CHAPTER 1

1.3 Graph Creation

In [ ]:

```
import networkx as nx
import math

G = nx.Graph()
G.add_edge(1, 2)
G.add_edge(2, 3, weight=0.9)

G.add_edge('y', 'x', function=math.cos)
G.add_node(math.cos)

elist = [(1, 2), (2, 3), (1, 4), (4, 2)]
G.add_edges_from(elist)
elist = [('a', 'b', 5.0), ('b', 'c', 3.0), ('a', 'c', 1.0), ('c', 'd', 7.3)
G.add_weighted_edges_from(elist)
```

1.5 Algorithms

In [ ]:

```
G = nx.Graph()
e = [('a', 'b', 0.3), ('b', 'c', 0.9), ('a', 'c', 0.5), ('c', 'd', 1.2)]
G.add_weighted_edges_from(e)
print(nx.dijkstra_path(G, 'a', 'd'))
```

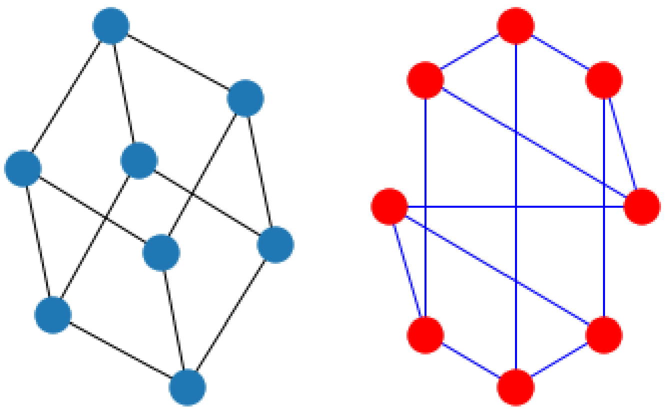
```
['a', 'c', 'd']
```

1.6 Drawing

In [ ]:

```
import matplotlib.pyplot as plt

G = nx.cubical_graph()
subax1 = plt.subplot(121)
nx.draw(G) # default spring_layout
subax2 = plt.subplot(122)
nx.draw(G, pos=nx.circular_layout(G), node_color='r', edge_color='b')
```



1.7 Data Structure

In [ ]:

```
G = nx.Graph()
G.add_edge('A', 'B')
G.add_edge('B', 'C')
print(G.adj)
```

```
{'A': {'B': {}}, 'B': {'A': {}, 'C': {}}, 'C': {'B': {}}}
```

In [ ]:

```
G = nx.Graph()
G.add_edge(1, 2, color='red', weight=0.84, size=300)
print(G[1][2]['size'])
print(G.edges[1, 2]['color'])
```

```
300
red
```

CHAPTER 2

2.2 Basic graph types

2.2.1 Graph—Undirected graphs with self loops

In [ ]:

```
G = nx.Graph()
G.add_node(1)
G.add_nodes_from([2, 3])
G.add_nodes_from(range(100, 110))
H = nx.path_graph(10)
G.add_nodes_from(H)
G.add_node(H)
G.add_edge(1, 2)
G.add_edges_from([(1, 2), (1, 3)])
G.add_edges_from(H.edges)
```

In [ ]:

```
G = nx.Graph(day="Friday")
G.graph
```

Out[ ]:

```
{'day': 'Friday'}
```

In [ ]:

```
G.add_node(1, time="5pm")
G.add_nodes_from([3], time="2pm")
G.nodes[1]
```

Out[ ]:

```
{'time': '5pm'}
```

In [ ]:

```
G.nodes[1]["room"] = 714 # node must exist already to use G.nodes
del G.nodes[1]["room"] # remove attribute
list(G.nodes(data=True))
```

Out[ ]:

```
[(1, {'time': '5pm'}), (3, {'time': '2pm'})]
```

In [ ]:

```
G.add_edge(1, 2, weight=4.7)
G.add_edges_from([(3, 4), (4, 5)], color="red")
G.add_edges_from([(1, 2, {"color": "blue"}), (2, 3, {"weight": 8})])
G[1][2]["weight"] = 4.7
G.edges[1, 2]["weight"] = 4
```

In [ ]:

```
1 in G # check if node in graph
```

Out[ ]:

```
True
```

In [ ]:

```
[n for n in G if n < 3] # iterate through nodes
```

Out[ ]:

```
[1, 2]
```

In [ ]:

```
len(G) # number of nodes in graph
```

Out[ ]:

```
5
```

In [ ]:

```
for n, nbrsdict in G.adjacency():
  for nbr, eattr in nbrsdict.items():
    if "weight" in eattr:
        # Do something useful with the edges
        pass

for u, v, weight in G.edges.data("weight"):
    if weight is not None:
        # Do something useful with the edges
        pass
```

In [ ]:

```
class ThinGraph(nx.Graph):
    all_edge_dict = {"weight": 1}

    def single_edge_dict(self):
        return self.all_edge_dict

    edge_attr_dict_factory = single_edge_dict

G = ThinGraph()
G.add_edge(2, 1)
```

In [ ]:

```
G[2][1]
```

Out[ ]:

```
{'weight': 1}
```

In [ ]:

```
G.add_edge(2, 2)
G[2][1] is G[2][2]
```

Out[ ]:

```
True
```

In [ ]:

```
G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
G = nx.Graph(name="my graph")
e = [(1, 2), (2, 3), (3, 4)] # list of edges
G = nx.Graph(e)
G = nx.Graph(e, day="Friday")
G.graph
```

Out[ ]:

```
{'day': 'Friday'}
```

In [ ]:

```
G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
G.add_node(1)
G.add_node("Hello")
K3 = nx.Graph([(0, 1), (1, 2), (2, 0)])
G.add_node(K3)
G.number_of_nodes()
```

Out[ ]:

```
3
```

In [ ]:

```
G.add_node(1, size=10)
G.add_node(3, weight=0.4, UTM=("13S", 382871, 3972649))
```

In [ ]:

```
G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
G.add_nodes_from("Hello")
K3 = nx.Graph([(0, 1), (1, 2), (2, 0)])
G.add_nodes_from(K3)
sorted(G.nodes(), key=str)
```

Out[ ]:

```
[0, 1, 2, 'H', 'e', 'l', 'o']
```

In [ ]:

```
G.add_nodes_from([1, 2], size=10)
G.add_nodes_from([3, 4], weight=0.4)

G.add_nodes_from([(1, dict(size=11)), (2, {"color": "blue"})])
G.nodes[1]["size"]
```

Out[ ]:

```
11
```

In [ ]:

```
H = nx.Graph()
H.add_nodes_from(G.nodes(data=True))
H.nodes[1]["size"]
```

Out[ ]:

```
11
```

In [ ]:

```
G = nx.path_graph(3) # or DiGraph, MultiGraph, MultiDiGraph, etc
list(G.edges)
```

Out[ ]:

```
[(0, 1), (1, 2)]
```

In [ ]:

```
G.remove_node(1)
list(G.edges)
```

Out[ ]:

```
[]
```

In [ ]:

```
G = nx.path_graph(3) # or DiGraph, MultiGraph, MultiDiGraph, etc
e = list(G.nodes)
e
```

Out[ ]:

```
[0, 1, 2]
```

In [ ]:

```
G.remove_nodes_from(e)
list(G.nodes)
```

Out[ ]:

```
[]
```

In [ ]:

```
G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
e = (1, 2)
G.add_edge(1, 2) # explicit two-node form
G.add_edge(*e) # single edge as tuple of two nodes
G.add_edges_from([(1, 2)]) # add edges from iterable container

G.add_edge(1, 2, weight=3)
G.add_edge(1, 3, weight=7, capacity=15, length=342.7)

G.add_edge(1, 2)
G[1][2].update({0: 5})
G.edges[1, 2].update({0: 5})
```

In [ ]:

```
G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
G.add_edges_from([(0, 1), (1, 2)]) # using a list of edge tuples
e = zip(range(0, 3), range(1, 4))
G.add_edges_from(e) # Add the path graph 0-1-2-3
```

```
G.add_edges_from([(1, 2), (2, 3)], weight=3)
G.add_edges_from([(3, 4), (1, 4)], label="WN2898")
```

In [ ]:

```
G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
G.add_weighted_edges_from([(0, 1, 3.0), (1, 2, 7.5)])
```

In [ ]:

```
G = nx.path_graph(4) # or DiGraph, etc
G.remove_edge(0, 1)
e = (1, 2)
G.remove_edge(*e) # unpacks e from an edge tuple
e = (2, 3, {"weight": 7}) # an edge with attribute data
G.remove_edge(*e[:2]) # select first part of edge tuple
```

In [ ]:

```
G = nx.path_graph(4) # or DiGraph, MultiGraph, MultiDiGraph, etc
ebunch = [(1, 2), (2, 3)]
G.remove_edges_from(ebunch)
```

In [ ]:

```
# dict-of-set/list/tuple
adj = {1: {2, 3}, 2: {1, 3}, 3: {1, 2}}
e = [(u, v) for u, nbrs in adj.items() for v in nbrs]
G.update(edges=e, nodes=adj)
```

In [ ]:

```
DG = nx.DiGraph()
# dict-of-dict-of-attribute
adj = {1: {2: 1.3, 3: 0.7}, 2: {1: 1.4}, 3: {1: 0.7}}
e = [
    (u, v, {"weight": d})
    for u, nbrs in adj.items()
    for v, d in nbrs.items()
]

DG.update(edges=e, nodes=adj)
e
```

Out[ ]:

```
[(1, 2, {'weight': 1.3}),
 (1, 3, {'weight': 0.7}),
 (2, 1, {'weight': 1.4}),
 (3, 1, {'weight': 0.7})]
```

In [ ]:

```
# dict-of-dict-of-dict
adj = {1: {2: {"weight": 1.3}, 3: {"color": 0.7, "weight": 1.2}}}
e = [
    (u, v, {"weight": d})
    for u, nbrs in adj.items()
    for v, d in nbrs.items()
]
DG.update(edges=e, nodes=adj)
e
```

Out[ ]:

```
[(1, 2, {'weight': {'weight': 1.3}}),
 (1, 3, {'weight': {'color': 0.7, 'weight': 1.2}})]
```

In [ ]:

```
# predecessor adjacency (dict-of-set)
pred = {1: {2, 3}, 2: {3}, 3: {3}}
e = [(v, u) for u, nbrs in pred.items() for v in nbrs]
e
```

Out[ ]:

```
[(2, 1), (3, 1), (3, 2), (3, 3)]
```

In [ ]:

```
# MultiGraph dict-of-dict-of-dict-of-attribute
MDG = nx.MultiDiGraph()
```

```
adj = {
    1: {2: {0: {"weight": 1.3}, 1: {"weight": 1.2}}},
    3: {2: {0: {"weight": 0.7}}},
}
e = [
    (u, v, ekey, d)
    for u, nbrs in adj.items()
    for v, keydict in nbrs.items()
    for ekey, d in keydict.items()
]
MDG.update(edges=e)
e
```

Out[ ]:

```
[(1, 2, 0, {'weight': 1.3}),
 (1, 2, 1, {'weight': 1.2}),
 (3, 2, 0, {'weight': 0.7})]
```

In [ ]:

```
G = nx.path_graph(5)
G.update(nx.complete_graph(range(4, 10)))
from itertools import combinations
edges = (
    (u, v, {"power": u * v})
    for u, v in combinations(range(10, 20), 2)
    if u * v < 225
)
nodes = [1000] # for singleton, use a container
G.update(edges, nodes)
```

In [ ]:

```
G = nx.path_graph(4) # or DiGraph, MultiGraph, MultiDiGraph, etc
G.clear()
list(G.nodes)
```

Out[ ]:

```
[]
```

In [ ]:

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

Out[ ]:

```
[]
```

In [ ]:

```
G = nx.path_graph(4) # or DiGraph, MultiGraph, MultiDiGraph, etc
G.clear_edges()
list(G.nodes)
```

Out[ ]:

```
[0, 1, 2, 3]
```

In [ ]:

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

Out[ ]:

```
[]
```

In [ ]:

```
G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
e = (1, 2)
G.add_edge(1, 2) # explicit two-node form
G.add_edge(*e) # single edge as tuple of two nodes
G.add_edges_from([(1, 2)]) # add edges from iterable container
```

In [ ]:

```
G.add_edge(1, 2, weight=3)
G.add_edge(1, 3, weight=7, capacity=15, length=342.7)

G.add_edge(1, 2)
G[1][2].update({0: 5})
G.edges[1, 2].update({0: 5})
```

In [ ]:

```
G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
G.add_edges_from([(0, 1), (1, 2)]) # using a list of edge tuples
e = zip(range(0, 3), range(1, 4))
G.add_edges_from(e) # Add the path graph 0-1-2-3

G.add_edges_from([(1, 2), (2, 3)], weight=3)
G.add_edges_from([(3, 4), (1, 4)], label="WN2898")

G.edges
```

Out[ ]:

```
EdgeView([(0, 1), (1, 2), (1, 4), (2, 3), (3, 4)])
```

In [ ]:

```
G = nx.Graph() # or DiGraph, MultiGraph, MultiDiGraph, etc
G.add_weighted_edges_from([(0, 1, 3.0), (1, 2, 7.5)])
G.nodes
```

Out[ ]:

```
NodeView((0, 1, 2))
```

In [ ]:

```
G.edges
```

Out[ ]:

```
EdgeView([(0, 1), (1, 2)])
```

In [ ]:

```
G = nx.path_graph(4) # or DiGraph, etc
G.remove_edge(0, 1)
e = (1, 2)
G.remove_edge(*e) # unpacks e from an edge tuple
e = (2, 3, {"weight": 7}) # an edge with attribute data
G.remove_edge(*e[:2]) # select first part of edge tuple
G.edges
```

Out[ ]:

```
EdgeView([])
```

In [ ]:

```
G = nx.path_graph(4) # or DiGraph, MultiGraph, MultiDiGraph, etc
ebunch = [(1, 2), (2, 3)]
G.remove_edges_from(ebunch)
G.edges
```

Out[ ]:

```
EdgeView([(0, 1)])
```

In [ ]:

```
# dict-of-set/list/tuple
adj = {1: {2, 3}, 2: {1, 3}, 3: {1, 2}}
e = [(u, v) for u, nbrs in adj.items() for v in nbrs]
G.update(edges=e, nodes=adj)

DG = nx.DiGraph()
# dict-of-dict-of-attribute
adj = {1: {2: 1.3, 3: 0.7}, 2: {1: 1.4}, 3: {1: 0.7}}
e = [
    (u, v, {"weight": d})
    for u, nbrs in adj.items()
    for v, d in nbrs.items()
]
DG.update(edges=e, nodes=adj)


# dict-of-dict-of-dict
adj = {1: {2: {"weight": 1.3}, 3: {"color": 0.7, "weight": 1.2}}}
e = [
    (u, v, {"weight": d})
    for u, nbrs in adj.items()
    for v, d in nbrs.items()
]
```

```
DG.update(edges=e, nodes=adj)


# predecessor adjacency (dict-of-set)
pred = {1: {2, 3}, 2: {3}, 3: {3}}
e = [(v, u) for u, nbrs in pred.items() for v in nbrs]

# MultiGraph dict-of-dict-of-dict-of-attribute
MDG = nx.MultiDiGraph()
adj = {
    1: {2: {0: {"weight": 1.3}, 1: {"weight": 1.2}}},
    3: {2: {0: {"weight": 0.7}}},
}
e = [
    (u, v, ekey, d)
    for u, nbrs in adj.items()
    for v, keydict in nbrs.items()
    for ekey, d in keydict.items()
]
MDG.update(edges=e)

G = nx.path_graph(5)
G.update(nx.complete_graph(range(4, 10)))
from itertools import combinations
edges = (
    (u, v, {"power": u * v})
    for u, v in combinations(range(10, 20), 2)
    if u * v < 225
)
nodes = [1000] # for singleton, use a container
G.update(edges, nodes)
```

In [ ]:

```
G = nx.path_graph(4) # or DiGraph, MultiGraph, MultiDiGraph, etc
G.clear()
list(G.nodes)
```

Out[ ]:

```
[]
```

In [ ]:

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

Out[ ]:

```
[]
```

In [ ]:

```
G = nx.path_graph(4) # or DiGraph, MultiGraph, MultiDiGraph, etc
G.clear_edges()
list(G.nodes)
```

Out[ ]:

```
[0, 1, 2, 3]
```

In [ ]:

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

Out[ ]:

```
[]
```

In [ ]:

```
G = nx.path_graph(3)
list(G.nodes)
```

Out[ ]:

```
[0, 1, 2]
```

In [ ]:

```
list(G)
```

Out[ ]:

```
[0, 1, 2]
```

In [ ]:

```
G.add_node(1, time="5pm")
G.nodes[0]["foo"] = "bar"
list(G.nodes(data=True))
```

Out[ ]:

```
[(0, {'foo': 'bar'}), (1, {'time': '5pm'}), (2, {})]
```

In [ ]:

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

Out[ ]:

```
[(0, {'foo': 'bar'}), (1, {'time': '5pm'}), (2, {})]
```

In [ ]:

```
list(G.nodes(data="foo"))
```

Out[ ]:

```
[(0, 'bar'), (1, None), (2, None)]
```

In [ ]:

```
list(G.nodes.data("foo"))
```

Out[ ]:

```
[(0, 'bar'), (1, None), (2, None)]
```

In [ ]:

```
list(G.nodes(data="time"))
```

Out[ ]:

```
[(0, None), (1, '5pm'), (2, None)]
```

In [ ]:

```
list(G.nodes.data("time"))
```

Out[ ]:

```
[(0, None), (1, '5pm'), (2, None)]
```

In [ ]:

```
list(G.nodes(data="time", default="Not Available"))
```

Out[ ]:

```
[(0, 'Not Available'), (1, '5pm'), (2, 'Not Available')]
```

In [ ]:

```
list(G.nodes.data("time", default="Not Available"))
```

Out[ ]:

```
[(0, 'Not Available'), (1, '5pm'), (2, 'Not Available')]
```

In [ ]:

```
G = nx.Graph()
G.add_node(0)
G.add_node(1, weight=2)
G.add_node(2, weight=3)
dict(G.nodes(data="weight", default=1))
```

Out[ ]:

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

In [ ]:

```
G = nx.path_graph(4)
[n for n in G]
```

Out[ ]:

```
[0, 1, 2, 3]
```

In [ ]:

```
list(G)
```

Out[ ]:

```
[0, 1, 2, 3]
```

In [ ]:

```
G = nx.path_graph(3) # or DiGraph, MultiGraph, MultiDiGraph, etc
G.has_node(0)
```

Out[ ]:

```
True
```

In [ ]:

```
0 in G
```

Out[ ]:

```
True
```

In [ ]:

```
G = nx.path_graph(4)
1 in G
```

Out[ ]:

```
True
```

In [ ]:

```
G = nx.path_graph(3) # or MultiGraph, etc
G.add_edge(2, 3, weight=5)
[e for e in G.edges]
```

Out[ ]:

```
[(0, 1), (1, 2), (2, 3)]
```

In [ ]:

```
G.edges.data()
```

Out[ ]:

```
EdgeDataView([(0, 1, {}), (1, 2, {}), (2, 3, {'weight': 5})])
```

In [ ]:

```
G.edges.data("weight", default=1)
```

Out[ ]:

```
EdgeDataView([(0, 1, 1), (1, 2, 1), (2, 3, 5)])
```

In [ ]:

```
G.edges([0, 3]) # only edges from these nodes
```

Out[ ]:

```
EdgeDataView([(0, 1), (3, 2)])
```

In [ ]:

```
G.edges(0)
```

Out[ ]:

```
EdgeDataView([(0, 1)])
```

In [ ]:

```
G = nx.path_graph(4) # or DiGraph, MultiGraph, MultiDiGraph, etc
G.has_edge(0, 1) # using two nodes
```

Out[ ]:

```
True
```

In [ ]:

```
e = (0, 1)
G.has_edge(*e) # e is a 2-tuple (u, v)
```

Out[ ]:

```
True
```

In [ ]:

```
e = (0, 1, {"weight": 7})
G.has_edge(*e[:2]) # e is a 3-tuple (u, v, data_dictionary)
```

Out[ ]:

```
True
```

In [ ]:

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

Out[ ]:

```
True
```

In [ ]:

```
1 in G[0] # though this gives KeyError if 0 not in G
```

Out[ ]:

```
True
```

In [ ]:

```
G = nx.path_graph(4) # or DiGraph, MultiGraph, MultiDiGraph, etc
G[0][1]
```

Out[ ]:

```
{}
```

In [ ]:
```
G[0][1]["weight"] = 7
G[0][1]["weight"]
```

Out[ ]:
```
7
```

In [ ]:
```
G[1][0]["weight"]
```

Out[ ]:
```
7
```

In [ ]:
```
G = nx.path_graph(4) # or DiGraph, MultiGraph, MultiDiGraph, etc
G.get_edge_data(0, 1) # default edge data is {}
```

Out[ ]:
```
{}
```

In [ ]:
```
e = (0, 1)
G.get_edge_data(*e) # tuple form
```

Out[ ]:
```
{}
```

In [ ]:
```
G.get_edge_data("a", "b", default=0) # edge not in graph, return 0
## Parado na pagina 26
```

Out[ ]:
```
0
```

2.3 Graph Views

2.3.2 subgraph_view

In [ ]:
```
G = nx.path_graph(6)
```

In [ ]:
```
def filter_node(n1):
    return n1 != 5

view = nx.subgraph_view(G, filter_node=filter_node)
view.nodes()
```

Out[ ]:
```
NodeView((0, 1, 2, 3, 4))
```

In [ ]:
```
G[3][4]["cross_me"] = False

def filter_edge(n1, n2):
    return G[n1][n2].get("cross_me", True)

view = nx.subgraph_view(G, filter_edge=filter_edge)
view.edges()
```

Out[ ]:
```
EdgeView([(0, 1), (1, 2), (2, 3), (4, 5)])
```

In [ ]:
```
view = nx.subgraph_view(G, filter_node=filter_node, filter_edge=filter_edge
view.nodes()
```

Out[ ]:
```
NodeView((0, 1, 2, 3, 4))
```

In [ ]:
```
view.edges()
```

Out[ ]:
```
EdgeView([(0, 1), (1, 2), (2, 3)])
```

2.3.3 reverse_view

In [ ]:

```
G = nx.DiGraph()
G.add_edge(1, 2)
G.add_edge(2, 3)
G.edges()
```

Out[ ]:

```
OutEdgeView([(1, 2), (2, 3)])
```

In [ ]:

```
view = nx.reverse_view(G)
view.edges()
```

Out[ ]:

```
OutEdgeView([(2, 1), (3, 2)])
```