The NetworkX library

Satyaki Sikdar

NetworkX is a Python package for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks.

Features

- Data structures for graphs, digraphs, and multigraphs
- Open source
- Many standard graph algorithms
- Network structure and analysis measures
- Generators for classic graphs, random graphs, and synthetic networks
- Nodes can be "anything" (e.g., text, images, XML records)
- Edges can hold arbitrary data (e.g., weights, time-series)
- Well tested with over 90% code coverage
- Additional benefits from Python include fast prototyping, easy to teach, and multi-platform

Background

- NetworkX was born in May 2002.
- First public release was in April 2005
- Written in pure Python using some NumPy and SciPy functions
- Not parallel
- Latest stable version 2.2

Graphs are stored as nested dictionaries

Provides easy access to nodes & edges as well as their attributes

Execution Model

- API calls
- Integrates very well with other Python code and libraries since it's pure Python

```
import networkx as nx
```

```
G = nx.Graph()  # simple undirected graph
# G = nx.DiGraph()  # simple directed graph
# G = nx.MultiGraph()  # unidirected multigraph
# G = nx.MultiDiGraph()  # directed multigraph
```

```
G.add_node('apple', color='red')

G.add_edge(1, 2, capacity=3)
G.add_edge('two', 3.0, weight=2.5)
```

```
# add nodes and edges from an iterable

G.add_nodes_from(['notre', 'dame'])
G.add_edges_from([(1, 5), ('two', 1)])
```

```
G.nodes()
```

```
NodeView(('apple', 1, 2, 'two', 3.0, 'notre', 'dame', 5))
```

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

```
NodeDataView({'apple': {'color': 'red'}, 1: {}, 2: {}, 'two': {}, 3.0: {},
'notre': {}, 'dame': {}, 5: {}})
```

```
G.edges()
```

```
EdgeView([(1, 2), (1, 5), (1, 'two'), ('two', 3.0)])
```

```
G.edges(data=True)
```

```
EdgeDataView([(1, 2, {'capacity': 3}), (1, 5, {}), (1, 'two', {}), ('two', 3.0, {'weight': 2.5})])
```

```
G.order(), G.size()
```

```
(8, 4)
```

Neighborhoods and basic traversals

```
G = nx.complete_graph(4)
```

```
list(G.neighbors(0))
```

```
[1, 2, 3]
```

```
from collections import deque

def BFS(G, s):
    """
    Runs BFS from source node 's'. Returns the shortest path dictionary 'd'
    """
    d = {}
    d[s] = 0
```

```
    Q = deque()
    Q.append(s)

    while len(Q) != 0:
        u = Q.popleft()
        for v in G.neighbors(u):
            if v not in d:
                d[v] = d[u] + 1
                Q.append(v)
    return d
```

```
BFS(G, 1)
```

```
{1: 0, 0: 1, 2: 1, 3: 1}
```

Graph Generators

```
# path graph
G = nx.path_graph(n=10)

# complete graph
G = nx.complete_graph(n=5)

# random graphs
G = nx.erdos_renyi_graph(n=100, p=0.2)
G = nx.watts_strogatz_graph(n=50, k=3, p=0.2)
G = nx.barabasi_albert_graph(n=20, m=2)

# configuration model
G = nx.configuration_model(deg_sequence=[1, 1, 2, 2])

# classic social networks
G = nx.karate_club_graph()
G = nx.davis_southern_women_graph()
G = nx.florentine_families_graph()
```

Official reference

Graph drawing

- With matplotlib - OK at best
- Export as gml / gexf use Gephi
- Export as dot code

Bipartite graphs and algorithms

```
from networkx.algorithms import bipartite
import matplotlib.pyplot as plt
```

```
B = nx.Graph()

# Add nodes with the node attribute "bipartite"
B.add_nodes_from([1, 2, 3, 4], bipartite=0)
B.add_nodes_from(['a', 'b', 'c'], bipartite=1)

# Add edges only between nodes of opposite node sets
B.add_edges_from([(1, 'a'), (1, 'b'), (2, 'b'), (2, 'c'), (3, 'c'), (4, 'a')])
```

```
bipartite.is_bipartite(B)
```

```
True
```

```
nx.bipartite.maximum_matching(B)
```

```
{1: 'a', 2: 'b', 3: 'c', 'a': 1, 'c': 3, 'b': 2}
```

```
G_projected = bipartite.weighted_projected_graph(B, [1, 2, 3, 4])
print(G_projected.edges(data=True))
```

```
[(1, 2, {'weight': 1}), (1, 4, {'weight': 1}), (2, 3, {'weight': 1})]
```

[Official reference]

Connectivity

```
G = nx.complete_graph(3)
G.add_edges_from([('a', 'b')])

print(nx.is_connected(G))
print(nx.number_connected_components(G))

for nodes in nx.connected_components(G):
    print(nodes)
```

```
False
2
{0, 1, 2}
{'a', 'b'}
```

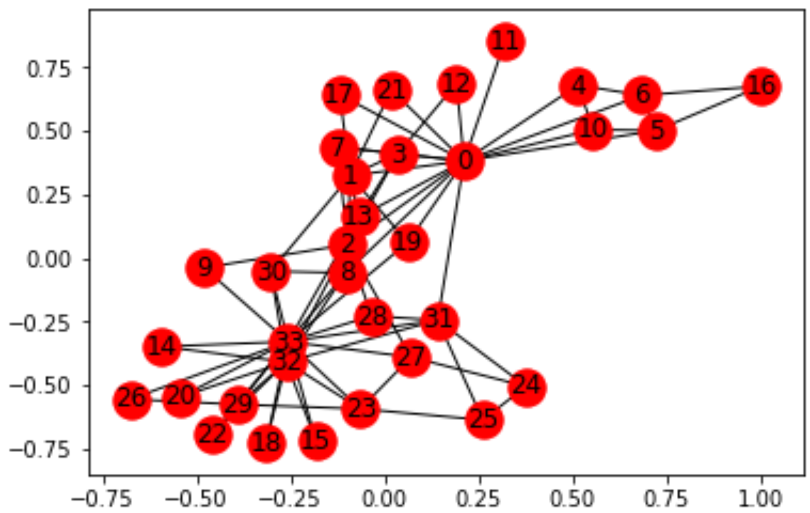
```
G = nx.complete_graph(3, create_using=nx.DiGraph())
G.add_edge(4, 5)
G.add_edge(5, 6)
G.add_edge(6, 5)

print(nx.is_strongly_connected(G))
for nodes in nx.strongly_connected_components(G):
    print(nodes)
```

```
False
{0, 1, 2}
{5, 6}
{4}
```

Centrality measures

```
G = nx.karate_club_graph()
```

```
nx.draw_networkx(G)
```




```
preds = nx.jaccard_coefficient(G, [(0, 33), (1, 33)])
for u, v, p in preds:
    print('(%d, %d) -> %.8f' % (u, v, p))
```

```
(0, 33) -> 0.13793103
(1, 33) -> 0.13043478
```

```
deg_centrality = nx.degree_centrality(G)  # gives back a dictionary

print('Top 3 nodes having the highest degree centrality')
for node, val in sorted(deg_centrality.items(),
                        key=lambda x: x[1],
                        reverse=True)[: 3]:
    print(node, val)
```

```
Top 3 nodes having the highest degree centrality
33 0.5151515151515151
0 0.48484848484848486
32 0.36363636363636365
```

```python
node_bet = nx.betweenness_centrality(G)

print('Top 3 nodes having the highest betweenness')
for node, val in sorted(node_bet.items(),
                        key=lambda x: x[1],
                        reverse=True)[: 3]:
    print(node, val)
```

```
Top 3 nodes having the highest betweenness
0 0.43763528138528146
33 0.30407497594997596
32 0.145247113997114
```

Also, closeness, eigenvector, PageRank, HITS, ...

[Official reference]

Coloring

Demo: [The four color map theorem]

[Official reference]

Communities

```python
G = nx.karate_club_graph()
```

```python
from networkx.algorithms import community
```

```python
print(community.kernighan_lin_bisection(G))
```

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

```python
for cmt in community.k_clique_communities(G, 4):  # clique percolation
    print(cmt)
```

```
frozenset({0, 1, 2, 3, 7, 13})
frozenset({32, 33, 8, 30})
frozenset({32, 33, 29, 23})
```

```
print(community.greedy_modularity_communities(G))
```

```
[frozenset({32, 33, 8, 14, 15, 18, 20, 22, 23, 24, 25, 26, 27, 28, 29, 30,
31}), frozenset({1, 2, 3, 7, 9, 12, 13, 17, 21}), frozenset({0, 4, 5, 6, 10,
11, 16, 19})]
```

```
H = community.LFR_benchmark_graph(100, 3, 2, average_degree=15,
                                  mu=0.3, min_community=40)
```

```
H.order(), H.size(), sum(nx.triangles(H).values()) / 3
```

```
(100, 920, 1536.0)
```

[Official reference]

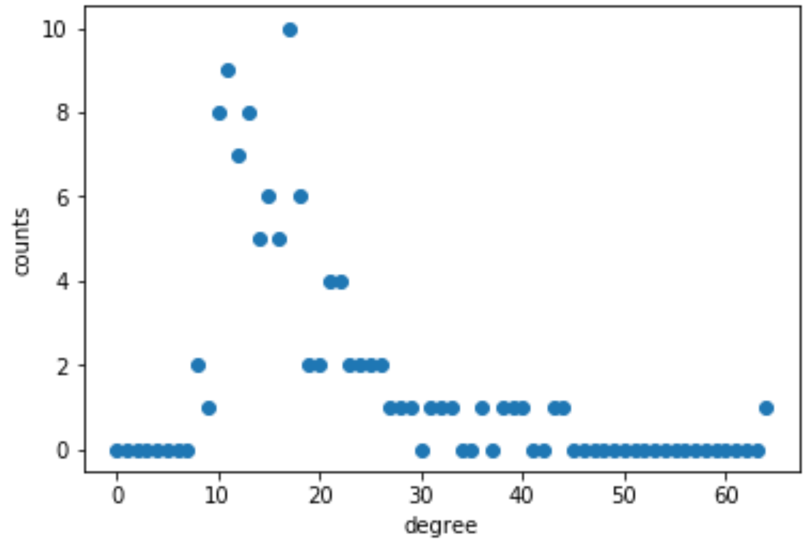
[NMI code]

Degree Distribution

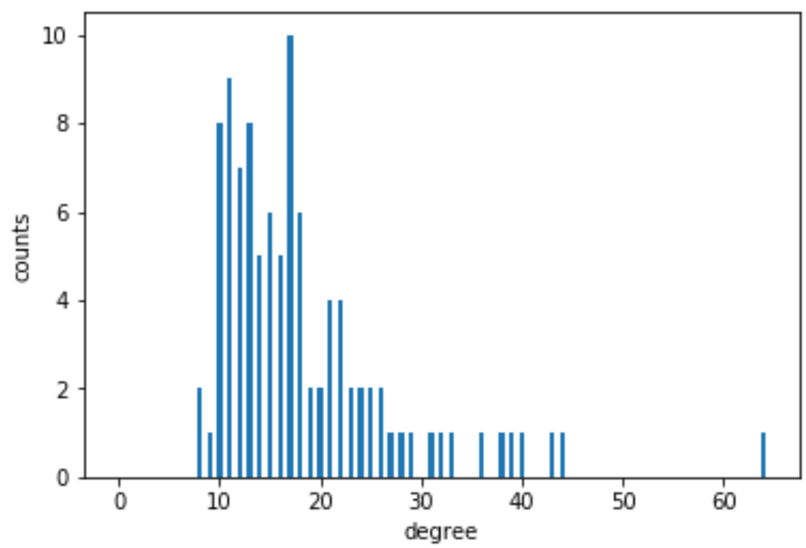
```
print(nx.degree_histogram(H))
```

```
[0, 0, 0, 0, 0, 0, 0, 0, 2, 1, 8, 9, 7, 8, 5, 6, 5, 10, 6, 2, 2, 4, 4, 2, 2,
2, 2, 1, 1, 1, 0, 1, 1, 1, 0, 0, 1, 0, 1, 1, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1]
```

```python
hist = nx.degree_histogram(H)
xs = range(len(hist))

plt.scatter(xs, hist)
plt.xlabel('degree');
plt.ylabel('counts');
```




```python
plt.bar(xs, hist, align='center', width=0.5)
# plt.xticks(xs);

plt.xlabel('degree');
plt.ylabel('counts');
```



Operators

```
G = nx.path_graph(10)
H = G.subgraph([1, 2])
I = nx.Graph()
I.add_edges_from([(13, 14)])

H = nx.complement(G)

I = nx.union(G, I)  # also check union_all
I = nx.intersection(G, H)  # also check intersection_all
```

Official reference

Shortest paths

official reference

Read-write

```
G = nx.read_edgelist
G = nx.read_gml
G = nx.read_gexf
G = nx.read_sparse6

nx.write_edgelist
nx.write_gml
nx.read_gexf
nx.to_numpy_array
nx.to_scipy_sparse_matrix```
```

Official reference

Thanks!