Basics-networkX

September 15, 2025

1 Basics - NetworkX

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
import networkx as nx
```

1.0.1 Creating a graph

Nodes can be any hashable object e.g., a text string, an image, an XML object, another Graph, a customized node object, etc.

[2]:
```
G = nx.Graph() # an empty undirected graph
```

Draw graphs via `nx.draw(G, pos=None, with_labels=False, node_size=300, node_color='r', edge_color='k', font_size=12, font_color='k')`

- `G`: The graph object (created using nx.Graph() or similar)
- `pos`: Node positions. By default, it uses spring layout (nx.spring_layout())
- `with_labels`: If True, displays node labels
- `node_size`: Size of the nodes (default: 300)
- `node_color`: Color of the nodes (default: 'r' for red)
- `edge_color`: Color of the edges (default: 'k' for black)
- `font_size`: Size of node labels (default: 12)
- `font_color`: Color of node labels (default: 'k' for black)
- `style`: Style of edges (solid, dashed, dotted)
- `alpha`: Transparency (0 = transparent, 1 = opaque)

  The indexes of a graph should equal to those of its positions!

1.0.2 Nodes

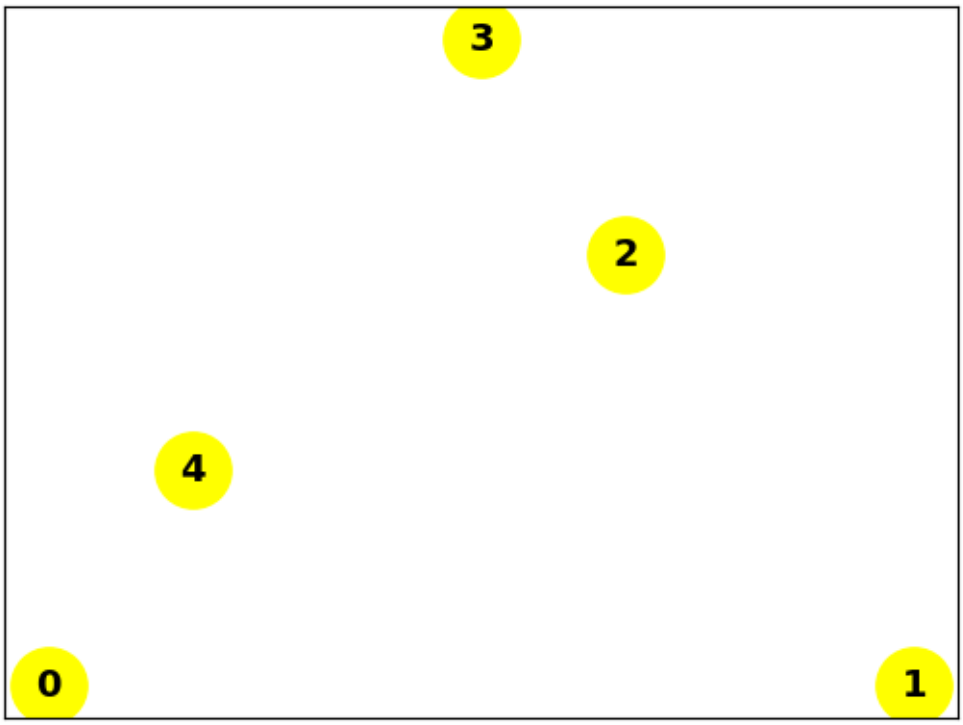
NetworkX includes many graph generator functions and facilities to read and write graphs in many formats.

[3]:
```
G.clear()
G.add_node(0) # add a single node
G.add_nodes_from([1, 2]) # from iterable container
G.add_nodes_from([(3, {"color": "red"}), (4, {"color": "green"})])

print(G.nodes, len(G)) # print the nodes
```

```
pos = np.zeros((len(G), 2))
for i in np.arange(len(G)):
    pos[i,0] = 0.3 * i

fig, ax = plt.subplots(1,1, figsize=(5, 2))
nx.draw_networkx( G, pos = pos, with_labels=True, node_size=800,␣
 ↪node_color='yellow',
        font_size=14, font_weight='bold')
ax.set_axis_off()
```

```
[0, 1, 2, 3, 4] 5
```

[4]:
```
pos = nx.planar_layout(G)
nx.draw_networkx( G, pos = pos, with_labels=True, node_size=800,␣
 ↪node_color='yellow',
        font_size=14, font_weight='bold')
ax.set_axis_off()
```



1.0.3 Edges

[5]:
```
G.add_edge(1,2) # add a single edge

e = (2, 3)
G.add_edge(*e) # Eq. G.add_edge(2,3), unpack edge tuple*

e = [(3,1), (3,4), (3,5)]
G.add_edges_from( e ) # iterable container

print(f'All node indexes: {G.nodes}') # print the edges

fig, ax = plt.subplots(1,1, figsize=(5, 2))
nx.draw( G, with_labels=True, node_size=600, node_color='yellow',
        font_size=14, font_weight='bold')
print(f'{G.number_of_nodes()} nodes, {G.number_of_edges()} edges')
```

```
All node indexes: [0, 1, 2, 3, 4, 5]
6 nodes, 5 edges
```

The order of adjacency reporting (e.g., `G.adj`, `G.successors`, `G.predecessors`) is the order of edge addition.

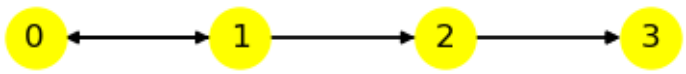
[6]:
```
print(G.adj) # adjacency matrix as a dictionary

# Get the adjacency matrix as a SciPy sparse matrix
adj_matrix_sparse = nx.adjacency_matrix(G)
print(adj_matrix_sparse)

# If you want it as a dense array
adj_matrix_dense = adj_matrix_sparse.toarray()

print(adj_matrix_dense)

# Get the adjacency matrix as a NumPy array
adj_matrix = nx.to_numpy_array(G)

print("Adjacency matrix:")
print(adj_matrix)
```

```
{0: {}, 1: {2: {}, 3: {}}, 2: {1: {}, 3: {}}, 3: {2: {}, 1: {}, 4: {}, 5: {}},
4: {3: {}}, 5: {3: {}}}
<Compressed Sparse Row sparse array of dtype 'int64'
        with 10 stored elements and shape (6, 6)>
  Coords        Values
  (1, 2)        1
  (1, 3)        1
  (2, 1)        1
  (2, 3)        1
  (3, 1)        1
  (3, 2)        1
  (3, 4)        1
  (3, 5)        1
  (4, 3)        1
  (5, 3)        1
[[0 0 0 0 0 0]
```

```
 [0 0 1 1 0 0]
 [0 1 0 1 0 0]
 [0 1 1 0 1 1]
 [0 0 0 1 0 0]
 [0 0 0 1 0 0]]
Adjacency matrix:
[[0. 0. 0. 0. 0. 0.]
 [0. 0. 1. 1. 0. 0.]
 [0. 1. 0. 1. 0. 0.]
 [0. 1. 1. 0. 1. 1.]
 [0. 0. 0. 1. 0. 0.]
 [0. 0. 0. 1. 0. 0.]]
```

[7]:
```
DG = nx.DiGraph()   # create a directed graph

DG.add_edge(1, 0)   # adds the nodes in order (2, 1)
DG.add_edge(0, 2)
DG.add_edge(1, 3)
DG.add_edge(0, 1)
DG.add_edge(2, 3)

print(DG.edges)
assert list(DG.successors(2)) == [3]
assert list(DG.edges) == [(1, 0), (1, 3), (0, 2), (0, 1), (2, 3)] # Caution the␣
 ↪ordering in the list

pos = np.zeros((len(DG), 2))
for i in np.arange(len(DG)):
    pos[i,0] = 0.3 * i

fig, ax = plt.subplots(1,1, figsize=(5, 2))
nx.draw_networkx(DG, pos=pos, with_labels=True, node_size=500,␣
 ↪node_color='yellow',
        edge_color='k', font_size=12, font_color='k')
ax.set_axis_off()
```

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



1.0.4 Examining elements of a graph

- Basic graph properties facilitate reporting: `G.nodes`, `G.edges`, `G.adj` and `G.degree`
- These are set-like views of the nodes, edges, neighbors (adjacencies), and degrees of nodes in a graph.
- We can look up node and edge data attributes via the views and iterate with data attributes using methods: `.items()`, `.data()`.

[8]:
```
list(G.edges)
list(G.adj[1])
list(G.neighbors(1))

for t in G.nodes.items():
    print(t)

G.degree[1] # # of connected nodes to node 1

print(G.adj)
```

```
(0, {})
(1, {})
(2, {})
(3, {'color': 'red'})
(4, {'color': 'green'})
(5, {})
{0: {}, 1: {2: {}, 3: {}}, 2: {1: {}, 3: {}}, 3: {2: {}, 1: {}, 4: {}, 5: {}},
4: {3: {}}, 5: {3: {}}}
```

1.0.5 Removing elements from a graph

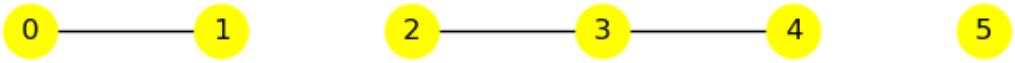
- remove nodes and edges from the graph in a similar fashion to adding
  - `G.remove_node()`
  - `G.remove_nodes_from()`
  - `Graph.remove_edge()`
  - `Graph.remove_edges_from()`

[9]:
```
G = nx.path_graph(5)

pos = np.zeros((len(G), 2))
for i in np.arange(len(G)):
    pos[i,0] = 0.3 * i

fig, ax = plt.subplots(1,1, figsize=(5, 2))
nx.draw_networkx( G, pos = pos, with_labels=True, node_size=800,␣
 ↪node_color='yellow',
```

```
        font_size=14, font_weight='bold')
ax.set_axis_off()
```

[10]:
```
G.add_node(5)
G.remove_edge(1,2)
print(G.nodes)
pos = np.zeros((len(G), 2))
for i in np.arange(len(G)):
    pos[i,0] = 0.3 * i
nx.draw(G, pos=pos, with_labels=True, node_size=500, node_color='yellow',
        edge_color='k', font_size=12, font_color='k')
```

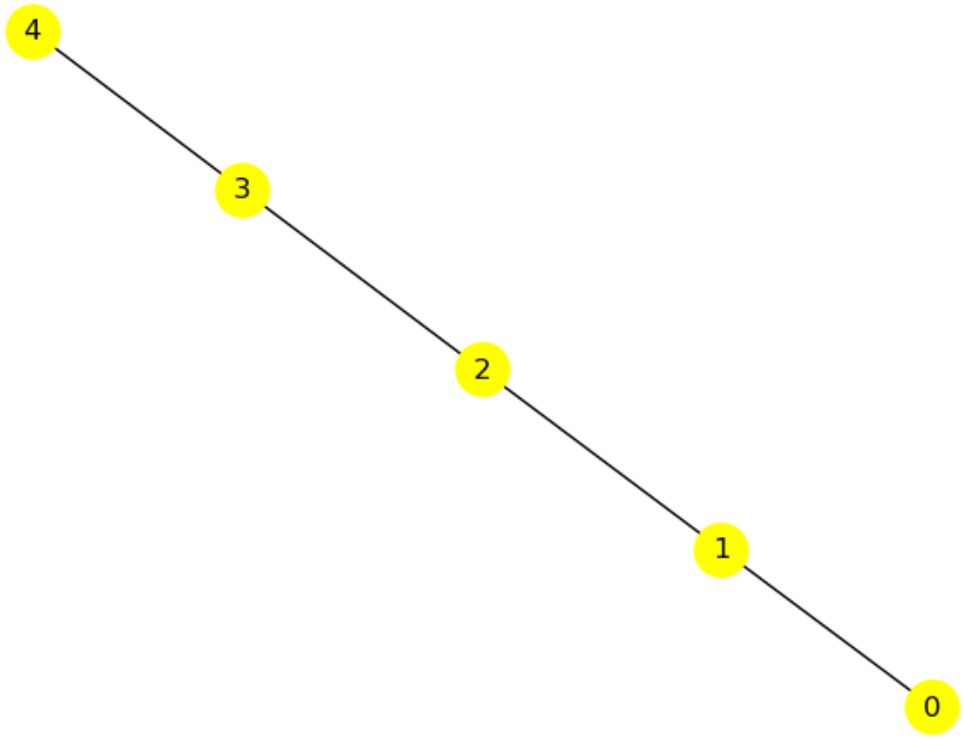
```
[0, 1, 2, 3, 4, 5]
```



[11]:
```
G.remove_nodes_from((1, 3))

nx.draw(G, pos=None, with_labels=True, node_size=500, node_color='yellow',
        edge_color='k', font_size=12, font_color='k')
```

[12]:
```
G = nx.path_graph(5)  # Creates a path graph with 5 nodes (0-4)

# Use 'nbunch' as a single node
neighbors = list(G.neighbors(2))  # Neighbors of node 2
print(neighbors)  # Output: [1, 3]

# Use 'nbunch' as a list of nodes
degrees = G.degree([0, 1, 2])  # Degree of nodes 0, 1, and 2
print(degrees)  # Output: [(0, 1), (1, 2), (2, 2)]

# Use 'nbunch' as a subset of nodes
subgraph = G.subgraph([1, 2, 3])  # Create a subgraph with nodes 1, 2, 3
print(subgraph.nodes)  # Output: [1, 2, 3]

nx.draw(G, pos=None, with_labels=True, node_size=500, node_color='yellow',
        edge_color='k', font_size=12, font_color='k')
```

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



[13]:
```
# Add multiple edges to a graph using `add_edges_from`
# Create an empty graph
G1 = nx.Graph()

edges = [(0, 1), (1, 2), (2, 0), (2, 3)]

# Add edges from the ebunch
G1.add_edges_from(edges)

# Print edges
print(G1.edges())

G1.clear()

# weighted graph
edges_with_weights = [(0, 1, {'weight': 4}), (1, 2, {'weight': 5}), (2, 3,␣
 ↪{'weight': 6})]

# Add edges with weights
G1.add_edges_from(edges_with_weights)
```

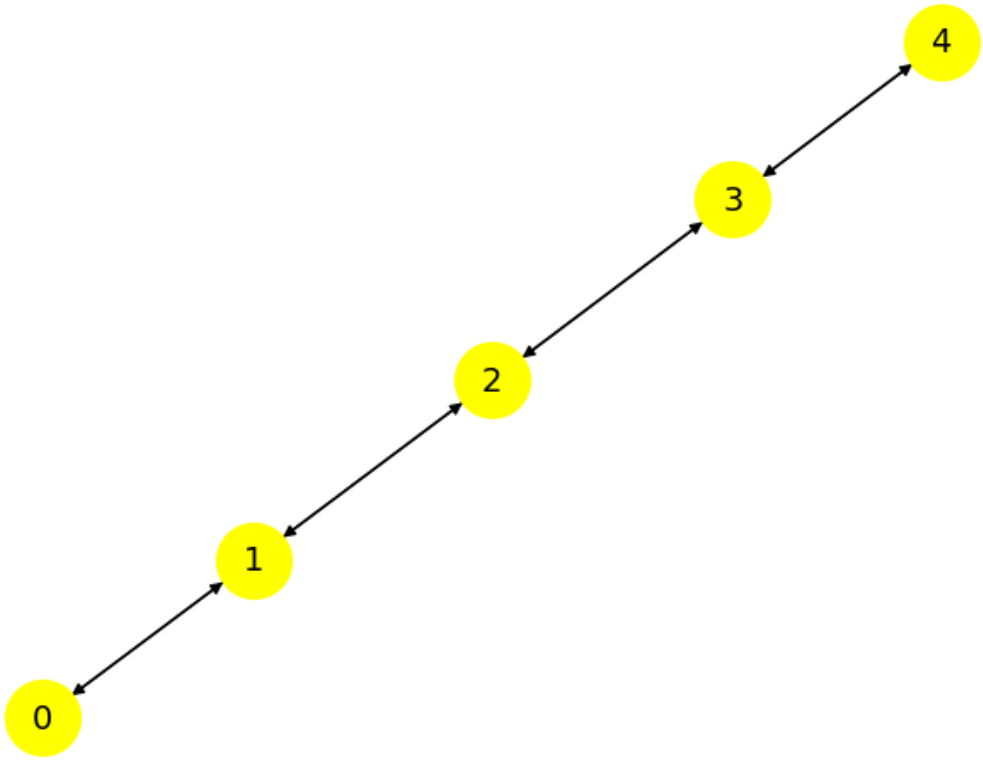
```
# Access edge attributes
print(G1[0][1]['weight'])  # Output: 4

# Example: Checking the edge data for a specific ebunch
specific_edges = [(0, 1), (1, 2)]
for u, v in G1.edges:
    print(f"{u}, {v}")
print(G1.edges)
```

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

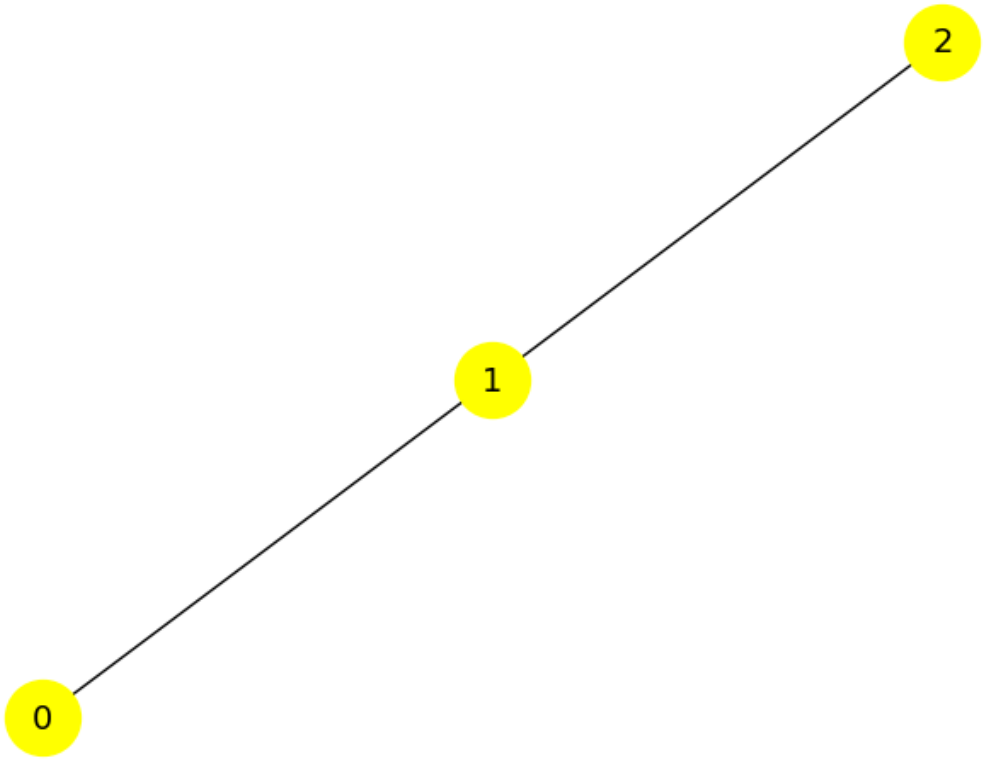
1.0.6 Using the graph constructors

- Graph objects do not have to be built up incrementally.
- Data specifying graph structure can be passed directly to the constructors of the various graph classes.

[14]:
```
H = nx.DiGraph(G)

H.add_nodes_from((1, 3))

e = [(0, 1), (2, 1), (2,3), (4,3)]
H.add_edges_from(e)

nx.draw(H, with_labels=True, node_size=1000, node_color='yellow',
        edge_color='k', font_size=14, font_color='k')
```



[15]:
```
adjacency_dict = {0: {1:{}}, 1: {2:{}}, 2: {1:{}}} # adjacency dictionary
H = nx.Graph(adjacency_dict)  # create a Graph dict mapping nodes to nbrs
H.edges()

nx.draw(H, with_labels=True, node_size=1000, node_color='yellow',
        edge_color='k', font_size=14, font_color='k')
```



1.0.7 What to use as nodes and edges

- Nodes and edges have meaningful items.
- The most common choices are numbers or strings, but a node can be any hashable object (except `None`), and an edge can be associated with any object `x` using `G.add_edge(n1, n2, object=x)`.
- Consider using `convert_node_labels_to_integers()` to obtain a more traditional graph with integer labels

1.0.8 Accessing edges and neighbors

In addition to the views `G.edges`, and `G.adj`, access to edges and neighbors is possible using subscript notation.

[16]:
```
G = nx.Graph([(1, 2, {"color": "yellow"})])
print(G[1])  # same as G.adj[1]
print(G[2])  # same as G.adj[1]
print(G[1][2]) # same as G.edges[1,2]
```

```
{2: {'color': 'yellow'}}
{1: {'color': 'yellow'}}
{'color': 'yellow'}
```

We can get / set the attributes of an edge using subscript notation if the edge already exists.

[17]:
```
G.add_edge(1, 3)
G[1][3]['color'] = "blue"
G.edges[1, 2]['color'] = "red"

print( G.edges[1,3] )
print( G.edges[1,2] )
print( G.edges[2,1] )
```

```
{'color': 'blue'}
{'color': 'red'}
{'color': 'red'}
```

- Fast examination of all (node, adjacency) pairs is achieved using `G.adjacency()`, or `G.adj.items()`.
- Note that for undirected graphs, adjacency iteration sees each edge twice.

[18]:
```
H = nx.Graph()
H.add_weighted_edges_from([(1, 2, 0.125), (1, 3, 0.75), (2, 4, 1.2), (3, 4, 0.
↪375)])
for n, nbrs in H.adj.items():
   for nbr, eattr in nbrs.items():
       wt = eattr['weight']
       if wt < 0.5: print(f"({n}, {nbr}, {wt:.3})")
```

```
(1, 2, 0.125)
(2, 1, 0.125)
(3, 4, 0.375)
(4, 3, 0.375)
```

Convenient access to all edges is achieved with the edges property.

[19]:
```
for (u, v, wt) in H.edges.data('weight'):
    if wt < 0.5:
        print(f"({u}, {v}, {wt:.3})")
```

```
(1, 2, 0.125)
(3, 4, 0.375)
```

1.0.9 Adding attributes to graphs, nodes, and edges

- Each graph, node, and edge can hold key / value attribute pairs in an associated attribute dictionary (the keys must be hashable).
- By default these are empty, but attributes can be added or changed using `add_edge`, `add_node` or direct manipulation of the attribute dictionaries named `G.graph`, `G.nodes`, and `G.edges` for a graph G.

[20]:
```
G = nx.Graph(day='Friday') # graph attribute
print( G.graph )

G.graph['day'] = 'Sunday'
print( G.graph )
```

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

[21]:
```
G.add_node(1, time='5pm') # node attribute
G.add_nodes_from([3], time='2pm')
G.nodes[1]['room'] = '405'

print( G.nodes[1] )
print( G.nodes.data() )
```

```
{'time': '5pm', 'room': '405'}
[(1, {'time': '5pm', 'room': '405'}), (3, {'time': '2pm'})]
```

[22]:
```
G.add_edge(1, 2, weight=4.7 ) # edge attributes
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[3, 4]['weight'] = 4.2

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

```
[(1, 2, {'weight': 4.7, 'color': 'blue'}), (3, 4, {'color': 'red', 'weight':
4.2}), (3, 2, {'weight': 8}), (4, 5, {'color': 'red'})]
```

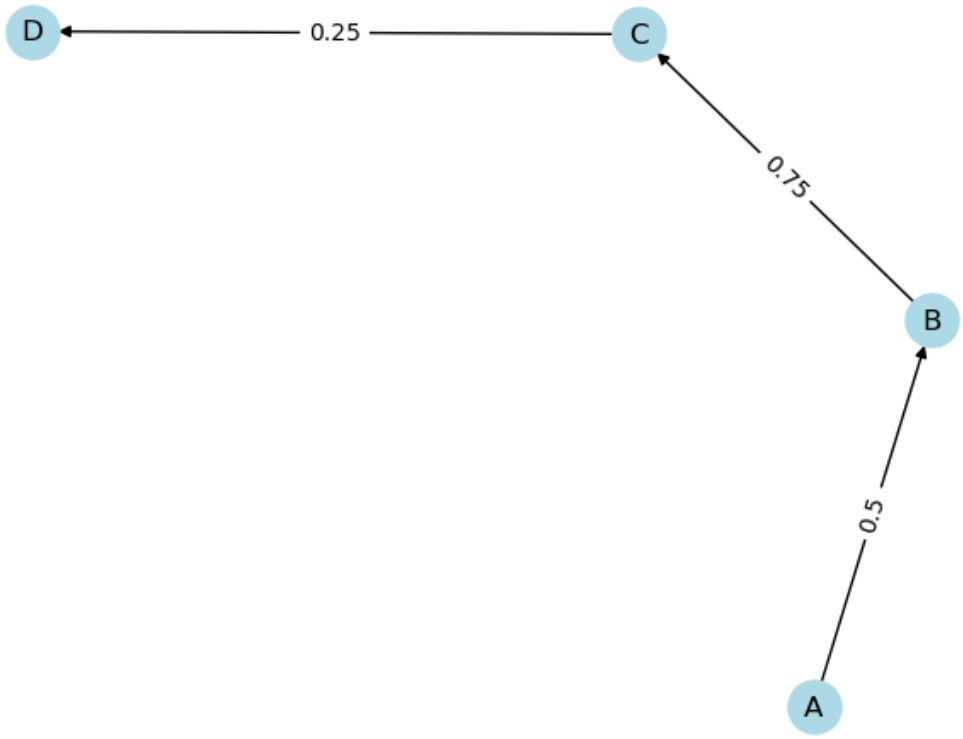
1.0.10 Directed graphs

- The `DiGraph` class provides additional methods and properties specific to directed edges: `DiGraph.out_edges`, `DiGraph.in_degree`, `DiGraph.predecessors()`, `DiGraph.successors()`.
- To allow algorithms to work with both classes easily, the directed versions of neighbors is equivalent to successors while DiGraph.degree reports the sum of DiGraph.in_degree and DiGraph.out_degree even though that may feel inconsistent at times.

[23]:
```
DG = nx.DiGraph()
DG.add_weighted_edges_from([('A', 'B', 0.5),    # or add_edge('A','B',weight=0.
 ↪5)
                            ('B', 'C', 0.75),
                            ('C', 'D', 0.25)])

DG.out_degree( 'A', weight='weight')

# Draw the graph
```

```
pos = nx.spring_layout(DG)  # Position nodes for better visualization
nx.draw(DG, pos, with_labels=True, node_size=500, node_color="lightblue")

# Extract edge weights
edge_labels = nx.get_edge_attributes(DG, 'weight')  # Get weights as labels
nx.draw_networkx_edge_labels(DG, pos, edge_labels = edge_labels)
```

[23]:
```
{('A', 'B'): Text(0.47038141259654576, -0.4519446346901067, '0.5'),
 ('B', 'C'): Text(0.31657980222968796, 0.18054806290763414, '0.75'),
 ('C', 'D'): Text(-0.4703789538304335, 0.4519466657422191, '0.25')}
```



[24]:
```
print( DG.degree('A', weight='weight') )
print( list(DG.successors('A')) )
print( list(DG.neighbors('A')) )
```

```
0.5
['B']
['B']
```

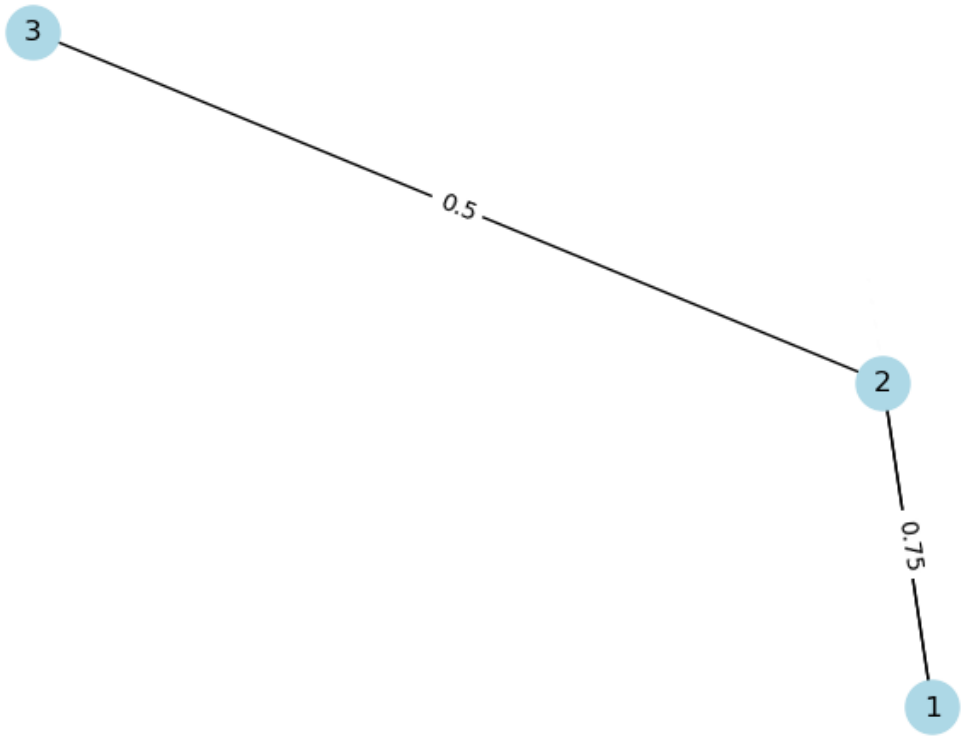
1.0.11 Multigraphs

- NetworkX provides classes for graphs which allow multiple edges between any pair of nodes.
- The `MultiGraph` and `MultiDiGraph` classes allow us to add the same edge twice, possibly with different edge data.

[25]:
```
MG = nx.MultiGraph()
MG.add_weighted_edges_from([(1, 2, 0.5), (1, 2, 0.75), (2, 3, 0.5)])
print( dict(MG.degree(weight='weight')) )

# Draw the graph
pos = nx.spring_layout(MG)  # Position nodes for better visualization
nx.draw(MG, pos, with_labels=True, node_size=500, node_color="lightblue")

# Extract edge weights
edge_labels = {(u, v): MG[u][v][k]['weight'] for u, v, k in MG.edges(keys=True)}
nx.draw_networkx_edge_labels(MG, pos, edge_labels = edge_labels)
```

```
{1: 1.25, 2: 1.75, 3: 0.5}
```

[25]:
```
{(1, 2): Text(0.28491982322256115, -0.49999874262570965, '0.75'),
 (2, 3): Text(-0.1545108925172094, 0.48746352620976996, '0.5')}
```



1.0.12 Graph generators and graph operators

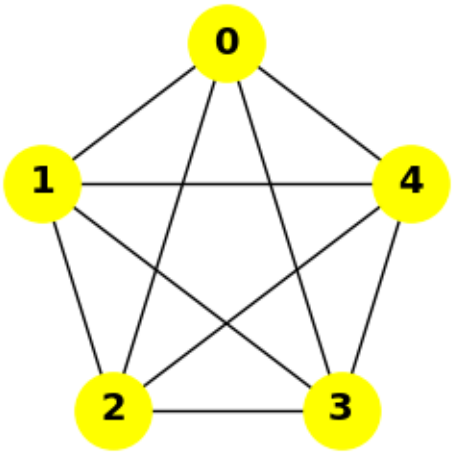
Constructing graphs node-by-node or edge-by-edges:

1. Classic graph operations
   - `subgraph(G, nbunch)` Returns the subgraph induced on nodes in nbunch.
   - `union(G, H[, rename])` Combine graphs G and H.
   - `disjoint_union(G, H)` Combine graphs G and H.
   - `cartesian_product(G, H)` Returns the Cartesian product of G and H.
   - `compose(G, H)` Compose graph G with H by combining nodes and edges into a single graph.
   - `complement(G)` Returns the graph complement of G.
   - `create_empty_copy(G[, with_data])` Returns a copy of the graph G with all of the edges removed.
   - `to_undirected(graph)` Returns an undirected view of the graph graph.
   - `to_directed(graph)` Returns a directed view of the graph graph.
2. Using a call to one of the classic small graphs
   - `petersen_graph([create_using])` Returns the Petersen graph.
   - `tutte_graph([create_using])` Returns the Tutte graph.
   - `sedgewick_maze_graph([create_using])` Return a small maze with a cycle.
   - `tetrahedral_graph([create_using])` Returns the 3-regular Platonic Tetrahedral graph.
3. Using a (constructive) generator for a classic graph
   - `complete_graph(n[, create_using])` Return the complete graph K_n with n nodes.
   - `complete_bipartite_graph(n1, n2[, create_using])` Returns the complete bipartite graph K_{n_1,n_2}.
   - `barbell_graph(m1, m2[, create_using])` Returns the Barbell Graph: two complete graphs connected by a path.
   - `lollipop_graph(m, n[, create_using])` Returns the Lollipop Graph; K_m connected to P_n.
4. Using a stochastic graph generator
   - `erdos_renyi_graph(n, p[, seed, directed, ...])` Returns a random graph, also known as an Erdős-Rényi graph or a binomial graph.
   - `watts_strogatz_graph(n, k, p[, seed, ...])` Returns a Watts–Strogatz small-world graph.
   - `barabasi_albert_graph(n, m[, seed, ...])` Returns a random graph using Barabási–Albert preferential attachment
   - `random_lobster(n, p1, p2[, seed, create_using])` Returns a random lobster graph.
5. Reading a graph stored in a file using common graph formats supports many popular formats, such as edge lists, adjacency lists, GML, GraphML, LEDA and others.

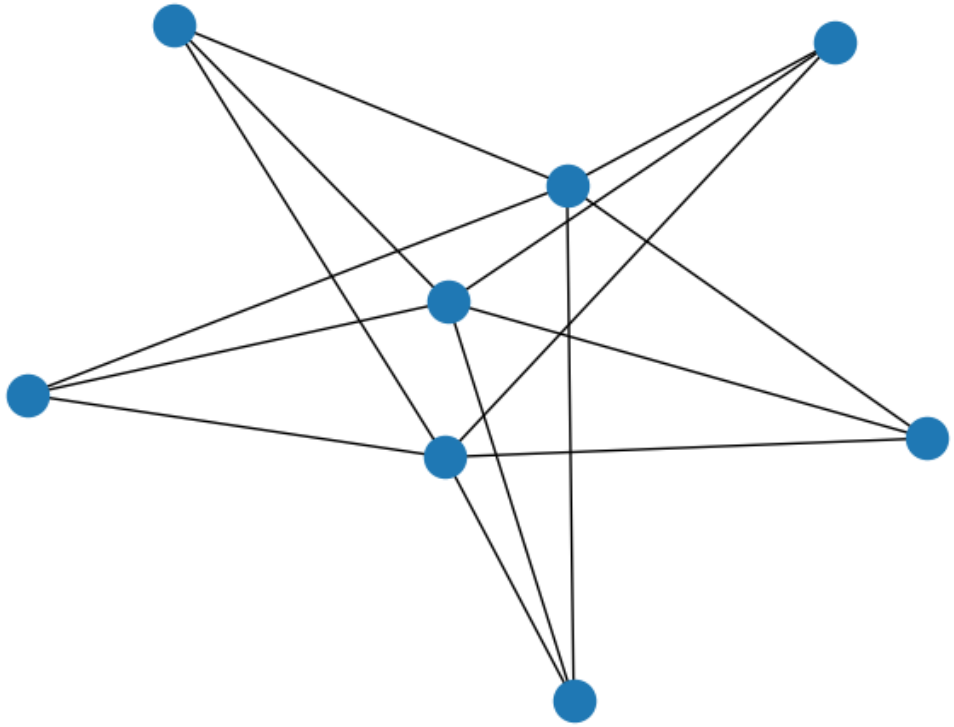
[26]:
```
import networkx as nx
import numpy as np
import matplotlib.pyplot as plt

# set drawing position
N = 5
```

```
t = np.linspace(np.pi/2, 2.5*np.pi, N+1)
print(t)
pos = np.zeros((N+1, 2))
pos[:, 0] = np.cos(t)
pos[:, 1] = np.sin(t)

# generate the graph
K_5 = nx.complete_graph(N)

# draw the draw
fig, ax = plt.subplots(1, 1, figsize=(3,3))
nx.draw_networkx(K_5, pos = pos, with_labels=True, node_size=800,␣
 ↪node_color='yellow',
        font_size=14, font_weight='bold')
ax.set_axis_off()
```

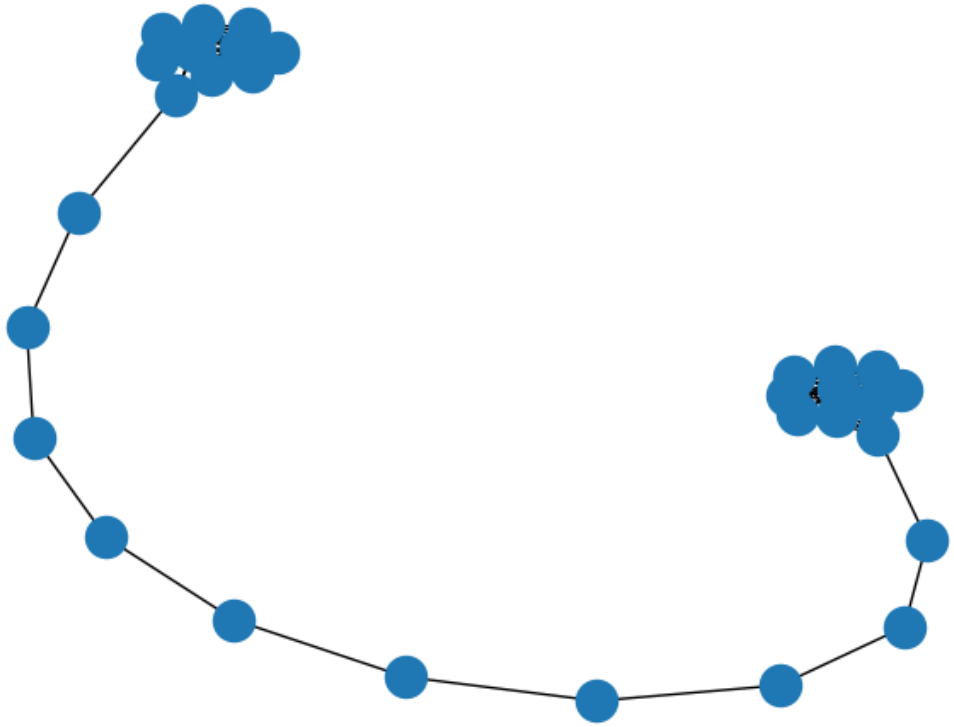
```
[1.57079633 2.82743339 4.08407045 5.34070751 6.59734457 7.85398163]
```



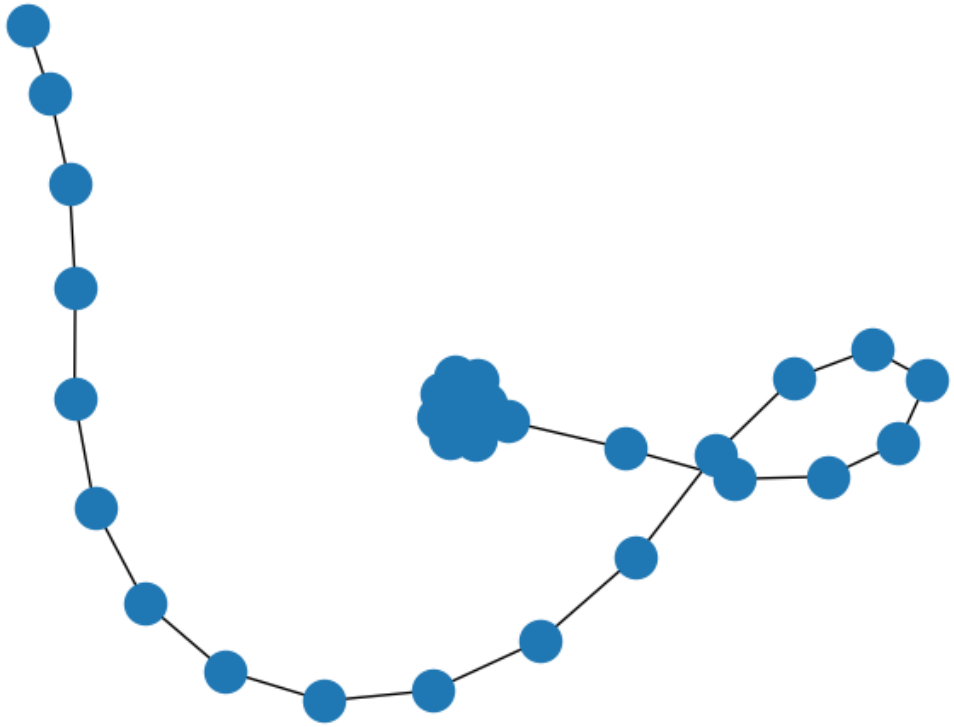
[27]:
```
K_3_5 = nx.complete_bipartite_graph(3, 5)
nx.draw(K_3_5)
```



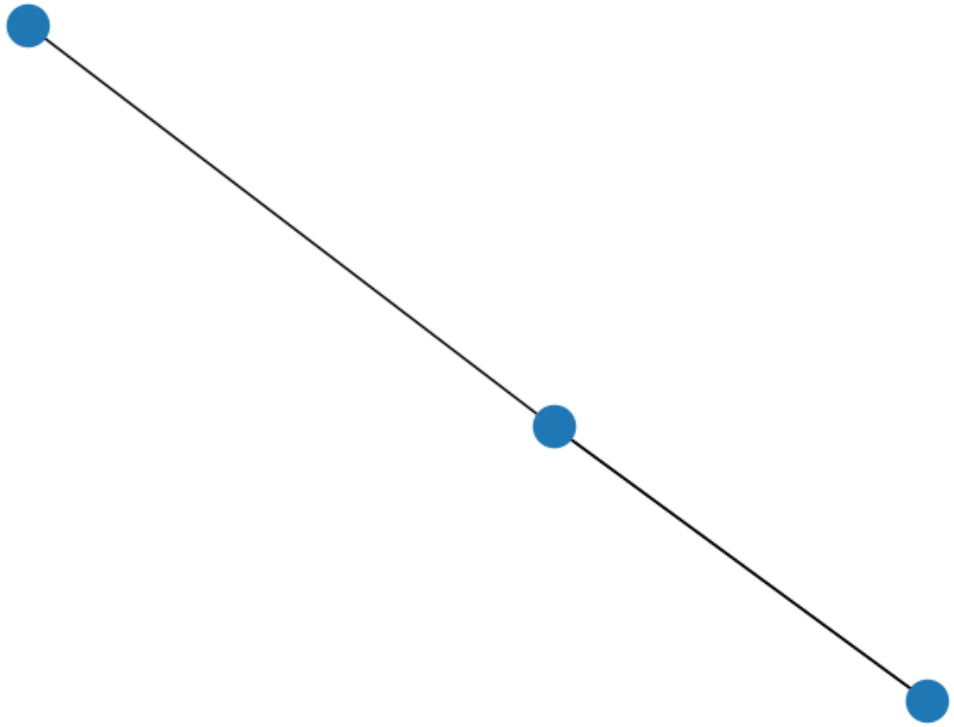
[28]:
```
barbell = nx.barbell_graph(10, 10)
nx.draw(barbell)
```



[29]:
```
lollipop = nx.lollipop_graph(10, 20)
nx.draw(lollipop)
```



[30]:
```
nx.write_gml(MG, 'path.to.find')     # store MG in a GML file
MG = nx.read_gml( 'path.to.find' )   # read a GML file
nx.draw(MG)
```



1.0.13 Analyzing graphs

[31]:
```
G = nx.Graph()
G.add_edges_from([(1, 2), (1, 3)])
G.add_node(4)       # adds node "spam"

print(list(nx.connected_components(G)))

print( sorted(d for n, d in G.degree()) )

print(nx.clustering(G))

nx.draw(G)

sp = dict(nx.all_pairs_shortest_path(G))
sp[3]
```

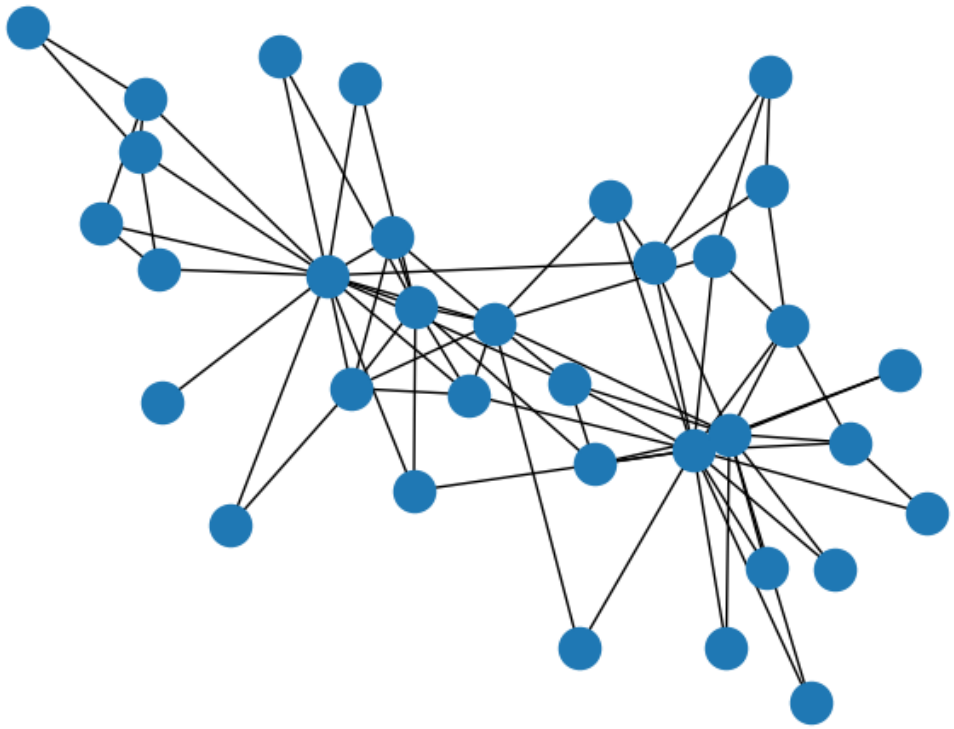
```
[{1, 2, 3}, {4}]
[0, 1, 1, 2]
{1: 0, 2: 0, 3: 0, 4: 0}
```

[31]: {3: [3], 1: [3, 1], 2: [3, 1, 2]}

1.0.14 Using NetworkX backends

- Be configured to use separate thrid-party backends to improve performance and add functionality.
- Backends are optional, installed separately, and can be enabled either directly in the user's code or through environment variables.

[32]:
```
import networkx as nx

#nx.config.backend_priority = ["fast_backend"] # set an environment variable
#print( nx.config.backend_priority)
G = nx.karate_club_graph()
pr = nx.pagerank(G)  # runs using backend from NETWORKX_BACKEND_PRIORITY, if set
nx.draw(G)
```

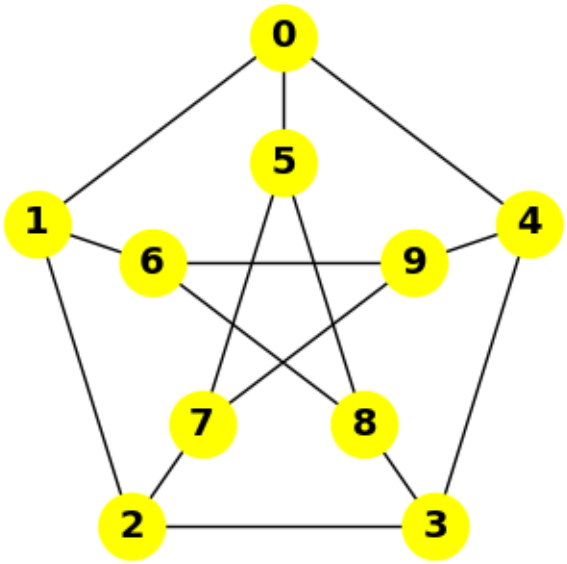


1.0.15 Drawing graphs

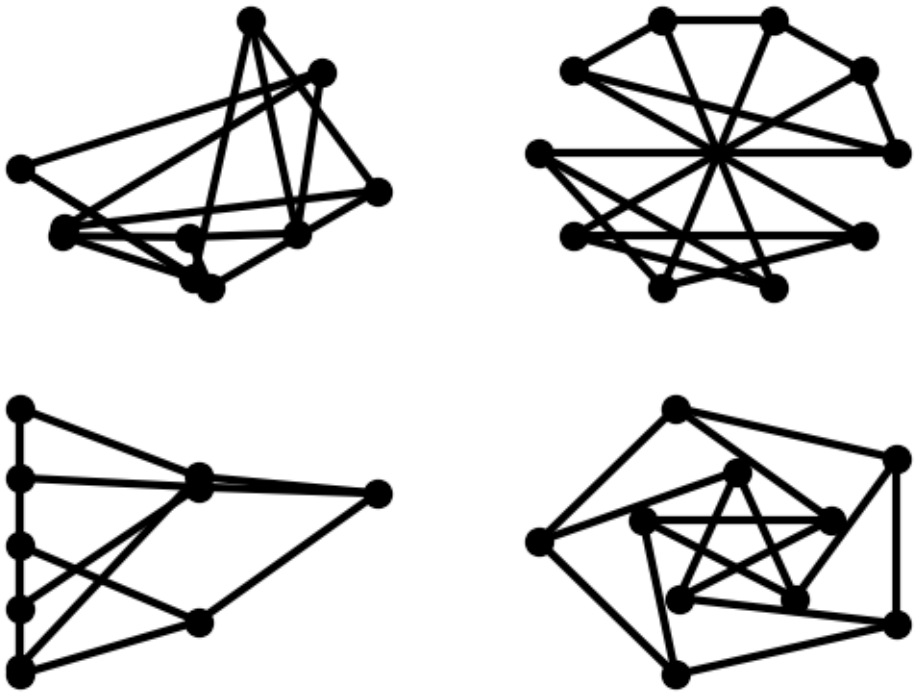
[33]:
```
import networkx as nx
import numpy as np
import matplotlib.pyplot as plt

# set drawing position
N = 10
M = int(N / 2)

print(f'N = {N}, M={M}')

t = np.linspace(np.pi/2, 2.5*np.pi, M+1)
pos = np.zeros((N, 2))
pos[0:M, 0] = 1.5 * np.cos(t)[0:M]
pos[0:M, 1] = 1.5 * np.sin(t)[0:M]

pos[M:N, 0] = 0.8 * np.cos(t)[0:M]
pos[M:N, 1] = 0.8 * np.sin(t)[0:M]
```

```
G = nx.petersen_graph()

fig, ax =  plt.subplots(1, 1, figsize=(4,4))
nx.draw_networkx(G, pos = pos, with_labels=True, node_size=600,␣
 ↪node_color='yellow',
       font_size=14, font_weight='bold')
ax.set_axis_off()
```

```
N = 10, M=5
```



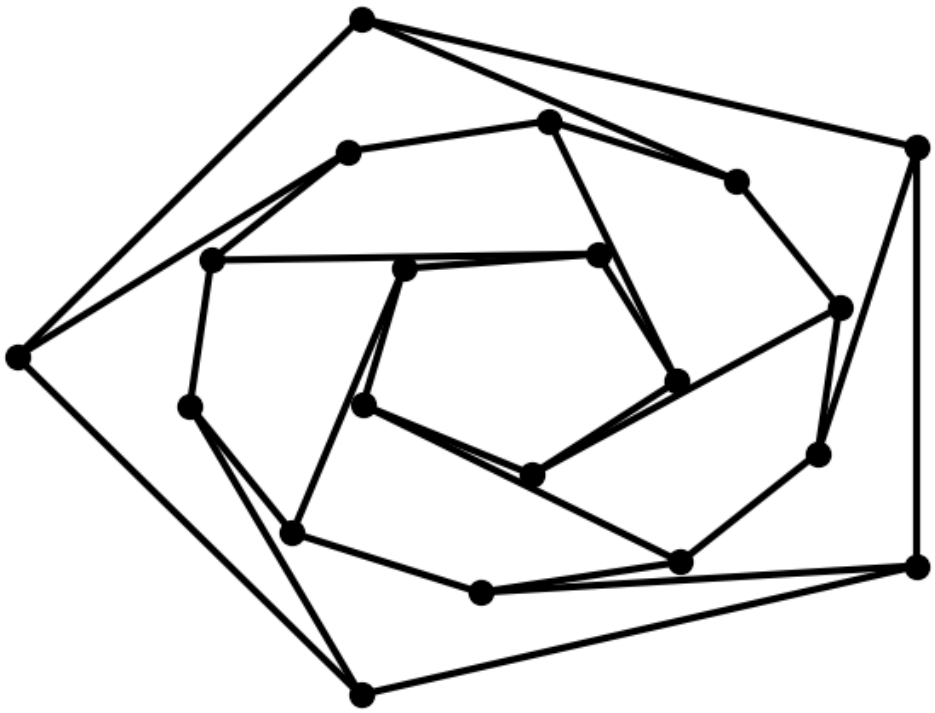
[34]:
```
options = {
    'node_color': 'black',
    'node_size': 100,
    'width': 3,
}
subax1 = plt.subplot(221)
nx.draw_random(G, **options)
subax2 = plt.subplot(222)
nx.draw_circular(G, **options)
subax3 = plt.subplot(223)
nx.draw_spectral(G, **options)
subax4 = plt.subplot(224)
nx.draw_shell(G, nlist=[range(5,10), range(5)], **options)
```



Check out additional options via `draw_networkx()` and layouts via the `layout module`

Use multiple shells with `draw_shell()`

[35]:
```
G = nx.dodecahedral_graph()
shells = [[2, 3, 4, 5, 6], [8, 1, 0, 19, 18, 17, 16, 15, 14, 7], [9, 10, 11,␣
 ↪12, 13]]
nx.draw_shell(G, nlist=shells, **options)
```



[36]:
```
nx.draw(G)
plt.savefig('path.png')
```