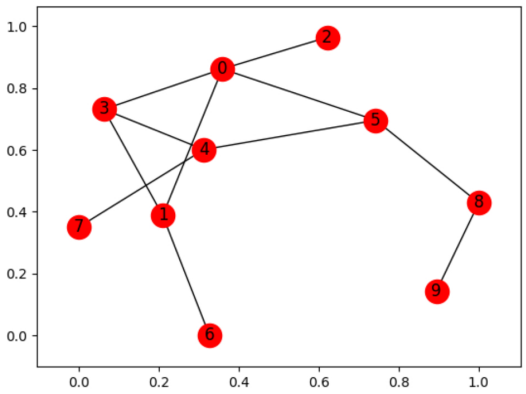
Loading Graphs in NetworkX

In [1]:
```
import networkx as nx
import numpy as np
import pandas as pd
%matplotlib notebook

# Instantiate the graph
G1 = nx.Graph()
# add node/edge pairs
G1.add_edges_from([(0, 1),
                   (0, 2),
                   (0, 3),
                   (0, 5),
                   (1, 3),
                   (1, 6),
                   (3, 4),
                   (4, 5),
                   (4, 7),
                   (5, 8),
                   (8, 9)])

# draw the network G1
nx.draw_networkx(G1)
```

```
<IPython.core.display.Javascript object>
```



Adjacency List

`G_adjlist.txt` is the adjaceny list representation of G1.

It can be read as follows:

- `0 1 2 3 5` → node `0` is adjacent to nodes `1, 2, 3, 5`
- `1 3 6` → node `1` is (also) adjacent to nodes `3, 6`
- `2` → node `2` is (also) adjacent to no new nodes
- `3 4` → node `3` is (also) adjacent to node `4`

and so on. Note that adjacencies are only accounted for once (e.g. node `2` is adjacent to node `0`, but node `0` is not listed in node `2`'s row, because that edge has already been accounted for in node `0`'s row).

In [2]:
```
!cat G_adjlist.txt
```

```
0 1 2 3 5
1 3 6
2
3 4
4 5 7
5 8
6
7
8 9
9
```

If we read in the adjacency list using `nx.read_adjlist`, we can see that it matches `G1`.

In [3]:
```
G2 = nx.read_adjlist('G_adjlist.txt', nodetype=int)
G2.edges()
```

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

Adjacency Matrix

The elements in an adjacency matrix indicate whether pairs of vertices are adjacent or not in the graph. Each node has a corresponding row and column. For

example, row `0`, column `1` corresponds to the edge between node `0` and node `1`.

Reading across row `0`, there is a '`1`' in columns `1`, `2`, `3`, and `5`, which indicates that node `0` is adjacent to nodes 1, 2, 3, and 5

In [4]:
```
G_mat = np.array([[0, 1, 1, 1, 0, 1, 0, 0, 0, 0],
                  [1, 0, 0, 1, 0, 0, 1, 0, 0, 0],
                  [1, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                  [1, 1, 0, 0, 1, 0, 0, 0, 0, 0],
                  [0, 0, 0, 1, 0, 1, 0, 1, 0, 0],
                  [1, 0, 0, 0, 1, 0, 0, 0, 1, 0],
                  [0, 1, 0, 0, 0, 0, 0, 0, 0, 0],
                  [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
                  [0, 0, 0, 0, 0, 1, 0, 0, 0, 1],
                  [0, 0, 0, 0, 0, 0, 0, 0, 1, 0]])
G_mat
```

Out[4]:
```
array([[0, 1, 1, 1, 0, 1, 0, 0, 0, 0],
       [1, 0, 0, 1, 0, 0, 1, 0, 0, 0],
       [1, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [1, 1, 0, 0, 1, 0, 0, 0, 0, 0],
       [0, 0, 0, 1, 0, 1, 0, 1, 0, 0],
       [1, 0, 0, 0, 1, 0, 0, 0, 1, 0],
       [0, 1, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 1, 0, 0, 0, 1],
       [0, 0, 0, 0, 0, 0, 0, 0, 1, 0]])
```

If we convert the adjacency matrix to a networkx graph using `nx.Graph`, we can see that it matches G1.

In [5]:
```
G3 = nx.Graph(G_mat)
G3.edges()
```

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

Edgelist

The edge list format represents edge pairings in the first two columns. Additional edge attributes can be added in subsequent columns. Looking at `G_edgelist.txt` this is the same as the original graph `G1`, but now each edge has a weight.

For example, from the first row, we can see the edge between nodes `0` and `1`, has a weight of `4`.

In [6]:
```
!cat G_edgelist.txt
```

```
0 1 4
0 2 3
0 3 2
0 5 6
1 3 2
1 6 5
3 4 3
4 5 1
4 7 2
5 8 6
8 9 1
```

Using `read_edgelist` and passing in a list of tuples with the name and type of each edge attribute will create a graph with our desired edge attributes.

In [7]:
```
G4 = nx.read_edgelist('G_edgelist.txt', data=[('Weight', int)])

G4.edges(data=True)
```

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

Pandas DataFrame

Graphs can also be created from pandas dataframes if they are in edge list format.

In [8]:
```
G_df = pd.read_csv('G_edgelist.txt', delim_whitespace=True,
                   header=None, names=['n1', 'n2', 'weight'])
G_df
```

Out[8]:

	n1	n2	weight
0	0	1	4
1	0	2	3
2	0	3	2
3	0	5	6
4	1	3	2
5	1	6	5
6	3	4	3
7	4	5	1
8	4	7	2
9	5	8	6
10	8	9	1

In [9]:
```
G5 = nx.from_pandas_dataframe(G_df, 'n1', 'n2', edge_attr='weight')
G5.edges(data=True)
```

Out[9]:
```
[(0, 1, {'weight': 4}),
 (0, 2, {'weight': 3}),
```

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

Chess Example

Now let's load in a more complex graph and perform some basic analysis on it.

We will be looking at chess_graph.txt, which is a directed graph of chess games in edge list format.

In [10]:
```
!head -5 chess_graph.txt
```

```
1 2 0   885635999.999997
1 3 0   885635999.999997
1 4 0   885635999.999997
1 5 1   885635999.999997
1 6 0   885635999.999997
```

Each node is a chess player, and each edge represents a game. The first column with an outgoing edge corresponds to the white player, the second column with an incoming edge corresponds to the black player.

The third column, the weight of the edge, corresponds to the outcome of the game. A weight of 1 indicates white won, a 0 indicates a draw, and a -1 indicates black won.

The fourth column corresponds to approximate timestamps of when the game was played.

We can read in the chess graph using `read_edgelist`, and tell it to create the graph using a `nx.MultiDiGraph`.

In [11]:
```
chess = nx.read_edgelist('chess_graph.txt', data=[('outcome', int), ('timestamp', float)],
                         create_using=nx.MultiDiGraph())
```

In [12]:
```
chess.is_directed(), chess.is_multigraph()
```

Out[12]:
```
(True, True)
```

In [13]:
```
chess.edges(data=True)
```

Out[13]:
```
[('1', '2', {'outcome': 0, 'timestamp': 885635999.999997}),
 ('1', '3', {'outcome': 0, 'timestamp': 885635999.999997}),
 ('1', '4', {'outcome': 0, 'timestamp': 885635999.999997}),
 ('1', '5', {'outcome': 1, 'timestamp': 885635999.999997}),
 ('1', '6', {'outcome': 0, 'timestamp': 885635999.999997}),
 ('1', '807', {'outcome': 0, 'timestamp': 896148000.000003}),
 ('1', '454', {'outcome': 0, 'timestamp': 896148000.000003}),
 ('1', '827', {'outcome': 0, 'timestamp': 901403999.999997}),
 ('1', '1240', {'outcome': 0, 'timestamp': 906660000.0}),
 ('1', '680', {'outcome': 0, 'timestamp': 906660000.0}),
 ('1', '166', {'outcome': -1, 'timestamp': 906660000.0}),
 ('1', '1241', {'outcome': 0, 'timestamp': 906660000.0}),
 ('1', '1242', {'outcome': 0, 'timestamp': 906660000.0}),
 ('1', '808', {'outcome': 0, 'timestamp': 925055999.999997}),
 ('1', '819', {'outcome': 0, 'timestamp': 925055999.999997}),
 ('1', '448', {'outcome': 0, 'timestamp': 927684000.000003}),
 ('1', '1214', {'outcome': 0, 'timestamp': 927684000.000003}),
 ('1', '1217', {'outcome': 0, 'timestamp': 927684000.000003}),
 ('1', '2454', {'outcome': 0, 'timestamp': 938196000.0}),
 ('1', '925', {'outcome': 1, 'timestamp': 1064340000.0}),
```

Looking at the degree of each node, we can see how many games each person played. A dictionary is returned where each key is the player, and each value is the number of games played.

In [14]:
```
games_played = chess.degree()
games_played
```

Out[14]:
```
{'1': 48,
 '2': 112,
 '3': 85,
 '4': 12,
 '5': 18,
 '6': 95,
 '7': 9,
 '8': 20,
 '9': 142,
 '10': 4,
 '11': 2,
 '12': 70,
 '13': 148,
 '14': 153,
 '15': 23,
 '16': 3,
 '17': 115,
 '18': 45,
 '19': 27,
 '20': 12,
```

Using list comprehension, we can find which player played the most games.

In [15]:
```
max_value = max(games_played.values())
max_key, = [i for i in games_played.keys() if games_played[i] == max_value]

print('player {}\n{} games'.format(max_key, max_value))
```

```
player 461
280 games
```

Let's use pandas to find out which players won the most games. First let's convert our graph to a DataFrame.

In [16]:
```
df = pd.DataFrame(chess.edges(data=True), columns=['white', 'black', 'outcome'])
df.head()
```

Out[16]:

	white	black	outcome
0	1	2	{'outcome': 0, 'timestamp': 885635999.999997}
1	1	3	{'outcome': 0, 'timestamp': 885635999.999997}
2	1	4	{'outcome': 0, 'timestamp': 885635999.999997}
3	1	5	{'outcome': 1, 'timestamp': 885635999.999997}
4	1	6	{'outcome': 0, 'timestamp': 885635999.999997}

Next we can use a lambda to pull out the outcome from the attributes dictionary.

In [17]:
```
df['outcome'] = df['outcome'].map(lambda x: x['outcome'])
df.head()
```

Out[17]:

	white	black	outcome
0	1	2	0
1	1	3	0
2	1	4	0
3	1	5	1
4	1	6	0

To count the number of times a player won as white, we find the rows where the outcome was '1', group by the white player, and sum.

To count the number of times a player won as back, we find the rows where the outcome was '-1', group by the black player, sum, and multiply by -1.

The we can add these together with a fill value of 0 for those players that only played as either black or white.

In [18]:
```
won_as_white = df[df['outcome']==1].groupby('white').sum()
won_as_black = -df[df['outcome']==-1].groupby('black').sum()
win_count = won_as_white.add(won_as_black, fill_value=0)
win_count.head()
```

Out[18]:

	outcome
1	7.0
100	7.0
1000	1.0
1002	1.0
1003	5.0

Using `nlargest` we find that player 330 won the most games at 109.

In [19]:
```
win_count.nlargest(5, 'outcome')
```

Out[19]:

	outcome
330	109.0
467	103.0
98	94.0
456	88.0
461	88.0