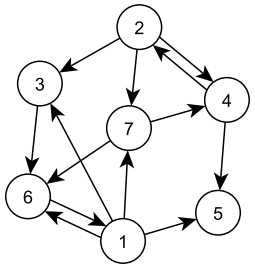
This matrix has two *rows* and three *columns*. We can refer to each entry in the matrix using index notation $[A]_{ij}$, where $i$ is the row number and $j$ is the column number. For example, $[A]_{1,2}$ refers to the entry in the first row, second column ($[A]_{1,2} = -1$, in our example case).

We can use matrices to represent a network through an *adjacency matrix*. In an adjacency matrix, entry $[A]_{i,j} = 1$ if there is an edge from node $i$ to node $j$ (i.e., if the nodes are **adjacent**). Otherwise, the entry is 0.




$$A = \begin{bmatrix} 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 1 & 1 & 0 \end{bmatrix}$$

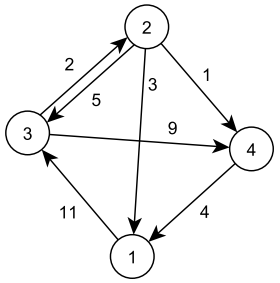
Figure 4: An undirected network and its associated adjacency matrix. Row $i$, column $j$ contains a 1 if there is an edge between nodes $i$ and $j$. Notice that since edges are undirected, an edge between $i$ and $j$ implies an edge between $j$ and $i$ (it's the same edge!), making $[A]_{ij} = 1$ **and** $[A]_{ji} = 1$. A matrix where $[A]_{ij} = [A]_{ji}$ for any choice of $i$ and $j$ is called a *symmetric matrix*. An undirected network will always have a symmetric adjacency matrix.




$$A = \begin{bmatrix} 0 & 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 \end{bmatrix}$$

Figure 5: A directed network and its associated adjacency matrix. Just like for undirected networks, row $i$, column $j$ contains a 1 if there is an edge from node $i$ to node $j$. Since edges have directions, however, this tells us nothing about whether there is an edge from $j$ to $i$. The adjacency matrix for a directed network is rarely symmetric.

We can also use matrices to organize edge weights through a *weight matrix*, usually denoted by $W$, where entry $[W]_{ij}$ is the weight of the edge from node $i$ to node $j$, if that edge exists at all. If the edge does not exist, we assign some default value, typically 0. If 0 is an allowed edge weight, then we need to look at both the adjacency matrix and the weight matrix to know what the network looks like.




$$A = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix}, \quad W = \begin{bmatrix} 0 & 0 & 11 & 0 \\ 3 & 0 & 5 & 1 \\ 0 & 2 & 0 & 9 \\ 4 & 0 & 0 & 0 \end{bmatrix}$$

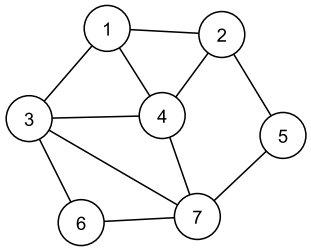
Figure 6: A weighted directed network along with its adjacency and weight matrices. In this case, since all weights are positive (non-zero), the weight matrix tells us the same information as the adjacency matrix Specifically, $[W]_{ij} = 0$ exactly where $[A]_{ij} = 0$, and similarly $[W]_{ij} \neq 0$ exactly where $[A]_{ij} = 1$.

One way we can use an adjacency matrix to analyze a network is through counting paths and cycles, which will be discussed in a later section.

### 1.2.2 Adjacency list

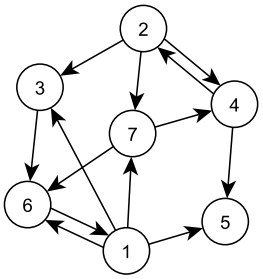
It is common for an adjacency matrix to be filled with mostly zeros, especially for very large networks. For example, a directed network with 100 nodes and 1000 edges will have an adjacency matrix that is 90% filled with zeros. A matrix that is more than half filled with zeros is called a *sparse matrix*, and a network with a sparse adjacency matrix is called a *sparse network*.

An alternative way to represent a network, particularly for sparse networks, is through an *adjacency list*. In an adjacency list, each node is followed by a list of only the nodes to which it has an edge.




$1 : 2, 3, 4$
$2 : 1, 4, 5$
$3 : 1, 4, 6, 7$
$4 : 1, 2, 3, 7$
$5 : 2, 7$
$6 : 3, 7$
$7 : 3, 4, 5, 6$

Figure 7: The same undirected network as before with its adjacency list representation. The first line lists all nodes that node 1 is connected to, the second line lists all nodes that node 2 is connected to, etc. Notice that again because the network is undirected, if we list that 1 is connected to 2, we must also list that 2 is connected to 1. The adjacency list reduced the 49 entry adjacency matrix to 22 pairs of numbers.




$1 : 3, 5, 6, 7$
$2 : 3, 4, 7$
$3 : 6$
$4 : 2, 5$
$5 :$
$6 : 1$
$7 : 4, 6$

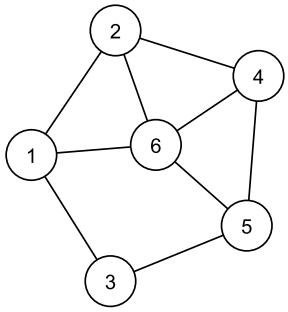
Figure 8: The same directed network as before with its adjacency list representation. Each line only lists outgoing edges, so the first line indicates which nodes are connected to node 1 by an edge starting at node 1. The adjacency list reduced the 49 entry adjacency matrix to 13 pairs of numbers.

In general, a sparse network is not an issue unless we are trying to analyze a very large network computationally. In our earlier example of 100 nodes with 1000 edges, an adjacency list would only require storing 1000 pairs of numbers, rather than an adjacency matrix's 10,000 values (most of these values being zero!). Not only is this more memory efficient, but we can also quickly determine which edges we need to care about: the adjacency list has them all listed for you front and center. If we wanted to determine which edges matter from an adjacency matrix, we would have to go through every entry, paying attention to only the 1s and ignoring all the 0s.

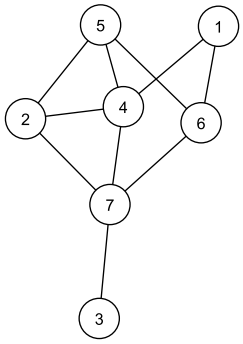
**Exercise 2: Creating adjacency matrices and adjacency lists**

For each of the following network drawings, write down the adjacency matrix and adjacency list representations. It may be easier to start with the adjacency list and use it to create the adjacency matrix.

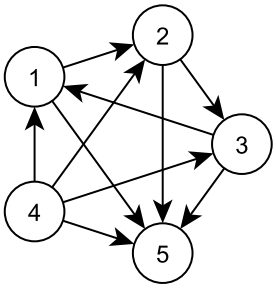
(a) Network 1



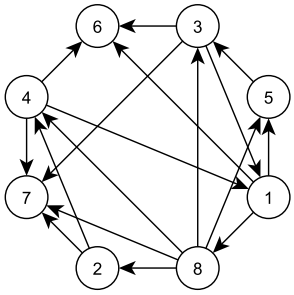

(b) Network 2




(c) Network 3




(d) Network 4




**Exercise 3: Creating a network from an adjacency matrix or list**

Draw the network based on its adjacency matrix or adjacency list below. Which networks are directed and which are undirected?

(a) Network 1

$$A = \begin{bmatrix} 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 \end{bmatrix}$$

(b) Network 2

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 \end{bmatrix}$$

(c) Network 3

$1 : 2, 3, 4$
$2 : 1, 4$
$3 : 1, 5$
$4 : 1, 2, 5$
$5 : 3, 4$

(d) Network 4

$1 : 2, 5$
$2 : 1, 3$
$3 : 1, 2$
$4 : 2, 3$
$5 : 2$

# 2 Paths and cycles

A common question is networks is identifying paths between nodes, such as in our road network application. Formally, a *path from $i$ to $j$* is a sequence of adjacent nodes, starting with $i$ and ending with $j$. If a path starts and ends at the same node, it is called a *cycle*. The number of edges involved in the path or cycle is its *length*.

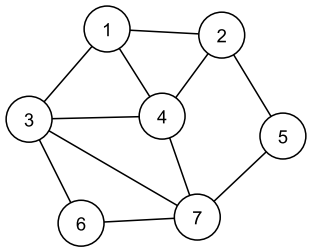



Figure 9: We can follow a path of length 2 from node 1 to node 7 by following the edges $1 \to 4 \to 7$. Alternatively, we can follow a path of length 4 by following $1 \to 2 \to 5 \to 7$. The edges and nodes in a path do not need to be distinct, so we could also follow the path $1 \to 2 \to 4 \to 1 \to 3 \to 1 \to 3 \to 6 \to 7$, which has a length of 8. This network has many cycles. The cycles $1 \to 2 \to 4 \to 3 \to 1$ and $1 \to 2 \to 1 \to 2 \to 1$ are both cycles of length 4 that include node 1.

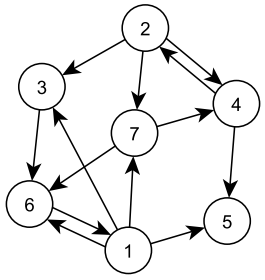



Figure 10: When looking at paths on directed networks, we can only follow edges in the direction that they point. We can construct a path from node 1 to node 2 ($1 \to 7 \to 4 \to 2$), which has a length of 3. Notice that we cannot construct **any** path that starts with node 5. Cycles are also much rarer in directed networks. This network has at least two cycles: $2 \to 7 \to 4 \to 2$ (length 3) and $1 \to 7 \to 4 \to 2 \to 3 \to 6 \to 1$ (length 6). The cycle $2 \to 7 \to 4 \to 2 \to 7 \to 4 \to 2$ is also a cycle of length 6, even though it only visits three distinct nodes.

## 2.1 Adjacency matrix method

The adjacency matrix is not just a "numerical" way to represent a network; it also has some properties that let us count the number of paths and cycles in a a network. In order to see the true power of the adjacency matrix, must first discuss matrix multiplication.

### 2.1.1 Matrix multiplication

Multiplying matrices together is quite different from multiplying two numbers. There are certain restrictions about which matrices are allowed to be multiplied, but if we focus on square matrices

of the same size we won't have to worry about these restrictions. A *square* matrix is a matrix with the same number of rows as columns. Every adjacency matrix must be square, since the numbers of rows and number of columns both equal the number of nodes.

Suppose we want to multiply two matrices $A$ and $B$ to get a new matrix $C = AB$. To get entry $[C]_{ij}$, we multiply the $i$th **row** of $A$ by the $j$th **column** of $B$. To multiple a row by a column, we multiply each corresponding entry (first entry in the row by first entry in the column, second entry in the row by second entry in the column, etc.) and add them together. For square matrices with $n$ rows (columns), this means

$$[C]_{ij} = [A]_{i,1}[B]_{1,j} + [A]_{i,2}[B]_{2,j} + \cdots + [A]_{i,n-1}[B]_{n-1,j} + [A]_{i,n}[B]_{n,j}.$$

For example, if we had the matrices

$$A = \begin{bmatrix} 2 & -2 \\ 1 & 4 \end{bmatrix}, B = \begin{bmatrix} 5 & 2 \\ -3 & 9 \end{bmatrix},$$

then their product $C = AB$ would be

$$C = \begin{bmatrix} (2)(5) + (-2)(-3) & (2)(2) + (-2)(9) \\ (1)(5) + (4)(-3) & (1)(2) + (4)(9) \end{bmatrix} = \begin{bmatrix} 16 & -14 \\ -7 & 38 \end{bmatrix}.$$

Notice that the order of the matrices **does** matter. For instance, if we instead wanted to find $D = BA$, the result would be

$$D = \begin{bmatrix} (5)(2) + (2)(1) & (5)(-2) + (2)(4) \\ (-3)(2) + (9)(1) & (-3)(-2) + (9)(4) \end{bmatrix} = \begin{bmatrix} 12 & -2 \\ 3 & 42 \end{bmatrix}.$$

Unlike multiplication of numbers, **matrix multiplication does not commute**, meaning $AB \neq BA$ in most cases.

Now that we know how to multiply matrices, we also know how to take **powers** of matrices. The matrix $A^2$ is defined as $A^2 = AA$ (just like how $x^2 = xx$ if $x$ is a number). In general, the matrix $A^n$ is the matrix $A$ multiplied by itself $n$ times using the rules of matrix multiplication. To find higher powers of matrices, we need to first find all the lower powers. For example, to find $A^3 = AAA$ we would calculate either $A^3 = A^2A$ or $A^3 = AA^2$ (this is one of the few special cases where matrix multiplication **does** commute). Through either approach, finding $A^3$ requires first calculating $A^2$.

**Exercise 4: Matrix multiplication**

Consider the following matrices

$$A = \begin{bmatrix} -1 & -1 & 0 \\ 5 & 4 & 6 \\ 2 & -4 & 5 \end{bmatrix}, \quad B = \begin{bmatrix} 1 & 6 & 5 \\ 5 & -3 & 4 \\ -4 & 5 & -3 \end{bmatrix}, \quad C = \begin{bmatrix} 4 & -3 \\ 5 & 3 \end{bmatrix}.$$

Compute the following matrix multiplications.

(1) $AB$ (2) $BA$ (3) $C^3$

### 2.1.2 Using powers of the adjacency matrix

Another way to think about an adjacency matrix $A$ is that it tells us where all the **paths of length 1** are located. If there is a path of length 1 from node $i$ to node $j$, then $[A]_{ij} = 1$. If there is no path between those two nodes, then $[A]_{ij} = 0$.

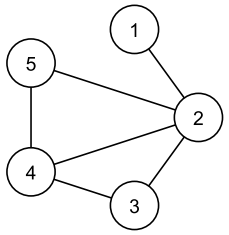
Let's now look at the matrix $A^2$. The entry in row $i$, column $j$ is given by

$$[A^2]_{ij} = [A]_{i,1}[A]_{1,j} + [A]_{i,2}[A]_{2,j} + \cdots + [A]_{i,n-1}[A]_{n-1,j} + [A]_{i,n}[A]_{n,j}.$$

Keeping in mind that each entry is either a 1 or a 0, each term in the above sum will also be either a 1 or 0. The only way for a term $[A]_{ik}[A]_{kj}$ to be 1 is if **both** $[A]_{ik} = 1$ **and** $[A]_{kj} = 1$, i.e., if there is a path of length 2 from node $i$ to node $j$ through some other node $k$ (the full path being $i \to k \to j$). Summing over all values of $k$ then counts how many paths of length 2 exist between node $i$ and node $j$. Therefore, $[A^2]_{ij}$ tells us how many **paths of length 2** exist in the network starting from node $i$ and ending with node $j$. Since a cycle is a path that starts and ends at the same node, $[A^2]_{ii}$ then tells us how many **cycles of length 2** exist in the network starting (and ending) with node $i$.

We can continue this reasoning to higher powers of the adjacency matrix and discover that $[A^n]_{ij}$ equals the number of **paths of length *n*** in the network that start with node $i$ and end with node $j$, and $[A^n]_{ii}$ equals the number of **cycles of length *n*** in the network beginning and ending with node $i$.

Let us consider the following network, shown with its adjacency matrix.




$$A = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \end{bmatrix}$$

To find how many paths of length 2 exist between nodes, we take

$$A^2 = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 & 1 & 1 \\ 0 & 4 & 1 & 2 & 1 \\ 1 & 1 & 2 & 1 & 2 \\ 1 & 2 & 1 & 3 & 1 \\ 1 & 1 & 2 & 1 & 2 \end{bmatrix}.$$

Entry $[A^2]_{2,4} = 2$, telling us that there are two paths of length 2 from node 2 to node 4. From the network, we can see that these are $2 \to 5 \to 4$ and $2 \to 3 \to 4$. Entry $[A^2]_{2,2} = 4$, meaning that there are four **cycles** of length 2 that start and end at node 2. These cycles are just starting at node 2, taking one of its edges, and returning along the same edge (for example, $2 \to 4 \to 2$). In total, there are $1 + 4 + 2 + 3 + 2 = 12$ cycles of length 2 in the network (the sum of each $[A^2]_{ii}$). Notice, however, that this counts $1 \to 2 \to 1$ and $2 \to 1 \to 2$ as two different cycles, even though they are really the same cycle starting at different nodes (or "rotated"). If we consider rotated cycles to be the same, there are actually $12/2 = 6$ distinct cycles.

If we want to now look at paths of length 3, we now calculate

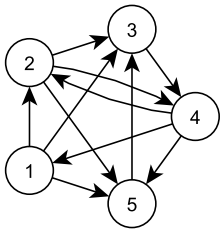
$$A^3 = AA^2 = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & 1 & 1 & 1 \\ 0 & 4 & 1 & 2 & 1 \\ 1 & 1 & 2 & 1 & 2 \\ 1 & 2 & 1 & 3 & 1 \\ 1 & 1 & 2 & 1 & 2 \end{bmatrix} = \begin{bmatrix} 0 & 4 & 1 & 2 & 1 \\ 4 & 4 & 6 & 6 & 6 \\ 1 & 6 & 2 & 5 & 2 \\ 2 & 6 & 5 & 4 & 5 \\ 1 & 6 & 2 & 5 & 2 \end{bmatrix}.$$

Entry $[A^3]_{4,1} = 2$ indicates that there are two paths of length 3 from node 4 to node 1 ($4 \to 3 \to 2 \to 1$ and $4 \to 5 \to 2 \to 1$, looking at the network). To find how many cycles of length 3 start at node 5,

we check entry $[A^3]_{5,5} = 2$. Again, looking back at our network, we see that these are $5 \to 2 \to 4 \to 5$ and $5 \to 4 \to 2 \to 5$. Notice that these are also the same cycles, but one is the reversal of the other. If we consider reversals and rotations as all the same cycle, we will end up counting the same cycle 6 times (double for reversals and triple for rotations). The total number of distinct cycles of length 3 are then $(0+4+2+4+2)/6 = 12/6 = 2$. These two cycles are $5 \to 2 \to 4 \to 5$ and $2 \to 4 \to 3 \to 2$.

**Exercise 5: Counting paths and cycles from the adjacency matrix**

Consider the following network.




(1) Write down the adjacency matrix for the network.

(2) Use the adjacency matrix and matrix multiplication to find how many paths of length 2 there are for each of the following. Use the network to determine what those paths are.

   (a) starting from node 1 and ending with node 4

   (b) starting from node 4 and ending with node 3

   (c) starting from node 2 and ending with node 1

(3) Use the adjacency matrix and matrix multiplication to find how many **total distinct** cycles of length 3 are in the network. Keep in mind that for a directed network we will automatically count rotations multiple times in $A^3$ but **not** reversals (since the edge directions matter).

(4) List all the distinct cycles of length 3 by looking at the network.

**Exercise 6: Six degrees of separation**

Consider the **six degrees of separation** problem, which hypothesizes that any two people on earth are separated by six or fewer social connections (everyone is a friend of a friend of a friend...). Suppose we have the adjacency matrix $A$ for the global friendship social network.

(1) Restate the hypothesis of the six degrees of separation problem in terms of paths of the social network.

(2) How can we use the adjacency matrix $A$ to test this hypothesis?

## 2.2 Connectedness

Another property of networks is their **connectedness**, which states whether all nodes in a network can reach each other through a path. If the entire network isn't connected, we can always break it down into pieces called **connected components**.

### 2.2.1 Undirected networks

A *connected component* is a set of nodes that can reach each other through at least one path. Formally, nodes $i$ and $j$ are in the same connected component if there is at least one path connecting $i$ and $j$. A network may consist of multiple connected components. In particular, every undirected network can be broken down into non-overlapping connected components, where each node belongs to exactly one component. A network made of a single connected component is a *connected* network. A network that is not connected (i.e., it contains more than one distinct connected component) is *disconnected*.

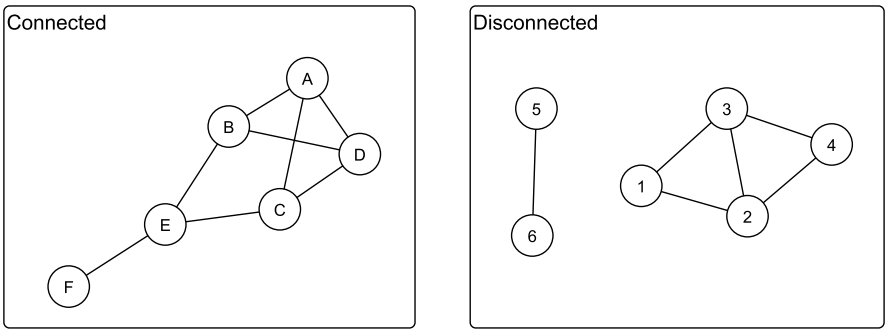


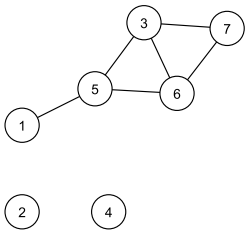

Figure 11: The network on the left consists of a single connected component (all nodes can reach each other), making it a connected network. The network on the right consists of two connected components: $\{1, 2, 3, 4\}$ and $\{5, 6\}$. Nodes within the same connected component can reach each other, but nodes in different connected components cannot. The network on the right is therefore disconnected.

In a road network, each connected component would correspond to the all destinations (intersections) that a driver can go between. When considering all roads in the world, destinations in the Americas would be one connected component, while destinations in Australia would be a separate connected component (there is no road connecting the Americas to Australia!).

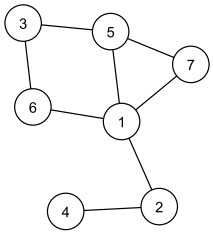
**Exercise 7: Connected components on undirected networks**

For each network, group the nodes into connected components and determine whether the network is connected or disconnected.

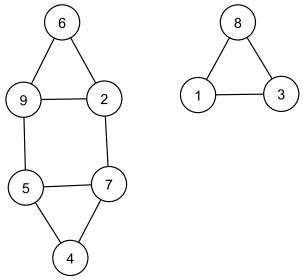
(a) Network 1




(b) Network 2




(c) Network 3



### 2.2.2 Directed networks

Directed networks have two notions of connectivity: weak and strong. A *weakly connected component* consists of first replacing all our edges with undirected edges and using the definition of a connected component for undirected networks.

A *strongly connected component* is a set of nodes that can all reach each other through **directed** paths. Nodes $i$ and $j$ are then in the same strongly connected component if there is a directed path from $i$ to $j$ **and** and a directed path from $j$ to $i$. Alternatively, we can state that nodes $i$ and $j$ are in the same strongly connected component if there is a **cycle** containing $i$ and $j$.

Just like how undirected networks can be broken down into connected components, we can break down a directed network into non-overlapping weakly connected components with each node belonging to exactly one component. A network consisting of a single weakly connected component is a *weakly connected* network. Similarly, a directed network can be broken down into non-overlapping strongly connected components with each node belonging to exactly one. A network consisting of a single strongly connected component is a *strongly connected* network. A directed network consisting of multiple weakly connected components is *disconnected*, as before.

Note that a network that is strongly connected **must** be weakly connected. However, a network that is weakly connected might not be strongly connected.

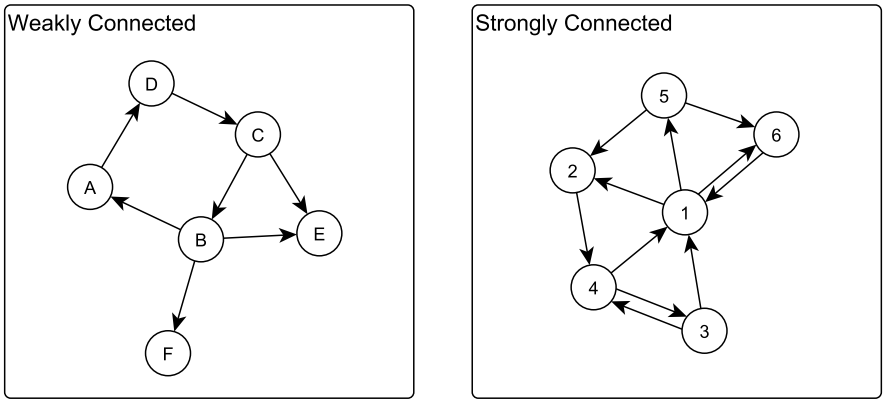



Figure 12: The network on the left is weakly connected. If we ignore the directions of the edges (treating it as an undirected network), then each node could reach any other. However, once we consider the direction, this is no longer true, indicating that it is not strongly connected. The strongly connected components of the left network are $\{A, D, C, B\}$, $\{E\}$, and $\{F\}$. The network on the right is strongly connected, since all nodes can reach each other through directed paths. Specifically, the network contains the cycle $1 \to 5 \to 6 \to 1 \to 2 \to 4 \to 3 \to 1$, which visits every node. Since the network on the right is strongly connected, it must also be weakly connected.

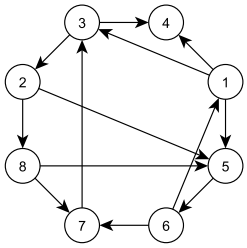
**Exercise 8: Challenge!**

Explain why a strongly connected directed network must always be weakly connected. (*Hint:* If two nodes are in the same strongly connected component, why does that mean they must also be in the same weakly connected component?)

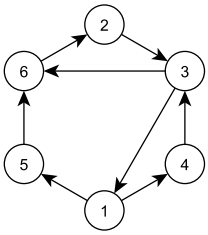
**Exercise 9: Connected components on directed networks**

For each network, group the nodes into weakly connected components and strongly connected components. Determine whether each network is weakly connected, strongly connected, or disconnected.

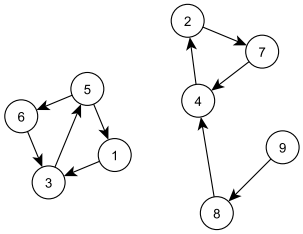
(a) Network 1




(b) Network 2




(c) Network 3




## 2.3 Eulerian paths and cycles

The **seven bridges of Königsberg** is considered to be the first problem ever studied in graph theory. The problem considers the map of Königsberg, Prussia (below) and asks if it is possible to walk through the city, crossing each bridge exactly once. Swiss mathematician Leonhard Euler (pronounced Oil-er) is credited with being the first person to solve the problem, meanwhile laying the foundation for graph theory in 1736.

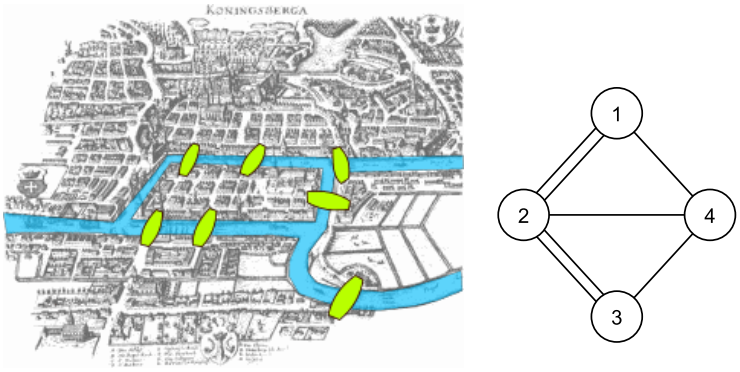



Figure 13: The original map of Königsberg from 1736, highlighting the seven bridges, and the network representation of the map.

Let us first construct a network representation of the map. Each landmass can be represented by a node, so that two nodes are connected by an edge if two landmasses are connected by a bridge. We will make each bridge a separate edge (creating a multigraph).

Euler observed that in order for this walk through Königsberg to be possible, you must be able to leave every landmass as many times as you enter, otherwise you get stuck. The exceptions are the locations you begin and end your path. Since you can only enter and exit through a bridge, **at most** two landmasses can have an odd number of bridges, with all others having an even number. In terms of our network, this means at most two nodes can have odd degree. Based on this reasoning,

it is not possible to walk through Königsberg crossing each bridge exactly once, since all four nodes have odd degree!

A path through a network that crosses every edge exactly once is called an *Eulerian path*. Euler's reasoning demonstrates that for any network to have an Eulerian path, at most two nodes can have odd degree. Notice that we have only shown that this is a **necessary** condition for an Eulerian path to exist. In other words, we only know that if a network **does not** satisfy this condition, it **cannot** have an Eulerian path. Euler believed that for connected networks, this was also **sufficient** to guarantee the existence of an Eulerian path, but this fact was not proved until over one hundred years later by German mathematician Carl Hierholzer.

We can also consider *Eulerian cycles*, which are Eulerian paths that begin and end at the same node. In this case, **every** node must have even degree. Together, we arrive at the following theorem.

**Theorem (existence of Eulerian paths and cycles).** A connected undirected network has an *Eulerian path* if and only if at most two nodes have odd degree. A connected undirected network has an *Eulerian cycle* if and only if all nodes have even degree.

**Exercise 10: Challenge!**

Explain why an Eulerian **cycle** cannot have **any** nodes with odd degree.

So far we have only considered the requirements for an Eulerian path or cycle to **exist**, but not how to actually find one. The two most common methods for finding Eulerian paths are Hierholzer's algorithm and Fleury's algorithm. For small networks, however, we can usually find these paths by inspection (looking at the network and using trial and error).

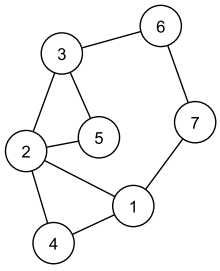
**Exercise 11: Eulerian paths and cycles**

For each of the following networks:

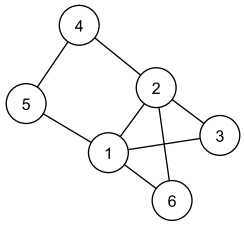
- Write down the degree of each node.
- Does the network have an Eulerian path? If so, find one.
- Does the network have an Eulerian cycle? If so, find one.

A network can have more than one Eulerian path or cycle, so there may be more than one correct answer!

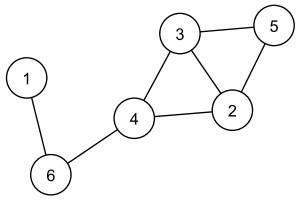
(a) Network 1




(b) Network 2




(c) Network 3



## 2.4 Hamiltonian paths and cycles

An Eulerian path is one that visits each **edge** exactly once, but what about paths that visit each **node** exactly once?

A path that visits every node in a network exactly once is a *Hamiltonian path*. If such a path is a cycle, i.e. it starts and ends at the same node, then it is called a *Hamiltonian cycle*. Unlike Eulerian paths, there is little we can say about the existence of Hamiltonian paths. However, one special case where a Hamiltonian path will always exist is a complete network. A network is *complete* if every node is adjacent to every other node, meaning every pair of nodes shares an edge.

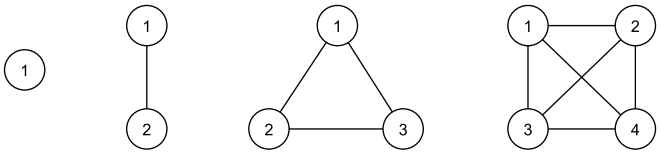



Figure 14: Complete networks for 1, 2, 3, and 4 nodes. Between any two nodes in each network, there must be an edge.

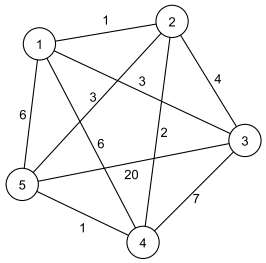
Since all nodes are adjacent, we can visit them in any order without having to pass through the same node twice, meaning we actually have many Hamiltonian paths. Also, as long as the network has at least three nodes, we can always return back to our starting point, giving us a Hamiltonian cycle.

Perhaps the most well-known problem relating to Hamiltonian paths is the **traveling salesman problem**. In this problem, we consider a salesman who has to drive to many stops all over the city. The goal is to find the order of stops that requires the least amount of total driving. We can represent this problem as a weighted network in which each stop is a node and edge weights are the total distance between each stop. We can assume that we can directly drive between any two stops (each pair of destinations share an edge), giving us a complete network. From our previous discussion, this means that many Hamiltonian paths exist, but our goal is to **find the Hamiltonian path with the smallest total path length**.

One possible solution to the traveling salesman problem is to simply test out every possible path, and just pick the one with smallest total edge weight. However, for a complete network with $n$ nodes, there are $n!$ (factorial) Hamiltonian paths. For very small networks, this may be fine, but if we have 10 nodes, that's nearly 4 million paths to check! If we have a problem with more than 10 nodes, this becomes even more unfeasible.

If we don't want to find the actual shortest path (because it will take too long to compute) and are okay with a "good enough" approximation, we can use a *heuristic method*. One simple heuristic method for the traveling salesman problem is to always take the edge with the smallest weight that leads to an unvisited node. This approach is called a "greedy" algorithm.

Consider the following network.



Let's say we want to use the greedy algorithm to find the shortest Hamiltonian path starting at node 5 (the node we start with is usually chosen at random). If we always take the edge of smallest weight, never revisiting the same node, we end up with the path $5 \to 4 \to 2 \to 1 \to 3$ with a total length of $1 + 2 + 1 + 3 = 7$. Instead, let's say we started with node 1. Using the greedy algorithm gives $1 \to 2 \to 4 \to 5 \to 3$ with a total path length of $1 + 2 + 1 + 20 = 24$. Immediately we can see that the greedy algorithm isn't perfect, since we can find a shorter Hamiltonian path that starts at 1. For example, the path $1 \to 2 \to 3 \to 4 \to 5$ has a path length of $1 + 4 + 6 + 1 = 12$.

**Exercise 12: Traveling salesmen greedy algorithm**

Use the greedy algorithm to approximate the shortest Hamiltonian path starting from node 1 for the network with the following weight matrix. Be sure to write down the path and its length. As a reminder, $[W]_{1,2}$ (row 1, column 2) is the weight of the edge connecting node 1 and node 2, $[W]_{1,3}$ is the weight of the edge connecting node 1 and node 3, and so on.

$$W = \begin{bmatrix} 0 & 1 & 4 & 2 & 7 & 2 & 3 \\ 1 & 0 & 6 & 3 & 4 & 9 & 2 \\ 4 & 6 & 0 & 7 & 1 & 3 & 3 \\ 2 & 3 & 7 & 0 & 6 & 5 & 1 \\ 7 & 4 & 1 & 6 & 0 & 3 & 3 \\ 2 & 9 & 3 & 5 & 3 & 0 & 4 \\ 3 & 2 & 3 & 1 & 3 & 4 & 0 \end{bmatrix}.$$

# 3 Algorithms

The following section makes use of terminology and syntax used in computer programming. Before proceeding, you should head to

`https://github.com/girlstalkmath-umd/patterns-and-fractals`

and complete the Python programming tutorial by opening `Python_Basics.ipynb`. Open the notebook by clicking "Open in Colab" at the top of the file. You will need a Google account.

Read the text in the tutorial and when you get to a cell (gray box), hit the play button that appears when you hover your mouse over the brackets in the top left corner of the cell. This executes the Python code in the cell. Feel free to play around with the code and see what happens!

Once you feel like you understand the basics, then continue on below to learn about different algorithms we use to analyze networks, specifically ones used in finding shortest paths.

## 3.1 Breadth-first search

**Breadth-first search (BFS)** is an algorithm used to find the shortest path in an unweighted network starting from a **start node** $s$ and ending at a **destination node** $d$. Although we only care about the shortest path from $s$ to $d$, a BFS will find all shortest paths in the network starting from $s$ as part of the process, stopping early once we find $d$. The strategy used to accomplish this is by searching the network in "layers".

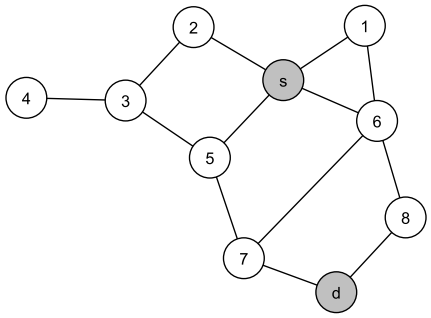
First, we check all neighbors of $s$, also referred to as *expanding* $s$. Each of these nodes has a shortest path of length 1, since they are only one edge away from $s$. Next, we check the neighbors of the neighbors of $s$, i.e., we expand each neighbor of $s$. Any of the nodes we haven't already seen are a shortest distance of 2 away from $s$. We don't want to include any nodes we've already seen, because we already know that these have a shortest path length of 1.

We then continue this process as much as needed. In general, if a node is a shortest distance $k$ away from $s$, then each of its neighbors will have a shortest distance of $k+1$, if the neighbor was not already encountered in an earlier step (in which case we already found a shorter distance). If at any stage we encounter node $d$, then we can stop, because we know we will not find any shorter path by continuing the process further. All nodes we have yet to visit will always be **at least as far** as the current nodes.

We now know how to find the shortest path **length** from $s$ to $d$, so our next challenge is to find which nodes form the shortest path itself. From our BFS process just described, we know that the first time we encounter a node $j$ it will be through finding $j$ in the shallowest layer possible. In particular, we encountered $j$ by expanding some node $i$, who had $j$ as a neighbor. Node $i$ was found as early as possible, so the shortest path from $s$ to $j$ must include the edge $i \to j$ and the entirety of the shortest path from $s$ to $i$. We refer to $i$ as the *parent* of $j$, meaning $i$ is the node leading directly to $j$ in the shortest path. By continuing this reasoning, we know that the shortest path must include the parent of $i$, and that node's parent, and so on until we reach back to $s$. This process is known as *backtracking*. Therefore, we can record the **parent** of each node as the node we **first** expanded to find it. Then, when we reach $d$, we can backtrack through the chain of parents to find the shortest path from $s$ to $d$.

### 3.1.1 Breadth-first search example

Let's work through the breadth-first search algorithm by hand for the following network.




We want to find the shortest path from $s$ to $d$. We begin by expanding node $s$, giving us all nodes that are distance 1 away:

$$\text{Layer 1: } 1, 2, 5, 6$$

We will track which nodes have been visited with a list called `visited` and the parent of each node in a list called `parent`. Their current statuses are below. A value of $-1$ in `parent` indicates that no parent node has been assigned yet.

| Node | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | $s$ | $d$ |
|---|---|---|---|---|---|---|---|---|---|---|
| `visited` | **T** | **T** | F | F | **T** | **T** | F | F | T | F |
| `parent` | ***s*** | ***s*** | $-1$ | $-1$ | ***s*** | ***s*** | $-1$ | $-1$ | $-1$ | $-1$ |

None of the nodes we visited are $d$, so we continue. We will expand each of these nodes to find all nodes that are a distance 2 away (in "layer 2"). We can expand these nodes in any order. Which order we choose doesn't affect the overall end goal; we will still find **a** shortest path. To make sure everyone gets the same answer, we will expand each node in increasing order, based on the node's label.

We expand node 1 next, but its neighbors are $s$ and 6, both of which have already been visited. Next is node 2, whose neighbors are $s$ (already visited) and 3 (a new node). We then expand 5, whose neighbors are 3 (already visited by expanding 2), $s$ (already visited), and 7 (a new node). Finally, we expand 6, giving the nodes $s$, 1, and 7 (all of which were already visited) and the new node 8. All nodes a distance of 2 away are the following.

$$\text{Layer 2: } 3, 7, 8$$

We also have to keep `visited` and `parent` up to date. Making sure we set the parent based on the **first** encounter, we get

| Node | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | $s$ | $d$ |
|---|---|---|---|---|---|---|---|---|---|---|
| `visited` | T | T | **T** | F | T | T | **T** | **T** | T | F |
| `parent` | $s$ | $s$ | **2** | $-1$ | $s$ | $s$ | **5** | **6** | $-1$ | $-1$ |

We still have yet to find $d$, so we continue the search. Next we expand node 3, whose only new neighbor is node 4. We then expand node 7, whose only new neighbor is node $d$ (our destination node!). We don't stop the program until we expand $d$, so we next expand node 8, giving no new nodes. All distance 3 nodes are

$$\text{Layer 3: } 4, d.$$

The state of `visited` and `parent` are

| Node | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | $s$ | $d$ |
|---|---|---|---|---|---|---|---|---|---|---|
| `visited` | T | T | T | **T** | T | T | T | T | T | **T** |
| `parent` | $s$ | $s$ | 2 | **3** | $s$ | $s$ | 5 | 6 | $-1$ | **7** |

Finally, we expand node 4, giving no new nodes, and then expand node $d$, telling us to stop. We now recreate the shortest path by backtracking through the parents. The parent of $d$ is 7, the parent of 7 is 5, and the parent of 5 is $s$. Therefore, a shortest path is $s \to 5 \to 7 \to d$.

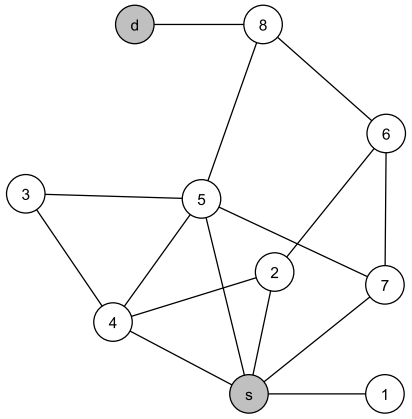
As mentioned before, if we expanded the nodes in a different order, we could have found a different shortest path (for example, $s \to 6 \to 8 \to d$). In any case, we would always have found a path of length 3.

**Exercise 13: Challenge!**

We know that any shortest path from $s$ to $d$ in the previous example must have length 3. Explain how we can use the adjacency matrix to come to this same conclusion. Furthermore, explain how we can use the adjacency matrix to know **how many** shortest paths exist between these two nodes. (You may need to refer back to Section 2.1).

**Exercise 14: Breadth-first search**

Consider the following network.




Perform a breadth-first search to find the shortest path from $s$ to $d$. Write down which nodes were found in each layer and the parent of each node. For nodes in the same layer, you should expand the nodes in increasing numerical order to make sure everyone finds the same shortest path. Keep track of `visited` and `parent` after expanding each layer, as was done in the example.

### 3.1.2 Pseudo-code algorithm

Below you can find the breadth-first search algorithm written in *pseudo-code*, which is a format based on computer code, but not written in any particular programming language. Let's analyze what each section will accomplish.

**Initialization (lines 1–7).** Before we start the algorithm, we need to *initialize* some objects by creating them and assigning initial values. First, we create the **boolean** list `visited`, which tracks

**Algorithm 1:** Breadth-first search algorithm with backtracking

```
Data: network G(V,E), starting node s, destination node d
Result: Returns the shortest path from s to d.
create the list visited
create the list parent
set found to False
create the queue Q

add s to Q
mark s as visited

while Q is not empty do
    remove front node from Q and assign to v
    if v is d then
        set found to True
        break out of the loop
    end

    foreach neighbor w of v do
        if w has not been visited then
            add w to Q
            mark w as visited
            set parent of w to v
        end
    end
end

if found then
    backtrack the path from d to s following parent
    return path from s to d
else
    return empty path
end
```

if we've seen a node before. A *boolean variable* is one that stores either the value of `True` or `False`. If `visited[k]` equals `False`, then we have not seen node $k$, and if `visited[k]` equals `True`, then we have seen node $k$ already. Before we start the algorithm, we have not seen any nodes except the starting node $s$. All values in `visited` are therefore initially set to `False`, except we set `visited[s] = True` (shown in line 7).

Next, we need to track the parent of each node, which is done in the list `parent`. If we determine that the parent of node $j$ is the node $i$, then we will set `parent[j] = i`. Before the algorithm starts, however, we set all values in `parent` equal to some default value that does not represent any node in the network. For example, if our nodes are numbered from 0 to 100, then we should set all parents equal to $-1$, since there is no node $-1$.

We then create the boolean variable `found`, which will tell us if we have found node $d$. Before we start the algorithm, we set `found` to `False`, since it has not been found yet, and we will change it to `True` if we ever encounter node $d$ in our BFS.

Finally, we create the queue `Q`. A queue is a list with "first-in, first-out" (FIFO) priority. In other words, objects are added to the "back" of the queue and removed from the "front" of the queue, just like a shopping queue. We will use `Q` to track the order we expand nodes, since the first nodes that are encountered (which will always be in the lowest layers) should be the first ones that are expanded. To start off the algorithm, we place $s$ in `Q`.

**Outer loop (lines 9–14).** One stopping criterion of BFS is that we found $d$. But if we never find $d$, we should stop once we run out of nodes to expand. The outer `while` loop controls this condition, since `Q` contains all remaining nodes we have left to expand; **while** `Q` is not empty, we should continue our search. If there are more nodes in `Q`, the frontmost node is removed (following the FIFO priority), assigned to the variable `v`, and expanded in the inner `for` loop. However, before we expand we should check if we found $d$, and if so, we can stop the search immediately by setting `found` to `True` and breaking out of the `while` loop using a `break` statement.

**Expansion (lines 16–22).** We use a `for` loop to expand the node `v` by iterating over each of its neighbors. **If** a neighbor has already been visited, we can ignore it and continue on to the next neighbor of `v`. When a neighbor `w` has not already been visited, meaning we are just now visiting it for the first time, we add `w` to `Q`, set `visited[w] = True`, and note that the parent of `w` is `v` by setting `parent[w] = v`. Once we finish expanding `v`, we return to the start of the outer loop (line 9).

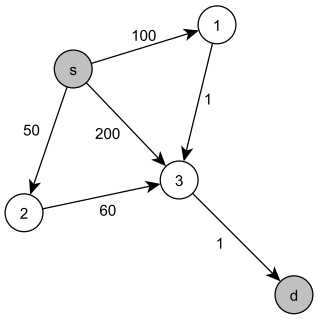
**Backtracking (lines 25–30).** If we explore the entire network and never find $d$, then we report to the user that there is no path, and therefore no shortest path, from $s$ to $d$. Otherwise, we **backtrack** by following the chain of parents from $d$ to $s$ to give the shortest path from $s$ to $d$.

## 3.2 Dijkstra's algorithm

If we want to find a shortest path on an unweighted network, breadth-first search is the algorithm to use. But what about weighted networks? For example, a road network uses nodes as intersections, and edges have weights representing the physical distance between these intersections. While breadth-first search can find the path with the fewest number of intersections involved, the physical distance between these intersections might be very large compared to a shorter physical path with more intersections.

Thankfully, we only need to make a few changes to our breadth-first search algorithm to make it work for weighted networks. The resulting algorithm is called **Dijkstra's algorithm**, named after the Dutch computer scientist Edsger W. Dijkstra (pronounced Dike-stra) who published the algorithm in 1959.

To understand some core principles behind Dijkstra's algorithm, consider the example network below.




To get from $s$ to $d$, there are three possible paths: $s \to 1 \to 3 \to d$ (length 102), $s \to 2 \to 3 \to d$ (length 111), and $s \to 3 \to d$ (length 201). We can make three observations here. First, when we look at just the neighbors of $s$, the "closest" node is node 2; however, the path that includes node 2 is not the shortest, because the later edge $2 \to 3$ adds too much length. Second, each of the paths pass through node 3, with the shortest path being the one with the shortest **subpath** to node 3. Third, the length of the path from $s$ to $d$ is the length of the subpath from $s$ to 3 plus the weight of the edge $3 \to d$. Each of these observations can be made into more general statements.

**Observation 1.** First, it is possible that a path with many "short" edges ends up not being the shortest path because of one very "long" edge. Alternatively, a large number of short edges could add up to have a greater **total length** than a single long edge. By the same reasoning, it is also possible that following many short edges will provide a shorter overall distance than a single long edge (e.g., 100 edges of length 1 vs 1 edge of length 200). Therefore, we cannot make any final conclusions about the shortest path until we have seen the **entirety of every potential path** from $s$ to $d$.

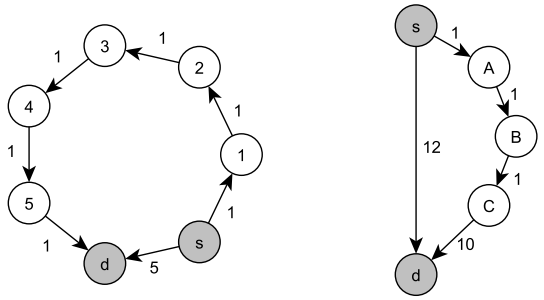



Figure 15: (Left) A network where many short paths (counterclockwise) add up to a greater length than a single "long" path from $s$ to $d$. (Right) A similar scenario, where the path through multiple edges is shorter up until the final edge is added in.

**Observation 2.** Second, suppose $i$ is some node on the shortest path from $s$ to $d$. Then, the shortest path from $s$ to $d$ is made up of the shortest (sub)path from $s$ to $i$ and the shortest (sub)path from $i$ to $d$, with the two subpaths joined end to end (or *concatenated*). This must be true for **all** nodes on the shortest path.

**Exercise 15: Challenge!**

Explain why for **any node** $i$ on the shortest path from $s$ to $d$ we can create the shortest path by concatenating the shortest paths from $s$ to $i$ and from $i$ to $d$. (*Hint:* If we could find a **shorter** path from $s$ to $i$, what would that mean about the path from $s$ to $d$?)

**Observation 3.** Finally, for any node $j$, the length of a path from $s$ to $j$ is the sum of the subpath from $s$ to $i$ plus the length of edge $i \to j$, where $i$ is the **parent** of $j$ along that path. In terms of shortest paths, if our network has the edges $i_1 \to j, i_2 \to j, \ldots, i_n \to j$, then the shortest path from $s$ to $j$ will be the one with the smallest value of

$$(\text{length of subpath from } s \text{ to } i_k) + (\text{length of edge } i_k \to j).$$

Before continuing, it is important to note that we will assume that all edge weights are non-negative whenever we use Dijkstra's algorithm. Without this restriction, one could find a shortest path of length $-\infty$ by repeatedly following an edge or a loop of negative length! Fortunately, most applications we care about only use positive edge weights. We will also assume that no edge has a weight of infinity, otherwise we would just remove that edge from the network.

### 3.2.1 The algorithm

Let's now go through the process of the algorithm. You should notice that it is very similar to BFS, but with a few changes to take into account our three observations.

As before, we begin by expanding the starting node $s$. We record the distance from $s$ to each neighbor $i_1, \ldots, i_m$ as the length of the edge $s \to i_k$ (the subpath from $s$ to $s$ has length 0) and mark the parent of $i_k$ as $s$. Next, we expand the neighbor with the shortest distance to $s$. Let's call this node $i^*$ and its neighbors $j_1, \ldots, j_n$. The distance from $s$ to $j_k$ is then the distance from $s$ to $i^*$ **plus** the edge length of $i^* \to j_k$ and the parent of $j_k$ is $i^*$. **But what if one of the neighbors of $i^*$ was also a neighbor of $s$?** If this happens, then we record a new distance and parent **only if** we found a **shorter** subpath through $i^*$.

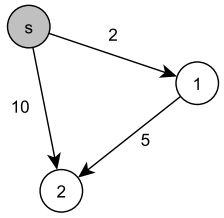



Figure 16: When expanding $s$, we see node 1 a distance of 2 away and node 2 a distance of 10 away. We then next expand node 1 and see node 2 again, but this time a distance of $2 + 5 = 7$ away. We can therefore reach node 2 **faster** through $s \to 1 \to 2$ than through $s \to 2$. So in a shortest path starting from $s$, the parent of node 2 is node 1 (not $s$).

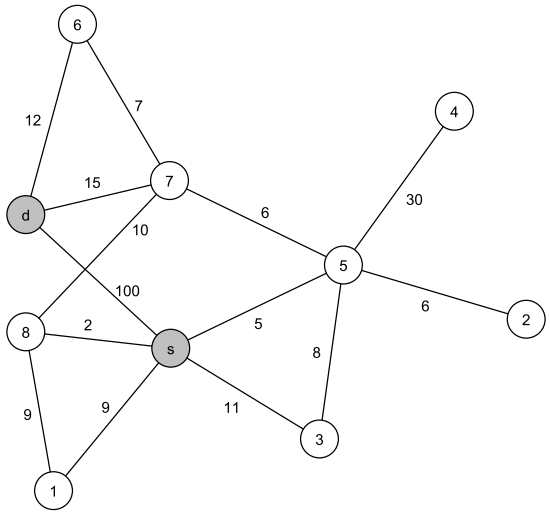
We continue in a similar manner. Eventually after many stages, we expand some node, let's call it $v$. For each of its neighbors $w$ we calculate the length of the path through $v$ as the current distance from $s$ to $v$ **plus** the length of edge $v \to w$. If this path is shorter than the current distance to $w$ (through some earlier path) **or** if this is the first time encountering $w$, we update the distance to $w$ and set its parent as $v$.

After all neighbors of $v$ have been checked, the next node we expand is the one with the **current shortest distance to $s$ that has not yet been expanded**. If we follow this order, the nodes

are always expanded in the order of shortest distance to $s$. Therefore, if $v$ is the node with the current shortest distance to $s$, we will never find an alternate path from $s$ to $v$ that is shorter. Any alternate paths must go through a node that is **at least** as far away as the current distance to $v$. If this weren't the case, we would have found a node on the alternate path sooner and expanded it first, but we didn't! Notice that this is **not** true for the current second closest node or beyond, since expanding $v$ may discover shorter alternate paths for other nodes.

### 3.2.2 Dijkstra's algorithm example

As we did with breadth-first search, let's work through Dijkstra's algorithm on an example network, shown below.




We want to find the shortest weighted path from $s$ to $d$. We start by expanding node $s$. We find five nodes: 1 (distance 9), 3 (distance 11), 5 (distance 5), 8 (distance 2), and $d$ (distance 100). Although we already found $d$ only 1 edge away, it has a fairly large edge weight, so we will have to wait and see if we can find a shorter alternate path. We want to track which nodes have been expanded, what the current shortest distance to each node is, and the current parent of each node, done in the table below. An X marks any nodes that have been expanded and a $-1$ indicates that a parent has yet to be assigned.

| Node | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | $s$ | $d$ |
|---|---|---|---|---|---|---|---|---|---|---|
| `expanded` | | | | | | | | | **X** | |
| `distance` | **9** | $\infty$ | **11** | $\infty$ | **5** | $\infty$ | $\infty$ | **2** | 0 | **100** |
| `parent` | ***s*** | $-1$ | ***s*** | $-1$ | ***s*** | $-1$ | $-1$ | ***s*** | $-1$ | ***s*** |

The node with the current shortest distance to $s$ that hasn't been expanded yet is node 8. Expanding node 8 gives three nodes: $s$ (distance $2+2=4$), 1 (distance $2+9=11$), and 7 (distance $2+10=12$). Of these nodes, only node 7 has a shorter path through 8 (since its current shortest path is length "$\infty$"), so this is the only node that is updated.

| Node | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | $s$ | $d$ |
|---|---|---|---|---|---|---|---|---|---|---|
| `expanded` | | | | | | | | **X** | X | |
| `distance` | 9 | $\infty$ | 11 | $\infty$ | 5 | $\infty$ | **12** | 2 | 0 | 100 |
| `parent` | $s$ | $-1$ | $s$ | $-1$ | $s$ | $-1$ | **8** | $s$ | $-1$ | $s$ |

The next unexpanded node with the shortest distance is node 5. Expanding 5 gives five nodes: $s$ (distance $5 + 5 = 10$), 2 (distance $5 + 6 = 11$), 3 (distance $5 + 8 = 13$), 4 (distance $5 + 30 = 35$), and 7 (distance $5 + 6 = 11$). Nodes 2 and 4 are new, and we have found a shorter alternative path to node 7, so we need to overwrite some information in our table.

| Node | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | $s$ | $d$ |
|---|---|---|---|---|---|---|---|---|---|---|
| `expanded` | | | | | **X** | | | X | X | |
| `distance` | 9 | **11** | 11 | **35** | 5 | $\infty$ | **11** | 2 | 0 | 100 |
| `parent` | $s$ | **5** | $s$ | **5** | $s$ | $-1$ | **5** | $s$ | $-1$ | $s$ |

Expanding node 1 next only brings us back to $s$ or 8, but not through any shorter alternate paths. We then have a three-way tie for the next node to expand (2, 3, and 7 have distance of 11). Let's do as before and break ties by expanding in increasing numeric order. Expanding nodes 2 and 3 will provide no new shorter paths, leaving our search in the following state (no changes to `distance` or `parent`).

| Node | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | $s$ | $d$ |
|---|---|---|---|---|---|---|---|---|---|---|
| `expanded` | **X** | **X** | **X** | | X | | | X | X | |
| `distance` | 9 | 11 | 11 | 35 | 5 | $\infty$ | 11 | 2 | 0 | 100 |
| `parent` | $s$ | 5 | $s$ | 5 | $s$ | $-1$ | 5 | $s$ | $-1$ | $s$ |

Next, we expand node 7, giving the following neighbors: 5 (distance $11 + 6 = 17$), 6 (distance $11 + 7 = 18$), 8 (distance $11 + 10 = 21$), and $d$ (distance $11 + 15 = 26$). Node 6 is new and the path to node $d$ is shorter than the current shortest path, so we update their corresponding values.

| Node | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | $s$ | $d$ |
|---|---|---|---|---|---|---|---|---|---|---|
| `expanded` | X | X | X | | X | | **X** | X | X | |
| `distance` | 9 | 11 | 11 | 35 | 5 | **18** | 11 | 2 | 0 | **26** |
| `parent` | $s$ | 5 | $s$ | 5 | $s$ | **7** | 5 | $s$ | $-1$ | **7** |

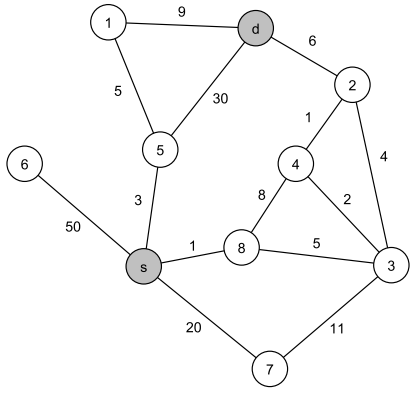
Once again, we have encountered $d$, but we cannot be sure that we found the shortest path yet, since there might be an even shorter path through node 6, for example. Speaking of which, node 6 has the shortest distance of unexpanded nodes and is next, giving nodes 7 (distance $18 + 7 = 25$) and $d$ (distance $18 + 12 = 30$). We did not end up finding a shorter path, so our search remains as the following.

| Node | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | $s$ | $d$ |
|---|---|---|---|---|---|---|---|---|---|---|
| `expanded` | X | X | X | | X | **X** | X | X | X | |
| `distance` | 9 | 11 | 11 | 35 | 5 | 18 | 11 | 2 | 0 | 26 |
| `parent` | $s$ | 5 | $s$ | 5 | $s$ | 7 | 5 | $s$ | $-1$ | 7 |

Finally, node $d$ is next to be expanded, and we can stop our search. We see that the shortest path from $s$ to $d$ has a length of 26. We can find the path by backtracking through the parents. The parent of $d$ is 7, the parent of 7 is 5, and the parent of 5 is $s$. The shortest path is therefore $s \to 5 \to 7 \to d$. We can double check that the length of the path is correct by adding the length of each edge, giving $5 + 6 + 15 = 26$.

**Exercise 16: Dijkstra's algorithm**

Consider the following network.




Perform Dijkstra's algorithm to find the shortest weighted path from $s$ to $d$. Be sure to track which nodes have been expanded, the current shortest distance to each node, and the parent of each node after each expansion, as was done in the example. As before, expand nodes in increasing numerical order based on their label in the case of any ties.

### 3.2.3 Pseudo-code algorithm

The psuedo-code for Dijkstra's algorithm is presented below. As before, let's walk through each section of the code and see how each piece works.

**Initialization (lines 1–6).** As with all programs, we need to first create some objects and set their initial values. The set `nodes` will be a list of all nodes in our network. As we expand each node, it will be removed from `nodes`. The difference between a list and a set is that a set has no specific order of its items and each item can only exist once. Why we would prefer a set over a list in this case is not too important to delve into, but the short answer is a set will be more efficient to manage.

Just like in breadth-first search, we also create a list `parent` to track the parent of each node. As a reminder, if $j$ has $i$ as its parent (meaning the shortest path includes the edge $i \to j$), then `parent[j]` would equal `i` by the end of the algorithm. Again, as in BFS, we assign an arbitrary value as the parent of every node before we begin, choosing some value that does not represent any node in the network.

For Dijsktra's algorithm, we also need to create the list `distance` which will track the **current** shortest distance found from node $s$ to each node that has been explored. If `distance[i]` equal 100, then the shortest path found so far from $s$ to $i$ has a length of 100. It is possible that this number will change as more nodes are expanded and shorter paths are found. By the end of the algorithm, we will know that the shortest path from $s$ to $d$ is stored in `distance[d]`.

Finally, before we begin the algorithm we set the distance from $s$ to $s$ equal to 0, by saying `distance[s] = 0`, and the distance from $s$ to all **other** nodes equal to infinity (or some very large number). Most programming languages, including Python, have some way to represent "infinity" for a numeric variable. We need to make sure it's a very large number because if we set all distances initially equal to 100, for example, and the shortest path has a length of 101, then the program won't give us the right answer (it can only update when it finds a **shorter** distance than the one currently assigned). To make sure we choose a number large enough to work for any network possible, we

**Algorithm 2:** Dijkstra's algorithm with backtracking

**Data:** network `G(V,E)`, weights `W`, starting node `s`, destination node `d`
**Result:** Returns the shortest weighted path from `s` to `d`.

```
1  create the set nodes
2  create the list parent
3  create the list distance
4
5  set distance to all nodes as INFINITY
6  set distance to s as 0
7
8  while nodes is not empty do
9  |  find the node in nodes with the smallest distance and set as v
10 |  remove v from nodes
11 |  if v is d then
12 |  |  break out of the loop
13 |  end
14 |
15 |  foreach neighbor w of v do
16 |  |  calculate alt as distance to v plus edge weight from v to w
17 |  |  if alt is smaller than current distance to w then
18 |  |  |  set distance to w as alt
19 |  |  |  set parent of w as v
20 |  |  end
21 |  end
22 end
23
24 if distance to d is INFINITY then
25 |  return empty path
26 else
27 |  backtrack the path from d to s following parent
28 |  return path
29 end
```

choose infinity.

**Outer loop (lines 8–13).** Our main loop again controls how long the algorithm will search and also selects the next node to expand. The next node to expand is always the one with the current shortest distance to $s$, because this node cannot possibly be found through any shorter path, as we explained before. We find which node `v` has the smallest value of `distance[v]` and select it to be expanded. We remove `v` from `nodes` so that we know not to expand it again (each node can only be expanded once).

There are two ways our search will stop. It is possible that `v` is $d$, in which case we found the shortest path from $s$ to $d$ and can stop. We perform this check and break out in lines 11–13. Notice that we do not need the variable `found` for Dijkstra's algorithm; we will see that there is another way to know that we found $d$ when attempting to recreate the path. The second way the search will stop is if we've run out of nodes in `nodes`, since there would be nothing else to expand. The outer `while` loop controls this condition by continuing to loop **while** there are still nodes in `nodes`.

**Expansion (lines 15–21).** Once we have selected a node `v` to expand, we look at each of its neighbors `w`. We first calculate the distance to `w` by following this alternative path through `v` as

$$\texttt{alt} = \texttt{distance[v]} + \texttt{W[v][w]},$$

where `W` is our weight matrix and `W[v][w]` is entry $[W]_{v,w}$. This is the formula for the length of the distance from $s$ to $v$ plus the length of the edge from $v$ to $w$.

**If** the length of `alt` is more than the current value of `distance[w]` (the current shortest distance to $w$), then we do nothing: the path through $v$ is not shorter. If instead `alt` is **smaller**, then we update `distance[w] = alt`, since `alt` is the length of the new shorter path, and mark that the parent of $w$ is now $v$ by saying `parent[w] = v`.

Once we have inspected all neighbors of $v$, we return to the start of the outer loop (line 8) and choose the next node to expand.

**Backtracking (lines 24–29).** Before we backtrack to find the path from $s$ to $d$ by following the parent nodes, we must first check `distance[d]`. If `distance[d]` is still equal to infinity, then no path exists from $s$ to $d$. Since no edge has a weight of infinity, the sum of all the edge weights must still be some finite number. The fact that `distance[d]` equal infinity means $d$ was never found by following a path starting at $s$. If `distance[d]` **does not** equal infinity, then we backtrack as normal.

## 3.3 Conclusion

Breadth-first search is probably the most important algorithm in network analysis. Here we saw its ability to find shortest paths and how to modify it to create Dijkstra's algorithm and find shortest **weighted** paths. The ability to **explore** a network through expanding nodes and looking at its neighbors is central to many other network algorithms.

For large networks, we really need to use a computer to analyze them, because it quickly becomes impractical by hand. You can find many of the pieces of this packet implemented in Python in the accompanying iPython notebook. Feel free to play around with the code in the notebook to see if you can adapt it to other exercises in this packet.

If you've completed all the exercises in the packet (including the Challenge! problems) and still have time, then you may want to look at some in-depth Python tutorials. You can find one free interactive tutorial at `https://www.codecademy.com/learn/learn-python`. The Python 2 tutorial is free for anyone, once you make an account. (Python 3 is actually the most up-to-date version of Python, but the Python 3 tutorial is only available to Pro members. Fortunately, the main differences between Python 2 and Python 3 aren't too important when learning the basics.)