Chapter 1

Basic Graph Theory: Communication and Transportation Networks

In this section, we will introduce some basics of graph theory with a view towards understanding some features of communication and transportation networks.

1.1 Notions of Graphs

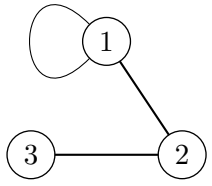
The term graph itself is defined differently by different authors, depending on what one wants to allow. First we will give a fairly typical definition. For elements u and v of a set V, denote by $\langle u, v\rangle$ the unordered pair consisting of u and v.* (Here u and v are not necessarily distinct, and the pair being unordered just means that $\langle v, u\rangle = \langle u, v\rangle$.) Denote by $\mathrm{Sym}(V \times V)$ the set of all unordered pairs $\langle u, v\rangle$ for $u, v \in V$.

Definition 1.1.1. *An* **(undirected) graph** *(or* **network***)* $G = (V, E)$ *consists of a set of* **vertices** *(or* **nodes***)* V *together with an* **edge set** $E \subset \mathrm{Sym}(V \times V)$*. The elements of* E *are called* **edges** *or* **links***. The number of elements in* V *is called the* **order** *of* G*, and we often say* G *is a graph on* V*.*

A priori, the order of a graph could be infinite, i.e., it could have infinitely many vertices. Infinite graphs can be quite useful in theory, but we will focus on networks that arise in real-life situations, which are finite, i.e., they have finitely many vertices.

▶ *Unless otherwise specified, we will always assume our graphs are finite.*

Example 1.1.2. *Let* $V = \{1, 2, 3\}$*. Then* $V \times V$ *is the the set of all order pairs of vertices and* E *should be a symmetric subset of this. Take* $E = \{\langle 1, 1\rangle, \langle 1, 2\rangle, \langle 2, 3\rangle\}$*. To draw the graph, draw and label each node, and draw a link between two vertices if there is an edge between them, like this:*



*This is not standard notation. Most authors write (u, v) or $\{u, v\}$, but I want to reserve (u, v) for the ordered pair and $\{u, v\}$ for the set of u and v.

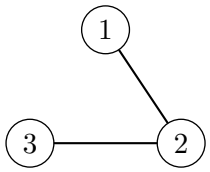


Figure 1.1: The simple graph associated to Example 1.1.2.

In the above example there is an edge from vertex 1 to itself.

Definition 1.1.3. *Let $G = (V, E)$ be a graph. An edge of the form $\langle v, v\rangle \in E$ is called a* **loop**. *If G has no loops, we say G is* **simple**.

It is clear that given any graph, we can make it into a simple graph just by deleting all loops. For instance, the above example gives rise to the simple graph:

Here the edge set is now $E = \{\langle 1, 2\rangle, \langle 2, 3\rangle\}$.

▶ *Unless otherwise specified, we assume all graphs are simple.*

Note that if we are working with simple graphs, then the unordered pair $\langle u, v\rangle$ is simply the set $\{u, v\}$, and you can define an edge as simply a subset of V of size 2. Some authors will use this definition, which implies that any graph for them is simple. (If we are not working with simple graphs, then you can have a loop $\langle v, v\rangle$ whose associated set $\{v, v\} = \{v\}$ has size 1, not 2.) So if you prefer curly braces to angle brackets, feel free to write your edges with those, e.g., $E = \{\{1, 2\}, \{2, 3\}\}$.

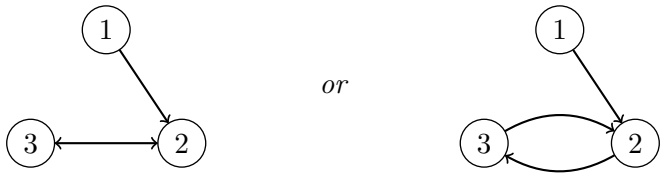
In many instances, the graphs we want to consider are not naturally "symmetric" (undirected). For example, we might want to make a graph of webpages and draw a directed link from one page to another if there is a hyperlink from one page to another. Another example is with citation graphs—graphs of all research documents in a field with a directed link from paper A to paper B if paper A cites paper B. In this case, one should think of these directed links as *ordered* pairs (u, v), rather than unordered pairs $\langle u, v\rangle$. This leads us to the following definition.

Definition 1.1.4. *A* **directed graph** *(or* **digraph***) $G = (V, E)$ consists of a set V of vertices and an edge set $E \subset V \times V$. The elements of E are called* **(directed) edges** *or* **links**. *If G contains no loops, then we say G is* **simple**.

If $e = (u, v) \in E$ is a directed edge, we say e is an edge **from** *u* **to** *v, or* **starting at** *u* **and ending at** *v. Further u is called the* **initial vertex** *of e and v is called the* **terminal vertex** *of e.*

▶ *Except where otherwise specified, the term graph used by itself means undirected graph.*

Example 1.1.5. *Consider $V = \{1, 2, 3\}$ and $E = \{(1, 2), (2, 3), (3, 2)\}$. We draw simple directed graphs as follows. If $(u, v) \in E$ but $(v, u) \notin E$, then we draw a edge from u to v with an arrow pointing towards v. If (u, v) and (v, u) are both in E, then we can either draw a single edge from u to v with an arrow on each end or two different edges, one with an arrow to u and one with an arrow to v.*

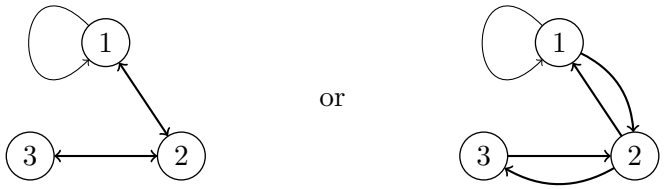


For directed graphs, edges are thought of as having direction, so the edge $(2,3)$ *is considered different than the edge* $(3,2)$*, and this digraph has 3 edges not 2, as one might think from the drawing on the left.*

Note that we can consider undirected graphs as a special case of directed graphs in the following way. Suppose $G = (V, E)$ is an undirected graph. Then one can consider a directed graph $G' = (V, E')$ on the same vertex set V where now the edge set

$$E' = \{(u, v), (v, u) : \langle u, v\rangle \in E\}$$

contains both edges (u, v) and (v, u) for any undirected edge $\langle u, v\rangle$ in G. (For non-simple graphs, the loop $\langle v, v\rangle$ just becomes (v, v).) E.g., for Example 1.1.2 the edge set is $E' = \{(1, 1), (1, 2), (2, 1), (2, 3), (3, 2)\}$. The corresponding directed graph can then be drawn as follows:

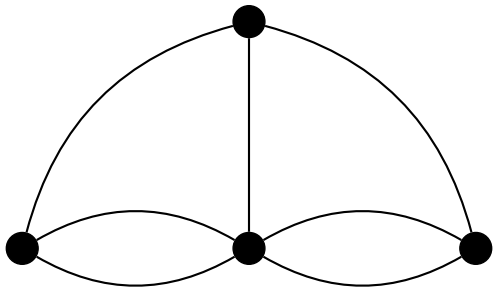


(For non-simple directed graphs, for any loop, we just draw an arrow on one end of the loop.)

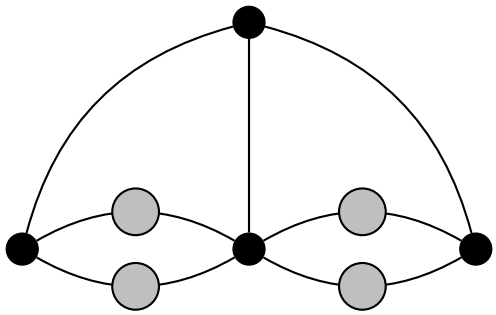
Consequently, we can think of undirected graphs simply as directed graphs whose edge sets E are *symmetric*, i.e., $(u, v) \in E$ implies $(v, u) \in E$. (The set of unordered pairs of elements of V corresponds to symmetric subsets of the set $V \times V$ of ordered pairs, which is why I used the notation $\mathrm{Sym}(V \times V)$ above.) The only real difference is that when counting edges, the directed graph will have more edges (precisely twice as many for simple graphs.) This perspective is useful as we can study both directed and undirected graphs in a unified framework. With this in mind, I will often use the ordered pair notation (u, v) for edges in an undirected graph. In this case, I will write the edge set as a symmetric subset of $V \times V$—for example, for Example 1.1.2, I may write the edge set as $E = \{(1, 1), (1, 2), (2, 1), (2, 3), (3, 2)\}$, with the understanding that (u, v) and (v, u) represent the same edge (so this graph still has 3 edges, not 5).

Realizing the edge set as a subset of the ordered pairs is also more natural from the matrix point of view, and often useful from an algorithmic point of view.

There are two other generalizations of graphs worth mentioning now. Some authors allow multiple edges between vertices in their definition of graph—I will call these **multigraphs**. For instance, consider the Seven Bridges of Königsberg in Figure 1. Euler considered this situation with the multigraph formed by making each landmass a vertex and each bridge an edge:



However, one can reduce multigraphs to graphs by adding in appropriate vertices, e.g., for Euler's example above, we can just add in new vertices along certain edges to get a graph:



Thus we will not have much reason to consider multigraphs except in certain special cases.

Another generalization of graph that we will sometimes consider is a **weighted graph**. This is just a (directed or undirected) graph $G = (V, E)$ with a weight function $w : E \to \mathbb{R}$ that assigns each edge a weight. For instance, if the graph represents cities on a map, then a natural weight function to consider would just be the distance between the cities. Another possible weight function is the cost to get from one city to the other. Note that a weighted graph generalizes the concept of multigraph: a multigraph can be considered as a weighted graph where the weight of an edge (u, v) is just the number of edges from u to v in the corresponding multigraph. We will say more about weighted graphs later.

Exercises

Exercise 1.1.1. *(a) For $V = \{1, 2\}$ and $V = \{1, 2, 3\}$, draw all possible graphs on V.*

(b) How many possible graphs are there on $V = \{1, 2, 3, 4\}$? Draw all such graphs having 4 edges.

(c) Draw all possible directed graphs on $V = \{1, 2\}$. Then draw all possible directed graphs $V = \{1, 2, 3\}$ which contain the edge $(1, 2)$.

1.2 Representations of Graphs

Now let's discuss different ways one can represent graphs in Python. You can work directly with sets in Python 2.7 (using curly brackets, as in math), so the most naive way you can represent a graph G is with an (ordered) list (denoted by square brackets) consisting of the vertex set V and the edge set E. For example, the graph in Figure 1.1 can be represented as:

Python 2.7
```
>>> V = {1, 2, 3}
>>> V
set([1, 2, 3])
>>> E = [ {1, 2}, {2, 3} ]
>>> E
[set([1, 2]), set([2, 3])]
>>> G = [V, E]
>>> G
[set([1, 2, 3]), [set([1, 2]), set([2, 3])]]
```

(Lines beginning with `>>>` denote input to the Python interpreter, and other lines denote the Python output. I wrote V, E, and G on separate lines after the definitions just so you can see how Python will output this data to you when you want it later. E.g., the Python output `set([1, 2])` just means the set consisting of 1 and 2, or what we would typically write in mathematical notation as $\{1, 2\}$.)

Also note that while spacing (indentation) is important in Python for nested statements over multiple lines (e.g., "for loops" or "if... then" statements), it is not important within individual lines. In particular, I could write something like `V={1,2,3}` or `V = { 1 , 2 , 3 }` for the first line with the same result.)

Here V is represented as a set of vertex names, and the edge set E is an ordered list of edges e, where each edge is represented as a set of size 2. For technical reasons, using the built-in set type in Python, one cannot write E as a set, i.e., `E = { {1, 2}, {2, 3} }` will result in an error, because Python does not by default handle sets of sets, or sets of lists. For similar reasons, G must also be defined as a list `G=[V,E]`, rather than a set `G={V,E}`. Even if it were possible to define `G={V,E}` in Python, it is better to define it as a list, because to actually do things with G, one will need to recover the vertex and edge sets V and E. If you define G as a list, you can just get the vertex set back with `G[0]` and the edge set by `G[1]`. (Python naturally numbers list positions starting at 0, not at 1.) But if one could and did define G as a set, there is no order, so it would not be as easy to recover the vertex set or the edge set.

Many programming languages do not have a built in data structure for sets (i.e., unordered lists), but one can similarly represent a graph in terms of lists (or arrays). In this case one can write the vertex sets as a list, and the edge set as a list of ordered pairs (a list of lists of size 2). For example, this same graph can be represented as

Python 2.7
```
>>> V = [1, 2, 3]
>>> V
[1, 2, 3]
>>> E = [ [1, 2], [2, 1], [2, 3], [3, 2] ]
>>> E
[[1, 2], [2, 1], [2, 3], [3, 2]]
>>> G = [V, E]
>>> G
[[1, 2, 3], [[1, 2], [2, 1], [2, 3], [3, 2]]]
```

This method of representing edges as (ordered) lists of size 2 is of course also advantageous as one can represent directed graphs in the same way.

However, these naive ways of representing a graph in a computer is not so useful in practice. For example, consider the following problem.

Given a node u of a (directed or undirected) graph $G = (V, E)$, we say a node $v \in V$ is **adjacent** to u, or a **neighbor** of u, if $(u, v) \in E$. (Note for undirected graphs, being neighbors is a symmetric relation, but not so for directed graphs, i.e., v may be adjacent to u without u being adjacent to v.) Write an algorithm which, given a node u returns all the neighbors v of u. This is one of the most basic procedures one will want to do when working with graphs.

Let's see how to do this using where we represent V as a list and E as a list of lists of size 2, as in the latter snippet of code. (Thus we are working with directed edges.) In fact, whether V is a set or a list is not important to our algorithm—however it does make a difference in syntax that each edge is a list of size 2, not a set.

Python 2.7

```
>>> V = [ 1, 2, 3 ]
>>> E = [ [1, 2], [2, 1], [2, 3], [3, 2] ]
>>> G = [V, E]
>>>
>>> def VE_neighbors(G, u):
...     neigh = set()               # start with an empty set
...     E = G[1]                    # let E be the edge set
...     for e in E:
...         if e[0] == u:           # for each edge of the form (u,v)
...             neigh.add(e[1])     # add v to the set neigh
...     return neigh
...
>>> VE_neighbors(G, 1)
set([2])
>>> VE_neighbors(G, 2)
set([1, 3])
```

Here I have written a function `VE_neighbors` that takes as input two things: the graph `G=[V,E]` represented in the above "vertex set-edge set representation" (I use "VE" at the beginning of this function name to indicate this), and the vertex `u` one wants to find the neighbors of. The algorithm is to just go through each element of the edge set E, and check if the edge starts at u (i.e., is of the form (u, v)), and if so, add the corresponding element to the set of neighbors. (If `e=[u,v]` is a directed edge, then `e[0]` returns the initial vertex `u` and `e[1]` returns the terminal vertex `v`.) The remarks after the hash signs `#` are comments to help you understand the code and ignored by the Python interpreter. (You should always comment your code.)

Note: one uses the double equals `==` in the `if` statement to test if two things are the same—do not use `e[0]=u`, which will set `e[0]` equal to `u`.

Then at the end of this snippet of code, I test the function `VE_neighbors` on the graph from Figure 1.1 for the vertices 1 and 2. As expected, the Python output says the set of neighbors for the vertex 1 is just $\{2\}$, and the set of neighbors for the vertex 2 is $\{1, 3\}$. In this implementation, I encoded the neighbors of u as a set, rather than a list, but one could do this also (Exercise 1.2.2).

When we write programs, we are often concerned with efficiency, particular if we are working with a large amount of data. For small graphs, this not a big deal, but if you want to work with graphs with hundreds or thousands or millions of nodes, it's crucial. The algorithm `VE_neighbors` requires going through each element of the edge set E, so we say the running time is $O(|E|)$.

(This notation, called Big Oh notation, will be explained in detail later—roughly it means that the algorithm requires on the order of $|E|$ steps to finish.) Here $|E|$ denotes the size of the set E, i.e, the number of (in this case directed) edges.

Note that for a not-necessarily-simple directed graph $G = (V, E)$ on n nodes, the maximum number of possible edges $|E|$ is n^2—this is simply the number of ordered pairs $V \times V$ (see Exercise 1.2.1). Thus we can give an upper bound for the run time of this algorithm as $O(n^2)$. This is horribly inefficient for such a basic operation, and we will see we can do much better using a different representation for a graph.

Adjacency matrices

Whenever you have a finite collection of objects, and some relations between them, you can keep track of them in a table. For example, in linear algebra if you're working with two variables x and y, you can keep track of linear combinations

$$3x + 2y$$
$$x - y$$

by just writing the coefficients in a box

$$\begin{pmatrix} 3 & 2 \\ 1 & -1 \end{pmatrix}.$$

Of course one needs to keep track of the order of x and y, so you know x corresponds to the first column, and y the second.

We can do something similar (though not exactly the same) for graphs.

Definition 1.2.1. *Let $G = (V, E)$ be a directed or undirected graph, not necessarily simple.* Write V as an* ordered *set $\{v_1, v_2, \dots, v_n\}$. The* **adjacency** *(or* **incidence***)* **matrix** *for G (with respect to the ordering $v_1, v_2, \dots, v_n$) is the $n \times n$-matrix*

$$A = (a_{ij}), \quad a_{ij} = \begin{cases} 1 & (v_i, v_j) \in E \\ 0 & (v_i, v_j) \notin E. \end{cases}$$

Example 1.2.2. *Let $V = \{1, 2, 3\}$, as an ordered set. Then the adjacency matrix for the undirected graph in Example 1.1.2 is*

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix} \end{array}.$$

For clarity, I labeled which rows and which columns correspond to which vertex in red, but I won't typically do this. In other words, there is an edge from 1 to 1 (a loop), and edge from 1 to 2, an edge from 2 to 1 an edge from 2 to 3, and an edge from 3 to 2.

*From now on, we will often implicitly realize the edge set E for undirected, graphs as a symmetric set of ordered pairs (u, v), rather than a set of unordered pairs.

Similarly, the adjacency matrix for the directed graph on V in Example 1.1.5 is

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix} \end{array}.$$

In other words, there is a (directed) link from 1 to 2, and a link both directions between 2 and 3.

Note that these adjacency matrices depend on the ordering we chose for V. If for some perverse reason, we wanted to use a different ordering, say $V = \{3, 2, 1\}$ then, e.g., the adjacency matrix for the directed graph in Example 1.1.5 is

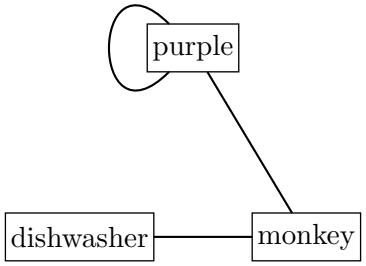
$$\begin{array}{c} \\ 3 \\ 2 \\ 1 \end{array} \begin{array}{c} \begin{array}{ccc} 3 & 2 & 1 \end{array} \\ \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \end{array}.$$

Let's make a couple of elementary observations now. First, having a loop $(v_i, v_i) \in E$ means that the i-th diagonal element of the matrix $a_{ii} = 1$, so having no loops (i.e., being simple) is equivalent to the statement that all the diagonal entries of the adjacency matrix are zero. (As we see from this argument, this fact does is independent of which ordering we choose for V.)

Moreover, note that the (a priori directed) graph G is undirected[†] if an only if the matrix A is symmetric. Again this will not depend on the ordering we choose for V. For any ordering, G being undirected means that $(v_i, v_j) \in E$ is equivalent to $(v_j, v_i) \in E$ for all $1 \leq i, j \leq n$, which means the value of a_{ij} must equal the value of a_{ji} for all i, j, which is equivalent to A being symmetric as asserted.

Next, given some vertex v_i, it is easy to read off its neighbors—just go to the i-th row and look at which spots have a 1. If there is a 1 in the column corresponding to v_j, this means there is a (directed) edge from v_i to v_j. This process requires going through each element of a single row in A, which has n elements, so the running time for such an algorithm is $O(n)$. This is in general much better than the $O(n^2)$ bound we got for using the "vertex set-edge" set representation above.

Note that to represent an arbitrary graph in a computer, we need a little more than the just the adjacency matrix—we also need the ordered list of vertices. For example, the graph



with respect to the vertex ordering {purple, monkey, dishwasher} has the same adjacency matrix we saw in the first part of Example 1.2.2. Of course this graph and the graph from Example 1.1.2 are essentially the same—only the names of the vertices have changed, but they are technically different graphs. We will discuss this more when we get to the notion of *graph isomorphisms* below.

[†]Technically, I mean G can be viewed as an undirected graph, i.e., that E is symmetric.

For now, I just want to make the point that to use adjacency matrices to encode the complete information about any graph $G = (V, E)$, we need to store the ordered pair (V, A), where V is an ordered set of vertices and A is the associated adjacency matrix.

For example, we can encode the purplemonkeydishwasher graph in Python as:

Python 2.7
```
>>> V = [ "purple", "monkey", "dishwasher" ]
>>> A = [ [ 1, 1, 0 ], [ 1, 0, 1 ], [ 0, 1, 0 ] ]
>>> G = [V, A]
>>> G
[['purple', 'monkey', 'dishwasher'], [[1, 1, 0], [1, 0, 1], [0, 1, 0]]]
```

Here we represent the adjacency matrix

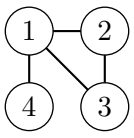
$$A = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$

as a list of lists. The lists `[1, 1, 0]`, `[1, 0, 1]` and `[0, 1, 0]` represent the three rows of A, and then the matrix A is encoded in Python as a list of the three row vectors. Then we can access, e.g., the top row of A by the code `A[0]` (this will give you `[1, 1, 0]`) and the individual entries of the top row by `A[0][0]`, `A[0][1]` and `A[0][2]`.

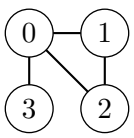
However, we don't really care about purplemonkeydishwashers in this class. We are primarily interested in just studying the structure of graphs in this class. The names of the vertices will only be important when we are looking at specific networks/applications (e.g., the graph in Figure 2). Consequently, to simplify things, we will often assume—at least when we are working by hand—that we are working with an ordered vertex set of the form $V = \{1, 2, \ldots, n\}$. When we are working with graphs on the computer, we will typically assume the vertex set $V = \{0, 1, \ldots, n-1\}$. With this assumption in mind, we can simplify our lives a bit and represent a graph G by just its adjacency matrix A. For instance, by default we will interpret the adjacency matrix

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

as representing the graph



if we are working by hand, but as



if we are working on the computer.

The reason for the difference working by hand versus on the computer is that humans naturally count from 1, where as computers naturally count from 0.* Namely, if $V = \{1, 2, 3, 4\}$ and you want to check if there is an edge from vertex 1 to vertex 3, you would look at the entry a_{13} of $A = (a_{ij})$ working by hand, but you would need to look at `A[0][2]` in Python. The need to shift indices in Python is just an unnecessary complication that we can avoid by assuming our vertex set is $V = \{0, 1, 2, 3\}$, for then the entry `A[0][2]` tells us about the existence or nonexistence of an edge from vertex 0 to vertex 2.

Now let's explain the algorithm to find the neighbors of a given vertex u in a graph G. Assume $V = \{0, 1, \ldots, n-1\}$, and say we want to find vertex i. (We'll often use i and j to denote vertices when are vertex set is $\{0, 1, \ldots, n-1\}$ or $\{1, 2, \ldots, n\}$.) Let A be the associated adjacency matrix. Then `A[i]` gives the i-th row of A, and we just need to go through each element of the row, and if there is a 1 in position j of this row, we add vertex j to our (initially empty) list of neighbors for i. The code, with an example on the above graph, is here.

Python 2.7

```
>>> def neighbors(A, i):
...     n = len(A)                  # let n be the size (number of rows) of A
...     neigh = []                  # start with an empty set neigh
...     for j in range(n):
...         if A[i][j] == 1:        # for each index 0 <= j < n
...             neigh.append(j)     # append j to the list neigh if the i-th
...     return neigh                #     row has a 1 in the j-th position
...
>>> A = [ [ 0, 1, 1, 1 ], [1, 0, 1, 0], [1, 1, 0, 0], [1, 0, 0, 0] ]
>>> neighbors(A,0)
[1, 2, 3]
>>> neighbors(A,1)
[0, 2]
>>> neighbors(A,2)
[0, 1]
>>> neighbors(A,3)
[0]
```

Adjacency Lists

There is a third common way to represent graphs, and this is with *adjacency lists*. Fix a (directed or undirected, simple or not) graph $G = (V, E)$—we do not need to assume V is ordered or consists of numbers. An adjacency list for G is merely a list of all the vertices $v \in V$ together with its set of neighbors $n(v) \subset V$. This can be implemented in Python with a structure known as a *dictionary*.

You can think of a dictionary in Python as basically a table consisting of keywords (called *keys*) and their associated data/definitions (called *values*). A dictionary is defined using curly braces like sets. Each dictionary entry is given in the form `key:value`, and the entries are separated by commas. For example, if I wanted to define a dictionary that gave me course titles associated to the course numbers I am teaching this semester, I can enter this as follows

*This is also one reason why androids don't make good life partners.

Python 2.7
```
>>> courses = { 4383 : "Cryptography", 4673 : "Graph Theory", \
... 5383 : "Cryptography", 5673 : "Graph Theory" }
>>> courses[4383]
'Cryptography'
>>> courses[5383]
'Cryptography'
>>> courses[4673]
'Graph Theory'
>>> courses[5673]
'Graph Theory'
```

Note the keys and the values can be numbers or strings (you can define strings in Python using single or double quotes). In fact, the values can be other things like lists or sets also. The single backslash on the first line just means the input will be continued on the subsequent line. Then we see we can access the entries of the dictionary by using the key in square brackets, in the same way we would access the elements of a list using their index.

Using this dictionary structure, we can encode our purplemonkeydishwasher graph as an adjacency list as follows

Python 2.7
```
>>> G = { "purple" : { "purple", "monkey" }, \
... "monkey" : {"purple", "dishwasher"}, \
... "dishwasher" : { "monkey" } }
>>> G["monkey"]
set(['purple', 'dishwasher'])
```

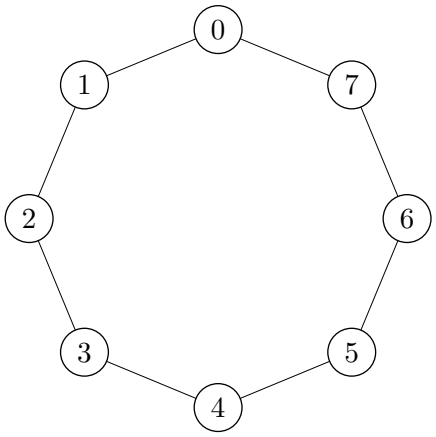
Here the keys are strings, the names of the vertices, and the values are the sets of neighbors, encoded as sets of strings. For instance, the first line says that the node `purple` is assigned the neighbors `purple` and `monkey`. The order in which the vertices are given in the adjacency list is irrelevant. One could alternatively encode the neighbors of each vertex as lists instead of sets.

Note that, conversely, given an adjacency list, one can reconstruct the graph. One simply draws all the vertices (keys) in the adjacency list and draws the (a priori directed) edges from each key to each of its neighbors. (Do this now for the purplemonkeydishwasher adjacency list.) Hence the adjacency list structure gives a valid way to represent a graph (i.e., all the information about the graph is present in the adjacency list).

By design, finding the neighbors of a given vertex using an adjacency list takes only one step! Using our Big Oh notation, which I'll formally get to soon, we would say this can be done in $O(1)$, or constant, time. In other words, it doesn't matter how many vertices there are in the graph, you just look at the entry for the vertex you want, which is the set of neighbors.*

Remarks on implementation: Dictionaries work a bit differently than lists in Python, so you can't append things to a dictionary. If you want to make an adjacency list in Python, and not enter everything by hand the easiest way is to make a list `al` of ordered pairs of the form $(v, \{\text{neighbors of } v\})$. For example, to make an adjacency list for the following graph

*Technically, there are a couple of issues here: (1) We've ignored the time it takes to *locate* the entry for a given vertex, however this can be implemented to be done very very quickly. (2) If we want to actually, say, print out the list of neighbors, the amount of time this takes depends upon the amount of neighbors, which is *a priori* only bounded by the order n of the graph. However, for large graphs that arise in practice, the number of neighbors of any vertex is generally much much smaller than n.



we can use the following code

Python 2.7

```
>>> al = []
>>> for x in range(8):
...     al.append((x, {(x-1)%8, (x+1)%8}))     # for x = 0, 1, 2, ..., 7
...                                # associate the set {x-1 mod 8, x+1 mod 8}
>>> al
[(0, set([1, 7])), (1, set([0, 2])), (2, set([1, 3])), (3, set([2, 4])),
(4, set([3, 5])), (5, set([4, 6])), (6, set([5, 7])), (7, set([0, 6]))]
>>> G = dict(al)
>>> G
{0: set([1, 7]), 1: set([0, 2]), 2: set([1, 3]), 3: set([2, 4]), 4: set([3, 5]),
5: set([4, 6]), 6: set([5, 7]), 7: set([0, 6])}
>>> G[7]
set([0, 6])
```

Here the command `x%8` returns x mod 8, which is the value $r \in \{0, 1, 2, \ldots, 7\}$ such that $8 = qx + r$, i.e., x mod 8 is (at least for $x \geq 0$) the remainder upon dividing x by 8. So, for $0 \leq x \leq 6$, $x + 1$ mod 8 is just x, for $x = 7$ it is `8%8` $= 0$. Similarly, for $1 \leq x \leq 7$, $x - 1$ mod 8 is just x, whereas for $x = 0$ it is `-1%8` $= 7$. In other words, by using the mod function we can use addition/subtraction to right/left shift the numbers $\{0, 1, 2, \ldots, 7\}$ with the convention that we wrap around at the edges.

Adjacency matrices versus adjacency lists

When working with graphs on computers, one typically uses either the adjacency matrix representation or the adjacency list representation. The vertex set-edge set representation that we used for the standard mathematical definition is too cumbersome and slow to work with in actual algorithms. We've seen this for just the problem of finding the neighbors of a given vertex, where the adjacency list representation runs in constant time ($O(1)$), the adjacency matrix representation runs in linear time ($O(n)$), and the vertex set-edge set representation runs in quadratic time ($O(n^2)$).

Adjacency matrices are suitable for small graphs, and have some advantages over adjacency lists. As an example, suppose you have a directed graph G on $V = \{1, 2, \ldots, n\}$ and want to find all the vertices with an edge *to* a fixed vertex j (the inverse to the problem of finding neighbors). With an adjacency matrix, one just looks at the j-th column of an adjacency matrix, where as

things are a bit more complicated with the adjacency list. In addition, it is easier to go between theory and practice using matrices (much of the theory is easier to present in terms of matrices, and some of the coding is also).

For large graphs, the adjacency list representation is typically far superior in practice, particularly for *sparse graphs*, i.e., graphs with relatively few edges (closer to n than n^2). Social networks tend to be rather sparse. (Consider the graph of webpages where the directed edges are hyperlinks. According to Kevin Kelly's *What technology wants* (2010), there are about a trillion webpages and each webpage has, on average, about 60 out of a possible 1 trillion links. If this graph weren't sparse, any useful sort of web searching might be virtually impossible.)

For these reasons, we will primarily use adjacency matrices, at least at the beginning of this course. Towards the end of the course, when we want to work with large graphs, we won't program our own algorithms for everything, and will use the graph theory library in SAGE, which (I believe) uses primarily adjacency lists.

Exercises

Remarks on programming exercises: In all exercises that I ask you to code up a function, you must also test your function on some examples. I will let you choose your own examples to test on (you might choose some from the notes, or some more complicated ones). The more complicated the code is, the more testing you should do. In this section, testing your code on a couple of examples should suffice to convince you (and me) whether it works correctly *all of the time* or not.

In coding, choosing good examples to test your code on is of paramount importance—you should try to test different situations (e.g., directed and undirected, simple or not, include vertices with no neighbors) as it often happens that code will fail for certain very specific cases (for mathematical code, it is often extreme cases, such as code failing when some parameter is minimal or maximal). You also need to choose test cases where you can easily verify that the answer you get is correct (or at least seems reasonable if you don't know the correct answer yourself). (Of course, the first step is to get the code to run without any errors.)

When there is a bug, it is often helpful to choose good examples and examine how the output differs from what it should be to figure out what the bug is. Many times one can figure out what the bug is just by looking a few sample inputs and outputs, and not even looking at the original code! (Though this approach comes easier with experience, but it can be very helpful to try to reason out how the computer is getting from your input to its output.)

If you are having trouble getting your code to run correctly, the first thing you should try to do is test different parts of your code separately. You can also try printing out the values of variables at various steps to help see what is going on.

Exercise 1.2.1. *Let V be a set with n elements.*

(a) How many simple undirected graphs are there on V? What is the maximum number of possible edges? What if we don't require simple?

(b) How many simple directed graphs are there on V? What is the maximum number of possible edges? What if we don't require simple?

Exercise 1.2.2. *Write an analogue of the function* `VE_neighbors`*, called* `VE_neighbors_list`*, that uses a list instead of a set for* `neigh`*, and consequently returns a list instead of a set. (Read the note above about testing your code.)*

Exercise 1.2.3. *Let G be a graph, directed or undirected, simple or not, on $V = \{0, 1, \ldots, n-1\}$. Let A the adjacency matrix for G (with respect to our usual ordering on V). Write a function called* `AM_to_AL`*, whose input is the adjacency matrix A and output is the adjacency list for G.*

Exercise 1.2.4. *Let G be a graph, directed or undirected, simple or not, on $V = \{0, 1, \ldots, n-1\}$, given as an adjacency list. Write a function called* `AL_to_AM`*, whose input is G and output is the adjacency matrix A for G (with respect to our usual ordering on V).*

1.3 Basic Algorithm Analysis

In this section we will explain the notion of *algorithms* and how to analyze their efficiency. To do this, we will first introduce Landau's Big Oh notation and discuss asymptotic growth.

1.3.1 Asymptotic growth and Big Oh notation

Let $\mathbb{N} = \{1, 2, 3, \ldots\}$ and $\mathbb{R}_{>0}$ denote the set of positive real numbers. Recall a function f on $\mathbb{N}$ is just the same thing as a sequence of numbers $(a_n)_n$ by taking $a_n = f(n)$.

Definition 1.3.1 (Big Oh, Version 1)**.** *Consider functions $f, g : \mathbb{N} \to \mathbb{R}_{>0}$, i.e., $(f(n))_n$ and $(g(n))_n$ are sequences of positive real numbers. We say $f(n)$ is* **(big) O** *of $g(n)$ if there exists a constant C such that $f(n) \leq Cg(n)$ for all $n \in \mathbb{N}$. In this case, we write $f(n) \in O(g(n))$ or $f(n) = O(g(n))$.*

Roughly what $f(n) \in O(g(n))$ means is that, for sufficiently large values of n, $f(x)$ grows no faster than $g(n)$. We can think of $O(g(n))$ as the class of functions which don't grow faster than $g(n)$, hence the notation $f(n) \in O(g(n))$. Typically for us $f(n)$ and $g(n)$ will be increasing functions that go to infinity, and you can think of $f(n) \in O(g(n))$ as meaning $f(n)$ is asymptotically $\leq$ (a constant times) $g(n)$. Getting a basic understanding of asymptotic growth rates is essential to understand which how efficient various algorithms are.

We remark that the notation $f(n) = O(g(n))$ is more common, though it is a bit misleading—$f(n) = O(g(n))$ does not mean $g(n) = O(f(n))$. It's usage is probably due to the fact that it is more intuitive for asymptotic expressions. For example, if $f(n)$ is the number of primes less than n, the Prime Number Theorem says

$$f(n) \sim \int_2^n \frac{1}{\log t}\, dt$$

(this is about $n/\log n$), so we can think of

$$f(n) = \int_2^n \frac{1}{\log t}\, dt + \epsilon(n)$$

where ϵ is some error term less than $n/\log n$ for n large. The Riemann Hypothesis gives a bound on the error term: $\epsilon(n) = O(\sqrt{n}\log n)$. Using an equals sign in our Big Oh notation allows us to write our asymptotic for $f(n)$ as

$$f(n) = \int_2^n \frac{1}{\log t}\, dt + O(\sqrt{n}\log n).$$

(Here there are a couple of technicalities with the definition we gave for our Big Oh notation: $\epsilon(n)$ is not always a positive number, and $\sqrt{n}\log n = 0$ for $n = 1$. We'll explain how to define Big Oh notation in a bit more generality below.)

In any case, I will primarily stick to the $f(n) \in O(g(n))$ notation in this course.

Example 1.3.2. *Let $f : \mathbb{N} \to \mathbb{R}_{>0}$ be a bounded function. Then $f(n) = O(1)$.*

Proof. By definition, we know there exists a constant M such that $0 < f(n) < M$ for all n. Consequently, taking $C = M$, we see $f(n) \leq C \cdot 1$ for all $n \in \mathbb{N}$. □

Example 1.3.3. *Consider a polynomial $f(n) = a_d n^d + a_{d-1} n^{d-1} + \cdots + a_1 n + a_0$ which is positive on each $n \in \mathbb{N}$ (e.g., this is true if each $a_i > 0$). Then $f(n) = O(n^d)$.*

In particular, we have things like $3n^2 + 5n - 2 \in O(n^2)$, so $f(n) \in O(g(n))$ does not necessarily mean that $f(n) \leq g(n)$ for n large—i.e., the constant C in the definition is important. Also, $f(n) = 5n^3$ is $O(n^3)$, $O(n^4)$, $O(n^5)$, and so on, but not $O(1)$, $O(n)$ or $O(n^2)$ (see Exercise 1.3.1).

Proof. Note that for $n \in \mathbb{N}$, we have

$$f(n) \leq |a_d| n^2 + |a_{d-1}| n^d + \cdots + |a_1| n^d + |a_0| n^d \leq C n^d$$

where $C = |a_d| + |a_{d-1}| + \cdots + |a_0|$. □

Proposition 1.3.4 (Transitivity)**.** *Suppose $f(n) \in O(g(n))$ and $g(n) \in O(h(n))$. Then $f(n) \in O(h(n))$.*

Proof. By assumption, we know there are constants C_1 and C_2 such that $f(n) \leq C_1 g(n)$ and $g(n) \leq C_2 h(n)$ for all $n \in \mathbb{N}$. Hence $f(n) \leq C h(n)$ for all n, where $C = C_1 C_2$. □

Again thinking of $O(g(n))$ as the class of functions which grow no faster than $g(n)$, this means if $g(n) \in O(h(n))$, then anything in $O(g(n))$ lies in $O(h(n))$, i.e., $O(g(n)) \subset O(h(n))$. Consequently, our example about polynomials shows we have the following nested sequence of asymptotic classes:

$$O(1) \subset O(n) \subset O(n^2) \subset O(n^3) \subset \cdots$$

If $f(n) \in O(n^d)$ for some d, we say that $f(n)$ has (at most) **polynomial growth**, because it grows no faster than some polynomial. In fact it's not hard to see that all of these $O(n^d)$ classes are different, i.e., the inclusions above are strict inclusions. (See Exercise 1.3.1 below.) For example, $O(n^3)$ contains (positive) polynomials $f(n)$ of degree ≤ 3, whereas $O(n^2)$ will only contain polynomials of degree ≤ 2. (These classes contain other functions besides polynomials as well, e.g., $6n^{2.34567} + n \log n + (-1)^n \in O(n^3)$.)

Now let's give alternative criteria for a function $f(n)$ to be $O(g(n))$, which will give us the right definition even when $f(n)$ and $g(n)$ are not necessarily positive (and sometimes undefined at some values).

Proposition 1.3.5. *Let $f, g : \mathbb{N} \to \mathbb{R}_{>0}$. Then the following are equivalent*

1. *$f(n) \in O(g(n))$;*
2. *There exist constants C, N such that $f(n) \leq C g(n)$ for all $n > N$.*
3. *The sequence of numbers $\left(\frac{f(n)}{g(n)}\right)_n$ is bounded.*

In particular, if $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ exists and is finite, then $f(n) \in O(g(n))$.

Proof. Clearly 1 $\implies$ 2, since they are equivalent if we take $N = 0$. On the other hand suppose 2 holds for some constants C and N. Let $C_0 = \max\{\frac{f(n)}{g(n)} : 1 \le n \le N\}$. Then by definition we have $f(n) \le C_0 g(n)$ for $1 \le n \le N$ and $f(n) \le Cg(n)$ for $n > N$. Thus, for any n, we have $f(n) \le C'g(n)$, where $C' = \max\{C, C_0\}$. Hence 2 $\implies$ 1, and we have the equivalence of the first two conditions.

Now let us show 1 $\iff$ 3. First suppose 1 holds, i.e., there exists C such that $f(n) \le Cg(n)$ for all n. Then, using positivity, we have $0 \le \frac{f(n)}{g(n)} \le C$ for all n, which yields 3. Conversely, 3 implies that there is a constant C such that $f(n) \le Cg(n)$ for all n.

The last statement follows because, if the limit exists, then 3 must hold. □

Definition 1.3.6 (Big Oh, Version 2)**.** *Let $f(n)$ and $g(n)$ be partially-defined real-valued functions on $\mathbb{N}$, but assume they are both well defined for n sufficiently large. Then we say $f(n)$ is* **(big) O** *of $g(n)$, and write $f(n) \in O(g(n))$ or $f(n) = O(g(n))$, if there exist constants C and N such that $|f(n)| \le C|g(n)|$ for all $n > N$.*

The point of this more general, though slightly more technical, definition is that $f(n) \in O(g(n))$ is an asymptotic condition, which means it should only be a statement about sufficiently large n, and for small values of n the condition $f(n) \le Cg(n)$ is not important, and we don't even care if the functions don't make sense for small n.

More precisely, the "partially-defined" condition means that we allow $f(n)$ and $g(n)$ to be undefined on some *finite* subset of $\mathbb{N}$. This is convenient because it allows us to handle functions like $\log(n-1)$ or $\log(\log(n))$, both of which are undefined when $n = 1$, but defined for all $n > 2$.

In addition, if $g(n)$ is not required to be positive, then $f(n) \le Cg(n)$ for all $n > N$ does not imply we can choose a possibly larger value for C' to get $f(n) \le C'g(n)$ for all n like we did in Proposition 1.3.5. The issue is if $g(n) = 0$ for some n. For example, if $f(n) = 3$ and $g(n) = \log(n)$, then we have $f(n) \le g(n)$ for any $n > 20$. In fact, we can get $f(n) \le 5g(n)$ for any $n > 1$, but we will never have $f(1) \le Cg(1)$ for any C since $g(1) = \log 1 = 0$.

The reason to add the absolute values in the definition was simply to give a more general statement of big O notation which is particularly useful in bounding errors in asymptotics, which might be positive or negative, as in the discussion about the Prime Number Theorem above. (Alternatively, one could just put the absolute values on f and require $g(n) \ge 0$ for n sufficiently large). However, for most of our purposes, we will just consider cases where both $f(n)$ and $g(n)$ are positive, at least for sufficiently large n and we can typically forget about these absolute values.

One final remark about this definition versus the previous version: even if your functions are positive everywhere, it is often a bit easier to check that an inequality holds for sufficiently large n than having to find an explicit C that works for all n. For example, suppose you want to check $f(n) = 4$ is $O(\log(n+1))$ by hand from the definition. It is (slightly) easier to use the second definition and simply observe that $\log 3 > 1$ so $f(n) \le 4\log(n+1)$ for $n > 1$, rather than trying to estimate $\log 2$ to find a C such that $4 \le C\log(2) \le C\log(n+1)$ for all n.

The following will be a convenient tool to show $f(n) \in O(g(n))$ in many cases.

Proposition 1.3.7. *Let $f(n)$ and $g(n)$ be partially-defined real-valued functions on $\mathbb{N}$. Assume $g(n) \neq 0$ for n sufficiently large. Then the following are equivalent*

1. *$f(n) \in O(g(n))$;*
2. *For some number N, the sequence of numbers $\left(\frac{f(n)}{g(n)}\right)_{n>N}$ is bounded.*

In particular, if $\lim_{n\to\infty} \frac{|f(n)|}{|g(n)|}$ *exists and is finite, then* $f(n) \in O(g(n))$.

Note we need the condition that $g(n) \neq 0$ for sufficiently large n just to ensure the ratios $\frac{f(n)}{g(n)}$ are well defined for all n large enough. E.g., if we take something like $f(n) = n \sin \frac{\pi}{2} n$ and $g(n) = n^2 \sin \frac{\pi}{2} n$, then $f(n)$ and $g(n)$ are just 0 when n is even n and $\pm n$ and $\pm n^2$ when n is odd. It is true that $f(n) \in O(g(n))$ but we can't say that condition 2 holds because the ratios are never well-defined for n even.

The proof is essentially the same as the $1 \iff 3$ part of the proof for Proposition 1.3.5, except that one includes absolute values and restricts the inequalities to $n > N$ for some N. (See Exercise 1.3.4.)

Example 1.3.8. $O(1) \subsetneq O(\log\log n) \subsetneq O(\log n) \subsetneq O(\sqrt{n}) \subsetneq O(\sqrt{n}\log n) \subsetneq O(n)$.

For increasing functions f and g, the statement $O(f(n)) \subsetneq O(g(n))$ (i.e., every function $h(n)$ in $O(f(n))$ is in $O(g(n))$ but not conversely) means that, asymptotically, f grows strictly slower than g does.

Proof. The structure of the proofs for each part is the same. Namely, we can show $O(f(n)) \subsetneq O(g(n))$ as follows. By transitivity (Proposition 1.3.4), if we show $f(n) \in O(g(n))$ then we will have $O(f(n)) \subset O(g(n))$. Then we show $g(n) \notin O(f(n))$ to get $O(f(n)) \subsetneq O(g(n))$.

The first part, that $O(1) \subsetneq O(\log\log n)$ is obvious because $f(n) = 1$ is bounded, whereas $g(n) = \log\log n$ goes to infinity.

For the second part, that $O(\log\log n) \subsetneq O(\log n)$, we use Proposition 1.3.7. Namely, by l'Hospital's rule, we have

$$\lim_{n\to\infty} \frac{\log\log n}{\log n} = \lim_{x\to\infty} \frac{\frac{d}{dx}\log\log x}{\frac{d}{dx}\log x} = \lim_{x\to\infty} \frac{1/(x\log x)}{1/x} = \lim_{x\to\infty} \frac{1}{\log x} = 0,$$

i.e., $\log\log n \in O(\log n)$. Similarly, an application of l'Hospital's rule on the reciprocal shows

$$\lim_{n\to\infty} \frac{\log n}{\log\log n} = \lim_{x\to\infty} \frac{1/x}{1/(x\log x)} = \lim_{x\to\infty} \log x = \infty,$$

so $\log n \notin O(\log\log n)$. Hence $O(\log\log n) \subsetneq O(\log n)$, as claimed.

The remaining parts are similar to the second part, and left as Exercise 1.3.5. □

Note in the proof of second part, instead of applying l'Hospital's rule a second time, we could just observe that if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$, then $\lim_{n\to\infty} \frac{|g(n)|}{|f(n)|} = \infty$. This observation gives the following corollary of Proposition 1.3.7.

Corollary 1.3.9. *Let* $f(n)$ *and* $g(n)$ *be partially-defined real-valued functions on* $\mathbb{N}$. *Assume* $g(n) \neq 0$ *for* n *sufficiently large. If* $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$, *then* $O(f(n)) \subsetneq O(g(n))$, *i.e.,* $f(n) \in O(g(n))$ *but* $g(n) \notin O(f(n))$.

Example 1.3.10. *Let* $a > 1$ *and* $d > 0$. *Then* $O(n^d) \subsetneq O(a^n)$.

Again, the proof is an exercise. A function $f(n) \in O(a^n)$ for some $a > 1$ is said to have (at most) **exponential growth**. (I include the "at most" because we don't typically say polynomials have exponential growth—they have polynomial growth!) This example should just be a translation

of something you know from calculus—exponential functions grow faster than any polynomial. In algorithm analysis, typically exponential growth is very bad, polynomial growth is good, and logarithmic growth ($O(\log n)$) is outstanding.

For our algorithm analysis, there is one more elementary thing to be aware of—the "arithmetic" of Big Oh.

Proposition 1.3.11. *Suppose $c > 0$ is a constant, $f_1(n) \in O(g_1(n))$ and $f_2(n) \in O(g_2(n))$. Assume $g_1(n)$ and $g_2(n)$ are positive for sufficiently large n. Then*
(i) $(f_1 + f_2)(n) \in O((g_1 + g_2)(n))$, and
(ii) $(f_1 f_2)(n) \in O((g_1 g_2)(n))$.

Proof. (i) There exist constants such that $|f_1(n)| \leq C_1|g_1(n)| = C_1 g_1(n)$ for $n > N_1$ and $|f_2(n)| \leq C_2|g_2(n)| = C_2 g_2(n)$ for $n > N_2$. Consequently

$$|f_1(n) + f_2(n)| \leq |f_1(n)| + |f_2(n)| \leq C_1 g_1(n) + C_2 g_2(n) \leq \max\{C_1, C_2\}(g_1(n) + g_2(n)),$$

for $n > \max\{N_1, N_2\}$, which is the assertion of (i).
(ii) is similar. □

The assumption about g_1 and g_2 being positive is just to rule out something like $f_1(n) = f_2(n) = n$, $g_1(n) = -g_2(n) = n^2$ where $g_1 + g_2$ cancels out the growth of g_1 and g_2. One could state (i) without the positivity assumption as $(f_1 + f_2)(n) \in O((|g_1| + |g_2|)(n))$. (Positivity is not needed for (ii).)

1.3.2 Algorithms

The point of the above diversion on big Oh asymptotic classes is that now we have some basic tools to explain some simple algorithm analysis, which is extremely important in practice when one wants to work with graphs of even moderate size.

With all this talk of analyzing algorithms, you might already be a little uneasy. Maybe you're thinking to yourself, I don't even know what an algorithm is. That's okay, because it's not entirely well-defined. Don't worry though, this won't cause any problems though—just because Plato wasn't sure what a table was, I'm sure he could build one or use one perfectly well.

For us, an **algorithm** is a (finite) sequence of instructions designed to accomplish a specific task. The instructions themselves might be a little vague, or even a lot vague. For example, consider the following two algorithms.

Algorithm 1.3.12. *Find the "most popular" member of a given social network $G = (V, E)$.*

1. *Go through each node $v \in V$, and count the number of neighbors* $\deg(v)$ *of v (called the* **degree** *of v).*
2. *Find the largest* $\deg(v)$*, and output the corresponding v.*

This algorithm is fairly specific, but there are still some things open to interpretation. First of all, there is the notion of "the most popular" member of a social network. How this should be interepreted might depend upon type the network and whether it is directed or undirected. However, let's assume it is undirected and that by most popular I really mean the node of highest

degree. One issue is that there might be a tie—e.g., for the graph in Figure 2, two nodes (Brady and Clay) are tied for the highest degree (5). In this case, should one output all nodes of highest degree, or just one? Probably it's reasonable to output all nodes of highest degree.

These clarifications make the algorithm rather well-defined, though it's still not as specific as it could be. For example: in what order do you go through the nodes in Step 1? how do you keep d_v and v associated, or don't you? (e.g., make a table?) what is the algorithm to find the largest d_v? (e.g., do you sort first or not?) However, it's good enough for anyone to be able to carry out by hand, or for someone with a moderate amount of programming experience to be able to code up easily.

Now here is a somewhat famous, but much worse, example of an algorithm.

Algorithm 1.3.13. *Find an optimal mate.*

1. *Estimate the number of people N you can date in your lifetime.*
2. *Date $N/e \approx 0.36N$ people, give them scores, break up with them (or get dumped—user choice), and let M be the maximum of these scores.*
3. *For each subsequent person you date, score them. If their score is below M, break up with them. If there score is above M, marry them.*

This comes from a probability exercise about figuring out how to maximize your chances of getting the best possible spouse (here you're not allowed to date multiple people at a time, or marry someone you've already broken up with). The reason I think this is a bad algorithm is perhaps different from the reason you might think this is a bad algorithm (or perhaps not). Sure, maybe you can't accurately estimate N or give your more-than-friends-but-less-than-spouses accurate scores. And maybe anyone who scores above your cutoff M won't want to marry you, i.e., you don't score above the cutoff value in their algorithm. But in some sense, these are problems of implementation of the algorithm, and we're not meant to worry about these issues in this theoretical exercise. (Or maybe you take ontological issue with the existence of such a thing as an "optimal mate." But we're working in the confines of an admittedly absurd exercise.)

The main problem with the algorithm is that often it doesn't give the correct solution to the problem (though sometimes it will). Already 36% of the time, you've broken up with your optimal mate in Step 2, which means in Step 3 you will break up with everyone until the end of (your) time, so the algorithm *doesn't terminate* (until you do). Even when you marry, it doesn't always give the optimal mate. However, within the confines of this theoretical exercise, there is no algorithm with will always produce an optimal mate, i.e., there is no good algorithm for this problem (which I'm sure you already knew). Really this algorithm is not a solution to the problem "find an optimal mate"—it is a solution to the problem "out of a specific class of bad algorithms to find an optimal mate, determine which bad algorithm is the least bad".

The point is that algorithms can be good or bad (they solve the problem always, sometimes, or never). They might terminate or not (e.g., they could get stuck in an infinite loop), or be very quick or very slow. The instructions might be clear or vague. At some point, if the instructions become too vague, we should probably not call it an algorithm anymore (e.g., "solve this problem" is not an algorithm for solving any problem), but there is no clear cut line as to what is "too vague." This is what I meant when I said I don't know exactly what an algorithm is, in the same way Plato wasn't sure exactly what a table was. The notion of an algorithm is like the notion of a

mathematical proof—it's essentially done by consensus. If people are convinced by an argument, it's considered a proof.* If people can figure out how to carry out the instructions, it's considered an algorithm.

Let's return to Algorithm 1.3.12, with programming in mind. If we're trying to write code for this algorithm, there are various ways it could be implemented. (Note: computer code is not the same as an algorithm—it's a specific implementation of an algorithm. For example, changing variable names in code changes the code, but not the algorithm. Or changing a for loop to a while loop changes the code, but not the algorithm. However, we won't try to be precise about when two blocks of code are consider as implementations of the same algorithm, or two different ones. Again this is done by common sense/consensus.)

An experienced programmer would have no trouble coding up this algorithm, however someone with little programming experience (which might be you) might vacillate a little with it. So where possible, particularly as we're getting started with programming, we'll try to make our algorithms a little more explicit. For example, we can write a more detailed algorithm as

Algorithm 1.3.14. *(Algorithm 1.3.12 refined.)*

1. *Set* `maxd` $= 0$.
2. *For each* $v \in V$*, calculate the degree* $\deg(v)$*. If* $\deg(v) >$ `maxd`*, set* `maxd` $= \deg(v)$.
3. *Make a new empty list* `mostpop`.
4. *For each* $v \in V$*, if* $\deg(v) =$ `maxd`*, append* v *to* `mostpop`.
5. *Output* `mostpop`.

This is a lot closer to computer code, and should be easy for you to code up once you're somewhat familiar with Python. (The one point we haven't explained here is how to calculate the degree—this is simple, but it depends upon the implementation of the graph. Let's take this for granted now and come back to it in a moment.) Hopefully, this is fairly straightforward to understand: the first two steps are the algorithm to find the maximum degree `maxd`, and the next two steps find all the vertices having this maximum degree.

Another way to express an algorithm in a ready-to-code way to do this is with *pseudocode*. This is something that looks a lot like computer code, but isn't quite. Typically one makes it a little easier to read than actual code, and sometimes avoids writing down all the details of actual code that will run. Since we're programming in Python, we'll use Pythonesque pseudocode. From the pseudocode, it should generally be a simple matter to write actual Python code (though you may need to look up some commands or syntax).

Here is sample pseudocode for Algorithm 1.3.12.

Pseudocode

```
set maxd = 0
for v in V:
    d = degree(G,v)
    if d > maxd then
```

*Even for mathematicians, who usually want everything to be precise, imprecisely-defined notions like what constitutes a proof are often more useful than exactly defined ones.

```
        set maxd = d
set mostpop = []
for v in V:
    if degree(G,v) == d then
        append v to mostpop
output mostpop
```

This pseudocode is pretty close to actual Python code, and how close you want to make your pseudocode to actual code is up to you. I wrote it essentially as I would Python code, but tried to make it flow more like English when you read it aloud.* Specifically, the differences are: I added the word `set` at the beginning of definitions, I used the word `then` instead of a colon in the `if` statements, the `append` line is different, and the word `output` instead of return. (I also didn't include a line to define the function.) Of course, I also used a function not yet written called `degree`, which computes the degree.

Note this pseudocode is more precise than Algorithm 1.3.14. For example, in our pseudocode, we are recomputing `degree(G,v)` in Step 4 of Algorithm 1.3.14. That is, we don't bother keeping track of the degree $\deg(v)$ for each v when we first compute it—we compute all the degrees once to determine the maximum degree, and then we compute them all again to see which vertices have maximum degree. Alternatively, we could have stored all the degrees in a list and just accessed the previously computed degrees in Step 4. Hence we have (at least) 2 different implementations of Algorithm 1.3.14. (Just to show you there are many ways to do things: a variant of Algorithm 1.3.14 would be to store all the degrees and vertices in a table (a 2-dimensional array) during Step 2, sort the table by degrees, and then output the vertices at the top of the table.)

Now let me tell you how to find the degree of a vertex. After this, you should be able to code up Algorithm 1.3.14 (see Exercises 1.3.7 and 1.3.8).

First, let's see how to do it if the graph is given as an adjacency matrix `A` (with respect to $V = \{0, 1, 2, \ldots, n-1\}$).

Python 2.7

```
>>> def deg(A,i):
...     d = 0                       # initialize the degree d to be 0
...     for j in range(len(A)):     # for j = 0, 1, 2, ..., n-1
...         d = d+A[i][j]           # add A[i,j] to d
...     return d
...
>>> A = [ [ 0, 1, 1, 1 ], [1, 0, 1, 0], [1, 1, 0, 0], [1, 0, 0, 0] ]
>>> deg(A,0)
3
>>> deg(A,1)
2
>>> sum(A[0])
3
>>> sum(A[1])
2
```

Here I define a function `deg(A,i)`, which takes in an adjacency matrix which and returns the degree of the i-th vertex, which is just the sum of the entries in row i. This is how my code computes the

*You can think of pseudocode as programming poetry. Bonus points for pseudocode in iambic pentameter, limerick, or haiku.

degree. (Recall `len(A)` returns the number of rows in `A`, i.e. the size of `A`.) However, Python already has a built-in function `sum`, which returns the sum of the entries in a list, so you can alternatively get the degree of vertex i just by calling `sum(A[i])`.

Now suppose the graph G is given as an adjacency list. Again we could write a function that gets the degree of vertex v, but it can be obtained simply by counting the length of the set of neighbors of v. (We could also use this algorithm in the adjacency matrix implementation, but counting the number of 1's in the i-th row is more straightforward.) If G is given as a dictionary, this can be done as follows.

Python 2.7
```
>>> G = { "purple" : { "purple", "monkey" }, \
... "monkey" : {"purple", "dishwasher"}, \
... "dishwasher" : { "monkey" } }
>>> len(G["monkey"])
2
```

Here `G[v]` returns the set of neighbors of `v`, and we pass this set of neighbors to the function `len`, which returns the length (size) of a set.

1.3.3 Algorithm Running Times

There are two basic constraints in computing: *data storage* and *computing time*. In the olden days, when games came on multiple diskettes and computer screens had 1 color—green—data storage was a serious concern, and programmers had to work hard so as not using any more memory/disk space than necessary. Now, memory/storage capacity is relatively cheap, and data storage is not a serious issue for most computing tasks. It is mostly only a concern for very specialized problems—e.g., keeping tabs on everything on the internet—though I think most program developers don't take data storage issues seriously enough. (Many programs are bloated, and unnecessarily slow down your computer—on the other hand, it's easier to write programs that aren't efficient.)

Nowadays, the main concern about efficiency is typically is the amount of time a program takes to run. This will be our main focus in algorithm analysis as well, though occasionally if the amount of space used becomes egregious we'll discuss it.

How should we gauge the efficiency of a program or algorithm? One way is simply to physically time how long it takes to run. There are a couple of issues with this. One, the amount of time depends on the implementation of the algorithm (both how you write your code, and how your programming language translates your code into machine operations), the task it is performing (what the input is) and the computer it is running on. Since, in the heyday of Moore's law, computer speeds were doubling every 18 months, just measuring physical running times is of limited use (though still useful). Further, trying all possible inputs is typically impractical.

Instead, we'd like a simple theoretical way to analyze algorithms that will allow us to estimate how fast or slow a program will be in practice. This approach will also have the considerable benefit that we don't actually have to write a working program to analyze the algorithm. The procedure is very simple: we just count the number of steps require to complete the algorithm, i.e., the number of lines of code that the program will run.

Let's start off with a simple, straightforward example: the program to find the degree of a vertex using adjacency matrices from Section 1.3.2. Let's recall the code.

Python 2.7
```
>>> def deg(A,i):
...     d = 0                       # initialize the degree d to be 0
...     for j in range(len(A)):     # for j = 0, 1, 2, ..., n-1
...         d = d+A[i][j]           # add A[i,j] to d
...     return d
```

Here there are two inputs, A and i, so we will let $f(A,i)$ be the number of steps required by this code given the input (A,i). Let n be the size of A. The first line (after verb+def+), $d = 0$, is run one time. The second line, you can take as also being run once. The third line, however, is run n times. Finally the last line is run once. Hence $f(A,i) = n+3$. In fact, since f depends only on the order n of the graph, we can think of this as a function of n, i.e., $f(n) = n+3$.

This is not exactly the number of steps the computer will do—there's a lot of stuff going on behind the scenes at the processor level for each line of code. However, it's a reasonable estimate thinking that each line of code takes 1 unit of time to run, and it would be a real headache to analyze what the processor is actually doing. There is one point to be careful about however—in the second line we call the functions `range` and `len`. The function call to `len` takes one step (regardless of how big A is, Python stores the length of A for easy access and you just need to retrieve this value from memory*). However the function `range` returns the list $[0, 1, 2, ..., n-1]$, which takes $n+2$ steps to create (n steps to put all the items in the list, 1 to make an empty list, and 1 to return the list). So perhaps it is better to say $f(n) = (n+3)+1+(n+2) = 2n+6$. (In fact, we could get pickier, but I'm sure none of you want that.)

We can also see here how the implementation makes a difference about the number of steps the algorithm will take—if one uses Python's `xrange` instead of `range`, or a `while` loop instead of the `for` loop, Python doesn't actually need to create a list of size n to do the loop, and we would have something like $f(n) = n+4$ or $f(n) = n+5$.

The point is that, however we do this analysis, and even if we get very picky, the number of steps the computer is doing behind the scenes, this $f(n)$ is a *linear function* in n, i.e., $f(n) \in O(n)$, i.e., $f(n)$ has **linear growth**. In other words, as the order n of the graph grows, the amount of time this function will take to run grows linearly in n. Thus we say the **running time** of this algorithm is $O(n)$. (While technically, we also have $f(n) \in O(n^2), O(n^3), O(2^n)$, etc., we don't say that this algorithm has running time $O(n^2)$ or $O(n^{n^n})$ because that would be morally reprehensible, even if legally permissible.)

Let me quickly give one more example before discussing algorithm running times in more generality. Recall the following algorithm for finding neighbors of a given vertex from an adjacency matrix.

Python 2.7
```
>>> def neighbors(A, i):
...     n = len(A)                  # let n be the size (number of rows) of A
...     neigh = []                  # start with an empty set neigh
...     for j in range(n):
...         if A[i][j] == 1:        # for each index 0 <= j < n
...             neigh.append(j)     # append j to the list neigh if the i-th
...     return neigh                #     row has a 1 in the j-th position
```

*I didn't check the actual implementation of the length function, but Python surely must do this.

To determine the running time of this algorithm, again let $f(A,i)$ denote the number of steps the algorithm takes to run with given input (A,i). The first three lines (after `def`) and the final line contribute 1 step each (not being picky with the `range` function in the `for` loop). The next line `if A[i][j] == 1:` runs n times. The next-to-last line runs somewhere between 0 and n times, depending on how many neighbors vertex i has. Hence $n+4 \leq f(A,i) \leq 2n+4$. Since both our upper and lower bounds for $f(A,i)$ are $O(n)$, we say this algorithm has running time $O(n)$.

In general, an algorithm is a sequence of instructions that takes in some input data, such as an integer, a list, a matrix, a graph, or possibly multiple inputs (17 lists, 3 matrices and a graph). Suppose we have an algorithm, Algorithm **A**, that takes in input $\mathcal{I}$. Let $f(\mathcal{I})$ denote the number of steps Algorithm **A** takes to run given input $\mathcal{I}$. Let $\alpha(\mathcal{I})$ denote the "size" of $\mathcal{I}$. (How we measure the size of the input depends upon the problem and our point of view, but for us it will typically be the order n of some graph.) As we saw in the last example, the number of steps $f(\mathcal{I})$ required may depend upon more than just the size $\alpha(\mathcal{I})$ of $\mathcal{I}$. Consequently, we define three notions of running times.

Definition 1.3.15. *Let $(\mathcal{I}_n)$ denote a sequence of inputs $\mathcal{I}_n$ such that $\alpha(\mathcal{I}_n) = n$.*

- *If $f(\mathcal{I}_n) \in O(g(n))$ for some sequence $(\mathcal{I}_n)$, we say Algorithm* **A** *has* **best case running time** $O(g(n))$.
- *If, on average, $f(\mathcal{I}_n) \in O(g(n))$ for sequences $(\mathcal{I}_n)$, we say Algorithm* **A** *has* **average case running time** $O(g(n))$.
- *If $f(\mathcal{I}_n) \in O(g(n))$ for all sequences $(\mathcal{I}_n)$, we say Algorithm* **A** *has* **worst case running time** $O(g(n))$.

The best case running time tells you what is the fastest your algorithm can run. The average case tells you how long it usually takes, and the worst case gives you an upper bound for all possible inputs.

Here is an alternative, slightly more formal, description. Let $\mathcal{S}$ be the space of all possible inputs $\mathcal{I}$. Define a function $\alpha : \mathcal{S} \to \mathbb{N}$ and assume that for each $n \in \mathbb{N}$, the preimage $\mathcal{S}_n := \alpha^{-1}(n) \in \mathcal{S}$ is finite. Here $\alpha(\mathcal{I})$ is what we called the size $n(\mathcal{I})$ of $\mathcal{I}$ above. Let

$$\begin{aligned} f_{\min}(n) &:= \min_{\mathcal{I} \in \mathcal{S}_n} f(\mathcal{I}) \\ f_{\text{avg}}(n) &:= \frac{1}{|\mathcal{S}_n|} \sum_{\mathcal{I} \in \mathcal{S}_n} f(\mathcal{I}) \\ f_{\max}(n) &:= \max_{\mathcal{I} \in \mathcal{S}_n} f(\mathcal{I}). \end{aligned}$$

Then we say, with respect to our choice of size function α, Algorithm **A** has best case running time $O(f_{\min}(n))$, average case running time $O(f_{\text{avg}}(n))$ and worst case running time $O(f_{\max}(n))$.)

Example 1.3.16. *Consider a graph (possibly directed, non-simple) with adjacency matrix A. Fix a vertex i, and say we want to find the vertices which are either neighbors of i, or neighbors of neighbors of i. (We will define the notion of distance on graphs later, and this is essentially the set of vertices of distance ≤ 2 from i.) Here is our algorithm.*

Algorithm 2-neighbors:

1. *Make an empty set* `2-neigh`.
2. *Find the neighbors of i and add them to* `2-neigh`.
3. *For each neighbor j of i, find the neighbors of j and add them to* `2-neigh`.
4. *Output* `2-neigh`.

Instruction 1 and Instruction 4 both take 1 step each. As we saw above, finding the neighbors of i takes $O(n)$ time (best, average or worst case), and adding these elements to `2-neigh` *should take at most n steps.* Hence, the second instruction always takes $O(n)$ steps. In the third step, we need to run the neighbors algorithm again for each neighbor we had. Let's say there were d neighbors (i.e., d is the degree of i), then this is $O(n) + O(n) + \cdots O(n)$ (d times), or $O(dn)$ steps. Adding the neighbors of neighbors in Instruction 3, takes no more than dn steps, so Instruction 3 runs in $O(dn)$ steps.*

Putting everything together, we see our algorithm runs in $2 + O(n) + O(dn) = O((d+1)n)$ steps (which is the same as $O(dn)$ if $d \neq 0$). Now $0 \leq d \leq n$. If $d = 0$ (so Instruction 3 never runs at all), we are led to the minimum number of steps possible, i.e., a best case running time of $O(n)$. Similarly, the maximum number of steps is clearly when $d = n$, i.e., the worst case running time is $O(n^2 + n) = O(n^2)$. One needs to do a bit more work to rigorously check what is average number of steps. I won't go through this, but it is what you might guess—on average d will be $n/2$, so the average case running time is also $O(n^2/2) = O(n^2)$.

In some sense, knowing the average case running time (how long does the algorithm normally take?) is what you most want to know, but can be more difficult to compute than best case or worst case. Knowing the best case running time is rarely of practical use. Therefore, we will usually just concern ourselves with the worst case running time, which provides an upper bound for the question how long does the algorithm normally take, and in many instances turns out to be the same as the average case running time. Consequently, when we say *the running time* of an algorithm, without further qualification, we mean the worst case running time.

Alternatively, rather than trying to break things up into best case/average case/worse case, we could've just left things at: the running time `2-neighbor` is $O((d+1)n)$. We will sometimes do this.

If an algorithm runs in $O(1)$ time, we say it has **constant running time** (it does not seriously depend upon the size of the input.) If it runs in $O(\log n)$ time, we say it has **logarithmic running time**. If it runs in $O(n^d)$ time for some $d \in \mathbb{N}$, we say it has **polynomial running time** (the special cases $d = 1$ and $d = 2$ are called **linear** and **quadratic** running times). If it runs in $O(a^n)$ time for some $a > 1$, we say it has **exponential running time**. What we can hope for in an algorithm depends upon what the problem is, and how often we plan to call this algorithm. Generally speaking, exponential running time is very bad, arbitrary polynomial running time is okay, linear or maybe quadratic running time is good, and logarithm running time is great. Constant running time is typically impossible.

One other remark about terminology: based on what we've said in this section, you might think of the number of vertices n as the "size" of the graph—this is the default parameter. However,

*Now there is a technicality about how long it takes to add an element to a set—it depends upon the implementation (the issue is that sets should have no repeated elements, so first you have to see if your element is already in the set S or not, which naively takes $O(|S|)$). However, it can be (and I believe is in Python) implemented so that adding an a new element essentially only takes $O(1)$ time, so for simplicity let's assume this is the case.

don't call it that—call it the order of the graph. For a graph $G = (V, E)$, the **size** of G is defined by many authors to be $|E|$, the number of edges, which could be anywhere between 0 and n^2.

Let me close with a brief remark on data storage, which does become important when you've got ridiculously huge graphs like the internet.

If you want to use an adjacency matrix, you need to store an $n \times n$ matrix, which means you'll require $O(n^2)$ space—you need to store each coordinate of the matrix. The exact amount of space needed depends on the actual implementation, and how much space is needed to store the names of the vertices. However, suppose you want a graph with 1 million nodes. Each matrix entry is 0 or 1, so the most efficiently we can store the matrix in a usable form is store each matrix entry as a single bit (in the usual implementation, each matrix entry will be 64 bits, but let's say we do things more efficiently). Then storing this matrix will take 10^{12} bits $\approx$ 100 Gigabytes (GB). If you wanted 10 million nodes, this would require 100 times more space, or about 10 Terabytes (TB). And while we're talking about really large graphs here, this isn't even close to size of a web graph—remember there are an estimated 1 *trillion* webpages out there (Google seems to index tens of billions), and many social network websites have over 100 million users.

Now suppose you want to use an adjacency list. Then you need a dictionary with n entries, and each entry requires a certain space depending on the number of neighbors. The total number of neighbors list in the adjacency list is the same as the number of edges of the graph (for directed graphs, or twice that for undirected graphs). Hence the storage space required is $O(n + |E|)$. For most kinds of graphs, $|E| > n$, so this can be though of as $O(|E|)$. Hence the size $|E|$ of the graph, as defined above, really measures how much space is required to store the graph. How much space would we need to stored a graph with 1 million nodes using adjacency lists? Let's suppose that, on average, each vertex is connected to 200 other nodes (this is quite reasonable in practice—this number is very close to the *average degree* for both Twitter and Facebook graphs). Then the size, $|E|$, is 200 million. If we identify each vertex by a 64-bit number (32-bits is still more than enough), then this would require about 1.6 Gigabytes (GB). This is still quite sizable, but only 1.6% of the space required for the adjacency list representation. (And we've been fairly conservative in our estimates.) If we have 10 million nodes, where the average vertex degree is still 100, then the size only multiplies by 10 to require about 16 GB, about 0.16% of the space require for the adjacency matrix.

Exercises

Exercise 1.3.1. *Let $0 < r < s$ be real numbers. Prove that $O(n^r) \subsetneq O(n^s)$, i.e., that $f(n) \in O(n^r)$ implies $f(n) \in O(n^s)$, but there exist $f(n) \in O(n^s)$ which do are not $O(n^r)$.*

Exercise 1.3.2. *Give an example of positive functions $f(n)$ and $g(n)$ on $\mathbb{N}$ such that $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ does not exist, but $f(n) \in O(g(n))$.*

Exercise 1.3.3. *Let $f(n)$ and $g(n)$ be positive functions on $\mathbb{N}$. Is it true that either $f(n) \in O(g(n))$ or $g(n) \in O(f(n))$?*

Exercise 1.3.4. *Prove Proposition 1.3.7.*

Exercise 1.3.5. *Complete Example 1.3.8 by showing $\subsetneq O(\log n) \subsetneq O(\sqrt{n}) \subsetneq O(\sqrt{n} \log n) \subsetneq O(n)$.*

Exercise 1.3.6. *Prove the assertion in Example 1.3.10.*

Exercise 1.3.7. *Write Python code for a function* `maxdegvert(A)` *which, given an adjacency matrix A, returns (as a Python set) the set of vertices of maximum degree.*

Exercise 1.3.8. *Write Python code for a function* `AL_maxdegvert(G)` *which, given a graph G as an adjacency list, returns (as a Python set) the set of vertices of maximum degree.*

Exercise 1.3.9. *Determine the (worst case) running times for your functions* `maxdegvert(A)` *and* `AL_maxdegvert(G)` *from the previous 2 exercises.*

Exercise 1.3.10. *Consider the following simple algorithm to find the position of a number i in an ordered list of size n.*

1. *Initialize a position counter variable* `pos = 0`
2. *For each object x in the list:*
3. *if $x = i$, return* `pos`.
4. *otherwise, increase* `pos` *by 1 and continue.*

Assume that the space $\mathcal{S}_n$ of allowable inputs of size n is the set of pairs (π, i) where π is a permutation (an ordering, represented as an ordered list) of $\{0, 1, 2, \ldots, n-1\}$ and $0 \leq i \leq n-1$. Determine the best case, average case, and worst case running times for this algorithm.

1.4 Graph Isomorphisms

▶ In this section, graphs may be simple or not, undirected or directed.

If we are just interested in understanding the structure of a graph, the names of the vertices are unimportant. In other words, we may often want to just consider *unlabelled graphs*, i.e., graphs where the vertices are not labelled. We can do this formally with the notion of an *isomorphism*.

For instance, the two graphs

are technically distinct graphs, because the vertices have different names, but we want to regard them as essentially the same. We will say they are *isomorphic.* Here is the formal definition.

Definition 1.4.1. *Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be graphs. If there is a bijection $\phi : V_1 \to V_2$ such that $(u, v) \in E_1$ if and only if $\phi((u, v)) := (\phi(u), \phi(v)) \in E_2$, then we say G_1 and G_2 are* **isomorphic***, and we write $G_1 \simeq G_2$. In this case, we say the map ϕ is an* **isomorphism** *of G_1 with G_2 (if $G_1 = G_2$, we say ϕ is an* **automorphism** *of G_1).*

Recall a bijection ϕ is a map which is one-to-one and onto, i.e., ϕ maps distinct elements of V_1 to distinct elements of V_2, and each element of V_2 is in the image of ϕ. There exist bijections from V_1 to V_2 if and only if V_1 and V_2 have the same cardinality.

This definition, in less formal terms, says the following: an isomorphic ϕ is a bijection between the vertex sets V_1 and V_2, such that, regarded as a map of pairs of vertices, it maps edges of G_1 to edges of G_2, and non-edges of G_1 to non-edges of G_2. (I.e., ϕ induces a bijection of the edge sets E_1 and E_2.) Even more colloquiually: two graphs will be isomorphic, if you can turn one graph into the other merely by relabelling the vertices.

Example 1.4.2. *The two graphs pictured above are isomorphic. Let G_1 be the graph on the left, and G_2 the graph on the right. Then we can take for our bijection $\phi : V_1 \to V_2$ the function $\phi(1) = A$, $\phi(2) = B$ and $\phi(2) = C$. Viewed as a map of pairs of vertices, we see $\phi((1,2)) = (A,B) \in E_2$, $\phi((2,3)) = (B,C) \in E_2$ and $\phi((1,3)) = (A,C) \notin E_2$. Hence ϕ is indeed an isomorphism—ϕ takes edges $e \in E_1$ to edges of E_2, and non-edges to non-edges.*

Note, there is another isomorphism we could have taken (in general, there may be many). We could take $\phi'(1) = C$, $\phi'(2) = B$ and $\phi'(3) = A$. One sees again that this is an isomorphism. The fact that there are two distinct isomorphisms is due to the fact that the map of G_1 given by interchanging 1 and 3, but fixing 2, is an automorphism of G_1, i.e., if we switch the labels 1 and 3 on G_1, the graph does not change.

Here are some basic properties of the notion of isomorphic.

Proposition 1.4.3. *Let G_1, G_2 and G_3 be graphs. Then*
(i) $G_1 \simeq G_1$
(ii) $G_1 \simeq G_2 \iff G_2 \simeq G_1$
(iii) If $G_1 \simeq G_2$ and $G_2 \simeq G_3$, then $G_1 \simeq G_3$.

Proof. The proofs are simple—it just involves checking certain maps are isomorphisms, which we leave as an exercise. For (i), check the identity map is an isomorphism. For (ii), if ϕ is an isomorphism from G_1 to G_2, check ϕ^{-1} is an isomorphism from G_2 to G_1. For (iii), if ϕ_1 is an isomorphism from G_1 to G_2 and ϕ_2 is an isomorphism from G_2 to G_3, check $\phi_2 \circ \phi_1$ is an isomorphism from G_1 to G_3. □

This mean being isomorphic defines an equivalence relation among graphs.

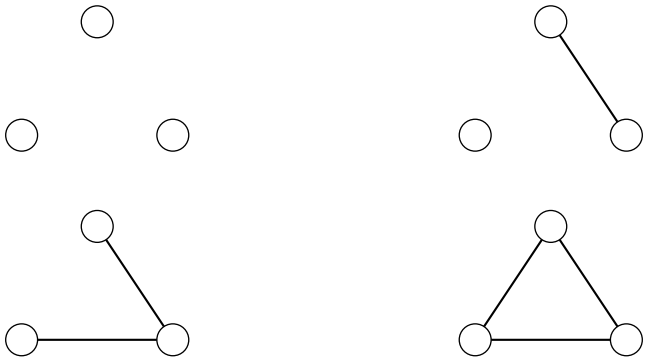
Definition 1.4.4. *An* **unlabelled graph** *is an equivalence (isomorphism) class of graphs.*

This may seem a bit strange definition if you're not familiar with this sort of idea, but the idea is quite simple. Graphs have this extra structure—the names of the vertices—that we often don't care about. So when we don't care about this, we can think of identifying all graphs isomorphic to a given graph G_0 (i.e., the same as G_0 except for this extra structure) as being the same "unlabelled" graph G. The technical way to do this is let G be the set of all graphs which are isomorphic to G_0. Then we think of any specific graph $G_i \in G$ as being a specific manifestation of the idea of G. Because being isomorphic is an equivalence relation, no two isomorphism classes intersect, and each graph corresponds to a unique unlabelled graph.

Occasionally, when we want to emphasize that we are working with honest graphs, not isomorphism classes, we may use the term **labelled graph.**

Since the notions of directed/undirected and simple/non-simple are preserved by equivalence classes (verify this!), it makes sense to say unlabelled graphs are directed/undirected or simple/non-simple if the underlying labelled graphs are.

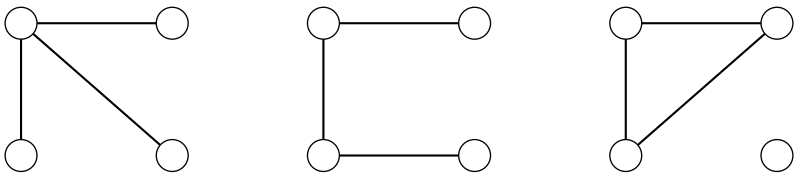
Example 1.4.5. *There are 4 simple, undirected graphs up to isomorphism (i.e., 4 unlabelled simple undirected graphs) on 3 vertices.*



In the above example, the unlabelled graphs are determined simply by the number of edges. This is no longer true when we move to 4 vertices.

Example 1.4.6. *There are two unlabelled undirected graphs on 4 vertices with two edges—either the both edges have a vertex in common or not.*

There are 3 with 3 edges (you can generate them by adding each possible edge to the previous graphs and throw out duplicates):



A basic question in graph theory is, given two graphs G_1 and G_2, determine if they are isomorphic. This is called the **graph isomorphism problem**. This is not easy in general (it might be NP-complete, if you know what that means), however in some cases it is easy to to check that two graphs are not isomorphic. For instance, if two graphs have different number of vertices, or different numbers of edges, it is easy to see there can be no isomorphism between them.

In general, data that can be associated to a graph which does not depend on its isomorphism class will be called an **invariant** of the graph. Then if two graphs have different invariants, we know they are not isomorphic. Some examples of invariants are: the order, the size, the maximum degree of a vertex, the minimum degree of a vertex, the number of isolated (degree 0) nodes, or more generally the number of vertices of degree d. We'll see many more examples of invariants later. An example of something that isn't an invariant could be something like: "the degree of vertex 1"—this evidently depends upon the labeling of the vertices.

Supposing we have two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ with the same number of vertices n, then the number of bijections from $V_1 \to V_2$ is $n!$ (there are n choices for where to map the first element of V_1, $(n-1)$ for the second, $(n-2)$ for the third, and so on.) Consequently, using

the naive algorithm to see if $G_1 \simeq G_2$ (check all possible bijections to see if they are isomorphisms) takes $O(n^2 n!)$ time (the extra n^2 is to check if each bijection is an isomorphism or not). By Stirling's approximation, $n! \sim \sqrt{2\pi n}(\frac{n}{e})^n$, so this algorithm has *worse than* exponential growth.

The best known algorithm has worst case running time $O(2^{\sqrt{n}\log n})$, which is *subexponential*—slower than exponential growth but faster than any polynomial growth, i.e., still quite bad. It is not known if there is a polynomial time algorithm which will determine if any two graphs are isomorphic or not (however there are algorithms that work quickly for most pairs of graphs, but have exponential worst case running time). This is a major unsolved problem in computational complexity theory, however we will not focus on this in our class.

Exercise 1.4.1. *Prove Proposition 1.4.3.*

Exercise 1.4.2. *Draw all unlabelled simple undirected graphs on 4 vertices. How many are there?*

Exercise 1.4.3. *Draw all unlabelled simple directed graphs on 3 vertices. How many are there?*

Exercise 1.4.4. *Let $n > 4$. How many unlabelled simple undirected graphs are there with n vertices and 1 edge? What about 2 edges? (You don't need to give formal proofs for your answers, but briefly explain your reasons.)*

Exercise 1.4.5. *How many unlabelled simple undirected graphs are there with 5 vertices and 3 edges? Draw them.*

1.5 Paths, Connectedness and Distance

▶ In this section, graphs may be directed and/or non-simple.

Now that we have various preliminaries out of the way, we can get to discussing some basic issues in networks. We'll start with communication and transportation networks in mind. For such networks, the fundamental issue is how things flow on the network—how do information or passengers or cargo flow? Can they can from point A to point B? If so, how long does it take? In networks, we allow things to travel from one vertex to another vertex along edges. The routes that things can travel along are called *paths* or *walks*.

Definition 1.5.1. *Let $G = (V, E)$ be a graph. We say a (non-empty) sequence of vertices $\gamma = (v_1, v_2, \dots, v_r, v_{r+1})$ in V is a* **path** *or* **walk** *if $(v_i, v_{i+1}) \in E$ for $1 \le i \le r$. The* **length** *of the path is $\mathrm{len}(\gamma) := r$. We say v_1 is the* **start vertex** *and v_{r+1} is the* **end vertex** *of γ. If $v_i \neq v_j$ for $1 \le i \ne j \le r+1$, we say the path is* **simple**.

If $v_{r+1} = v_1$, we say γ is **closed**. *If $\gamma = (v_1, \dots, v_r, v_1)$ is a closed path with $v_i \neq v_j$ for $1 \le i \ne j \le r$, we say γ is a* **(simple) cycle** *or* **circuit**.

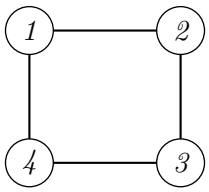
Alternatively, we can specify a path by a sequence of edges, rather than a sequence of vertices. Namely, a sequence of *adjacent* edges $(e_1, e_2, \dots, e_r)$ defines a path of length r. Here adjacent means that e_2 starts where e_1 ends, e_3 starts where e_2 ends, and so on. Thus the length of a path is the number of edges in the path, not the number of vertices. Just as we will allow vertices to repeat in our paths, edges may also repeat. On the other hand, since vertices may not repeat in simple paths or cycles (except for the first and last vertex of a cycle), edges cannot repeat in simple paths or cycles.

We will allow for paths of length 0, i.e., of the form (v) for any vertex $v \in V$. This does not require G having a loop at v, i.e., $(v, v) \in E$. If G does have a loop at v, this means there is a closed path (or cycle) of length 1, denoted (v, v)—which in this case coincides with our edge notation, which starts and ends at v.

Note: this terminology is not entirely standard. Many authors assume all paths are simple. We will not. On the other hand, we will assume all cycles are simple (not all authors do this, or some may only admit cycles of length ≥ 3), and use the term closed path when we want to discuss non-simple cycles.

The terms walk and circuit, however, are fairly standard.

Example 1.5.2. *Let $n > 2$. A* **cycle graph** *of order n is a graph of the form $G = (V, E)$ where $V = \{v_1, v_2, \ldots, v_n\}$ and $E = \{\{v_1, v_2\}, \{v_2, v_3\}, \ldots, \{v_{n-1}, v_n\}, \{v_n, v_1\}\}$. (Recall $\{v_i, v_j\}$ means an undirected edge, as opposed to (v_i, v_j).) Here is a cycle graph of order 4.*



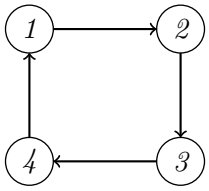
There are infinitely many paths from 1 to 4: (1,4), (1,2,1,4), (1,2,3,4), (1,2,3,4,1,2,3,4), ... However, there are only two simple paths from 1 to 4: (1,4) and (1,2,3,4).

For arbitrary order n, there are $2n$ cycles on G, all of length n—for each vertex v, there are 2 that start and end at v—e.g., (1,2,3,4,1), (1,4,3,2,1). However, there are infinitely many closed paths—you can keep going around the cycle as many times as you want.

All cycle graphs of order n are isomorphic, so we sometimes say the *cycle graph of order n, and denote it C_n.*

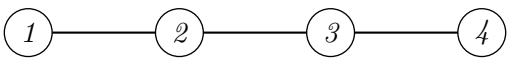
If G is any graph and γ is a cycle of length n, then the vertices and edges of γ define a cycle graph of order n. Hence cycles in any graph may be regarded as cycle graphs.

One can also consider directed cycle graphs, e.g.,



In this case there are exactly n cycles (all of length n) since one can only travel in one direction.

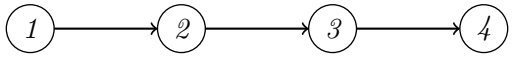
Example 1.5.3. *A* **linear graph** *(or* **path graph***) of order n, is a graph of the form $G = (V, E)$ where $V = \{v_1, v_2, \ldots, v_n\}$ and $E = \{\{v_1, v_2\}, \{v_2, v_3\}, \ldots, \{v_{n-1}, v_n\}, \{v_n, v_1\}\}$. Here is a line graph on 4 vertices.*



Again, all linear graphs on n vertices are isomorphic, and a simple path of length n yields a linear graph of order n.

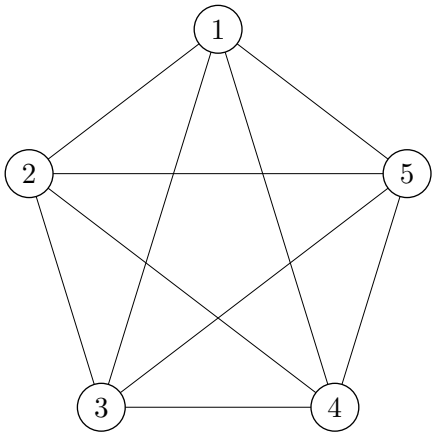
On a linear graph, there is a unique simple path between any given pair of vertices. Of course, if we do not require simple, an infinite number of paths are possible. This has no cycles of length ≥ 3*. (Note: any undirected edge defines a cycle of length 2—e.g., we have the cycle* $(1, 2, 1)$*.)*

We can also consider directed linear graphs, e.g.,



Here, there is a path from 1 to 4, but not from 4 to 1. This has no cycles.

Example 1.5.4. *A* **complete graph** *of order* n *is a simple undirected graph on* n *vertices that has all possible* $n(n-1)/2$ *edges. I've shown you one on 5 vertices before:*

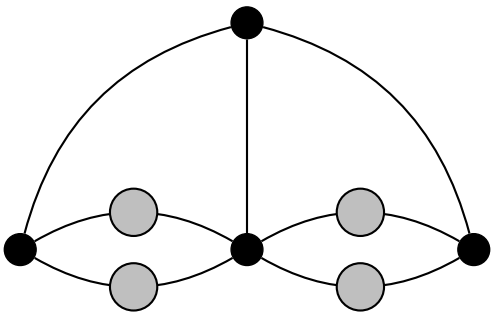


Again, all complete graphs on n *vertices are isomorphic, and we usually speak of* the *complete graph on* n *vertices, and denote it by* K_n *(you know, for komplete).*

In this case, there are loads of paths and cycles. For example, here are the some simple paths from 1 to 5: (1,5), (1,2,5), (1,3,5), (1,4,5), (1,2,3,5), (1,3,2,5), ... (If we want to enumerate them all, it's easiest to be systematic—I started counting by length.) For any two vertices, there are simple paths between them of lengths 1, 2, 3 and 4. There are cycles starting at any vertex of lengths 2, 3, 4 and 5.

The directed complete graph is the same as the undirected complete graph, by our convention of regarding directed graphs with symmetric edge sets as undirected graphs.

Example 1.5.5 (Königsberg bridge problem)**.** *Recall the graph from the Königsberg bridge problem.*



Here each of the black vertices represent landmasses, the edges represent bridges, and the grey vertices are just auxillary vertices used to turn the hypergraph (i.e., the multiedges) into a graph (i.e., ordinary edges). The problem was to find a path that traverses each edge exactly once (note the problem has not changed by our addition of auxillary vertices).

Euler's solution was the following. If there is such a path, then for each vertex in the path, except possible the start and end vertices, one needs to arrive at this vertex the same number of times one leaves this vertex. Hence, the degree of such vertices must be even. However, all black vertices on this graph have odd degree. So such a path is impossible. (Nowadays such paths are called Eulerian paths*, and one can show they exist if and only if the number of vertices of odd degree is either 0 or 2.)*

Connectedness

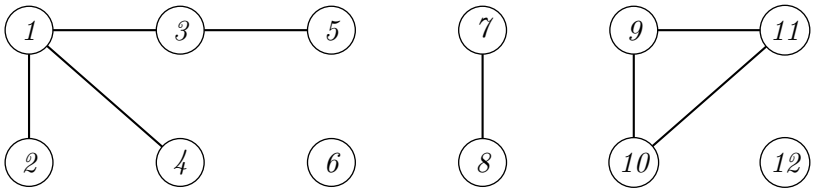
Now we can introduce the notion of *connectedness* which, at least for undirected graphs, will (essentially tautologically) tell us if things can get from point A to point B on a graph.

Definition 1.5.6. *Let $G = (V, E)$ be an undirected graph, and $v_0 \in V$. The* **connected component** *of v_0 is the set of all $v \in V$ such that there exists a path from v_0 to v. The* **connected components** *of G are the subsets of V which arise as connected components of some $v_0 \in V$.*

Proposition 1.5.7. *The connected components of an undirected graph $G = (V, E)$ partition V into disjoint subsets.*

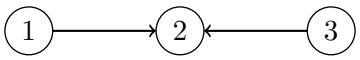
Proof. This follows because being in the same connected component is an equivalence relation; see Exercise 1.5.3. □

Example 1.5.8. *Consider the graph*



The connected component of 1 is the same as the connected component of 2, or 3, or 4, or 5. Similarly for 7 and 8, or 9, 10, and 11. Then the connected components of G are $\{1, 2, 3, 4, 5\}$, $\{6\}$, $\{7, 8\}$, $\{9, 10, 11\}$, and $\{12\}$. Hence the connected componets of G partition the vertices into 5 disjoint sets.

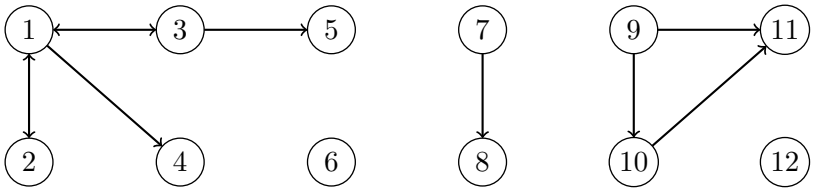
For directed graphs G, we obviously cannot do the same thing. Consider for example



Then, if we were to use the above definition, the connected component of 1 would be $\{1, 2\}$, the connected component of 2 would be $\{2\}$ and the connected component of 3 would be $\{2, 3\}$. So this doesn't give a partition of our digraph. There are a couple of possible ways to try to define connected components for digraphs. Here is perhaps the most naive way.

Definition 1.5.9. *Let $G = (V, E)$ be a directed graph, and let $G' = (V, E')$ be the associated undirected graph, i.e., let $E' = \{(u, v), (v, u) : (u, v) \in E\}$. The* **connected components** *of G are the connected components of G'.*

By definition, the connected components again partition the vertices of a digraph into disjoint subsets. For example, the connected components of the digraph

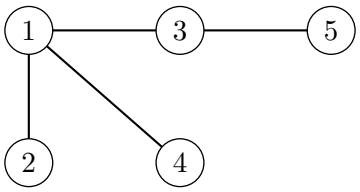


are the same as those for the graph in Example 1.5.8, as the associated undirected graph is the same.

One virtue of this definition of connected components is that it allows us to break up arbitrary graphs into smaller, and hopefuly bite-size, pieces.

Definition 1.5.10. *Let $G = (V, E)$ and $G' = (V', E')$ be graphs. If G is undirected, we assume G' is also undirected. We say $G' = (V', E')$ is a* **subgraph** *of G if $V \subset V'$ and $E \subset E'$.*

Often we will consider connected components as subgraphs of $G = (V, E)$. Note that a subgraph is not determined by just selecting the vertices—you also need to decide which edges to include. However, by convention, if we specify a subgraph only by a subset V_0 of vertices, we mean the graph $G_0 = (V_0, E_0)$ where $E_0 = \{(u, v) \in E : u, v, \in V_0\}$, i.e., we include all possible edges using only the vertices in V_0. For example, the subgraph associated to the connected component of 1 in Example 1.5.8 is



Definition 1.5.11. *Let G be a graph. We say G is* **connected** *if G has exactly one connected component.*

Then any connected component of any graph defines a connected subgraph. The number of connected components as well as their orders/sizes (number of vertices or number of edges), and the property of being connected, are all invariants of graphs. Furthermore, if we know all the connected components of G, we "union" them back together to get the original graph G.

For many problems, one reduces to the study of connected graphs. For undirected graphs G, we can get from vertex u to vertex v if and only if they are in the same connected component. In particular, we can get from any vertex u to any other vertex v if and only if G is connected. Consequently, being connected is one basic property we typically want in things like communication and transportation networks. From a practical point of view—this means we want algorithms to determine if a graph is connected, or to determine the connected components. We will briefly discuss algorithms later.

For directed graphs G (which most communication and transportation networks are not), the notion of connected components is not sufficient—if u and v are not in the same connected component, then v is not reachable from u, but if they are in the same connected component, v may or may not be reachable from u.

Now let's take a look at an alternative notion of connectedness for digraphs.

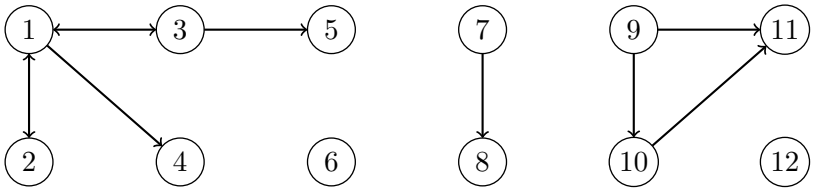
Definition 1.5.12. *Let $G = (V, E)$ be a directed or undirected graph, and $v_0 \in V$. The* **strongly connected component** *of v_0 is the set of all $v \in V$ such that there exists both a path from v_0 to v and a path from v to v_0. The* **strongly connected components** *of G are the subsets of V which arise as strongly connected components of some $v_0 \in V$.*

We say G is **strongly connected** *if it has exactly one strongly connected component.*

Note that if G is undirected, strongly connected components are the same as connected components since having a path from v_0 to v is equivalent to having a path from v to v_0.

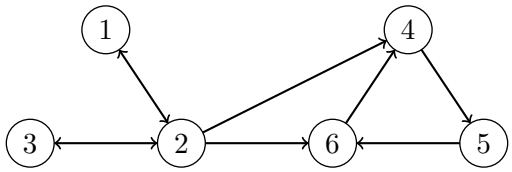
As with connected components, the strongly connected componentes partition the vertices into disjoint subsets (Exercise 1.5.5), and these components are maximal such that one can get from any vertex to any other vertex in same strongly connected component. In particular, G is strongly connected if and only if one can get from vertex u to vertex v for any two vertices u, v in G.

However, knowing the strongly connected components (even together with the connected components) is not enough to completely answer the question can one get from u to v. Namely, tt still may be possible to get from u to v though u and v are in different strongly connected components. For instance, in the digraph



the strongly connected components are $\{1, 2, 3\}$, $\{9, 10, 11\}$ and then the singleton sets $\{4\}$, $\{5\}$, $\{7\}$, $\{8\}$ and $\{12\}$. Just looking at the strongly connected components does not tell us if there is a path from 4 to 5 or a path from 7 to 8, though we can rule out the possibility of a path from 4 to 8 by looking at connected components. In general, there is no way to partition the vertices of a directed graph G in such a way that one can definitively and easily say if there is a path from one given vertex u to another given vertex v. Rather, one can compute the set of all vertices reachable from u (cf. our original definition for connected component for undirected graphs) and check if v is in this set or not.

We remark many authors don't consider our notion of connected components for digraphs—they only consider strongly connected components, and may occasionally just refer to them as the connected components or components of the digraph. (We may sometimes say components of G for connected components of G.) However, I defined the above notion of connected components because is useful for problems where we may want to break up digraphs into smaller digraphs. Note that one typically cannot do this with strongly connected components because one cannot piece together a digraph G from just its strongly connected components (viewed as subgraphs). For instance, the strongly connected component graphs of



are

Just knowing the two strongly connected component graphs does not tell us how to paste them together to get our original graph, since there are many ways these two strongly connected components could be "weakly connected."

Distance

Now, assuming that we can get from point A to point B in the graph, our next question is how do we determine how long it takes? We use the model that it takes 1 time unit to traverse each edge. Later we will account for different time (or money) costs per edge by using weighted graphs.

Definition 1.5.13. *Let $G = (V, E)$ be a graph. For $u, v \in V$, let $\Gamma(u, v)$ denote the set of paths from u to v. We define the* **distance** *$d(u, v)$ between u and v to be*

$$d(u,v) := \begin{cases} \infty & \text{there is no path from } u \text{ to } v; \\ \min\{\text{len}(\gamma) : \gamma \in \Gamma(u,v)\} & \text{else.} \end{cases}$$

In other words, the distance between two vertices is the least number of steps (edges) it takes to get from one to the other (if we are working with directed graphs, which vertex is first is important here). In particular, the vertices which are distance 1 from u are the **neighbors** of u. For any vertex u, $d(u, u) = 0$ since we have allowed paths from u to u of length 0 in our definition of path.

Example 1.5.14. *Let's consider $d(1, 4)$ from our above (undirected) examples. In the cycle graph C_4, 1 and 4 are adjacent, so $d(1, 4) = 1$. In the line graph, $d(1, 4) = 3$. In the complete graph K_5, all vertices are adjacent, so $d(1, 4) = 1$.*

Proposition 1.5.15. *Let $G = (V, E)$ and $u, v \in V$. Suppose $0 \neq d(u, v) < \infty$. Then there is exists a path γ from u to v such that $\text{len}(\gamma) = d(u, v)$. Furthermore, any such γ must be a simple path.*

Proof. The assumptions mean $\Gamma(u, v)$ is non-empty. Since the set $\{\text{len}(\gamma) : \gamma \in \Gamma(u, v)\} \subset \{0, 1, 2, \ldots\}$, it has a least element, i.e., the minimum is well-defined, and so there exists some γ such that $\text{len}(\gamma) = d(u, v)$. Consider any such $\gamma = (u = v_1, v_2, \ldots, v_r = v)$. If γ is not simple, then some $v_i = v_j$ for $i \neq j$. (By assumption $u \neq v$, so $(i, j) \neq (1, r)$.) Say $i < j$. Then we can consider the strictly shorter sequence $\gamma' = (u = v_1, v_2, \ldots, v_i, v_{j+1}, v_{j+2}, \ldots, v_r = v)$. Since v_i, v_j, we must have $(v_i, v_{j+1}) \in E$, whence γ' is also a path from u to v. However, it is shorter than γ, contradicting the minimal length of γ. □

The following proposition shows that graph distances behave at least somewhat like Euclidean ones.

Proposition 1.5.16 (Triangle Inequality)**.** *Let $G = (V, E)$. If $u, v, w \in V$, then*

$$d(u, v) + d(v, w) \geq d(u, w).$$

Proof. Suppose this is not true for some u, v, w. Then $d(u, v) + d(v, w) < d(u, w)$. It suffices to assume all these distances are finite (why?). Hence there is a path from u to v of length $d(u, v)$, and a path from v to w of length $d(v, w)$. "Adding" these paths together (following one, then the other) gives us a path from u to w of length $d(u, v) + d(v, w) < d(u, w)$, contradicting that $d(u, w)$ is the minimum length of paths from u to w. □

If you've studied topology, this makes any undirected graph G into a metric space—i.e., the distance function satisfies all the usual properties ($d(u, v) \geq 0$ with equality if and only if $u = v$, $d(u, v) = d(v, u)$ and the triangle inequality). This is not true for directed graphs, since $d(u, v) \neq d(v, u)$ in general (e.g., consider the directed linear graph on 4 vertices above—$d(1, 4) = 3$ but $d(4, 1) = \infty$).

If we have some sort of communication or transportation network, we want some measure (or measures) of efficiency (i.e., how fast things can travel between two nodes). Here is the most basic one.

Definition 1.5.17. *Let $G = (V, E)$ be a graph (directed or not). The* **diameter** *of G, denoted* $\operatorname{diam}(G)$*, is the maximum distance $d(u, v)$ for $u, v \in V$.*

The diameter is finite if and only if G is connected and undirected, or G is strongly connected and directed.

Example 1.5.18. *For the cyclic graph C_n of order n, we have* $\operatorname{diam}(C_n) = \lfloor \frac{n}{2} \rfloor$.* *(Draw a few examples.) The diameter of the directed cyclic graph of order n is $n - 1$.*

The diameter of a linear graph of order n is $n - 1$. The diameter of a directed linear graph is ∞.

The diameter of the complete graph K_n is $\operatorname{diam}(K_n) = 1$.

The diameter provides an upper bound on the time it takes to get between two points in the graph. Thus smaller diameters indicate higher efficiencies for graphs. You can also think of diameter of being a measure of "how connected" a graph is—the smaller the diameter, the closer together the nodes are, so things are better connected in some sense. For (strongly) connected (di)graphs, the diameter of $n - 1$ for directed cyclic or undirected linear graphs is the worst case possible, as this proposition shows.

Proposition 1.5.19. *Let $G = (V, E)$ be a connected undirected or strongly connected directed graph of order n. Then* $\operatorname{diam}(G) \leq n - 1$.

Proof. Let $u, v \in V$. Then $d(u, v) < \infty$ and there is a path γ of length $d(u, v)$ from u to v. By Proposition 1.5.15, γ must be simple. This means γ has no repeated vertices, i.e., it has at most n vertices, i.e., it has at most $n - 1$ edges. □

*$\lfloor x \rfloor$ denotes the floor, or greatest integer, function—round x down to the nearest integer, while $\lceil x \rceil$ denotes the ceiling function—round x up to the nearest integer.

Another measure of how well connected a graph is to look at the *average distance* between vertices. This will give us a (often times better) estimate on how long it will take to get from a random vertex to another random vertex. This is like looking at the average case running time of an algorithm instead of the worst case running time. Which measure is more appropriate depends upon the particular problem, but as the diameter is an easier quantity to get a handle on, we will focus primarily on that.

However, let us at least give a precise definition. For a directed or undirected graph $G = (V, E)$, define the **average distance** on G to be

$$d_{avg}(G) := \frac{1}{n(n-1)} \sum_{u \in V} \sum_{v \in V, v \neq u} d(u, v).$$

Note $n(n-1)$ is the total number of ordered pairs (u, v) of distinct vertices, and we average the distance over those. This is not much harder to compute than the diameter when working with specific graphs on the computer, but is considerably harder to analyze theoretically. (For instance, try calculating the average distance for a cyclic graph C_n or linear graph of order n in terms of n. It is not horrible, but not nearly as easy calculating the diameter.)

Algorithms

Let's start off with the question of designing an algorithm to find the connected component of a given v_0 of an undirected graph G of order n. The idea is straightforward, though I'll write a reasonable amount of detail which will make it easier to code.

Algorithm 1.5.20. *Find the connected component of v_0.*

1. *Add v_0 to a new set* `visited` *(this keeps track of which vertices we've already visited, and will be the connected component of v_0 when we're done).*
2. *Add each neighbor of v_0 to* `visited`*. Let* `newverts` *be this set of neighbors just added.*
3. *For each vertex in* `newverts`*, find their neighbors. For each neighbor not in* `visited`*, add this vertex to* `visited`*. Then let* `newverts` *be the set of these vertices just added.*
4. *Repeat last step until* `newverts` *is empty (i.e., until you're no longer adding more vertices).*
5. *Ouput* `visited`*.*

In other words, we start at v_0, find its neighbors, find its neighbors' neighbors, find the neighbors' neighbors' neighbors, and so on. This process is known as a breath-first search—we search in layers for all the vertices in the component of v_0 (as opposed to a depth-first search, where one searches in succesive lines out from v_0). At each step in this process, we only travel out from vertices we haven't previously visited. This avoids an infinite loop, and makes our algorithm fairly efficient.

Let's think about how this search is expanding out in a little more detail. (This is what the set `newverts` is at each stage.) From v_0, we go to it's neighbors, i.e., vertices distance 1 from v_0. Then we find the neighbors of the distance 1 vertices that we haven't already seen. These will be of distance ≤ 2 from v_0. Well, the only things we've seen are the things of distance 0 and 1 from v_0. Hence our new set of vertices is precisely the vertices distance 2 from v_0. Continuing in this process, after d iterations, the set `newverts` is precisely the set of vertices of distance d from v_0.

Note: I could've absorbed Step 2 into Step 3 of this algorithm by just letting `newverts = {v_0}` in Step 1. I would do this when coding—I just separated out Step 2 for the purposes of exposition.

Now, let's analyze this algorithm. At some point in the algorithm, for each vertex in the connected component of v_0, I need to find the neighbors of v_0 and go through each neighbor, check if it was already visited and either add it to the connected component or not. Let's say there are m vertices in the connected component of v_0. Each such vertex has at most $m-1$ neighbors (we can ignore loops), hence this algorithm has a running time of $O(m^2)$. Since $m \leq n$, we can also say this algorithm runs in $O(n^2)$ time. In fact, if one uses adjacency lists, this algorithm can be implemented in $O(n + |E|)$ time.

With this algorithm in hand, it is easy to find all connected components of $G = (V, E)$. Pick a random $v_0 \in V$. Find its connected component V_0. Now take a random $v_1 \in V - V_0$, and find its connected component V_1. Continue this process until all vertices have been exhausted.

Similarly, one can determine if G is connected as follows. Pick a random $v_0 \in V$. Find its connected component V_0. Then G is connected if and only if $|V_0| = n$.

Algorithms to find strongly connected components of a digraph G are a bit more involved, and we will not get into them, but just mention this can also be done in $O(n^2)$ time.

Now that we've addressed algorithms pertaining to connectedness, let's move on to distance. Fix two vertices u, v of a graph G (directed or undirected). How can we compute the distance $d(u, v)$? What have we been doing by hand? We've (at least I have, and I assume this is what you've been doing too) essentially been finding all simple paths from u to v, of which there are finitely many and see what path or paths are shortest possible. This is easy to by hand for small graphs, but to do for large graphs, or to automate on the computer, it requires some work to generate all simple paths from u to v.

However, if we remember our algorithm for finding the connected component of u, we organized all vertices in the connected component of u by their distance from u. If we just kept track of that information in our algorithm, we'll have the distance not just from u to v, but from u to any other vertex in the graph (if the other vertex is not in the connected component of u, we know the distance is infinite).

Here is Python code to do just that, using adjacency matrices and our previous function `neighbors`. The function is called `spheres` for the following reason. Given a graph $G = (V, E)$ and a vertex $u \in V$, the **sphere of radius** r **centered at** u is the set

$$S_u(r) = \{v \in V : d(u, v) = r\}.$$

It is called a sphere because this is the same definition as for spheres in the Euclidean space familiar to you.

Python 2.7

```
def spheres(A, i):
    sph = [ { i } ]
    visited = { i }
    newvert = { i }
    while len(newvert) > 0:
        new = set()
        for j in newvert:
            neigh = neighbors(A,j)
            for k in neigh:
```

```
                if k not in visited:
                    new.add(k)
        newvert = new
        if len(newvert) > 0:
            sph.append(newvert)
            visited = visited.union(new)
    return sph
```

This function returns the list $[S_u(0), S_u(1), S_u(2), \ldots, S_u(m)]$ where m is the maximum distance from u of any vertex in the component of u. Here each $S_u(r)$ is returned as a Python set. Consequently, if you enter `sph = spheres(A, i)`, then you can access $S_u(r)$ simply by `sph[r]`. This function works for directed and undirected graphs. It really is essentially an implementation of Algorithm 1.5.20 where we simply keep track of which vertices are distance r from u, so the same analysis applies and it runs in $O(n^2)$ time.

With this function, we can compute $d(u, v)$ as follows.

Algorithm 1.5.21. *Compute $d(u, v)$.*

1. *Compute the spheres $S_u(r)$ centered at u.*
2. *For each possible value of r, check to see if $v \in S_u(r)$. If so, output r.*
3. *Otherwise, output ∞ (which in the computer we often code as -1).*

Here the first step take $O(n^2)$ times, the second can be done in $O(n)$ time (the number of vertices in the union of the spheres is at most n), and the last step takes $O(1)$ time. Hence this algorithm for computing the distance takes $O(n^2) + O(n) + O(1) = O(n^2)$ time.

We remark one could make this more efficient by not computing all spheres $S_u(r)$ first, but compute them inductively and check at each step if $v \in S_u(r)$.

Lastly, we present an algorithm for computing the diameter. One could simply try to use the definition and compute $d(u, v)$ for all u, v, and take the maximum distance. However, we can do it more efficiently than that.

Algorithm 1.5.22. *Compute* $\operatorname{diam}(G)$*, where $G = (V, E)$.*

1. *For each $u \in V$, do the following:*
2. *Compute the spheres $S_u(r)$ centered at u.*
3. *Let $B(u) = \bigcup_r S_u(r)$. If $|B(u)| < n$, some vertex is not reachable from u, so return ∞.*
4. *Otherwise, let d_u be the maximum r for which $S_u(r)$ is nonempty. (We can get this by `len(spheres(A,i))`.) This is the maximum distance any vertex can be from u.*
5. *Output* $\max\{d_u : u \in V\}$*, which must be the diameter.*

See Exercise 1.5.13 for the analysis. Note that if we were just working with undirected graphs, one could avoid doing Step 3 for each u, and just do it for one u at the beginning to ensure G is connected.

Exercises

Exercise 1.5.1. *Consider the komplete graph K_4 on $\{1,2,3,4\}$.*
(i) Enumerate all simple paths from 1 to 4. How many are there?
(ii) How many cycles of lengths 2, 3 and 4 are there on K_4?

Exercise 1.5.2. *(i) Consider a cycle graph C_5 on $\{1,2,3,4,5\}$. For each vertex j, compute $d(1,j)$.*
(ii) Do the same for the directed cycle graph on $\{1,2,3,4,5\}$.

Exercise 1.5.3. *Let $G=(V,E)$ be an undirected graph. Show that being in the same connected component is an equivalence relation, i.e., show:*
(i) for any $v_0 \in V$, v_0 is in the connected component of v_0;
(ii) if v_1 is in the connected component of v_0, then v_0 is in the connected component of v_1; and
(iii) if v_2 is in the connected component of v_1 and v_1 is in the connected component of v_0, then v_2 is in the connected component of v_0.

Exercise 1.5.4. *Let $G=(V,E)$ be a graph. Show that $v_0 \in V$ is an isolated node (i.e., degree 0) if and only if its connected component has size 1.*

Exercise 1.5.5. *Let $G=(V,E)$ be a digraph. Show the strongly connected components partition V into disjoint subsets by showing that being in the same strongly connected component is an equivalence relation.*

Exercise 1.5.6. *Let G be a connected undirected graph of order n. Show G has at least $n-1$ edges.*

Exercise 1.5.7. *Let $n \geq 2$. Consider the cycle graph $C_{2n}=(V,E_0)$, and form the graph $G=(V,E)$ on the same vertex set $V=\{1,2,\dots,2n\}$ (with the usual choice of cycle, i.e., the edges are $\{1,2\}$, $\{2,3\}$, ..., $\{2n-1,2n\}$ and $\{2n,1\}$) with $E=E_0 \cup \{n,2n\}$. In other words, we add the "diagonal" edge to C_n from n to $2n$. In G, what is $d(1,n+1)$? Determine* $\mathrm{diam}(G)$.

Exercise 1.5.8. *Let $C_m=(V_1,E_1)$ and $C_n=(V_2,E_2)$ be cycle graphs. Consider the graph $G=(V,E)$ obtained by taking the union (or "direct sum") of C_m and C_n and connecting them with a single edge. Precisely, fix $v_1 \in V_1$ and $v_2 \in V_2$. Then $V=V_1 \cup V_2$ and $E=E_1 \cup E_2 \cup \{\{v_1,v_2\}\}$. Determine* $\mathrm{diam}(G)$.

Exercise 1.5.9. *Determine, in terms of n, the running time of the algorithm described in the text (after Algorithm 1.5.20) to find all connected components of G.*

Exercise 1.5.10. *Using the* `spheres` *function, write functions* `component(A,i)`, `components(A)` *and* `is_connected(A)` *to find the connected component of vertex i, all components of G, and determine if G is connected, where A is the adjacency matrix for a directed or undirected graph G. (Caution: remember to convert A to the adjacency matrix for the associated undirected graph.)*

Exercise 1.5.11. *Using the* `spheres` *function, write a function* `distance(A,i,j)` *that computes the distance from vertex i to vertex j given a directed or undirected graph adjacency matrix A.*

Exercise 1.5.12. *Write a function* `diameter(A)` *to compute the diameter of a graph given its adjacency matrix A using the above algorithm.*

Exercise 1.5.13. *Analyze the running times for the following two algorithms to compute the diameter: (i) the naive algorithm of computing all possible distances and taking the maximum, and (ii) Algorithm 1.5.22.*

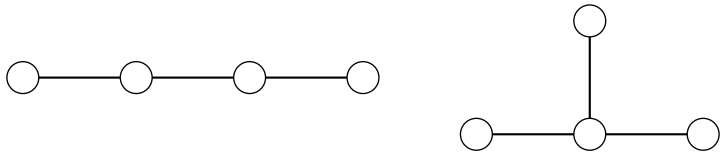
1.6 Network design, trees, k-connectedness and regularity

▶ Here graphs are undirected unless otherwise stated.

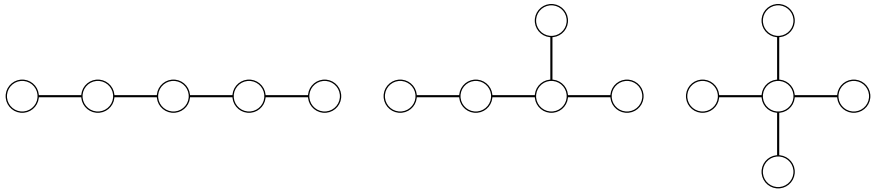
Now let's consider the problem of designing a network. When designing something in real life, there are pros and cons, costs and benefits, that we need to balance out. In network design, there is typically a cost associated to each edge of the nework (for construction, maintenance, or both—e.g., think of a highway network). Hence one wants to minimize the number of edges in the network while maintaining certain standards of performance.

Let's first consider just the following simple constraint: the network should be connected. This is of course a minimum necessity for communication and transportation networks. How few edges do we need to make a connected network on n vertices? Recall Exercise 1.5.6 says we need at least $n-1$ edges. Furthermore we can always make a network connected with $n-1$ edges by using a linear or path graph. What else can we do?

Well, for $n = 2$, there is only 1 graph with 1 edge, and it is connected. For $n = 3$, again there are only 2 possibilities with 2 edges. For $n = 4$, we have 2 possibilities (up to isomorphism):



For $n = 5$, we have 3 possibilities:



The above graphs are all examples of an important family of graphs, namely trees.

Definition 1.6.1. *Let G be a (simple undirected) graph. We say G is a* **tree** *if it is connected and has no cycles of length > 2.*

Proposition 1.6.2. *Let $G = (V, E)$ a connected graph of order n. Then G is a tree if and only if $|E| = n - 1$.*

Proof. Both directions will be proved by proving the contrapositive.

($\Leftarrow$) First we claim that if G has a cycle of length $r > 2$, it must have more than n edges. By relabelling vertices, we may assume $V = \{v_1, \dots, v_n\}$, where there is a cycle of length r on $\{v_1, \dots, v_r\}$. The existence of the cycle means there are at least r edges involving only $v_1, \dots, v_r$. Since G is connected, one of the remaining vertices, say v_{r+1} must have an edge to one of $v_1, \dots, v_r$. Thus there are at least $r + 1$ edges involving only $v_1, \dots, v_{r+1}$. Continuing this argument shows there are at least n edges involving $v_1, \dots, v_n$, $|E| \geq n$ as claimed.

Hence if G is connected with $n - 1$ edges, it has no cycles of length > 2, i.e., is a tree.

($\Rightarrow$) Now suppose $|E| \geq n \geq 3$. We want to show G has a cycle of length > 2.

A **leaf** is a vertex with degree 1. It is clear no cycle of length > 2 will involve a leaf. Thus we may prune all the leaves, i.e., delete the leaves and the corresponding edges from the graph G to

get a subgraph G'. If there were l leaves, now G' is a graph on $n-l$ vertices with at least $n-l$ edges. It is impossible for all vertices of a connected graph on ≥ 3 vertices to be leaves (if this is not clear, think about the argument in ($\Leftarrow$)), so G' indeed has some vertices. Furthermore, since there are no graphs on $m = 1$ or 2 vertices with m edges, G' must have at least 3 vertices.

Now prune G'. Continue this process of pruning leaves until there are no more. (This process must terminate as the number of vertices becomes strictly smaller at each step, with a lower bound of 3.) This leaves (no pun intended) us with a graph G_0 with $m \geq 3$ vertices and at least m edges. Since G_0 has no leaves, each vertex of G_0 has degree ≥ 2. This means G_0 has a cycle of length > 2 by Exercise 1.6.3, as desired.

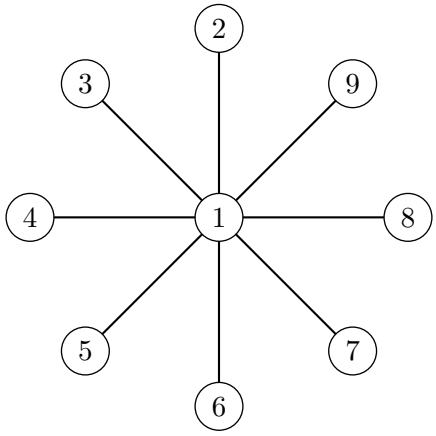
This means that if G is a tree, it has at most $n-1$ edges (we just showed the contrapositive for $n \geq 3$, but this statement is trivial when $n = 1$ or $n = 2$), but it has to have at least $n-1$ since it is connected, i.e., $|E| = n - 1$. □

Remark: the above proof is typical of classical graph theory, and we'd be doing a lot more arguments like this in a standard graph theory course than we will in this one.

In other words, the trees are precisely connected graphs with the minimum possible number of vertices, i.e., the best candidates for our overly-simplified network design problem. Now we can ask, is there any way in which some of the trees might form a better network than others?

Well, another nice property we would like our network to have, besides being connected, is **efficiency**, i.e., one should be able to get between two points in the network relatively quickly, so we want small diameter (or average distance). If we look back at our trees for $n = 4$ and $n = 5$, it is clear the ones on the right are more efficient, and the ones on the left (i.e., the linear graphs) are least efficient. We can generalize the trees on the right to n vertices as follows.

Example 1.6.3. *Let $n \geq 3$. The* **star graph** *of order n is the undirected graph $G = (V, E)$ with $V = \{1, 2, \ldots, n\}$ and $E = \{\{1, 2\}, \{1, 3\}, \ldots \{1, n\}\}$. That is, vertex 1 is connected to all other vertices, and there are no other edges. We can picture 1 as being the hub at the center of the network. Here is the star graph on 9 vertices is*



This graph is connected with $n-1$ edges and has diameter 2. In particular, it is a tree.

Since the only graph with diameter 1 is the complete graph (why?), the star graph minimizes diameter among all undirected graphs with less than $n(n-1)$ edges, and this uses the fewest number of edges possible for any connected graph on n vertices.

This leads us to the following, perhaps surprising, observation. Naively, you might expect the more edges there are in your graph, the smaller your diameter should be—perhaps if you gradually allow more edges, you can get smaller and smaller diameters. Instead, there's a very sharp trichotomy here. If you have $< n-1$ edges, you must be disconnected so the diameter is infinite. If you have between $n-1$ edges and n^2-n-1 edges (anything short of K_n), you can achieve diameter 2. If you have the maximum possible n^2-n edges, then you must be K_n and have diameter 1. (However, you can get smaller and smaller average distances by adding more edges—see Exercise 1.6.4.)

Rather, what seems to be more important for getting a small diameter is that the edges are well chosen, e.g., making a star graph as opposed to a linear graph. This should indicate that some care should be take in the design to make a good network. On the other hand, we'll encounter a differing philosophy later which says that "random graphs" tend to make good networks. Roughly the idea is that, sure if you pick a tree at random you're unlikely to end up with the star graph, but you're equally unlikely to end up with the linear graph, and chances are that your random tree will have diameter closer to 2 than to $n-1$. This is explored in Exercises 1.6.6 and Exercises 1.6.7.

Okay, so we seem to have given a reasonable answer to the problem of designing an efficient network on n nodes. Is there anything we've overlooked? Well, in a perfect world, not really. There's the aspect that for a physical network, different links will need to be different lengths, and may have different costs associated with them (both for building/maintaining and for travelling along), but we'll revisit this issue later, albeit fairly briefly.

There are two main issues with using a star graph for a network. First, in the real world, things fail all the time. A cable (edge) could get severed, a server (node) might be down for maintenance or have hardware issues, roads (edges) or airports (nodes) might be due to the weather. If the hub of a star graph fails, then the whole network goes down. Or if a single edge goes down, the corresponding outer vertex becomes stranded. A network that can reasonably handle such failures is said to be **robust**.

The second main issue has to do with traffic, or network flow. If we use a star graph as a network, all traffic must pass through the central hub. Then during busy times, it may be that traffic gridlocks at the hub rendering the network essentially non-functional for a period of time. If we want to study traffic issues precisely, then one can define formal notions of the capacity of a network (how much/how fast information/traffic can pass through) and the network flow. However, if our network is robust, this will mean that there are several different ways to get from one point to another, and therefore traffic can be rerouted when necessary to cut down on gridlock. Hence we will focus on robustness now.

Here are a couple of basic measures of robustness.

Definition 1.6.4. *Let $G = (V, E)$ be a graph (possibly directed and non-simple) of order $n > k$ with $n > 1$ We say G is k-**connected**, or k-**vertex-connected**, if the removal of any subset of $< k$ vertices (and involved edges) yields a connected subgraph. The* **vertex connectivity** *$\kappa(G)$ of G is the maximal non-negative integer such that G is $\kappa(G)$-connected. Alternatively, $\kappa(G)$ is the minimal number of vertices one needs to remove to make G disconnected or have order 1.*

A **vertex cut** *is a set of vertices V_0 of V such that the subgraph $V - V_0$ is disconnected. Hence the minimal size of a vertex cut (when one exists) is $\kappa(G)$.*

Note $\kappa(G)$ tells us that if $< \kappa(G)$ nodes of our network fail, the remainder of our network will still be functional (connected). Note that G is 1-connected if and only if G is connected (and

$n > 1$), and $\kappa(G) = 0$ means G is disconnected. However, G being 0-connected does not mean G is disconnected—any k-connected graph is automatically $(k-1)$-connected from the definition (for $k > 0$).

For a directed graph G, being k-connected means the same as the associated undirected graph being k-connected.

Definition 1.6.5. *Let $G = (V, E)$ be a graph (possibly directed and non-simple) of order $n \geq 2$. We say G is k-***edge-connected** *if the removal of any subset of $< k$ edges (but no vertices) yields a connected subgraph. The* **edge connectivity** *$\lambda(G)$ is the maximal non-negative integer such that removing any subset of $< \lambda(G)$ edges (but no vertices) leaves G connected. Alternatively, $\lambda(G)$ is the minimal number of edges one needs to remove to make G disconnected.*

A **cut***, or an* **edge cut** *is a subset of edges E_0 such that the graph $(V, E - E_0)$ is disconnected. The minimal size of a cut equals $\lambda(G)$.*

Note $\lambda(G)$ means that if $< \lambda(G)$ edges of our network fail, our network will still be functional (connected). Again G is 1-edge-connected if and only if G is connected (and $n > 1$), and $\lambda(G) = 0$ means G is disconnected. Also, k-edge-connected implies $(k-1)$-edge-connected (assuming $k > 0$).

For a directed graph G, being k-edge-connected is not the same as the associated undirected graph G' being k-edge-connected, as 1 edge in G' might correspond to 1 or 2 edges in G. However, an edge cut of size k in G corresponds to an edge cut in G' of size $\leq k$, so we can say $\lambda(G') \leq \lambda(G)$.

We avoided defining vertex and edge connectivity for a graph of order 1 (which is connected) to avoid putting more technicalities in the definitions.

We remark that (if $n \geq 2$) edge cuts always exist (you can cut off all edges from a given vertex to isolate it), but vertex cuts do not. For instance, take the linear graph of order 2—we can only remove 1 vertex and still be left with a subgraph (I'm not allowing "empty graphs" on 0 vertices), and either vertex you remove leaves you with a (connected) graph on 1 vertex. This is why the alternative definition of $\kappa(G)$ includes the condition that removing $\kappa(G)$ vertices leaves you with a single vertex.

Example 1.6.6. *The linear graph L_n of order $n \geq 2$ has $\kappa(L_n) = \lambda(L_n) = 1$. To see this, note L_n is connected so $\kappa(L_n), \lambda(L_n) \geq 1$. If $n = 2$, we can only remove 1 vertex as discussed above, so $\kappa(L_2) = 1$. If $n > 2$, we can remove any vertex "in the middle" and this will disconnect the graph, so $\kappa(L_n) = 1$. Similarly, for any $n \geq 2$, if we remove any edge, we disconnect the graph, so $\lambda(L_n) = 2$.*

Example 1.6.7. *More generally, let T be any tree of order $n \geq 2$. Then again $\kappa(T) = \lambda(T) = 1$. To see this, note T must have at least one leaf (otherwise, it has minimum degree 2 and therefore a cycle of length > 2 by Exercise 1.6.3). We can cut the edge from the leaf to disconnect T, so $\lambda(T) = 1$. Again, for $n = 2$ it is trivial to see $\kappa(T) = 1$, so assume $n \geq 3$ now. Then removing a neighbor of a leaf cuts of the leaf from the rest of the tree, so we have a vertex cut of size 1, i.e., $\kappa(T) = 1$.*

Example 1.6.8. *Consider the cycle graph C_n, $n \geq 3$. We can isolate any vertex by removing the two adjacent vertices or edges, however removing any single vertex or edge leaves us with a connected subgraph (a linear graph of order $n - 1$). Hence $\kappa(C_n) = \lambda(C_n) = 2$.*

Example 1.6.9. *The complete graph K_n, $n \geq 2$, has $\kappa(K_n) = \lambda(K_n) = n - 1$. To see this, observe removing any set of $k < n$ vertices leaves us with a complete subgraphs K_{n-k}, i.e., $\kappa(K_n) =$*

$n-1$. On the other hand, suppose there is an edge cut that leave two vertices u and v in different components. How many ways can we get from u to v in K_n? There are many, but here are $n-1$ possibilities: we can go straight from u to v, or for any of the other $n-2$ vertices w, we can go from u to w, then w to v. Note these paths are independent *in the sense that they have no edges in common. Hence to disconnect u from v, we need to get rid of at least 1 edge from each of these paths, i.e., the minimum size of a cut is at least $n-1$. In fact it is exacly $n-1$ since we can just remove all $n-1$ edges from u. Thus $\lambda(K_n) = n-1$.*

Remark: A fundamental theorem about connectivity is **Menger's Theorem**. It states that the number of edges needed to disconnect u and v is the maximum number of independent paths from u to v. There is also a vertex connectivity version. This theorem is a special case of the famous *Max Flow-Min Cut Theorem*, which is a generalization to the setting where one considers each edge having a certain capacity for traffic.

Up till now, we haven't seen any examples with $\kappa(G) \neq \lambda(G)$, so you may be wondering if they exist. Well, they do. How might we construct one? First observe the following.

Proposition 1.6.10. *Let $G = (V, E)$ be a connected graph (possibly directed non-simple) of order $n \geq 2$. Suppose C is an edge cut of minimal size. Then the cut graph $G' = (V, E - C)$ has two connected components, i.e., C partitions V into two disjoint subsets.*

Proof. Exercise. □

Proposition 1.6.11. *Let $G = (V, E)$ be a graph (possibly directed non-simple) of order $n \geq 2$. Then $\kappa(G) \leq \lambda(G) \leq n-1$ if G is undirected and $\kappa(G) \leq \lambda(G) \leq 2n-2$ if G is directed.*

Proof. Consider a minimum edge cut $C = \{e_1, e_2, \dots e_k\}$. This cannot contain any loops, otherwise we would remove the loops and get a smaller edge cut. Thus $k \leq n-1$ if G is undirected and $k \leq 2n-1$ if G is directed.

Now let's show $\kappa(G) \leq \lambda(G) = k$. We may assume now G is undirected, otherwise we can replace it with the associated undirected graph G', which satisfies $\lambda(G') \leq \lambda(G)$.

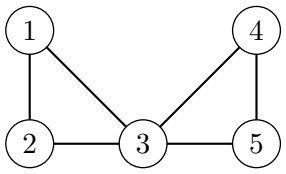
The idea is the following: for each edge in the edge cut, remove a vertex at one end to get a vertex cut. However, one has to be a little careful because doing this arbitrarily may not give us a vertex cut (indeed, they don't exist for K_n).

By definition, $\kappa(G) \leq n-1$, our proposition is true whenever $\lambda(G) \geq n-1$. Hence we may assume $\lambda(G) \leq n-2$. (This rules out K_n.) The previous proposition says that C partitions V into two disjoint subsets V_1 and V_2. We claim that there exist $u \in V_1$ and $v \in V_2$ such that (u, v) is not an edge in C. Otherwise, there must be an edge from each vertex in V_1 to each vertex in V_2, which would give us $|V_1| \times |V_2|$ edges. Consequently, for C to disconnect V_1 from V_2, we would need $k = |V_1| \times |V_2|$. It is easy to see $|V_1| \times |V_2|$ is minimized when either $|V_1|$ or $|V_2|$ is 1, and the other is $n-1$, so $k = \lambda(G) \geq n-1$, which we are assuming is not the case. Therefore, the claim is true.

Now each edge $e_i \in C$, pick a vertex v_i at one end of e_i such that $u \in V_1$ and $v \in V_2$ do not lie in $V_0 = \{v_1, v_2, \dots, v_k\}$. Here the v_i's need not be distinct, so this set may have less than k elements. Removing V_0 removes all the edges in C also, so V_0 forms a vertex cut of size $\leq k$ (it must disconnect u from v). Hence $\kappa(G) \leq k = \lambda(G)$. □

The above argument suggests that, in order to construct a graph with $\kappa(G) < \lambda(G)$, we need a graph where there is a minimal edge cut that involves repeated vertices. Here is an example.

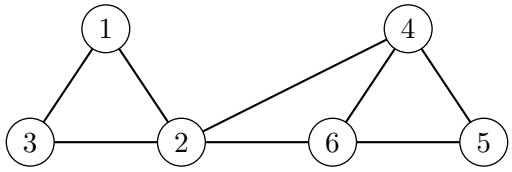
Example 1.6.12. *Consider the following graph G.*



Then $\kappa(G) = 1$ ($\{3\}$ is a vertex cut), but there is no edge cut of size 1. However, there are edge cuts of size 2, e.g., $\{\{1,2\},\{1,3\}\}$, so $\lambda(G) = 2$.

In fact, we can construct an infinite family of examples.

Example 1.6.13. *Take two cycle graphs C_m and C_n, and make a new graph G by connecting one vertex v_0 of C_m to two different vertices of C_n. For example, if $m = n = 3$ (the smallest size of cycle graphs), we can do this*



Then removing any single edge from C_m, or from C_n, or one of the 2 connecting edges will not disconnect the graph since C_m and C_n are 2-edge-connected. Hence our new graph is also 2-edge-connected. However, removing the vertex v_0 of C_m (2 in the above picture) which connects to C_n will disconnect C_n from C_m minus v_0. Thus this graph satisfies $\kappa(G) = 1$ and $\lambda(G) = 2$.

Let's return to the question of network design. First consider the problem of designing a robust network at minimal cost (let's not worry about efficiency yet). Say we want a network that will still be functional (connected) if some number of nodes or edge fail. Then we can set a threshold number k, depending on our expectations of this network, such that if any set of $< k$ nodes or edges fail, our network is still connected. That is, we want a k-connected network. Now we can ask what is the minimum number of edges we need.

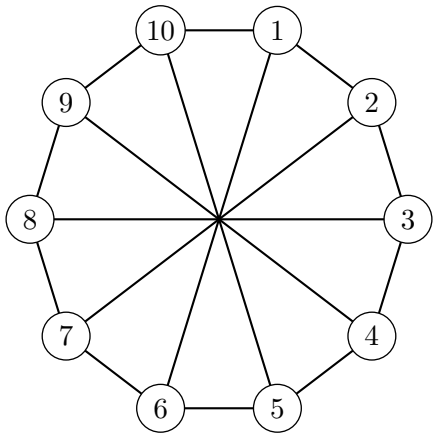
Proposition 1.6.14. *Let $G = (V, E)$ be an (undirected) graph on n nodes. If G is k-connected (or k-edge-connected), then each vertex of G has degree at least k. In particular, $|E| \geq \frac{nk}{2}$.*

For k-connected directed graphs, we will have $|E| \geq nk$.

Proof. Clearly the first statement implies the second, so we just need to prove the first. Fix any vertex $v_0 \in V$, and let d be the degree of v_0. Then if we remove the d edges coming out of v_0, we disconnect the graph. Hence we have an edge cut of size d. Thus $\kappa(G) \leq \lambda(G) \leq d$, i.e., $d \geq k$. □

Now we can ask if one can actually construct a k-connected graph on n vertices with $\frac{nk}{2}$ edges. Well, clearly this is impossible if nk is odd, so really the best one can ask for is a k-connected graph with $\lceil \frac{nk}{2} \rceil$ edges. When $k = 2$, we want a 2-connected graph with n edges. Here we see the cycle graph C_n is 2-connected with n edges, so this is possible for $k = 2$. For general k, such graphs were constructed by Frank Harary, and now known as Harary graphs, and denoted $H_{n,k}$.

Example 1.6.15. *We will construct $H_{2n,3}$, i.e., a 3-connected graph on $2n$ vertices with the minimum possible number of edges, $3n$. Start with a cycle graph C_{2n} on $V = \{1, 2, \ldots, 2n\}$. Now connect each vertex to its diametrically opposite pair (this is why we assume an even number of vertices, but the construction is similar for an odd number of vertices). For example, for $n = 10$ we have:*

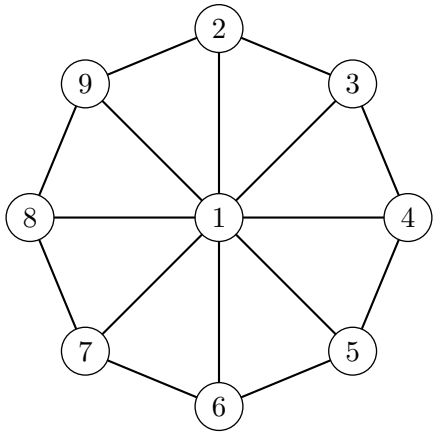


It is an exercise to check $\kappa(H_{2n,3}) = \lambda(H_{2n,3}) = 3$.

We won't bother going through the general construction of Harary graphs, though see the exercises for $k = 4$. The reason is the following: they don't have very small diameter, and therefore aren't appropriate for making efficient networks. For example, when $k = 2$, $H_{n,2} = C_n$ has diameter $\lfloor \frac{n}{2} \rfloor$. Similarly, the diameter of $H_{2n,3}$ grows linearly in n (see Exercise 1.6.10).

If we allow ourselves to increase the cost (i.e., the number of edges), we can bring ourselves down to diameter 2, by combining the star graph with a cycle graph (or if you prefer, adding a vertex to the center of $H_{2n,3}$).

Example 1.6.16. *Let $n \geq 4$. The* **wheel graph** *of order n is the undirected graph $G = (V, E)$ with vertex set $V = \{1, 2, \ldots, n\}$ obtained by taking a cycle graph on $\{2, 3, \ldots, n\}$ and connecting the vertex 1 to each of the vertices 2, 3, ..., n. Here is a picture of the wheel graph of order 9.*



The vertex 1 acts as the hub for the network.

It is easy to see the wheel graph has diameter 2 and is 3-connected (and 3-edge-connected), with $2n - 2$ edges (compared to a minimum possible $\lceil \frac{3}{2}n \rceil$ edges for 3-connected graphs).

Now how good would a wheel graph be for an actual network? Well, it depends on what exactly we want, but let's suppose 3-connectivity is enough for us. While it is somewhat robust, and efficient, and not too costly (the number of edges grows linearly in n, as opposed to quadratically), it is highly reliant on the hub (the central vertex 1) for most short paths. For many networks, hubs (nodes that are connected to a relatively large number of other nodes) are quite desirable. For organization/administrative purposes, it's often convenient to have a few central nodes (think of a flight path network). However, the wheel graph has only one hub, so it will experience all of the traffic and if there is a problem with the hub, the network, while still connected becomes very inefficient (a cycle graph with diameter $\lfloor \frac{n}{2} \rfloor$). Imagine if 90% of all US flights went through Chicago. Now imagine this during a winter snowstorm.

In practice, networks may have a number of hubs, of varying sizes. You may not be able to get everywhere in 2 steps, but maybe you can get to most places in 2 or 3 steps, and in several ways, so traffic can be rerouted. This provides some sort of compromise between our three desired qualities: efficiency, robustness and cost effectiveness.

At the other end of the spectrum, we could have highly decentralized networks, meaning an absence of hubs. It turns out these make excellent networks in practice also, provided there aren't administrative reasons for wanting a centralized network.

Definition 1.6.17. *Let G be a graph (simple or not, but undirected). We say G is* **k-regular** *if each vertex of G has degree k. We say G is* **regular** *if it is k-regular for some k.*

Note C_n, K_n and $H_{2n,3}$ are regular graphs, where as linear graphs, star graphs and wheel graphs are not. We know by Proposition 1.6.14 that a k-regular graph is at most k-connected, and the Harary graphs show we can achieve k-connected k-regular graphs when nk is even. Now we can ask, how efficient can k-regular graphs be?

Proposition 1.6.18. *Let G be a k-regular graph on n nodes with $k \geq 2$. Then* $\operatorname{diam}(G) > \log_k(n(k-1)) - 1$.

Proof. Assume $G = (V, E)$ is connected and fix a vertex $v_0 \in V$. Now run through the algorithm to find the connected component of v_0. In other words, we find the neighbors of v_0, then its neighbors' neighbors, and so on. At the first step, we find k neighbors, i.e., k elements distance 1 from v_0. At the next step, for each neighbor, we have at most k neighbors of this neighbor, so there are at most $1 + k + k^2$ vertices of distance ≤ 2. Similarly, there are at most $\frac{k^d-1}{k-1} = 1 + k + k^2 + \cdots + k^d$ vertices of distance $\leq d$ from v_0. Our algorithm cannot terminate before this number is $\geq n$, i.e., there exists some vertex v of distance d from v_0 with $\frac{k^d-1}{k-1} \geq n$, which implies there are two nodes in G which are distance $d > \log_k(n(k-1)) - 1$ apart. □

This is not the best possible lower bound for the diameter of a k-regular graph, but it is not too far off. The point is the diameter has to grow at least logarithmically in n. It turns out that if we look at a random k-regular graph, it will with very high probability be k-connected (say nk is even, otherwise k-regular graphs don't exist) and have diameter that is essentially logarithmic in n, which is much much better than the linear growth of diameter for graphs like C_n or $H_{2n,3}$. Time permitting, we will explain this in greater detail when we begin our discussion of random graphs in earnest.

In closing this section, you might notice that we don't have a single measure of how good a network is, we have several—diameter/average distance, vertex/edge connectivity, and size. We also don't have any direct measure of how "robustly efficient" a network is—meaning, if not too

many nodes go down, will the network still be efficient? This of course is very important in practice. It's reasonable to guess that if a graph is k-connected but not too many nodes or edges go down in comparison to k (e.g., $k/4$ or $\sqrt{k}$), that the graph will still be fairly efficient. This is not necessarily always true (e.g., if you have a few central hubs, and they all go down), but it's often true. When we get to spectral graph theory, we will see that looking at eigenvalues provides a way to measure how good the "network flow" is. This will give us a convenient quantity which provides a nice balance of the qualities of efficiency, robustness and robust efficiency.

Exercises

Exercise 1.6.1. *Draw all possible unlabelled trees of order 6.*

Exercise 1.6.2. *Draw all possible unlabelled trees of order 7.*

Exercise 1.6.3. *Let G_0 be a (simple undirected) graph with minimum degree ≥ 2, i.e., each vertex has degree ≥ 2. Show G_0 has a cycle of length > 2. (Hint: start at any vertex try to trace out a simple path, and show it must eventually lead to a repeated vertex.)*

Exercise 1.6.4. *Let $\mathcal{G}_{n,e}$ denote the set of (simple undirected) connected graphs of order n with e edges. Let $f_n(e)$ denote the minimum possible average distance for a graph $G \in \mathcal{G}_{n,e}$. Show that $f_n(e+1) < f_n$ for $n-1 \leq e < n(n-1)$. In other words, unlike diameter, you can always get smaller average distances by adding in more edges.*

Exercise 1.6.5. *Let G be a connected graph of order n with n edges. What is the maximum possible value for $\operatorname{diam}(G)$? Explain why, and explain how to construct graphs with this diameter.*

Exercise 1.6.6. *For $n = 4, 5, 6, 7$, do the following. Compute the number of (unlabelled) trees of a given diameter $2 \leq d \leq n-1$, and determine the probability that a given tree of order n has diameter d. Assume each tree of a given order is equally likely (this is not the case in practice if you try to randomly generate trees by a reasonable method, e.g., it is not very likely you will generate a linear graph).*

Exercise 1.6.7. *Write a Python function* `randtree(n)`*, that randomly generates a tree on n nodes as follows. Start with vertex 2. Now add vertex 2 and connect it to vertex 1. Then add vertex 3, randomly select one of vertices 1 and 2, and connect 3 to that vertex. Continue in this process until you have n nodes, and return the adjacency matrix. Then, using this function:*

(i) By generating 100 random trees of order 4, estimate the probability of getting each type of unlabelled tree of order 4. Do the same for order 5. (Hint: for $n = 4, 5$, you can determine the isomorphism type of the tree by looking, e.g., at the diameter, maximum degree, or number of leaves.)

(ii) For each $n = 5, 10, 20, 50, 100$, generate 100 random trees of order n, and estimate the expected diameter of a random tree of order n.

Exercise 1.6.8. *Let G be a simple undirected graph on n nodes. Show if $G \neq K_n$, then G has a vertex cut.*

Exercise 1.6.9. *Prove Proposition 1.6.10.*

Exercise 1.6.10. *Show that $H_{2n,3}$ from Example 1.6.15 has vertex and edge connectivities of 3. Determine the diameter.*

Exercise 1.6.11. *For $n = 5, 6, 7$ vertices, construct a graph with $2n$ edges which is 4-connected. Can you generalize this to arbitrary n?*

1.7 Weighted Graphs and Travelling Salesmen

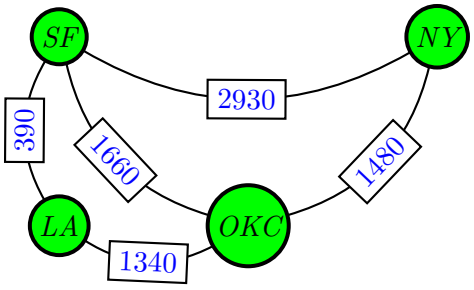
▶ Here graph means undirected graph.

Definition 1.7.1. *A* **weighted graph** $G = (V, E, w)$ *is a graph* (V, E) *together with a weight function* $w : E \to \mathbb{R}_{>0}$.

In other words, a weighted graph is a graph to which we assign each (undirected) edge a weight, which is a positive real number. (One can also consider nonpositive weights, but for our applications, we want positive weights.) The weight of an edge is typically thought of as the cost of using this edge (which might be a physical distance, or a financial cost, or a time cost, or some combination of these relevant for the problem at hand). We draw this graphically by drawing our graph as usual, and then writing the weights on or next to each edge. Much of what we have done so far can be done in the context of weighted graphs.

First, we can still represent graphs with matrices. If the vertex set is $V = \{1, 2, \dots, n\}$, put $w_{ij} = w(i, j)$ if $(i, j) \in E$ and $w_{ij} = 0$ else. Then we can represent the weighted graph $G = (V, E, w)$ with the weighted adjacency matrix $A = (w_{ij})_{ij}$.

Example 1.7.2. *Here is a weighted graph which depicts some approximate road distances among four cities: New York, Oklahoma City, San Francisco and Los Angeles.*



The weight between two cities is an approximate road distance (in miles). We did not include an edge between LA and NY because going through OKC is approximately the shortest way to get from LA to NY. The weighted adjacency matrix with respect to the vertex ordering {*NY, OKC, SF, LA*} *is*

$$A = \begin{pmatrix} 0 & 1480 & 2930 & 0 \\ 1480 & 0 & 1660 & 1340 \\ 2930 & 1660 & 0 & 390 \\ 0 & 1340 & 390 & 0 \end{pmatrix}.$$

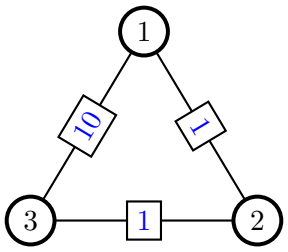
Paths are defined the same way for weighted graphs as for unweighted graphs, except now one might define the length of the path to be sum of the weights of the edges. To avoid confusion of terminology, we won't use the word length for weighted paths (so you won't just think the number of edges), but we'll use the word cost. That is, if γ is a path in G represented by a sequence of edges $(e_1, e_2, \dots, e_k)$, then the **cost** of γ is $\sum_{i=1}^{k} w(e_i)$. For instance, in our example above the cost of the path from LA to NY given by (LA, OKC, NY) is $1340 + 1480 = 2820$.

Note that if $G = (V, E, w)$ is a weighted graph where we assign each edge weight 1, the cost is the same as our definition of length for the unweighted graph (V, E). Indeed, we can view the theory of graphs as a special case of the theory of weighted graphs where all edges have weight

1. (One can define weighted directed graphs similarly, however we will only discuss the weighted undirected case here.)

Then one defines distance in the same way: $d(u, v)$ is the minimum possible cost of a path from u to v, or sets $d(u, v) = \infty$ if no such paths exist. Again, we have the triangle inequality so distance defines a metric on weighted graphs (i.e., it satisfies the primary properties you expect from a notion of distance). Diameter again is the maximum distance between two vertices. The notions of (vertex or edge) connectedness are the same for weighted graphs as for unweighted graphs, as the weights on the edges play no role in vertex or edge cuts.

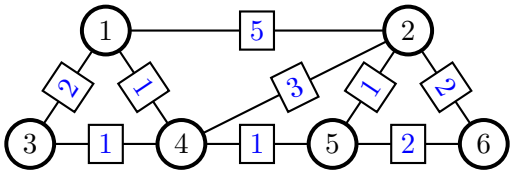
A path of minimal cost between u and v will be simple by the same argument given for unweighted graphs. One thing to be careful of is that the cheapest (i.e., lowest cost) path (or paths) from u to v may not use the fewest number of edges. For instance, consider the weighted graph



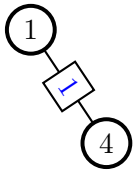
Here the shortest path from 1 to 3 goes through 2, rather than taking the direct edge from 1 to 3.

For this reason, the algorithm for computing distances that we discussed for unweighted graphs needs to be modified to work for weighted graphs. The basic idea is the same as the `spheres` function. Starting at some vertex u, we will do a breadth-first search to find the closest vertices, then the next closest, and so on. In the process, we will construct what is known as an **minimum spanning tree**. This is a subgraph of the connected component of u, which is a tree that contains only the edges needed to reach any vertex in the component of v with a shortest possible path.

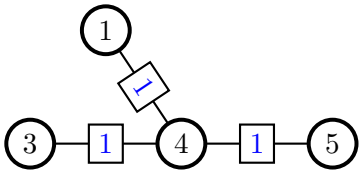
We will just explain the algorithm by way of an example. Consider the following weighted graph.



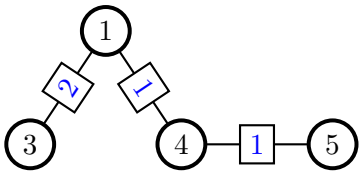
Here's how we can grow out a minimum spanning tree (MST) starting from vertex 1. Initially, the MST just contains 1. First we look at the neighbors of 1: there is 2, 3, and 4. Of these 4 is the closest (distance 1). Therefore, the shortest path from 1 to 4 must be the direct edge from 1 to 4 (any path from 1 to 4 must start from going to either 2, 3 or 4—if we go to 2 first, the path must have cost greater than 5, and if we go to 3 first, the path must have cost greater than 2). Thus we will add the edge from 1 to 4 to our MST, so it looks like this.



Now we look at the closest neighbors of 4—besides 1, there are two: 3 and 5. This gives us two paths to consider from 1 to 3, either $(1,3)$ or $(1,4,3)$, both of which have cost 2. We can choose either of these to be in our MST since they have the same cost. We also get one path from 1 to 5, namely $(1,4,5)$ which has cost 2. Again, looking at the neighbors of 1 tells us no other path to 5 can be shorter to this one, so we will add the edge $(4,5)$ to our tree. Hence at this stage, we have either the MST

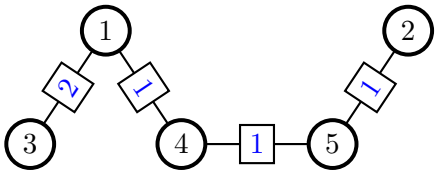


or the MST

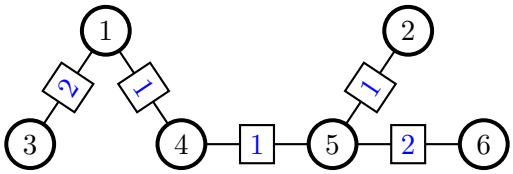


depending on which of the shortest paths we used to get from 1 to 3. For this example, let's choose the latter.

Now we can explore the new neighbors of 3 and 5. The neighbors of 3 don't get us anywhere new, so just consider the new neighbors of 5: 2 and 6. The closest one is 2, and the corresponding path cost $(1,4,5,2)$ is cost 3. Now we compare this with the other paths we've already found to 2: $(1,2)$ and $(1,4,2)$. They have costs 5 and 4, so $(1,4,5,2)$ is the shortest. Thus we will add the edge $(5,2)$ to our MST:



Now we look at the new neighbors of 2: there is just 6. We've now found 2 paths to 6: $(1,4,5,2,6)$ and $(1,4,5,6)$. The latter is shorter, so we add the edge $(5,6)$ to our MST, giving:



Since 6 has no new neighbors, and now we've included all vertices we've encountered, so this completes our minimal spanning tree. In a tree, there is a unique path between any pair of vertices (Exercise 1.7.1) so we can unambiguously read off the distance from 1 to any other vertex by looking at the path in the MST. Namely, we see $d(1,4) = 1$, $d(1,3) = d(1,5) = 2$, $d(1,2) = 3$ and $d(1,6) = 4$.

The reason this algorithm works, and works efficiently, is the following: if we have a path from u to v of minimal cost that passes through w, the part of the path going from u to w must also be of minimal cost. For example, when we were considering paths from 1 to 6, we didn't need to consider paths like $(1, 2, 6)$ or $(1, 3, 4, 5, 6)$ because at that point in our algorithm we already knew that $(1, 2)$ is not the most efficient way to get to 2 and $(1, 3, 4)$ is not the most efficient way to get to 4. This algorithm runs in $O(n^2 \log n)$ time, or to be more precise, $O(|E| \log n)$ time.

Of course, an MST will not tell you all paths of least possible cost—there are 2 from 1 to 2, but only 1 can be in the MST. Finding all paths of least possible cost is a different problem, but can be done with a simple variant of this algorithm.

With this algorithm to compute an MST for a vertex u, one can compute distance as mentioned above or the diameter. Again the algorithm to compute diameter is similar. If the graph is disconnected, it is infinity. Otherwise, do the following. Given an MST for u, one can find the vertex (or vertices) furthest from u, and record this maximum possible distance d_u. Now do this for every u and take the maximum.

Consequently, even for weighted graphs, it is not too difficult to find an optimal way to get from Point A to Point B. However, what if we want to the optimal way to visit multiple places? This might seem like a problem that should not be too much harder than finding an optimal route between two locations, but it is not so simple because trees no longer suffice to address this problem.

Definition 1.7.3. *Let G be a weighted or unweighted graph on n nodes. A* **Hamiltonian cycle** *(or* **Hamiltonian circuit***) is a cycle on G containing all n nodes.*

If $n > 1$, this is equivalent to being a cycle of cost n. Such cycles sometimes exist and sometimes do not, and deciding whether they do or not is a computationally hard problem. Technically, it is what is known as an NP-hard problem—in particular, it is believed this problem is not solvable in polynomial time. There are at most $n!$ possible cycles of cost n in a given graph (Exercise 1.7.2), and constructing a given cycle takes $O(n)$ time, so we can at solve this problem (and construct a Hamiltonian cycle when they exist) in $O(n \cdot n!)$ time. Recall Stirling's approximation says $n! \sim \sqrt{2\pi n}(n/e)^n$, which is superexponential. In the 1960's, an $O(n^2 2^n)$ algorithm was discovered by Bellman and Held–Karp (independently). This is exponential as $O(n^2 2^n) \subset O(3^n)$.

Given a weighted graph G, the **travelling salesman problem (TSP)** is to find a Hamiltonian circuit of minimum possible cost. More colloquially, suppose there are n cities you need to visit for business, but the order in which you go is not important. How can you plan a route to all the cities, and go back home, that is as short (or cheap) as possible? Hence the name travelling salesman problem. The TSP is a fundamental problem in optimization, and has applications to areas such as logistics problems, microchip design and DNA sequencing. (In DNA sequencing, the nodes are DNA fragments, and the distance measures similarity between two fragments.)

If you think about it a little, you might notice there's a slight difference between my definition of the TSP in terms of Hamiltonian circuits and my colloquiual description. Namely, a Hamiltonian circuit visits each node except the start node exactly once, whereas the least cost tour of n cities may involve taking a path through a city you've already visited. However, we can account for this with Hamiltonian cycles as follows. Let G be the weighted graph representing n cities, with an edge representing a direct physical route between 2 cities. (There may be more than one direct physical route, but for this problem it suffices to include only the one of minimal cost, which will be the weight of the edge.) Now it may be that there are two cities u and v with no edge between them (i.e., no direct physical route—e.g., no road or direct flight between the two), or it may happen that the direct route from u to v is not the most economical. In this case, we make an edge (or

replace the existing one) between u and v whose weight is the cost of the most economical path from u to v. This transforms G into a complete weighted graph G' (a weighted graph with edges between all pairs of distinct vertices, i.e., K_n with weights on the edges) where the weight of any edge (u, v) is precisely $d(u, v)$. Solving the TSP on G' really is equivalent to finding the least cost physical tour of the n cities in G, though one needs to keep track of what physical route in G is represented by each edge in G'. The construction of G' from G can be done in polynomial time, as distances can be computed in polynomial time.

Assume now G is a complete weighted graph. For complete graphs, it is easy to generate a Hamiltonian cycle—we can visit all nodes in any order we like! Consequently, we get $n!$ Hamiltonian cycles (cf. Exercise 1.7.2). Once we specify a starting vertex, there are $(n-1)!$ Hamiltonian cycles. Computing the costs of each of these cycles takes $O(n)$ time, so it is possible to solve the TSP in $O(n \cdot (n-1)!) = O(n!)$ time. Again one can do better—$O(n^2 2^n)$ run time is possible, but the TSP is also an NP-hard problem, and we expect that it cannot be solved in polynomial time—in fact, exponential time may be the best possible*.

Even though these two problems—the TSP and finding Hamiltonian circuits—are closely related and solvable in the same amount of time (essentially the same algorithm solves them both, and you can provably reduce† solving one problem to solving the other), here is one feature of TSP that is harder than the Hamiltonian cycle problem. Given a possible solution to the TSP, i.e., some Hamiltonian cycle, it is still hard to determine if this proposed solution has minimal cost. On the other hand, given a possible solution to the Hamiltonian cycle problem, it is easy to determine if it is a Hamiltonian cycle or not.

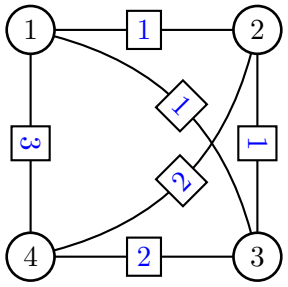
Since the TSP is hard, but of practical importance, what can we do? We have a few options: we can try to solve the TSP for special classes of graphs, we can try to find probabilistic algorithms to solve the TSP in faster time (this means, they will work some percentage of the time, but they won't give the correct solution, or at least not quickly, in some cases), or we can try to find faster *heuristic algorithms* which give approximate solutions to the TSP (i.e., find Hamiltonian cycles of relatively low cost, but not necessarily the minimum possible). Of these approaches, the latter is typically the most practical, and we'll discuss this briefly now.

The simplest algorithm you might imagine is, starting from your home vertex, travel to the neighbor of minimum distance away (or pick one if there are several). From there, again travel to the closest neighbor (or pick one if there are several) that you haven't already visited. Repeat this until you have visited all nodes, and take the unique path home. Remember, we are working with complete graphs, so this algorithm will always give some Hamiltonian cycle. This is called *the greedy algorithm*, because at each step it chooses the cheapest available. However, this may not be cheapest in the long run as the following example shows.

Consider this graph:

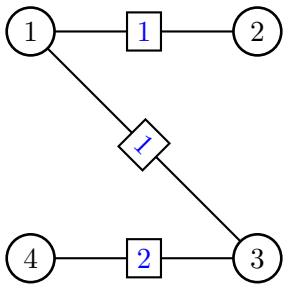
*There a subexponential/superpolynomial range of functions growing faster than polynomials but slower than exponential functions, so not being solvable in polynomial time does not mean exponential running time is the best possible. On the other hand, there are some things that take longer than exponential time (superexponential) to solve! The TSP can be solved in exponential time, but the naive $O(n!)$ algorithm is superexponential—it is essentially of the same order as $O(n^n)$.

†Meaning a polynomial-time reduction.



Let's try to find a minimal cost Hamiltonian cycle starting and ending at vertex 1. What happens if we use the greedy algorithm. At first, there are two cheapest options available—we can go to 2 or 3. Let's say we go to 2. Then there is a unique cheapest option available—going to 3. From 3, we have to go to 4, and back to 1. Thus we have taken the path $(1, 2, 3, 4, 1)$, with cost $1+1+2+3 = 7$. If we had alternatively selected the path from 1 to 3 at the first step, this would force us into the cycle $(1, 3, 2, 4, 1)$, which in this example also has cost 7. Neither of these is as cheap as possible, as both require traveling along the most expensive edge $(4, 1)$ in the graph. The best we can do for this graph is the cycle $(1, 2, 4, 3, 1)$ or the cycle $(1, 3, 4, 2, 1)$, both of which have cost 6.

Many times the greedy algorithm gives an approximate solution to the TSP that is not too far from optimal, but sometimes is can be much worse. However, it is certainly fast. It runs in $O(n^2)$ time. In 1976, Christofides discovered an polynomial-time algorithm based on a minimal spanning tree which find a Hamiltonian cycle that costs no more than 1.5 times the cost of an optimal solution. Roughly the idea is to make a minimal spanning tree, travel along a path in the tree until it ends, then jump to another path. However, this jumping paths is done in an intelligent way. In the above example, we can take for a minimal spanning tree from 1 the following:



We can start out by either picking the path $(1, 2)$ or $(1, 3, 4)$ in the MST. Then we should jump to the end of the other one, and travel back to 1. By chance, either of these choices give a Hamiltonian cycle of minimal cost, either $(1, 2, 4, 3, 1)$ or $(1, 3, 4, 2, 1)$. Of course for larger graphs, one needs to be more careful about how to jump from one path to another as there are many choices to make, and we won't typically get an optimal cycle, but at least one that's not too far off.

After 3 decades of essentially no progress along the lines of Christofides' algorithm, there have been exciting developments in this direction over the past several years, and now one can find a Hamiltonian cycle that is no more than 1.4 times the cost of an optimal one in polynomial time (Sebö–Vygen, 2012).

This is not to say that research on the TSP was stagnant from the late 70's to the mid 2000's. There has been, and still is, much work on alternative kinds of fast heuristic algorithms to approximate solutions to the TSP. However, it is very difficult to accurately assess how close the

approximate solution is to the optimal solution—and of course, it should be difficult to assess, since we don't have a good way to get a handle on the optimal solutions for comparison.

One simple alternative approach is the *Monte Carlo* method. We just start from our initial vertex, and at each stage, travel to a new vertex chosen at random, and repeat until we have to go home. We do this many times and keep track of the best solution so far. In other words, the Monte Carlo approach just tries a fairly large number of random paths, and selects the best among those. This will work well if we are in a situation where most Hamiltonian cycles are relatively cheap, and we just need to avoid certain bad paths. However, it won't work well if there are only a few good paths to find.

There are also a lot more interesting approaches, such as genetic algorithms (algorithms that evolve themselves based on past performance), simulated annealing, Tabu search and ant colony optimization (based on artificial intelligence models of ant colony behavior). In fact, the TSP is often used as a benchmark to compare different kinds of general optimization philosophies. The TSP has also crept into cognitive psychology—psychologist have studied how good humans are at solving the TSP (we're pretty good, though I'm not sure if we're as good as ants), and what algorithms most closely model how Earthlings "naturally" (approximately) solve the TSP.

Exercises

Exercise 1.7.1. *Let T be a tree, and u and v be nodes in T. Show there is a unique simple path from u to v.*

Exercise 1.7.2. *Let G be a (simple) graph of order $n > 1$. Show that G has at most $n!$ cycles of length n, with exactly $n!$ occurring in the case that G is complete.*

Exercise 1.7.3. *Let G be a complete weighted graph of order $n > 1$. Suppose you have enumerated all $n!$ Hamiltonian cycles and computed their costs and stored them in a table. Show that you can find the smallest possible cost in $O(n!)$ time. Is it possible to do better than this (just using this table)? (Note: even for the naive algorithm to solve TSP, one would not store all Hamiltonian cycles and their costs in a table as this would require superexponential space—instead, we can just keep track of the best so far.)*

Exercise 1.7.4. *Let G be the weighted complete graph on $V = \{1, 2, 3, 4\}$, where the weight of an edge (i, j) is given by $\min(i, j)$. Solve the TSP by hand for G, with initial vertex 1. (Give a minimal cost Hamiltonian cycle, and the cost.) Do the same for initial vertices 2, 3, and 4.*

Exercise 1.7.5. *Let G be the weighted complete graph on $V = \{1, 2, 3, 4, 5\}$, where the weight of an edge (i, j) is given by $\min(i, j)$. Solve the TSP by hand for G, with initial vertex 1. (Give a minimal cost Hamiltonian cycle, and the cost.)*

1.8 Further topics

Since graphs arise in many ways in many situations, there are a plethora of angles from which one can come to the study of graph theory. We've barely touched the surface of classical graph theory, and now it's time to move on. (By classical graph theory, I mean something like: the aspects in graph theory that whose study began before humans started sending things to the moon, or the parts of graph theory whose study involves mostly just combinatorics, or what I knew something

about when I was an undergrad. The important thing is I mean certain parts of graph theory that people thought about before they had to worry about really large graphs or being bothered by sociologists and economists.) In fact, a 1-semester course just on classical graph theory still isn't enough to cover all the "basics."

I'd like to give you a little overview of the classical graph theory that we're skipping in this class, but it a vast, sprawling field, sort of like a a big, complicated graph, and difficult to summarize succinctly. At a very general level, a lot of graph theory is studying invariants of graphs and seeing what they tell you—e.g., if you have two graph invariants (e.g., number of edges and vertex connectivity), does one of them imply anything about the other? Often people study these questions restricted to certain types of graphs, e.g., trees or regular graphs, where one often gets nicer answers. Another general type of question is: what conditions imply certain properties of the graph (e.g., when can we guarantee the existence of a Hamiltonian cycle?).

One large subarea is *extremal graph theory*, where one tries to determine the optimal bounds on one invariant in terms of others. We've touched on this above—if the vertex connectivity of an undirected graph on n nodes is k, then it must have at least $\lceil \frac{nk}{2} \rceil$ edges, and this bound is optimal because one can construct Harary graphs $H_{n,k}$ with vertex connectivity k and exactly $\lceil \frac{nk}{2} \rceil$ edges. Another typical question is: what is the minimum number of edges required for a graph on n nodes to have *clique* of order m, i.e., a subgraph isomorphic to K_m. (We'll say a little about cliques later.) Or: given a k-regular graph on n nodes, what is the minimum possible diameter (we gave a lower bound, but it is not optimal).

There is a large overlap of graph theory with the field of *enumerative combinatorics*. Here the typical question is to count the number of graphs (or subgraphs, or path, or cuts, etc) with certain properties. For example, count all undirected graphs (or trees, or k-regular graphs) on n vertices up to isomorphism. Sometimes one is interested in a question not originally phrased in terms of graphs, and then one interprets it in terms of certain kinds of graphs, and tries to count these kinds of graphs (or prove something about them).

Another subarea is *algebraic graph theory*, which uses linear algebra and group theory (groups are a fundamental object in algebra—a group is essentially the symmetries of some object). to study graphs. E.g., what do the eigenvalues of the adjacency matrix tell us, or what do the group of automorphisms of a graph tell us? Conversely, graphs are often used as tools to study groups. We'll look at eigenvalues in the third part of the course.

Graphs are also closely related to *finite geometries*—these are finite sets of points and lines which satisfy a set of axioms like Euclid's axioms for plane geometry. This is part of *algebraic combinatorics* and has applications to cryptography and the theory of error-correcting codes, which are important in engineering and communications.

There are many other aspects and areas of graph theory that we won't get to in this course, but let me just tell you about what is perhaps the most famous result in classical graph theory: the *four-color theorem*. Draw any map on a piece of paper. That is, draw a set of continuous curves on your paper that divide the space up into a finite number of contiguous regions. This is what we'll call a planar map. We can turn this map into a graph by making each region a vertex and connecting two vertices with an edge when the corresponding regions share a common border (not just a point). This is essentially what we did for the Königsberg bridge problem, except we used edges to denote bridges, not borders there.

A graph is called **planar**, if we can draw it in the plane $\mathbb{R}^2$ with no two edges overlapping. It is clear by construction that if you start with a planar map, the associated graph is also planar.

Definition 1.8.1. *Let S be a set of size k, which we think of as denoting k different colors. A* **k-coloring** *of a graph $G = (V, E)$ is a map $\alpha : V \to S$ such that if $(u, v) \in E$, then $\alpha(u) \neq \alpha(v)$. The minimal k such that G has a k-coloring is called the* **chromatic number** *$\chi(G)$ of G.*

In English, a k-coloring of a graph G is just a way to color the vertices of G with k distinct colors in such a way that no two adjacent vertices are the same color. If G has order n, then we can clearly color G with n different colors. The chromatic number $\chi(G)$ is just the smallest number of colors we need to color the vertices of G with the above rule.

Theorem 1.8.2 (Four-color theorem)**.** *Let G be a planar graph. Then $\chi(G) \leq 4$.*

In other words, any planar graph can be colored with at most 4 different colors. Thus we can color the regions of any map in the plane using at most 4 colors so that no two bordering regions have the same color.

This is a nice, simple-to-understand result, of course, but the reason it's so famous is because of it's history. Despite it's simplicity, it resisted proof for over 100 years, the proof was controversial at the time, and we still don't have a good way to understand *why* it is true.

The four-color theorem was originally stated in 1852, but with an incorrect proof. Many "proof" and "counterexamples" were since proposed, but were later discovered to also be incorrect. On the other hand, in 1890, Heawood provided a simple (correct) proof that any planar graph can be colored with at most 5 colors. Finally, in 1976, Appel and Haken announced a proof of the four-color theorem that is now believed to be correct. Based on Heawood's result, it suffices to show no planar map requires 5 colors. The basic idea of the proof is to assume there is a planar graph G that requires 5 colors, and use a reduction argument to yield a smaller graph that requires 5 colors. Appel and Haken, through much work, reduced the problem to considering an explicit set of 1482 cases which were checked by a computer to all be 4-colorable. At the time, the computer calculations themselves were a great achievement, which took over 1000 hours of computing time.

Proofs are deemed correct or flawed or incomplete by consensus. Typically, especially for problems of significant interest, the proofs are carefully checked by other experts by hand. However, this was the first serious example of a computer-proved theorem and doubts remained about it's validity—both general doubts about computer proofs because it could not feasibly be verified by hand and specific doubts about the actual code. People are always skeptical of new things, but it really is very hard to verify correctness of complicated computer output. First, there is the issue of guaranteeing that the machine is doing exactly what you tell it (no hardware/environment issues), then verifying the correctness of the code itself, which can easily have a minor, hard-to-find bug. (For example, there is a problem in combinatorics/coding theory known as Berkekamp's Switching Game, proposed by Berlekamp in 1960. This was "solved" by computer in 1989. I had two undergraduates work on a generalization of this in the summer of 2002 and, the night before the end of the summer program, they unexpectedly discovered, again by computer, that the original solution was wrong!)

In response to some of these doubts, Appel and Haken published a very detailed monograph of the proof of the four color theorem in 1989, including computer calculations, which was over 700 pages. This is now generally accepted, and since then other researchers have done separate computer proofs of the four-color theorem to double-check its correctness, but there is no known complete proof by hand.

The other issue with this computer proof is that it is not very enlightening—traditional proofs (at least good ones) typically do not just verify the truth of a statement, but also give us intuitive

understanding of why it is true. Reducing a problem to 1500 cases, which are checked individually, fails to give a good *reason* why it is true. This is not to take away from Appel and Haken great accomplishment—there was a very hard result, and there may be no real simple or enlightening proof of the four-color theorem.

There are innumerably many texts on graph theory and combinatorics that you can see for more information on the topics discussed on this chapter. Eventually, I may add a list of specific references here or in the introduction or at the end, but most of what we have talked about can be found on almost any introductory text on graph theory, though many books will stick to the case of undirected and possibly simple and/or unweighted graphs. (Part of the problem is, there are so many books, it's hard to choose what to put on a reference list.) One exception is the TSP, for which you should turn to a book on algorithmic graph theory, or a general book on algorithms or combinatorial optimization, for more details.

Exercises

Exercise 1.8.1. *Show K_4 is planar, but K_5 is not.*

Exercise 1.8.2. *Determine the chromatic number of the cycle graph C_n.*

Exercise 1.8.3. *Determine the chromatic number of the complete graph K_n.*

Exercise 1.8.4. *Determine the chromatic number of the star graph of order n.*

Exercise 1.8.5. *Determine the chromatic number of the wheel graph of order n.*