https://github.com/cajunwa/phys7332_fa25

Solutions to Assignment 3
Network Science
PHYS 7332, Fall 2025

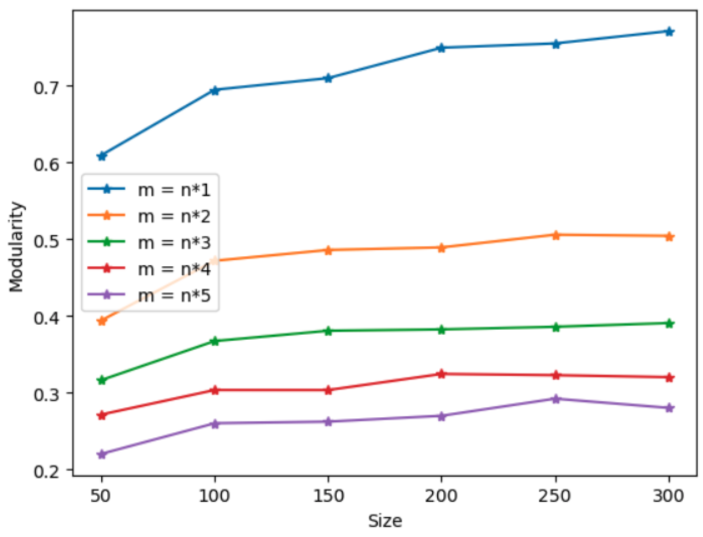
Chelsea Ajunwa

1. *Question 1.*

(a)

```
#Replicating Figure 2 from Guimera, Sales-Pardo, and Amaral, 2004
fig, ax = plt.subplots(1,1)


#setting network sizes
sizes = [x for x in range(50,301,50)]

comm_num_dict = {size:None for size in sizes} #not needed
modularity_dict = {size:None for size in sizes}

for n in sizes:
    m_s = [n*x for x in range(1,6)]
    num_communities_n = []
    modularity_n = []
    for m in m_s:
        G = nx.gnm_random_graph(n,m)
        partition = community.best_partition(G)
        num_communities = len(set(partition.values()))
        num_communities_n.append(num_communities)
        modularity = community.modularity(partition,G)
        modularity_n.append(modularity)

    comm_num_dict[n]=num_communities_n
    modularity_dict[n]=modularity_n

Q_df = pd.DataFrame(modularity_dict).T

m = ['n','n*2','n*3','n*4','n*5']

for line_num in range(5):
    ax.plot(Q_df[line_num],marker='*',label=f"m = n*{line_num+1}")
    line_num +=1
```

```
ax.set_ylabel("Modularity")
ax.set_xlabel("Size")
ax.set_xticks(sizes)
ax.legend()
```



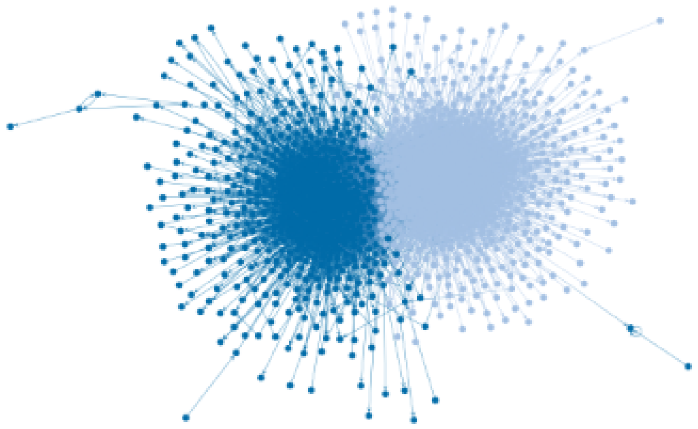
As described in Guimera, Sales-Pardo, and Amaral, 2004, at each m value, modularity rises as a function of size and then plateaus. Modularity also increases as the number of edges relative to nodes decreases.

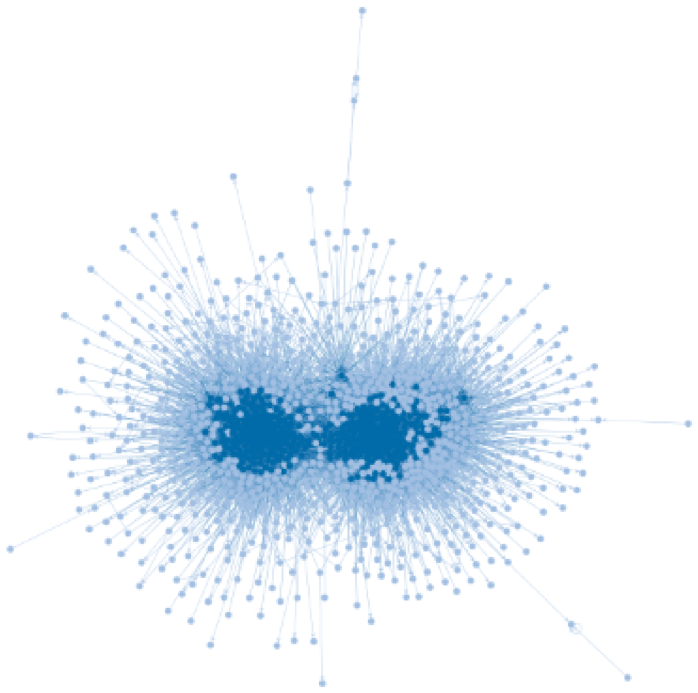
The less dense graphs were considered less modular despite the fact that they were constructed in the same random manner as the more dense graphs. This suggests that modularity, calculated from modularity maximization-determined network partitions, may not fully reflect the underlying structure of graphs.

(b)

```
fig ,ax = plt.subplots(1,2,figsize=(8,6),constrained_layout=True)

#loading data
G_0=gt.collection.ns["polblogs"]

#getting largest component
G = topology.extract_largest_component(G_0,directed=False)

#fitting blockmodel
state_dc =
    gt.minimize_blockmodel_dl(G,state_args=dict(deg_corr=True),multilevel_mcmc_args={

state_no_dc =
    gt.minimize_blockmodel_dl(G,state_args=dict(deg_corr=False),multilevel_mcmc_args=

#visualizing
state_dc.draw(ax=ax[0])
state_no_dc.draw(ax=ax[1])
```

```
ax[0].set_title('Degree-corrected SBM: Polblogs largest component')
ax[1].set_title('Non-degree-corrected SBM: Polblogs largest component')
ax[0].set_axis_off()
ax[1].set_axis_off()


plt.show()
```



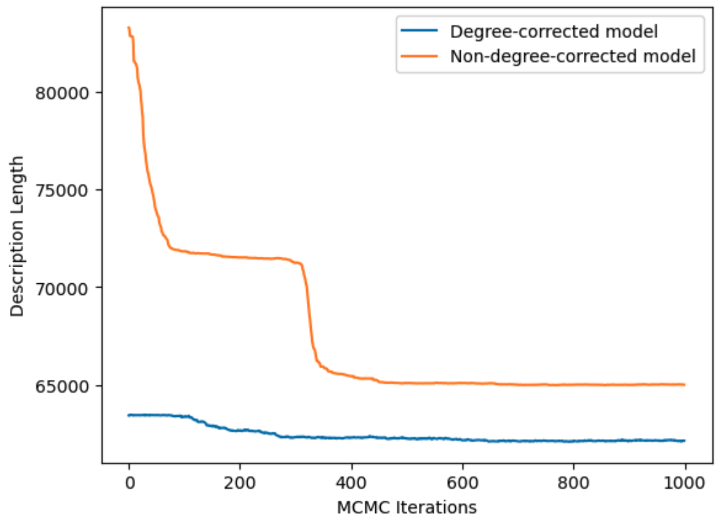


(c)

```
#get state entropies
print(f"Description length of degree-corrected model:
    {state_dc.entropy()}")
```

```
print(f"Description length of non-degree-corrected model:
    {state_no_dc.entropy()}")


#get description lengths for 1000 iterations of mcmc algorithm
dc_dls = []

state_dc =
    gt.minimize_blockmodel_dl(G,state_args=dict(deg_corr=True),multilevel_mcmc_args={

for i in range(1000):
    state_dc.mcmc_sweep(niter=1)
    dl = state_dc.entropy()
    dc_dls.append(dl)



no_dc_dls = []

state_no_dc =
    gt.minimize_blockmodel_dl(G,state_args=dict(deg_corr=False),multilevel_mcmc_args=

for i in range(1000):
    state_no_dc.mcmc_sweep(niter=1)
    dl = state_no_dc.entropy()
    no_dc_dls.append(dl)


#plot

fig,ax = plt.subplots(1,1)

ax.plot(dc_dls,label="Degree-corrected model")
ax.plot(no_dc_dls,label="Non-degree-corrected model")
ax.legend()
ax.set_ylabel("Description Length")
ax.set_xlabel("MCMC Iterations")
```

The description length of the degree corrected model was about 61739. The description length of the non-degree-corrected model was about 64509.



(d) The SBM ended up dividing the network by degree. This is because it assumes that nodes belonging to the same group are statistically equivalent and receive the same number of edges. Degree correction addresses this issue.

One proposed method of fixing this issue was adding additional parameters to the model to account for each node's expected degree and the expected number of edges between groups. A potentially better way to fix this is to use Bayesian inference and include a prior for the probability of generating a given degree sequence.

2. *Question 2.*

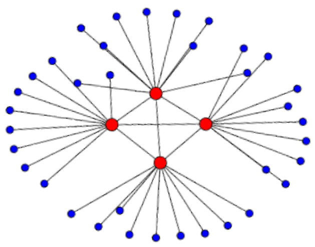
(a) "Rich club" network organization is a phenomenon in which a network has a few nodes with high centrality (hubs) that are highly connected to each other. One can use the rich club coefficient to quantify this.

(b) The most confusing part of this concept is calculating the rich club coefficient. The formula requires first defining a threshold k, identifying nodes above that threshold, and determining the number of edges connecting those nodes. You then you divide that number by the maximum possible number of edges between the identified high degree nodes. The resulting coefficient indicates to what extent a network features rich-club organization.

(c) Flashcard:

Rich Club Organization

“Rich club” network organization is a phenomenon in which a network has a few nodes with high centrality (hubs) that are disproportionality highly-connected to each other.



Rich club organization can be assessed using the rich club coefficient.

$$\varphi(k) = \frac{2E_{\mathrm{k}}}{N_k(N_k - 1)}$$

1. Define a degree threshold k.
2. Identify nodes (N_k) that are above that threshold.
3. Determine the number of edges (E_{k}) connecting those nodes.
4. Determine the maximum possible number of edges between those N_k high-degree nodes. This is $N_k(N_k - 1)$.
5. Divide $2E_{\mathrm{k}}$ by that number.
6. (Optional) Compare your $\varphi(k)$ to those of degree-preserving randomized null models to validate.

3. *Question 3.*

(a)

```python
#function that computes commute time of a random walker

def commute_time(G,start,end):
    """
    Computed the commute time of a single random walker between nodes i
        and j in a network.

    Parameters:
    ---------------
    G: networkx.Graph
        Network
    start: node
        Node i in network
    end: node
```

```
        Node j in network

    Returns:
    -------------
    commute_time (int):
        Time it takes random walker to get from node i to node j

    Example:
    -------------
    >>> commute_time(G, 4, 90)
    35

    """

    #start path list
    path = [start]

    current_node = start

    while current_node != end:
        neighbors = list(G.neighbors(current_node))
        if neighbors:
            next_node = np.random.choice(neighbors)
            path.append(next_node)
            current_node=next_node
        else:
            break

    commute_time = len(path)

    return commute_time
```

```
        G = nx.karate_club_graph()
nodes = list(G.nodes())

for i in range(3):
    start = np.random.choice(nodes)
    end = np.random.choice(nodes)

    ct = commute_time(G,start,end)
    print(f"Commute time between node {start} and node {end} is {ct}.")
```

Commute time between node 19 and node 17 was 334.

Commute time between node 0 and node 1 was 146.

Commute time between node 10 and node 4 was 34.

(b)

```
        #Writing a function to calculate mean commute time within
            networks

def mean_commute(G):
    """

    Parameters:
    ---------------
    G: networkx.Graph
        Network

    Returns:
    --------------
    mean_commute (float):
        Time it takes random walker to get from node i to node j

    Example:
    --------------
    >>> mean_commute(G)
    34.2


    """

    nodes = list(G.nodes())

    #get all pairs of nodes

    node_pairs = list(itertools.combinations(nodes,2))

    commutes = []

    #get all commute times

    for pair in node_pairs:
        start = pair[0]
        end = pair[1]
        commute = commute_time(G,start,end)
        commutes.append(commute)

    #get mean
    mean_commute = np.mean(commutes)
    return mean_commute
```

Note: I had to decrease N to 100 for these graphs because my computer was having difficulty completing the calculation on the 500 node graphs.

```
        #Calculating mean commute time for Erdos-Renyi network with N =
            500 and k = 8
N = 100
k_avg = 8
p = 8/(N-1)
G_ER = nx.erdos_renyi_graph(N,p)

mean_commute_ER = mean_commute(G_ER)

print(f"Mean commute time for an Erdos-Renyi network with N = {N} and
    k = {k_avg} is {mean_commute_ER}")

#Get average across 10 samples

commutes = []
for i in range(10):
    G_ER = nx.erdos_renyi_graph(N,p)
    mean_commute_ER = mean_commute(G_ER)
    commutes.append(mean_commute_ER)
commutes10 = np.mean(commutes)

print(f"Mean commute time across 10 samples: {commutes10}")
```

Mean commute time for an Erdos-Renyi network with N = 100 and $\langle k \rangle = 8$ was 130.68161616161615
Mean commute time across 10 samples: 132.9420202020202

```
        #Calculating mean commute time for Barabasi-Albert networks
            with N = 100 and m = 4
N = 100
m = 4

G_BA = nx.barabasi_albert_graph(N,m)

mean_commute_BA = mean_commute(G_BA)

print(f"Mean commute time for a Barabasi-Albert network with N = {N}
    and m = {m} is {mean_commute_BA}")

commutes = []
for i in range(10):
    G_BA = nx.barabasi_albert_graph(N,m)
    mean_commute_BA = mean_commute(G_BA)
    commutes.append(mean_commute_BA)
commutes10 = np.mean(commutes)

print(f"Mean commute time across 10 samples: {commutes10}")
```

```
Mean commute time for a Barabasi-Albert network with N = 100 and m = 4 was
177.61111111111111
Mean commute time across 10 samples: 180.34484848484848
```

The commute times were greater for the Barabasi-Albert networks. This may be due to the differing graph generation procedure and resulting structure (including the degree distribution, which is a power law).

4. *Question 4.*
   Random Walk Network Dynamics and Spectral Properties of Graphs

   (a) Generate a Barabasi-Albert graph with 100 nodes and m = 4. Make a plot visualizing a random walk of 50 steps through this graph, with nodes labeled in order of appearance in walk.

   (b) Generate matrices, plot, and compare.
      i. Generate the adjacency matrix associated with this graph.
      ii. Calculate the degree vector by summing across rows of the matrix. Generate a degree matrix in which the graph's degrees are the diagonal of the matrix.
      iii. Calculate the transition matrix.
      iv. Calculate the normalized adjacency matrix.
      v. Create a plot with the adjacency matrix, degree matrix, transition matrix, and normalized adjacency matrix as subplots.
      vi. Provide any observations. Conceptually, how might the transition matrix be related to the normalized adjacency matrix?

   (c) Generate a symmetric normalized Laplacian matrix from and get eigenvalues/eigenvectors. Visualize matrix.

   (d) Visualize your graph twice, first with node colors corresponding to eigenvectors from the normalized adjacency matrix, and second, with node colors corresponding to eigenvectors from the symmetric normalized Laplacian matrix. Include a legend. What do you notice about the eigenvectors from the adjacency matrix versus those from the Laplacian? How are they related, mathematically?

   (e) There is a mathematical relationship between the random walk network dynamics explored in part A (as depicted in the transition matrix) and both the normalized adjacency matrix and the Laplacian. What information can we glean from the normalized adjacency matrix regarding random walk network dynamics? How about the Laplacian? What does it tell us about random walk network dynamics?