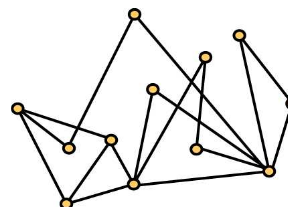
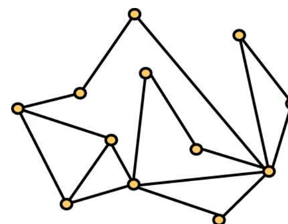
1 Graph representations

We've already seen a number of networks—both large and small—represented visually. A visual representation is terrific for smaller networks, and a well-designed layout can make a small network easy to understand at a glance. There is, in fact, an entire subfield of computer science called *graph drawing*, which is devoted to taking networks and producing good (clear, aesthetic, ...) images of the networks. But for almost all types of networks, once the number of nodes gets past a hundred or so, the pictures become so cluttered that it's almost impossible to get much useful information about of the image. (So a node is worth about ten words, I suppose: a thousand words per picture; a hundred nodes per picture.) For larger networks—e.g., Facebook or the web—we'll need another representation. And, for that matter, when we're thinking about representing data on a computer, an image is not a feasible way to store a network.

One exception to the "only small networks can be fruitfully displayed visually" claim is a class of networks called planar graphs. *A graph is called planar if it's possible to draw it on a sheet of paper in such a way that no edges cross each other on the page. For example, the graph*



has some crossing edges, but if you rearrange the layout of the nodes as follows



then there are no edge crossings. (These images are from `http://www.math.gatech.edu/~trotter/Section3-planar.htm`*.) The most famous type of planar graph is the type derived from maps: think of the countries on a map as nodes and draw an edge between two countries/nodes if those two countries share a border. (There's a minor caveat that I'll informally call the "Alaska problem": in order for the resulting graph to be guaranteed to be planar, a country must be a contiguous area.) Determining how to lay out a planar graph without edge crossings can be an interesting amusement—see* `www.planarity.net`*.*

To represent a large network on a computer, we need to find a way to store both the nodes of the graph and the edges of the graph. Depending on the graph, we may also need to store additional information about the nodes (name, salary, political views, ...) or the edges (number of lanes, length of flight, whether the two connected nodes went to school together or hooked up or don't even know each other, ...). In computer science, the topic of how to organize and store information is called *data structures* (and there are typically entire courses taken relatively early in CS curricula devoted to the topic). We'll leave most of data structures to CS 201 (which naturally you should plan to take in the spring, naturally after you take CS 111 in the winter), but in this course we're going to have to confront the particular issue of how to store a graph in a computer.

The most straightforward way to store a network on a computer is by literally storing a list of nodes and a list of edges. This is a reasonable beginning, but it conflicts with the computer scientist's natural Lamborghini-like inclinations. You'll notice that many of the natural questions that you might typically ask about a network (or even that you can very easily read off of a picture of a network) are difficult to answer quickly in when a network is stored this way. (E.g., what are all of the neighbors of Alice? Are Eve and Charlie friends?) Imagine that I gave you a file of this form containing all of the nodes and edges of Facebook, and then I asked you to tell me whether your sister is a friend of my cousin.

A miniature graph to consider as an example:

nodes
```
Alice
Charlie
Eve
Bob
Danielle
```

edges
```
Alice, Eve
Bob, Alice
Eve, Bob
```

What would you do? Sadly, answering this question would require you to go through every edge in the graph in the above representation—and that seems silly. There are tens of millions of nodes and billions of edges in Facebook, and if clicking on "all friends" in Facebook required Mark Zuckerberg to read through billions of lines of data, you sure wouldn't have time to login as many times per day as you (on average) do.

There are two representations of graphs that are standard ways to store networks on a computer that let us do things faster. Each of them is tailored to make it possible to answer one or the other of the above types of questions (What are all of u's neighbors? Is there an edge between u and v?) extremely quickly:

1. *Adjacency list.* For each node u in the network, we store a list of all of u's neighbors in the network. (To make it easier to find a particular node, we also store the nodes in sorted order.) Obviously this representation is perfectly tailored to answering the "what are all of u's neighbors?" question, because we've simply stored that information for each node in the graph. To answer the "is there an edge between u and v?" question, we need to look through the list stored at node u and check to see if v is in that list. Thus we have to look at a list of length $\mathsf{degree}(u)$ to answer this question—and $\mathsf{degree}(u)$ is typically very small (even compared to the number of nodes in the graph, let alone the number of edges in the graph).

   An adjacency list for our miniature graph:

   ```
   Alice:      Eve, Bob
   Bob:        Alice, Eve
   Charlie:    ---
   Danielle:   ---
   Eve:        Alice, Bob
   ```

2. *Adjacency matrix.* We store the graph using an n-by-n table (where n is the number of nodes in the graph), where row $\#u$ (in alphabetical order) corresponds to the neighbors of node u. A `yes` in column v indicates that an edge exists between the two nodes u and v; a `no` indicates that it does not.

   An adjacency matrix for our miniature graph:

	Alice	Bob	Charlie	Danielle	Eve
Alice	no	yes	no	no	yes
Bob	yes	no	no	no	yes
Charlie	no	no	no	no	no
Danielle	no	no	no	no	no
Eve	yes	yes	no	no	no

   Actually, since computers actually store all of their information using *bits* (data values that are either zero or one), we would instead store a table of zeroes and ones that represent whether an edge exists between node $\#i$ and node $\#j$ (stored as a 1) or does not exist (stored as a 0) between the ith node and the jth node.

   Or, more realistically:

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

   You may recognize this representation as a matrix *from your high-school algebra class, and that's why the name is used.*

   This representation is perfect for answering the question of whether there is an edge between two particular nodes: just look at the appropriate spot in the table. Finding all of the neighbors of a

particular node requires looking at one entire row of the table. It turns out that there are some interesting questions about graphs that can be answered using this matrix representation, and we may return to this soon. (In the meantime, try to remind yourself of how matrix multiplication works—look at Wikipedia if you want a refresher.)

2 Connectivity in Graphs

One of the most basic questions that one can ask about a network—and one that seems agonizingly simple when you look at a picture of a small network—is whether there is a path connecting some node s to some node t. Can you get from Missoula to Madison by car? Is there some chain of friends that connects me to Phil Collins? If the coconut tree goes extinct, will the African swallow be affected? (I.e., does the African swallow eat something that eats something that eats something ... that eats coconuts?) Or, even more simply, is a given network *connected*? (I.e., are all pairs of nodes in this network connected by some path?) These questions are pretty simple to answer for small networks presented visually—just look!—but it's much trickier to figure out how you might answer these questions for a network stored as an adjacency list or adjacency matrix.

Here's a network in adjacency-list form:

```
0:    3, 7
1:    9, 2, 5
2:    1, 10
3:    0, 7, 1
4:    10, 7
5:    1
6:    7, 11
7:    0, 4, 6
8:    11, 12
9:    1
10:   2, 4
11:   6, 8
12:   8
```

Is there a path from node 0 to node 12?

Before we launch into a full-blooded explanation of how to solve this problem, it's worth pondering how you might solve this problem, say with a social network represented via adjacency list. Imagine that I ask you whether there's a way to get from you to me in a large social network. You can get an answer to the question "who are u's friends?" by asking u for a list of his or her neighbors. This description is analogous to a network stored by adjacency list: you have access to lists of neighbors for any node in the network. And you want to know if a path exists between two specified nodes in the network. What would you do? Think about the example on the right before you read on!

I think that there are a number of different approaches that you might consider. You might try wandering aimlessly through the network (pick one of your friends, see if it's me, pick one of their friends, see if it's me, ...)—and, in fact, there's a whole field of study devoted to *random walks* which have exactly this style of exploration. (We'll come back to random walks later in the term: it turns out that the way that Google figures out which

pages to display in response to a web search has a lot to do with a slightly more complicated random walk.)

But let's try to do something more systematic. Let's say that you want to figure out whether there's a path that connects node s to node t in the network. Here's the idea. Let's keep a list L that represents all of the nodes that we can reach from a node s in the network. To start with, the list L will contain just one node—namely the node s itself. Now we'll figure out what other nodes are directly connected to nodes in the list L, and add those nodes to the list L. After all, if you can reach a node from s, then you can also reach all of that node's neighbors from s (via that node). But now we've found some more nodes that can be reached from s, which means that we can also reach any nodes that are directly connected to *them* from s. So we'll repeat that process with the updated list L: we'll figure out what other nodes are directly connected to nodes in the expanded list L, and expand L again to include all of those nodes too. And we'll do it again, and again, and again. Once we've figured out *all* of the other nodes that we can reach from s, we'll check whether t is one of the nodes that we can reach.

Let's make this a little more concrete:

To determine whether there is a path from s to t:

1. Let L be a list containing only the node s.
2. For every node u in the list L, look at the list of neighbors of u in the network. Make a new list L' that contains every node that is a neighbor of any node u in L. Now add to L every node in L' that's not already in L. Unless we didn't make any changes to list L, repeat Step 2 again.
3. If t is in the list L, then the answer is yes—there is a path from s to t. If t is not in L, then the answer is no—there is no path from s to t.

The above is an example of an *algorithm*—a step-by-step recipe for solving a problem. The key point is that this is a completely general recipe: for *any* network and nodes s and t, you can use this recipe to figure out whether there's a path from s to t. (Similarly, you can use the change-making algorithm that we talked about in the first week of the term to make change for any desired amount of money.)

This algorithm is known as *breadth-first search* (or, for short, *BFS*), for reasons that ought to be somewhat intuitive. If you imagine a picture of the network, and

The description of the algorithm on the left is written in what's called pseudocode. *It's a systematic description of the steps of the algorithm, but it's written in English, not in some programming language that could be directly interpreted by a computer. The key idea in writing an algorithm is to communicate with humans. In this class we'll be thinking at this algorithmic level; in other classes (e.g., CS 111, which you'll take in the winter), you'll learn how to translate these descriptions into something that can be understood directly by a computer. For your edification, here's what this algorithm might look like in Python, the programming language used in CS 111:*

```
L = [s]
oldL = []

while not L == oldL:
   oldL = L
   Lprime = []
   for u in L:
      Lprime = Lprime + neighbors[u]
   for v in Lprime:
      if v not in L:
         L = L + [v]

if t in L:
   print "there is a path from s to t"
else:
   print "there is no path from s to t"
```

This code is a very direct translation of the pseudocode on the left, with one addition: in order to figure out whether we made any changes to L when we went through Step 2, this code "remembers" what L looked like before we went through Step 2 this time around, and if the 'before' and 'after' lists are the same, then it stops repeating Step 2.

you watch the nodes that are in the list L, you can think of an ever-expanding frontier around s. The first time through Step 2, the list L contains all neighbors of s. The second time through, it contains all neighbors of s and all neighbors of neighbors of s. The third time through, it contains all neighbors of s and all neighbors of neighbors of s and all neighbors of neighbors of neighbors of s. And so forth. Every step of this process takes the full breadth of the frontier and expands it out by one more "layer" in the network. You can think about BFS using the same analogy by which I was introduced to the Renaissance in my World Cultures class in tenth grade: think of throwing a pebble onto the graph at the node s, and watching the ripples expanding out from s. (According to Doc Wilkerson, anyway, in the Renaissance s = Florence.)

`http://love2all.org/WaterMenu.htm`

It's pretty obvious that if BFS finds a path from s to t, then a path from s to t exists (after all, BFS found one). If you think it through a bit more, you should be able to convince yourself that if BFS doesn't find a path from s to t, then s and t are not connected. (By the time BFS has stopped finding new nodes, it must have collected every node in s's connected component in the list L.) We won't try to go through a detailed proof of this (we'll leave that for CS 202, which you'll take next fall), but it should be believable: the algorithm only stopped when Manifest Destiny was achieved—in other words, when there were no new nodes that were reachable from any of the nodes that had already been discovered.