This tutorial is Part 1 of an introduction to social network analysis in Python. It covers how to structure network data, as well as how to use NetworkX to: construct graphs, explore their features, and implement simple algorithms. The primary example used for replication is Zachary's (1977) paper on divisions within a collegiate karate club. The paper is available here (https://www.jstor.org/stable/pdf/3629752.pdf) and is the source of all tables/figures included below. We use the built-in dataset available from NetworkX, supplemented with a parsed and cleaned version of the weights provided here (http://vlado.fmf.uni-lj.si/pub/networks/data/ucinet/ucidata.htm#zachary).

Table of Contents

Let's get started! First, we need to import all of the packages we'll use to do our analysis.

In [ ]:
```python
import pandas as pd               # For analyzing tabular data
import numpy as np                # For working with arrays and numerical operations
import networkx as nx             # For network data specifically
import matplotlib.pyplot as plt   # For making plots
from pyvis import network as pv   # For making interactive plots
import pprint                     # For making output easier to read

%matplotlib inline
pd.set_option('display.max_columns', None)  # For preventing horizontal shortening of tables when displayed
print(nx.__version__)                       # Confirm that we have the latest networkx version
```

1 Working with network data

1.1 Structuring network data

The first challenge you'll face is figuring out how to structure your network data:

- What entities will form the **nodes** or **vertices**?
- What will define the **edges** or **links**?

In some cases the answer may seem obvious. For example, we are used to thinking about friendship networks, in which the nodes are people, and the edges are personal connections. In other cases, the natural representation is less clear.

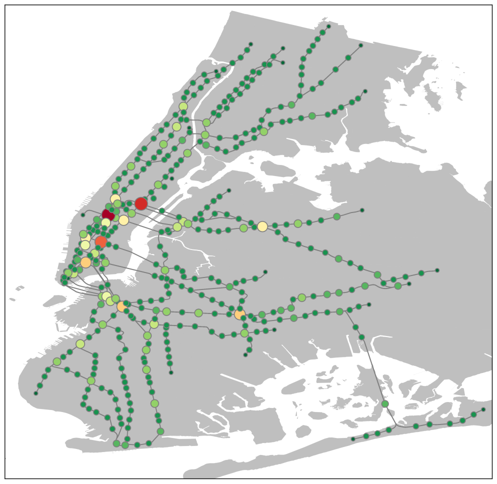
Example: Consider the Spotify dataset.

- Are nodes: songs, albums, artists, playlists, genres, or listeners?
- What should the links between these entities be?

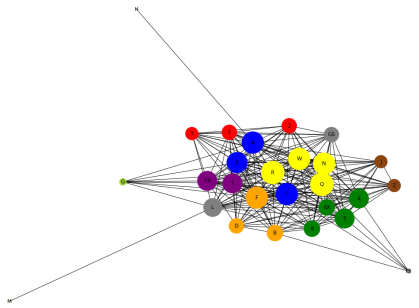
There are many different possible ways to structure this data. The right answer will depend on the question you're asking.

1.1.1 (Optional) Subway example

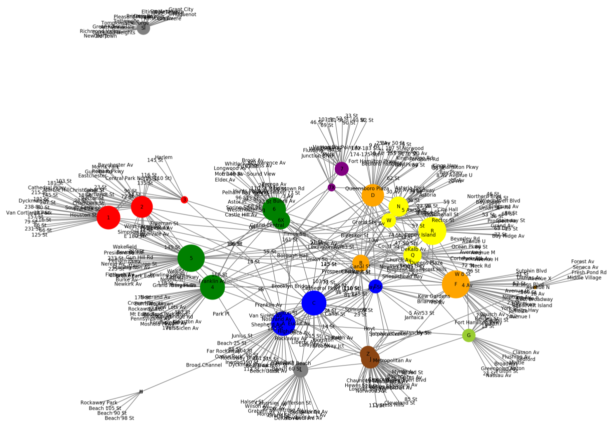
For example, consider the NYC subway. An intuitive representation might have subway stops as nodes, and subway lines as edges:



However, we could also build a network of lines connected by transfers:



Or a bi-partite network in which nodes are lines and stations, and edges represent the fact that a line stops at a station:



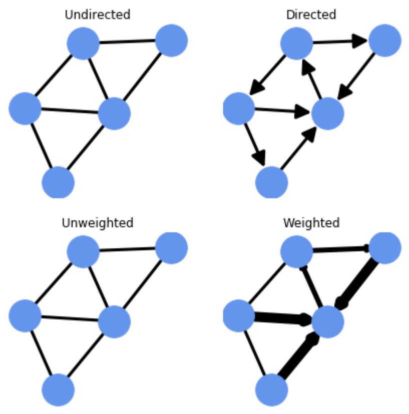
1.2 Types of network datasets

Once you have your nodes and edges, recall that there are two helpful dimensions along which networks can be categorized:

1. Whether their edges are **directed** or **undirected**
2. Whether their edges are **weighted** or **unweighted**

In this tutorial, we will discuss how to store and work with data from these different types of networks.

In [2]:



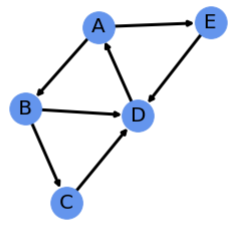
1.3 Storing network data

One of the first challenges of working with network data is to figure out how to store information about the network structure. There are three common ways to store network structure data:

1. **An edge list** has one row for each edge, containing the origin node and the destination node.
2. **A dictionary** has one key per node, which corresponds to a list of the node's neighbors.
3. **An adjacency matrix** has one entry *per* possible edge, set to 1 if the edge exists and 0 otherwise.

Consider the following example:

In [3]:



```
Edge list:                    Dictionary:               Adjacency matrix:
__________                    ___________               _________________

A,B                           A: B,E                       A  B  C  D  E
A,E                           B: C,D                    A  0  1  0  0  1
B,C                           C: D                      B  0  0  1  1  0
B,D                           D: A                      C  0  0  0  1  0
C,D                           E: D                      D  1  0  0  0  0
D,A                                                     E  0  0  0  1  0
E,D
```

Observe that the storage requirements for each of these data structures is very different. For example, an adjacency matrix has to store lots of zero entries, so in sparse graphs we might prefer to use edge lists or dictionaries.

Question: How would the adjacency matrix look different if the graph was weighted? If the graph was undirected?

Altogether, we can think of four main pieces of information that we'll need to store: **nodes, edges, node attributes, and edge attributes.**
Attribute information could be attached to one of the data structures above, or stored in a separate dataset.

- For example, the cells of an adjacency matrix might contain information on weights.
- For example, information about nodes (e.g. demographic characteristics) might be kept in a separate table.

2 Preparing network data with NetworkX

Let's start with a simple example using the graph above. This will help us build some intuition on the networkX package.

We will rely on the following commands:

```
nx.Graph()                    # Initialize a graph
nx.DiGraph()                  # Initialize a directed graph

G.clear()                     # Clear the graph
G.is_empty()                  # Confirm that the graph is empty

G.add_edge()                  # Add/remove single nodes/edges from graph
G.add_node()
G.remove_edge()
G.remove_nodes()

G.add_edges_from([...])       # Add/remove a list of multiple nodes/edges from graph
G.add_nodes_from([...])
G.remove_edges_from([...])
G.remove_nodes_from([...])
```

2.1 Building a graph from scratch

Let's build a simple graph from scratch. For this exercise, let's pretend that we're trying to map which NYU schools allow their students to cross-enroll with another.

2.1.1 Initializing a graph

In [ ]:
```
# Let's create an empty graph
G = nx.Graph()

# Sadly, there are no nodes or edges, which we can see if we plot:
nx.draw(G, with_labels=True)
```

In [ ]:
```
# As we can confirm, it's empty
nx.is_empty(G)
```

2.1.2 Adding nodes

In [ ]:
```
# Add a single node ...
G.add_node('Steinhardt')

nx.draw(G, with_labels=True)
```

In [ ]:
```
# ... or  a list of nodes
G.add_nodes_from(['Stern', 'Courant', 'Silver',
                  'Wagner', 'Tandon'])

nx.draw(G, with_labels=True)
```

2.1.3 Adding edges

In [ ]:
```
# Add a single edge...
G.add_edge('Silver', 'Steinhardt')

nx.draw(G, with_labels=True)
```

In [ ]:
```
# ... or a list of edges
G.add_edges_from([ ('Stern' , 'Courant'),
                   ('Tandon', 'Courant'),
                   ('Wagner', 'Silver'),
                   ('Wagner', 'Steinhardt'),
                   ('Wagner', 'Stern'),
                   ('Wagner', 'Tandon') ])

# Let's see if it worked:
nx.draw(G, with_labels=True)
```

2.2 Building a graph from an edgelist

Alternatively, you can just supply a list of edges to the graph

In [ ]:
```
H = nx.Graph([     ('Stern' , 'Courant'),
                   ('Tandon', 'Courant'),
                   ('Wagner', 'Silver'),
                   ('Wagner', 'Steinhardt'),
                   ('Wagner', 'Stern'),
                   ('Wagner', 'Tandon'),
                   ('Silver', 'Steinhardt')
               ])

nx.draw(H, with_labels=True)
```

2.3 Building special graph types

2.3.1 Weighted graphs

We can specify a weighted graph by including a dictionary with the weight as an attribute.

In [ ]:
```
H.clear()
H = nx.Graph([     ('Stern' , 'Courant',    {'weight': 5}),
                   ('Tandon', 'Courant',    {'weight': 7}),
                   ('Wagner', 'Silver',     {'weight': 1}),
                   ('Wagner', 'Steinhardt', {'weight': 2}),
                   ('Wagner', 'Stern',      {'weight': 9}),
                   ('Wagner', 'Tandon',     {'weight': 3}),
                   ('Silver', 'Steinhardt', {'weight': 6})
               ])

nx.draw(H, with_labels=True)
```

In [ ]:
```
H.edges.data()
```

2.3.2 Directed graphs

We specify a directed graph using `nx.DiGraph` instead of `nx.Graph`

In [ ]:
```
H.clear()
H = nx.DiGraph([   ('Stern' , 'Courant'),
                   ('Tandon', 'Courant'),
                   ('Wagner', 'Silver'),
                   ('Wagner', 'Steinhardt'),
                   ('Wagner', 'Stern'),
                   ('Wagner', 'Tandon'),
                   ('Silver', 'Steinhardt')
               ])

nx.draw(H, with_labels=True, arrowsize=25)
```

Can you modify H to reflect the following changes?
- Courant now allows its students to enroll in Tandon.
- GSAS allows enrollment in Steinhardt, Wagner, and Stern.
- Silver no longer wants to participate in any cross-enrolment.

Answer: H.add_edge('Courant', 'Tandon') H.add_edges_from([('GSAS', 'Steinhardt'),('GSAS', 'Wagner'),('GSAS', 'Stern')]) H.remove_node('Silver') nx.draw(H, with_labels=True, arrowsize=25)

In [ ]:

2.4 Importing built-in graphs

In the next lesson, we'll learn how to read in graphs from external data sources. For now, perhaps the most interesting way to get a graph is to use built-in graphs! You can find a list of existing graphs, or options for generating certain types of graphs, here: https://networkx.github.io/documentation/stable/reference/generators.html (https://networkx.github.io/documentation/stable/reference/generators.html)

In [ ]:
```
# For example, let's generate a random tree
T = nx.random_tree(20)
nx.draw(T)
```

In [ ]:
```
# Or, we can import Zachary (1977)'s Karate Club:
Z = nx.karate_club_graph()
nx.draw(Z)
```

3 Analyzing network data

Let's stick with Zachary's Karate club and try to learn some more about it. Recall the key facts:

- This is a network consisting of 34 members of a Karate club, observed for two years (1970-72).
- The teacher, Mr. Hi, and the president, John A. had a dispute about fees.
- The club split into two factions, which formed new clubs: Mr. Hi's faction, and John's faction (the officers' club).

Just a quick side note: For replicating the original paper and general ease of use, I have included a quick bit of helper code in the cell below. You don't need to understand it fully right now, but make sure you've run the cell.

In [ ]:
```
# Let's shift the keys by 1 to match the original paper.
new_keys = {key: key+1 for key in range(0,34)}
nx.relabel_nodes(Z, new_keys, copy=False)

# Let's set a standard plot layout
pos = nx.spring_layout(Z, seed=2,  k=.25)
```

3.1 Inspecting graphs

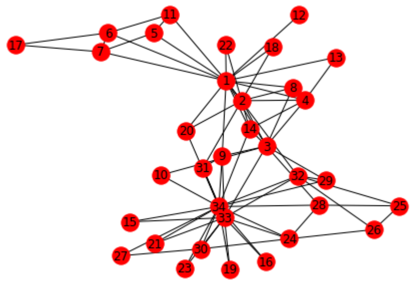
Reference: https://networkx.github.io/documentation/stable/reference/functions.html (https://networkx.github.io/documentation/stable/reference/functions.html)

We'll rely on the following commands:

```
nx.info(G)                         # Print a quick summary of the graph
G.is_directed()                    # Check if the graph is directed

G.number_of_nodes()                # Get count of nodes
G.number_of_edges()                # Get count of edges

G.nodes                            # Get a list of nodes
G.edges                            # Get a list of edges

 .data()                           # Get data attached to nodes/edges
 .items()                          # Get list of nodes/edges in iterable format

nx.get_node_attributes(G, 'attr')  # Get a dictionary of nodes and their values of 'attr'
```

3.1.1 Graph size

In [7]:



In [ ]:
```
print(nx.info(Z))
```

In [ ]:
```
# Let's get the graph size another way
n_V = Z.number_of_nodes()
n_E = Z.number_of_edges()

print(f"Our graph has {n_V} nodes and {n_E} edges.")
```

3.1.2 Node data

In [ ]:
```
# Let's get the nodes
Z.nodes()
```

In [ ]:
```
# Let's get any attributes of nodes ...
Z.nodes.data()
```

In [ ]:
```
# Great! There is an attribute called club. Let's extract it:
nx.get_node_attributes(Z, 'club')
```

3.1.3 Edge data

In [ ]:
```
# Let's get the edges
Z.edges()
```

In [ ]:
```
# Let's get any attributes of edges ...
Z.edges.data()
```

In [ ]:
```
# OK. We can see that there are no attributes.

# Instead, let's ask: Are these edges directed?
Z.is_directed()
```

Interesting. We see that we have some labels on which club everyone joined (Mr. Hi's or Officer's), but we are using the unweighted version of the graph. As we expect from the paper, it is undirected.

Question: What do the edges represent? When Zachary does provide weights, how does he determine them?

3.2 Local structure

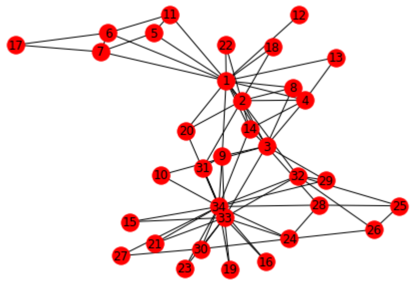
Reference: https://networkx.github.io/documentation/stable/reference/functions.html (https://networkx.github.io/documentation/stable/reference/functions.html)

We'll use the following commands:

```
nx.neighbors(G, n)                     # Get a list of neighbors for node n
nx.common_neighbors(G, n1, n2)         # Get common neighbors for nodes n1 and n2
nx.non_neighbors(G, n)                 # Get non-neighbors of node n

G.degree(n)                            # Get n's degree
G.degree                               # Get all degrees for all nodes

nx.local_bridges(G, with_span=False)  # Get a list of all edges that form local bridges

nx.number_connected_components(G)     # Get count of connected components
list(nx.connected_components(G))      # Get a list of connected components
```

Let's take a second look at the graph and start exploring.

In [8]:



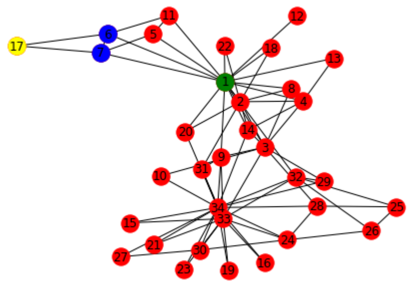
3.2.1 Exploring neighbors

In [ ]:
```python
# Node 17 looks lonely. Let's confirm that (s)he's only friends with 6 and 7
list(nx.neighbors(Z, 17))
```

In [ ]:
```python
# Node 1 is important - it's Mr. Hi himself. Let's see who his neighbors are:
list(nx.neighbors(Z, 1))
```

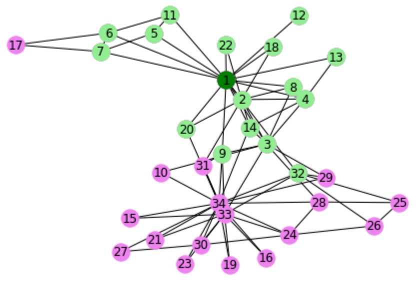
In [ ]:
```python
# Do they share any common neighbors?
list(nx.common_neighbors(Z,1,17))
```

In [9]:



In [ ]:
```python
# Is there anyone that Mr. Hi is not friends with?
sorted(list(nx.non_neighbors(Z, 1)))
```

In [10]:



3.2.2 Exploring degree

In [ ]:
```python
# How many neighbors does Mr. Hi have after all? Let's get the degree.
Z.degree(1)
```

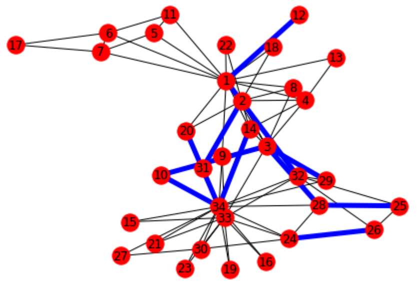
In [ ]:
```python
# Actually, we can get the degree of all nodes:
Z.degree
```

3.2.3 Local bridges

In [ ]:
```python
# We can even get a list of local bridges.
list(nx.local_bridges(Z, with_span=False))
```

In [ ]:
```python
# Recall that a local bridge consists of two nodes who don't share any neighbors
# Let's check if they fit the definition:
list(nx.common_neighbors(Z,34,20))
```

In [11]:



3.2.4 Connected components

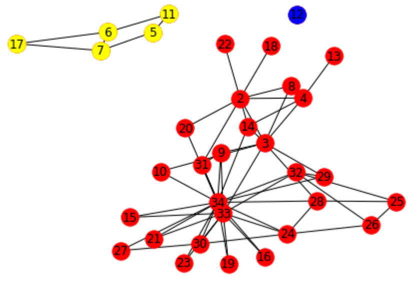
Reference: [https://networkx.github.io/documentation/stable/reference/algorithms/component.html (https://networkx.github.io/documentation/stable/reference/algorithms/component.html)](https://networkx.github.io/documentation/stable/reference/algorithms/component.html)

In [ ]:
```
# This graph is kind of un-interesting: there is only one component
nx.number_connected_components(Z)
```

In [ ]:
```
# But, we saw that Mr. Hi was a very central node. What happens if he quits Karate altogether?

# Let's first copy the graph so we don't overwrite Z
Z_without_hi = Z.copy()

# Now, remove him
Z_without_hi.remove_node(1)

# See how many components are left
nx.number_connected_components(Z_without_hi)
```

In [ ]:
```
# And, we can see these components -- e.g. list(nx.connected_components(Z_without_hi))
for i in nx.connected_components(Z_without_hi):
    print(i)
```

In [14]:



3.3 Algorithms

We'll use the following commands:

```
nx.average_shortest_path_length(G)          # Get average length of the shortest path between any two nodes
nx.single_source_shortest_path(G, source)   # Get the shortest paths from a given source node
nx.shortest_path(G, source, sink)           # Get the shortest path between a source and a sink node

nx.minimum_cut(G, source, sink,
               capacity='weight')           # Partition the graph using the minimum cut
```

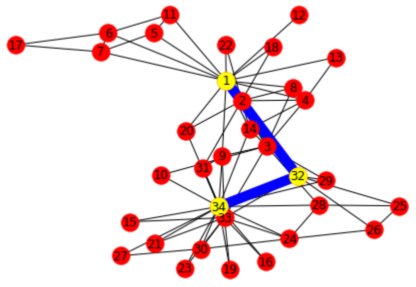
3.3.1 Shortest path

In [ ]:
```
# Let's get the average shortest path length for this club. We can see that there are only 2.4 degrees of separation!
nx.average_shortest_path_length(Z)
```

In [ ]:
```
# We could also get the shortest path to all nodes from a given source, e.g. Mr. Hi
nx.single_source_shortest_path(Z, 1)
```

In [ ]:
```
# Finally, we could also get the shortest path for a specific origin and destination.
# For example how far is Mr. Hi from John?
nx.shortest_path(Z, 1, 34)
```

In [15]:

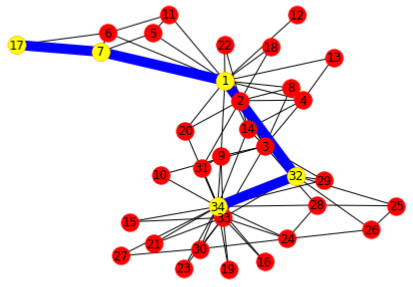


Exercise:

- What is the shortest path between John and node 17?
- If Mr. Hi leaves, what is the shortest path between John and node 17? (don't forget that we already made the graph: Z_without_hi)

Answer: nx.shortest_path(Z, 17,34) nx.shortest_path(Z_without_hi, 17,34)

In [16]:

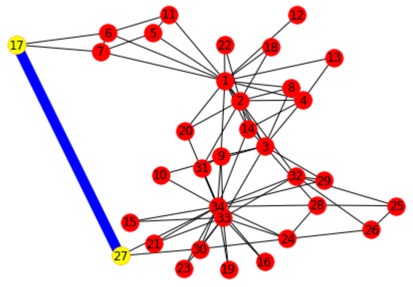


Exercise:

- If we add a node between nodes 17 and 27, how will the average shortest path in the graph change? (Don't forget to copy the graph first).

Answer: Z_copy = Z.copy() Z_copy.add_edge(17,27) old_average = nx.average_shortest_path_length(Z) new_average = nx.average_shortest_path_length(Z_copy) print(f"The average shortest path decreases from {old_average} to {new_average}.")

In [17]:



3.3.2 Max flow / min cut

Zachary uses a simple model of information flow and conflict:

- Information flows between two poles in the network: Mr. Hi's, and John's.
- Information flows over edges, i.e. interaction between members.
- Information flow increases with the strength of these edges, i.e. the number of interactions.
- Bottlenecks in information flow represent weak parts of the network.
- These bottlenecks can be used to predict how the network will split in the face of conflict.

His two main hypotheses are:

H1: "[There exists] some structural feature in the network inhibiting information flow between factions."

H2: "A bottleneck in the network, representing a structural limitation on information flow from the source to the sink, will predict the break that occurred in the club at the time of the fission."

Computationally, he identifies bottlenecks using the maximum-flow minimum-cut labeling procedure of Ford and Fulkerson. In short,

- In a network, we can find the **maximum flow** that can be sustained between a source and a sink node.
  - It is determined by the graph structure and the capacity of the edges in the graph.
- This is equivalent in cost to the **minimum cut** needed to partition the graph into two components (one with the source, the other with the sink).
  - It is defined as the minimum total weight of edges that can be removed from the graph to partition it.

- Ford and Fulkerson provide an algorithm for identifying these edges.
  - If we switch the source and the sink, repeat the min-cut procedure, and still get the same answer, then our cut is unique.

> Per Zachary, "intuitively stated, they proved that the maximum flow is equal to the capacity of the smallest possible break in the network separating the source from the sink."

Brief detour: In order to apply the max flow / min cut algorithm to replicate Zachary, we need edge weights. The code below imports and assigns them to our graph. We'll go over importing data from outside sources in the next section. For now, let's take it on faith that we can read in a weighted edgelist from CSV...

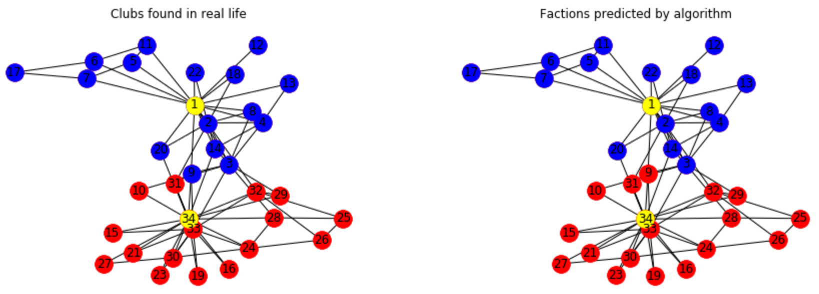
In [ ]:
```python
# Import edge weights...
C = pd.read_csv("data/zachary_edge_weights.csv")

# ... and add the edges to our graph
for edge in np.array(C):
    u,v,w= edge
    Z[u][v]['weight'] = w

# Make sure it worked
Z.edges.data()
```

3.3.2.1 Find the minimum cut

In [ ]:
```python
source = 1   # Mr. Hi
sink   = 34  # John A

# Let's apply the algorithm to make the minimum cut on the graph
cut_value, partition = nx.minimum_cut(Z, source, sink, capacity='weight')
```

In [ ]:
```python
# We can inspect the remaining partition of the graph:
partition
```

In [22]:



We can see that we have replicated Zachary's findings: we have one misprediction, node 9.

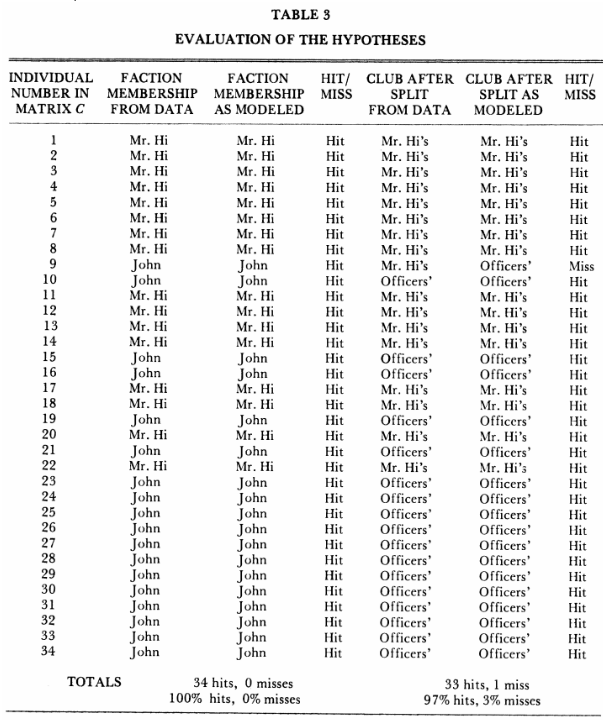


TABLE 3

EVALUATION OF THE HYPOTHESES

INDIVIDUAL NUMBER IN MATRIX *C*	FACTION MEMBERSHIP FROM DATA	FACTION MEMBERSHIP AS MODELED	HIT/ MISS	CLUB AFTER SPLIT FROM DATA	CLUB AFTER SPLIT AS MODELED	HIT/ MISS
1	Mr. Hi	Mr. Hi	Hit	Mr. Hi's	Mr. Hi's	Hit
2	Mr. Hi	Mr. Hi	Hit	Mr. Hi's	Mr. Hi's	Hit
3	Mr. Hi	Mr. Hi	Hit	Mr. Hi's	Mr. Hi's	Hit
4	Mr. Hi	Mr. Hi	Hit	Mr. Hi's	Mr. Hi's	Hit
5	Mr. Hi	Mr. Hi	Hit	Mr. Hi's	Mr. Hi's	Hit
6	Mr. Hi	Mr. Hi	Hit	Mr. Hi's	Mr. Hi's	Hit
7	Mr. Hi	Mr. Hi	Hit	Mr. Hi's	Mr. Hi's	Hit
8	Mr. Hi	Mr. Hi	Hit	Mr. Hi's	Mr. Hi's	Hit
9	John	John	Hit	Mr. Hi's	Officers'	Miss
10	John	John	Hit	Officers'	Officers'	Hit
11	Mr. Hi	Mr. Hi	Hit	Mr. Hi's	Mr. Hi's	Hit
12	Mr. Hi	Mr. Hi	Hit	Mr. Hi's	Mr. Hi's	Hit
13	Mr. Hi	Mr. Hi	Hit	Mr. Hi's	Mr. Hi's	Hit
14	Mr. Hi	Mr. Hi	Hit	Mr. Hi's	Mr. Hi's	Hit
15	John	John	Hit	Officers'	Officers'	Hit
16	John	John	Hit	Officers'	Officers'	Hit
17	Mr. Hi	Mr. Hi	Hit	Mr. Hi's	Mr. Hi's	Hit
18	Mr. Hi	Mr. Hi	Hit	Mr. Hi's	Mr. Hi's	Hit
19	John	John	Hit	Officers'	Officers'	Hit
20	Mr. Hi	Mr. Hi	Hit	Mr. Hi's	Mr. Hi's	Hit
21	John	John	Hit	Officers'	Officers'	Hit
22	Mr. Hi	Mr. Hi	Hit	Mr. Hi's	Mr. Hi's	Hit
23	John	John	Hit	Officers'	Officers'	Hit
24	John	John	Hit	Officers'	Officers'	Hit
25	John	John	Hit	Officers'	Officers'	Hit
26	John	John	Hit	Officers'	Officers'	Hit
27	John	John	Hit	Officers'	Officers'	Hit
28	John	John	Hit	Officers'	Officers'	Hit
29	John	John	Hit	Officers'	Officers'	Hit
30	John	John	Hit	Officers'	Officers'	Hit
31	John	John	Hit	Officers'	Officers'	Hit
32	John	John	Hit	Officers'	Officers'	Hit
33	John	John	Hit	Officers'	Officers'	Hit
34	John	John	Hit	Officers'	Officers'	Hit
TOTALS	34 hits, 0 misses 100% hits, 0% misses			33 hits, 1 miss 97% hits, 3% misses		

4 Exporting NetworkX graphs to other data structures

Let's revisit the graph representations from above. Now that we have this nice Karate club graph, we can convert it back into the canonical representations and replicate the adjacency matrix from the paper. We'll use the following commands:

```
nx.to_edgelist(Z)           # Convert to edge list
nx.to_dict_of_lists(Z)      # Convert to a dictionary of lists
nx.to_pandas_adjacency(Z)   # Convert to an adjacency matrix in pandas
nx.to_numpy_matrix(Z)       # Convert to an adjacency matrix in numpy
```

4.1 As edgelist

In [ ]:
```
# The simplest representation is a basic edgelist
list(nx.to_edgelist(Z))
```

4.2 As dictionary

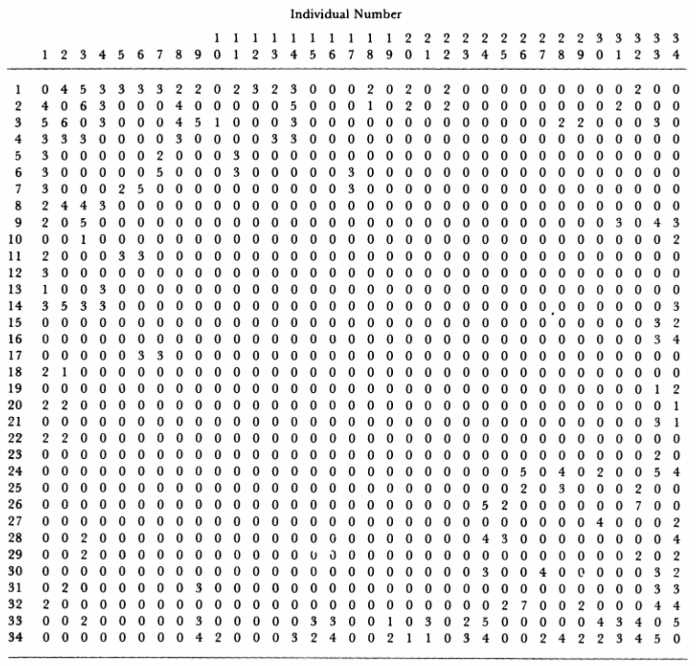
In [ ]:
```
# We can also convert to a dictionary of lists
nx.to_dict_of_lists(Z)
```

4.3 As adjacency matrix

In [ ]:
```
# To adjacency matrix (the add-ons are to make the print output match the paper)
nx.to_pandas_adjacency(Z)  .astype(int).sort_index().sort_index(axis=1)
```

FIGURE 3

QUANTIFIED MATRIX OF RELATIVE STRENGTHS OF THE RELATIONSHIPS IN THE KARATE CLUB: THE MATRIX *C*



Individual Number

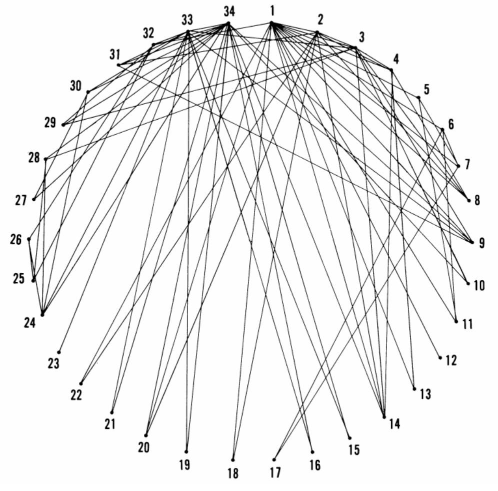
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34
1	0	4	5	3	3	3	3	2	2	0	2	3	2	3	0	0	0	2	0	2	0	2	0	0	0	0	0	0	0	0	0	2	0	0
2	4	0	6	3	0	0	0	4	0	0	0	0	0	5	0	0	0	1	0	2	0	2	0	0	0	0	0	0	0	0	2	0	0	0
3	5	6	0	3	0	0	0	4	5	1	0	0	0	3	0	0	0	0	0	0	0	0	0	0	0	0	0	2	2	0	0	0	3	0
4	3	3	3	0	0	0	0	3	0	0	0	0	3	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	3	0	0	0	0	0	2	0	0	0	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	3	0	0	0	0	0	5	0	0	0	3	0	0	0	0	0	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7	3	0	0	0	2	5	0	0	0	0	0	0	0	0	0	0	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
8	2	4	4	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
9	2	0	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3	0	4	3
10	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2
11	2	0	0	0	3	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
12	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
13	1	0	0	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
14	3	5	3	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3
15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3	2
16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3	4
17	0	0	0	0	0	3	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
18	2	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
19	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	2
20	2	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
21	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3	1
22	2	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
23	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0
24	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	5	0	4	0	2	0	0	5	4
25	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0	3	0	0	0	2	0	0
26	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	5	2	0	0	0	0	0	0	7	0	0
27	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4	0	0	0	2
28	0	0	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4	3	0	0	0	0	0	0	0	0	4
29	0	0	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0	2
30	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3	0	0	4	0	0	0	0	0	3	2
31	0	2	0	0	0	0	0	0	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3	3
32	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	7	0	0	2	0	0	0	4	4
33	0	0	2	0	0	0	0	0	3	0	0	0	0	0	3	3	0	0	1	0	3	0	2	5	0	0	0	0	0	4	3	4	0	5
34	0	0	0	0	0	0	0	0	4	2	0	0	0	3	2	4	0	0	2	1	1	0	3	4	0	0	2	4	2	2	3	4	5	0

5 Wrap-up

5.1 Plots

Let's end with a simple plot to replicate the classic diagram from Zachary:

FIGURE 1
Social Network Model of Relationships in the Karate Club



We will cover graphing in the next notebook, but just for fun...

In [ ]:
```
# Let's tell networkX we want a circular layout
pos = nx.circular_layout(Z)

# We need to jump through some hoops to rotate the graph
new_pos = {}
for k,v in pos.items():
    new_pos[ (k+8)%34 +1 ] = v

# And there we go!
nx.draw(Z, pos=new_pos, with_labels=True)
```

5.2 Conclusion

We have used the Zachary (1977) Karate club data to practice some simple network analysis. We have learned about graph representations, how to read data into `networkX` graphs, and some basic analytic steps like finding connected components and calculating shortest paths. Next, we'll dig into visualizing graphs using a more complex network as an example.

Further reading and tutorials:

- http://datenstrom.gitlab.io/cs532-s17/notebooks/karate_club.html (http://datenstrom.gitlab.io/cs532-s17/notebooks/karate_club.html)
- https://petterhol.me/2018/01/28/zacharys-zachary-karate-club/ (https://petterhol.me/2018/01/28/zacharys-zachary-karate-club/)
- http://studentwork.prattsi.org/infovis/labs/zacharys-karate-club/ (http://studentwork.prattsi.org/infovis/labs/zacharys-karate-club/)

If you **really** enjoyed yourself today, you should aspire to join the Zachary Karate Club Club (http://networkkarate.tumblr.com/)!