Homework 8
Colorado CSCI 5454

Alex Book

November 2, 2021

People I studied with for this homework: Cole Sturza
Other external resources used: N/A

Problem 1

Part a

As proven in lecture 11 notes, $A_G^t(i,j)$ is equal to the number of paths from i to j in the graph G of length exactly t. So to find the number of distinct triangles, we simply need to find the number of paths of length 3 that start and end at the same vertex. To do so, we take the trace of A_G^3, as we need to sum the elements along the diagonal (as those will be the paths that start and end at the same vertex). This will count (a,b,c) and all other triangles including vertices $a, b,$ and c as unique, as they are all considered as having distinct starting points and ordering of vertices. Therefore, the number of distinct triangles where order matters can be represented by $\mathbf{Tr}(A_G^3)$.

Part b

To find the number of triplets (triangles where the vertex ordering doesn't matter, only the vertices included matters), we can simply divide the answer to the previous problem by 6. This is because when order matters, we have 3 possible starting vertices and 2 possible directions in which to move through the vertices, giving us 6 possible representations of any 3 vertices $a, b,$ and c. Therefore, the number of distinct triangles where order doesn't matter can be represented by $\frac{\mathbf{Tr}(A_G^3)}{6}$.

Problem 2

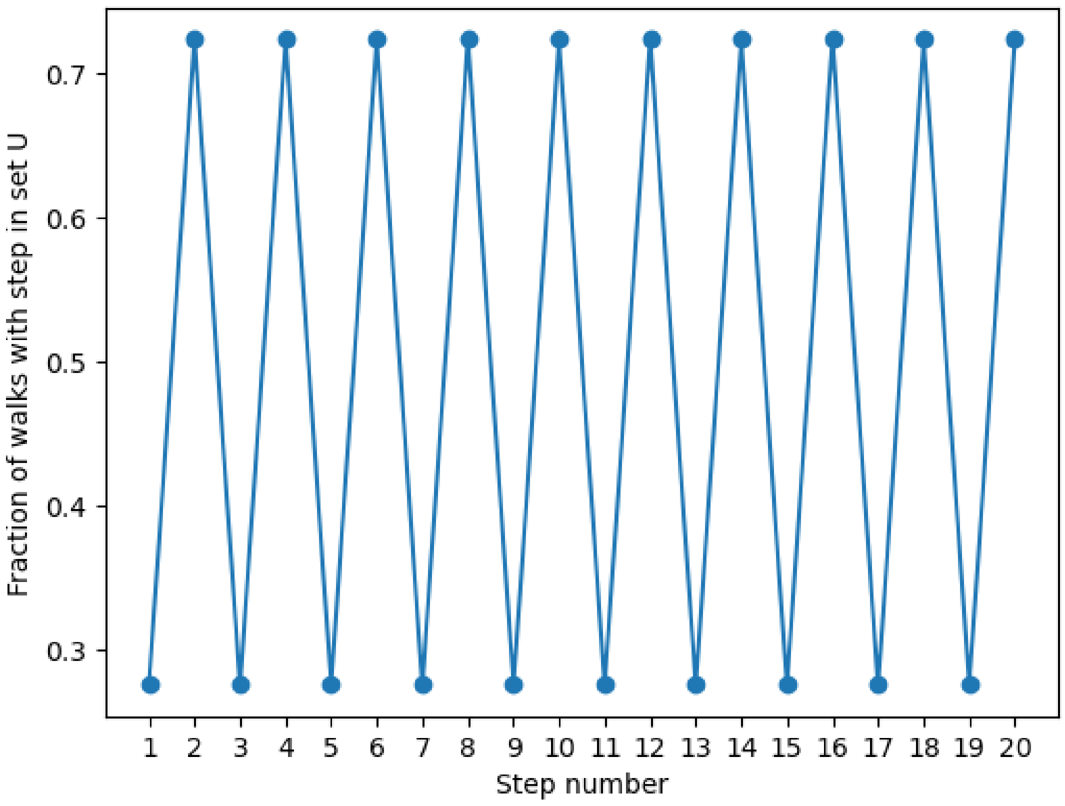
Note: All code is attached at the end of this PDF.

Part a

To create a nontrivial, unweighted, undirected bipartite graph, I used the Python NetworkX library. I add 10 edges to each set U and V, then randomly add edges connecting the two sets until the graph is no longer disconnected.

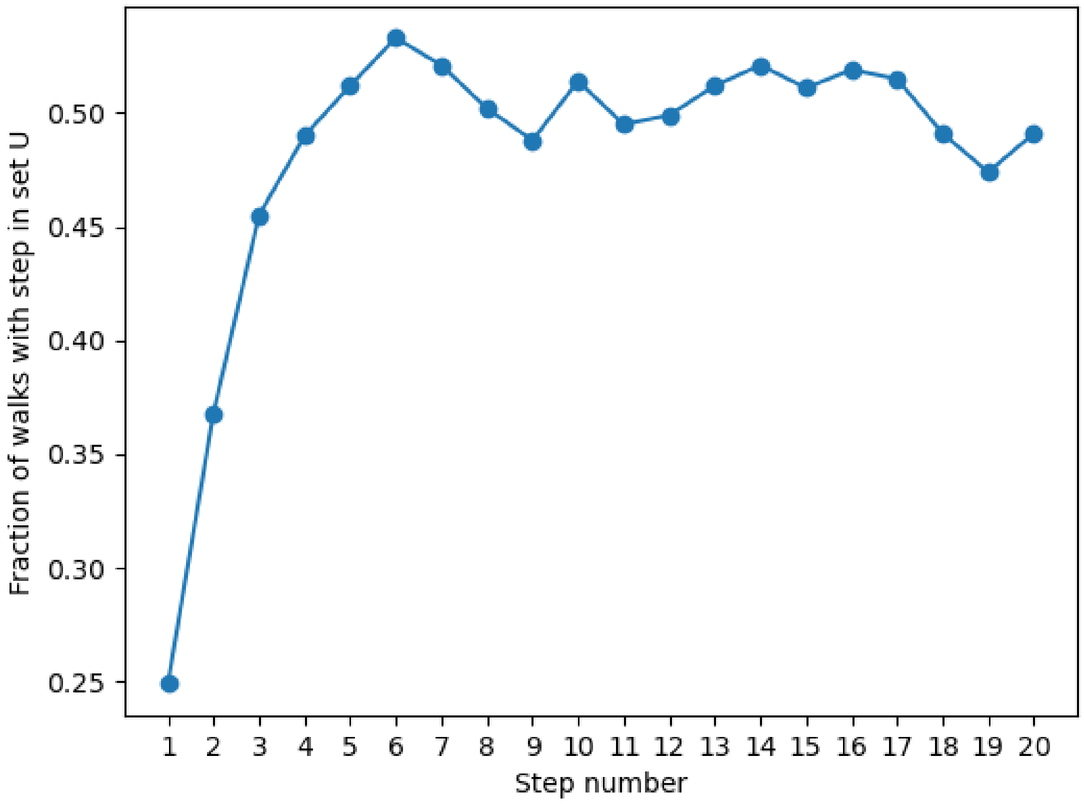
Part b

I find that odd-number steps are in U 25% of the time, while even-number steps are in U 75% of the time. This is to be expected, as in step 1 (the start) we have a 25% chance of starting in U. Then we are switching from U to V each step, as the bipartite nature of the graph dictates. So, which group each step will leave us in is entirely dictated by where we start.

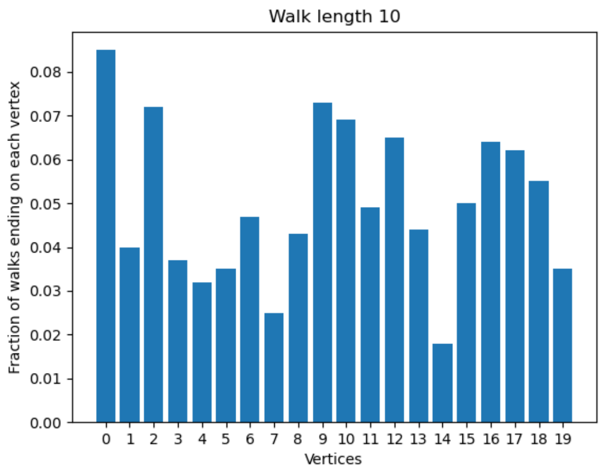


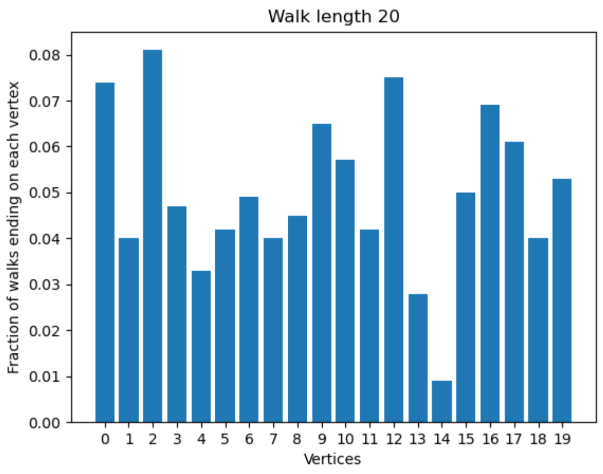
Part c

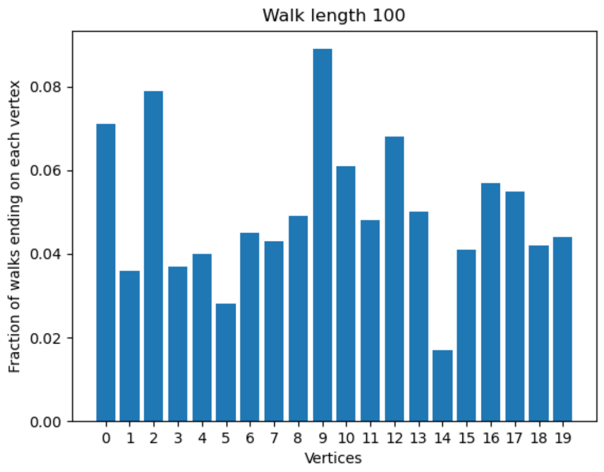
The results of this experiment do indeed differ from the previous one. As previously, the starting step is in U with 25% probability. After that, however, the previously-found dependency on the starting step goes away. This is because regardless of where you currently are (U or V), you have exactly 25% probability of staying and 75% of going to the other side of the bipartite graph. This forces a divergence toward a fraction of walk steps in U of .5 as you move through more and more steps. This is to be expected, as the dependence on the starting step no longer applies due to not being guaranteed to switch sides of the graph on each step, and the procedure for deciding the next node location is the same regardless of present node location.



Part d







All 3 walks of different lengths seem to yield distributions that are very similar, but the distribution itself doesn't seem to be anything specific (it changes for different instances of G). It seems that the longer the walk, the less noisy the distribution becomes, which suggests that a longer walk length brings the distribution closer and closer to the stationary distribution.

Problem 3

Part a

If the maximum degree of any vertex r isn't reasonably small, Metropolis-Hastings (MH) will have a smaller and smaller chance of exploring new nodes (the probability of staying at the current node $\frac{r-\ell}{r}$ converges to 1 as r increases).

Part b

For neighbors u and v, neither $\frac{f(u)}{f(v)}$ nor $\frac{f(v)}{f(u)}$ should be too small. If neighbors have significantly different weights, MH will show a preference toward staying at nodes with high weights. Similar to part a, this will lead to a smaller chance of exploring new nodes.

Part c

The graph should have a good mixing time. If it doesn't, we may not converge to the stationary distribution within the set number of trials.

Code for problem 2:

```
import networkx as nx
import numpy as np
from matplotlib import pyplot as plt

# Part a
G = nx.Graph()

U = range(10)
V = range(10,20)
E = []

G.add_nodes_from(U, bipartite=0)
G.add_nodes_from(V, bipartite=1)

while not nx.is_connected(G):
U_vertex = np.random.randint(0, 10)
V_vertex = np.random.randint(10, 20)
G.add_edge(U_vertex, V_vertex)

# Part b
counter_in_U = {}
for i in range(1, 21):
counter_in_U[i] = 0

for walk in range(1000):
if np.random.rand() < .25:
start = np.random.randint(0, 10)
else:
start = np.random.randint(10, 20)

curr = start
for curr_step in range(1, 21):
if curr in range(0, 10):
counter_in_U[curr_step] += 1
neighbors = [n for n in G.neighbors(curr)]
curr = np.random.choice(neighbors)

plt.plot(range(1, 21), np.array(list(counter_in_U.values()))/1000, 'o-')
plt.xticks(range(1, 21))
plt.xlabel('Step number')
plt.ylabel('Fraction of walks with step in set U')
plt.show()

# Part c
counter_in_U = {}
for i in range(1, 21):
counter_in_U[i] = 0

for walk in range(1000):
if np.random.rand() < .25:
start = np.random.randint(0, 10)
else:
start = np.random.randint(10, 20)

curr = start
for curr_step in range(1, 21):
if curr in range(0, 10):
counter_in_U[curr_step] += 1
if np.random.rand() < .25:
neighbors = [n for n in G.neighbors(curr)]
curr = np.random.choice(neighbors)

plt.plot(range(1, 21), np.array(list(counter_in_U.values()))/1000, 'o-')
plt.xticks(range(1, 21))
plt.xlabel('Step number')
plt.ylabel('Fraction of walks with step in set U')
plt.show()

# Part d
for walk_length in [10, 20, 100]:
counter_final = {}
for i in range(0, 20):
counter_final[i] = 0

for walk in range(1000):
if np.random.rand() < .25:
start = np.random.randint(0, 10)
else:
start = np.random.randint(10, 20)

curr = start
for curr_step in range(1, walk_length+1):
if np.random.rand() < .25:
```

```
neighbors = [n for n in G.neighbors(curr)]
curr = np.random.choice(neighbors)

counter_final[curr] += 1

plt.bar(range(20), np.array(list(counter_final.values()))/1000, align='center')
plt.xticks(range(20))
plt.title(f'Walk length {walk_length}')
plt.xlabel('Vertices')
plt.ylabel('Fraction of walks ending on each vertex')
plt.show()
```