1. Social circles: Facebook

This dataset consists of 'circles' (or 'friend's lists') from Facebook. Facebook data was collected from survey participants using this Facebook app. The dataset includes node features (profiles), circles, and ego networks.

This database contains the following information

```
1. print(nx.info(G))

1. [Out]: Name:
2.         Type: Graph
3.         Number of nodes: 4039
4.         Number of edges: 88234
5.         Average degree:  43.6910
```

2. Degree Centrality

Degree centrality is a simple count of the total number of connections linked to a vertex. It can be thought of as a kind of popularity measure, but a crude one that does not recognize a difference between quantity and quality.

Equation presents how degree centrality is calculated. Although it might seem a simple task to just add up the number of connections of each node, that is essentially what this equation is doing!

```
1. pos = nx.spring_layout(G1)
2. degCent = nx.degree_centrality(G)
3. node_color = [20000.0 * G.degree(v) for v in G]
4. node_size =  [v * 10000 for v in degCent.values()]
5. plt.figure(figsize=(15,15))
6. nx.draw_networkx(G, pos=pos, with_labels=False,
7.                  node_color=node_color,
8.                  node_size=node_size )
9. plt.axis('off')
10.
```

3. Betweenness Centrality

Betweenness centrality is a measure of centrality in a graph based on shortest paths.

To **calculate Betweenness centrality**, you take every pair of the **network** and count how many times a node can interrupt the shortest paths (geodesic distance) between the two nodes of the pair.

Where $\sigma uw(n_i)$ is the number of those paths that pass through n_i and σuw is the total number of shortest paths from node u to node w.

```
pos = nx.spring_layout(G)
betCent = nx.betweenness_centrality(G, normalized=True, endpoints=True)
node_color = [20000.0 * G.degree(v) for v in G]
node_size =  [v * 10000 for v in betCent.values()]
plt.figure(figsize=(20,20))
nx.draw_networkx(G, pos=pos, with_labels=False,
                 node_color=node_color,
                 node_size=node_size )
plt.axis('off')
```

4. Closeness Centrality

Closeness centrality is a measure of the average shortest **distance** from each vertex to each other vertex. Specifically, it is the inverse of the average shortest **distance** between the vertex and all other vertices in the **network**.

The **formula** is

1 / (average **distance** to all other vertices)

```
pos = nx.spring_layout(G)
cloCent = nx.closeness_centrality(G)
node_color = [20000.0 * G1.degree(v) for v in G]
node_size =  [v * 10000 for v in cloCent.values()]
plt.figure(figsize=(13,13))
nx.draw_networkx(G, pos=pos, with_labels=False,
                 node_color=node_color,
                 node_size=node_size )
plt.axis('off')
```

5. Eigenvector Centrality

Eigenvector centrality is a **centrality** index that calculates the **centrality** of an actor based not only on their connections, but also based on the **centrality** of that actor's connections.

```
pos = nx.spring_layout(G)
eigCent = nx.eigenvector_centrality(G)
node_color = [20000.0 * G.degree(v) for v in G]
node_size =  [v * 10000 for v in eigCent.values()]
plt.figure(figsize=(15,15))
nx.draw_networkx(G, pos=pos, with_labels=False,
                 node_color=node_color,
                 node_size=node_size )
plt.axis('off')
```

6. Find the shortest path between nodes along with their length

```
sources = [20,40,65,75]
targets = [650,802,920,1010]
for i in range(4):
    path = nx.shortest_path(G1,source=sources[i],target=targets[i])
    length = nx.shortest_path_length(G1,source=sources[i],target=targets[i],method='dijkstra')
    print("Shortest Path between Node ", str(sources[i])," ---> ", str(targets[i]), " is ",
          str(path), " ,Length = ", str(length))
```

7. Find the shortest path between nodes along with their length

```
neigh = [1,20,40,65,75,90,1000,]
for i in range(len(neigh)):
    all_neighbors = list(nx.classes.function.all_neighbors(G1,1))
    print("All neighbors for Node ", str(neigh[i])," ---> ", str(all_neighbors))
```

8. Find the degrees of the nodes along with the number of degrees in the graph

```
1. from collections import Counter
2. deg = dict(G1.degree()).values()
3. Counter(deg )
```