Network Analysis and Modeling
CSCI 5352, Fall 2020
Prof. Dan Larremore
Problem Sets 9 and 10
Student: Alex Book; Collaborators: None

1. (100 pts total) *Minimum Violations Rankings.* Here, you will implement the minimum violations algorithm and explore rankings in directed random networks.

   (a) (60 pts) *Coding up and testing the MVR algorithm.* Recall the MVR algorithm that we discussed in class: let π be an ordering of the network's n vertices, such that π_i is the ranking of vertex i, and using the convention that n is the best ranking and 1 is the worst ranking. Let $V(\pi)$ be the number of directed edges that violate the ordering by pointing from a lower ranked vertex to a higher ranked vertex. In other words, in a network where A_{ij} denotes an edge from i to j,

   $$V(\pi) = \sum_{ij} A_{ij} I_{\pi_i < \pi_j},$$

   where $I_{\pi_i<\pi_j}$ is an indicator function taking on the value 1 when $\pi_i < \pi_j$, and 0 otherwise.

   The algorithm begins with a random assignment of vertices to ranks. Then, at each step t, two nodes are chosen uniformly at random from the current ranking $\pi^{(t)}$ and we *propose* to swap them to create a new ordering $\pi^{(t+1)}$. If $V(\pi^{(t+1)}) \leq V(\pi^{(t)})$, the proposal is accepted, and we keep the swap and call the new ordering $\pi^{(t+1)}$. Otherwise, we reject the swap and let $\pi^{(t+1)} = \pi^{(t)}$. As a stopping condition, let the algorithm end only when $\binom{n}{2}$ proposals in a row have passed (accepted or unaccepted) with no decrease in V.

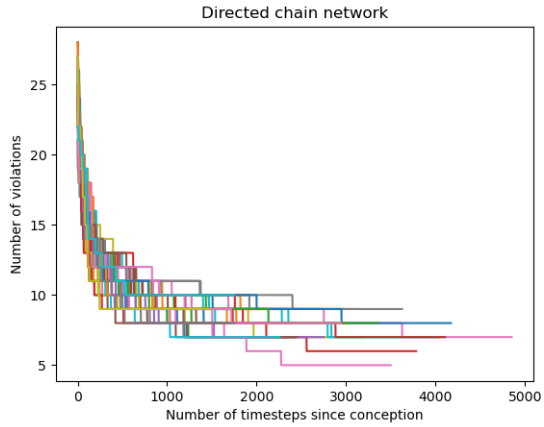
   After your algorithm is implemented, confirm that it works as you expect by testing two cases: (i) a directed chain of $n = 50$ vertices such that $A_{i,i+1} = 1$ for $i = 1, 2, \ldots, 49$. (ii) a $G(n = 50, p = 0.15)$ random network in which edges are made directed by forcing each undirected edge between i and j to point from i to j when $i < j$ and from j to i when $j < i$.

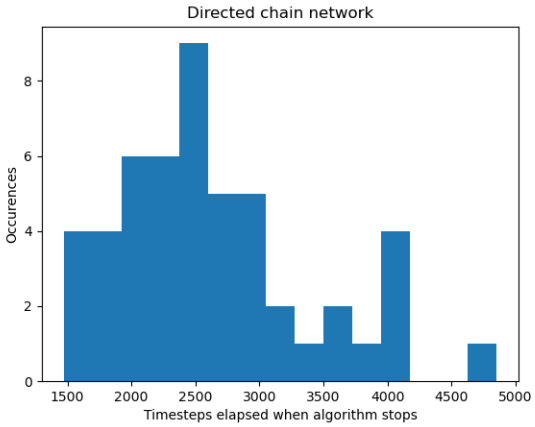
   First, write down what you expect the final ranking π of the algorithm to be, for each of the two cases.

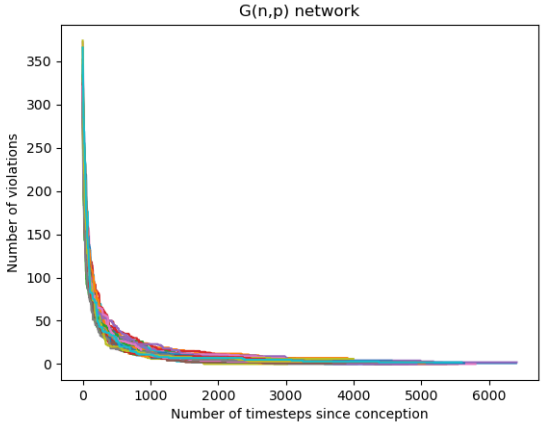
   I expect that the results of the directed chain will take the form of the biggest values being toward the beginning of the chain and the smallest values toward the end of the chain (ideally, it would be exactly ordered by descending value). I expect the results of the G(n,p) network will be that of a network with the largest value nodes having the highest out-degrees (ideally, the larger the value, the lesser the chance of having in-edges at all) and the smallest value nodes having the highest in-degrees (ideally, the smaller the value, the lesser the chances of having out-edges at all).

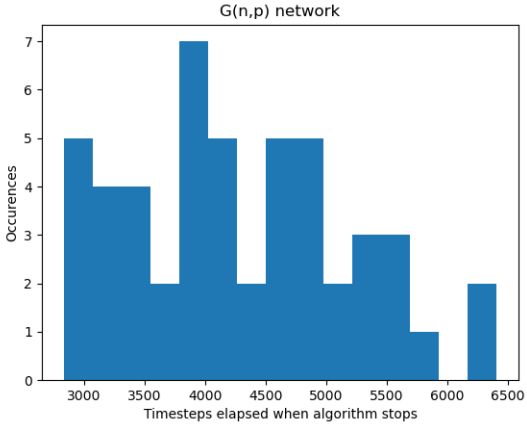
Second, run your algorithm on each of the two network types (starting from random initial π) 50 times. For case (ii), regenerate a new random network and new random initial π each time. For each of these two cases, produce two plots. Plot 1: time-series plot of the number of violations $V(t)$ vs the number of timesteps t for each repetition of the algorithm. In other words, there should be 50 lines. Plot 2: a histogram of the total number of timesteps elapsed when the algorithm stops.

Please don't forget to include your code as an appendix to this homework file.









As can be seen from the figures above, the number of violations decreases over time on each network as you accept the swaps. The shape of the curve makes sense, as it is "easier" to make improvements when there are many violations (there is a high chance that any random swap will improve the ranking), and "harder" when there are fewer violations (there is a low chance that any random swap will improve the ranking). The distribution of the timesteps elapsed when the algorithm stops has a roughly normal-looking distribution (with only 50 runs, the histogram is bound to be less than ideal in the smoothness department, but the trend is still fairly clear), which is to be expected as well (random initial ranking and random edges lead to a normal distribution of length for this algorithm).

(b) (40 pts) *Misled! Rankings in randomness!* Here, you will generate random directed networks, in which we expect no meaningful linear hierarchy to exist. However, just like modularity maximization, we can always minimize violations, *even if the resulting communities or hierarchy are simply fluctuations in randomness.* You will explore how the parameters of a directed $G(n, p)$ network affect the apparent strength of hierarchies that the MVR algorithm finds.

First, write some code to create *directed* random networks. Let us place an edge from i to j independently for all *ordered* pairs of i and j with probability p. (This means that we will create $i \to j$ with probability p, and then independently make $j \to i$ with probability p.) Let us also prohibit self loops. Note that this method for forming a network is different from the method in the previous question. There, we created an undirected random network and then made it directed. Here, we create a directed network from the beginning.

Conduct an exploration of the effects of n and p on the expected number of violations found by the MVR algorithm when applied to directed $G(n, p)$ networks. Write a brief one-page summary of this exploration by telling me (1) how you attacked the problem, (2) what you found, (3) and what you expect to happen if we dramatically increase the size of the network or the probability of connection p. You may include any figures or mathematics that you wish!

This exploration is intended to be open-ended! Don't forget to repeat any numerical experiments a reasonable number of times so that your expected values of V as a function of n and p are not too noisy.

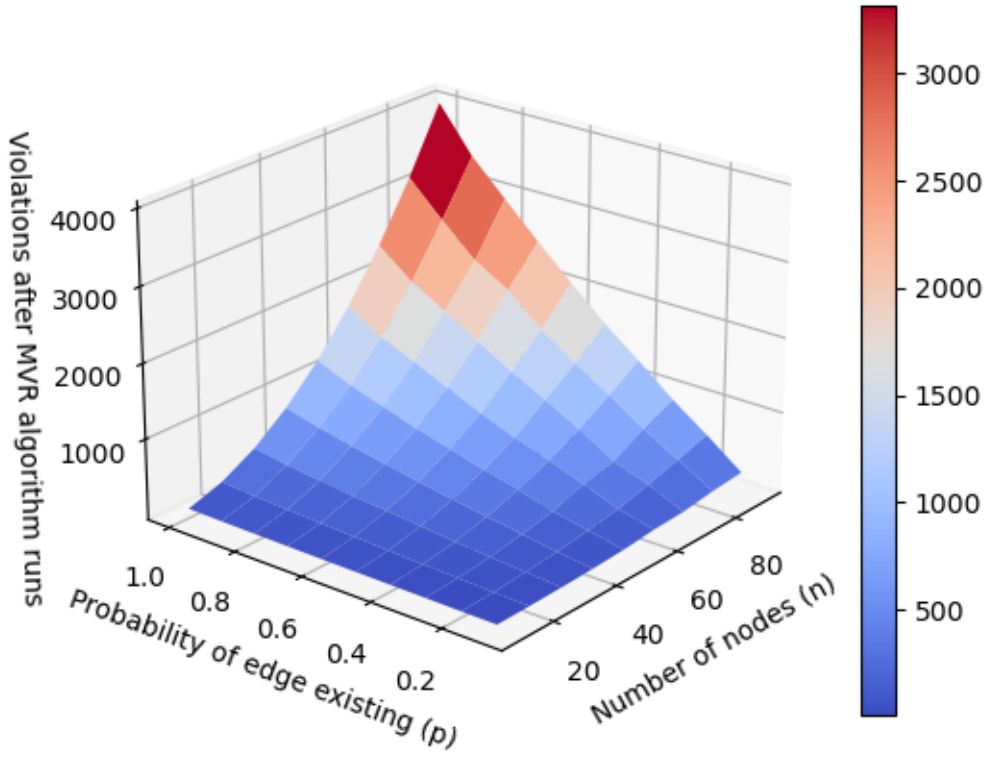
I attacked this problem as in the steps that follow:

i. Create G(n, p) graph
   A. Create networkx graph with n nodes
   B. Iterate over non-self-loop directed edge possibilities, creating that edge with probability p
ii. Perform the MVR algorithm on the created graph
iii. Once the algorithm has completed, find the number of violations "left over" in the graph
iv. Repeat steps i-iii some defined number of times and store the average of those runs
v. Repeat steps i-iv for some number of defined (n, p) pairs
vi. Plot the number of "left over" violations against n and p, and look for trends

I first found that the number of violations is significantly higher than what we have seen in the graphics from part (a). This makes sense, because in part (a), the structure of the graph essentially created pre-defined "slots" where optimally, certain vertices would end up (at the end of the MVR algorithm). For example, the graph is created with the lowest-value vertex having only outgoing edges. That "slot" is exactly where you would want a high-value vertex to end up (as high-value vertices ideally should only have outgoing edges, or at least a very low chance of having any incoming edges). So, the graph structure from part (a) allows for better eventual rankings than the random structure from part (b) (since it doesn't create these well-fitting "slots"), leaving more

violations at the end of the MVR algorithm's completion in part (b).

I also found that the number of violations experiences exponential growth as n and p increase. In all honesty, I'm not entirely sure what I expected to happen when we drastically increased n and p. It's clear that the number of violations increases exponentially as n and p increase, but the reasoning behind that isn't quite clear to me. It could be that for some constant p, the number of violations is, on average, some fraction of the total number of edges, so increasing n (which effectively increases the total number of edges) would strictly increase the number of violations. It could also be that for some constant n, the number of violations is, on average, some fraction of the total number of edges, so increasing p (which effectively increases the total number of edges) would strictly increase the number of violations. Or, it could be that both of those are true, which means that increasing n and p would make the number of violations skyrocket. This is the explanation that makes the most sense to me, but I'm not absolutely positive about this line of reasoning.



In order to lessen the strain on my computer, I limited the number of values for p and n for the figure. However, the same trends would be seen with a more granular selection of values.

2. (20 pts extra credit) Using your *SI* simulation from Problem Set 4, construct a new centrality measure, which we will call *spreading centrality*. We define the spreading centrality s_i of a node i to be the average size of a cascade (number of infected nodes when the epidemic is complete) that is seeded at i and when the transmission probability is $1/c$, for the network's mean degree c. Thus, the bigger the average cascade that a node i tends to produce, the more important it is under this measure.

   Choose two moderately-sized networks ($n > 500$) from the *Index of Complex Networks* and numerically calculate s for all vertices in each network. Produce a table listing the names of the top 10 nodes, ordered by their spreading centrality, and report their centrality scores and their degree, for each network. Briefly comment on what you discover, with respect to the two networks you chose.

3. (10 pts extra credit) Suppose that there exists a directed and unweighted adjacency matrix A with n vertices. Suppose further that we have used SpringRank to compute the ranks $\hat{s}$ by minimizing the SpringRank Hamiltonian,

$$\hat{s} = \arg\min_s \left\{ \frac{1}{2} \sum_{ij} A_{ij} \mu (s_i - s_j - \ell)^2 \right\} \tag{1}$$

   Now, suppose that a new node t appears. Suppose further that t has neighbors in a set $\mathcal{N}_t$, and that t has out-degree of $|\mathcal{N}_t|$ and therefore t also has in-degree of 0. Given existing estimates of the ranks, $\hat{s}$, which are assumed to be fixed, where should we place the new node t in the embedding space? (Hint: imagine t to be a newcomer to a league in which we already have SpringRank-derived rankings; if t wins all her games, and her opponents' rankings are given by $\hat{s}$, can you derive the SpringRank $\hat{s}_t$ of our new competitor t?) Your answer should be expressed in terms of existing ranks, i.e. elements of $\hat{s}$ and other variables above.

Code for problem 1a:

```
import numpy as np
from pprint import pprint
import networkx as nx
import matplotlib
from matplotlib import pyplot as plt
import pandas as pd

def MVR(G):
    n = G.number_of_nodes()
    iterSinceImprovement = 0
    total_iter = 0
    violations_vs_iter = {}
    while iterSinceImprovement < np.math.factorial(n)/(2*np.math.factorial(n-2)):
        # pick two random nodes
        a, b = np.random.randint(n), np.random.randint(n)
        # propose swap
        swapped_G = nx.relabel.relabel_nodes(G, {a:b, b:a})
        original_V = V(G)
        swap_V = V(swapped_G)
        # if the swap decreases V, keep it and reset iterSinceImprovement back to zero
        if swap_V < original_V:
            G = swapped_G.copy()
            iterSinceImprovement = 0
        # if the swap keeps V the same, keep it and increment iterSinceImprovement
        elif swap_V == original_V:
            G = swapped_G.copy()
            iterSinceImprovement += 1
        # if the swap increases V, keep the original graph and increment iterSinceImprovement
        else:
            iterSinceImprovement += 1
        violations_vs_iter[total_iter] = V(G)
        total_iter += 1
    return violations_vs_iter

def V(G):
    count = 0
    for node in G.nodes():
        for neighbor in G.neighbors(node):
            if node < neighbor:
                count += 1
    return count

def createLinearGraph(n):
```

```
    G = nx.DiGraph()
    arr = np.arange(n)
    np.random.shuffle(arr)
    for i in range(n):
        G.add_node(i)
    for i in range(len(arr)-1):
        G.add_edge(arr[i], arr[i+1])
    # nx.draw(G, with_labels=True)
    # plt.show()
    return G

def G_n_p(n, p):
    G = nx.DiGraph()
    arr = np.arange(n)
    for i in range(n):
        G.add_node(i)

    for node1 in G.nodes():
        for node2 in G.nodes():
            if np.random.uniform() <= p:
                if node1 <= node2:
                    G.add_edge(node1, node2)
                else:
                    G.add_edge(node2, node1)
    # nx.draw(G, with_labels=True)
    # plt.show()
    return G


if __name__ == '__main__':
    dict_arr1 = []
    len_arr1 = []
    for i in range(50):
        print(i)
        G = createLinearGraph(50)
        dict_arr1.append(MVR(G))

    df = pd.DataFrame(dict_arr1[0], index=[0]).transpose()
    ax = df.plot()
    len_arr1.append(len(dict_arr1[0]))

    for dict in dict_arr1[1:]:
        df = pd.DataFrame(dict, index=[0]).transpose()
        df.plot(ax=ax)
        len_arr1.append(len(dict))
```

```
ax.get_legend().remove()
ax.set_title('Directed chain network')
ax.set_xlabel('Number of timesteps since conception')
ax.set_ylabel('Number of violations')
plt.show()

plt.hist(len_arr1, bins=15)
plt.title('Directed chain network')
plt.xlabel('Timesteps elapsed when algorithm stops')
plt.ylabel('Occurences')
plt.show()

dict_arr1 = []
len_arr1 = []
for i in range(50):
    print(i)
    G = G_n_p(50, .15)
    dict_arr1.append(MVR(G))

df = pd.DataFrame(dict_arr1[0], index=[0]).transpose()
ax = df.plot()
len_arr1.append(len(dict_arr1[0]))

for dict in dict_arr1[1:]:
    df = pd.DataFrame(dict, index=[0]).transpose()
    df.plot(ax=ax)
    len_arr1.append(len(dict))

ax.get_legend().remove()
ax.set_title('G(n,p) network')
ax.set_xlabel('Number of timesteps since conception')
ax.set_ylabel('Number of violations')
plt.show()

plt.hist(len_arr1, bins=15)
plt.title('G(n,p) network')
plt.xlabel('Timesteps elapsed when algorithm stops')
plt.ylabel('Occurences')
plt.show()
```

Code for problem 1b:

```
import numpy as np
from pprint import pprint
import networkx as nx
import matplotlib
from matplotlib import pyplot as plt
import pandas as pd
from matplotlib import cm

def MVR(G):
    n = G.number_of_nodes()
    iterSinceImprovement = 0
    total_iter = 0
    # violations_vs_iter = {}
    while iterSinceImprovement < np.math.factorial(n)/(2*np.math.factorial(n-2)):
        # pick two random nodes
        a, b = np.random.randint(n), np.random.randint(n)
        # propose swap
        swapped_G = nx.relabel.relabel_nodes(G, {a:b, b:a})
        original_V = V(G)
        swap_V = V(swapped_G)
        # if the swap decreases V, keep it and reset iterSinceImprovement back to zero
        if swap_V < original_V:
            G = swapped_G.copy()
            iterSinceImprovement = 0
        # if the swap keeps V the same, keep it and increment iterSinceImprovement
        elif swap_V == original_V:
            G = swapped_G.copy()
            iterSinceImprovement += 1
        # if the swap increases V, keep the original graph and increment iterSinceImprovement
        else:
            iterSinceImprovement += 1
        # violations_vs_iter[total_iter] = V(G)
        total_iter += 1
    return V(G)

def V(G):
    count = 0
    for node in G.nodes():
        for neighbor in G.neighbors(node):
            if node < neighbor:
                count += 1
    return count
```

```
def G_n_p(n, p):
    G = nx.DiGraph()
    arr = np.arange(n)
    for i in range(n):
        G.add_node(i)

    for node1 in G.nodes():
        for node2 in G.nodes():
            if np.random.uniform() <= p and node1 != node2:
                G.add_edge(node1, node2)
    # nx.draw(G, with_labels=True)
    # plt.show()
    return G

def make_plots(filename, a, b):
    dfNew = pd.read_csv(filename)

    X = list(dfNew['n'])
    Y = list(dfNew['p'])
    Z = list(dfNew['violations after algorithm run'])
    x = np.reshape(X, (a, b))
    y = np.reshape(Y, (a, b))
    z = np.reshape(Z, (a, b))
    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')
    q = ax.plot_surface(x, y, z, cmap = cm.coolwarm)
    fig.colorbar(q)
    ax.set_xlabel('Number of nodes (n)')
    ax.set_ylabel('Probability of edge existing (p)')
    ax.set_zlabel('Violations after MVR algorithm runs')
    plt.show()

if __name__ == '__main__':
    V_n_p_dict = {}
    n_arr = [10, 20, 30, 40, 50, 60, 70, 80, 90]
    p_arr = [.1, .2, .3, .4, .5, .6, .7, .8, .9, 1]

    for n in n_arr:
        for p in p_arr:
            print(n, p)
            V_arr = []
            for i in range(50):
                G = G_n_p(n, p)
                V_arr.append(MVR(G))
            V_n_p_dict[(n,p)] = np.mean(V_arr)
```

```
print(V_n_p_dict)
df = pd.DataFrame(columns = ['n', 'p', 'violations after algorithm run'])
for key, value in V_n_p_dict.items():
    df.loc[-1] = [key[0], key[1], value]
    df.index += 1
df.to_csv('data.csv')

a = len(n_arr)
b = len(p_arr)
make_plots('data.csv', a, b)
```