NetworkX

Contents

NetworkX is a Python package for dealing with complex networks (graphs). It provides Graph classes, graph algorithms, and visualization tools.

You may need to

```
(pycourse) conda install networkx
```

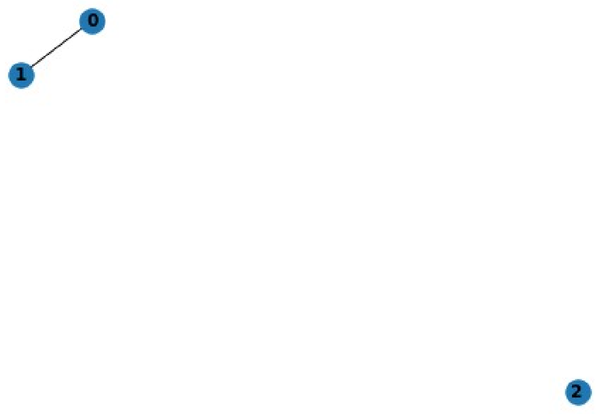
```
import numpy as np
import matplotlib.pyplot as plt
import networkx as nx
nx.__version__
```

```
'2.5'
```

We'll cover some basics in NetworkX. You can find additional information in the NetworkX documentation, which includes a tutorial.

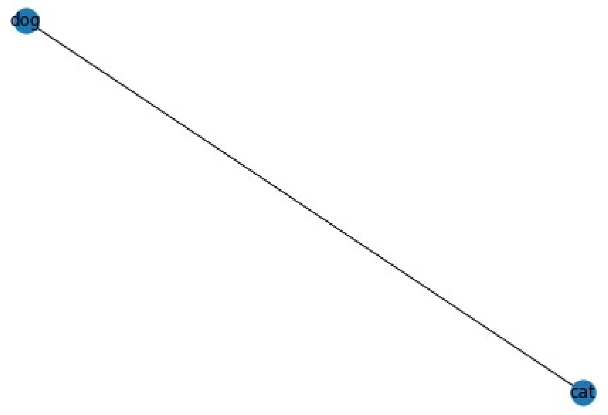
First, let's create a simple graph:

```
G = nx.Graph()

# add nodes
G.add_node(0, name="dog")
G.add_node(1)
G.add_nodes_from(range(3)) # adds nodes 0, 1

# add edge from node 0 to node 1
G.add_edge(0,1)

# draws the graph to pyplot axes
nx.draw(G, with_labels=True, font_weight='bold')
plt.show()
```



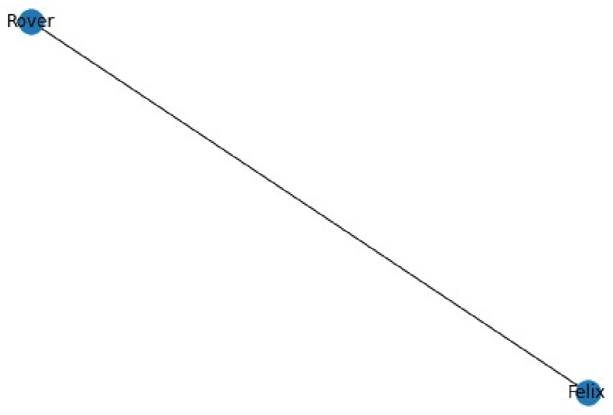
Graph nodes can be any hashable object, which covers many things you might use

```
G = nx.Graph()

# add nodes
G.add_node("dog")
G.add_node("cat")

# add edge
G.add_edge("cat","dog")

# draws the graph to pyplot axes
nx.draw(G, with_labels=True)
plt.show()
```



You can also associate arbitrary data with each node and edge by passing in keyword arguments

```
G = nx.Graph()

# add nodes
G.add_node("dog", name="Rover", weight=10.0)
G.add_node("cat", name="Felix", height=5)

# add edge
G.add_edge("cat","dog", time=2.0)

# draws the graph to pyplot axes
nx.draw(G, with_labels=True, labels=nx.get_node_attributes(G, "name"))
plt.show()
```



```
G.get_edge_data("cat", "dog")
```

```
{'time': 2.0}
```

```
nx.get_edge_attributes(G, "time")
```

```
{('dog', 'cat'): 2.0}
```

```
names = nx.get_node_attributes(G, "name")
names
```

```
{'dog': 'Rover', 'cat': 'Felix'}
```

```
weights = nx.get_node_attributes(G, "weight")
weights
```

```
{'dog': 10.0}
```

Generating Graphs

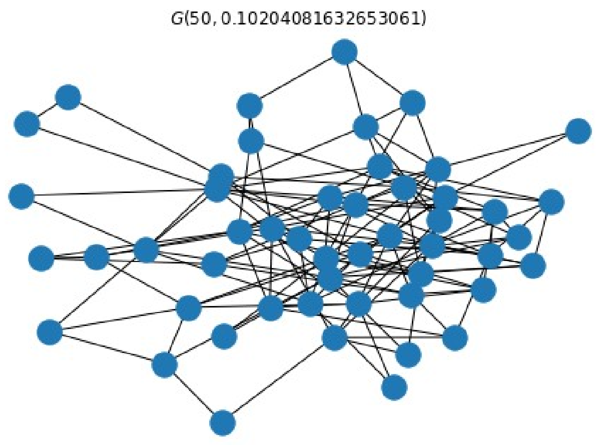
NetworkX provides Graph generators to generate a variety of random graphs.

Random Graphs

Erdos-Renyi Graphs

An Erdos-Renyi random graph $G_{n,p}$ is a graph on n nodes, where the probability of an edge (i,j) existing is p. In NetworkX, this is called a `gnp` graph.

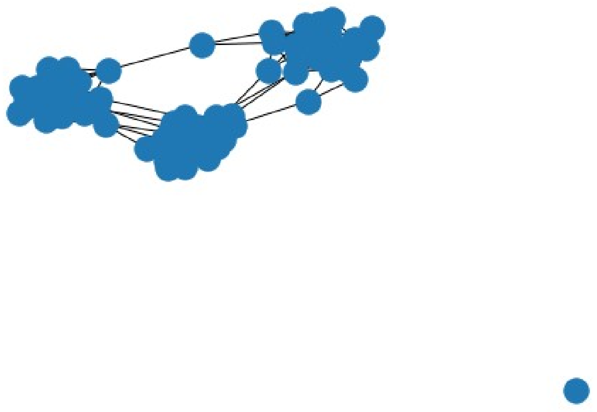
```
n = 50
p = 5 / (n-1) # 5 is expected number of neighbors of a single vertex
G = nx.gnp_random_graph(n, p)
nx.draw(G, with_labels=False)
plt.title(r'$G({},{})$'.format(n,p))
plt.show()
```



Stochastic Block Models

A Graph generated from a stochastic block model (SBM) has k clusters, and a $k \times k$ (symmetric) matrix P of probabilities, where $P_{i,j}$ is the probability that a pair of nodes (a,b) will be joined by an edge if a is in cluster i and b is in cluster j.

```
ns = [25, 25, 25] # size of clusters
ps = [[0.3, 0.01, 0.01], [0.01, 0.3, 0.01], [0.01, 0.01, 0.3]] #
probability of edge
G = nx.stochastic_block_model(ns, ps)
nx.draw(G, with_labels=False)
plt.show()
```



Visualization

Visualizing graphs is often a great way to see and understand information.

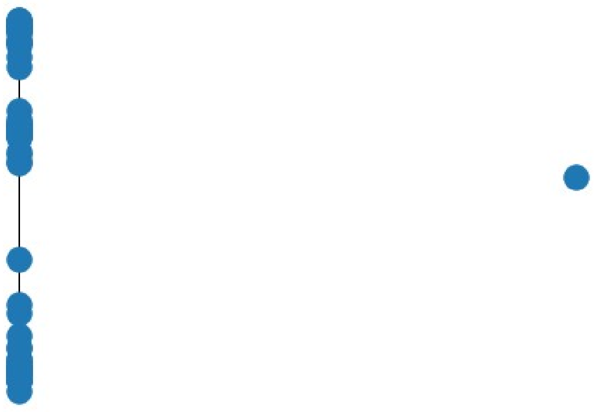
Unless there is some “ground truth” way to lay out nodes (meaning each node has an associated coordinate), we must choose some way to place nodes in our visualization. There are a variety of methods for this.

Layouts

There are several layout methods you might use, which are good for certain applications.

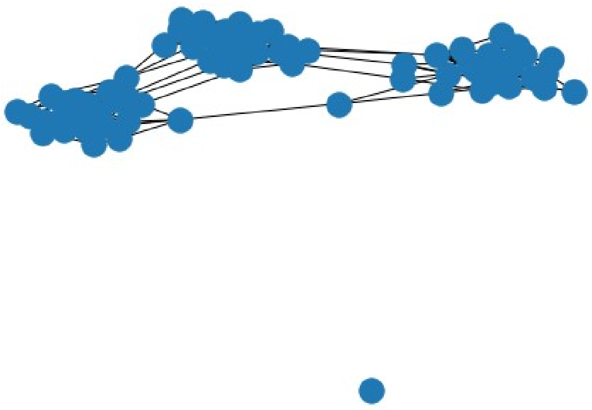
Spectral Embeddings are good for partitioning the graph into clusters.

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



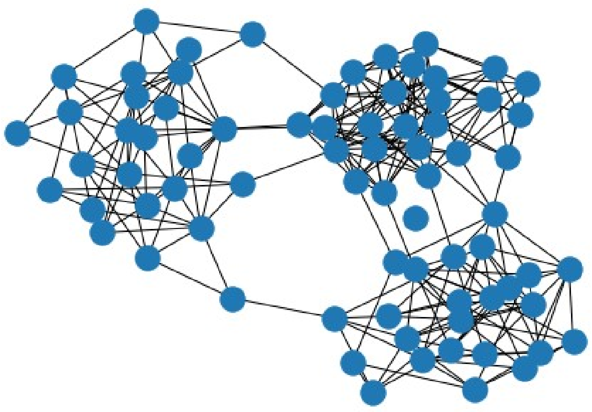
Spring layouts tend to do a good job of separating nodes, which can be nice for visualization

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



the Kamada-Kawai algorithm also gives visually pleasing results.

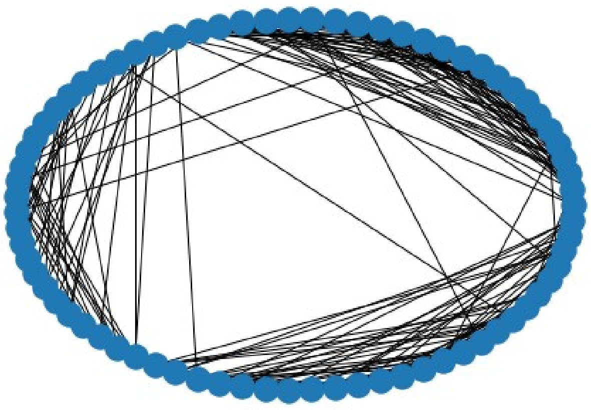
```
nx.draw_kamada_kawai(G)
plt.show()
```



```
pos = nx.kamada_kawai_layout(G)
pos
```

```
{0: array([0.55848456, 0.28631111]),
 1: array([0.2723952 , 0.06022474]),
 2: array([0.29600044, 0.42521603]),
 3: array([0.14204467, 0.50618183]),
 4: array([ 0.13569054, -0.00391481]),
 5: array([0.36586802, 0.14155925]),
 6: array([0.58235413, 0.40263966]),
 7: array([0.22701682, 0.4735505 ]),
 8: array([0.20471613, 0.2449802 ]),
 9: array([0.04167401, 0.45013743]),
 10: array([-0.02230495,  0.36050668]),
 11: array([0.26310646, 0.55601801]),
 12: array([0.09507921, 0.2511831 ]),
 13: array([-0.01296102,  0.15589922]),
 14: array([0.29740876, 0.25122146]),
 15: array([0.30338164, 0.3425249 ]),
 16: array([0.46148538, 0.34601535]),
 17: array([-0.04543286,  0.24141499]),
 18: array([0.04895396, 0.03792357]),
 19: array([0.48120167, 0.46254991]),
 20: array([0.24102299, 0.16557828]),
 21: array([0.10568134, 0.16036921]),
 22: array([0.16803906, 0.37395204]),
 23: array([-0.12459958,  0.24998575]),
 24: array([0.52080826, 0.12520194]),
 25: array([-0.83509014,  0.29632483]),
 26: array([-0.7970804 ,  0.09993388]),
 27: array([-0.3340208 , -0.40362148]),
 28: array([-0.85562945,  0.43181013]),
 29: array([-0.60377777,  0.20106874]),
 30: array([-0.59737155, -0.05315006]),
 31: array([-0.65431893,  0.06491301]),
 32: array([-0.46842143,  0.53144003]),
 33: array([-0.59851362,  0.64011876]),
 34: array([-0.65732526,  0.22843393]),
 35: array([-0.36024348,  0.23178491]),
 36: array([-0.30253631,  0.02616656]),
 37: array([-0.49436719,  0.444842  ]),
 38: array([-0.2693406 ,  0.58999807]),
 39: array([-0.63188034,  0.35842707]),
 40: array([-0.73433804, -0.15805037]),
 41: array([-0.76823287, -0.0704386 ]),
 42: array([-0.42971896, -0.13705016]),
 43: array([-1.        ,  0.22030478]),
 44: array([-0.89866356,  0.00399537]),
 45: array([-0.59349463, -0.25297497]),
 46: array([-0.46461129,  0.13242792]),
 47: array([-0.54012952,  0.31147695]),
 48: array([-0.51027701,  0.01206457]),
 49: array([-0.63818229,  0.44121927]),
 50: array([ 0.25449128, -0.6329606 ]),
 51: array([ 0.41765881, -0.74847725]),
 52: array([ 0.4427453 , -0.20906612]),
 53: array([ 0.20804549, -0.55597445]),
 54: array([ 0.04027806, -0.64605528]),
 55: array([ 0.37076363, -0.41763146]),
 56: array([ 0.57373344, -0.6693159 ]),
 57: array([ 0.58391881, -0.31600168]),
 58: array([ 0.37674834, -0.48387407]),
 59: array([ 0.47358989, -0.40345153]),
 60: array([ 0.34410606, -0.59696046]),
 61: array([ 0.15019332, -0.47281503]),
 62: array([ 0.71101042, -0.29324512]),
 63: array([ 0.23374494, -0.2948483 ]),
 64: array([ 0.6000465 , -0.42802762]),
 65: array([ 0.17297465, -0.2675781 ]),
 66: array([ 0.45073499, -0.61345329]),
 67: array([-0.01417495, -0.47971352]),
 68: array([ 0.354409  , -0.24604809]),
 69: array([ 0.5229518 , -0.36106347]),
 70: array([ 0.23368975, -0.10406808]),
 71: array([ 0.481575  , -0.08849365]),
 72: array([ 0.61958812, -0.60681014]),
 73: array([ 0.72421881, -0.56604564]),
 74: array([ 0.10340916, -0.75671666])}
```

```
nx.draw_circular(G)
```



Algorithms

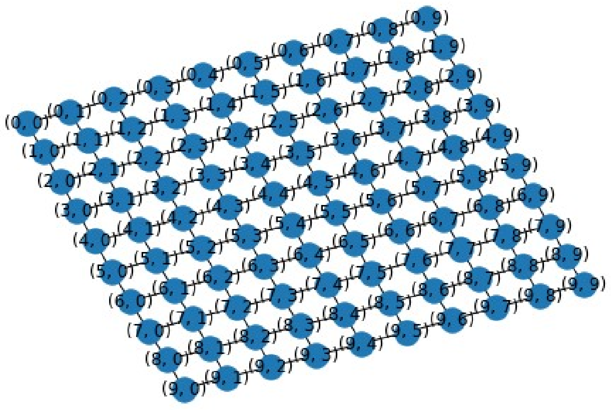
NetworkX implements a variety of graph algorithms.

Shortest Paths

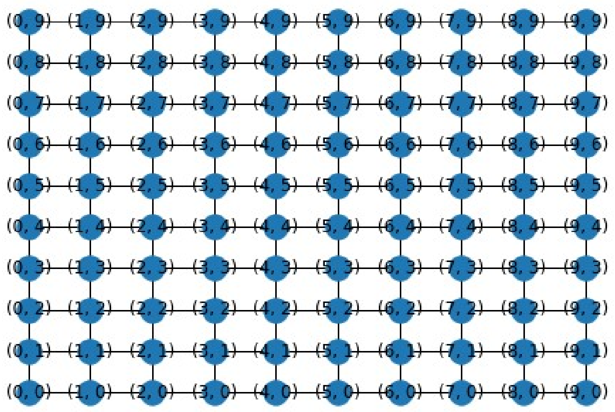
A path between nodes x and y is a sequence of edges where the target of an edge is the source of the next edge. \begin{equation} (x, v_0), (v_0, v_1), \dots, (v_k, y) \end{equation} The length of the path is the number of edges in the sequence. A shortest path is a path with the shortest length over all possible paths.

We'll illustrate on a 2D grid:

```
G = nx.grid_2d_graph(10,10)
pos = nx.kamada_kawai_layout(G)
nx.draw(G, pos, with_labels=True)
```

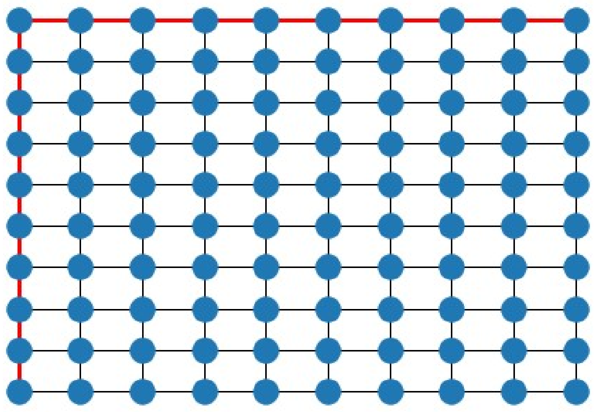


```
pos = {(i,j): np.array([i,j]) for i in range(10) for j in range(10)}
nx.draw(G, pos, with_labels=True)
```



The nodes of this 2D grid are indexed by coordinates (i, j).

```python
p = nx.shortest_path(G, (0,0), (9,9))
elist = [[p[i], p[i+1]] for i in range(len(p)-1)]
nx.draw(G, pos)
nx.draw_networkx_edges(G, pos, elist, edge_color='r', width=3.0)
plt.show()
```



The length of the shortest path between any two nodes in a graph G defines a metric on the vertex set (this is analagous to the geodesic metric on a manifold).

```python
dists = nx.shortest_path_length(G)
```

This creates an iterator over shortest path lengths for each node

Exercise

Write a function which returns a distance matrix for the shortest path length.

```python
## Your code here
```

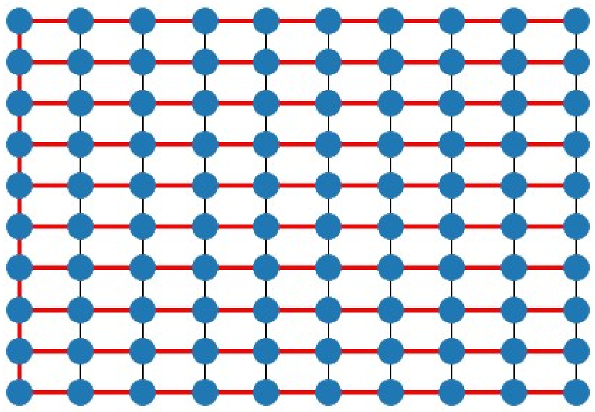
Spanning Trees

A tree is a connected graph with no cycles. If the graph has n nodes, it must have $m = n - 1$ edges to be a tree.

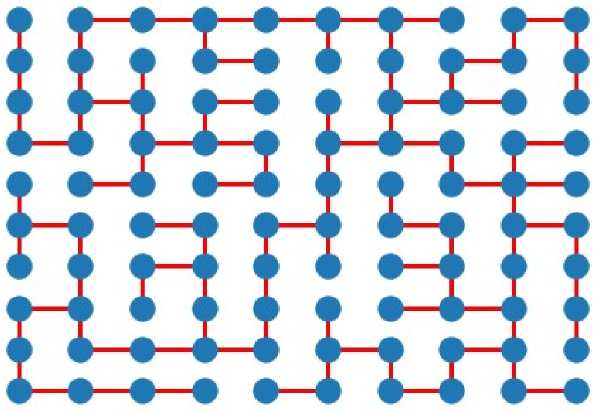
A spanning tree of a graph G is a sub-graph with the same set of nodes, and a subset of edges that forms a tree. A minimum spanning tree (MST) is a spanning tree with minimum weight (you can weight edges using a `'weight'` attribute).

You can compute a spanning tree of a graph using `nx.tree.minimum_spanning_tree`, or `nx.tree.minimum_spanning_edges`. Other tree algorithms are provided as well.

```python
T = nx.tree.minimum_spanning_tree(G)

nx.draw(G, pos)
nx.draw(T, pos, edge_color='r', width=3.0)
plt.show()
```



```python
# set random weights
for i,j in G.edges():
    G[i][j]['weight'] = np.random.rand()


T = nx.tree.minimum_spanning_tree(G)

# nx.draw(G, pos, )
nx.draw(T, pos, edge_color='r', width=3.0)
plt.show()
```



Exercise

Create a function that generates a maze on a 2-dimensional $m \times n$ grid using `nx.grid_2d_graph` and a randomly weighted MST. Place the start and end points at $(0, 0)$ and $(m-1, n-1)$ respectively.

Create a function that solves a maze by computing the shortest path between the start and end points.

```python
## Your code here
```

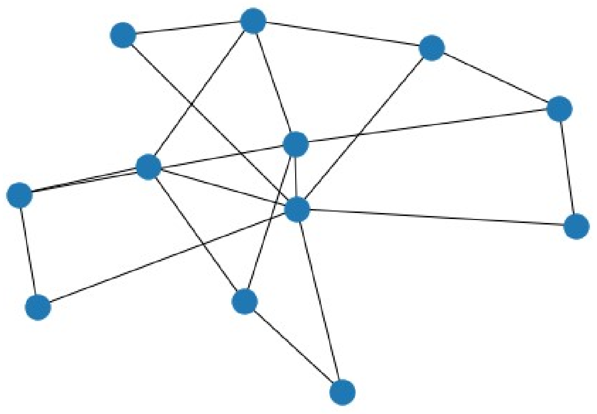
Bipartite Graphs

A bipartite graph is a graph where nodes are separated into two sets U, V, and where edges are of the form (u, v) with $u \in U$, and $v \in V$ (there are no edges between two points in U or two points in V).

Bipartite graphs are often used to encode matching problems. For example, dating websites might match romantic partners, and ride apps such as Uber and Lyft might match riders with drivers.

NetworkX functionality for biparite graphs is in `nx.algorithms.bipartite`

```
from networkx.algorithms import bipartite

B = bipartite.random_graph(5, 7, 0.5)

nx.draw(B)
```

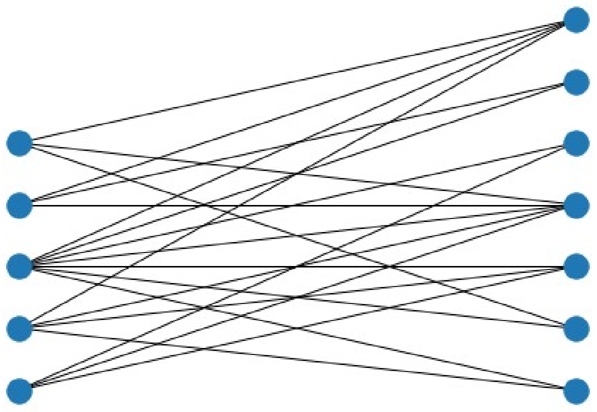


```
B.nodes(data=True)
```

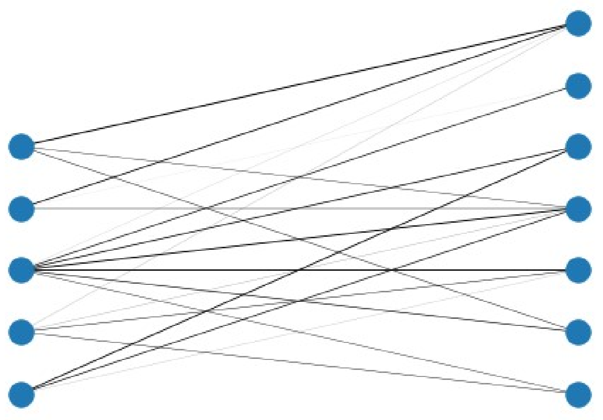
```
NodeDataView({0: {'bipartite': 0}, 1: {'bipartite': 0}, 2: {'bipartite':
0}, 3: {'bipartite': 0}, 4: {'bipartite': 0}, 5: {'bipartite': 1}, 6:
{'bipartite': 1}, 7: {'bipartite': 1}, 8: {'bipartite': 1}, 9:
{'bipartite': 1}, 10: {'bipartite': 1}, 11: {'bipartite': 1}})
```

Let’s explicitly define positions to visualize the bipartite graph

```
pos = {}
cts = [0, 0]

for i, data in B.nodes(data=True):
    group = data['bipartite']
    pos[i] = np.array([group, cts[group]])
    cts[group] = cts[group] + 1

nx.draw(B, pos)
```



```
# set random weights
for i,j in B.edges():
    B[i][j]['weight'] = np.random.rand()

weights = [B[u][v]['weight'] for u,v in B.edges()]

nx.draw(B, pos, width=weights)
```



```
# compute maximum matching
match = bipartite.maximum_matching(B)
match
```

```
{0: 7, 1: 5, 2: 6, 3: 8, 4: 11, 5: 1, 6: 2, 7: 0, 8: 3, 11: 4}
```

```
elist = [[k, v] for k, v in match.items()]
```