A First Course on Network Science: Selected Exercise Solutions

Filippo Menczer, Santo Fortunato, Clayton A. Davis

November 2, 2019

Chapter 1

Exercise 1.13. $L_{max} = N_1 N_2 = N_1(N - N_1)$ links. The value of N_1 that maximizes the number of links is obtained by setting the derivative of L_{max} to zero: $\frac{d}{dN_1} L_{max} = N - 2N_1 = 0 \implies N_1 = N/2$, so that $L_{max} = N^2/4$.

Exercise 1.24. With a doubling of L alone, density would increase (double): $d' = \frac{2L}{\binom{N}{2}} = 2d$. With a doubling of N, the maximum number of links would grow roughly quadratically, so a doubling of L would not suffice to keep the density constant; it would decrease. Mathematically, you can prove that $d' = \frac{2L}{\binom{2N}{2}} < \frac{L}{\binom{N}{2}} = d$.

Exercise 1.25. Let us use the subscripts u and m for users and movies, respectively. $L = N_u \langle k_u \rangle = 33 \times 10^9$. $d = \frac{L}{L_{max}} = \frac{N_u \langle k_u \rangle}{N_u N_m} = \frac{\langle k_u \rangle}{N_m} \approx 10^{-2}$. So the network is fairly sparse.

Chapter 2

Exercise 2.24. $\ell_{max} = 2$

Exercise 2.35. $\ell = 1, 2, 2, 3, 3, 4, 4, 1, 1, 2, 2, 3, 3, 1, 2, 2, 3, 3, 1, 1, 2, 2, 2, 3, 3, 1, 1, 2$

Exercise 2.36. The four leaves have undefined clustering coefficient. For the other nodes, $C = 1/3, 1, 1/6, 0$.

Exercise 2.37. $\ell_{a,b} = 2, \ell_{a,c} = 2, \ell_{a,d} = 4, \ell_{a,e} = 3, \ell_{a,f} = 1, \ell_{a,g} = 1, \ell_{a,h} = 3, \ell_{b,c} = 1, \ell_{b,d} = 2, \ell_{b,e} = 1, \ell_{b,f} = 2, \ell_{b,g} = 1, \ell_{b,h} = 1, \ell_{c,d} = 2, \ell_{c,e} = 1, \ell_{c,f} = 2, \ell_{c,g} = 1, \ell_{c,h} = 2, \ell_{d,e} = 1, \ell_{d,f} = 4, \ell_{d,g} = 3, \ell_{d,h} = 3, \ell_{e,f} = 3, \ell_{e,g} = 2, \ell_{e,h} = 2, \ell_{f,g} = 1, \ell_{f,h} = 3, \ell_{g,h} = 2$. $\langle \ell \rangle \approx 2.0$. This would have been quicker to do in code than by hand:

```
FIG12 = nx.Graph()
FIG12.add_edges_from([('d','e'),('e','c'),('e','b'),('c','b'),
                      ('c','g'),('b','g'),('b','h'),('g','f'),
                      ('g','a'),('f','a')])
p = dict(nx.shortest_path_length(FIG12))
for s in p:
  for t in p[s]:
    print('length(%s,%s) = %d' % (s, t, p[s][t]))
print('APL = ', nx.average_shortest_path_length(FIG12))
```

Exercise 2.38. $C(a) = 1, C(b) = 1/3, C(c) = 2/3, C(e) = 1/3, C(f) = 1, C(g) = 1/3$. $C \approx 0.6$. In code (building upon the previous exercise):

```
for n,c in nx.clustering(FIG12).items():
  print('C(%s) = %f' % (n,c))
print('C =',nx.average_clustering(FIG12))
```

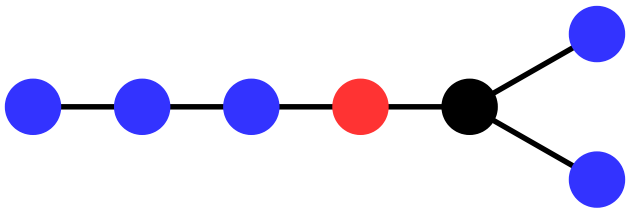
Note that NetworkX includes nodes with degree below 2 and assigns clustering coefficient zero to them, instead of excluding them as undefined. As a result, the network clustering coefficient calculated by NetworkX is lower.

Chapter 3

Exercise 3.4. We use the max function with degree as key:

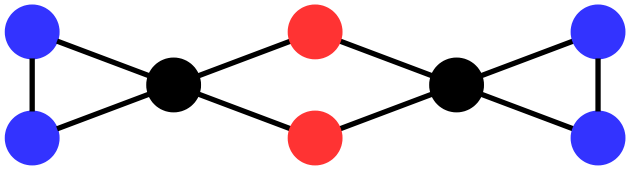
```
# 1 Finding node with largest degree for graph G
highest_degree_node = max(G.nodes, key=G.degree)
highest_degree_node

# 2 Printing the value of the maximum degree
G.degree(highest_degree_node)
```

Exercise 3.10. 1. The node with the largest closeness is the red node (degree 2, closeness 1/11), not the black node (degree 3, closeness 1/12).



2. The nodes with the largest betweenness are the black ones (betweenness 10 each, as there are 10 pairs of nodes whose shortest paths must cross the black nodes; their closeness is 1/12). The ones with the largest closeness are the red ones (betweenness 4.5, as there are 9 pairs of nodes whose shortest paths cross a red node, but each of them counts 1/2 because

they can go through the other red node as well; their closeness is 1/11).



Exercise 3.19. Using NetworkX:

```
def avg_degree_neighbors(G, friend):
    if G.degree(friend) > 0:
        k_nn = 0
        for node in G.neighbors(friend):
            k_nn += G.degree(node)/G.degree(friend)
        return k_nn
    else:
        print('The k_nn of', friend, 'is undefined because',
              friend, 'has no neighbors!')
```

The average degree of the nearest neighbors of the nodes is 64.05, much larger than the average node degree 10.19 — a result consistent with the Friendship Paradox.

Exercise 3.23. The node with lower clustering coefficient. A high clustering coefficient means that many neighbors of the node are connected to each other, so paths going through the node can be made shorter by going directly from the neighbor of the node preceding it in the path to the neighbor following it in the path. Therefore not many shortest paths are expected to go through a node with high clustering coefficient (see also the solution of Exercise 3.22 above).

Chapter 4

Exercise 4.4. We can do this by hand or with code. The PageRank update formulas for the three pages with $\alpha = 0$ are: $A(t+1) = B(t) + C(t)/2$, $B(t+1) = A(t)/2 + C(t)/2$, $C(t+1) = A(t)/2$. In the table below we calculate the values rounding to the second decimal digit. They converge to the second decimal digit after 8 steps.

step	A	B	C
0	0.33	0.33	0.33
1	0.50	0.33	0.17
2	0.42	0.33	0.25
3	0.46	0.33	0.21
4	0.44	0.33	0.23
5	0.45	0.33	0.22
6	0.44	0.33	0.22
7	0.45	0.33	0.22
8	0.44	0.33	0.22
9	0.44	0.33	0.22

Exercise 4.9. Let's use Twython as in the book's tutorials:

```
# 1
from twython import Twython
API_KEY             = '...'
API_SECRET_KEY      = '...'
ACCESS_TOKEN        = '...'
ACCESS_TOKEN_SECRET = '...'
twitter = Twython(API_KEY, API_SECRET_KEY,
                  ACCESS_TOKEN, ACCESS_TOKEN_SECRET)
user = twitter.show_user(screen_name='clayadavis')
user['friends_count']
# 179 --- so answer is b.

# 2
import itertools
user = twitter.show_user(screen_name='truthyatindiana')
NUM_TWEETS_TO_FETCH = user['friends_count']
cursor = twitter.cursor(twitter.get_friends_list,
                        screen_name='truthyatindiana', count=200)
friends = list(itertools.islice(cursor, NUM_TWEETS_TO_FETCH))
from statistics import mean, median
followers = {}
for friend in friends:
    followers[friend['screen_name']] = friend['followers_count']
print('I have', user['followers_count'], 'followers, while my')
print('friends have', round(mean(followers.values())), 'on average')
# I have 3295 followers, while my
# friends have 224245 on average

# 3
print('I have', user['followers_count'], 'followers, while')
print('most of my friends have', median(followers.values()))
# I have 3295 followers, while
# most of my friends have 3598
```

```
# 4
most_popular = max(followers, key=followers.get)
print('Among my friends,', '@'+most_popular,
      'has most followers:', followers[most_popular])
# Among my friends, @WhiteHouse has most followers: 18891275
```

Chapter 5

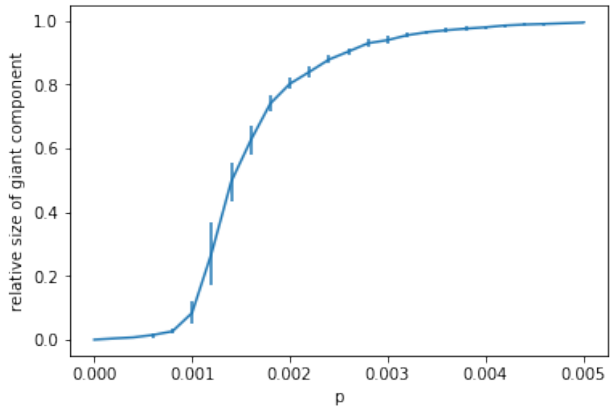
Exercise 5.6. We can check using this function:

```
def APL(N, avk=10):
    p = avk / (N-1)
    G = nx.gnp_random_graph(N, p)
    return nx.average_shortest_path_length(G)
```

We get $\langle \ell \rangle \approx 2.9$ for $N = 500$, so the answer is (d). Another approach is to use Equation (5.5), which gives a rough estimate of the diameter as well as the average path length of the network: $\langle \ell \rangle \approx \log N / \log \langle k \rangle$. For $\langle \ell \rangle = 3$ and $\langle k \rangle = 10$ we get: $\log N \approx \langle \ell \rangle \log \langle k \rangle = 3 \log 10 = \log 1000$, so $N \approx 1000$. The closest answer is (d).

Exercise 5.10. Here is Python code using NetworkX:

```
from statistics import mean, stdev
averages = []
errors = []
ps = [i/5000 for i in range(26)]
for p in ps:
    GC_ratios = []
    for _ in range(20):
        G = nx.gnp_random_graph(1000, p)
        GC_ratios.append(len(max(nx.connected_components(G),
                                 key=len))/1000)
    averages.append(mean(GC_ratios))
    errors.append(stdev(GC_ratios))
plt.errorbar(p, averages, yerr=errors)
plt.xlabel('p')
plt.ylabel('relative size of giant component')
```

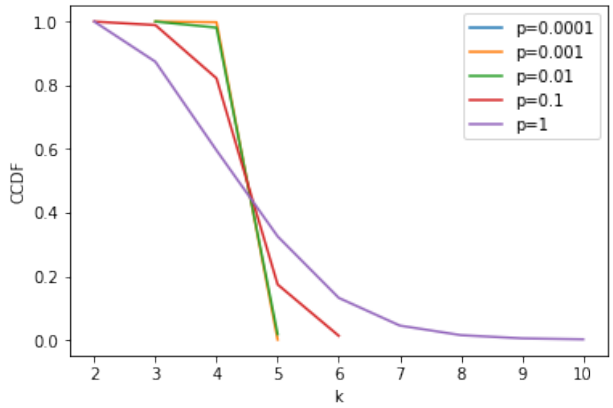


Exercise 5.15. We start with a function that takes a degree sequence and returns a dictionary with the cumulative distribution (CCDF):

```
import collections
def CCDF(degrees):
    counter = collections.Counter(degrees)
    sorted_degrees = sorted(list(counter))
    N = len(degrees)
    remaining = N
    cumcount = {}
    for d in sorted_degrees:
        cumcount[d] = remaining/N
        remaining -= counter[d]
    return cumcount
```

Use NetworkX to generate the networks and then plot the distributions:

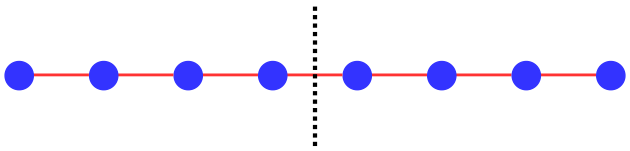
```
ps = [0.0001, 0.001, 0.01, 0.1, 1]
for p in ps:
    G = nx.watts_strogatz_graph(1000, 4, p)
    ccdf = CCDF([k for _,k in G.degree()])
    plt.plot(ccdf.keys(), ccdf.values(), label='p='+str(p))

plt.xlabel('k')
plt.ylabel('CCDF')
plt.legend()
```



Exercise 5.21. If $\eta_j = k_j$ for every node j, the link probability for the fitness model (Equation 5.12) becomes $k_j^2/(\sum_l k_l^2)$, so it takes the expression of the link probability of the non-linear preferential attachment (Equation 5.10) with $\alpha = 2$. Therefore, we expect that one of the nodes will end up being connected to a fraction of all other nodes while all other nodes will have approximately the same (low) degree.

Chapter 6

Exercise 6.6. The problems are not equivalent because even if one minimizes the separation between the clusters, the latter are not guaranteed to be communities because the density of internal links might not be high. A simple example is illustrated in the figure below. The dotted vertical line shows the best solution of the bisection problem for a chain-like network with eight nodes. The two clusters separated by the cut, however, cannot be considered communities, as they have the lowest possible density of internal links (they are both trees).



Exercise 6.10. In a bipartite network with N_A and N_B nodes in classes A and B, respectively, there are no links within the clusters corresponding to the classes. So, in modularity's equation 6.4 L_C is zero for $C = A$ and $C = B$ and the degree k_C of both clusters is the number of links L of the network, as the degree of cluster A (B) concides with the external degree, i.e. with the number of links joining A (B) with B (A). Therefore, the modularity for the bipartition $\{A, B\}$ is

$$Q_{A,B} = -\frac{L^2}{4L^2} - -\frac{L^2}{4L^2} = -\frac{1}{2},$$

independently of the values of N_A, N_B and L.

Exercise 6.20. Here are two functions to sample the network. For snowball sampling we use the NetworkX function that returns edges in breadth-first search order:

```
def random_sample(G, n_nodes):
    sampled_nodes = random.sample(G.nodes, n_nodes)
    return nx.subgraph(G, sampled_nodes)

def snowball_sample(G, n_nodes):
    root = random.choice(list(G.nodes()))
    ordered_edges = nx.bfs_edges(G, root)
    ordered_nodes = [root] + [v for _, v in ordered_edges]
    sampled_nodes = ordered_nodes[:n_nodes]
    return nx.subgraph(G, sampled_nodes)
```

Let us read and sample the actor co-star network. We'll ignore the weights:

```
costar = nx.read_edgelist('actors_costar.edges.gz', data=[('w',int)])
costar1k_random = random_sample(costar, 1000)
costar1k_snowball = snowball_sample(costar, 1000)
```

1. Let us compare the densities of the two subnetworks:

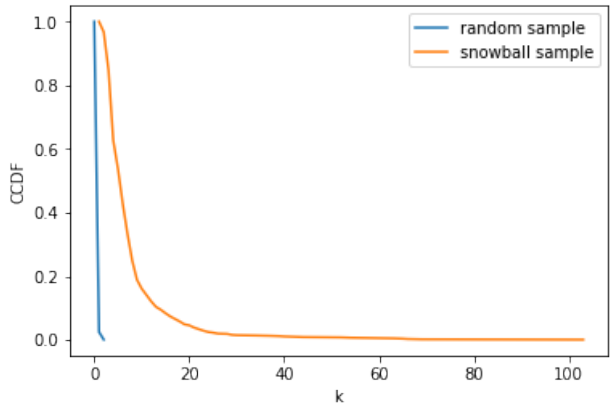
```
print('density of random sample:', nx.density(costar1k_random))
print('density of snowball sample:', nx.density(costar1k_snowball))
# density of random sample: 2.6026026026026025e-05
# density of snowball sample: 0.006764764764764765
```

The densities are not the same: the random subnetwork has much lower density because it is more likely to sample peripheral nodes with low degree (the majority) and less likely to sample hubs with high degree (the minority). Also it misses many of a sampled node's links because the chances are low that it will sample the node's neighbors. Snowball sampling gets all the links of a node and is more likely to find the hubs by following the links. 2. Let us compare the average path lengths. Note that the random subnetwork is not necessarily connected, whereas the snowball subnetwork is connected by definition:

```
if nx.is_connected(costar1k_random):
    print('APL of random sample is',
          nx.average_shortest_path_length(costar1k_random))
else:
    GC = max(nx.connected_component_subgraphs(costar1k_random),
             key=len)
    print('APL of random sample giant component is',
          nx.average_shortest_path_length(GC))
print('APL of snowball sample is',
      nx.average_shortest_path_length(costar1k_snowball))
# APL of random sample giant component is 1.3333333333333333
# APL of snowball sample is 4.257621621621622
```

In fact the random subnetwork is not connected and we must focus on its giant component, which has a much shorter APL than the snowball subnetwork. A likely reason is the small size of the random subnetwork's giant component. 3. To compare the distributions we use the CCDF function defined in Ex. 5.15:

```
ccdf_random = CCDF([k for _,k in costar1k_random.degree()])
ccdf_snowball = CCDF([k for _,k in costar1k_snowball.degree()])
plt.plot(ccdf_random.keys(), ccdf_random.values(),
         label="random sample")
plt.plot(ccdf_snowball.keys(), ccdf_snowball.values(),
         label="snowball sample")
plt.xlabel('k')
plt.ylabel('CCDF')
plt.legend()
```



The distributions are very different: the snowball sampling preserves the heavy tail of the original network's degree distribution, while the random sampling does not. In fact, we do not use log scale in the plot because the majority of nodes in the random subnetwork are singletons and we would not see $k = 0$ with log scale. The random sample method not only misses the hubs, but does not preserve the degree of the sample nodes.

Exercise 6.22. Carefully following the hints, we read the network from file as undirected, remove self-loops, and apply core decomposition before running the Louvain community detection. Then we can color the nodes using the community labels:

```
import community
email = nx.read_edgelist('email-Enron.edges.gz',
                         create_using=nx.Graph(),
                         data=[('weight',int)])
email.remove_edges_from(nx.selfloop_edges(email))
email_core = nx.k_core(email, k=43)
email_part_louvain = community.best_partition(email_core)
colors = [email_part_louvain[n] for n in email_core.nodes()]
sizes = [email_core.degree(n) for n in email_core.nodes()]
nx.draw(email_core, node_color=colors, node_size=sizes)
```

The yellow community is the module at the right-hand side of Figure 0.4

Chapter 7

Exercise 7.4. Any of the nodes **2**, **3**, **5** and **8** would trigger a cascade leading to the activation of all nodes of the respective community to which they belong. For instance, if we activate node **2**, node **1** would be activated as it has two neighbors, one of which is active (**1**), then node **3** would be activated, since it has two active neighbors out of three (**1** and **2**) and finally **4** would become active because 2/3 of its neighbors are active (**2** and **3**). The process stops there, however, because node **5** has four neighbors and only one of them is active (**4**), which is not sufficient to activate **5**. So, in order to trigger a cascade that activates all nodes we need to activate at least one node in each community, e.g. **2** and **5**.

Exercise 7.10. From Equation 7.5 we see that the basic reproduction number R_0 is proportional to the average number of contacts (degree) $\langle k \rangle$ and inversely proportional to the recoevery rate μ. If we can decrease $\langle k \rangle$ by a half, the reproduction number goes down to $R_0/2 = 1.25$, still above the threshold. So, if we can increase μ by a factor of 1.25 (25%) or more, we can bring R_0 below one, and stop the epidemic.

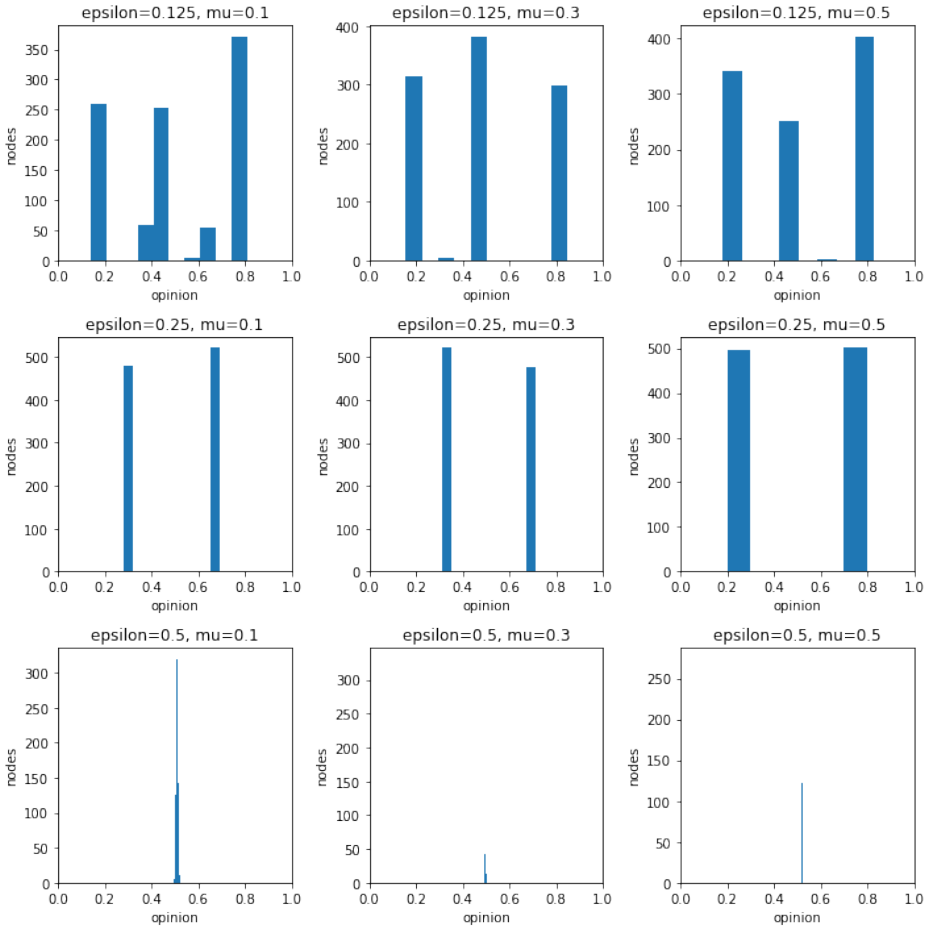
Exercise 7.19. We can adapt the examples in the Chapter 7 tutorial for the initial state and state transition functions to the bounded-confidence model. We update the states asynchronously and in random order. We also need a function to check the stop condition:

```
def initial_state(G):
    state = {}
    for node in G.nodes:
        state[node] = random.random()
    return state
```

```
# asynchronous updates in random order
def state_transition(G, current_state, epsilon, mu):
    state = current_state.copy()
    shuffled_nodes = list(G.nodes())
    random.shuffle(shuffled_nodes)
    for node in shuffled_nodes:
        neigh = random.choice(list(G.neighbors(node)))
        if abs(state[node] - state[neigh]) < epsilon:
            oi = state[node] + mu * (state[neigh] - state[node])
            oj = state[neigh] + mu * (state[node] - state[neigh])
            state[node] = oi
            state[neigh] = oj
    return state

def convergence(old, new):
    for n in old:
        if abs(old[n] - new[n]) >= 0.01 * old[n]:
            return False
    return True
```

Now we can run the simulations. This solution does not use the `Simulation` class from the tutorial. We plot a histogram of the final opinions for each value of ϵ and μ:

```
complete = nx.complete_graph(1000)

fig = plt.figure(figsize=(10,10))
epsilons = [0.125, 0.25, 0.5]
mus = [0.1, 0.3, 0.5]
i = 0
for eps in epsilons:
    for mu in mus:
        i += 1
        old_state = initial_state(complete)
        while True:
            new_state = state_transition(complete, old_state,
                                         eps, mu)
            if convergence(old_state, new_state):
                break
            old_state = new_state
        plt.subplot(3, 3, i)
        plt.xlim(0,1)
        plt.hist(new_state.values())
        plt.title('epsilon=' + str(eps) + ', mu=' + str(mu))
        plt.xlabel('opinion')
        plt.ylabel('nodes')
plt.tight_layout()
plt.show()
```



The number of opinion clusters decreases as ϵ increases, according to the relationship discussed in the previous exercise. The value of μ does not affect the number of opinion clusters.