CSCI - 5352 (Network Analysis and Modeling) - Problem Set 5

Name of Student : Rajarshi Basak

November 7, 2018

[1] (a) (i) ***Expectation of final ranking:*** For a directed chain of $n = 50$ vertices such that $A_{i,i+1} = 1$ for i = 1,2,...,49, the final ranking is expected to be such that Node 1 has rank 50 (best rank), Node 2 has rank 49 (second best rank), Node 3 has rank 48 (third best rank), ... , Node 50 has rank 1 (lowest rank).

First we show the plots for a single run.

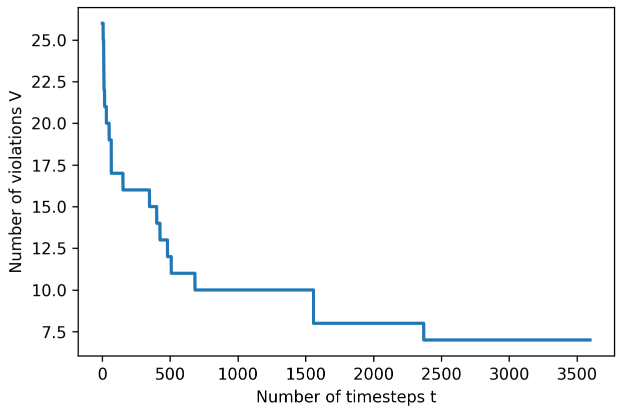


Figure 1: Plot of Number of violations V(t) vs. Number of timesteps for a single run with a directed chain of 50 vertices

Second, we show the plots when we run the algorithm after starting from the initial π 50 times.

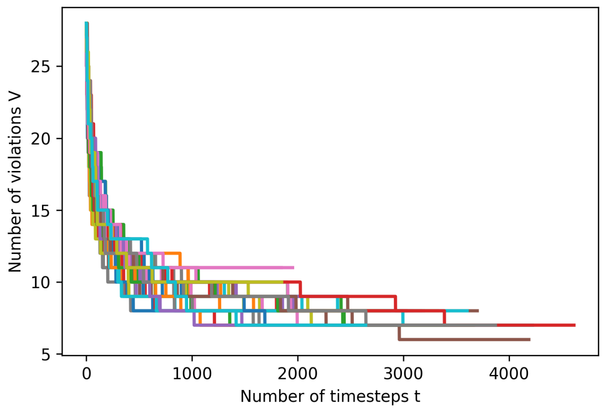


Figure 2: Plot of Number of violations V(t) vs. Number of timesteps for 50 runs with a directed chain of 50 vertices

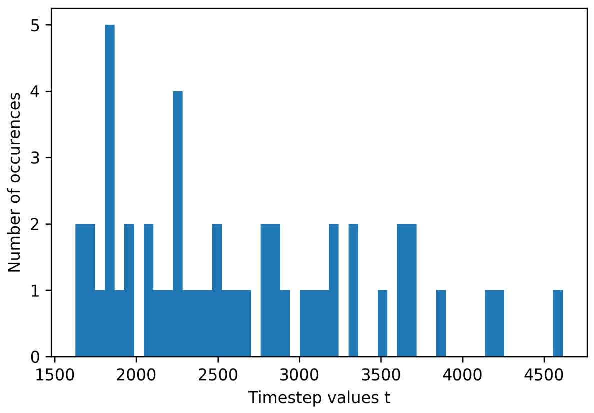


Figure 3: Histogram of the total number of timesteps elapsed when the algorithm stops for 50 runs with a directed chain of 50 vertices

Finally, we show the plots when we run the algorithm after starting from the initial π 10,000 times.

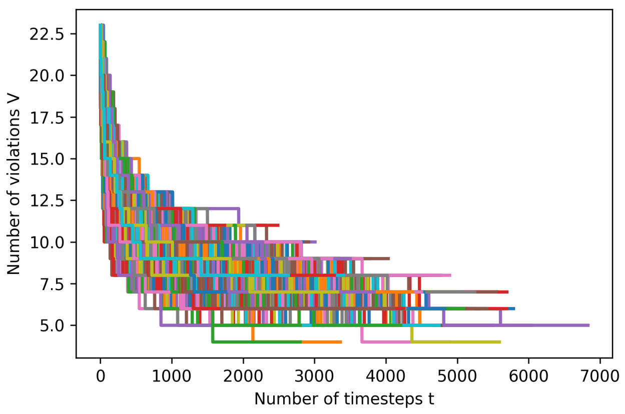


Figure 4: Plot of Number of violations V(t) vs. Number of timesteps for 10000 runs with a directed chain of 50 vertices

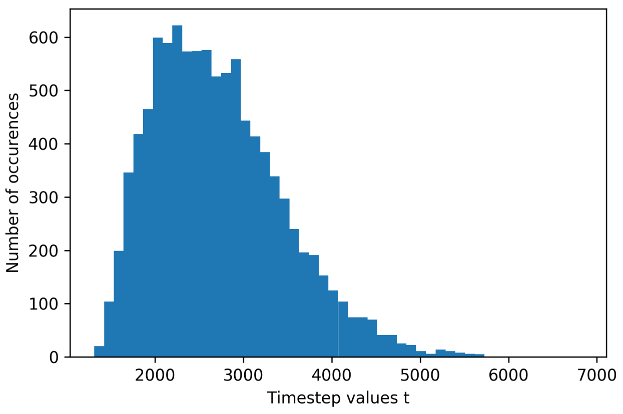


Figure 5: Histogram of the total number of timesteps elapsed when the algorithm stops for 10000 runs with a directed chain of 50 vertices

In the last plot for the histogram (for 10000 runs of the algorithm), a clearer picture of the distribution is seen. The maximum number of timesteps (600 times) appears to occur for a value 2300 steps.

[1] (a) (ii) ***Expectation of final ranking:*** For 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 $j < i$, the final rankings are expected to be somewhat similar to that of directed chain, but slightly different due to the inherent randomness of the way the graphs were generated. Here, for the correct ordering we again have an adjacency matrix such that there are only entries above the diagonal, but without any order or pattern.

First we show the plots for a single run.

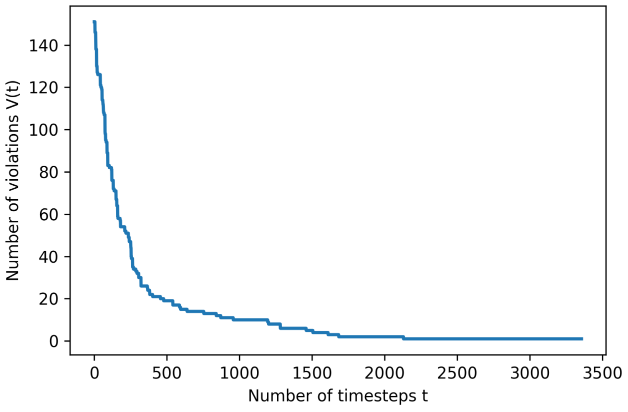


Figure 6: Plot of Number of violations V(t) vs. Number of timesteps for a single run with a random graph of 50 vertices

Second, we show the plots when we run the algorithm after starting from the initial π 50 times.

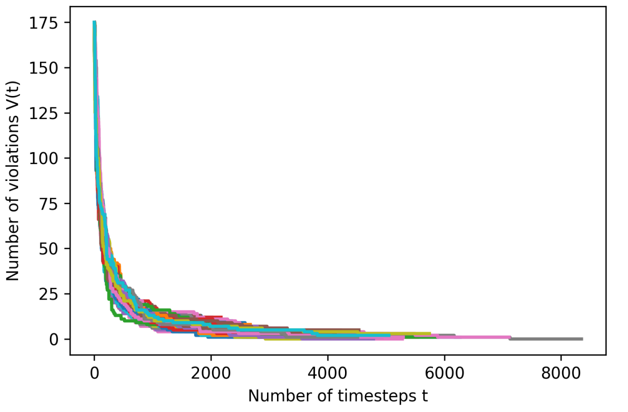


Figure 7: Plot of Number of violations V(t) vs. Number of timesteps for 50 runs with a random graph of 50 vertices

Here we note the plots for the number of violations as a function of the number of time-steps in the random graph exhibits a smoother decline compared to that for the directed chain. Since there are more edges in the random graph than the directed chain (single edges, or a linear network), the probability of the number of violations going down after a single swap of two vertices is much higher compared to that for the directed chain.

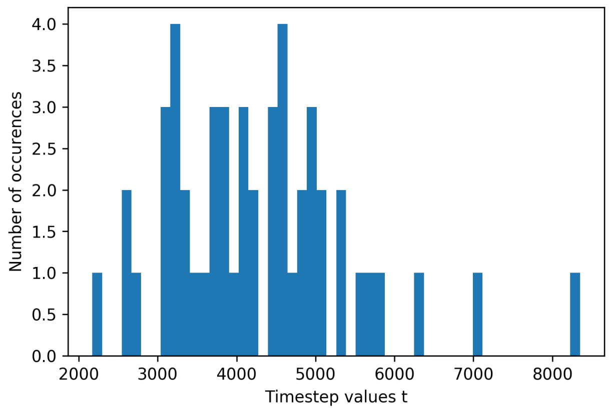


Figure 8: Histogram of the total number of timesteps elapsed when the algorithm stops for 50 runs with a random graph of 50 vertices

Finally, we show the plots when we run the algorithm after starting from the initial π 10,000 times.

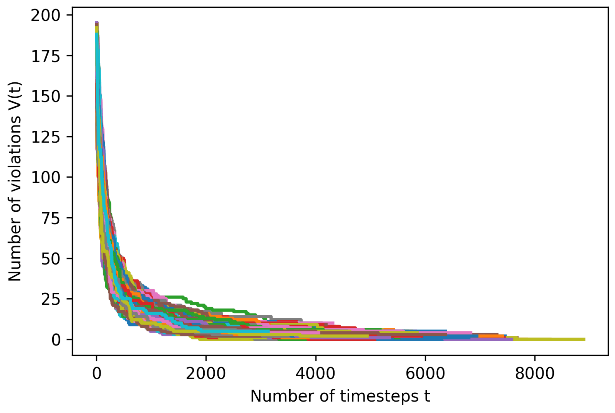


Figure 9: Plot of Number of violations V(t) vs. Number of timesteps for 1000 runs with a random graph of 50 vertices

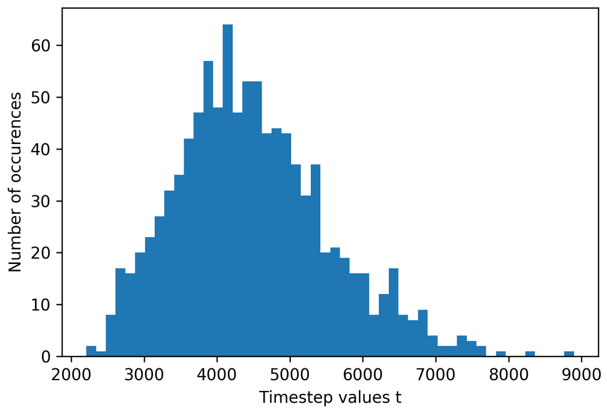


Figure 10: Histogram of the total number of timesteps elapsed when the algorithm stops for 1000 runs with a random graph of 50 vertices

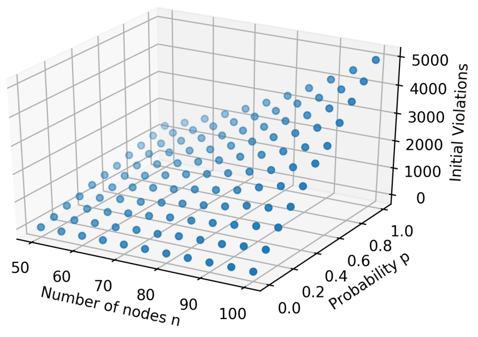
Effects of n and p on the expected number of violations found by the MVR algorithm when applied to directed $G(n,p)$ networks

[1] (b) (1) ***How the problem was attacked:***

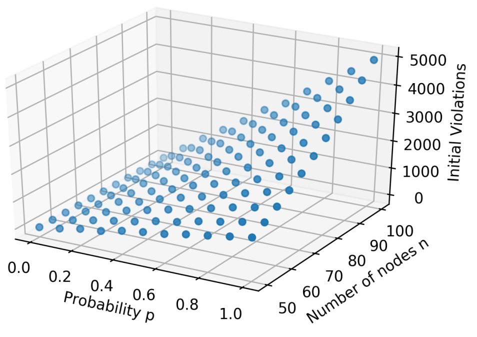
In this scenario, we generated random $G(n,p)$ graphs with n, the number of nodes in the graph in the range [50,100] in steps of 5, and p values (the probability of drawing an edge) in the range [0,1] in steps of 0.1. For each of the combinations of n and p, a random graph was generated, and the

1. Initial number of violations V_I
2. Average number of final violations $< V_F >$
3. Average decrease in number of violations $< \Delta V >$
4. Average number of time-steps $< t >$

were plotted as a function of both n and p in a 3d plot. There are two 3d plots for each of the four variables mentioned above, with the n and p axes interchanged for greater clarity. The term 'average' refers to the fact that the variable values were averaged over 10 runs to smooth noisy values.

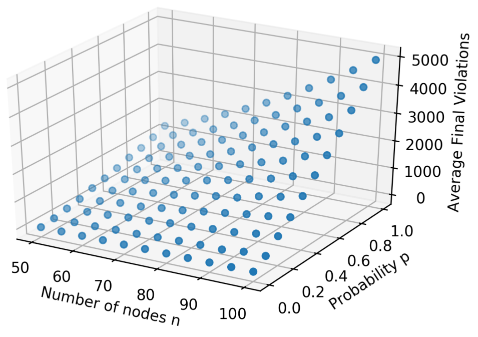


(a) Figure 1

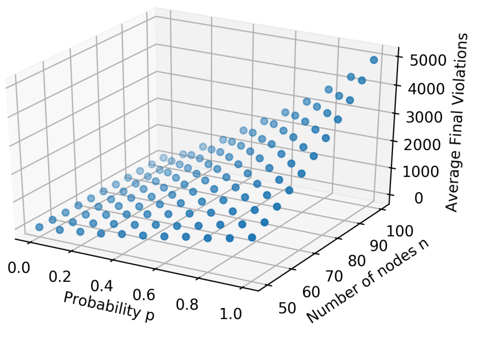


(b) Figure 2

Figure 11: Plot of Initial number of violations V_I as a function of the number of nodes n and the probability of drawing an edge p

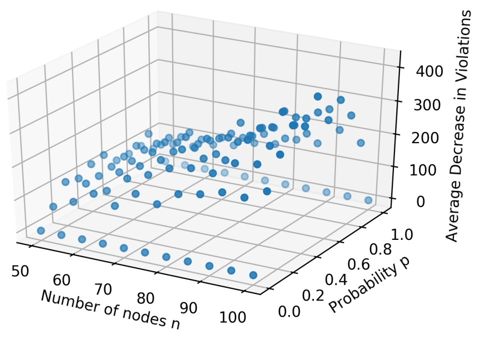


(a) Figure 1

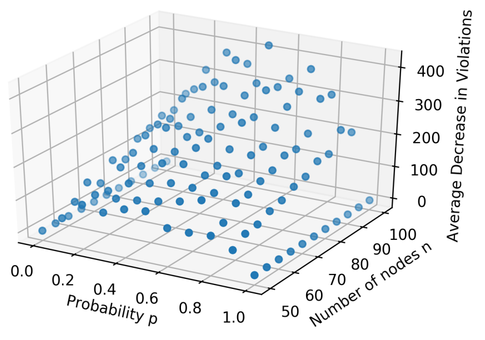


(b) Figure 2

Figure 12: Plot of Average number of final violations $< V_F >$ as a function of the number of nodes n and the probability of drawing an edge p

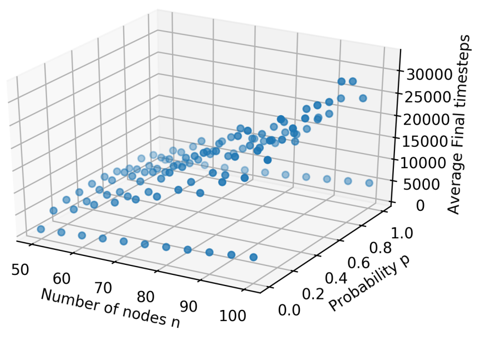


(a) Figure 1

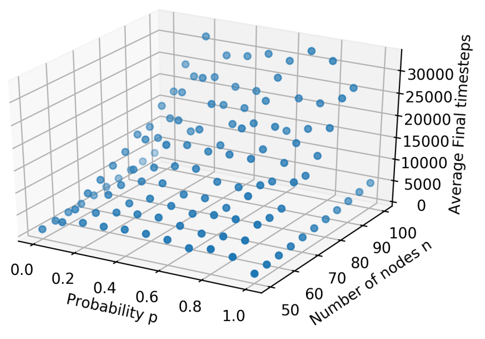


(b) Figure 2

Figure 13: Plot of Average decrease in number of violations $< \Delta V >$ as a function of the number of nodes n and the probability of drawing an edge p



(a) Figure 1



(b) Figure 2

Figure 14: Plot of Average number of time-steps $< t >$ as a function of the number of nodes n and the probability of drawing an edge p

[1] (b) (2) ***What we found:***

(1) In the 3d plots shown above for the Initial Number of Violations V_I, it is clear that as we increase the number of nodes n in the random graph, the initial number of violations increases steadily with the probability p of drawing a edge. This is simply because as the number of nodes n in the graph increases, the number of entries in the corresponding adjacency matrix also increases, and hence the number of entries in the lower triangle of the matrix also increases. In a similar fashion, as the probability p of drawing edge increases, the number of edges and hence the number of entries in the adjacency matrix increases.

(2) For the 3d plots representing the Average number of final violations $< V_F >$, we observe a similar trend - the number of average final violations increases gradually with the probability p of drawing an edge as we increase the number of nodes n in the graph, and this happens for reasons similar to those for the initial number of violations.

(3) When observing the plots for the Average decrease in number of violations $< \Delta V >$, an interesting trend is observed. For a given number of nodes n in the random graph, as we increase the probability p of drawing an edge from 0 to 1, the average decrease in the number of violations (after the algorithm stops) rises slowly from $p = 0$, reaches a maximum around $p = 0.5$ and descends almost symmetrically until $p = 1$. As we increase the number of nodes n in the graph, the rise tends to be more sharp, and the value of the maximum (around $p = 0.5$) goes up.

This behaviour can be attributed to the fact that as the probability p goes up, the number of edges in the graph increase and so does the number of entries in its adjacency matrix. The more the number of entries in the matrix, the more the chances that a random swap will lead to a lowering of the number of violations. However, there are two situations in which a random swap will never lead to a decrease in the number of violations - when the matrix has all zeros and all ones, which when either none of the nodes are connected, or all of the nodes are connected. Thus from the symmetry of the matrix with respect to these two situations, we can expect the decrease in number of violations to peak when roughly half of the possible edges in the graph exist, or roughly half of the entries in the matrix are 0 and half are 1.

(4) Finally, the 3d plot for the Average number of time-steps $< t >$, as expected, exhibits a trend somewhat similar to that of the plot for Average decrease in number of violations $< \Delta V >$. This is because, the number of time-steps the algorithm takes to complete is directly proportional to the the number of successful (so that the number of violations decreases in that swap) swaps, and hence the decrease in the number of violations.

For a given value of n, the curve rises as we increase the probability p, and then rises gradually till $p = 0.5$, and finally goes to 0 at $p = 1$.

[1] (b) (3) ***What we expect to happen if we dramatically increase the size of the network*** n ***or the probability of connection*** p***:***

(1) Keeping the value of probability of connection p fixed, if we increase the size of the network (drastically), the number of initial violations should go up, the number of final violations should go up, the average decrease in number of final violations should go up, and the number of time-steps taken to complete the algorithm should go up. In other words, all the parameters should scale with the size of the network.

(2) However, keeping the value of the size of the network fixed, if we increase the probability of connection p drastically, then due to the symmetry of the graph, and hence the adjacency matrix, the decrease in the number of violations and the total number of time-steps should go up, peak around $p = 0.5$ and then should go down symmetrically. The reason for this has been explained above. The initial and final number of violations though, still display the increasing trend.

The code for Problem [1] (a) (i) has been shown below.

```
import networkx as nx
import matplotlib.pyplot as plt
import random
import operator as op
import math
import numpy as np
import copy
import time

# Count time
start_time = time.time()

# Create a directed graph of n = 50 vertices such that A(i,i+1) = 1
# for i = 1,2,3,...,49
adj = [] # Adjacency matrix for the graph
k = [] # Actual ranking of the vertices
for i in range(50):
    rows = [0] * 50
    if (i != 49):
        rows[i + 1] = 1
    adj.append(rows)
    k.append(i)

# Define the factorial function for determining the stopping timr
# of the algorithm
def nCr(n,r):
    f = math.factorial
    return f(n) / f(r) / f(n-r)

# Define a function which shuffles the adjacency matrix based on the
# list of random (initially) or swapped ranks
def generate_matrix(r,a):
    s = (len(r), len(r))
    new_adj = np.zeros(s)
    #print(type(new_adj))
    xi = 0
    for i in r:
        xj = 0
        for j in r:
            #print(a[di[i]][di[j]])
            new_adj[xi][xj] = a[i][j]
            xj += 1
        xi += 1
    return new_adj

# Define a function which calculates the number of violations in the
# adjacency matrix by computing the number of non-zero entries below
# the diagonal
def violations(a):
    lowertriangle = np.tril(a)
    viol = np.sum(lowertriangle)
    return (int(viol))

min_vio1 = 0
# Shuffle the rankings
random.shuffle(k)

# Generate the initial number of violations
min_vio1 = violations(generate_matrix(k,adj))
print(min_vio1)

# Calcualate the stopping time
stopping_time = int(nCr(50, 2))

# Initialize the data structures for storing the timesteps and the
# values of the minimum violation values
minimum_violations_lists = []
minimum_violations_timesteps = []
hist_timesteps = []

# Outer loop for number of runs
for runs in range(0,1):
    time_taken = 0
    min_vio_list = []
    r = copy.deepcopy(k)
    min_vio_timestep = []
    min_vio = min_vio1
    mvt = 0
    #while there are no drops in V for nC2 iterations
    while (time_taken != stopping_time):
        mvt += 1
        min_vio_timestep.append(mvt)
        rand1 = 0
        rand2 = 0
        # Randomly pick any two different nodes
        while (rand1 == rand2):
            rand1 = random.randint(0, len(r)-1)
            rand2 = random.randint(0, len(r)-1)
        # Swap the two nodes in the rankings list r
        r[rand1],r[rand2] = r[rand2],r[rand1]
        # Get the adjacency matrix after swapping the two nodes
        shuffledmatrix = generate_matrix(r, adj)
        # Get the number of violations for the shuffled matrix
        vio = violations(shuffledmatrix)
        # If the number of violations ddcreases or stays the same
        if (vio <= min_vio):
            if (vio < min_vio):
                time_taken = 0
            else:
                time_taken += 1
            min_vio = vio
        else:
            r[rand1],r[rand2] = r[rand2],r[rand1]
            time_taken += 1
        min_vio_list.append(min_vio)

    # Update the lists for plotting
    minimum_violations_lists.append(copy.deepcopy(min_vio_list))
    minimum_violations_timesteps.append(copy.deepcopy(min_vio_timestep))
    hist_timesteps.append(len(min_vio_timestep))

# Plotting V vs. t for all runs
for list1, list2 in zip(minimum_violations_timesteps,
                        minimum_violations_lists):
    plt.plot(list1, list2, linestyle='-',linewidth=2)
plt.xlabel('Number of timesteps t')
plt.ylabel('Number of violations V')
plt.savefig('V_versus_t_directed-chain-50_single-run', dpi = 300)
plt.show()

# Plotting the histogram for the timesteps
x = hist_timesteps
plt.hist(x, normed=False, bins=50)
plt.xlabel('Timestep values t')
plt.ylabel('Number of occurences')
plt.savefig('Histogram_directed-chain-50_single-run', dpi = 300)
plt.show()

# Record elapsed time
elapsed_time = time.time() - start_time

#print((elapsed_time/3600), 'hours' )
print(elapsed_time, 'seconds' )
```

The code for [1] (a) (ii) has been shown below.

```
import numpy as np
import random
import math
import matplotlib.pyplot as plt
import copy
import time

# Count time
start_time = time.time()

# Generating a random graph
n = 50
m = 50
p = 0.15
k = []
ad = np.zeros((n,m))
for i in range(0,n):
    for j in range(0,m):
        rand = random.uniform(0, 1)
        if (rand <= p):
            if (i < j):
                ad[i][j] = 1
            if (j < i):
                ad[j][i] = 1
    k.append(i)

adj = ad.tolist()

# Define the factorial function for determining the stopping timr
# of the algorithm
def nCr(n,r):
    f = math.factorial
    return f(n) / f(r) / f(n-r)

# Define a function which shuffles the adjacency matrix based on the
# list of random (initially) or swapped ranks
def generate_matrix(r,a):
    s = (len(r), len(r))
    new_adj = np.zeros(s)
    #print(type(new_adj))
    xi = 0
    for i in r:
        xj = 0
        for j in r:
            #print(a[di[i]][di[j]])
            new_adj[xi][xj] = a[i][j]
            xj += 1
        xi += 1
    return new_adj

# Define a function which calculates the number of violations in the
# adjacency matrix by computing the number of non-zero entries below
# the diagonal
def violations(a):
    lowertriangle = np.tril(a)
    viol = np.sum(lowertriangle)
    return (int(viol))

min_vio1 = 0
# Shuffle the rankings
random.shuffle(k)

# Generate the initial number of violations
min_vio1 = violations(generate_matrix(k,adj))
print(min_vio1)

# Calcualate the stopping time
stopping_time = int(nCr(50, 2))

# Initialize the data structures for storing the timesteps and the
# values of the minimum violation values
minimum_violations_lists_r = []
minimum_violations_timesteps_r = []
hist_timesteps_r = []

# Outer loop for number of runs
for runs in range(0,1000):
    time_taken = 0
    min_vio_list_r = []
    r = copy.deepcopy(k)
    min_vio_timestep_r = []
    min_vio = min_vio1
    mvt = 0
    #while there are no drops in V for nC2 iterations
    while (time_taken != stopping_time):
        mvt += 1
        min_vio_timestep_r.append(mvt)
        rand1 = 0
        rand2 = 0
        # Randomly pick any two different nodes
        while (rand1 == rand2):
            rand1 = random.randint(0, len(r)-1)
            rand2 = random.randint(0, len(r)-1)
        # Swap the two nodes in the rankings list r
        r[rand1],r[rand2] = r[rand2],r[rand1]
        # Get the adjacency matrix after swapping the two nodes
        shuffledmatrix = generate_matrix(r, adj)
        # Get the number of violations for the shuffled matrix
        vio = violations(shuffledmatrix)
        # If the number of violations ddcreases or stays the same
        if (vio <= min_vio):
            if (vio < min_vio):
                time_taken = 0
            else:
                time_taken += 1
            min_vio = vio
        else:
            r[rand1],r[rand2] = r[rand2],r[rand1]
            time_taken += 1
        min_vio_list_r.append(min_vio)

    # Update the lists for plotting
    minimum_violations_lists_r.append(copy.deepcopy(min_vio_list_r))
    minimum_violations_timesteps_r.append(copy.deepcopy(min_vio_timestep_r))
    hist_timesteps_r.append(len(min_vio_timestep_r))


# Plotting V vs. t for all runs
for list1, list2 in zip(minimum_violations_timesteps_r,
                        minimum_violations_lists_r):
    plt.plot(list1, list2, linestyle='-',linewidth=2)
plt.xlabel('Number of timesteps t')
plt.ylabel('Number of violations V(t)')
plt.savefig('V_versus_t_random-graph-50_runs-1000', dpi = 300)
plt.show()

# Plotting the histogram for the timesteps
x = hist_timesteps_r
plt.hist(x, normed=False, bins=50)
plt.xlabel('Timestep values t')
plt.ylabel('Number of occurences')
plt.savefig('Histogram_random-graph-50_runs-1000', dpi = 300)
plt.show()

# Record elapsed time
elapsed_time = time.time() - start_time

print(elapsed_time, 'seconds')
print((elapsed_time/3600), 'hours')
```

The code for [1] (b) has been shown below.

```
import numpy as np
import random
import math
import matplotlib.pyplot as plt
import copy
import time
import matplotlib as mpl
from mpl_toolkits.mplot3d import Axes3D

# Plot the violations V(t) as a function of time (t) and a function of
# the probabilities [0,1] in steps of 0.1 keeping n = 50 fixed

# Count time
start_time = time.time()

number_of_nodes = np.arange(50.0, 100.01, 5.0)
probabilities = np.arange(0.0, 1.001, 0.1)
runs = np.arange(1.0, 10.01, 1.0)

# Generating a random graph
def random_graph_generator(n, m, p):

    k = []
    n1 = int(n)
    m1 = int(m)
    ad = np.zeros((n1,m1))
    for i in range(0,n1):
        for j in range(0,m1):
            rand = random.uniform(0, 1)
            if (rand <= p):
                if (i != j):
                    ad[i][j] = 1
        k.append(i)

    adj = ad.tolist()

    return (k, adj)

# Define the factorial function for determining the stopping timr
# of the algorithm
def nCr(n,r):
    f = math.factorial
    return f(n) / f(r) / f(n-r)

# Define a function which shuffles the adjacency matrix based on the
# list of random (initially) or swapped ranks
def generate_matrix(r,a):
    s = (len(r), len(r))
    new_adj = np.zeros(s)
    #print(type(new_adj))
    xi = 0
    for i in r:
        xj = 0
        for j in r:
            #print(a[di[i]][di[j]])
            new_adj[xi][xj] = a[i][j]
            xj += 1
        xi += 1
    return new_adj

# Define a function which calculates the number of violations in the
# adjacency matrix by computing the number of non-zero entries below
# the diagonal
def violations(a):
    lowertriangle = np.tril(a)
    viol = np.sum(lowertriangle)
    return (int(viol))


#----------------------------------------------------------------------

array_probability = []
array_nodenumber = []
array_initial_violations = []
array_final_violations = []
array_decrease_in_violations = []
array_final_timesteps = []


for node_num in number_of_nodes:

    for pr in probabilities:

        k, adj = random_graph_generator(node_num, node_num, pr)
        random.shuffle(k)
        min_vio1 = 0
        min_vio1 = violations(generate_matrix(k,adj))
        stopping_time = int(nCr(node_num, 2))



        total_final_violations = 0
        total_final_timesteps = 0

        for run in runs:
            time_taken = 0
            min_vio_list_r = []
            r = copy.deepcopy(k)
            min_vio_timestep_r = []
            min_vio = min_vio1
            mvt = 0
            #while there are no drops in V for nC2 iterations
            while (time_taken != stopping_time):
                mvt += 1
                min_vio_timestep_r.append(mvt)
                rand1 = 0
                rand2 = 0
                # Randomly pick any two different nodes
                while (rand1 == rand2):
                    rand1 = random.randint(0, len(r)-1)
                    rand2 = random.randint(0, len(r)-1)
                # Swap the two nodes in the rankings list r
                r[rand1],r[rand2] = r[rand2],r[rand1]
                # Get the adjacency matrix after swapping the two nodes
                shuffledmatrix = generate_matrix(r, adj)
                # Get the number of violations for the shuffled matrix
                vio = violations(shuffledmatrix)
                # If the number of violations ddcreases or stays the same
                if (vio <= min_vio):
                    if (vio < min_vio):
                        time_taken = 0
                    else:
                        time_taken += 1
                    min_vio = vio
                else:
                    r[rand1],r[rand2] = r[rand2],r[rand1]
                    time_taken += 1
                min_vio_list_r.append(min_vio)



            total_final_violations += min_vio
            total_final_timesteps += mvt

        array_probability.append(pr)
        array_nodenumber.append(node_num)

        array_initial_violations.append(min_vio1)
        array_final_violations.append(total_final_violations/len(runs))
        array_decrease_in_violations.append(min_vio1 - (total_final_violations/len(runs)))
        array_final_timesteps.append(total_final_timesteps/len(runs))



mpl.rcParams['legend.fontsize'] = 10
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.scatter(array_nodenumber, array_probability, array_initial_violations)
ax.set_xlabel('Number of nodes n')
ax.set_ylabel('Probability p')
ax.set_zlabel('Initial Violations')
ax.legend()
plt.savefig('Initial-Number-of-Violations_vs_Number-of-Nodes_vs_Probability', dpi = 300)
plt.show()

mpl.rcParams['legend.fontsize'] = 10
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.scatter(array_probability, array_nodenumber,  array_initial_violations)
ax.set_xlabel('Probability p')
ax.set_ylabel('Number of nodes n')
ax.set_zlabel('Initial Violations')
ax.legend()
plt.savefig('Initial-Number-of-Violations_vs_Probability_vs_Number-of-Nodes', dpi = 300)
plt.show()

mpl.rcParams['legend.fontsize'] = 10
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.scatter(array_nodenumber, array_probability, array_final_violations)
ax.set_xlabel('Number of nodes n')
ax.set_ylabel('Probability p')
ax.set_zlabel('Average Final Violations')
ax.legend()
plt.savefig('Final-Avg-Number-of-Violations_vs_Number-of-Nodes_vs_Probability', dpi = 300)
plt.show()

mpl.rcParams['legend.fontsize'] = 10
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.scatter(array_probability, array_nodenumber, array_final_violations)
ax.set_xlabel('Probability p')
ax.set_ylabel('Number of nodes n')
ax.set_zlabel('Average Final Violations')
ax.legend()
plt.savefig('Final-Avg-Number-of-Violations_vs_Probability_vs_Number-of-Nodes', dpi = 300)
plt.show()

mpl.rcParams['legend.fontsize'] = 10
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.scatter(array_nodenumber, array_probability, array_decrease_in_violations)
ax.set_xlabel('Number of nodes n')
ax.set_ylabel('Probability p')
ax.set_zlabel('Average Decrease in Violations')
ax.legend()
plt.savefig('Decrease-in-Violations_vs_Number-of-Nodes_vs_Probability', dpi = 300)
plt.show()

mpl.rcParams['legend.fontsize'] = 10
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.scatter(array_probability, array_nodenumber, array_decrease_in_violations)
ax.set_xlabel('Probability p')
ax.set_ylabel('Number of nodes n')
ax.set_zlabel('Average Decrease in Violations')
ax.legend()
plt.savefig('Decrease-in-Violations_vs_Probability_vs_Number-of-Nodes', dpi = 300)
plt.show()

mpl.rcParams['legend.fontsize'] = 10
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.scatter(array_nodenumber, array_probability, array_final_timesteps)
ax.set_xlabel('Number of nodes n')
ax.set_ylabel('Probability p')
ax.set_zlabel('Average Final timesteps')
ax.legend()
plt.savefig('Final-Avg-Number-of-Timesteps_vs_Number-of-Nodes_vs_Probability', dpi = 300)
plt.show()

mpl.rcParams['legend.fontsize'] = 10
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.scatter(array_probability, array_nodenumber, array_final_timesteps)
ax.set_xlabel('Probability p')
ax.set_ylabel('Number of nodes n')
ax.set_zlabel('Average Final timesteps')
ax.legend()
plt.savefig('Final-Avg-Number-of-Timesteps_vs_Probability_vs_Number-of-Nodes', dpi = 300)
plt.show()

# Record elapsed time
elapsed_time = time.time() - start_time

print(elapsed_time, 'seconds' )
print((elapsed_time/3600), 'hours' )
```