HOMEWORK-8

Resources - Lecture notes,Wikipedia, https://networkx.github.io/documentation/networkx-1.10/index.html

Collaboration- Nischal and Abhilash

Problem 1:

Part a:

- $A^3_G = A_G * A_G * A_G$ gives us number of paths from a vertex i to j of exactly path length 3. To form a triangle we want to return back to the original vertex we started from after 3 hops.
- $A_G(i,i)$ gives the number of paths from vertex i to vertex i after taking 3 hops starting from i.

Algorithm:

- Multiply adjacency matrix A_G with itself for 3 times giving A^3_G
- We're just interested in the numbers on the main diagonal matrix because our aim is to calculate the number of paths where each path starts on a vertex i and ends on vertex i after 3 hops. If we reach same vertex after 3 hops we've one triangle(path).
- Let number_triangles be the number of triangles.

```
For i in range(0,num_vertices):
        For j in range(0,num_vertices):
            If i==j:
                number_traingles+=A^3_G [i][j]
```

Number_distinct_triangles = number_triangles

Part b:

- We repeat the same algorithm from part a but in the end we divide number_triangles by 6.
- Reason being (a,b,c),(b,c,a) and (c,b,a) are the same triangles and for each triangle we two paths starting and ending at the same vertex so 3*2=6
- Result for part b = number_triangles/6

Problem 2

- The algorithm is similar to binary search but with slight modification.
- When $A_{ij}^{mid} = 0$ we need to decide whether we want to move to the right checking if i--->j path exists increasing the number of steps or to the left to check if we can reach j in fewer steps.
- Crux: if A_{ij}^{mid} and A_{ij}^{mid-1} both are 0 then we move to the right. Because the path lengths mid and mid-1 are too short to even reach j at least once.
- Meaning : We check A_{ij}^{mid-1} because it might have happened that we have reached j in mid-1 steps but after taking one more step we moved one step away from j. Leading to $A_{ij}^{mid} = 0$ and $A_{ij}^{mid-1} > 0$.

Algorithm:

- Left = 0 , right = n -1, shortest_path = INT_MAX
- while(left<=right):

    mid = (left+right)/2

    if $A_{ij}^{mid} > 0$ or $A_{ij}^{mid-1} > 0$:

      Shortest_path = min(shortest_path, A_{ij}^{mid} , A_{ij}^{mid-1})

      right = mid - 2

    if $A_{ij}^{mid} == 0$ and $A_{ij}^{mid-1} == 0$:

      left = mid+1

  return shortest_path

Problem 3:

Part a

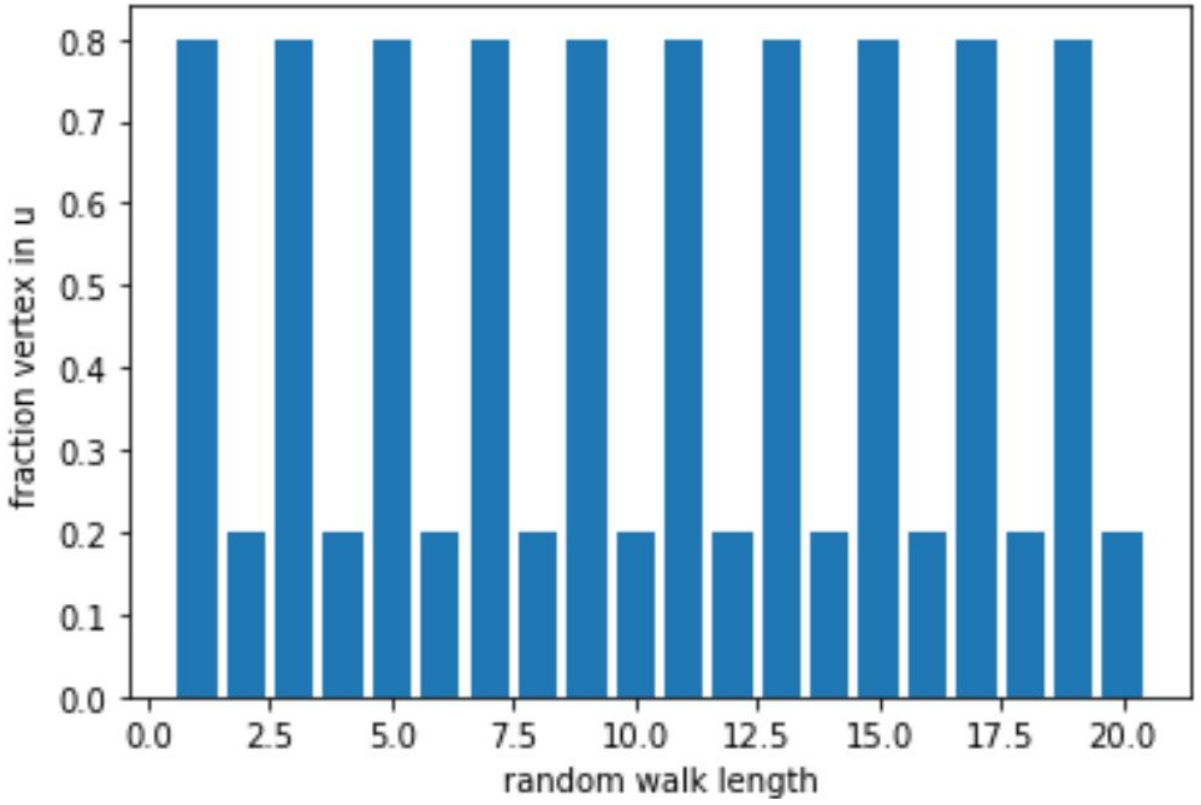
- Using Python NetworX library to build a complete binary bipartite graph.
- The graph is connected G(E,U,V). Call to **complete_bipartite_graph** takes two arguments which number of vertices in U and V and returns a complete and connected graph.

```
import networkx as nx
import random
from networkx.algorithms import bipartite
import matplotlib.pylab as plt
%matplotlib inline
G = nx.complete_bipartite_graph(30, 50)
u, v = nx.bipartite.sets(G)
u = list(u)
v = list(v)
```

Part b

- We observe that at any t=1,2,3..20 at step t_i if fraction of vertex being in U is high then in the next step t_{i+1} we observe that fraction of vertex being in V is high.
- We observe this pattern because of the nature of graph which is bipartite and at step t_i if vertex is in U we choose a random vertex from V for step t_{i+1} thereby increasing the fraction of vertex being V for step t_{i+1}

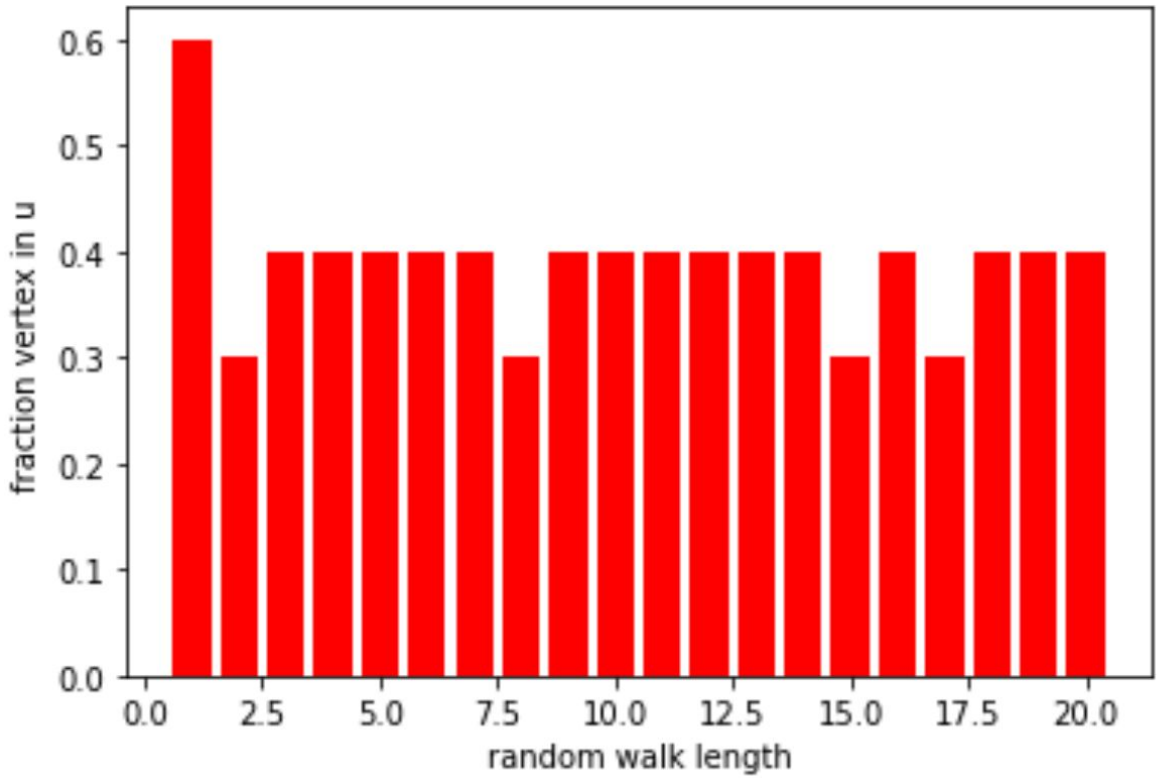
```
du = {}
dv = {}
for i in range(1,1001):
    startVertex = 0
    if random.choice([1,2,3,4])==1:
        startVertex = random.choice(list(u))
    else:
        startVertex = random.choice(list(v))
    for t in range(1,21):
        if startVertex in u:
            startVertex = random.choice(list(v))
            dv[t] = dv.get(t,0)+1
        else:
            startVertex = random.choice(list(u))
            du[t] = du.get(t,0)+1
x = list(range(1,21))
y=[]
for i in range(1,21):
    y.append(round(du.get(i,0)/1000,1))
plt.bar(x,y,align='center')
plt.show()
```


fraction vertex in u
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
random walk length

Part c

- The graph changes in this case because in previous case we moved to neighbouring vertices with probability 1 i.e every time but in this case we step to the neighbouring vertices with probability ¾ thereby increasing the probability of staying at the current vertex for next iteration from 0 to ¼ . Thereby increasing the fraction of vertex being in current set(U/V).

```
du = {}
dv = {}
for i in range(1,1001):
    startVertex = 0
    if random.choice([1,2,3,4])==1:
        startVertex = random.choice(list(u))
    else:
        startVertex = random.choice(list(v))
    for t in range(1,21):
        if startVertex in u:
            if random.choice([1,2,3,4])!=1:
                startVertex = random.choice(list(v))
                dv[t] = dv.get(t,0)+1
        else:
            if random.choice([1,2,3,4])!=1:
                startVertex = random.choice(list(u))
                du[t] = du.get(t,0)+1
x = list(range(1,21))
y=[]
for i in range(1,21):
    y.append(round(du.get(i,0)/1000,1))
plt.bar(x,y,align='center',color='r')
plt.xlabel('random walk length')
plt.ylabel("fraction vertex in u")
plt.show()
```


0.6
0.5
0.4
0.3
0.2
0.1
0.0
fraction vertex in u
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
random walk length

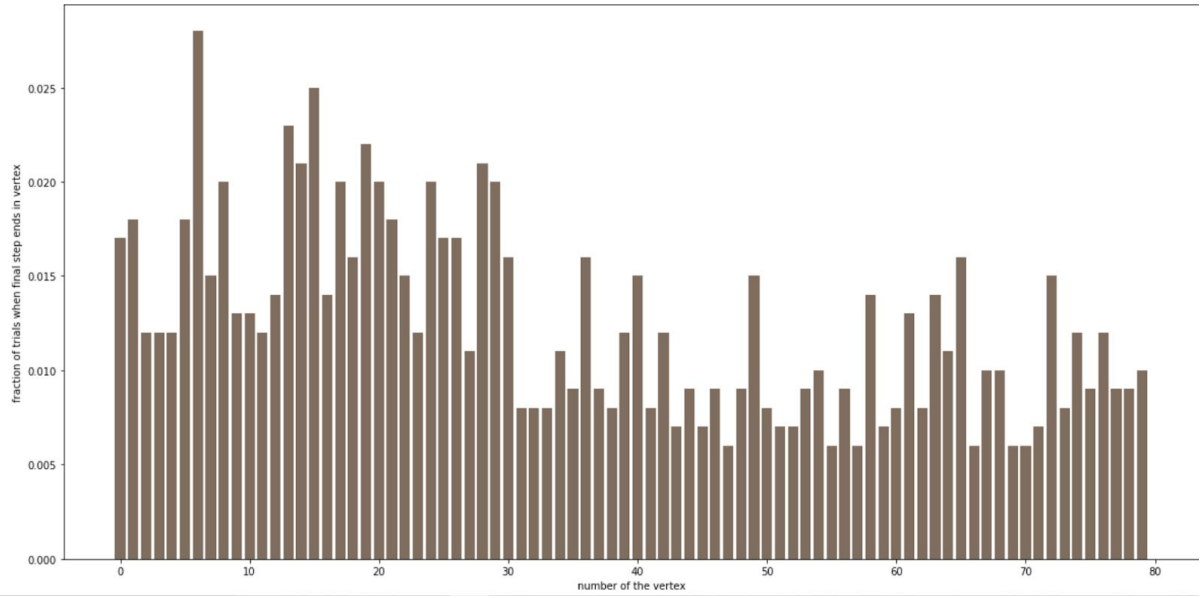
Part d

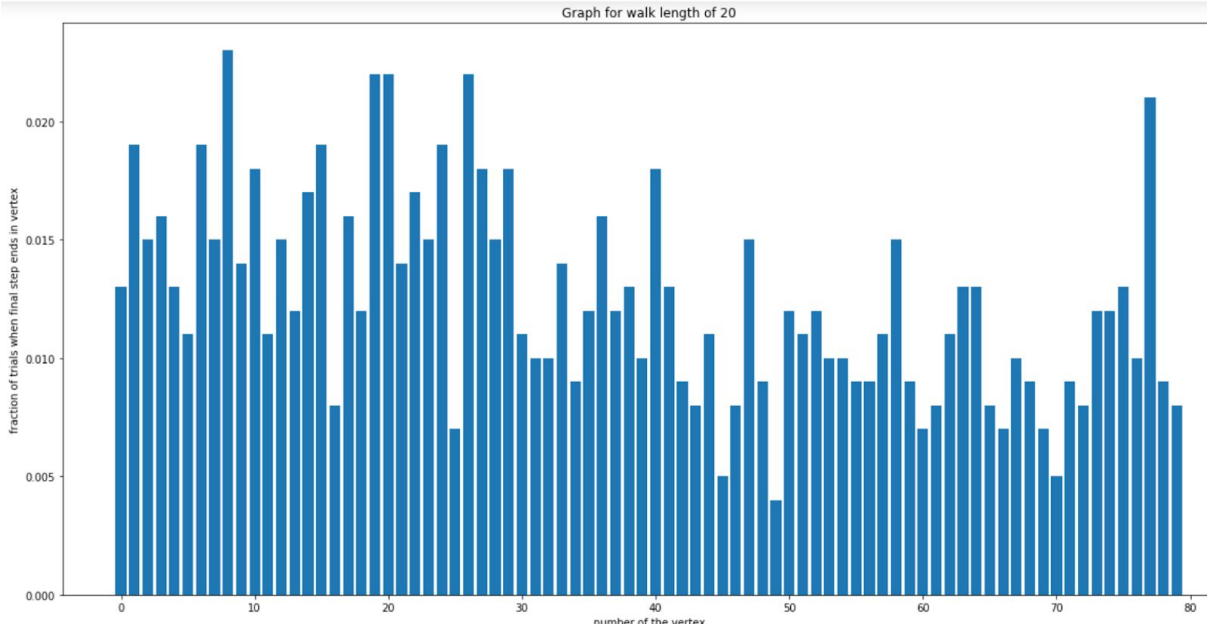
- Distribution of random walk ending in a particular vertex does not change(approximately remains the same) as we change the number of steps from 10,20 to 100 because it is no longer dependent on the number of steps nor on the probability distribution that we specified in part c i.e as t reaches infinity the probability of being in state u_t no longer depends on posterior probability that u_{t-k} state was visited before which means states are independent.

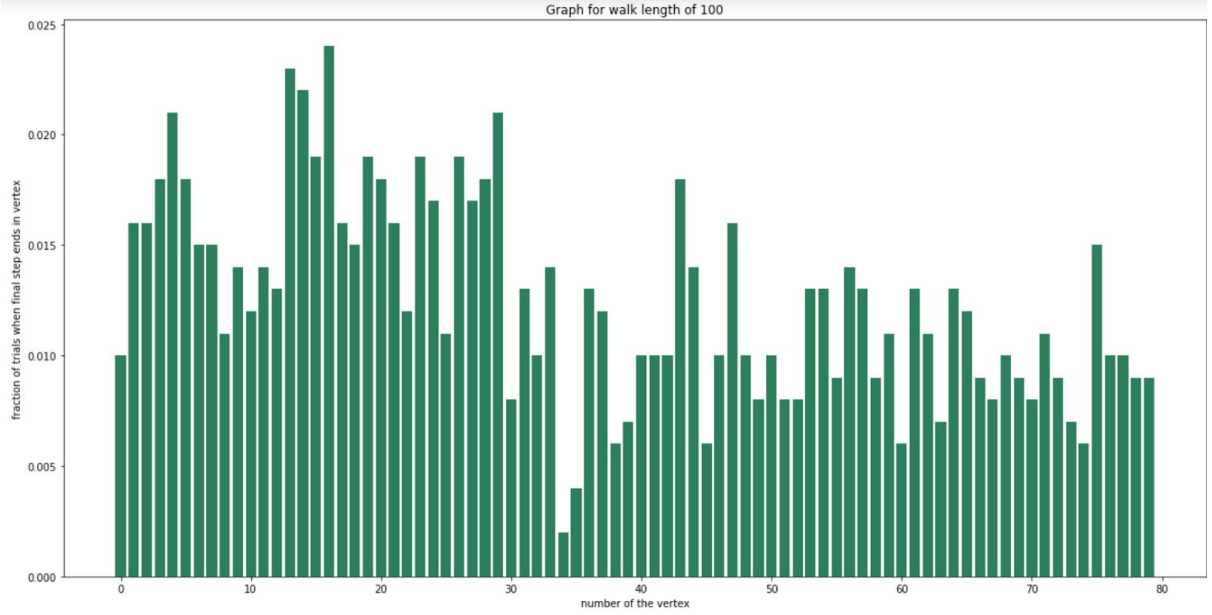
```
d10 = {}
d100 ={}
for i in range(1,1001):
    startVertex = 0
    if random.choice([1,2,3,4])==1:
        startVertex = random.choice(list(u))
    else:
        startVertex = random.choice(list(v))
    for t in range(1,21):
        if startVertex in u:
            if random.choice([1,2,3,4])!=1:
                startVertex = random.choice(list(v))
                dv[t] = dv.get(t,0)+1
        else:
            if random.choice([1,2,3,4])!=1:
                startVertex = random.choice(list(u))
                du[t] = du.get(t,0)+1
        if t==20:
            d20[startVertex] = d20.get(startVertex,0)+1
for i in range(1,1001):
    startVertex = 0
    if random.choice([1,2,3,4])==1:
        startVertex = random.choice(list(u))
    else:
        startVertex = random.choice(list(v))
    for t in range(1,11):
        if startVertex in u:
            if random.choice([1,2,3,4])!=1:
                startVertex = random.choice(list(v))
                dv[t] = dv.get(t,0)+1
        else:
            if random.choice([1,2,3,4])!=1:
                startVertex = random.choice(list(u))
                du[t] = du.get(t,0)+1
        if t==10:
            d10[startVertex] = d10.get(startVertex,0)+1
```

```
for i in range(1,1001):
    startVertex = 0
    if random.choice([1,2,3,4])==1:
        startVertex = random.choice(list(u))
    else:
        startVertex = random.choice(list(v))
    for t in range(1,101):
        if startVertex in u:
            if random.choice([1,2,3,4])!=1:
                startVertex = random.choice(list(v))
                dv[t] = dv.get(t,0)+1
        else:
            if random.choice([1,2,3,4])!=1:
                startVertex = random.choice(list(u))
                du[t] = du.get(t,0)+1
        if t==100:
            d100[startVertex] = d100.get(startVertex,0)+1
x1 = [i for i in range(0,80)]
y1 = []
y2 = []
y3 = []

for i in range(0,80):
     y2.append(d10.get(i,0)/1000)
plt.figure(figsize=(20,10))
plt.bar(x1,y2,align='center',color="#7f6d5f")
plt.title("Graph for walk length of 10")
plt.xlabel("number of the vertex")
plt.ylabel("fraction of trials when final step ends in vertex")
plt.show()
```

```
for i in x1:
    y1.append(d20.get(i,0)/1000)
plt.figure(figsize=(20,10))
plt.bar(x1,y1,align='center')
plt.title("Graph for walk length of 20")
plt.xlabel("number of the vertex")
plt.ylabel("fraction of trials when final step ends in vertex")
plt.show()

for i in range(0,80):
     y3.append(d100.get(i,0)/1000)
plt.figure(figsize=(20,10))
plt.bar(x1,y3,align="center",color="#2d7f5e")
plt.title("Graph for walk length of 100")
plt.xlabel("number of the vertex")
plt.ylabel("fraction of trials when final step ends in vertex")
plt.show()
```




Graph for walk length of 20
fraction of trials when final step ends in vertex
0.000
0.005
0.010
0.015
0.020
0
10
20
30
40
50
60
70
80


Graph for walk length of 100
fraction of trials when final step ends in vertex
0.000
0.005
0.010
0.015
0.020
0.025
0
10
20
30
40
50
60
70
80
number of the vertex

Problem 4:

Part a:

- The maximum degree r of any vertex must be small because we transition to neighboring states with probability 1/r.
- If r were to be too large then we would end up staying at current states for quite a long time before transitioning to the next state.This increases the time to converge and reach closer to the stationary distribution.
- If r is too large then time to calculate estimation $\pi(u)$ will take more time because in $f(u)/\Sigma f(v)$ denominator calculation will take long.

Part b

- If f(u) and f(v) weights are too different then the ratio f(u)/f(v) or f(v)/f(u) is too small.
- We want to move to neighbouring states that are similar to the current state so that we end up sampling less number of states and reach convergence faster.
- If ratio is small we take very small steps and convergence takes more time as we end up staying in current state with higher probability.
- If we end up moving to vertex v, it might lead to a lower density probability region and we end up staying there for a long time before we return to approximating stationary distribution on states we're interested in.

Part c

- We want the distribution to be independent of the random walk path length. To reach converge we might have to go through a number of trials but we want it to be as small as possible.
- A good mixing time means we can get to approximately estimating π and a converging distribution as fast as possible.
- If the distribution doesn't converge even after some threshold say 10000 iterations we can be pretty much sure that for the subset of states on which we have constructed a graph have no stationary distribution, and there is no point in increasing the iterations further because the distribution will keep swinging.

Problem 5:

Part a

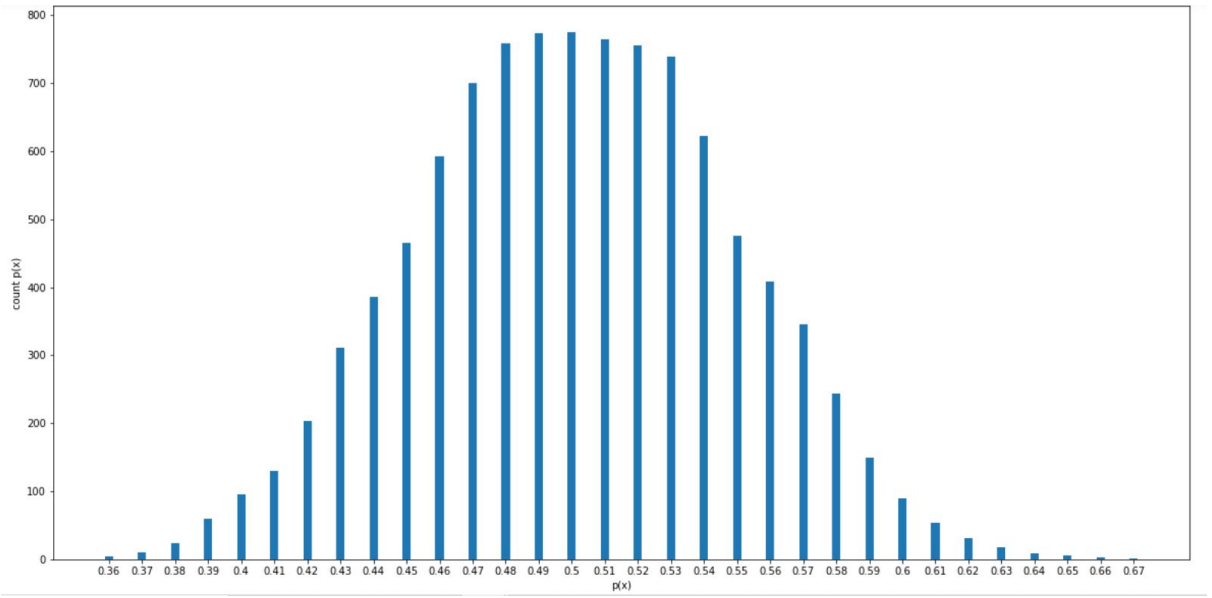
- If there are n particles then we represent each state as n-bits where change in even one bit changes the state. Where each particle is a bit with value in $\{0,1\}$.
- If we consider the above explanation then there are 2^n states. Maximum degree is n. We can compute neighbours by flipping one bit at a time.

Part b

```
import numpy as np
import math
d = {}
def return_set_bits(state):
    return state.count(1)
n = 100
u = []
for i in range(0,n):
    u.append(random.choice([0,1]))
trials = 10000
for i in range(trials):
    neighbours = []
    for j in range(0,n):
        neighbour = u[:]
        if u[j]==1:
            neighbour[j] = 0
        else:
            neighbour[j] = 1
        neighbours.append(neighbour)
    v = random.choice(neighbours)
    p_u = return_set_bits(u)/n
    p_v = return_set_bits(v)/n

    f_u = p_u*math.log(1/p_u,2) + (1-p_u)*math.log(1/(1-p_u),2)
    f_v = p_v*math.log(1/p_v,2) + (1-p_v)*math.log(1/(1-p_v),2)

    if f_v>=f_u:
        u = v[:]
    elif np.random.uniform()<=(f_u/f_v):
        u = v[:]
    d[p_u] = d.get(p_u,0)+1
```

```
x = list(d.keys())
x.sort()
y = []
for i in x:
    y.append(d[i])
for i in range(0,len(x)):
    x[i] = str(x[i])
plt.figure(figsize=(20,10))
plt.bar(x,y,width=0.25)
plt.ylabel("count p(x)")
plt.xlabel("p(x)")
plt.show()
```

Part c


count p(x)
p(x)
0
100
200
300
400
500
600
700
800
0.36 0.37 0.38 0.39 0.4 0.41 0.42 0.43 0.44 0.45 0.46 0.47 0.48 0.49 0.5 0.51 0.52 0.53 0.54 0.55 0.56 0.57 0.58 0.59 0.6 0.61 0.62 0.63 0.64 0.65 0.66 0.67