Lecture-8-introduction_to_python_graphs

July 9, 2021

1 Lecture 8:

1.1 Announcement: Project Guidelines will be posted on Monday.

- Start thinking about a project.
- You will have to do a real world modelling and present.
- Max 2 participants per group
- More details on Monday

1.1.1 Homework for Week 4 posted

1.2 Graphs in Python

1.2.1 Creating a graph

Today we will use the NetworkX module to draw graphs in Python. Please `!pip install networkx` if you do not have networkx installed. Lets begin by creating a graph G

[161]:

```python
import networkx as nx
G = nx.Graph()
```

1.2.2 Adding Nodes

Now that we have an empty graph G, let us add vertices and edges to G. In `networkx` vertices are known as nodes. Nodes can be added individually or from a list. They may be numbered by integers or strings.

[162]:

```python
list_nodes = range(1,9)
G.add_nodes_from(list_nodes)
```

[164]:

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

[164]:

```
[1, 2, 3, 4, 5, 6, 7, 8]
```

1.2.3 Adding edges

Just like nodes, edges can also be added one at a time, for from a list of tuples. Use `add_edge()` or `add_edges_from()` respectively.

[166]:

```
G.add_edge(1,3)
listedges = [(1,4), (2,3), (4,5), (5,9), (9,6), (9,8), (6,7), (6,8), (7,8)]
G.add_edges_from(listedges)
```

[167]:

```
G.edges
```

[167]:

```
EdgeView([(1, 3), (1, 4), (2, 3), (4, 5), (5, 9), (6, 9), (6, 7), (6, 8), (7,
8), (8, 9)])
```

[ ]:

```

```

1.2.4 Removing Nodes and Edges

Instead of adding edges or nodes we can remove by `remove_node()` or `remove_edge()` or remove a list of nodes or edges by `remove_nodes_from()` or `remove_edges_from()`

[168]:

```
G.remove_node(3)
```

[169]:

```
G.nodes
```

[169]:

```
NodeView((1, 2, 4, 5, 6, 7, 8, 9))
```

[170]:

```
G.edges
```

[170]:

```
EdgeView([(1, 4), (4, 5), (5, 9), (6, 9), (6, 7), (6, 8), (7, 8), (8, 9)])
```

1.2.5 Information about the graph

To check out the number of nodes or edges use `number_of_nodes()` or `number_of_edges()` respectively. Moreover `nodes` and `edges` stores the nodes and edges explicitly

[171]:

```
print(G.number_of_nodes())
print(G.number_of_edges())
print(G.nodes)
print(G.edges)
```

```
8
8
[1, 2, 4, 5, 6, 7, 8, 9]
[(1, 4), (4, 5), (5, 9), (6, 9), (6, 7), (6, 8), (7, 8), (8, 9)]
```

We can also view the neighbours via `G.adj` and know the degree of a node via `G.degree`.

[177]:

```
print(G.adj)
print(G.degree[6])
```

```
{1: {4: {}}, 2: {}, 4: {1: {}, 5: {}}, 5: {4: {}, 9: {}}, 6: {9: {}, 7: {}, 8:
{}}, 7: {6: {}, 8: {}}, 8: {9: {}, 6: {}, 7: {}}, 9: {5: {}, 6: {}, 8: {}}}
3
```

1.3 Drawing the graph

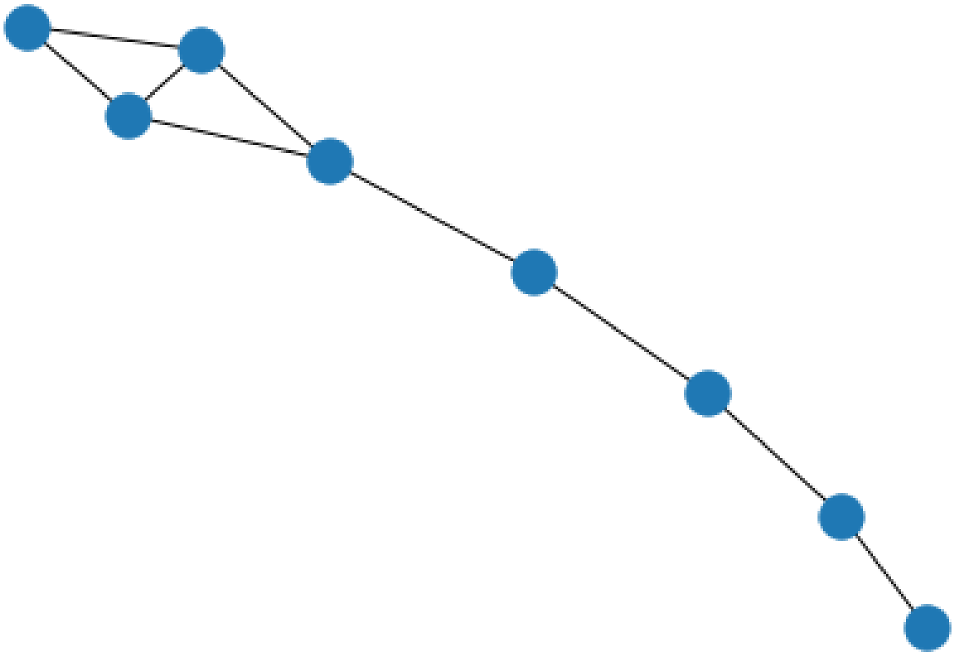
The simplest way to draw is by `networkx.draw()` function. But for more control it is preferred to use it with `matplotlib`. To install `matplotlib` use `!pip install matplotlib`

[180]:

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

[184]:

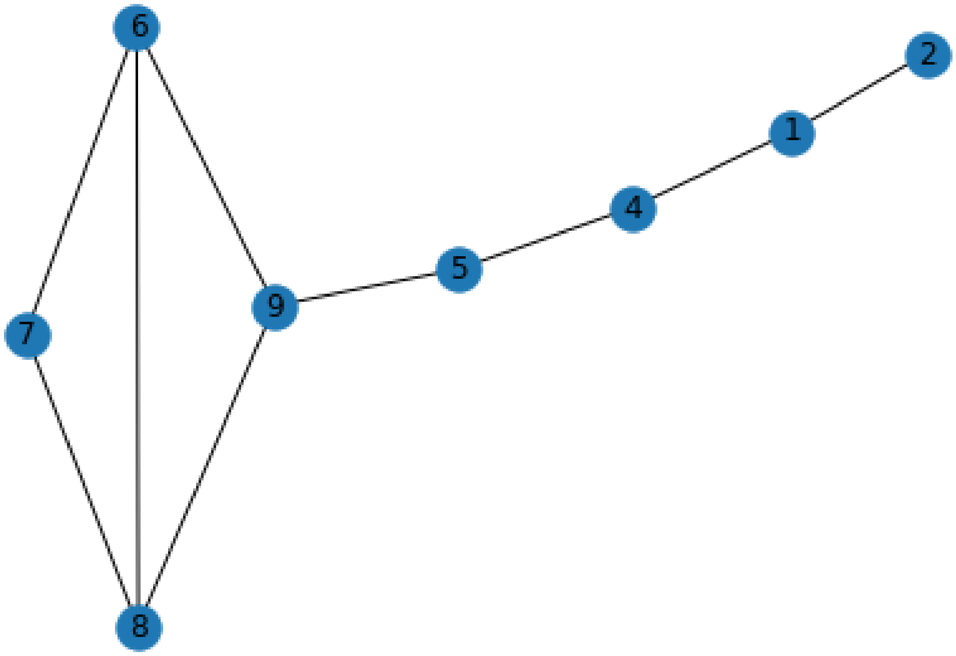
```
nx.draw(G)
```



To draw with the node labels, use the `with_labels` flag.

[185]:

```
nx.draw(G, with_labels=True)
```



We observe that the `draw` function draws a different graph everytime. To get more control we can use other types of graph drawing. For more info check the documentation for drawing

1.3.1 Adding attributes to graphs

Networkx stores the graph as a dictionary of dictionaries. This allows us to add multiple attributes to the edges (say weights or color or anything else). #### Node attributes We can access the attributes of a node by `G.nodes[node]`

Suppose in the above graph the nodes represent classes or timings we can add that info during graph creation.

```
[188]: G.add_node(10, time='9 AM')
       G.nodes[1]
```

```
[188]: {}
```

All other nodes do not have that time attribute yet, for example node 7

```
[191]: G.nodes[7]
       G.add_node(3)
       G.add_edge(3,2)
```

[195]:

```
G.nodes[1]['color'] = 'red'
G.nodes[2]['time'] = '1 PM'
G.nodes[3]['time'] = '2 PM'
```

[196]:

```
G.nodes[3]
```

[196]:

```
{'time': '2 PM'}
```

[197]:

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

[197]:

```
NodeDataView({1: {'time': '12 PM', 'color': 'red'}, 2: {'time': '1 PM'}, 4: {},
5: {}, 6: {}, 7: {}, 8: {}, 9: {}, 10: {'time': '9 AM'}, 3: {'time': '2 PM'}})
```

Edge attributes Similar to node attributes, edge attributes can be added during edge creation or via `G.edges`

[198]:

```
G.add_edge(10,3, weight=5)
```

[202]:

```
G[6][7]['color'] = 'orange'
G[6][8]['weight'] = 2
G[9][6]['weight'] = 1
```

[203]:

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

[203]:

```
EdgeDataView([(1, 4, {}), (1, 2, {}), (2, 3, {}), (4, 5, {}), (5, 9, {}), (6, 9,
{'weight': 1}), (6, 7, {'weight': 4, 'color': 'orange'}), (6, 8, {'weight': 2}),
(7, 8, {}), (8, 9, {}), (10, 3, {'weight': 5})])
```

1.3.2 Application: Coloring

Let us try to use edge and node attributes to color graphs. To have an interesting graph we will use a predetermined graph say the peterson graph. We use a random coloring, so we import numpy and for visualization we use matplotlib. To install these use `!pip install numpy` for example

[204]:

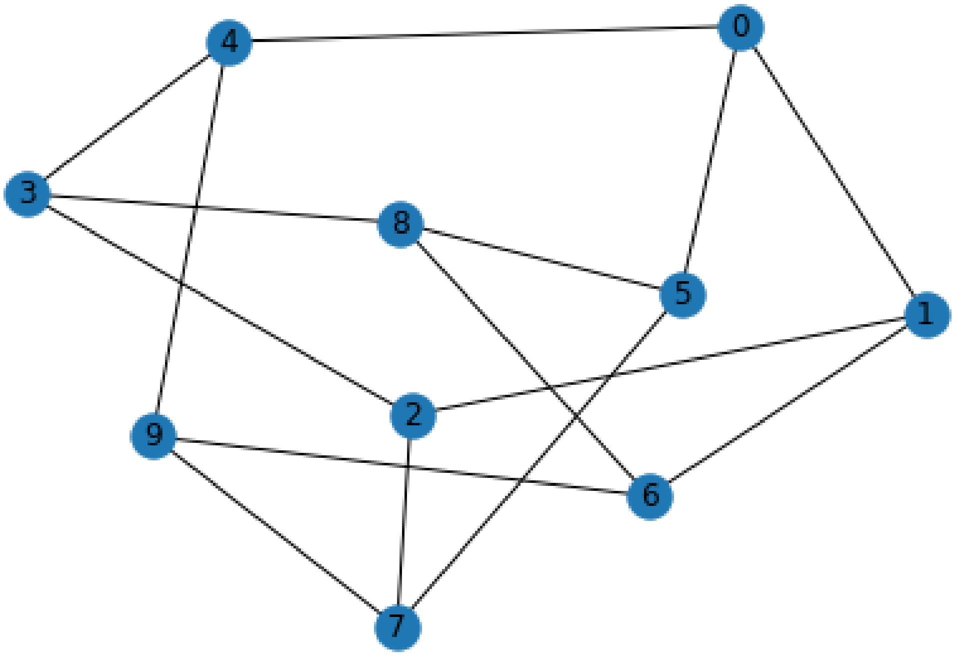
```
import numpy as np
import matplotlib.pyplot as plt
import networkx as nx
```

[205]:

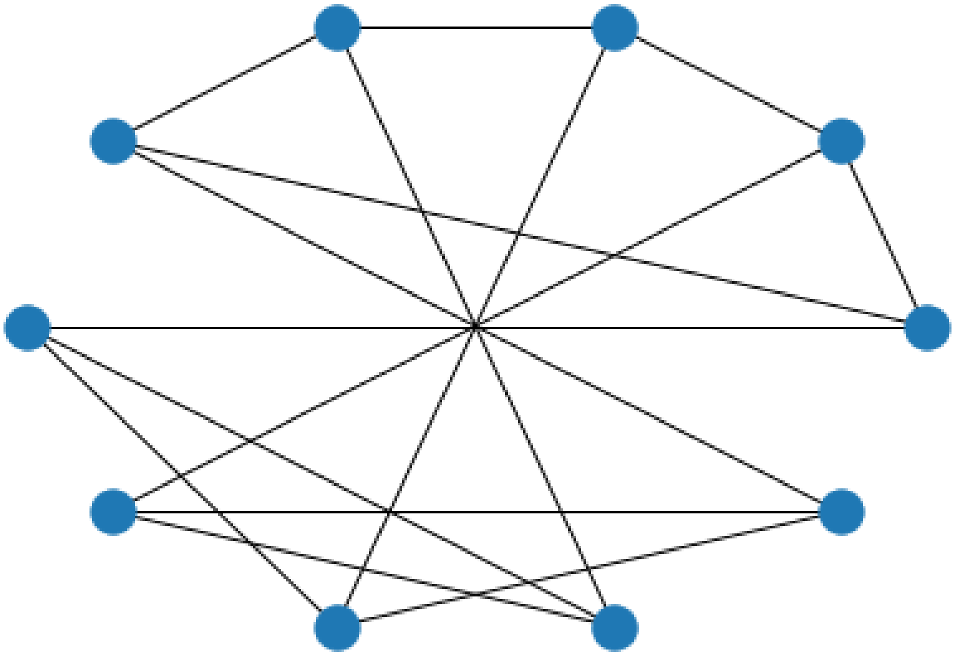
```
G = nx.petersen_graph()
```

[206]:

```
plt.figure()
nx.draw(G, with_labels=True)
```



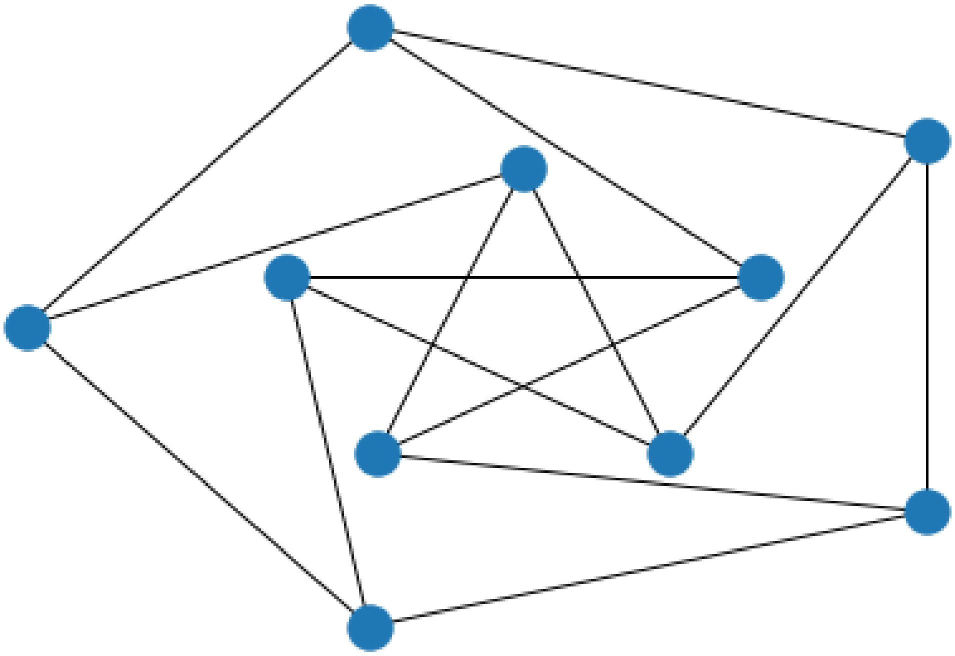
[208]:
```
nx.draw_circular(G)
```



The `draw_shell()` drawing takes two lists of nodes and the first list is placed inside while the second list is placed outside. There are multiple other ways of visualization, type nx.draw and press tab to view them. For example `draw_circular` places them on a circle

[207]:

```
nx.draw_shell(G, nlist=[[5,6,7,8,9],[0,1,2,3,4]])
```

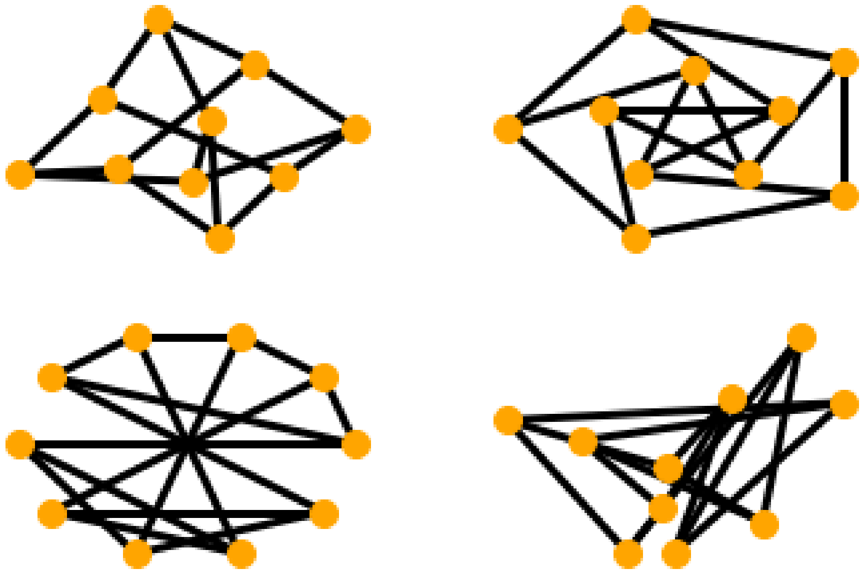


To show multiple plots in one row we use matplotlib pyplot `subplot`. The first two arguments are number of rows and columns and the next one is the index

[210]:

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

plt.subplot(2,2,1)
nx.draw(G, **options)

plt.subplot(2,2,2)
nx.draw_shell(G, nlist=[[5,6,7,8,9],[0,1,2,3,4]], **options)
```

```
plt.subplot(2,2,3)
nx.draw_circular(G, **options)

plt.subplot(2,2,4)
nx.draw_random(G, **options)

plt.show()
```



Coloring the graph Suppose numbers represent colors (we can use the hex code and stuff but for simplicity let us start with numbers).

The function `random_coloring_nodes` assigns a random integer to each node representing the color.

[211]:
```
def random_coloring_nodes(graph, n_colors):
    coloring = {}
    for node in graph.nodes():
        coloring[node] = np.random.randint(0, n_colors)
    return coloring
```

[212]:
```
def random_coloring_edges(graph, n_colors):
    coloring = {}
    for edge in graph.edges():
        coloring[edge] = np.random.randint(0, n_colors)
    return coloring
```

[213]:
```
some_node_coloring = random_coloring_nodes(G, 5)
some_node_coloring
```

[213]:
```
{0: 3, 1: 4, 2: 2, 3: 1, 4: 3, 5: 3, 6: 4, 7: 2, 8: 4, 9: 3}
```

[215]:
```
some_edge_coloring = random_coloring_edges(G, 6)
some_edge_coloring
```

[215]:
```
{(0, 1): 3,
 (0, 4): 3,
 (0, 5): 2,
 (1, 2): 4,
 (1, 6): 5,
 (2, 3): 1,
 (2, 7): 0,
 (3, 4): 5,
 (3, 8): 1,
 (4, 9): 1,
 (5, 7): 3,
 (5, 8): 1,
 (6, 8): 5,
 (6, 9): 5,
 (7, 9): 5}
```

Before visualizing the graph we must assign the colors to the numbers. There are many ways of doing this. For example we could take a list `color_list = ['red', 'blue', 'green', 'yellow', 'purple']` and use the five colors for the five numbers. But what if we want to keep the number of colors as a variable

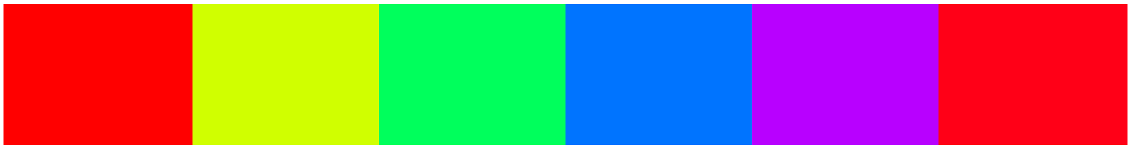
[217]:
```
def get_cmap(n, name='Spectral'):
    '''Returns a function that maps each index in 0, 1, ..., n-1 to a distinct
    RGB color; the keyword argument name must be a standard mpl colormap name.
 ↪'''
    return plt.cm.get_cmap(name, n)
```

[220]:
```
cmap = get_cmap(5)
cmap
```

[220]:

[221]:
```
get_cmap(6)
```

[221]:

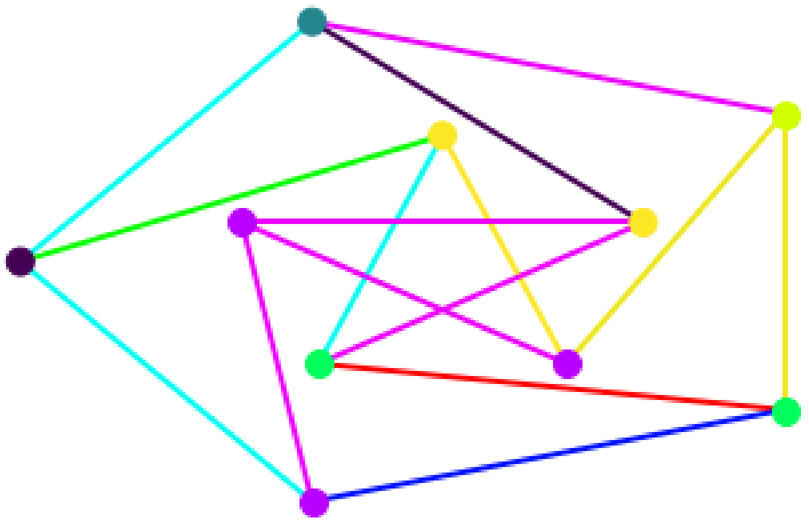


Let us now draw the coloring.

```
[224]: def draw_coloring(G,node_coloring, edge_coloring, pos="0"):
           if pos == "0":
                         pos = nx.random_layout(G)
           fig = plt.figure()
           n_node_colors = max(node_coloring[i] for i in node_coloring)+1
           n_edge_colors = max(edge_coloring[i] for i in edge_coloring)+1

           cmap_node = get_cmap(n_node_colors+1)
           cmap_edge = get_cmap(n_edge_colors+1)


           for i in range(n_node_colors):
               nx.draw_networkx_nodes(G, pos, [x for x in G.nodes() if␣
        ↪node_coloring[x]==i],node_color=cmap_node(i), node_size = 100)

           for i in range(n_edge_colors):
               nx.draw_networkx_edges(G, pos, [x for x in G.edges() if␣
        ↪edge_coloring[x]==i],edge_color=cmap_edge(i), width=2)


           plt.axis('off')
           plt.show()
           return fig
```

```
[225]: fig2 = draw_coloring(G,some_node_coloring, some_edge_coloring, nx.
        ↪shell_layout(G, nlist=[[5,6,7,8,9],[0,1,2,3,4]]))
```

```
*c* argument looks like a single numeric RGB or RGBA sequence, which should be
avoided as value-mapping will have precedence in case its length matches with
*x* & *y*.  Please use the *color* keyword-argument or provide a 2D array with a
single row if you intend to specify the same RGB or RGBA value for all points.
*c* argument looks like a single numeric RGB or RGBA sequence, which should be
avoided as value-mapping will have precedence in case its length matches with
*x* & *y*.  Please use the *color* keyword-argument or provide a 2D array with a
single row if you intend to specify the same RGB or RGBA value for all points.
*c* argument looks like a single numeric RGB or RGBA sequence, which should be
avoided as value-mapping will have precedence in case its length matches with
*x* & *y*.  Please use the *color* keyword-argument or provide a 2D array with a
single row if you intend to specify the same RGB or RGBA value for all points.
```



1.3.3 Homework:

Write a linear program to solve the coloring problem. Use networkx to visualize the graph.

[ ]: