Homework 8
Colorado CSCI 5454

Cameron Mattson

November 2, 2021

People I studied with for this homework: Soumyadeb Maity, Hari Anantharaman, Saiyma Sarmin

Other external resources used: Notes, Book

Problem 1

Part a

To count the number of triangles, where the order of the triangle's vertices matter, first compute A_G^3. Then, the total number of triangles $= \sum_{i=0}^{n-1} A_G^3(i,i)$, where n is the number of vertices in the graph. Cubing the adjacency matrix will result in a matrix where each element $A_G^3(i,j)$ represents the number of paths from vertex i to vertex j after stepping 3 times. From this we know that $A_G^3(i,i)$ represents the number of paths from vertex i to vertex i in 3 steps.

Assuming no self-loops, the only way to end at the same starting vertex in 3 steps is to follow a triangular path. For each triangle, there are exactly 2 paths from any vertex in that triangle to that same vertex. This would not be the case if the graph were directed. Therefore, the total number of triangles is the sum of the number of triangles for each vertex $= \sum_{i=0}^{n-1} A_G^3(i,i)$.

Part b

To count the number of triangles we implement the same approach in Part a except, after computing A_G^3, the total number of triangles in this problem $= \sum_{i=0}^{n-1} \frac{A_G^3(i,i)}{6}$, since the order of the triangles vertices doesn't matter.

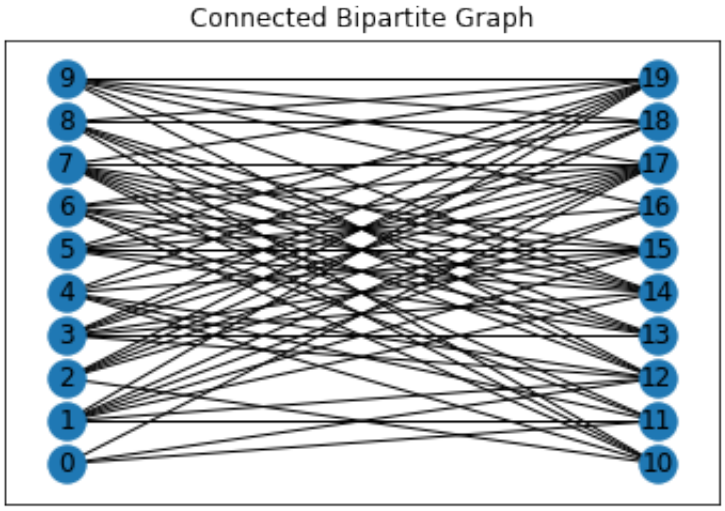
We divided by 6, because there are 3 vertices in a triangle, and for each of these vertices there are 2 repeat triangles. Therefore, for every unique triangle we will count 6 triangles

with repeat vertices, so we can divide the total number of duplicate triangles by 6. This will be equivalent to dividing the result from part a by 6.

Problem 2

Please find the code to all parts of Problem 2 in the last section of this document.

Part a



To generate the graph, a builtin function from the bipartite module in networkx was called. The name of this function is *random_graph*, and it generates the bipartite graph with four parameters. The first parameter is the number of vertices in partition U, and the second parameter is the number of vertices in the partition V. The probability of an edge being created between any two pairs of vertices in opposite partitions is the third parameter. The last parameter is the seed, which is used to generate randomness when creating edges. The specific values of these parameters were 10, 10, 0.5, and 101, respectively.

When this graph is generated there is no guarantee the graph will be connected. To solve this problem, I used a while loop to generate random bipartite graphs until a connected graph was generated. If a connected graph was generated, the *is_connected* function of the networkx module would return True and the loop would exit.

Part b

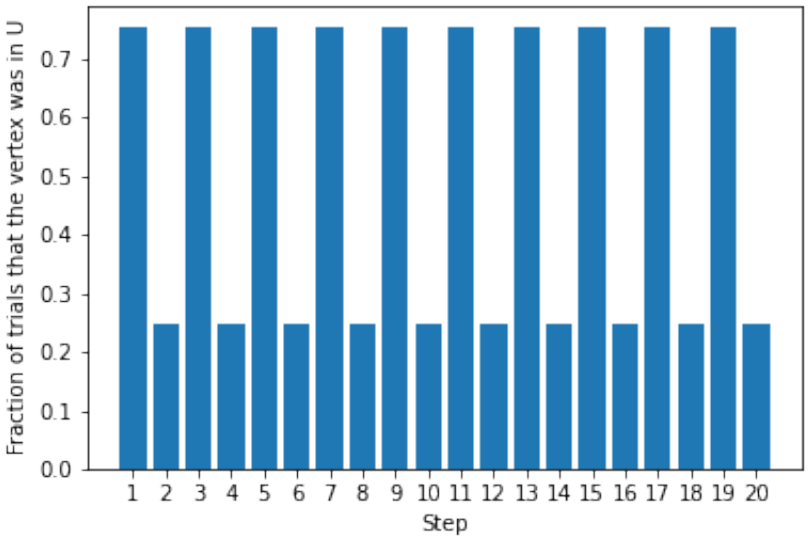


Figure 1. Fraction of trials that the vertex was in U per Step

In this plot the values tend to alternate between 0.75 and 0.25. This is because there is a 0.75 chance the starting vertex will be in partition V, so the probability the vertex is in partition U in the first step is 0.75, since each step corresponds to jumping to a vertex in the opposite partition. If the algorithm and graph were designed differently, this might not be the case, but since we are using a bipartite graph and in each step we jump to another neighbor, this holds true. Therefore, the probability that a vertex will be in the U partition on a particular step follows from the starting distribution, since the walk is guaranteed to jump to the opposite partition every step.

Part c

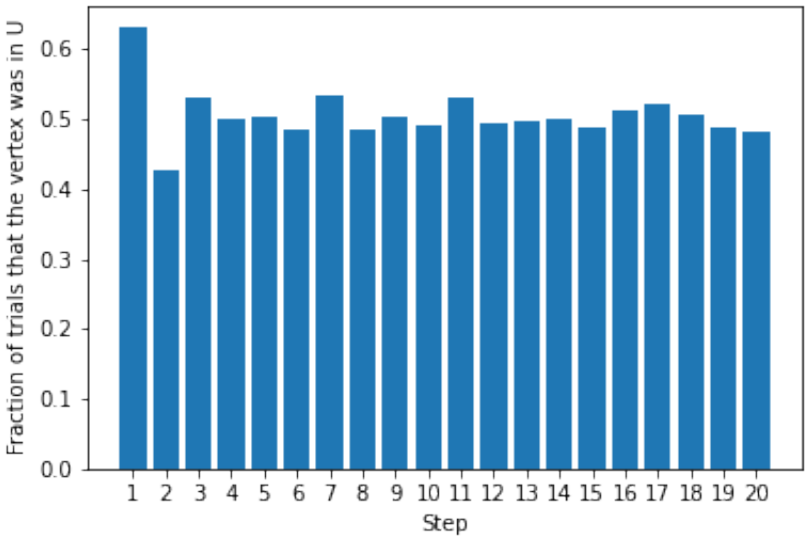


Figure 2. Fraction of trials that the vertex was in U with a 3/4 chance of jumping to a neighboring vertex per Step

In this plot the fraction of trials that the vertex was in partition U is more uniform because there is a non-zero chance that the walk will remain at the same vertex rather than jumping to a neighboring vertex in the opposite partition. This increases the chance that a walk will select a vertex in the U partition, such that the initial distribution influences the distribution of vertices selected to a smaller extent for each step.

In Part b the fraction of the trials in which the first step was in partition U was about 0.75, but that fraction has been reduced in this graph, since there is a chance there will be no jump to a neighboring vertex in step 1. Similarly, the steps corresponding to the fraction of trials that were 0.25 in Part b are larger in this plot because there is non-zero chance the walk will remain at the same vertex in a previous step. This means that there is a greater chance that we will select a vertex in partition U for one of these steps.

Part d

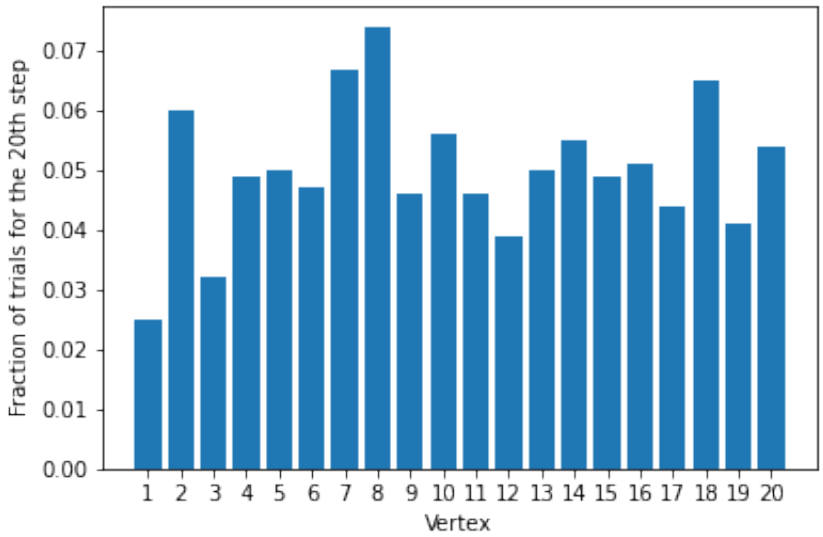


Figure 3. Fraction of trials that the final step was a particular vertex for the 20th step

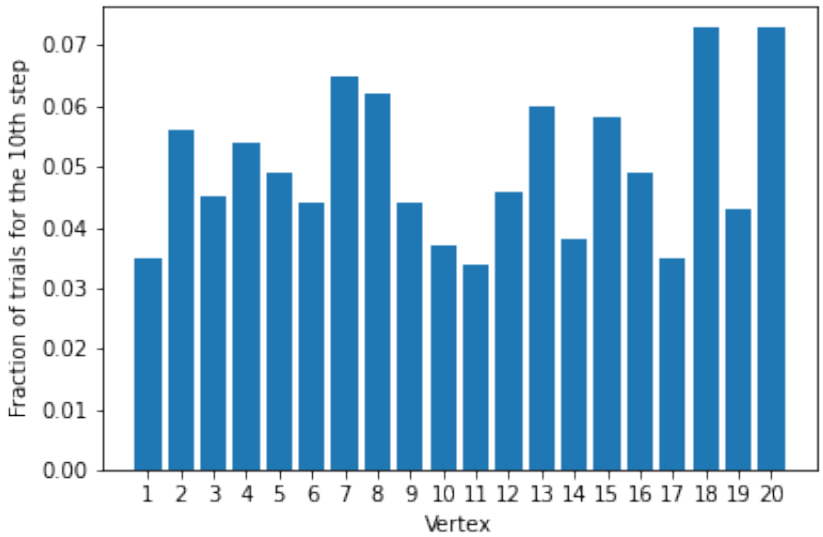


Figure 4. Fraction of trials that the final step was a particular vertex for the 10th step

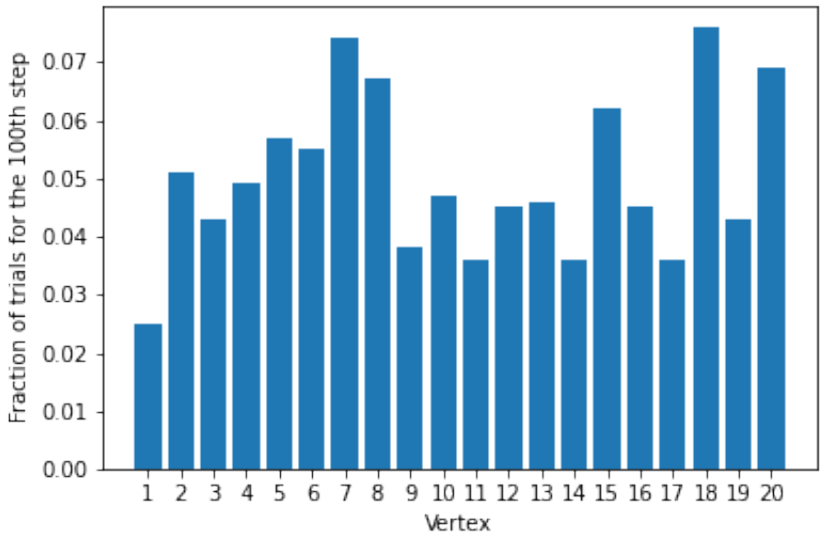


Figure 5. Fraction of trials that the final step was a particular vertex for the 100th step

Each of these histograms appears less uniform than the histogram in Part c. The distribution of the final vertex does not change significantly as we take more steps. This is because we are converging to the stationary distribution, since we are taking 1000 walks for each walk length, where we will still have the same number of vertices each time.

Problem 3

Part a

If the maximum degree of a vertex, r, does not stay reasonably small than the probability of jumping to a neighboring vertex will be less likely as r increases. This probability will be proportional to $\frac{1}{r}$. If this is the case, than converging to the stationary distribution may take more iterations, or may not converge with the computational resources available.

Similarly, if there are many neighbors for a vertex, than finding these neighbors will increase the time complexity as compared to finding the neighbors of a vertex with a small degree.

Part b

We know that $\frac{f(u)}{f(v)}$ is inversely proportional to $\frac{f(v)}{f(u)}$. Considering our weight function, we do not want $\frac{f(v)}{f(u)}$ to be too small because than there would be a smaller chance of jumping to a neighboring vertex in this case. This would lead to one of the same problems in Part a, which was that convergence to the stationary distribution would be more difficult.

Conversely, if $\frac{f(v)}{f(u)}$ was larger, which means that $\frac{f(u)}{f(v)}$ is smaller, than there would be a greater chance of jumping to a neighboring vertex. If the algorithm already converged to the stationary distribution than continuing to jump to neighboring vertices may result in the algorithm converging to a false or unrepresentative stationary distribution.

Part c

If the graph does not have a good mixing time, than it will generally take longer to converge to a stationary distribution, which will increase time complexity due to the increase in the number of iterations.

Similarly, without a good mixing time, a random walk might converge to a false or unrepresentative stationary distribution as in Part b.

Code for Problem 2

If running this code, please make sure to run the code for Part a first so as to generate the graph data for the remaining parts.

Part a

```
import networkx as nx
import numpy as np
from random import seed
from random import randint
from networkx.algorithms import bipartite
import matplotlib.pyplot as plt
import time

pr = 1
edgepr = 0.5 # probability of edge creation
uvert = 10 # Number of vertices in U partition
vvert = 10 # Number of vertices in V partition
steps = 20 # Number of steps
walks = 1000
uvertcount = [0] * (uvert+vvert)
vertcount = 0
xax = np.arange(1,uvert+vvert + 1)
connected = bool(False)


# Creates a connected bipartite graph Part a:
while (not connected):
    B = bipartite.random_graph(uvert,vvert, edgepr, seed = 101)
    connected = nx.is_connected(B)

bottom_nodes, top_nodes = bipartite.sets(B) # Should create two set partitions

# Computes the lengths of the vertex partitions:
botlen = len(bottom_nodes) # U length
toplen = len(top_nodes) # V length
# Creates lists of the vertex partitions:
bottom_nodes = list(bottom_nodes) # Convertes the nodes to a list
top_nodes = list(top_nodes)

nx.draw_networkx(B, pos = nx.drawing.layout.bipartite_layout(B, bottom_nodes))
plt.title('Connected Bipartite Graph')
```

```
plt.savefig('parta.png')
adjmat = nx.adjacency_matrix(B, weight=None).toarray() # The adjacency matrix should hav
# Saves my data for use in the other parts of problem 2
np.save('adjacency_matrix.npy',adjmat)
np.save('bottom_nodes.npy',bottom_nodes)
np.save('top_nodes.npy',top_nodes)
```

Part b

```
import networkx as nx
import numpy as np
from random import seed
from random import randint
from networkx.algorithms import bipartite
import matplotlib.pyplot as plt
import time

pr = 1
edgepr = 0.5 # probability of edge creation
uvert = 10 # Number of vertices in U partition
vvert = 10 # Number of vertices in V partition
steps = 20 # Number of steps
walks = 1000
uvertcount = [0] * (uvert+vvert)
vertcount = 0
xax = np.arange(1,uvert+vvert + 1)

# Loads my saved graph data
adjmat = np.load('adjacency_matrix.npy')
bottom_nodes = np.load('bottom_nodes.npy')
bottom_nodes = bottom_nodes.tolist()
top_nodes = np.load('top_nodes.npy')
top_nodes = top_nodes.tolist()


def randwalk(): # Part b
    global vertcount
    startval = randint(1,4) # Generates a random number between 1 and 4 inclusive
    #print(startval)

    if startval == pr: # If true, the first vertex will be in the U partition
        randind = randint(0,botlen-1)
        startver = bottom_nodes[randind]

    else: # Otherwise, the first vertex will be in the V partition
        randind = randint(0,toplen-1)
        startver = top_nodes[randind]

    for step in range(0,steps):
        indvec = np.where(adjmat[startver,:] == 1)[0] # Vector of vector containing indi
```

```
        neilen = len(indvec) # degree of start vertex
        randnei = indvec[randint(0,neilen-1)] # the random neighbor index choosen
        startver = randnei # The new start vertex is the random neighbor index
        vertcount = vertcount + 1

        myind = 0
        match = bool(False)
        while (not match):
            if startver == bottom_nodes[myind]:
                match = bool(True)
                uvertcount[step] = uvertcount[step] + 1

            if myind == (botlen - 1):
                match = bool(True)

            myind = myind + 1

for walk in range(0,walks):
    randwalk() # Part b

uvertcount = np.asarray(uvertcount)
uvertcount = uvertcount[:] / (walks)

# Code used to generate the figure:

plt.bar(xax,uvertcount,align='center')
plt.xticks(np.arange(1,steps+1))
plt.xlabel('Step')
plt.ylabel('Fraction of trials that the vertex was in U')
#plt.title('Fraction of trials that the vertex was in U per Step')
plt.savefig('partb.png')
```

Part c

```
import networkx as nx
import numpy as np
from random import seed
from random import randint
from networkx.algorithms import bipartite
import matplotlib.pyplot as plt
import time

pr = 1
edgepr = 0.5 # probability of edge creation
uvert = 10 # Number of vertices in U partition
vvert = 10 # Number of vertices in V partition
steps = 20 # Number of steps
walks = 1000
uvertcount = [0] * (uvert+vvert)
vertcount = 0
xax = np.arange(1,uvert+vvert + 1)

# Loads my saved graph data
adjmat = np.load('adjacency_matrix.npy')
bottom_nodes = np.load('bottom_nodes.npy')
bottom_nodes = bottom_nodes.tolist()
top_nodes = np.load('top_nodes.npy')
top_nodes = top_nodes.tolist()

def lazyrandwalk(): # Part c
    global vertcount
    startval = randint(1,4) # Generates a random number between 1 and 4 inclusive

    if startval <= pr: # If true, the first vertex will be in the U partition
        randind = randint(0,botlen-1)
        startver = bottom_nodes[randind]

    else: # Otherwise, the first vertex will be in the V partition
        randind = randint(0,toplen-1)
        startver = top_nodes[randind]

    for step in range(0,steps):
        prswitchval = randint(1,4)
        if prswitchval != pr:
            indvec = np.where(adjmat[startver,:] == 1)[0] # Vector of vector containing
```

```
            neilen = len(indvec) # degree of start vertex
            randnei = indvec[randint(0,neilen-1)] # the random neighbor index choosen
            startver = randnei # The new start vertex is the random neighbor index
        vertcount = vertcount + 1

        myind = 0
        match = bool(False)
        while (not match):
            if startver == bottom_nodes[myind]:
                match = bool(True)
                uvertcount[step] = uvertcount[step] + 1

            if myind == (botlen - 1):
                match = bool(True)

            myind = myind + 1

for walk in range(0,walks):
    lazyrandwalk() # Part c


uvertcount = np.asarray(uvertcount)
uvertcount = uvertcount[:] / (walks)

# Code used to generate the figures:

plt.bar(xax,uvertcount,align='center')
plt.xticks(np.arange(1,steps+1))
plt.xlabel('Step')
plt.ylabel('Fraction of trials that the vertex was in U')
#plt.title('Fraction of trials that the vertex was in U with a 3/4 chance of switching p
plt.savefig('partc.png')
```

Part d

```
# Change the steps variable value for each subpart of Part d for steps = 20, 10, and 100
import networkx as nx
import numpy as np
from random import seed
from random import randint
from networkx.algorithms import bipartite
import matplotlib.pyplot as plt
import time

pr = 1
edgepr = 0.5 # probability of edge creation
uvert = 10 # Number of vertices in U partition
vvert = 10 # Number of vertices in V partition
steps = 20 # Number of steps. This needs to be changed for each subpart so that steps =
walks = 1000
uvertcount = [0] * (uvert+vvert)
vertcount = 0
xax = np.arange(1,uvert+vvert + 1)

# Loads my saved graph data
adjmat = np.load('adjacency_matrix.npy')
bottom_nodes = np.load('bottom_nodes.npy')
bottom_nodes = bottom_nodes.tolist()
top_nodes = np.load('top_nodes.npy')
top_nodes = top_nodes.tolist()

def limlazywalk(steplen): # Part d
    steps = steplen
    global vertcount
    startval = randint(1,4) # Generates a random number between 1 and 4 inclusive

    if startval <= pr: # If true, the first vertex will be in the U partition
        randind = randint(0,botlen-1)
        startver = bottom_nodes[randind]

    else: # Otherwise, the first vertex will be in the V partition
        randind = randint(0,toplen-1)
        startver = top_nodes[randind]

    for step in range(0,steps):
        prswitchval = randint(1,4)
```

```
        if prswitchval != pr:
            indvec = np.where(adjmat[startver,:] == 1)[0] # Vector of vector containing
            neilen = len(indvec) # degree of start vertex
            randnei = indvec[randint(0,neilen-1)] # the random neighbor index choosen
            startver = randnei # The new start vertex is the random neighbor index
        vertcount = vertcount + 1

        if step == (steps-1):
            uvertcount[startver] = uvertcount[startver] + 1

# Change the steps variable value for each subpart of Part d for steps = 20, 10, and 100
for walk in range(0,walks):
    limlazywalk(steps) # Part d

uvertcount = np.asarray(uvertcount)
uvertcount = uvertcount[:] / (walks)

# Code used to generate the figures:
fig1, ax1 = plt.subplots()
plt.bar(xax,uvertcount,align='center')
plt.xticks(np.arange(1,(uvert+vvert)+1))
plt.xlabel('Vertex')
plt.ylabel('Fraction of trials for the ' + str(steps) + 'th step')
#plt.title('Fraction of trials that the final step was a particular vertex for the ' + s
plt.savefig('partd3.png')
```