1 Graphs and networks

A *graph* or a *network* is a collection of vertices (or nodes or sites or actors) joined by edges (or links or connections or bonds or ties). We'll use the terms interchangeably, although depending on who you're talking to, some people have strong preferences on terminology. Before we dive into graph algorithms, here's a little historical perspective.

1.1 A little perspective

Graphs have been studied as mathematical objects for hundreds of years, and graph theory is a deep mathematical field whose history is usually traced back to Leonhard Euler's 1736 solution to the famous Seven Bridges of Königsberg problem (see Section 4.1 below).[1] Prior to the second half of the 20th century, most models of graphs were extremely general and most empirical networks were very small. Modern computers have changed all that, making it relatively easy to measure, store, draw and analyze the structure of extremely large graphs. The study of graph-like structures in the real world has led to the investigation of more structured mathematical objects, and many of the models and analytic tools today are highly sophisticated and rely on the tools of probability theory.

Any object that can be represented as a set of discrete entities with pairwise[2] interactions can modeled as a graph. For instance:

network	vertex	edge
Internet	computer	network protocol interaction
World Wide Web	web page	hyperlink
power grid	generating station or substation	transmission line
friendship network	person	friendship
metabolic network	metabolite	metabolic reaction
gene regulatory network	gene	regulatory effect
neural network	neuron	synapse
food web	species	predation or resource transfer

In some cases, the network representation is a close approximation of the underlying system's structure. In others, however, it's a stretch. For instance, in molecular signaling networks, some signals are conglomerates of several proteins, each of which can have its own independent signaling role. A network representation here would be a poor model because proteins can interact with other proteins either individually or in groups, and it's difficult to represent these different behaviors

[1] For additional study on graph theory, I recommend Douglas B. West's *Introduction to Graph Theory* (2nd ed.) from Prentice Hall.

[2] Higher-order interactions can also be defined, and networks of these are called *hypergraphs*. Examples include collaboration networks like actors appearing in a film, scientists coauthoring a paper, etc.

within a simple network.[3] In general, it's important to think carefully about how well a network representation captures the important underlying structure of a particular system, and how we might be misled if that representation is not very good. If a graph is a poor representation, for instance, the shortest path between some pair of vertices i and j might be useless for the problem we care about, even if we can find it quickly.

1.2 Types of graphs

There are many types of graphs, for example, multigraphs, simple graphs, hypergraphs, graphs with self-loops, bipartite graphs, acyclic graphs, weighted graphs, etc. We'll go through most of these and point out their differences. Figure 1 below shows some of them schematically.

A graph in which a pair of nodes i, j can have multiple, distinct connections is called a *multigraph*, e.g., two cities can be joined by multiple roads and two neurons can interact through multiple synapses. Sometimes, it is convenient to collapse the different multi-edges between two nodes into a single edge annotated by a *weight* w_{ij} equal to the count of those multi-edges, or some function of the weights on the multi-edges. Weights can also be used to represent the strength or capacity or frequency of the interaction, and are generally real-valued. A graph with weighted edges is called a *weighted graph*. Nodes can also be annotated, e.g., with a vector of personal attributes (as on Facebook).

If the connection (i, i) is allowed, such an edge is called a *self-loop*. A graph with no self-loops and no multi-edges is called a *simple graph*. A *directed graph* is one in which connections can be asymmetric, i.e., node i can connect to node j without the reverse being true. The World Wide Web is an example of a directed network. An *acyclic graph* is a special kind of directed graph that contains no cycles, i.e., for all choices of i, j, if there exists a path $i \to \cdots \to j$ then there does not exist a path in the reverse direction $j \to \cdots \to i$. In the undirected case, exactly one such reverse path is allowed and it must be the same as the forward path. For example, a (undirected) tree is a simple kind of acyclic graph. Citation networks should be examples of acyclic directed networks, but are often not in practice. Citation networks are also examples of dynamic or *temporal graphs*, where the set of edges changes over time. *Spatial graphs* have vertices that are embedded in some kind of metric space. And, graphs can almost any combination of these characteristics, e.g., spatial, temporal, directed and weighted.

Sometimes, we wish to represent the interactions between distinct types of things within a graph structure, e.g., actors and the films they're in, scientists and the papers the coauthor, etc. These are called k-partite graphs, where nodes of type μ only connect to nodes of type $\nu \neq \mu$. When $k = 2$, as in actors and films, they're called bipartite graphs. We can always convert a k-partite graph into a

[3] That being said, such a network could be represented using a mixed hypergraph, in which some edges are defined pairwise, while others are hyperedges of different orders, defined as interactions among sets of nodes.

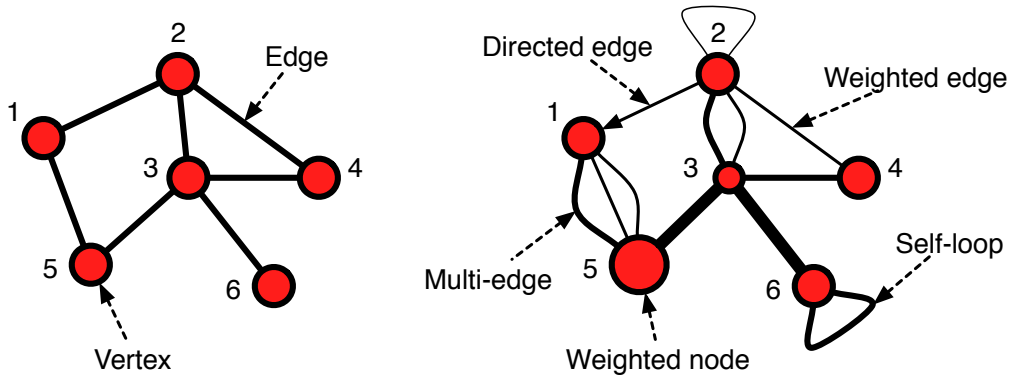

Figure 1: Examples of different types of edge and node structures. The left-hand graph is an unweighted, undirected simple graph. The right-hand graph is more exotic.

simple graph, where every node is of the same type, by computing the *one-mode projection*. In this construction, all the nodes of a single type are retained and pairs of these i, j are connected in the projection if and only if they had a common neighbor in the original graph. For example, we link actors together if they appeared in the same film. One consequence of a one-mode projection is the construction of cliques, i.e., a subgraph of size ℓ in which every pair of nodes is connected. For instance, all the actors in a given movie will be joined in a clique in the one-mode actor projection.

1.3 Graph notation

Before we discuss representations of graphs, let's review some notational conventions.

If we say $G = (V, E)$, we mean that G is a graph composed of the set of vertices V and the set of edges E, where each edge $e \in E$ is defined as $e = (i, j)$ for $i, j \in V$.

For convenience, sometimes we let the number of vertices $|V| = n$ and the number of edges $|E| = m$. In asymptotic notation, sometimes we drop the cardinality bars and simply say $O(V)$ as shorthand for $O(|V|)$ or $O(n)$. For pairwise interactions, $E = O(V^2)$ and thus an algorithm that runs in $O(V + E) = O(V^2)$ in the worst case.

If $|E| = \Theta(V^2)$, then we say that the network is *dense*, while a network is *sparse* if $|E| = \Theta(V)$. For the time and space requirements of graph algorithms, we often distinguish between these two cases. (At home exercise: write down mathematical definitions of the different types of graphs described above.)

1.4 Representations of graphs

There are two common ways (and a third less common way) to represent a graph, and which you use can have a large impact on the space and time requirements of algorithms.

1.4.1 The adjacency matrix

The first is an *adjacency matrix* A, where

$$A_{ij} = \begin{cases} w_{ij} & \text{if } i \text{ and } j \text{ are connected} \\ 0 & \text{otherwise} \end{cases}$$

If A represents an *unweighted graph*, then $w_{ij} = 1$ for all i, j.

Adjacency matrices are often used in mathematical expressions, e.g., when describing what a graph algorithm does but sometimes also in a graph algorithm itself. The disadvantage of adjacency matrices is that they take $\Theta(V^2)$ memory, regardless of the number of edges. If the network is sparse, this is wasteful.

Here's the adjacency matrix for the graph in Fig. 1a:

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

The fact that the diagonal is all zeros and the non-zero entries are binary indicates that this is a simple graph. The fact that the matrix is symmetric across the diagonal indicates that it is undirected; the upper triangle represents connections (i, j) for $i > j$ and the lower triangle represents (j, i) for $j > i$.

1.4.2 The adjacency list

The second representation is an *adjacency list*, which stores a vector of length n, one for each vertex in the graph. The ith element of this array points to a linked list containing all the vertices j for which there is an edge originating at i and terminating at j. Notably, the list of adjacencies for some i, denoted $Adj[i]$, is not necessarily ordered, and undirected edges are stored as a pair of directed edges. That is, if an undirected $(i, j) \in E$, then both $j \in Adj[i]$ and $i \in Adj[j]$.

In this way, the adjacency list stores only the non-zero elements of the adjacency matrix. For sparse graphs, in which $m = O(n)$, this is a huge savings in space over the $O(n^2)$ space for the

adjacency matrix. The time to test whether some undirected $(i, j) \in E$ is simply the time to scan either $Adj[i]$ for j or $Adj[j]$ for i; this may still be a slow operation if the degree of that vertex is large. Instead of using a list to store the adjacencies, a more efficient approach would store a given vertex's adjacency in a data structure that supports fast find operations, e.g., a balanced binary tree structure like a red-black tree. (This is typically how efficient *sparse matrix* data structures store their contents.)

Here's an adjacency list representation of Fig. 1a, where we store each vertex's adjacencies in an unordered linked list:

$$Adj = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{bmatrix} \begin{array}{l} : \ 2 \to 5 \to \emptyset \\ : \ 3 \to 1 \to 4 \to \emptyset \\ : \ 5 \to 6 \to 4 \to 2 \to \emptyset \\ : \ 2 \to 3 \to \emptyset \\ : \ 3 \to 1 \to \emptyset \\ : \ 3 \to \emptyset \end{array}$$

1.4.3 The edge list

A third, but rarely used, representation is an *edge list*, which stores only the non-zero elements of the adjacency matrix. Here's an undirected edge list representation of Fig. 1a:

$$\{(1,2),(1,5),(2,3),(2,4),(3,5),(3,6)\},$$

where the absence of a weight w_{ij} in each tuple implies that all edges have unit weight. In this case, the presence of an edge (i, j) implies an undirected tie between i, j, but this is not obvious from the list. Edge lists are sometimes used to store a network in a file, but they present no genuine space savings over the adjacency list (do you see why?).

2 Basic graph analysis

For any graph $G = (V, E)$, there are any number of functions we could define $f(G)$ that return something informative about the structure of a graph. Here, we'll cover some of the more conventional ones.[4]

[4] "Networks" are currently a hot field of research, both from the perspective of investigating and explaining patterns in the structure or dynamics of networks of all kinds (e.g., social, biological or technological) and from the perspective of developing algorithms to analyze their structure or predict things about their evolution. Many of these algorithms are statistical algorithms, in the sense that they engage directly with data and use probability in various ways. In some ways, Google and Facebook both are fundamentally "networks" companies. If you're interested in this stuff, it's one of my research areas, so I'd be happy to provide pointers into the literature.

2.1 Degrees

In an undirected graph, the *degree* of a node k_i is a count of the number of connections terminating (equivalently: originating) at that node. (In directed graphs, we must distinguish between the in-degree k_i^{in} and out-degree k_i^{out}.) Using the adjacency matrix, the degree of vertex i is defined as

$$k_i = \sum_{j=1}^{n} A_{ij} = \sum_{j=1}^{n} A_{ji} \ , \tag{1}$$

which is equivalent to the ith column (or row) sum of the adjacency matrix A. (In the directed case, the second equality holds iff $k_i^{\text{in}} = k_i^{\text{out}}$.) If A represents a weighted network, this sum is called the node *strength* or weighted degree.

Each edge in an undirected network contributes twice to some degree, and so the sum of all degrees in a network must be equal to twice the total number of edges in a network m:

$$m = \frac{1}{2}\sum_{i=1}^{n} k_i = \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} A_{ij} = \sum_{i=1}^{n}\sum_{j=i}^{n} A_{ij} \ . \tag{2}$$

The mean degree of a node $\langle k \rangle$ in the network is

$$\langle k \rangle = \frac{1}{n}\sum_{i=1}^{n} k_i = \frac{2m}{n} \ . \tag{3}$$

If we divide the mean degree by its maximum value

$$\rho = \frac{2m}{\binom{n}{2}} = \frac{2m}{n(n-1)} = \frac{\langle k \rangle}{n-1} \ , \tag{4}$$

we have a quantity ρ that is sometimes called the network's *connectance* or *density*.[5]

One of the most common uses of the degree measure is in tabulating the *degree distribution* for a network $\Pr(k)$, which gives the distribution of a vertex selected uniformly at random. (This is distinct but related to the *degree sequence*, which is simply a list of the degrees of every node in a graph.) The degree distribution for Fig. 1a is

k	$\Pr(k)$
1	1/6
2	3/6
3	1/6
4	1/6

[5]Sometimes, connectance is defined as $\langle k \rangle / n$, which is asymptotically equivalent to $\langle k \rangle / (n-1)$.

where $\Pr(k) = 0$ for all other values of k.

In studies of empirical networks, the degree distribution is often used as a clue to determine what kinds of generative models to consider as explanations of the observed structural patterns. Generally, empirical social, biological and technological networks all exhibit right-skewed degree distributions, with a few nodes having very large degrees, many nodes having intermediate degrees, and a large number having small degrees.

2.2 Shortest paths, diameters and components

A *path* in a network is a sequence of vertices $x \to y \to \cdots \to z$ such that each consecutive pair of vertices $i \to j$ is connected by an edge (i, j) in the graph. A *shortest path*, which is also called a *geodesic path* (from geometry), is the shortest of all possible paths between two vertices. Shortest paths are examples of "self-avoiding" paths, meaning that they do not intersect themselves. The length of the longest of these paths is called the *diameter* of a graph, which should evoke the notion of a volume in a metric space.

Given a vertex x, a second vertex z is said to be *reachable* from x if G contains a path $x \to y \to \cdots \to z$. A *component* is a set of vertices that are pairwise reachable. If a graph is directed, that x is reachable from y does not imply that y is reachable from x. A *strongly connected component* is a set of vertices which are pairwise reachable, while a *weakly connected component* is a set of vertices which are pairwise reachable in at least one direction.

We'll spend a fair amount of time looking at graphs algorithms related to these topics.

3 Search trees on graphs

An enormous number of operations on graphs can be reduced to some kind of variation on a *search tree* or a function that calls a search tree as a sub-routine. Two of the most well-known graph algorithms are the *breadth-first search* (BFS) and *depth-first search* (DFS) algorithms. These are closely related algorithms: both take as input a graph $G = (V, E)$ and a source vertex $s \in V$ and proceed by exploring the structure of the graph one edge at a time. Their output can be the path $s \to t \to \cdots \to z$ to some vertex $z \in V$, the set of all such paths to all vertices in $V - s$, which we call the search tree T, or some property of one or the other of these.

3.1 The basics

All search-tree algorithms use a *queue* as an underlying data structure,[6] but the way they interact with the queue varies. The basic structure of all search tree algorithms looks like this

```
Search-Tree(G,s) {
  for i = 1 to n {
     v[i] = 0                  % mark each vertex is unvisited
     p[i] = NULL               % initialize predecessor array
     d[i] = INF                % distance from s to i
   }
  enqueue(Q,s)                 % add s to the queue Q
  d[s] = 0
  while Q not empty {
     x = dequeue(Q)
     if v[x]==0                % if x unvisited
        for each neighbor y of x {
           if v[y]==0 {        % if y unvisited
              enqueue(Q,y)     %   add y to queue
              p[y] = x         %   x is y's predecessor in T
              d[y] = d[x] + 1  %   record distance to y
           }
        }
        v[x] = 1               % mark x as visited
}}}
```

When `Search-Tree` terminates, all nodes reachable from s have been marked. The contents of the array `p` are the search tree T and the value `d[i]` gives the distance in the search tree from s to i.

3.1.1 Running time

We now analyze this algorithm's asymptotic running time.

Assume (i) that G is connected, i.e., for all i, j, there exists a path $i \to \cdots \to j$, and (ii) that it takes $O(1)$ time to add or remove a vertex to Q and $O(k_x)$ time to get a list of the k_x neighbors of a vertex x (that is, $O(1)$ each neighbor). How long does `Search-Tree` take?

[6] If you've forgotten what a queue is, it is an ADT that supports only the operations `add` and `remove`—and, search is not supported as part of adding and removing—and is often implemented using a linked list where we keep a pointer to both the first and last items in the list. If we both add and remove items only at the front of the list, the queue is a "first-in-last-out" or FILO queue, which is equivalent to a *stack*. If we add items to the front and remove them from the end (or vice versa), it is a "first-in-first-out" or FIFO queue, which is equivalent to a *pipe*.

When the function initializes, it takes $\Theta(n)$ time to mark each vertex as unvisited. The running time of the remainder of `Search-Tree` depends only on the number of items placed into the queue. Note that only when we remove an unmarked vertex x from Q, do we mark a vertex and add new vertices to Q. Once a vertex is marked, it is never marked again; thus, we will hit the "add vertices" lines at most n times. At worst, a vertex x will be added to the queue k_x times, once each time we visit (and mark) one of its neighbors. Thus, the number of items we add to the queue is $\sum_{x=1}^{n} k_x = 2m = \Theta(E)$. Ergo, the running time is $O(V + E)$.

At home questions:
1. Currently, we add all the neighbors of some x to the queue, even if thy have already been visited. Does the asymptotic behavior change if instead we only enqueue a neighbor y if $v[y] = 0$?
2. How much space does the function use?

3.1.2 Breadth vs. depth

Both breadth-first and depth-first search algorithms are special cases of our `Search-Tree` algorithm, and thus they have the same asymptotic behavior as `Search-Tree`.

The difference is simply in the way we add and remove vertices to the queue: in BFS, we use it as a FILO queue, adding vertices to the end of the Q and removing them from the front; in DFS, we use it as a FIFO queue, both adding and removing vertices from the same side of the Q.[7]

Although their running times are the same, BFS and DFS produce very different output trees: BFS trees are very "bushy", growing such that all vertices at a (geodesic) distance ℓ from the source vertex s are marked in a single consecutive sequence before any vertices at a distance $\ell + 1$ are marked. Thus, the tree contained in the array `p` is the union of all shortest paths from s to all vertices in V that are reachable from s and `d` contains the length of those shortest paths. In contrast, DFS always pushes "deeper" into the graph, producing long and "stringy" trees, backtracking only when it can discover no new (unmarked) vertices.[8]

Figure 2 gives an illustrative example of how BFS and DFS operate, on a simple graph with six vertices, where we assume the order of neighbors in any particular neighbor list is alpha-numeric.

3.1.3 From trees to forests

Suppose G is not a single connected component, but rather a collection of components? How could we write a new function that calls `Search-Tree` (or rather, a slight variation of it) as a subroutine

[7] We could also define a "random-first" search (RFS) algorithm, in which we add items to the Q in any way we like, and we remove them by choosing an item uniformly at random from within Q.

[8] DFS is a general backtracking search algorithm and this is a fundamental search algorithm, finding applications in robot motion planning, constraint satisfaction, Sudoku puzzle solving, etc.

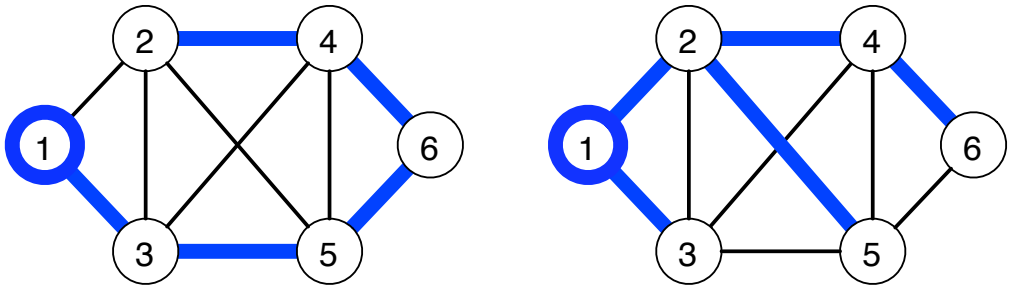

Figure 2: The left-hand figure shows a DFS while the right-hand figure shows a BFS; the bold circled vertex 1 is the source.

to explore all the components? First, we'd need to modify `Search-Tree` slightly, to move the initialization routines up a level in the program so that they are only executed once. And, because distance is meaningless between disconnected components, we can omit that piece.

The key insight in generalizing `Search-Tree` to `Search-Forest` is that the vector v contains a list of all vertices we have visited. Each time `Search-Tree` exists, all vertices in the component we just explored will now have their entries in v marked. To find a new, unvisited component, we simply need to scan through v looking for an unmarked entry, which can serve as the root for a new search tree.

```
Search-Forest(G) {
  for i = 1 to n {
     v[i] = 0                  % mark each vertex is unvisited
     p[i] = NULL               % initialize predecessor array
   }
   for i = 1 to n {
      if v[i] == 0 { Search-Tree(G,i) }
}}
```

When `Search-Forest` terminates, `p` contains a search *forest*, whose structure depends on the order in which nodes are enqueued and dequeued in the `Search-Tree` function, and all nodes are marked. Note that each time `Search-Tree(G,i)` is called, i is the root of a new tree.

How can we measure the diameter of a graph using these algorithms?

4 An aside

4.1 Eulerian Paths and the Seven Bridges of Königsberg

The classic example of graph theory is the Seven Bridges of Königsberg problem, and we already know everything we need to solve it. The story goes that the Königsberg (now Kaliningrad, Russia) aristocracy enjoyed the puzzle of trying to find a path through downtown Königsberg that would cross each of the seven bridges of the Pregel river exactly once. Euler modeled this problem using a graph and proved that no such path existed; see Figure 3.[9] For his solution, the problem of finding a path through any graph that crosses each edge exactly once is called an *Eulerian path*. If the path must begin and end at the same vertex, then it is called an *Eularian tour*.

Figure 3: The Seven Bridges of Königsberg and their graph representation (images from Wikipedia).

Lemma 1: No Eulerian path exists if an odd number of vertices have odd degrees.

Proof: Suppose that a graph G has an odd number of vertices with odd degrees and assume some path $\sigma = v_1 \to v_2 \to \cdots \to v_j$ is an Eulerian path. Note that each vertex in the path has degree 2 except for the first and last vertices, which have degree 1, and thus, the total degree of the path is an even number. For σ to be Eulerian, it must cross each edge in G exactly once, thus, the total degree of the path must equal the total degree of the graph. But, the sum of an odd number of odd numbers is itself an odd number, and thus there must be at least one edge not in σ. □

Here's a stronger claim.

Lemma 2: No Eulerian path exists if more than two vertices have odd degrees.

Proof: Same set up as above. Note that for σ to cover all the edges of some vertex, it must enter and leave the vertex an even number of times, or else that vertex is either the first or last vertex

[9] Trivia: since 1736, two of the seven bridges have been removed and there now exists an Eulerian Path.

in the path. Thus, σ can only cover at most two vertices with odd degrees and σ cannot be an Eulerian path. □

If a graph is stored as an adjacency matrix, how long does it take to identify whether it contains an Eulerian path? What if the graph is stored as an adjacency list? What conditions on degrees must be fulfilled for a graph to contain an Eulerian tour? (At-home exercise: if we have some graph G that satisfies the conditions for the existence of an Eulerian path, that doesn't mean we know how to find it easily; can you think of an algorithm that takes only $O(V + E)$ time to do so? Hint: it's a greedy algorithm, and not a variation of the `Search-Tree` algorithm.)

5 For Next Time

1. All-Pairs-Shortest-Paths
2. Read Chapters 24 and 25