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 will use the terms interchangeably, although depending on who you are talking to, some people have strong preferences on terminology. Before we dive into graph algorithms, here's a little 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

Acknowledgement: Adopted from Prof. Aaron Clauset's lecture notes.

[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 Graph notation

To define a graph, we say $G = (V, E)$, which means that the graph G is composed of a set of vertices V and a set of edges E, where each edge $e \in E$ is defined as $e = (i, j)$ for $i, j \in V$. If each edge is unique, then the maximum number of edges is $|E| \leq |V \times V|$.

For convenience, we often call the number of vertices $|V| = n$ and the number of edges $|E| = m$. In asymptotic notation, we often 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.

1.3 Types of graphs

There are many types of graphs, for example, multigraphs, simple graphs, 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 *multigraph* is a graph in which a pair of nodes i, j can have multiple, distinct connections, e.g., two cities can be joined by multiple roads and two neurons can interact through multiple synapses. That is, the edge set E can contain a particular edge (i, j) more than once. If this is true, then the size of the edge set $|E|$ can be larger than the number of unique pairs $|V \times V|$.

A *simple graph* is a graph with no self-loops and no multi-edges. A *self-loop* is a connection (i, i) for $i \in V$. Formally, we define a *simple graph* as a graph $G = (V, E)$ such that for all i the edge $(i, i) \notin E$ (no self-loops) *and* every edge $(i, j) \in E$ is unique (no multi-edges).

A *weighted graph* is one in which an edge (i, j) is annotated with a scalar *weight* w_{ij} or by a weight function $w(e)$ for $e = (i, j) \in E$. Weights can be any value, but are typically either reals or inte-

[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.

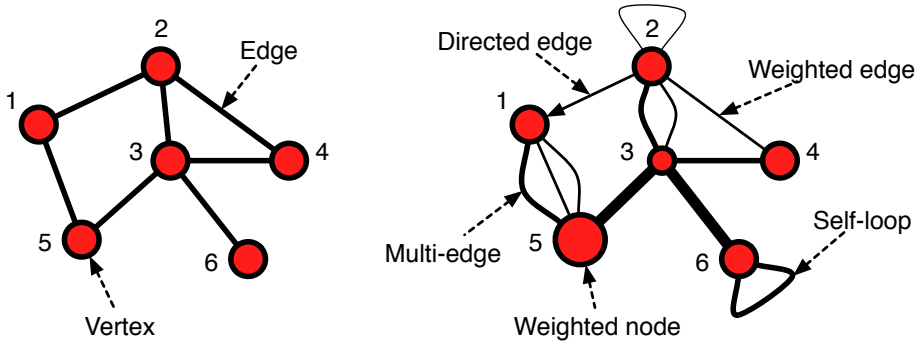

Figure 1: Examples of different types of edge and node structures. The left-hand graph is a *simple graph*. The right-hand graph is more complicated.

gers. If a weight counts something, then it must be a natural number. A simple transformation of a graph is to convert a multigraph $G' = (V, E')$ into a weighted simple graph $G = (V, E)$, in which an edge weight w_{ij} counts the number of times a connection $(i, j) \in E'$. Note that the vertex set is the same in both cases. Edge weights can also represent things like the strength or capacity or frequency of the interaction.

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. That is, if $(i, j) \in E$, then (j, i) is not necessarily also in E. 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.

A *tree* is simply an undirected acyclic graph. (Do you see why?)

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.

The formal definition of a *bipartite graph* is a graph $G = (V, E)$ for which there exists some *bipartition* of V into groups $L, R \subset V$, such that $L \cup R = V$ and $L \cap R = \emptyset$, and for every $(i, j) \in E$, either $i \in L$ and $j \in R$ or vice versa. More succinctly, bipartite graphs are all graphs that contain no odd-length cycles. Do you see why?

1.4 Representations of graphs

There are two common ways (and a third less common way) to represent a graph, and which one 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. (Think about how large an adjacency matrix you would need in order to represent the Facebook friendship graph!)

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, in a simple list. That is, an edge list simply lists the set of edges in E. Note, however, that it does not list the vertex information—that information, i.e., the names of the vertices, has to be recovered from the edge list by tabulating which indices i, j are used. Moreover, when we write down an edge list, we can use arbitrary labels to represent vertex names, and so the first step in processing an edge list is nearly always first determining the set of vertex labels it contains (typically using a dictionary ADT as a *set* data structure).

Here is an 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 (a.k.a., they are unweighted). In this representation, it is not obvious that the edge list represents an undirected graph, so that information must either be specified elsewhere or assumed.

Edge lists are sometimes used to store a network in a file, but they present no asymptotic space savings over the adjacency list (do you see why?).

2 Simple graph calculations

Many algorithms questions use graphs as a data structure to represent a problem's structure, meaning that to arrive at the correct solution, we must perform some kind of operation or calculation using the graph. Before we learn more complicated graph algorithms, we'll start with a few simple calculations that use the adjacency matrix representation of a graph.[4]

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.[5] 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.[6]

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:

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

[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.

[5] In a directed graph, there are two types of edges, ones that start at a vertex i ("out" edges) and ones that terminate at i ("in" edges), and so we must distinguish between the in-degree k_i^{in} and out-degree k_i^{out}.

[6] In the directed case, the second equality holds iff $k_i^{\text{in}} = k_i^{\text{out}}$.

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

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

As an at-home exercise, write pseudocode for calculating the degrees k_i, the number of edges m, and the mean degree $\langle k \rangle$ using the adjacency list representation.

There are, of course, many other things we could calculate about a graph's structure, and many of these are of specific interest in areas like social network analysis or algorithms for the World Wide Web. We may cover some of these much later in the class.[7]

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.

[7] 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	1	2	3	4
$\Pr(k)$	1/6	3/6	1/6	1/6

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, and this pattern can both facilitate algorithm development and the use of graph algorithms to make predictions in specific systems.

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.

The two most well-known graph algorithms are the *breadth-first search* (BFS) and *depth-first search* (DFS) algorithms. These are closely related: 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,[8] 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 {
     state[i] = NEW                % mark each vertex is unvisited
  }

  Q = newQueue(s)              % make a queue, containing s

  while Q not empty {          % while not yet done
     x = dequeue(Q)            % get a vertex x out of Q
     if state[x] == NEW {          % if x is unvisited
        state[x] = OLD             % mark it as visited
        for each neighbor y of x {
           enqueue(Q,y)        %   add y to Q
}}}
```

When run on a connected graph G, this algorithm will mark every vertex as visited exactly once, and the order in which it marks them depends on the order in which they are added to and then removed from the queue Q.

If we want to keep track of other information, such as the search tree T itself, or the distance along the tree from s to a particular vertex, then we need to modify the algorithm to keep track of these things, as the algorithm proceeds. For instance, to recover the search tree T, we need to store not just a vertex y in the Q, but also the neighbor x along whose edge we found y. That is, Q should

[8] If you've forgotten what a queue is, it is an ADT that supports only the operations `add` and `remove`. Crucially, search is *not* supported as part of adding and removing. Queues are often implemented via a simple 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*.

store an ordered pair, instead of only a single value. And, to track the distances, we need to store the distance to each x that we mark in an auxiliary array. That way, when we add an edge (x, y) to the tree T, we can write down the distance to y as the distance to x plus one, since there is one more edge in the path to y than the path to x.

```
Search-Tree(G,s) {
  for i = 1 to n {
     state[i] = NEW            % mark each vertex is unvisited
     pred[i]  = NULL           % initialize predecessor array
     dist[i]  = INF            % distance from s to i
  }

  dist[s]  = 0                 % distance to s = 0
  Q = newQueue( (NULL,s) )     % make a queue, containing s

  while Q not empty {          % while not yet done
     (p,x) = dequeue(Q)        % get a vertex x out of Q
     if state[x] == NEW {      % if x is unvisited
        state[x] = OLD         % mark it as visited
        pred[x]  = p           % record how we found x
        dist[x] = dist[p] + 1  % distance to x
        for each neighbor y of x {
           enqueue(Q,(x,y))    %   add y to Q
}}}
```

When `Search-Tree` terminates, several things are true: (i) all nodes reachable from s have been marked `OLD` and the only nodes that are still marked `NEW` are those for which there does not exist a path from s, (ii) the array `pred` contains the search tree T, and (iii) the value `dist[i]` gives the distance (number of edges) in the search tree from s to i.

If we use a FIFO queue as Q, then `Search-Tree` grows a BFS tree T. Abstractly, T grows outward from s in layers. "Tree edges" always connect some vertex in layer ℓ to a vertex in layer $\ell + 1$, and all nodes in some layer ℓ are exactly a distance ℓ from s. Furthermore, each edge in G that is not in the BFS tree T must connect a pair of nodes at the same distance from s. The tree T is thus a kind of *spanning tree* for all the nodes reachable in G from s, although it is not necessarily a minimum spanning tree.[9]

3.1.1 Running time

We now analyze this algorithm's asymptotic running time.

[9] We will cover minimum spanning trees a little later in the semester.

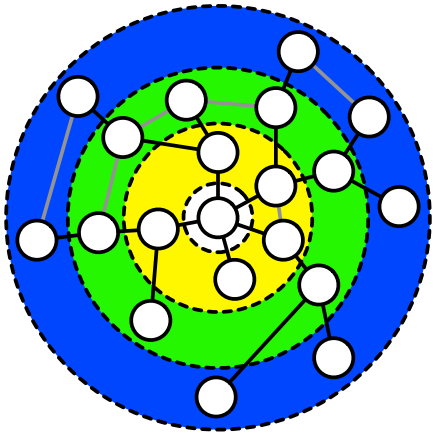

Figure 2: A BFS tree T on a simple graph $G(V, E)$. BFS explores the network in layers; edges in T are shown in black and non-tree edges $E - T$ are shown in grey.

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?

Initializing the three arrays `state[.]`, `pred[.]` and `dist[.]` at the top of the function takes $\Theta(V)$ time. The next two steps, setting the distance to s as 0 and adding the pair (NULL, s) to the Q, both take $\Theta(1)$ time. Hence, the initialization part of the algorithm takes $\Theta(V)$ time.

The time required for the search part of the algorithm depends on the number of items placed into the queue Q. Each time we remove some vertex x from Q, we first check whether it has been previously visited. If it has, then it must already be in the search tree T. This must be the case because the only time we mark a vertex as visited is when we record its distance from s and the predecessor vertex in the tree. The event happens the first time a vertex is dequeued from the queue. Hence, we may discard it, and move to the next vertex in the queue. For this reason, a vertex is marked as visited at most once before the algorithm halts.

If it has not previously been visited when it is dequeued, we mark it as visited and only then enqueue each of its k_x neighbors. That process takes $O(k_x)$ time. Hence, the maximum number of items that can ever be placed in the queue is given by summing the degree of each vertex over all vertices, i.e., $\sum_{y=1}^{|V|} k_y = 2\,|E| = \Theta(E)$. (Shorter argument: every edge is placed in the queue twice, once for (x, y) and once for (y, x).) Each time around the `while` loop, we dequeue exactly

one vertex, and when the queue is empty, the algorithm terminates. Hence, the total running time is given by the sum of the initialization part and the search part, giving a total running time of $O(V + E)$.

3.1.2 Correctness

Assume that G is a connected graph. The vertex s is the first vertex marked as visited by the algorithm, and initially all other vertices are marked as unvisited. Let T represent the set of edges contained in the `pred` array. Initially, T contains no edges.

Each time around the `while` loop, exactly one edge (u, v) is dequeued. If the vertex v has previously been marked as visited, then v is already part of T, having been added when the algorithm dequeued some other edge (w, v) prior to dequeuing (u, v). Once a vertex is marked as visited, the algorithm cannot mark it as un-visited.

If the vertex v is marked as unvisited, the algorithm marks v as visited and adds the edge (u, v) to T, implying that the algorithm has found a path $s \to \cdots \to u \to v$. (In the base case, $u = s$.) Because G is connected, such a path always exists. In order to dequeue the edge (u, v), the algorithm must have previously marked u as visited and, in the inner `for` loop, added the edge (u, v) to the queue. Hence, by induction on the length of the path from s to v, the algorithm visits every vertex reachable from s, and T is a spanning tree on G. Once T spans G, every subsequent edge dequeued will terminate at a vertex already marked as visited, and the algorithm will empty the queue and terminate. □

3.1.3 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 FIFO queue, adding vertices to the end of the Q and removing them from the front; in DFS, we use it as a FILO queue, both adding and removing vertices from the same side of the Q.[10]

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,

[10] 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.

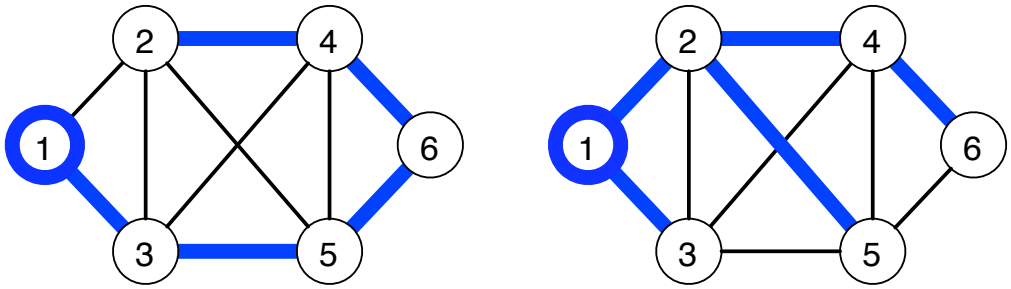

Figure 3: The left-hand figure shows a DFS while the right-hand figure shows a BFS; the bold circled vertex 1 is the source.

DFS always pushes "deeper" into the graph, producing long and "stringy" trees, backtracking only when it can discover no new (unmarked) vertices.[11]

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.

3.1.4 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 to explore all the components? First, we need to modify `Search-Tree` slightly, to move the initialization routines up a level in the program so that they are only executed once (otherwise, each time we call `Search-Tree` on a new vertex, we overwrite all our previous work). And, distances are only defined relative to a fixed source vertex s, but in this generalization, we may use multiple source vertices (one for each component). In the code below, we omit the distance vector completely, but we could just as easily include it as an output of `Search-Tree` if we wanted to do something with it before growing the next search tree.

The key insight in generalizing `Search-Tree` to `Search-Forest` is that the vector v contains a list of all vertices we have, so far, visited. Each time `Search-Tree` is called on some s, all vertices in the component that contains s will, when `Search-Tree` exits, be labeled `OLD`. Thus, if any vertex u is still labeled `NEW` when `Search-Tree` exits, then there exists some component (containing u) that has not yet been explored. A simple solution to growing a complete search-tree forest is to

[11] 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.

iteratively try each vertex as the root of a new search tree. If a vertex has been labeled `OLD` by a tree rooted at some previous value, we simply move to the next vertex:

```
Search-Forest(G) {
  for i = 1 to n {
     state[i] = NEW
     pred[i]  = NULL        % initialize auxiliary arrays
   }
   for i = 1 to n {
      if state[i] == NEW { 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.

On your own: 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.[12] 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*.

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. □

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

Figure 4: The Seven Bridges of Königsberg and their graph representation (images from Wikipedia).

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 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 On your own

1. Read Chapter 22 "Elementary Graph Algorithms"