Chapter 1)

1)
Basics of NetworkX API, using Twitter network

To get you up and running with the NetworkX API, we will run through some basic functions that let you query a Twitter network that has been pre-loaded for you and is available in the IPython Shell as `T`. The Twitter network comes from **KONECT**, and shows a snapshot of a subset of Twitter users. It is an anonymized Twitter network with metadata.
You're now going to use the NetworkX API to explore some basic properties of the network, and are encouraged to experiment with the data in the IPython Shell.
Wait for the IPython shell to indicate that the graph that has been preloaded under the variable name `T` (representing a **T**witter network), and then answer the following question:
What is the size of the graph `T`, the type of `T.nodes()`, and the data structure of the third element of the last entry of `T.edges(data=True)`?
The `len()` and `type()` functions will be useful here. To access the last entry of `T.edges(data=True)`, you can use `T.edges(data=True)[-1]`.

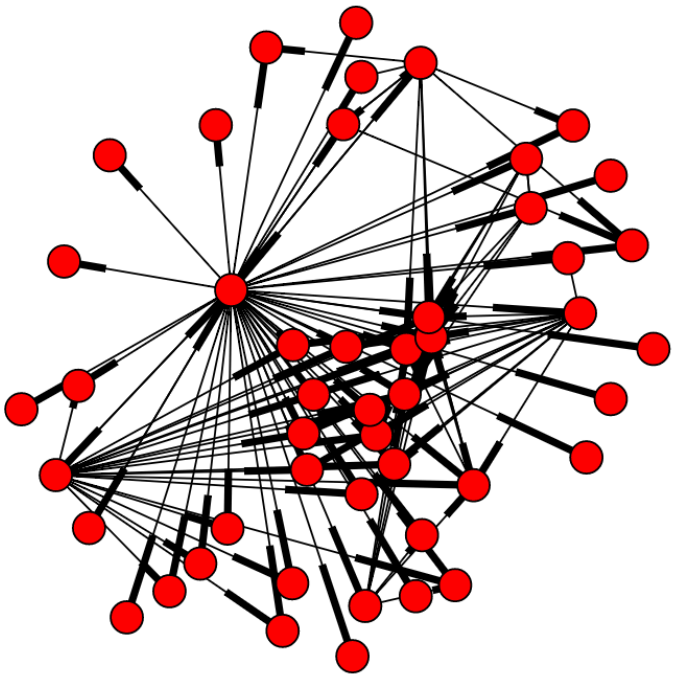
```
In [4]: type(T.nodes())
Out[4]: list

In [5]: len(T.nodes())
Out[5]: 23369

In [6]: T.edges(data=True)[-1]
Out[6]: (23324, 23327, {'date': datetime.date(2008, 2, 9)})
```

2)
Basic drawing of a network using NetworkX

NetworkX provides some basic drawing functionality that works for small graphs. We have selected a subset of nodes from the graph for you to practice using NetworkX's drawing facilities. It has been pre-loaded as `T_sub`.

```
# Import necessary modules
import matplotlib.pyplot as plt
import networkx as nx

# Draw the graph to screen
```

```
nx.draw(T_sub)
plt.show()
```



3)
Queries on a graph

Now that you know some basic properties of the graph and have practiced using NetworkX's drawing facilities to visualize components of it, it's time to explore how you can query it for nodes and edges. Specifically, you're going to look for "nodes of interest" and "edges of interest". To achieve this, you'll make use of the `.nodes()` and `.edges()` methods that Eric went over in the video.
The `.nodes()` method returns a list of nodes, while the `.edges()` method returns a list of tuples, in which each tuple shows the nodes that are present on that edge. Recall that passing in the keyword argument `data=True` in these methods retrieves the corresponding metadata associated with the nodes and edges as well.
You'll write list comprehensions to effectively build these queries in one line. For a refresher on list comprehensions, refer to **Part 2** of DataCamp's Python Data Science Toolbox course. Here's the recipe for a list comprehension:
`[` *output expression* `for` *iterator variable* `in` *iterable* `if` *predicate expression*`]`.
You have to fill in the `_iterable_` and the `_predicate expression_`. Feel free to prototype your answer by exploring the graph in the IPython Shell before submitting your solution.

- Use a list comprehension to get a **list of nodes** from the graph `T` that have the `'occupation'` label of `'scientist'`.

    - The *output expression* `n` has been specified for you, along with the *iterator variables* `n` and `d`. Your task is to fill in the *iterable* and the *conditional expression*.
    - Use the `.nodes()` method of `T` access its nodes, and be sure to specify `data=True` to obtain the metadata for the nodes.
    - The iterator variable `d` is a dictionary. The key of interest here is `'occupation'` and value of interest is `'scientist'`.
- Use a list comprehension to get a **list of edges** from the graph `T` that were formed for at least 6 years, i.e., from before **1 Jan 2010**.
    - Your task once again is to fill in the *iterable* and *conditional expression*.
    - Use the `.edges()` method of `T` to access its edges. Be sure to obtain the metadata for the edges as well.
    - The dates are stored as `datetime.date` objects in the metadata dictionary `d`, under the key `'date'`. To access the date 1 Jan 2009, for example, the dictionary value would be `date(2009, 1, 1)`.

```
# Use a list comprehension to get the nodes of interest: noi
noi = [n for n, d in T.nodes(data=True) if d['occupation'] == 'scientist']

# Use a list comprehension to get the edges of interest: eoi
eoi = [(u, v) for u, v, d in T.edges(data=True) if d['date'] < date(2010,1,1)]
```

Note)

Undirected graphs

```
In [1]: import networkx as nx

In [2]: G = nx.Graph()

In [3]: type(G)
Out[3]: networkx.classes.graph.Graph
```

Directed graphs

```
In [4]: D = nx.DiGraph()

In [5]: type(D)
Out[5]: networkx.classes.digraph.DiGraph
```

Multi-edge (Directed) graphs

```
In [6]: M = nx.MultiGraph()

In [7]: type(M)
Out[7]: networkx.classes.multigraph.MultiGraph

In [8]: MD = nx.MultiDiGraph()

In [9]: type(MD)
Out[9]: networkx.classes.multidigraph.MultiDiGraph
```

4)

Specifying a weight on edges

Weights can be added to edges in a graph, typically indicating the "strength" of an edge. In NetworkX, the weight is indicated by the `'weight'` key in the metadata dictionary.
Before attempting the exercise, use the IPython Shell to access the dictionary metadata of `T` and explore it, for instance by running the commands `T.edge[1][10]` and then `T.edge[10][1]`. Note how there's only one field, and now you're going to add another field, called `'weight'`.

- Set the `'weight'` attribute of the edge between node `1` and `10` of `T` to be equal to `2`. Refer to the following template to set an attribute of an edge: `network_name.edge[node1][node2]['attribute'] = value`. Here, the `'attribute'` is `'weight'`.
- Set the weight of every edge involving node `293` to be equal to `1.1`. To do this:
    - Using a `for` loop, iterate over all the edges of `T`, including the `metadata`.
    - If `293` is involved in the list of nodes `[u, v]`:
        - Set the weight of the edge between `u` and `v` to be `1.1`.

```
# Set the weight of the edge
T.edge[1][10]['weight'] = 2

# Iterate over all the edges (with metadata)
for u, v, d in T.edges(data=True):

    # Check if node 293 is involved
    if 293 in [u,v]:

        # Set the weight to 1.1
```

```
T.edge[u][v]['weight'] = 1.1
```

5)

Checking whether there are self-loops in the graph

As Eric discussed, NetworkX also allows edges that begin and end on the same node; while this would be non-intuitive for a social network graph, it is useful to model data such as trip networks, in which individuals begin at one location and end in another.
It is useful to check for this before proceeding with further analyses, and NetworkX graphs provide a method for this purpose: `.number_of_selfloops()`.
In this exercise as well as later ones, you'll find the `assert` statement useful.
An `assert`-ions checks whether the statement placed after it evaluates to True, otherwise it will return an `AssertionError`.
To begin, use the `.number_of_selfloops()` method on `T` in the IPython Shell to get the number of edges that begin and end on the same node. A number of self-loops have been synthetically added to the graph. Your job in this exercise is to write a function that returns these edges.

- Define a function called `find_selfloop_nodes()` which takes one argument: `G`.
  - Using a `for` loop, iterate over all the edges in `G` (excluding the metadata).
  - If node `u` is equal to node `v`:
    - Append `u` to the list `nodes_in_selfloops`.
    - Return the list `nodes_in_selfloops`.
- Check that the number of self loops in the graph equals the number of nodes in self loops.

```
# Define find_selfloop_nodes()
def find_selfloop_nodes(G):
    """
    Finds all nodes that have self-loops in the graph G.
    """
    nodes_in_selfloops = []

    # Iterate over all the edges of G
    for u, v in G.edges():

    # Check if node u and node v are the same
        if u==v:

            # Append node u to nodes_in_selfloops
            nodes_in_selfloops.append(u)
```

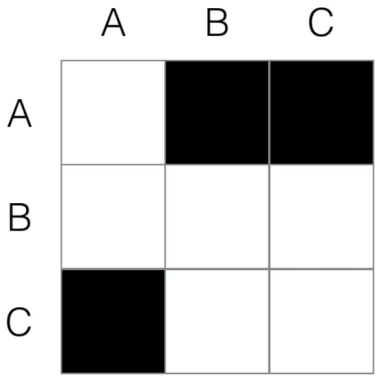
```
    return nodes_in_selfloops

# Check whether number of self loops equals the number of nodes in self loops
assert T.number_of_selfloops() == len(find_selfloop_nodes(T))
```

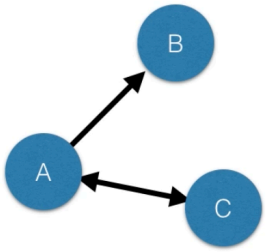
Note)

Visualizing networks

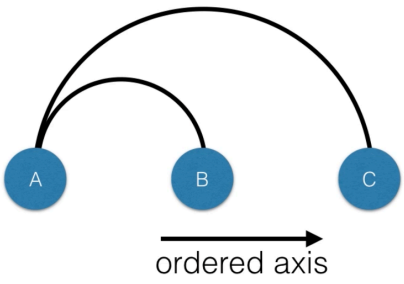
- Matrix plots
- Arc plots
- Circos plots

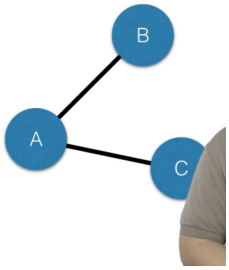
Directed matrices

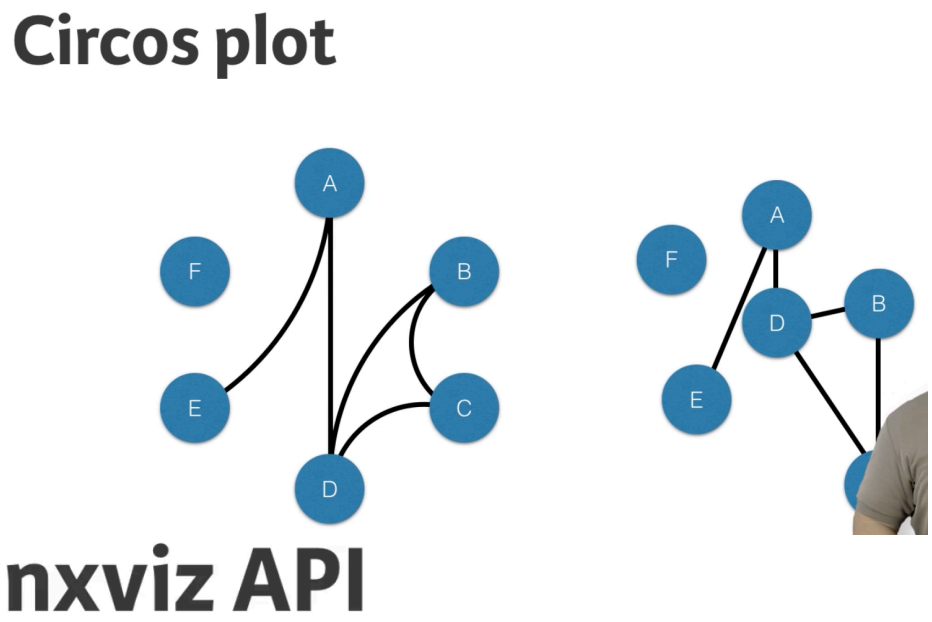




Arc plot








```
In [1]: import nxviz as nv

In [2]: import matplotlib.pyplot as plt

In [3]: ap = nv.ArcPlot(G)

In [4]: ap.draw()

In [5]: plt.show()
```

6)

Visualizing using Matrix plots

It is time to try your first "fancy" graph visualization method: a matrix plot. To do this, `nxviz` provides a `MatrixPlot` object.

`nxviz` is a package for visualizing graphs in a rational fashion. Under the hood, the `MatrixPlot` utilizes `nx.to_numpy_matrix(G)`, which returns the matrix form of the graph. Here, each node is one column and one row, and an edge between the two nodes is indicated by the value 1. In doing so, however, only the `weight` metadata is preserved; all other metadata is lost, as you'll verify using an `assert` statement.

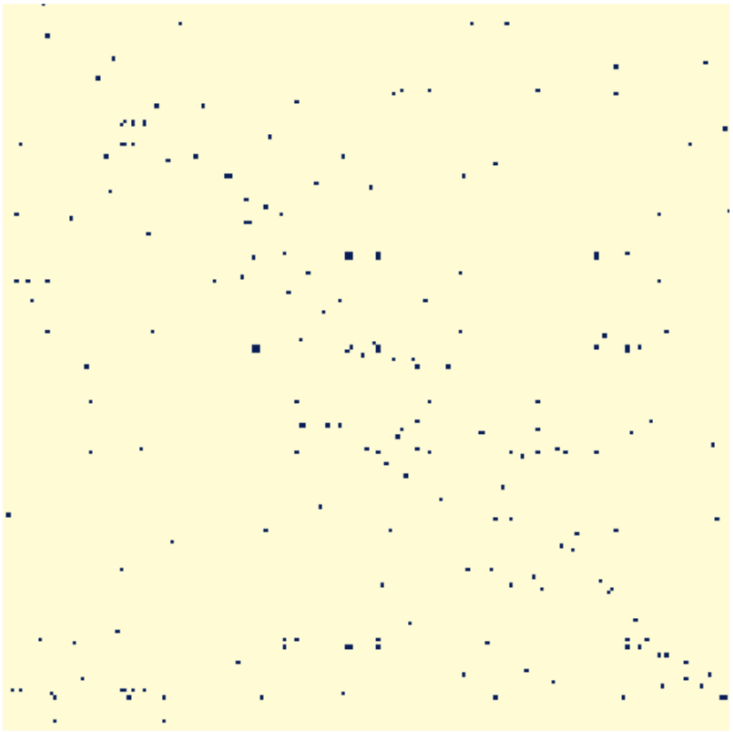
A corresponding `nx.from_numpy_matrix(A)` allows one to quickly create a graph from a NumPy matrix. The default graph type is `Graph()`; if you want to make it a `DiGraph()`, that has to be specified using the `create_using`keyword argument, e.g. (`nx.from_numpy_matrix(A, create_using=nx.DiGraph)`).

One final note, `matplotlib.pyplot` and `networkx` have already been imported as `plt` and `nx`, respectively, and the graph `T` has been pre-loaded. For simplicity and speed, we have sub-sampled only 100 edges from the network.

- Import `nxviz` as `nv`.
- Plot the graph `T` as a matrix plot. To do this:
    - Create the `MatrixPlot` object called `m` using the `nv.MatrixPlot()`function with `T` passed in as an argument.
    - Draw the `m` to the screen using the `.draw()` method.
    - Display the plot using `plt.show()`.

- Convert the graph to a matrix format, and then convert the graph to back to the NetworkX form from the matrix as a directed graph. This has been done for you.
- Check that the `category` metadata field is lost from each node.

```
# Import nxviz
import nxviz as nv

# Create the MatrixPlot object: m
m = nv.MatrixPlot(T)

# Draw m to the screen
m.draw()

# Display the plot
plt.show()

# Convert T to a matrix format: A
A = nx.to_numpy_matrix(T)

# Convert A back to the NetworkX form as a directed graph: T_conv
T_conv = nx.from_numpy_matrix(A, create_using=nx.DiGraph())

# Check that the `category` metadata field is lost from each node
for n, d in T_conv.nodes(data=True):
    assert 'category' not in d.keys()
```



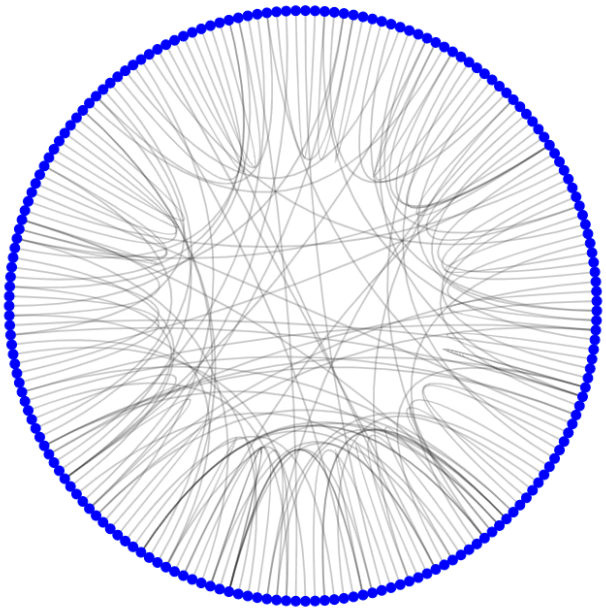
7)
Visualizing using Circos plots

Circos plots are a rational, non-cluttered way of visualizing graph data, in which nodes are ordered around the circumference in some fashion, and the edges are drawn within the circle that results, giving a beautiful as well as informative visualization about the structure of the network.
In this exercise, you'll continue getting practice with the nxviz API, this time with the `CircosPlot` object. `matplotlib.pyplot` has been imported for you as `plt`.

- Import `CircosPlot` from `nxviz`.
- Plot the Twitter network `T` as a Circos plot without any styling. Use the `CircosPlot()` function to do this. Don't forget to draw it to the screen using `.draw()` and then display it using `plt.show()`.

```
# Import necessary modules
import matplotlib.pyplot as plt
from nxviz import CircosPlot

# Create the CircosPlot object: c
c = CircosPlot(T)

# Draw c to the screen
c.draw()

# Display the plot
```

plt.show()



8)
Visualizing using Arc plots

Following on what you've learned about the nxviz API, now try making an ArcPlot of the network. Two keyword arguments that you will try here are `node_order='keyX'` and `node_color='keyX'`, in which you specify a key in the node metadata dictionary to color and order the nodes by.
`matplotlib.pyplot` has been imported for you as `plt`.

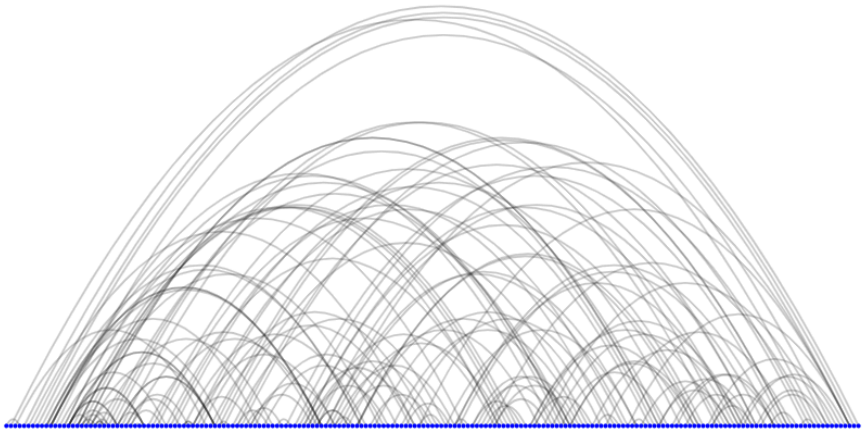
- Import `ArcPlot` from `nxviz`.
- Create an un-customized ArcPlot of `T`. To do this, use the `ArcPlot()`function with just `T` as the argument.
- Create another ArcPlot of `T` in which the nodes are ordered and colored by the `'category'` keyword. You'll have to specify the `node_order` and `node_color` parameters to do this. For both plots, be sure to draw them to the screen and display them with `plt.show()`.

In [1]: T.edges(data=True)[1]
Out[1]: (13829, 13834, {'date': datetime.date(2012, 5, 24)})

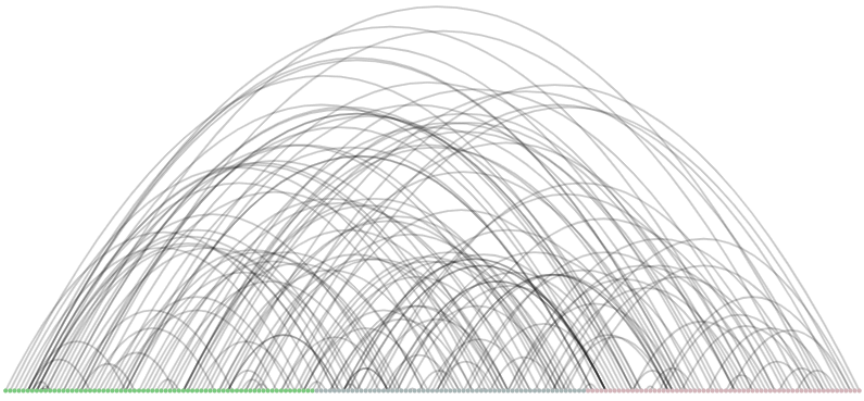
In [2]: T.nodes(data=True)[2]
Out[2]: (13834, {'category': 'D', 'occupation': 'politician'})

# Import necessary modules
import matplotlib.pyplot as plt
from nxviz import ArcPlot

```
# Create the un-customized ArcPlot object: a
a = ArcPlot(T)

# Draw a to the screen
a.draw()

# Display the plot
plt.show()
```



```
# Create the customized ArcPlot object: a2
a2 = ArcPlot(T,node_order='category',node_color='category')

# Draw a2 to the screen
a2.draw()

# Display the plot
plt.show()
```



Chapter 2)

Note)

Degree centrality

- Definition:

$$\frac{\text{Number of Neighbours I Have}}{\text{Number of Neighbours I Could Possibly Have}}$$

- Examples of nodes with high degree centrality:
  - Twitter broadcasters
  - Airport transportation hubs
  - Disease super-spreaders

Number of neighbors

```
In [1]: G.edges()
Out[1]: [(1, 2), (1, 3), (1, 4), (1, 5), (1, 6), (1, 7), (1, 8),
(1, 9)]

In [2]: G.neighbors(1)
Out[2]: [2, 3, 4, 5, 6, 7, 8, 9]

In [3]: G.neighbors(8)
Out[3]: [1]

In [4]: G.neighbors(10)
……
NetworkXError: The node 10 is not in the graph.
```

Degree centrality

```
In [5]: nx.degree_centrality(G)
Out[5]:
{1: 1.0,
 2: 0.125,
 3: 0.125,
 4: 0.125,
 5: 0.125,
 6: 0.125,
 7: 0.125,
 8: 0.125,
 9: 0.125}
```

1)
Compute number of neighbors for each node

How do you evaluate whether a node is an important one or not? There are a few ways to do so, and here, you're going to look at one metric: the number of neighbors that a node has.
Every NetworkX graph `G` exposes a `.neighbors(n)` method that returns a list of nodes that are the neighbors of the node `n`. To begin, use this method in the IPython Shell on the Twitter network `T` to get the neighbors of of node `1`. This will get you familiar with how the function works. Then, your job in this exercise is to write a function that returns all nodes that have `m` neighbors.

- Write a function called `nodes_with_m_nbrs()` that has two parameters - `G`and `m` - and returns all nodes that have `m` neighbors. To do this:
  - Iterate over all nodes in `G` (**not** including the metadata).
  - Use the `len()` function together with the `.neighbors()` method to calculate the total number of neighbors that node `n` in graph `G` has.
    - If the number of neighbors of node `n` is equal to `m`, add `n` to the set `nodes` using the `.add()` method.
  - After iterating over all the nodes in `G`, return the set `nodes`.
- Use your `nodes_with_m_nbrs()` function to retrieve all the nodes that have 6 neighbors in the graph `T`

```
# Define nodes_with_m_nbrs()
def nodes_with_m_nbrs(G,m):
    """
    Returns all nodes in graph G that have m neighbors.
    """
    nodes = set()

    # Iterate over all nodes in G
    for n in G.nodes():

        # Check if the number of neighbors of n matches m
        if len(G.neighbors(n)) == m:

            # Add the node n to the set
            nodes.add(n)

    # Return the nodes with m neighbors
    return nodes

# Compute and print all nodes in T that have 6 neighbors
six_nbrs = nodes_with_m_nbrs(T,6)
print(six_nbrs)
```

{22533, 1803, 11276, 11279, 6161, 4261, 10149, 3880, 16681, 5420, 14898, 64, 14539, 6862, 20430, 9689, 475, 1374, 6112, 9186, 17762, 14956, 2927, 11764, 4725}

2)
Compute degree distribution

The number of neighbors that a node has is called its "degree", and it's possible to compute the degree distribution across the entire graph. In this exercise, your job is to compute the degree distribution across `T`.

- Use a list comprehension along with the `.neighbors(n)` method to get the degree of every node. The result should be a list of integers.
  - Use `n` as your *iterator variable*.
  - The *output expression* of your list comprehension should be the number of neighbors that node `n` has - that is, its degree. Use the `len()` function together with the `.neighbors()` method to compute this.
  - The *iterable* in your list comprehension is the all the nodes in `T`, accessed using the `.nodes()` method.
- Print the degrees.

```
# Compute the degree of every node: degrees
degrees = [len(T.neighbors(n)) for n in T.nodes()]

# Print the degrees
print (degrees)
```

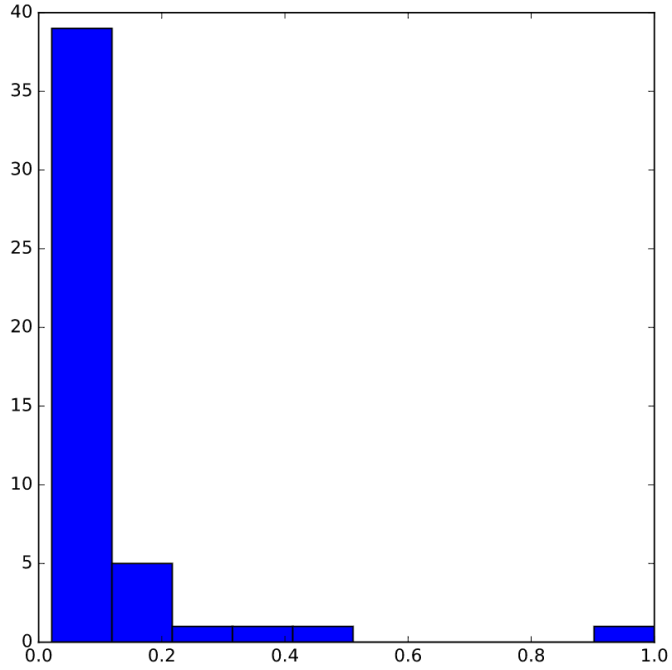
[47, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 17, 0, 10, 27, 0, 0, 0, 0, 0, 0, 0, 0, 34, 0, 0, 0, 0, 0, 60, 0, 11, 4, 0, 12, 0, 0, 56, 53, 0, 30, 0, 0, 0, 0, 12, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

3)
Degree centrality distribution

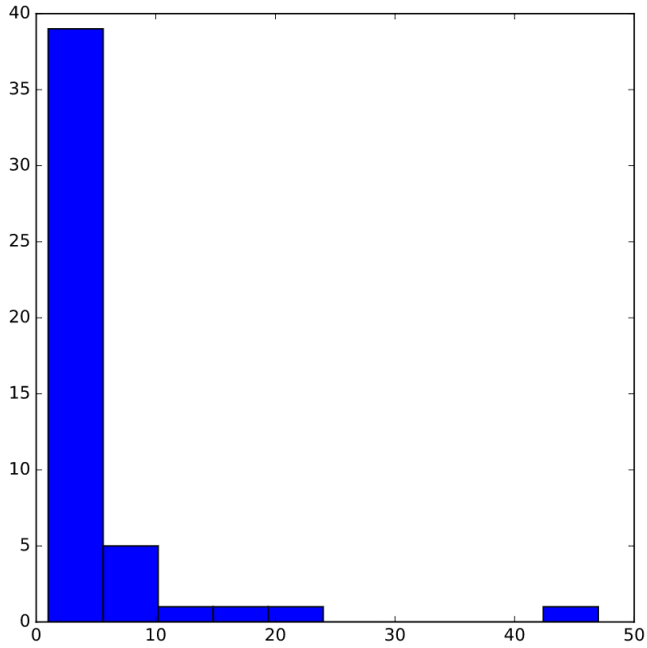
The degree of a node is the number of neighbors that it has. The degree centrality is the number of neighbors divided by all possible neighbors that it could have. Depending on whether self-loops are allowed, the set of possible neighbors a node could have could also include the node itself.
The `nx.degree_centrality(G)` function returns a dictionary, where the keys are the nodes and the values are their degree centrality values.
The degree distribution `degrees` you computed in the previous exercise using the list comprehension has been pre-loaded.

- Compute the degree centrality of the Twitter network `T`.
- Using `plt.hist()`, plot a histogram of the degree centrality distribution of `T`. This can be accessed using `list(deg_cent.values())`.
- Plot a histogram of the degree distribution `degrees` of `T`. This is the same list you computed in the last exercise.
- Create a scatter plot with `degrees` on the x-axis and the degree centrality distribution `list(deg_cent.values())` on the y-axis.

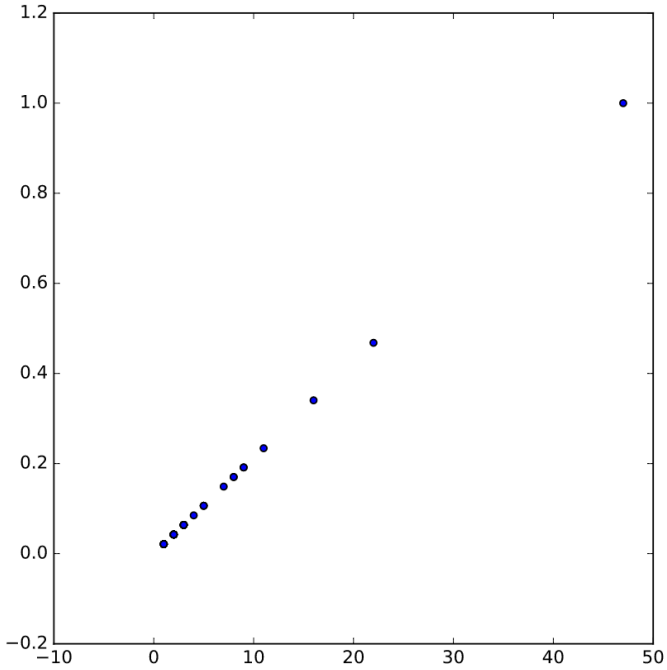
```
# Import matplotlib.pyplot
import matplotlib.pyplot as plt

# Compute the degree centrality of the Twitter network: deg_cent
deg_cent = nx.degree_centrality(T)

# Plot a histogram of the degree centrality distribution of the graph.
plt.figure()
plt.hist(list(deg_cent.values()))
plt.show()
```




```
# Plot a histogram of the degree distribution of the graph
plt.figure()
plt.hist(degrees)
plt.show()
```




```
# Plot a scatter plot of the centrality distribution and the degree distribution
plt.figure()
plt.scatter(degrees,list(deg_cent.values()))
plt.show()
```



4)

Shortest Path I

You can leverage what you know about finding neighbors to try finding paths in a network. One algorithm for path-finding between two nodes is the "breadth-first search" (BFS) algorithm. In a BFS algorithm, you start from a particular node and

iteratively search through its neighbors and neighbors' neighbors until you find the destination node.
Pathfinding algorithms are important because they provide another way of assessing node importance; you'll see this in a later exercise.
In this set of 3 exercises, you're going to build up slowly to get to the final BFS algorithm. The problem has been broken into 3 parts that, if you complete in succession, will get you to a first pass implementation of the BFS algorithm.

- Create a function called `path_exists()` that has 3 parameters - `G`, `node1`, and `node2` - and returns whether or not a path exists between the two nodes.
- Initialize the queue of cells to visit with the first node, `node1`. `queue` should be a list`.
- Iterate over the nodes in `queue`.
- Get the neighbors of the node using the `.neighbors()` method of the graph `G`.
- Check to see if the destination node `node2` is in the set of `neighbors`. If it is, return `True`.

```
# Define path_exists()
def path_exists(G,node1,node2):
    """
    This function checks whether a path exists between two nodes (node1, node2) in graph G.
    """
    visited_nodes = set()

    # Initialize the queue of cells to visit with the first node: queue
    queue = [node1]

    # Iterate over the nodes in the queue
    for node in queue:

        # Get neighbors of the node
        neighbors = G.neighbors(node)

        # Check to see if the destination node is in the set of neighbors
        if node2 in neighbors:
            print('Path exists between nodes {0} and {1}'.format(node1, node2))
            return True
            break
```

5)
Shortest Path II

Now that you've got the code for checking whether the destination node is present in neighbors, next up, you're going to extend the same function to write the code for the condition where the destination node is **not** present in the neighbors.
All the code you need to write is in the `else` condition; that is,
if `node2` is *not* in `neighbors`.

- Using the `.add()` method, add the current node `node` to the set `visited_nodes` to keep track of what nodes have already been visited.
- Add the *neighbors* of the current node `node` that have not yet been visited to `queue`. To do this, you'll need to use the `.extend()` method of `queue`together with a list comprehension. The `.extend()` method appends all the items in a given list.
    - The *output expression* and *iterator variable* of the list comprehension are both `n`. The *iterable* is the list `neighbors`, and the conditional is if `n` is *not* in the visited nodes.

```
def path_exists(G, node1, node2):
    """
    This function checks whether a path exists between two nodes (node1, node2) in graph G.
    """
    visited_nodes = set()
    queue = [node1]

    for node in queue:
        neighbors = G.neighbors(node)
        if node2 in neighbors:
            print('Path exists between nodes {0} and {1}'.format(node1, node2))
            return True
            break

        else:
            # Add current node to visited nodes
            visited_nodes.add(node)

            # Add neighbors of current node that have not yet been visited
            queue.extend([n for n in neighbors if n not in visited_nodes])
```

6)
Shortest Path III

This is the final exercise of this trio! You're now going to complete the problem by writing the code that returns `False` if there's no path between two nodes.

- Check to see if the queue has been emptied. You can do this by inspecting the last element of queue with `[-1]`.
- Place the appropriate return statement for indicating whether there's a path between these two nodes.

```
def path_exists(G, node1, node2):
    """
    This function checks whether a path exists between two nodes (node1, node2) in graph G.
    """
    visited_nodes = set()
    queue = [node1]

    for node in queue:
        neighbors = G.neighbors(node)
        if node2 in neighbors:
            print('Path exists between nodes {0} and {1}'.format(node1, node2))
            return True
            break

        else:
            visited_nodes.add(node)
            queue.extend([n for n in neighbors if n not in visited_nodes])

        # Check to see if the final element of the queue has been reached
        if node == queue[-1]:
            print('Path does not exist between nodes {0} and {1}'.format(node1, node2))

            # Place the appropriate return statement
            return False
```

Chapter 3)

Note)

Betweenness centrality

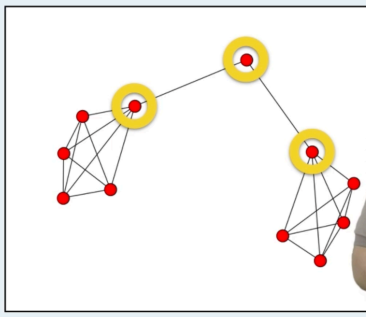
- Definition:

$$\frac{\text{num. shortest paths through node}}{\text{all possible shortest paths}}$$

- Application:
  - Bridges between liberal- and conservative-leaning Twitter users
  - Critical information transfer links

Betweenness centrality

```
In [5]: import networkx as nx

In [6]: G = nx.barbell_graph(m1=5, m2=1)

In [10]: nx.betweenness_centrality(G)
Out[10]:
{0: 0.0,
 1: 0.0,
 2: 0.0,
 3: 0.0,
 4: 0.5333333333333333,
 5: 0.5555555555555556,
 6: 0.5333333333333333,
 7: 0.0,
 8: 0.0,
 9: 0.0,
 10: 0.0}
```

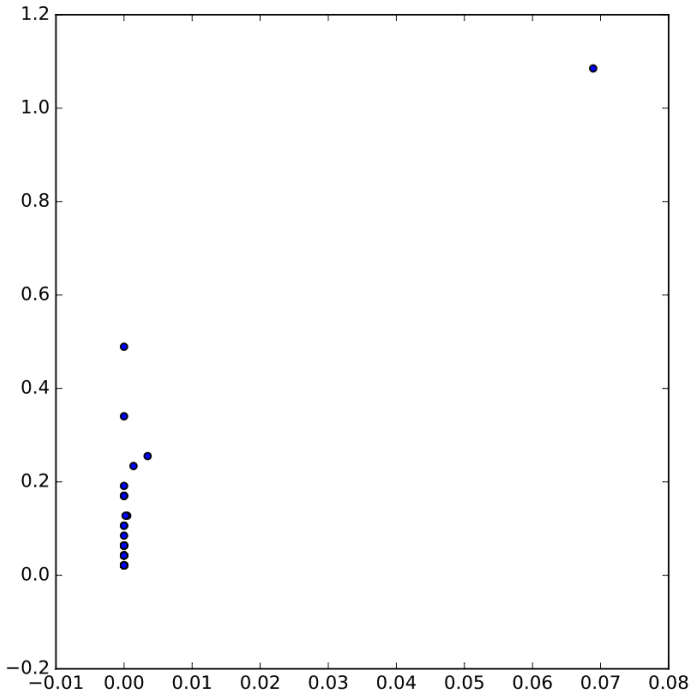


1)
NetworkX betweenness centrality on a social network

Betweenness centrality is a node importance metric that uses information about the shortest paths in a network. It is defined as the fraction of all possible shortest paths between any pair of nodes that pass through the node.
NetworkX provides the `nx.betweenness_centrality(G)` function for computing the betweenness centrality of every node in a graph, and it returns a dictionary where the keys are the nodes and the values are their betweenness centrality measures.

- Compute the betweenness centrality `bet_cen` of the nodes in the graph `T`.
- Compute the degree centrality `deg_cen` of the nodes in the graph `T`.
- Compare betweenness centrality to degree centrality by creating a scatterplot of the two, with `list(bet_cen.values())` on the x-axis and `list(deg_cen.values())` on the y-axis.

```
# Compute the betweenness centrality of T: bet_cen
bet_cen = nx.betweenness_centrality(T)

# Compute the degree centrality of T: deg_cen
deg_cen = nx.degree_centrality(T)

# Create a scatter plot of betweenness centrality and degree centrality
plt.scatter(list(bet_cen.values()),list(deg_cen.values()))

# Display the plot
plt.show()
```



2)

Deep dive - Twitter network

You're going to now take a deep dive into a Twitter network, which will help reinforce what you've learned earlier. First, you're going to find the nodes that can broadcast messages very efficiently to lots of people one degree of separation away.
NetworkX has been pre-imported for you as `nx`.

- Write a function `find_nodes_with_highest_deg_cent(G)` that returns the node(s) with the highest degree centrality using the following steps:
  - Compute the degree centrality of `G`.
  - Compute the maximum degree centrality using the `max()` function on`list(deg_cent.values())`.
  - Iterate over the degree centrality dictionary, `deg_cen.items()`.
  - If the degree centrality value `v` of the current node `k` is equal to `max_dc`, add it to the set of nodes.
- Use your function to find the node(s) that has the highest degree centrality in `T`.

- Write an assertion statement that checks that the node(s) is/are correctly identified. This has been done for you, so hit 'Submit Answer' to see the result!

```
# Define find_nodes_with_highest_deg_cent()
def find_nodes_with_highest_deg_cent(G):

    # Compute the degree centrality of G: deg_cent
```

```
    deg_cent = nx.degree_centrality(G)

    # Compute the maximum degree centrality: max_dc
    max_dc = max(list(deg_cent.values()))

    nodes = set()

    # Iterate over the degree centrality dictionary
    for k, v in deg_cent.items():

        # Check if the current value has the maximum degree centrality
        if v == max_dc:

            # Add the current node to the set of nodes
            nodes.add(k)

    return nodes

# Find the node(s) that has the highest degree centrality in T: top_dc
top_dc = find_nodes_with_highest_deg_cent(T)
print(top_dc)

# Write the assertion statement
for node in top_dc:
    assert nx.degree_centrality(T)[node] ==
max(nx.degree_centrality(T).values())
```

{11824}

3)

Deep dive - Twitter network part II

Next, you're going to do an analogous deep dive on betweenness centrality! Just a few hints to help you along: remember that betweenness centrality is computed using `nx.betweenness_centrality(G)`.

- Write a function `find_node_with_highest_bet_cent(G)` that returns the node(s) with the highest betweenness centrality.
    - Compute the betweenness centrality of `G`.
    - Compute the maximum betweenness centrality using the `max()` function on `list(bet_cent.values())`.
    - Iterate over the degree centrality dictionary, `bet_cent.items()`.
    - If the degree centrality value `v` of the current node `k` is equal to `max_bc`, add it to the set of nodes.

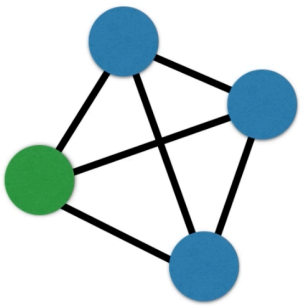
- Use your function to find the node(s) that has the highest betweenness centrality in `T`.

```
# Define find_node_with_highest_bet_cent()
def find_node_with_highest_bet_cent(G):

    # Compute betweenness centrality: bet_cent
    bet_cent = nx.betweenness_centrality(G)

    # Compute maximum betweenness centrality: max_bc
    max_bc = max(list(bet_cent.values()))

    nodes = set()

    # Iterate over the betweenness centrality dictionary
    for k, v in bet_cent.items():

        # Check if the current value has the maximum betweenness centrality
        if v == max_bc:

            # Add the current node to the set of nodes
            nodes.add(k)

    return nodes

# Use that function to find the node(s) that has the highest betweenness
centrality in the network: top_bc
top_bc = find_node_with_highest_bet_cent(T)

# Write an assertion statement that checks that the node(s) is/are correctly
identified.
for node in top_bc:
    assert nx.betweenness_centrality(T)[node] ==
max(nx.betweenness_centrality(T).values())
```

Note)

Cliques

- Social cliques: tightly-knit groups
- Network cliques: completely connected graphs



Clique code

```
In [1]: G
Out[1]: <networkx.classes.graph.Graph at 0x10c99ecf8>

In [2]: from itertools import combinations

In [3]: for n1, n2 in combinations(G.nodes(), 2):
   ...:     print(n1, n2)
0 1
0 2
0 3
0 4
0 5
...
...
```

4)

Identifying triangle relationships

Now that you've learned about cliques, it's time to try leveraging what you know to find structures in a network. Triangles are what you'll go for first. We may be interested in triangles because they're the simplest complex clique. Let's write a few functions; these exercises will bring you through the fundamental logic behind network algorithms.

In the Twitter network, each node has an `'occupation'` label associated with it, in which the Twitter user's work occupation is divided into `celebrity`, `politician` and `scientist`. One potential application of triangle-finding algorithms is to find out whether users that have similar occupations are more likely to be in a clique with one another.

- Import `combinations` from `itertools`.
- Write a function `is_in_triangle()` that has two parameters - `G` and `n` - and checks whether a given node is in a triangle relationship or not.
    - `combinations(iterable, n)` returns combinations of size `n` from `iterable`. This will be useful here, as you want combinations of size `2`from `G.neighbors(n)`.
    - To check whether an edge exists between two nodes, use the `.has_edge(node1, node2)` method. If an edge exists, then the given node is in a triangle relationship, and you should return `True`.

```
from itertools import combinations

# Define is_in_triangle()
def is_in_triangle(G, n):
    """
    Checks whether a node `n` in graph `G` is in a triangle relationship or not.

    Returns a boolean.
    """
    in_triangle = False

    # Iterate over all possible triangle relationship combinations
    for n1, n2 in combinations(G.neighbors(n),2):

        # Check if an edge exists between n1 and n2
        if G.has_edge(n1,n2):
            in_triangle = True
            break
    return in_triangle
```

5)
Finding nodes involved in triangles

NetworkX provides an API for counting the number of triangles that every node is involved in: `nx.triangles(G)`. It returns a dictionary of nodes as the keys and number of triangles as the values. Your job in this exercise is to modify the function defined earlier to extract all of the nodes involved in a triangle relationship with a given node.

- Write a function `nodes_in_triangle()` that has two parameters - `G` and `n`- and identifies all nodes in a triangle relationship with a given node.
    - In the `for` loop, iterate over all possible triangle relationship combinations.
    - Check whether the nodes `n1` and `n2` have an edge between them. If they do, add both nodes to the set `triangle_nodes`.

- Use your function in an `assert` statement to check that the number of nodes involved in a triangle relationship with node `1` of graph `T` is equal to `35`

```
from itertools import combinations

# Write a function that identifies all nodes in a triangle relationship with a given node.
def nodes_in_triangle(G, n):
    """
    Returns the nodes in a graph `G` that are involved in a triangle relationship with the node `n`.
    """
    triangle_nodes = set([n])

    # Iterate over all possible triangle relationship combinations
    for n1, n2 in combinations(G.neighbors(n),2):

        # Check if n1 and n2 have an edge between them
        if G.has_edge(n1,n2):

            # Add n1 to triangle_nodes
            triangle_nodes.add(n1)

            # Add n2 to triangle_nodes
            triangle_nodes.add(n2)

    return triangle_nodes

# Write the assertion statement
assert len(nodes_in_triangle(T, 1)) == 35
```

6)
Finding open triangles

Let us now move on to finding open triangles! Recall that they form the basis of friend recommendation systems; if "A" knows "B" and "A" knows "C", then it's probable that "B" also knows "C".

- Write a function `node_in_open_triangle()` that has two parameters - `G`and `n` - and identifies whether a node is present in an open triangle with its neighbors.
    - In the `for` loop, iterate over all possible triangle relationship combinations.

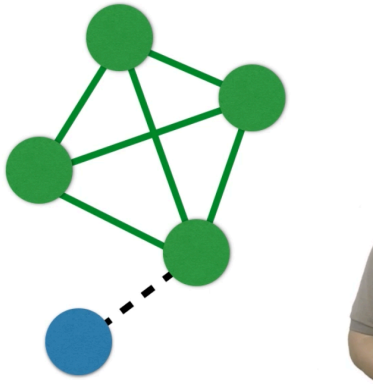
    - If the nodes `n1` and `n2` do *not* have an edge between them, set `in_open_triangle` to `True`, break out from the `if` statement and return `in_open_triangle`.
- Use this function to count the number of open triangles that exist in `T`.
    - In the `for` loop, iterate over all the nodes in `T`.
    - If the current node `n` is in an open triangle, increment `num_open_triangles`.

```
from itertools import combinations

# Define node_in_open_triangle()
def node_in_open_triangle(G, n):
    """
    Checks whether pairs of neighbors of node `n` in graph `G` are in an 'open triangle' relationship with node `n`.
    """
    in_open_triangle = False

    # Iterate over all possible triangle relationship combinations
    for n1, n2 in combinations(G.neighbors(n),2):

        # Check if n1 and n2 do NOT have an edge between them
        if not G.has_edge(n1,n2):

            in_open_triangle = True

            break

    return in_open_triangle

# Compute the number of open triangles in T
num_open_triangles = 0

# Iterate over all the nodes in T
for n in T.nodes():

    # Check if the current node is in an open triangle
    if node_in_open_triangle(T,n):

        # Increment num_open_triangles
        num_open_triangles += 1
print(num_open_triangles)
```

Note)

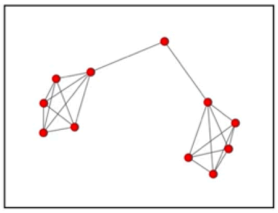
Maximal cliques

- Definition: a clique that, when extended by one node is no longer a clique



Maximal cliques

```
In [1]: import networkx as nx

In [2]: G = nx.barbell_graph(m1=5, m2=1)

In [3]: nx.find_cliques(G)
Out[3]: <generator object find_cliques at 0x1043f1f68>

In [4]: list(nx.find_cliques(G))
Out[4]: [[4, 0, 1, 2, 3], [4, 5], [6, 8, 9, 10, 7], [6, 5]]
```



7)

Finding all maximal cliques of size "n"

Now that you've explored triangles (and open triangles), let's move on to the concept of maximal cliques. Maximal cliques are cliques that cannot be extended by adding an adjacent edge, and are a useful property of the graph when finding communities. NetworkX provides a function that allows you to identify the nodes involved in each maximal clique in a graph: `nx.find_cliques(G)`. Play around with the function by using it on `T` in the IPython Shell, and then try answering the exercise.

- Write a function `maximal_cliques()` that has two parameters - `G` and `size` - and finds all maximal cliques of size `n`.

- In the `for` loop, iterate over all the cliques in `G` using the `nx.find_cliques()` function.
- If the current clique is of size `size`, append it to the list `mcs`.

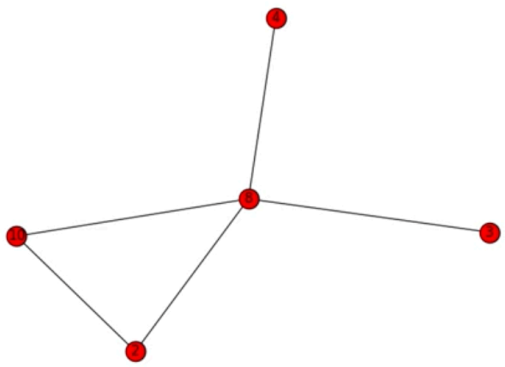
- Use an assert statement and your `maximal_cliques()` function to check that there are `33` maximal cliques of size `3` in the graph `T`.

```
# Define maximal_cliques()
def maximal_cliques(G,size):
    """
    Finds all maximal cliques in graph `G` that are of size `size`.
    """
    mcs = []
    for clique in list(nx.find_cliques(G)):
        if len(clique) == size:
            mcs.append(clique)
    return mcs

# Check that there are 33 maximal cliques of size 3 in the graph T
assert len(maximal_cliques(T,3)) == 33
```

Note)

Subgraphs

```
In [11]: nx.draw(G_eight, with_labels=True)
```



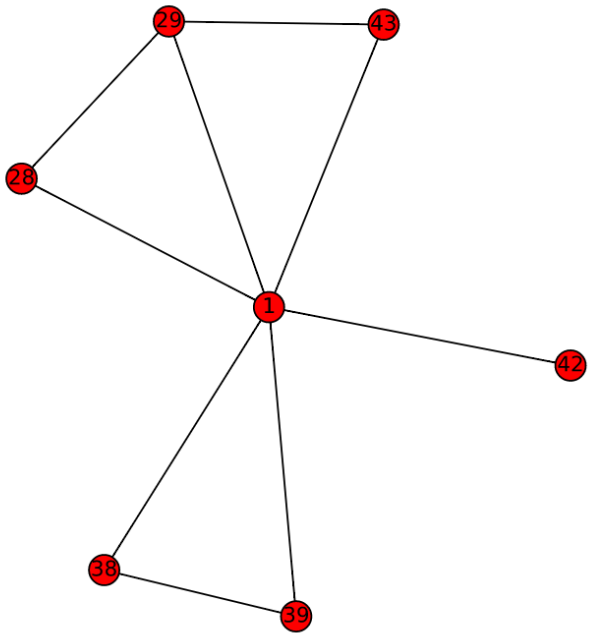
8)
Subgraphs I

There may be times when you just want to analyze a subset of nodes in a network. To do so, you can copy them out into another graph object using `G.subgraph(nodes)`, which returns a new `graph` object (of the same type as the original graph) that is comprised of the iterable of `nodes` that was passed in. `matplotlib.pyplot` has been imported for you as `plt`.

- Write a function `get_nodes_and_nbrs(G, nodes_of_interest)` that extracts the subgraph from graph `G` comprised of the `nodes_of_interest` and their neighbors.
    - In the first `for` loop, iterate over `nodes_of_interest` and append the current node `n` to `nodes_to_draw`.
    - In the second `for` loop, iterate over the neighbors of `n`, and append all the neighbors `nbr` to `nodes_to_draw`.
- Use the function to extract the subgraph from `T` comprised of nodes 29, 38, and 42 (contained in the pre-defined list `nodes_of_interest`) and their neighbors. Save the result as `T_draw`.
- Draw the subgraph `T_draw` to the screen.

```
nodes_of_interest = [29, 38, 42]

# Define get_nodes_and_nbrs()
def get_nodes_and_nbrs(G, nodes_of_interest):
    """
    Returns a subgraph of the graph `G` with only the `nodes_of_interest` and their neighbors.
    """
    nodes_to_draw = []

    # Iterate over the nodes of interest
    for n in nodes_of_interest:

        # Append the nodes of interest to nodes_to_draw
        nodes_to_draw.append(n)

        # Iterate over all the neighbors of node n
        for nbr in G.neighbors(n):

            # Append the neighbors of n to nodes_to_draw
            nodes_to_draw.append(nbr)

    return G.subgraph(nodes_to_draw)
```

```
# Extract the subgraph with the nodes of interest: T_draw
T_draw = get_nodes_and_nbrs(T,nodes_of_interest)

# Draw the subgraph to the screen
nx.draw(T_draw,with_labels=True)
plt.show()
```



9)

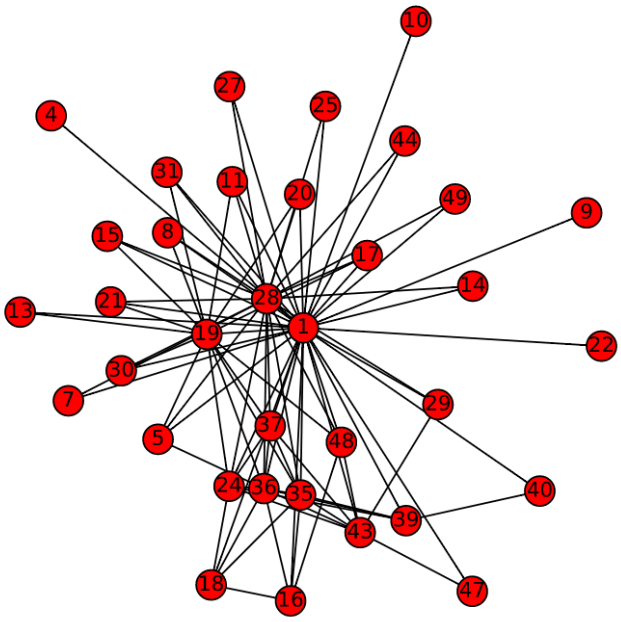
Subgraphs II

In the previous exercise, we gave you a list of nodes whose neighbors we asked you to extract.
Let's try one more exercise in which you extract nodes that have a particular metadata property and their neighbors. This should hark back to what you've learned about using list comprehensions to find nodes. The exercise will also build your capacity to compose functions that you've already written before.

- Using a list comprehension, extract nodes that have the metadata `'occupation'` as `'celebrity'` alongside their neighbors:
  - The *output expression* of the list comprehension is `n`, and there are two *iterator variables*: `n` and `d`. The *iterable* is the list of nodes of `T`(including the metadata, which you can specify using `data=True`) and the conditional expression is if the `'occupation'` key of the metadata dictionary `d` equals `'celebrity'`.
- Place them in a new subgraph called `T_sub`. To do this:

    - Iterate over the nodes, compute the neighbors of each node, and add them to the set of nodes `nodeset` by using the `.union()` method. This last part has been done for you.
    - Use `nodeset` along with the `T.subgraph()` method to calculate `T_sub`.
- Draw `T_sub` to the screen.

```
# Extract the nodes of interest: nodes
nodes = [n for n, d in T.nodes(data=True) if d['occupation'] == 'celebrity']

# Create the set of nodes: nodeset
nodeset = set(nodes)

# Iterate over nodes
for n in nodes:

    # Compute the neighbors of n: nbrs
    nbrs = T.neighbors(n)

    # Compute the union of nodeset and nbrs: nodeset
    nodeset = nodeset.union(nbrs)

# Compute the subgraph using nodeset: T_sub
T_sub = T.subgraph(nodeset)

# Draw T_sub to the screen
nx.draw(T_sub,with_labels=True)
plt.show()
```



Chapter 4)

Note)

Data

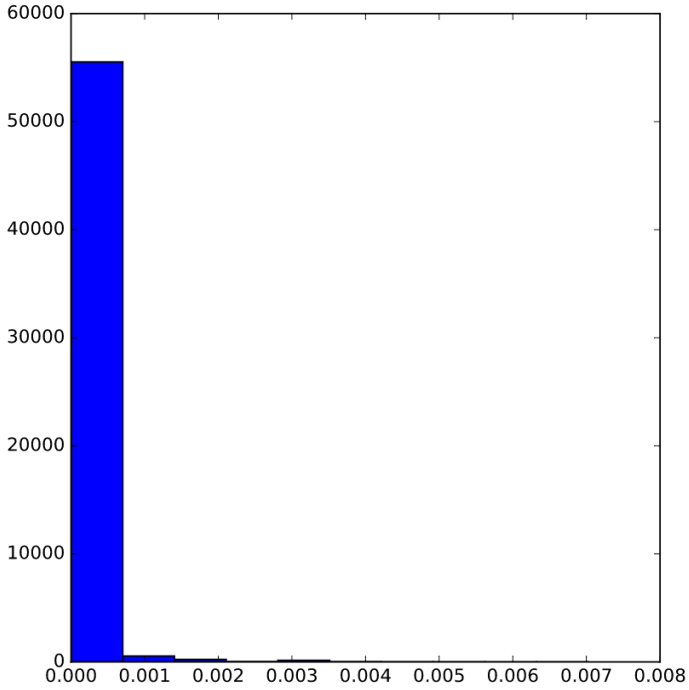
- Github user collaboration network
- Nodes: users
- Edges: collaboration on same GitHub repository
- Goals:
  - Analyze structure
  - Visualize
  - Build simple recommendation system

Graph properties

```
In [1]: import networkx as nx

In [2]: G = nx.erdos_renyi_graph(n=20, p=0.2)

In [3]: len(G.edges())
Out[3]: 29

In [4]: len(G.nodes())
Out[4]: 20
```

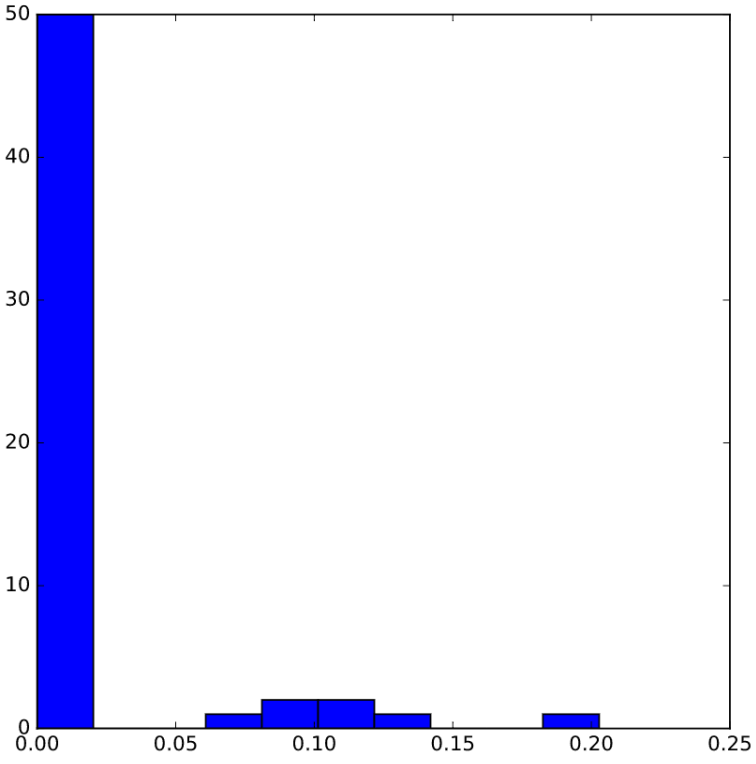
1)

```
# Import necessary modules
import matplotlib.pyplot as plt
import networkx as nx

# Plot the degree distribution of the GitHub collaboration network
plt.hist(list(nx.degree_centrality(G).values()))
plt.show()
```



2) Betweenness_centrality

```
# Import necessary modules
import matplotlib.pyplot as plt
import networkx as nx

# Plot the degree distribution of the GitHub collaboration network
plt.hist(list(nx.betweenness_centrality(G).values()))
plt.show()
```



Note)

NetworkX API

```
In [1]: import networkx as nx

In [2]: G = nx.erdos_renyi_graph(n=100, p=0.03)

In [3]: nx.connected_component_subgraphs(G)
Out[3]: <generator object connected_component_subgraphs at
0x10cb2c990>

In [4]: list(nx.connected_component_subgraphs(G))
Out[4]:
[<networkx.classes.graph.Graph at 0x10ca24588>,
 <networkx.classes.graph.Graph at 0x10ca244e0>]

In [5]: for g in list(nx.connected_component_subgraphs(G)):
   ...:     print(len(g.nodes()))
   ...:
Out[5]: 99
   ...: 1
```

3)

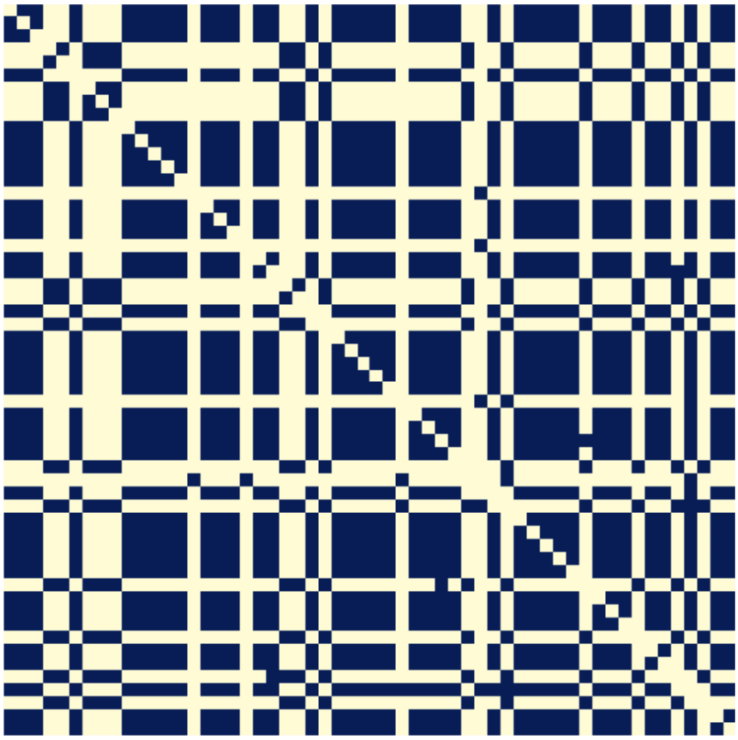
MatrixPlot

Let's now practice making some visualizations. The first one will be the MatrixPlot. In a MatrixPlot, the matrix is the representation of the edges.

- Make a MatrixPlot visualization of the largest connected component subgraph, with authors grouped by their user group number.
    - First, calculate the largest connected component subgraph by using the `nx.connected_component_subgraphs(G)` inside the provided `sorted()`function. Python's built-in **`sorted()` function** takes an iterable and returns a sorted list (in ascending order, by default). Therefore, to access the largest connected component subgraph, the statement is sliced with `[-1]`.
    - Create the `MatrixPlot` object `h`. You have to specify the parameters `graph` and `node_grouping` to be the largest connected component subgraph and `'grouping'`, respectively.
    - Draw the `MatrixPlot` object to the screen and display the plot.

```
# Import necessary modules
from nxviz import MatrixPlot
import matplotlib.pyplot as plt

# Calculate the largest connected component subgraph: largest_ccs
largest_ccs = sorted(nx.connected_component_subgraphs(G), key=lambda x:
len(x))[-1]
```

```
# Create the customized MatrixPlot object: h
h = MatrixPlot(graph=largest_ccs,node_grouping='grouping')

# Draw the MatrixPlot to the screen
h.draw()
plt.show()
```

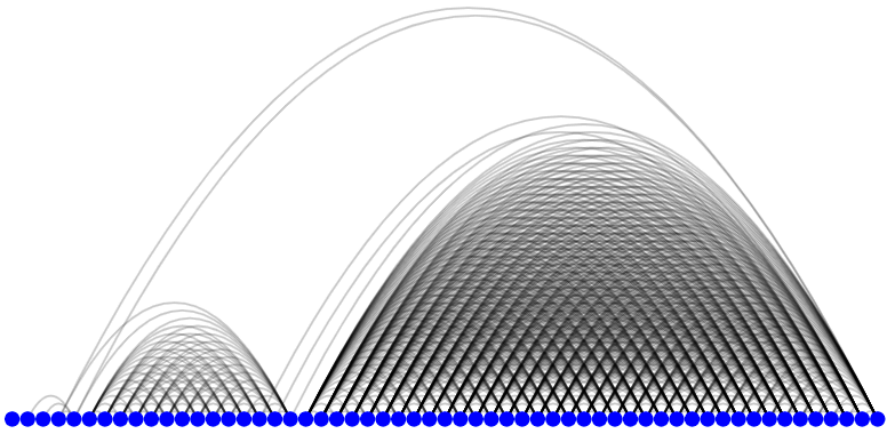


4)
ArcPlot

Next up, let's use the ArcPlot to visualize the network. You're going to practice sorting the nodes in the graph as well.
Note: this exercise may take about 4-7 seconds to execute if done correctly.

- Make an ArcPlot of the GitHub collaboration network, with authors sorted by degree. To do this:
  - Iterate over all the nodes in `G`, including the metadata (by specifying `data=True`).
  - In each iteration of the loop, calculate the degree of each node `n` with `nx.degree()` and set its `'degree'` attribute. `nx.degree()` accepts two arguments: A graph and a node.
  - Create the `ArcPlot` object `a` by specifying two parameters: the `graph`, which is `G`, and the `node_order`, which is `'degree'`, so that the nodes are sorted.
  - Draw the `ArcPlot` object to the screen.

```
# Import necessary modules
from nxviz.plots import ArcPlot
import matplotlib.pyplot as plt

# Iterate over all the nodes in G, including the metadata
for n, d in G.nodes(data=True):

    # Calculate the degree of each node: G.node[n]['degree']
    G.node[n]['degree'] = nx.degree(G,n)

# Create the ArcPlot object: a
a = ArcPlot(graph=G,node_order='degree')

# Draw the ArcPlot to the screen
a.draw()
plt.show()
```

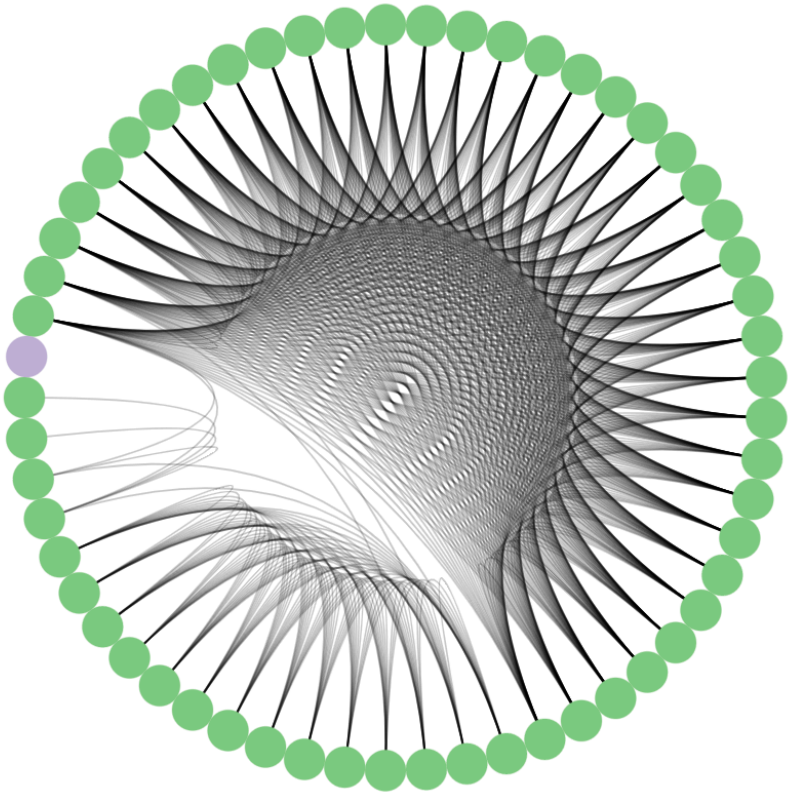


5)
ircosPlot

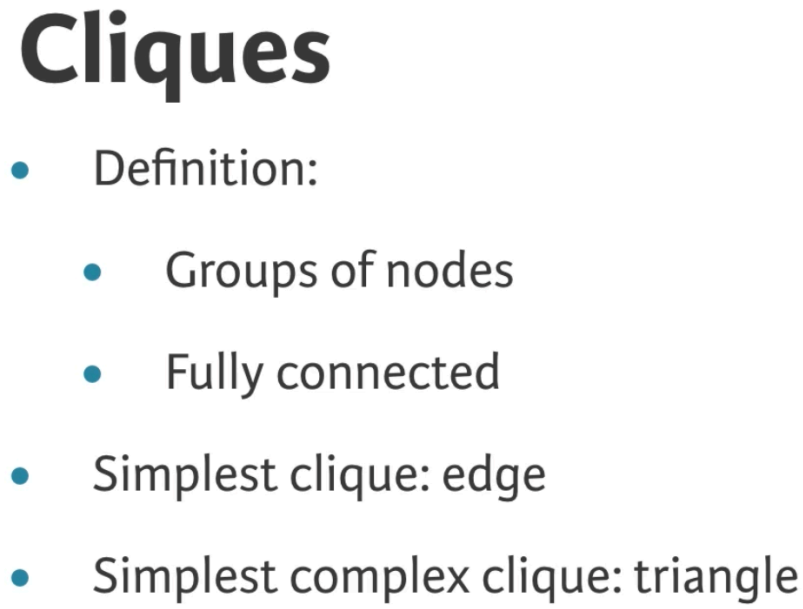
Finally, you're going to make a CircosPlot of the network!

- Make a CircosPlot of the network, again, with GitHub users sorted by their degree, and grouped and coloured by their `'grouping'` key. To do this:
    - Iterate over all the nodes in `G`, including the metadata (by specifying `data=True`).
    - In each iteration of the loop, calculate the degree of each node `n` with `nx.degree()` and set its `'degree'` attribute.
    - Create the `CircosPlot` object `c` by specifying three parameters in addition to the graph `G`: the `node_order`, which is `'degree'`, the `node_grouping` and the `node_color`, which are both `'grouping'`.
    - Draw the `CircosPlot` object to the screen.

```
# Import necessary modules
from nxviz import CircosPlot
import matplotlib.pyplot as plt

# Iterate over all the nodes, including the metadata
for n, d in G.nodes(data=True):

    # Calculate the degree of each node: G.node[n]['degree']
    G.node[n]['degree'] = nx.degree(G,n)

# Create the CircosPlot object: c
c =
CircosPlot(graph=G,node_order='degree',node_grouping='grouping',node_color
='grouping')

# Draw the CircosPlot object to the screen
c.draw()
plt.show()
```



Note)



6)
Finding cliques (I)

You're now going to practice finding cliques in `G`. Recall that cliques are "groups of nodes that are fully connected to one another", while a maximal clique is a clique that cannot be extended by adding another node in the graph.

- Count the number of maximal cliques present in the graph and print it.
  - Use the `nx.find_cliques()` function of `G` to find the maximal cliques.
  - The `nx.find_cliques()` function returns a generator object. To count the number of maximal cliques, you need to first convert it to a list with `list()` and then use the `len()` function. Place this inside a `print()`function to print it.

```
# Calculate the maximal cliques in G: cliques
cliques = nx.find_cliques(G)

# Count and print the number of maximal cliques in G
print(len(list(cliques)))
```

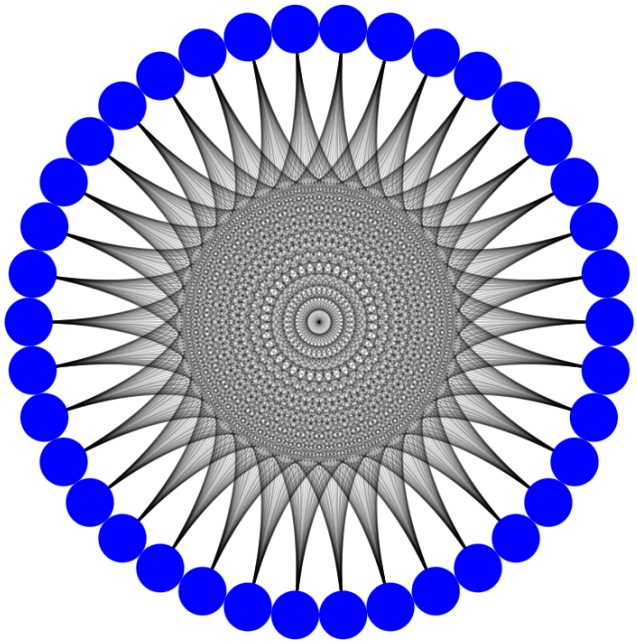
```
8
```

7)
Finding cliques (II)

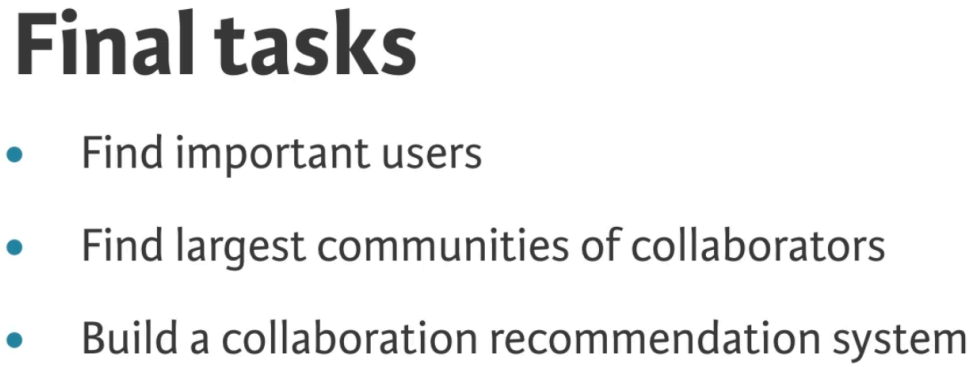
Great work! Let's continue by finding a particular maximal clique, and then plotting that clique.

- Find the author(s) that are part of the largest maximal clique, and plot the subgraph of that/one of those clique(s) using a CircosPlot. To do this:
    - Use the `nx.find_cliques()` function to calculate the maximal cliques in `G`. Place this within the provided `sorted()` function to calculate the *largest* maximal clique.
    - Create the subgraph consisting of the largest maximal clique using the `.subgraph()` method and `largest_clique`.
    - Create the `CircosPlot` object using the subgraph `G_lc` (without any other arguments) and plot it.

```
# Import necessary modules
import networkx as nx
from nxviz import CircosPlot
import matplotlib.pyplot as plt

# Find the author(s) that are part of the largest maximal clique: largest_clique
largest_clique = sorted(nx.find_cliques(G), key=lambda x:len(x))[-1]

# Create the subgraph of the largest_clique: G_lc
G_lc = G.subgraph(largest_clique)

# Create the CircosPlot object: c
c = CircosPlot(G_lc)

# Draw the CircosPlot to the screen
c.draw()
plt.show()
```



Note)



10)

Finding important collaborators

Almost there! You'll now look at important nodes once more. Here, you'll make use of the `degree_centrality()` and `betweenness_centrality()` functions in NetworkX to compute each of the respective centrality scores, and then use that information to find the "important nodes". In other words, your job in this exercise is to find the user(s) that have collaborated with the most number of users.

- Compute the degree centralities of `G`. Store the result as `deg_cent`.
- Compute the maximum degree centrality. Since `deg_cent` is a dictionary, you'll have to use the `.values()` method to get a list of its values before computing the maximum degree centrality with `max()`.

- Identify the most prolific collaborators using a list comprehension:
  - Iterate over the degree centrality dictionary `deg_cent` that was computed earlier using its `.items()` method. What condition should be satisfied if you are seeking to find user(s) that have collaborated with the most number of users? *Hint*: It has do to with the maximum degree centrality.

```
# Compute the degree centralities of G: deg_cent
deg_cent = nx.degree_centrality(G)

# Compute the maximum degree centrality: max_dc
max_dc = max(list(deg_cent.values()))

# Find the user(s) that have collaborated the most: prolific_collaborators
prolific_collaborators = [n for n, dc in deg_cent.items() if dc == max_dc]

# Print the most prolific collaborator(s)
print(prolific_collaborators)
```

['u89']

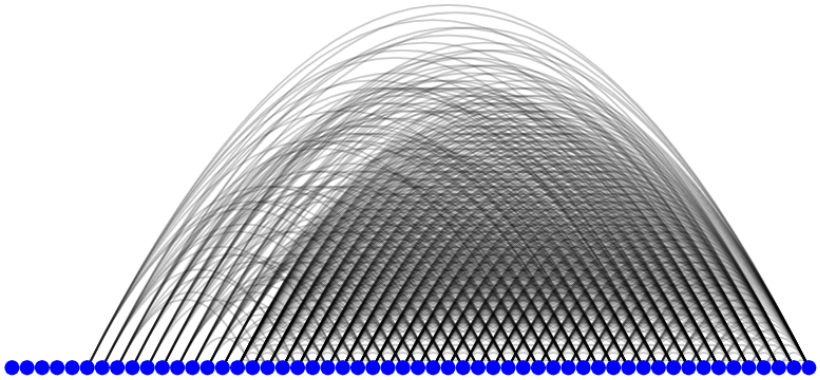
11)
Characterizing editing communities

You're now going to combine what you've learned about the BFS algorithm and concept of maximal cliques to visualize the network with an ArcPlot.
The largest maximal clique in the Github user collaboration network has been assigned to the subgraph `G_lmc`.

- Go out 1 degree of separation from the clique, and add those users to the subgraph. Inside the first `for` loop:
  - Add nodes to `G_lmc` from the neighbors of `G` using the `.add_nodes_from()` method and `.neighbors()` methods.
  - Using the `.add_edges_from()`, method, add edges to `G_lmc` between the current node and all its neighbors. To do this, you'll have create a list of tuples using the `zip()` function consisting of the current node and each of its neighbors. The first argument to `zip()` should be `[node]*len(G.neighbors(node))`, and the second argument should be the neighbors of `node`.
- Record each node's degree centrality score in its node metadata.
  - Do this by assigning `nx.degree_centrality(G_lmc)[n]` to `G_lmc.node[n]['degree centrality']` in the second `for` loop.
- Visualize this network with an ArcPlot sorting the nodes by degree centrality (you can do this using the keyword argument `node_order='degree centrality'`).

```
# Import necessary modules
from nxviz import ArcPlot
import matplotlib.pyplot as plt

# Identify the largest maximal clique: largest_max_clique
largest_max_clique = set(sorted(nx.find_cliques(G), key=lambda x: len(x))[-1])

# Create a subgraph from the largest_max_clique: G_lmc
G_lmc = G.subgraph(largest_max_clique)

# Go out 1 degree of separation
for node in G_lmc.nodes():
    G_lmc.add_nodes_from(G.neighbors(node))
    G_lmc.add_edges_from(zip([node]*len(G.neighbors(node)), node))

# Record each node's degree centrality score
for n in G_lmc.nodes():
```

```
    G_lmc.node[n]['degree centrality'] = nx.degree_centrality(G_lmc)[n]

# Create the ArcPlot object: a
a = ArcPlot(G_lmc,node_order='degree centrality')

# Draw the ArcPlot to the screen
a.draw()
plt.show()
```



12)

Recommending co-editors who have yet to edit together

Finally, you're going to leverage the concept of open triangles to recommend users on GitHub to collaborate!

- Compile a list of GitHub users that should be recommended to collaborate with one another. To do this:
  - In the first `for` loop, iterate over all the nodes in `G`, including the metadata (by specifying `data=True`).
  - In the second `for` loop, iterate over all the possible triangle combinations, which can be identified using the `combinations()` function with a `size`of `2`.
  - If `n1` and `n2` **do not** have an edge between them, a collaboration between these two nodes (users) **should** be recommended, so increment the `(n1), (n2)` value of the `recommended` dictionary in this case. You can check whether or not `n1` and `n2` have an edge between them using the `.has_edge()` method.
- Using a list comprehension, identify the top 10 pairs of users that should be recommended to collaborate. The *iterable* should be the key-value pairs of the `recommended` dictionary (which can be accessed with the `.items()` method), while the conditional should be satisfied if `count` is *greater* than the top 10 in `all_counts`. Note that `all_counts` is sorted in ascending order, so you can access the top 10 with `all_counts[-10]`.

```
# Import necessary modules
from itertools import combinations
from collections import defaultdict

# Initialize the defaultdict: recommended
recommended = defaultdict(int)

# Iterate over all the nodes in G
for n, d in G.nodes(data=True):

    # Iterate over all possible triangle relationship combinations
    for n1, n2 in combinations(G.neighbors(n), 2):

        # Check whether n1 and n2 do not have an edge
        if not G.has_edge(n1, n2):

            # Increment recommended
            recommended[(n1, n2)] += 1

# Identify the top 10 pairs of users
all_counts = sorted(recommended.values())
top10_pairs = [pair for pair, count in recommended.items() if count >
all_counts[-10]]
print(top10_pairs)
```

```
[('u37', 'u2159'), ('u37', 'u3305'), ('u366', 'u322'), ('u366', 'u3305')]
```