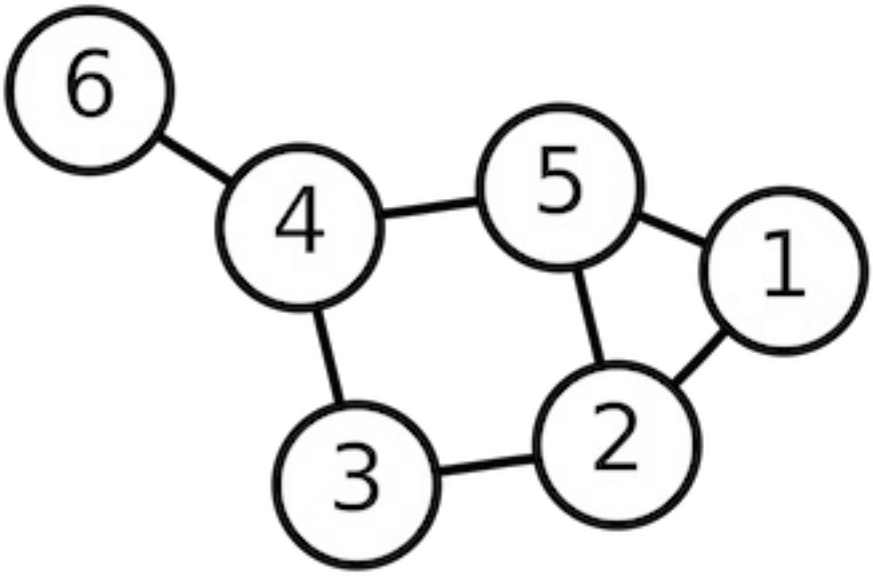
Graphs

Graphs are a different way of looking at your data. The best way to understand a graph is by looking at some examples. We look at data points as *nodes* and see the connections between them (*edges*).

Example Graphs

1. A Social Network (e.g. Facebook)

    - A node is a person.
    - Two nodes are connected with an edge if they are friends.



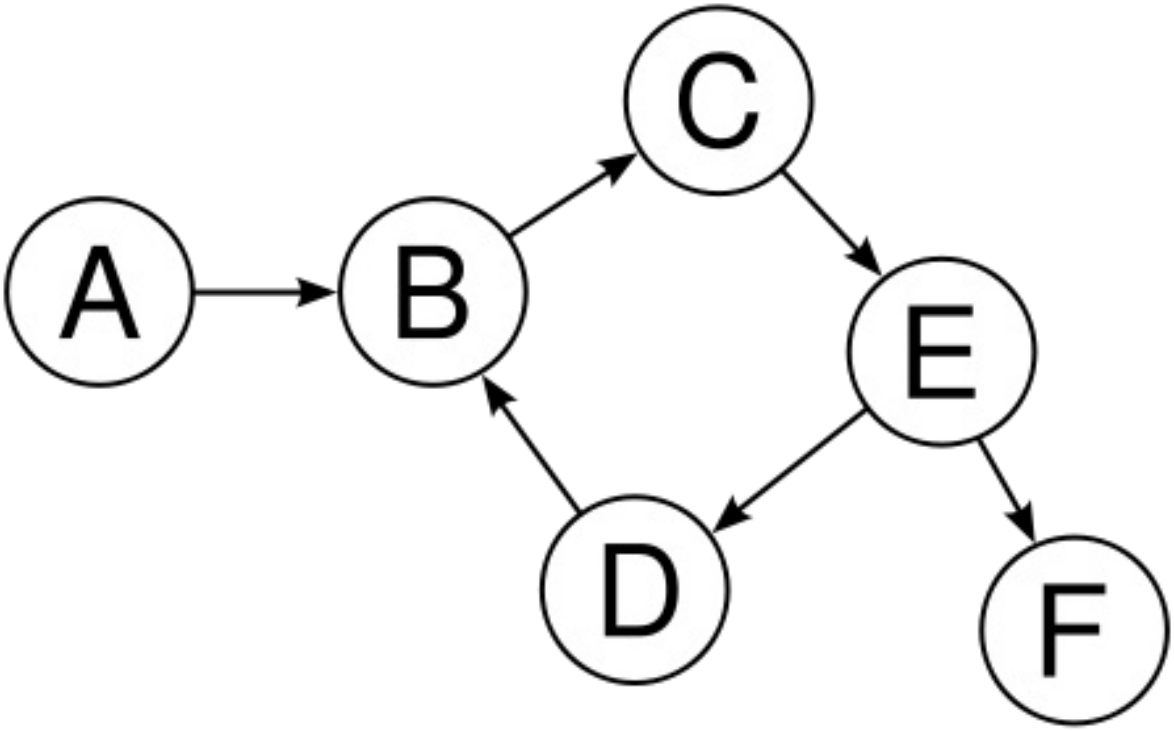
Here there are 6 users. These are their friend lists:

```
1: 2, 5
2: 1, 3, 5
3: 2, 4
4: 5, 6
```

```
5: 1, 2, 4
6: 4
```

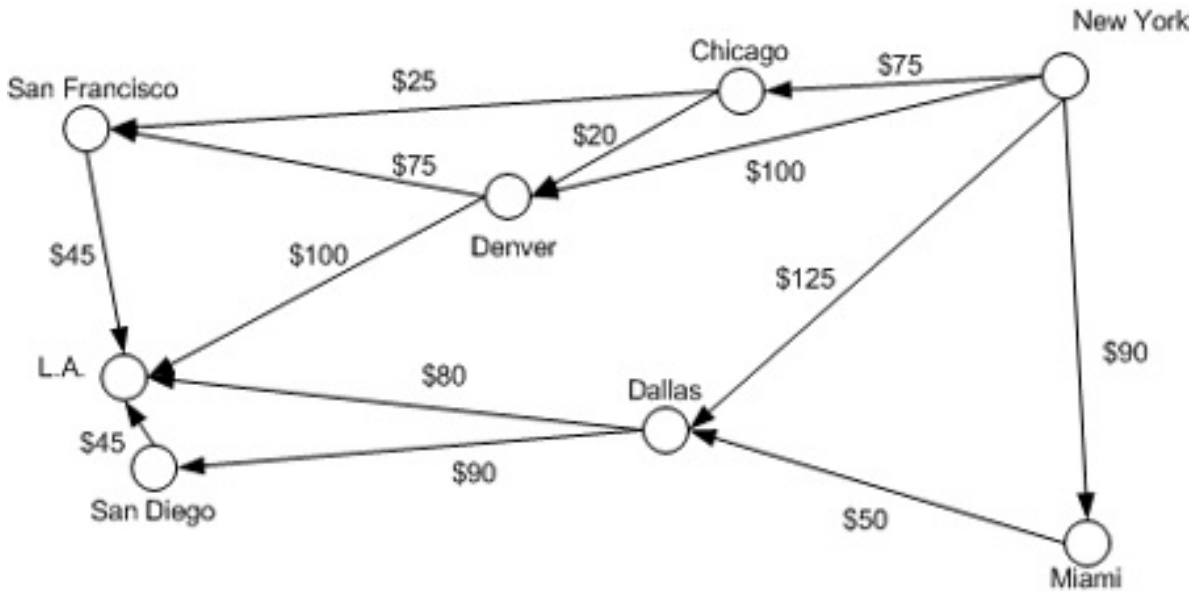
2. A One-Directional Social Network (e.g. Twitter)

    - A node is a person.
    - There is an edge from person A to person B if person A follows person B



3. Plane Routes

    - A node is an airport.
    - Two airports are connected if there is a flight between them. The edge is *weighted* according to the cost of the flight.



Terminology

Graphs come with a lot of terminology….

Node (Vertex)	A *node* (aka *vertex*) is a datapoint in the graph.
Edge	An *edge* is a connection between two nodes. If there's an edge between two nodes, we say that those nodes are *connected*. This can be that two users are friends, that there's a bus line between two cities, a link between two webpages, etc.

Types of Graphs

Directed Graph	A *directed graph* is a graph where edges only go in one direction. For example, in Twitter, user A can follow user B without user B following user A.
Undirected Graph	In an *undirected graph*, edges go both ways. For example, with Facebook, if user A is friends with user B, then user B is friends with user A.
Weighted Graph	In a *weighted graph*, the edges have weights. For example, in a graph of airports and flight routes, the edges can have weights of the cost or

	number of miles.
Unweighted Graph	In an *unweighted graph*, all edges are the same. For example, with Facebook, there aren't different levels of FB (though you could imagine them existing…)

More Terminology

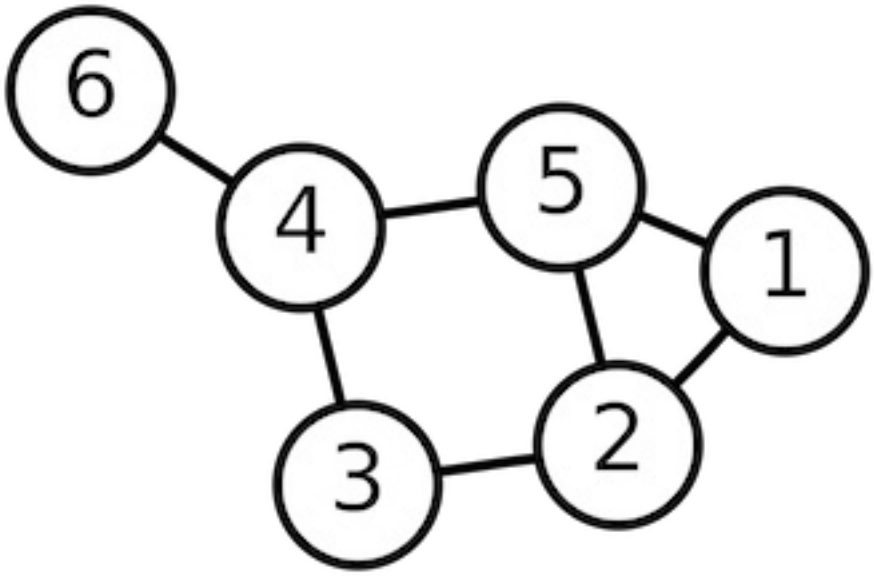
Neighbors	The *neighbors* of a node are the nodes that it is connected to. e.g., B is a neighbor of A if there is an edge from A to B.
Degree	The *degree* is the number of neighbors a node has. If the graph is directed, we have separate values for the *outdegree* (number of edges leaving the node) and *indegree* (number of edges coming into the node)
Complete	A graph is *complete* if there is an edge from every node to every other node. This is generally only seen in weighted graphs.
Path	A *path* is a series of nodes and the edges that connect them. For example, in the airport route example, a path from New York to San Diego is `New York, Miami, Dallas, San Diego`. Note that there are often multiple paths and this example isn't the *shortest* path from New York to San Diego.
Connected	A graph is *connected* if there is a path from every node to every other node.
Subgraph	A *subgraph* is a subset of the nodes of a graph and all the edges between them.
Connected Component	A *connected component* is a subgraph that is *connected*.

Representing Graphs

There are two main ways of storing graphs. Keep in mind the simple operations that are often useful:

- Getting a list of a nodes neighbors
- Determining if two nodes are connected
- Calculating the degree of a node

Take another look at the first example so we can see the representations:



Adjacency Lists

This is the most common way of storing a graph.

For each node, we store a list of its neighbors. In python, we could use a set for this (but we would still probably call it an adjacency list).

```
1: {2, 5}
2: {1, 3, 5}
3: {2, 4}
4: {5, 6}
5: {1, 2, 4}
6: {4}
```

You can see that all of the above operations are easy! If we have an undirected graph, we do need to make sure that if we remove edge (1, 2), we remove 2 from 1's list as well as removing 1 from 2's list.

Adjacency Matrix

We create an `n x n` matrix (where `n` is the number of nodes). The value in cell `i, j` is 1 if the nodes are connected and 0 if they aren't connected. If it's a weighted graph, that value would

be the weight.

Here's the adjacency matrix for the above example.

```
    1 2 3 4 5 6
1 [[0 1 0 0 1 0]
2  [1 0 1 0 1 1]
3  [0 1 0 1 0 1]
4  [0 0 1 0 1 1]
5  [1 1 0 1 0 1]
6  [0 0 0 1 0 0]]
```

Searching

Graph Search Algorithms are ways of traversing all of the nodes of a graph.

These are useful for several reasons:

- Finding the shortest path between two nodes.
    - Find out an actor's Kevin Bacon number: You have a Kevin Bacon number of 1 if you've been in a film with him. Your number is 2 if you've been in a film with someone who's been in a film with him, etc…
    - Find out how many clicks it takes to get from cnn.com to galvanize.com.
- Finding an extended network.
    - Find all the users who are my friends of friends of friends).

Breadth First Search

The most common and most useful search algorithm is Breadth First Search (BFS). Here's the general algorithm:

- We start at a node
- We investigate all the neighbors of that node
- We investigate all the neighbors of neighbors
- We investigate all the neighbors of neighbors of neighbors
- etc, etc, …

If we're trying to find the shortest path from A to B, we stop after we find B.
If we're trying to find an extended network, we stop after how every many iterations we're looking for.

Queues

In order to implement the BFS algorithm, we first need to talk about queues. A *queue* is a data structure where you have these operations:

- Add to end
- Remove from beginning
- See if it's empty

We see queues used in processor queues, since a computer can only run one program at a time, so it needs a queue of the processes that are waiting to run. It will run them in the order that they got in the line.

Note: A python list would not be good for implementing a queue. We can add to the end of the list efficiently, but to remove from the beginning, we need to slide everything over to replace the hole. They are generally implemented with what are called linked lists. Python has an implementation of queues for us, so we can just use that.

BFS Pseudocode

```
function BFS(A, B):
    create an empty queue Q
    initialize empty set V (set of visited nodes)
    add A to Q
    while Q is not empty:
        take node off Q, call it N
        if N isn't already visited:
            add N to the visited set
            do whatever we want to do with N (will depend on the application)
            add every neighbor of N to Q
```

Shortest Path Pseudocode

As we mentioned above, one use case of BFS is for finding the shortest path. Here is the pseudocode. Note that along with storing the nodes we've visited, we need to store the length of the path from A to that node.

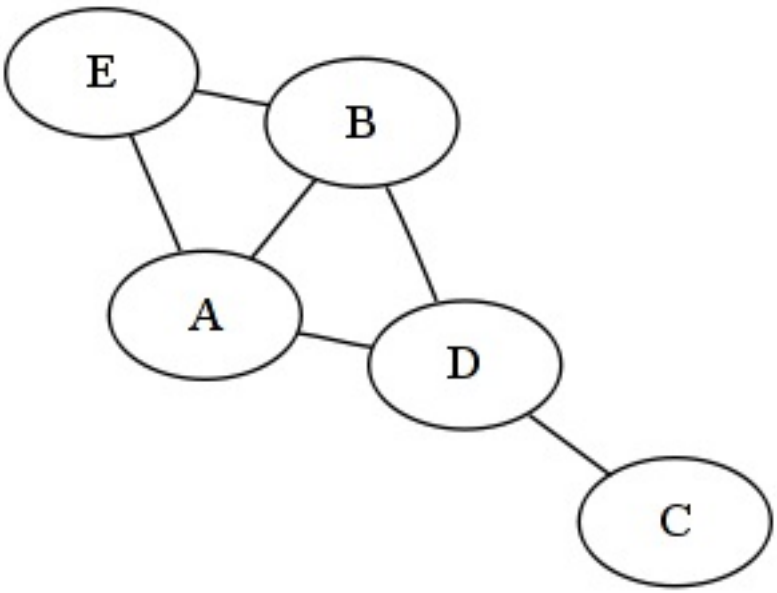
If we cared about the path and not just the distance, we'd have to store the whole path.

```
function shortest_path(A, B):
    create an empty queue Q
    initialize empty set V (set of visited nodes)
    add (A, 0) to Q
```

```
    while Q is not empty:
        take node off Q, call it N (comes with distance d)
        if N isn't already visited:
            add N to the visited set
            if N is B: we're done!
            else: add every neighbor of N to Q with distance d+1
```

Example

Let's take a look at the following example:



Let's find the shortest path from node A to node C.

At each iteration we can look at the values for `Q`, `V` and `N`

	Current Node (`N`)	Queue (`Q`)	Visited (`V`)	explanation
		`A,0`		*initialization*
1	`A,0`			*take first node off queue*
	`A,0`	`E,1|B,1|D,1`	`A`	*add A's neighbors to queue*
2	`E,1`	`B,1|D,1`	`A`	*take E off queue*
	`E,1`	`B,1|D,1|A,2|B,2`	`A,E`	*add E's neighbors to queue*
3	`B,1`	`D,1|A,2|B,2`	`A,E`	*take B off queue*
	`B,1`	`D,1|A,2|B,2|E,2|D,2`	`A,E,B`	*add B's neighbors to queue*
4	`D,1`	`A,2|B,2|E,2|D,2`	`A,E,B`	*take D off queue*
	`D,1`	`A,2|B,2|E,2|D,2|C,2|B,2`	`A,E,B,D`	*add D's neighbors to queue*
5	`A,2`	`B,2|E,2|D,2|C,2|B,2`	`A,E,B,D`	*take A off queue*
	`A,2`	`B,2|E,2|D,2|C,2|B,2`	`A,E,B,D`	*skip A since already visited*

6	`B,2`	`E,2	D,2	C,2	B,2`	`A,E,B,D`	*take B off queue*
	`B,2`	`E,2	D,2	C,2	B,2`	`A,E,B,D`	*skip B since already visited*
7	`E,2`	`D,2	C,2	B,2`	`A,E,B,D`	*take E off queue*
	`E,2`	`D,2	C,2	B,2`	`A,E,B,D`	*skip E since already visited*
8	`D,2`	`C,2	B,2`	`A,E,B,D`	*take D off queue*
	`D,2`	`C,2	B,2`	`A,E,B,D`	*skip D since already visited*
9	`C,2`	`B,2`	`A,E,B,D`	*take C off queue*
	`C,2`	`B,2`	`A,E,B,D`	*We're done!!*

Centrality

Centrality measures are measures of the importance of nodes. There are several ways that we compute these, but we generally standarize them all to be normalized from 0 to 1.

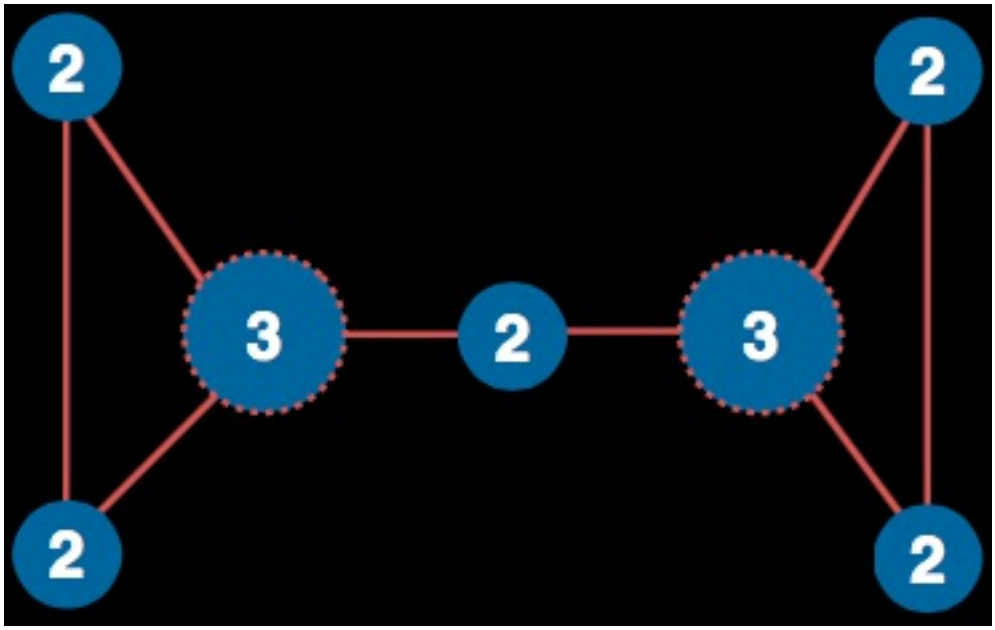
Degree Centrality

The *degree centrality* is the degree of the node divided by a normalizing factor. We divide by `n-1` since that's the maximum possible degree.

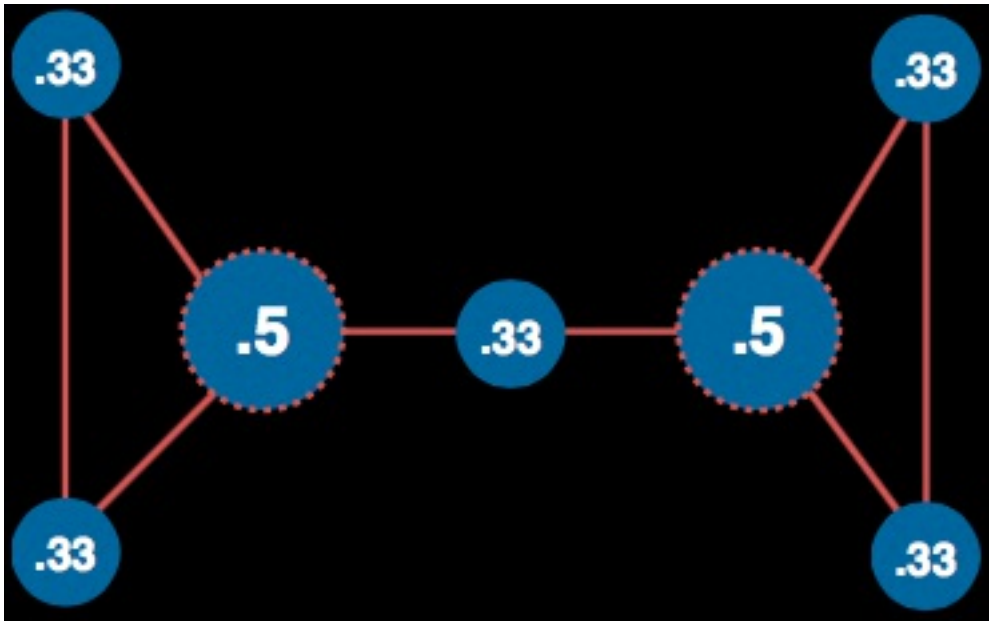
```
                          d(n)            degree of n
degree centrality(n) = --------- = --------------------------
                       |V| - 1     number of nodes in G - 1
```

In the same example from above, node 4 has a degree of 3 and a degree centrality of `3/(6+1) = 0.429`.

Note: Degree centrality doesn't always capture the important nodes. Take a look at the following example with the degree centralities calculated. You could argue that the middle node is the most important.



Normalized:



Betweenness Centrality

To better represent the importance of a node, we can use the *betweenness centrality*. This is a measure of how many paths the node is a part of.

Formally, here's the definition:

$$
\begin{aligned}
\text{betweenness}(v) &= \sum_{s \neq v \neq t} \text{percent of shortest paths from } s \text{ to } t \text{ which pass through } v \\
&= \sum_{s \neq v \neq t} \frac{\sigma_{st}(v)}{\sigma_{st}} \\
\text{where} & \\
\sigma_{st}(v) &= \#\text{ of shortest paths from } s \text{ to } t \text{ which pass through } v \\
\sigma_{st} &= \#\text{ of shortest paths from } s \text{ to } t
\end{aligned}
$$

$$
\text{normalized betweenness}(v) = \frac{\text{betweenness(v)}}{(n-1)(n-2)}
$$

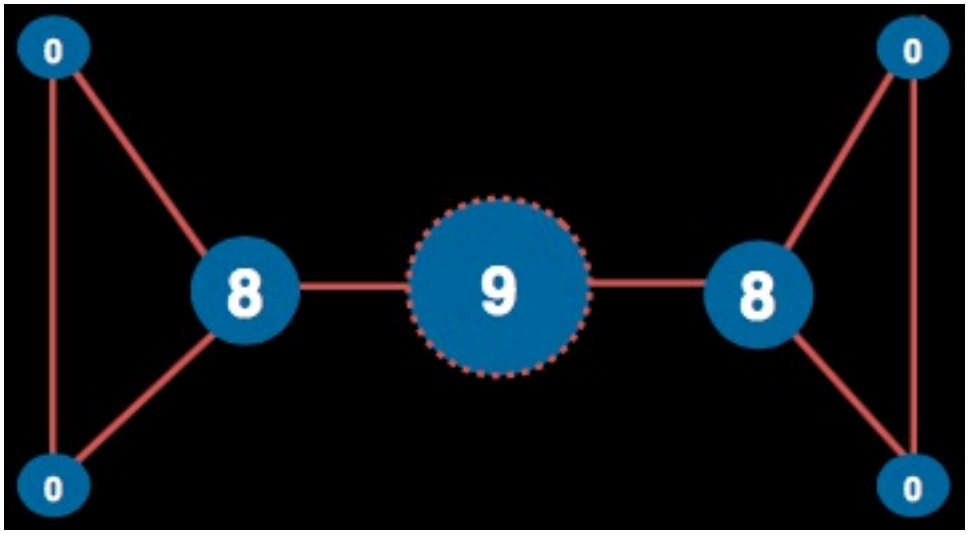
To normalize, we are dividing by `(n-1)(n-2)` for directed graphs and `(n-1)(n-2)/2` for undirected graphs, since that's the maximum possible value for the betweenness. This would happen if every single shortest path went through the node.

Let's calculate the betweenness centrality for node 4 above. There are `5*4=20` terms, so the 0 terms are omitted here for simplicity.

$$
\begin{aligned}
\text{betweeness}(4) &= \sum_{s \neq 4 \neq t} \frac{\sigma_{st}(4)}{\sigma_{st}} \\
&= \frac{\sigma_{16}(4)}{\sigma_{16}} + \frac{\sigma_{26}(4)}{\sigma_{26}} + \frac{\sigma_{36}(4)}{\sigma_{36}} + \frac{\sigma_{56}(4)}{\sigma_{56}} + \frac{\sigma_{35}(4)}{\sigma_{35}} \\
&= \frac{1}{1} + \frac{2}{2} + \frac{1}{1} + \frac{1}{1} + \frac{1}{2} \\
&= 4.5
\end{aligned}
$$

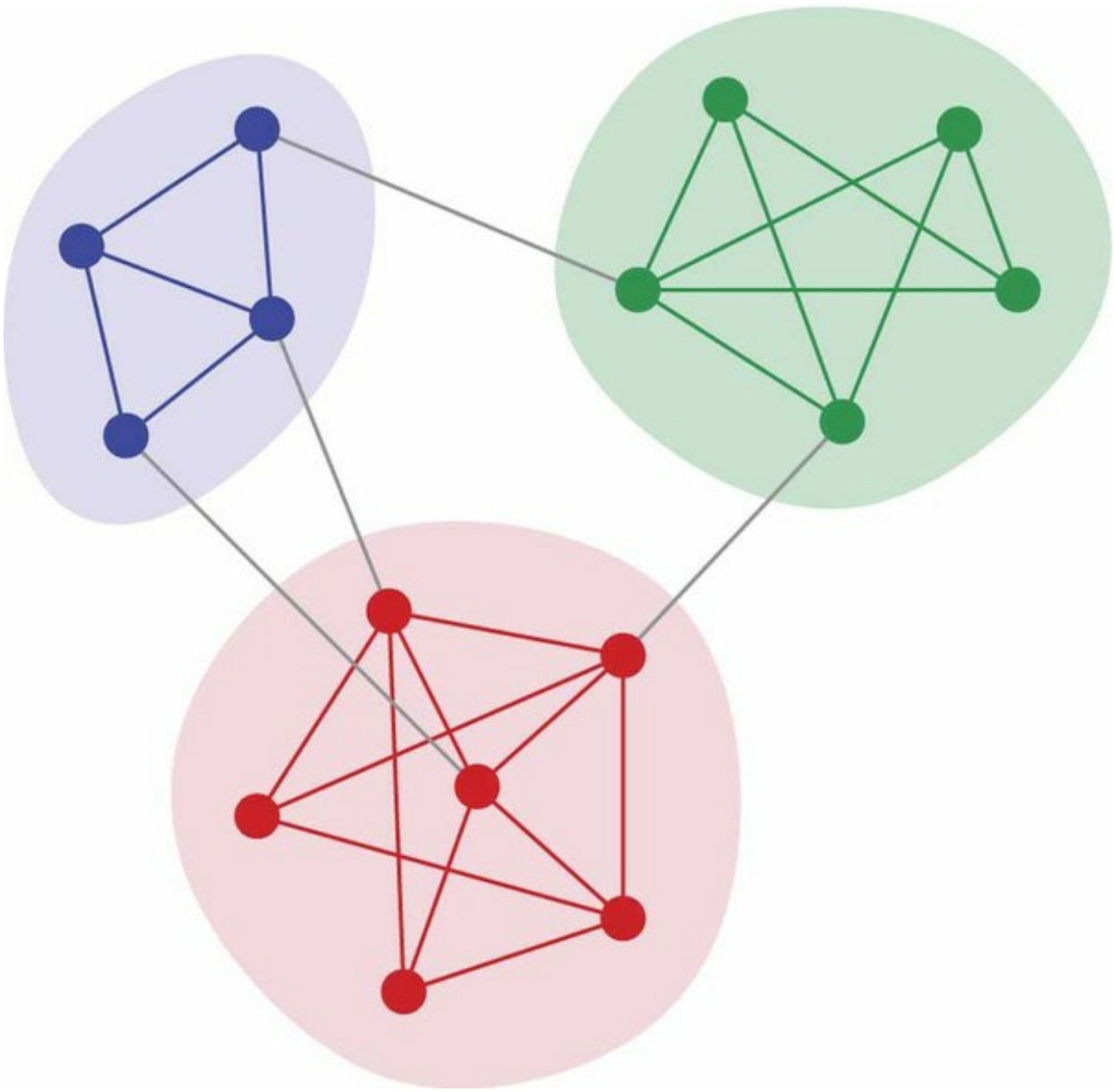
$$
\text{normalized betweeness}(4) = \frac{4.5}{(6-1)(6-2)} = 0.225
$$

Here's the betweenness centrality for the example from above:



Community Detection

A common task is to determine the *communities* within a graph.



If you think about the Facebook graph, you probably have a network of friends who are mostly friends with each other. This would be a community. While you have friends outside of your community, you have different connections outside of your community.

The idea to communities is that the there will be more edges inside the community than outside!

Modularity

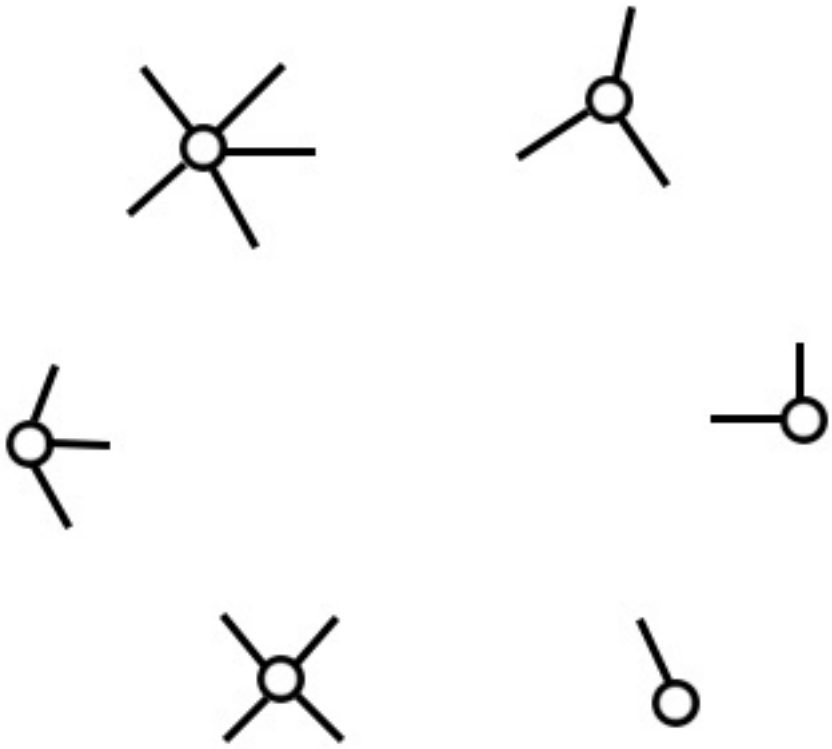
Modularity is a measure of how much of the edges are within the communities versus between communities. The higher the modularity, the better the communities.

The idea is that if we created a random graph where every node maintained it's degree, how many edges would there be within the communities (and not between)? How does this

compare to how many edges there actually are within communities.

We can start by calculating the expected number of edges within the communities.

The idea is to imagine cutting every edge in half. The two halves each remain connected to one node and you end up with something that looks like this:



Now randomly connect stubs together to form a graph! We can calculate the probability that node i gets connected to node j.

$$\mathrm{P}(\text{single edge stub gets connected to } j) = \frac{d(j)}{2m}$$

$$\mathrm{P}(\text{edge from } i \text{ to } j) = d(i) \cdot \frac{d(j)}{2m} = \frac{d(i)d(j)}{2m}$$

where

$$d(i) = \text{degree of node } i$$

$$m = \text{number of edges in the graph}$$

The expected number of edges which occur within communities is as follows:

$$\text{E(edges within communities)} = \sum_{i,j \text{ in same community}} \frac{d(i)d(j)}{2m}$$

The modularity is the true number of edges within communities minus the expected number of edges within communities. We also normalize by dividing by 2m.

$$\text{modularity}(G, \mathcal{C}) = \frac{1}{2m}\left(\sum_{C \in \mathcal{C}}(\#\text{ edges in } C) + \sum_{i,j \in \text{ same } C} \frac{d(i)d(j)}{2m}\right)$$

$$\text{where } \mathcal{C} \text{ is the collection of communities}$$

We often will see it written like this:

$$\text{modularity}(G, \mathcal{C}) = \frac{1}{2m}\sum_{C \in \mathcal{C}}\sum_{i,j \in C} A_{ij} - \frac{d(i)d(j)}{2m}$$

$$\begin{aligned}
&\text{where} \\
&\mathcal{C} = \text{the collection of communities} \\
&m = \text{number of edges in the graph} \\
&A_{ij} = \begin{cases} 1 & \text{if } (i,j) \text{ is an edge} \\ 0 & \text{if } (i,j) \text{ is not an edge} \end{cases} \\
&d(i) = \text{degree of node } i
\end{aligned}$$

Girvan-Newman Algorithm

The *Girvan-Newman Algorithm* is the most common community detection algorithm.

We iteratively remove the edge with the highest *edge betweenness*. The *edge betweenness* is a measure of how many paths an *edge* is part of.

Here's the formal definition for betweenness of an edge e (note that it's essentially the same as the node betweenness defined above).

$$\begin{aligned}\text{betweenness}(e) &= \sum_{s \neq v \neq t} \text{percent of shortest paths from } s \text{ to } t \text{ which pass through } e \\ &= \sum_{s \neq v \neq t} \frac{\sigma_{st}(e)}{\sigma_{st}} \\ \text{where} \\ \sigma_{st}(e) &= \#\text{ of shortest paths from } s \text{ to } t \text{ which pass through } e \\ \sigma_{st} &= \#\text{ of shortest paths from } s \text{ to } t\end{aligned}$$

$$\text{normalized betweenness}(e) = \frac{\text{betweenness(e)}}{(n-1)(n-2)}$$

Here's the pseudocode for the algorithm:

```
function GirvanNewman:
    repeat:
        repeat until a new connected component is created:
            calculate the edge betweenness centralities for all the edges
            remove the edge with the highest betweenness
```

This will iteratively create new communities. To determine the appropriate number of communities, we calculate the modularity for each set of communities and pick the one with the maximum modularity.