APPLICATIONS

OF DATA SCIENCE

NetworkX

Applications of Data Science - Class 8

Giora Simchoni

`gsimchoni@gmail.com and add #dsapps in subject`

Stat. and OR Department, TAU

2022-12-26

Why NetworkX?

Pros and Cons of `NetworkX`

Pros:

- Maintained!
- Friendly: Community, Docs, Installation (as opposed to `igraph`)
- Relatively easy interface (as opposed to `igraph`)
- Speaks pandas, numpy and other important python tools
- Nodes can be anything, including python objects

Cons:

- Slow, probably not for large networks (as opposed to `igraph`, `graph-tool`)
- Unimpressive viz (as opposed to Gephi, ggraph, others)
- Not the most comprehensive algorithms selection

Creating a Graph(): Manually

Four types of Graph objects

Undirected:

```
import networkx as nx

G = nx.Graph()
```

Directed:

```
G = nx.DiGraph()
```

Multiedge undirected:

```
G = nx.MultiGraph()
```

Multiedge directed:

```
G = nx.MultiDiGraph()
```

Add Nodes

One or more nodes:

```
G = nx.Graph()
G.add_node('John')
G.add_nodes_from(['Paul', 'George', 'Ringo'])
```

Watch a Graph's nodes:

```
G.nodes
```

```
## NodeView(('John', 'Paul', 'George', 'Ringo'))
```

Print nicer as a list:

```
list(G.nodes)
```

```
## ['John', 'Paul', 'George', 'Ringo']
```

Add Edges

One or more edges:

```
G.add_edge('John', 'Paul')
G.add_edges_from([('Paul', 'George'), ('John', 'Ringo')])
```

Surprisingly, this would also work:

```
G.add_edge('John', 'Brian')
```

Watch a Graph's edges:

```
print(list(G.edges))
```

```
## [('John', 'Paul'), ('John', 'Ringo'), ('John', 'Brian'), ('Paul', 'Georg
```

Remove nodes or edges

```
G.remove_node('Brian')
```

Did `NetworkX` bother to remove the edge as well?

```
print(list(G.edges))
```

```
## [('John', 'Paul'), ('John', 'Ringo'), ('Paul', 'George')]
```

Remove an edge, remove from a list:

```
G.remove_edge('Paul', 'John') # why did it work?
G.remove_nodes_from(['John', 'George'])
```

Break the Beatles altogether:

```
G.clear()
```

Good to know

- Adding an edge also adds its nodes if they didn't exist before
- Adding a duplicate node/edge is ignored unless `MultiGraph()`
- Removing an edge does not remove its nodes
- Removing a node removes all edges to/from it
- Removing a non-existent node/edge raises an error unless it is part of a list in which case - ignored

Nodes and Edges Attributes

While adding to Graph:

```
G.add_node('Ringo', alive=True)
G.add_nodes_from([('George', {'alive': False}), ('John', {'alive':
# can also do: G.add_nodes_from(['George', 'John'], alive=False)
G.add_edge('John', 'Paul', year=1957)
G.add_edges_from([('Ringo', 'John'), ('Ringo', 'Paul'), ('Ringo',
```

Watch attributes with the `nodes()` and `edges()` methods:

```
print(G.nodes(data=True))
```

```
## [('Ringo', {'alive': True}), ('George', {'alive': False}), ('John', {'al
```

```
print(G.nodes(data='alive'))
```

```
## [('Ringo', True), ('George', False), ('John', False), ('Paul', None)]
```

Nodes and Edges Attributes

Setting an attribute of an existing node/edge through the `nodes` and `edges` dictionary attributes:

```
print(G.nodes['Paul'])
```

```
## {}
```

```
G.nodes['Paul']['alive'] = True #?
```

```
G.add_edge('John', 'George')
G.edges[('John', 'George')]['year'] = 1958
```

As a dictionary you can also delete an attribute this way:

```
del G.nodes['Paul']['alive']
del G.edges[('John', 'George')]['year']
```

Nodes and Edges Attributes

Finally you can set an attribute from a simple dictionary:

```
year_met = {
   ('John', 'Paul'): 1957,
   ('Paul', 'Ringo'): 1960,
   ('Paul', 'George'): 1958,
   ('John', 'George'): 1958
}
nx.set_edge_attributes(G, year_met, 'year')
```

Or multiple attributes with a nested dictionary:

```
node_attrs = {
   'John': {'alive': False, 'born': 1940},
   'Paul': {'alive': True, 'born': 1942},
   'George': {'alive': False, 'born': 1943},
   'Ringo': {'alive': True, 'born': 1940}
}
nx.set_node_attributes(G, node_attrs)
```

Good to have

The edge attribute `weight` is so important it got its own method:

```
G = nx.Graph()
G.add_weighted_edges_from([(0, 1, 3.0), (1, 2, 7.5)])
```

Quick methods to know:

```
print(G.number_of_nodes())
```

```
## 3
```

```
print(G.number_of_edges())
```

```
## 2
```

```
print(G.is_directed())
```

```
## False
```

Good to have

```
print(G.has_node(3))
```

```
## False
```

```
print(G.has_edge(0, 1))
```

```
## True
```

```
print(G.subgraph([0, 1]).number_of_edges())
```

```
## 1
```

```
print(list(G.neighbors(1)))
```

```
## [0, 2]
```

What about `DiGraph()`?

Convert an undirected graph to a directed graph:

```
D = nx.DiGraph(G)
```

Edges direction matters:

```
D.add_edge(2, 3, weight=10.0)
print(D.has_edge(2, 3))
```

```
## True
```

```
print(D.has_edge(3, 2))
```

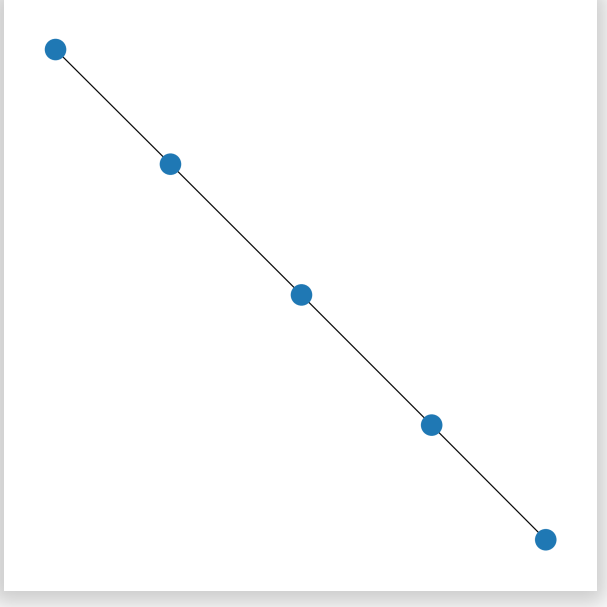
```
## False
```

Other than that...

Visualize a Graph(): matplotlib

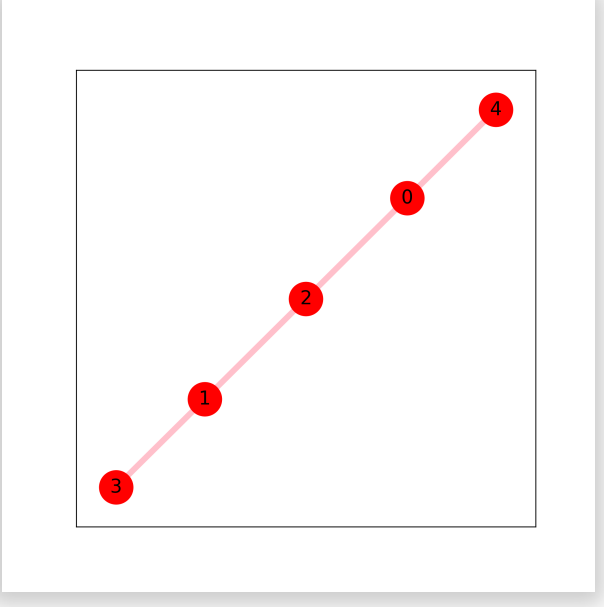
`draw()`: The very basic

```
import matplotlib.pyplot as plt

G = nx.erdos_renyi_graph(n=5, p=0.6) # a.k.a binomial_graph()

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



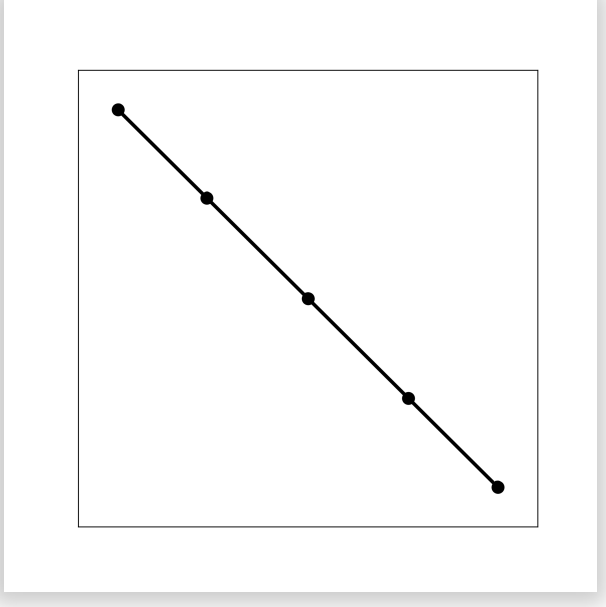
`draw_networkx()`: Some more options

```
nx.draw_networkx(G, node_size=800, node_color='red', edge_color='p
  width=5, font_size=16)
plt.show()
```



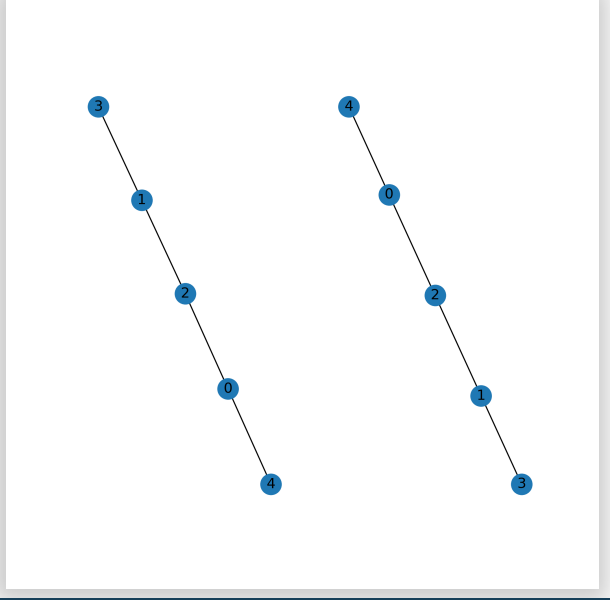
You may like the unpacking a dictionary option better:

```
options = {
    'node_color': 'black',
    'node_size': 100,
    'width': 3,
}

nx.draw_networkx(G, **options)
plt.show()
```



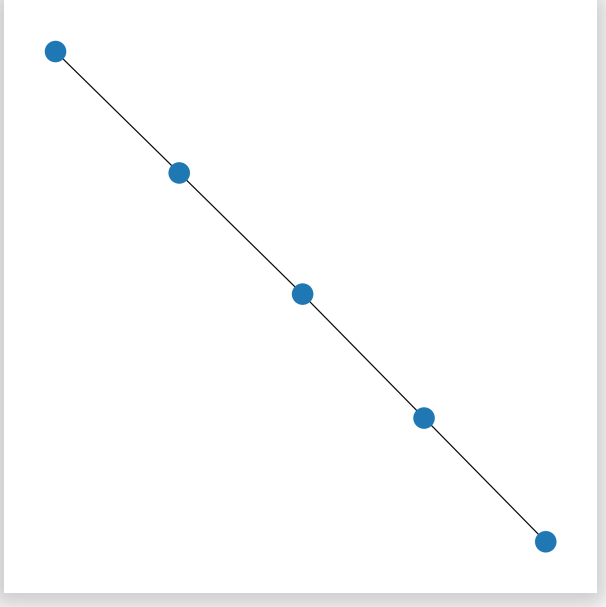
Layouts Galore

```
plt.subplot(121)
limits = plt.axis('off')
nx.draw_networkx(G, pos=nx.kamada_kawai_layout(G))
plt.subplot(122)
limits = plt.axis('off')
nx.draw_networkx(G, pos=nx.fruchterman_reingold_layout(G))
plt.show()
```



Use a Layout directly

```
nx.draw_kamada_kawai(G)
plt.show()
```



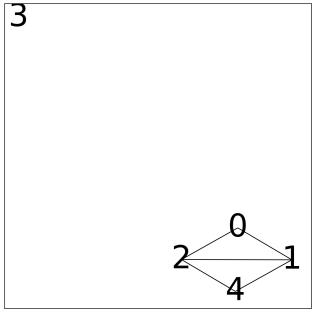
For nicer graphs go to...

- Grave?
- nxviz?
- Plotly (interactive graphs for Python or R)
- Gephi (a Desktop app, neither R nor Python but both export to Gephi format)
- d3 (JS delight)
- ggraph (R)?

Create a Graph(): In Real Life

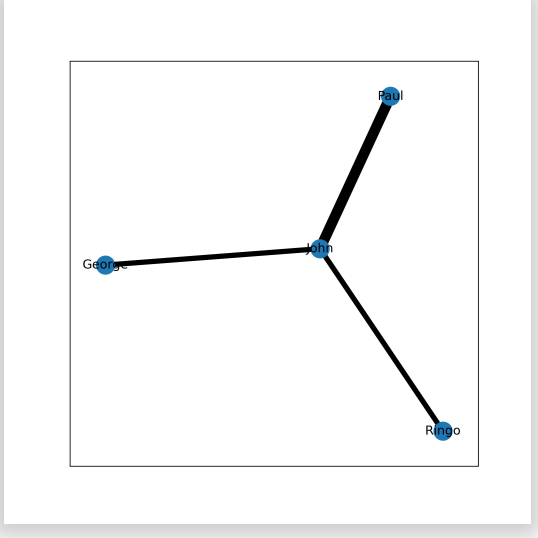
Numpy

```
import numpy as np

A = np.array([
[0, 1, 1, 0, 0],
[1, 0, 1, 0, 1],
[1, 1, 0, 0, 1],
[0, 0, 0, 0, 0],
[0, 1, 1, 0, 0]])

G = nx.from_numpy_array(A)
nx.draw_networkx(G, node_size=0, font_size=40)
plt.show()
```



Edgelist - a list of edges

```
beatles_edgelist = [('John', 'Paul', {'weight': 1.0}), ('John', 'G
  ('John', 'Ringo', {'weight': 0.5})]

G = nx.from_edgelist(beatles_edgelist)

weights = [10 * attrs['weight'] for u, v, attrs in G.edges(data=Tr
nx.draw_networkx(G, width=weights)
plt.show()
```



Edgelist - a file

This is how the `beatles.edgelist` file looks like:

```
John Paul 1
John George 0.5
John Ringo 0.5
```

Read it with `read_edgelist()` or `read_weighted_edgelist()`:

```
with open('../data/beatles.edgelist', 'rb') as edgelist_file:
  G = nx.read_edgelist(edgelist_file, data=(('weight', float),))
print(G.edges(data=True))
```

```
## [('John', 'Paul', {'weight': 1.0}), ('John', 'George', {'weight': 0.5}),
```

```
with open('../data/beatles.edgelist', 'rb') as edgelist_file:
  G = nx.read_weighted_edgelist(edgelist_file)
print(G.edges(data=True))
```

```
## [('John', 'Paul', {'weight': 1.0}), ('John', 'George', {'weight': 0.5}),
```

CSV

This is how the `beatles.csv` file looks like:

```
John,Paul,1957,1.0
John,George,1958,0.5
John,Ringo,1960,0.5
```

You can still use `read_edgelist()` to read it:

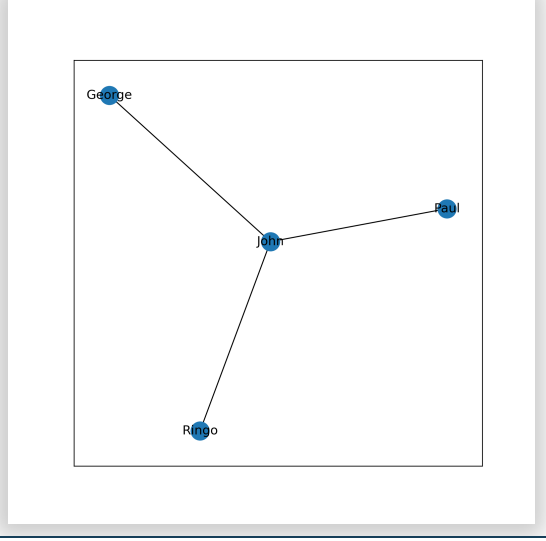
```
with open('../data/beatles.csv', 'rb') as edgelist_file:
  G = nx.read_edgelist(edgelist_file, delimiter=',', data=(('year'
print(G.edges(data=True))
```

```
## [('John', 'Paul', {'year': 1957, 'weight': 1.0}), ('John', 'George', {'y
```

Or, you can just use Pandas `read_csv()` and...

Pandas

```
import pandas as pd

beatles_df = pd.read_csv('../data/beatles.csv', header=None)
beatles_df.columns = ['source', 'target', 'year', 'weight']

G = nx.from_pandas_edgelist(beatles_df, 'source', 'target', ['year

nx.draw_networkx(G)
plt.show()
```



What about writing?

All methods we've seen have a `write_` or `to_` complementary function:

```
print(nx.to_edgelist(G))
```

```
## [('John', 'Paul', {'year': 1957, 'weight': 1.0}), ('John', 'George', {'y
```

What about `DiGraph()`?

All methods we've seen have a `create_using` parameter:

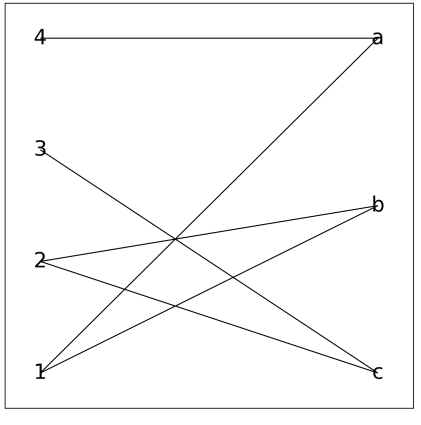
```
D = nx.from_pandas_edgelist(beatles_df, 'source', 'target',
  ['year', 'weight'], create_using=nx.DiGraph)
```

⚠️ When creating `DiGraph()` NetworkX assumes A_{ij} corresponds to the edge from i to j, contrary to our convention. Therefore, you should use `A.transpose()`

Bipartite

For creating a bipartite network, use the `bipartite` node attribute:

```
G = nx.Graph()
G.add_nodes_from([1, 2, 3, 4], bipartite=0)
G.add_nodes_from(['a', 'b', 'c'], bipartite=1)
G.add_edges_from([(1, 'a'), (1, 'b'), (2, 'b'), (2, 'c'), (3, 'c')

nx.draw_networkx(G, pos=nx.bipartite_layout(G, [1, 2, 3, 4]), node
plt.show()
```

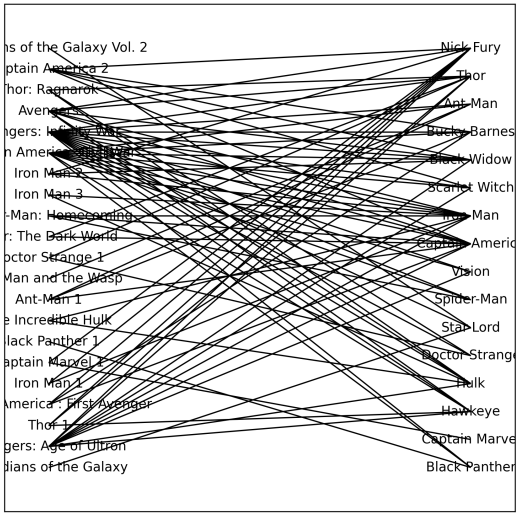


"In real life" you would probably have something like the Marvel incidence matrix in a CSV, what `NetworkX` calls a *biadjacency_matrix*, or a list of edges you could make a Scipy *sparse matrix* with, from which you can create a bipartite graph:

```
from scipy import sparse

marvel = pd.read_csv("../data/marvel_incidence_matrix.csv")
B = marvel.iloc[:, 1:].values

sB = sparse.csr_matrix(B)

G = nx.bipartite.from_biadjacency_matrix(sB)
print(G.nodes(data=True))
```

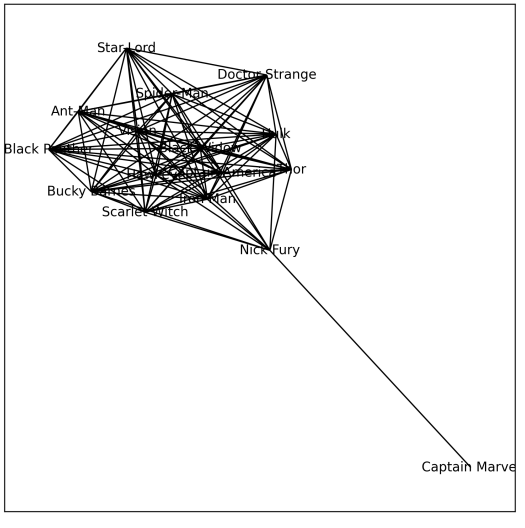
```
## [(0, {'bipartite': 0}), (1, {'bipartite': 0}), (2, {'bipartite': 0}), (3
```

```
films = marvel['Film'].to_list()
characters = marvel.columns[1:].to_list()

id_to_label = {id: char for id, char in enumerate(films + characte
G = nx.relabel_nodes(G, id_to_label)

nx.draw_networkx(G, pos=nx.bipartite_layout(G, films), node_size=(
plt.show()
```



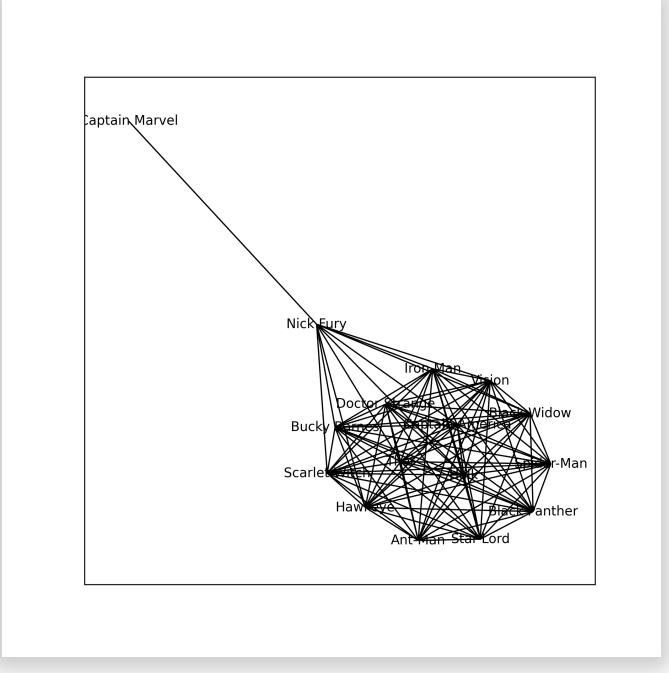
You've seen how to project an incidence matrix with numpy:

```
P = B.transpose() @ B
np.fill_diagonal(P, 0)
Gc = nx.from_numpy_array(P)

id_to_char = {id: char for id, char in enumerate(characters)}
Gc = nx.relabel_nodes(Gc, id_to_char)
nx.draw_networkx(Gc, node_size=0, font_size=10)
plt.show()
```



But NetworkX can do all that for you:

```
Gc = nx.bipartite.projected_graph(G, characters)

nx.draw_networkx(Gc, node_size=0, font_size=10)
plt.show()
```



Similarity

Let's recreate the sci-fi similarity network. We'll do some filtering:

- The original CSV includes 156 books, we'll take only those marked as "popular"
- Take only books with at least 2 features marked > 0
- Won't be looking at the author's name, gender, or date of release

```
scifi = pd.read_csv('../data/sci_fi_books.csv', index_col='book')
scifi['n_features'] = (scifi.iloc[:, 6:17] > 0).sum(axis=1)
scifi = scifi[scifi['popular'] == 1]

scifi_fil = scifi[scifi['n_features'] >= 2]
cols_to_drop = ['date', 'author', 'frequency', 'author_gender',
  'quarter_century', 'century', 'popular', 'n_features']
scifi_fil = scifi_fil.drop(cols_to_drop, axis=1)

# to use Pandas corr function need to transpose data frame
scifi_corr = scifi_fil.T.corr(method='spearman')
```

We reached a 25x25 symmetric correlation matrix for 25 books:

```
scifi_corr.values[:5, :5]
```

```
## array([[ 1.        ,  0.62769528,  0.2975283 ,  0.69310571, -0.2111842 ]
##        [ 0.62769528,  1.        ,  0.68760573,  0.3303204 ,  0.23530427]
##        [ 0.2975283 ,  0.68760573,  1.        ,  0.0037037 ,  0.48989795]
##        [ 0.69310571,  0.3303204 ,  0.0037037 ,  1.        , -0.04082483]
##        [-0.2111842 ,  0.23530427,  0.48989795, -0.04082483,  1.        ]
```

Turn all upper triangle to `Nan`, so we'll get only 25 * 24 / 2 correlations:

```
scifi_corr.values[np.triu_indices_from(scifi_corr.values)] = np.na

scifi_corr.values[:5, :5]
```

```
## array([[        nan,         nan,         nan,         nan,         nan]
##        [ 0.62769528,         nan,         nan,         nan,         nan]
##        [ 0.2975283 ,  0.68760573,         nan,         nan,         nan]
##        [ 0.69310571,  0.3303204 ,  0.0037037 ,         nan,         nan]
##        [-0.2111842 ,  0.23530427,  0.48989795, -0.04082483,         nan]
```

Convert this to a long edgelist, filter out the `Nan`:

```
scifi_edgelist = scifi_corr.reset_index().melt(id_vars='book',
  value_vars=scifi_fil.index, var_name='book2', value_name='corr')

scifi_edgelist = scifi_edgelist[~pd.isna(scifi_edgelist['corr'])]
print(scifi_edgelist.shape)
```

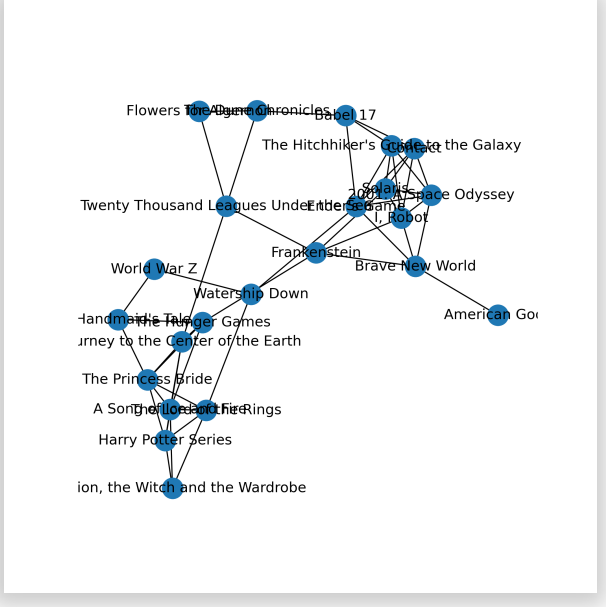
```
## (300, 3)
```

```
scifi_edgelist.head(5)
```

```
##                                        book         book2      corr
## 1  Twenty Thousand Leagues Under the Sea  Frankenstein  0.627695
## 2     Journey to the Center of the Earth  Frankenstein  0.297528
## 3                          Brave New World  Frankenstein  0.693106
## 4   The Lion, the Witch and the Wardrobe  Frankenstein -0.211184
## 5                                I, Robot  Frankenstein  0.700617
```

Filter all correlations above 0.5 and convert to a `Graph()`:

```
G = nx.from_pandas_edgelist(scifi_edgelist[scifi_edgelist['corr']
  'book', 'book2', ['corr'])

nx.draw_networkx(G)
limits = plt.axis('off')
plt.show()
```



Let NetworkX work

Adjacency Matrix

You can get a Numpy array with `to_numpy_array()`:

```
A = nx.to_numpy_array(G)

print(A.shape)
```

```
## (23, 23)
```

```
A[:4, :4]
```

```
## array([[0., 1., 0., 0.],
##        [1., 0., 1., 1.],
##        [0., 1., 0., 1.],
##        [0., 1., 1., 0.]])
```

> 💡 Say, didn't we have 25 books?

But a realistic network would be large and sparse so the `adjacency_matrix()` method returns a SciPy sparse matrix:

```
A = nx.adjacency_matrix(G)

print(A.shape)
```

```
## (23, 23)
```

```
print(A[:4, :4])
```

```
##   (0, 1)    1
##   (1, 0)    1
##   (1, 2)    1
##   (1, 3)    1
##   (2, 1)    1
##   (2, 3)    1
##   (3, 1)    1
##   (3, 2)    1
```

> 💡 Do you know what a sparse matrix is?

Easy to check if...

```
print(nx.is_directed(G))
```

```
## False
```

```
print(nx.is_directed_acyclic_graph(G))
```

```
## False
```

```
print(nx.is_bipartite(G))
```

```
## False
```

```
print(nx.is_connected(G))
```

```
## True
```

Degree and Density

Degree is a big deal, there's an attribute *and* a method for that:

```
G.degree
```

```
## DegreeView({'Twenty Thousand Leagues Under the Sea': 4, 'Frankenstein': 
```

```
list(G.degree)
```

```
## [('Twenty Thousand Leagues Under the Sea', 4), ('Frankenstein', 5), ('Br
```

Average degree:

```
np.mean(list(dict(G.degree).values()))
```

```
## 4.434782608695652
```

Get the degree of a specific node:

```
G.degree['Contact']
```

```
## 6
```

Get the density of a network:

```
nx.density(G)
```

```
## 0.2015810276679842
```

For directed graphs, use `in_degree` and `out_degree`:

```
D = nx.DiGraph(G)

D.in_degree()
D.out_degree()
```

Paths and Diameter

Is there a path between two nodes?

```
nx.has_path(G, 'Frankenstein', 'Contact')
```

```
## True
```

What is the shortest path distance between two nodes:

```
nx.shortest_path_length(G, 'Frankenstein', 'Contact')
```

```
## 2
```

Give me (one, there could be more) shortest path:

```
nx.shortest_path(G, 'Frankenstein', 'Contact')
```

```
## ['Frankenstein', 'I, Robot', 'Contact']
```

Give me all shortest paths between two nodes:

```
all_sps = nx.all_shortest_paths(G, 'Frankenstein', 'Contact') # ge
list(all_sps)
```

```
## [['Frankenstein', 'I, Robot', 'Contact'], ['Frankenstein', 'Solaris', 'C
```

Get the diameter of a network:

```
nx.diameter(G)
```

```
## 5
```

Components

How many connected components?

```
nx.number_connected_components(G)
```

```
## 1
```

All connected components, each is a set of nodes:

```
cc = nx.connected_components(G) # generator!
list(cc)
```

```
## [{'The Dune Chronicles', 'The Hunger Games', "The Hitchhiker's Guide to
```

Get component of a specific node:

```
nx.node_connected_component(G, 'Contact')
```

```
## {'The Dune Chronicles', 'The Hunger Games', "The Hitchhiker's Guide to t
```

Get largest component (a.k.a Giant Connected Component)

```
gcc = max(nx.connected_components(G), key=len)
len(gcc)
```

```
## 23
```

For directed graphs use `strongly_`/`weakly_` prefixes with previous functions:

```
D = nx.DiGraph(G)

nx.number_strongly_connected_components(D)
nx.strongly_connected_components(D)
```

Laplacian

```
L = nx.laplacian_matrix(G)

L.todense()[:5, :5]
```

```
## matrix([[ 4, -1,  0,  0,  0],
##         [-1,  5, -1, -1, -1],
##         [ 0, -1,  5, -1,  0],
##         [ 0, -1, -1,  6, -1],
##         [ 0, -1,  0, -1,  5]], dtype=int32)
```