The University of Texas at Dallas

PROJECT 3

DEPENDENCE OF NETWORK RELIABILITIES ON INDIVIDUAL LINK RELIABILITIES

CS 6385

Debaspreet Chowdhury **UTD ID - 2021211268**

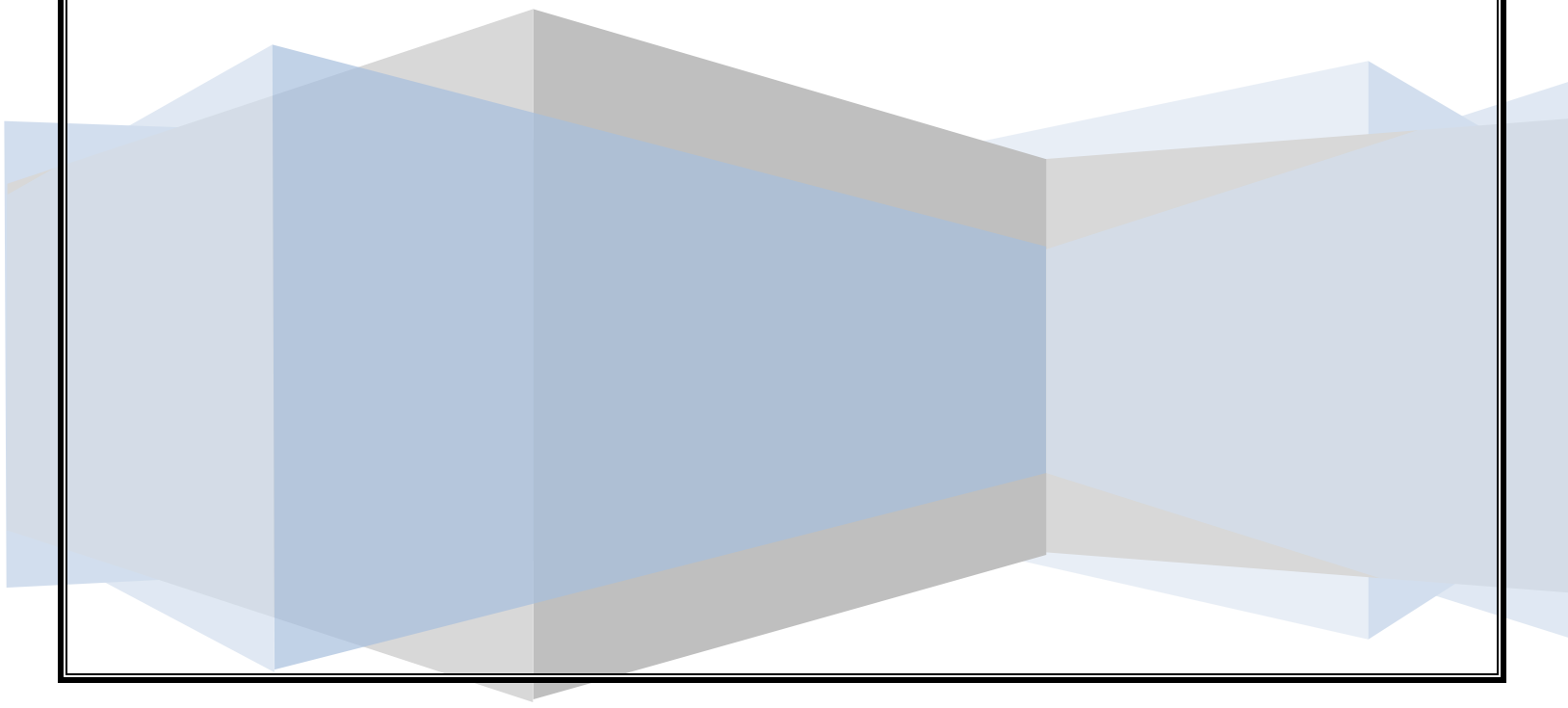


TABLE OF CONTENTS

PROJECT DESCRIPTION

I chose to implement the project on Python and use NetworkX package for it. NetworkX offers a functionality rich environment to implement graph algorithms. Most of the basic operations in a graph, like finding the number of nodes, edges, shortest path from source to destination etc have already been implemented by built in functions. The coder is just required to import these functionalities into his code and build his algorithm. That doesn't mean that the built packages cannot be changed. On the contrary, NetworkX offers a high degree of flexibility in terms of binding the built in functions in our code.

The built in functions that I have directly used in my code by importing the networkx package and other built in package in python without any changes are –

1) **dense_gnm_random_graph(n,m)**

   This function creates a random undirected graph with 'n' nodes and 'm' edges. The selection of edges is random and done by the function itself. The source code for the same can be found at
   https://networkx.github.io/documentation/latest/_modules/networkx/generators/random_graphs.html#dense_gnm_random_graph.

2) **add_edge(i,j)**

   This function adds an edge between two nodes 'i' and 'j'. The source code can be found at
   https://networkx.github.io/documentation/latest/_modules/networkx/classes/graph.html#Graph.add_edge

3) **remove_edge(i,j)**

   This function removes an edge between two nodes 'i' and 'j'. The source code is at
   https://networkx.github.io/documentation/latest/_modules/networkx/classes/graph.html#Graph.add_edge

   Apart from this, all the other functions and their source codes can be found at:
   https://networkx.github.io/documentation/latest/index.html

4) **G.edges()**

   This returns a tuple of edges present is graph 'G'. The source code can be found at-
   https://networkx.github.io/documentation/latest/_modules/networkx/classes/graph.html#Graph.edges

5) **combinations(Edges,i)**

   This returns tuples/subsets of all possible combinations of Edges taken 'i' at a time. More can be found at this page:
   https://docs.python.org/2/library/itertools.html#itertools.combinations

6) **all_simple_paths(G,S,T)**

Returns a tuple of all the simple paths from Source 'S' to target 'T' in graph 'G'. Each of these path do not visit each node more than once. More code can be found on this at : https://networkx.github.io/documentation/latest/_modules/networkx/algorithms/simple_paths.html#all_simple_paths

7) **random.sample(range(N),m)**

Returns a tuple of 'm' numbers whose range is from 0 to N. More can be found at : https://docs.python.org/2/library/random.html

The algorithms/functions I have written myself are the following. The source code and working of the same have been described in Task1, Task2, Task3 and Task4 –

1) **reliabilityonlinks()**

This calculates the reliability based on the number of links brought down in the graph and eventally prints the reliability value corresponding to the number of links pulled down.

2) **probability_reliability()**

This function calculates the reliability of the entire network for each value of 'p' ie the individual link reliability. The reliability is calculated by removing each link or a combination of them and then checking the connectivity of the source to the target node.

3) **reliability(num=[[]])**

This function calculated the overall reliability of the network based on removal of one link or a combination of links and then checking the source to target connectivity. It takes input argument 'num' which is a list of elements containing the reliability and the corresponding state of the network.

4) **main()**

This function basically calculates the network reliability before and after the state of 'K' links are flipped. This function makes use of the reliability (num) function to calculate network reliability.

NOTE:

1) **Throughout the documentation, comments have been mentioned with a hashtag in red color in the program itself. Example-**

2) *#nagamochi algorithm using maximum adjacency ordering and graph contraction*
3) *#takes graph 'G' as input parameter*
4) *#repeat the above process till K==0 upon which exit the inner while loop.*

HOW TO RUN THE CODE

The source codes have been written in Python on a windows 7 system. The following steps need to be taken to run the code in windows. For Linux/MAC system, please visit:

For Python: https://www.python.org/downloads/release/python-278/

For NetworkX: http://networkx.github.io/documentation/latest/install.html

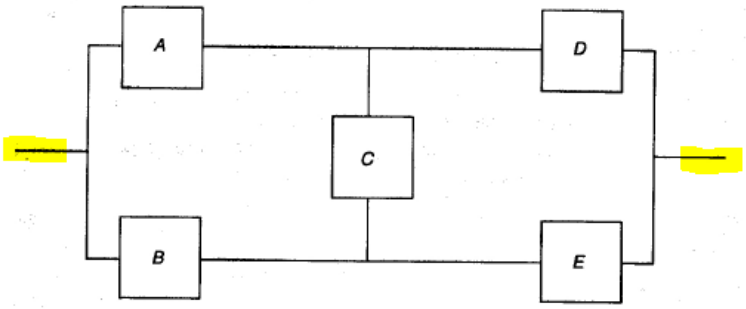
For Gephi: https://gephi.github.io/users/download/

1) Install Python 2.7 in system from here https://www.python.org/downloads/release/python-278/.
2) After installation is complete, open up the command prompt and go to the directory where Python is installed. Type 'Python' and press return. If the editor opens, then Python is installed else repeat the steps above to install Python.
3) Be sure to install the correct version of Python on the system ie. x86 or x64 installer.
4) Now we need to install a library called NetworkX in the system which is imported into my Python programs to do graph computations. To install NetworkX, follow these instructions. https://pypi.python.org/pypi/setuptools#installation-instructions
5) To very if NetworkX is installed, from step 2, after the editor is entered into, type 'import NetworkX as nx'. If there are no errors, Network is successfully installed.
6) exit() at the editor prompt exits the editor.
7) I have included the name of the python file(ending with .py) by which the code needs to be saved in a directory to be later ran from the command prompt in windows. I've given this name towards the beginning of the program.
8) To run the python program, simply type 'python abcd.py' at the command prompt after entering the Python installation directory. Abcd.py is the name of the python file from step 8)
9) I also use Gephi to display the graphs my programs generate. Gephi can be found at - https://gephi.github.io/users/download/
10) Because my programs generate a .gexf file, I use Gephi to open a .gexf file. Open Gephi and then open the .gexf file the Python program generated.

TASK 1

Method of Exhaustive Enumeration

Let’s assume a network with 5 nodes and connections between then as shown below :



We need to test the network reliability between the highlighted yellow terminals. We list all possible states of the system and assign "up" and "down" system condition to each state based on the fact that the two yellow highlighted terminals remain connected(for 'up') and disconnected(for 'down'). Then the reliability can be obtained by summing the probability of the ""up" states. Even though the obtained expression may be simplified, this method is only practical for small systems, due to the exponential growth of the number of states. For N components there are 2^N possible states, making exhaustive enumeration a non-scalable solution.

Overall network reliability based on ‘up’ state:

$$
\begin{aligned}
R_{\text{network}} = {} & R_A R_B R_C R_D R_E + (1-R_A) R_B R_C R_D R_E \\
& + R_A (1-R_B) R_C R_D R_E + R_A R_B (1-R_C) R_D R_E \\
& + R_A R_B R_C (1-R_D) R_E + R_A R_B R_C R_D (1-R_E) \\
& + (1-R_A) R_B (1-R_C) R_D R_E + (1-R_A) R_B R_C (1-R_D) R_E \\
& + (1-R_A) R_B R_C R_D (1-R_E) + R_A (1-R_B)(1-R_C) R_D R_E \\
& + R_A (1-R_B) R_C (1-R_D) R_E + R_A (1-R_B) R_C R_D (1-R_E) \\
& + R_A R_B (1-R_C)(1-R_D) R_E + R_A R_B (1-R_C) R_D (1-R_E) \\
& + R_A (1-R_B)(1-R_C) R_D (1-R_E) \\
& + (1-R_A) R_B (1-R_C)(1-R_D) R_E
\end{aligned}
$$

All the possible states exhaustively enumerated:

Number of Component Failures	Event	System Condition
0	1. $ABCDE$	Up
1	2. $\overline{A}BCDE$	Up
	3. $A\overline{B}CDE$	Up
	4. $AB\overline{C}DE$	Up
	5. $ABC\overline{D}E$	Up
	6. $ABCD\overline{E}$	Up
2	7. $\overline{A}\overline{B}CDE$	Down
	8. $\overline{A}B\overline{C}DE$	Up
	9. $\overline{A}BC\overline{D}E$	Up
	10. $\overline{A}BCD\overline{E}$	Up
	11. $A\overline{B}\overline{C}DE$	Up
	12. $A\overline{B}C\overline{D}E$	Up
	13. $A\overline{B}CD\overline{E}$	Up
	14. $AB\overline{C}\overline{D}E$	Up
	15. $AB\overline{C}D\overline{E}$	Up
	16. $ABC\overline{D}\overline{E}$	Down
3	17. $AB\overline{C}\overline{D}E$	Down
	18. $A\overline{B}C\overline{D}\overline{E}$	Down
	19. $A\overline{B}\overline{C}D\overline{E}$	Up
	20. $A\overline{B}\overline{C}\overline{D}E$	Down
	21. $\overline{A}BC\overline{D}\overline{E}$	Down
	22. $\overline{A}B\overline{C}D\overline{E}$	Down
	23. $\overline{A}B\overline{C}\overline{D}E$	Up
	24. $\overline{A}\overline{B}CD\overline{E}$	Down
	25. $\overline{A}\overline{B}C\overline{D}E$	Down
	26. $\overline{A}B\overline{C}\overline{D}\overline{E}$	Down
4	27. $A\overline{B}\overline{C}\overline{D}\overline{E}$	Down
	28. $\overline{A}B\overline{C}\overline{D}\overline{E}$	Down
	29. $\overline{A}\overline{B}C\overline{D}\overline{E}$	Down
	30. $\overline{A}\overline{B}\overline{D}D\overline{E}$	Down
	31. $\overline{A}\overline{B}C\overline{D}E$	Down
5	32. $\overline{A}\overline{B}\overline{C}\overline{D}\overline{E}$	Down

Brief Description of my Algorithm

My algorithm works on any undirected graph without any parallel edges and self loops. ‘dense_gnm_random_graph(5,10)’ creates a graph ‘G’ with 5 nodes and 10 edges. After this graph is created, we bring down ‘k’ combinations of links between node 0 and 3 which are our ‘S’(source) and ‘T’(target) nodes respectively and where k=0,1,2,3…..9,10. So this results in 2^10=1024 total network states : each state having a certain number of links down. Even if the number of links down may be the same between the network states, what will vary in that case is ‘WHICH’ links are down. After we bring down the ‘kth’ combination of link down, I apply the function all_simple_paths(G,S,T). ‘all_simple_paths(G,S,T)’ is a built in NetworkX function that counts all paths from source node ‘S’ to target nodes ‘T’ without visiting any node twice on a single path. I make use of this function to check if my source and target nodes are connected despite of bringing any number of links between them down. If the check comes up positive, then I include this state reliability value into the overall network reliability and keep updating this sum iteratively over all ‘k’ combinations of link down states.

Pseudo code and other general description

1) Import the following packages into the program – itertools, combinations, network, random.
2) **dense_gnm_random_graph(5,10)** crates an undirected graph 'G' with 5 nodes and 10 edges, no parallel edges and self loops.
3) After the graph is created, we need the all the possible existent edges in the graph. **G.edges()** creates a tuple of all such edges and stores it in **total_Edges**.
4) Select a link probability 'p' value between (0,1], set reliability=0 initially
5) Take k=1 random link and remove it from graph using **remove_Edge()**
6) Now check the connectivity of node 0(S) with node 2(T) upon removal of that link. **len(list(nx.all_simple_paths(G,0,2))** returns the number of simple paths connecting S with T. If this count is 0, means that S and T are disconnected.
7) If connectivity exists, we find – **reliability=reliability+(pow(1-probability,count))*(pow(probability,(10-count)))**
8) We restore the link/s which we brought down using **add_Edge()** function.
9) We repeat steps 5) to 8) for all other possible k=1 links in the graph 'G'.
10) Now we take k=2. That is we bring down 2 links at a time in the graph 'G' and repeat step 5) through 8).
11) We do like this for all tuples given by **combinations(total_Edges,k)**.
12) We execute steps 5) through 8) for k=0,1,2,3…,9,10
13) Upon going through step 12), we end up having 1024 network state combinations. Few of them have connectivity, few of them don't. We calculate the reliability only for the states that have connectivity.
14) According to task 3, we have to take several values of link probability and plot them against the calculate reliability. So for this we repeat the above steps from 4) to 13).

TASK 2

Copy paste the following code from the first line of comment in the 'Task2.py' named file and run it from command prompt in windows using python interpretor. Example: python task2.py

reliabilityonlinks():

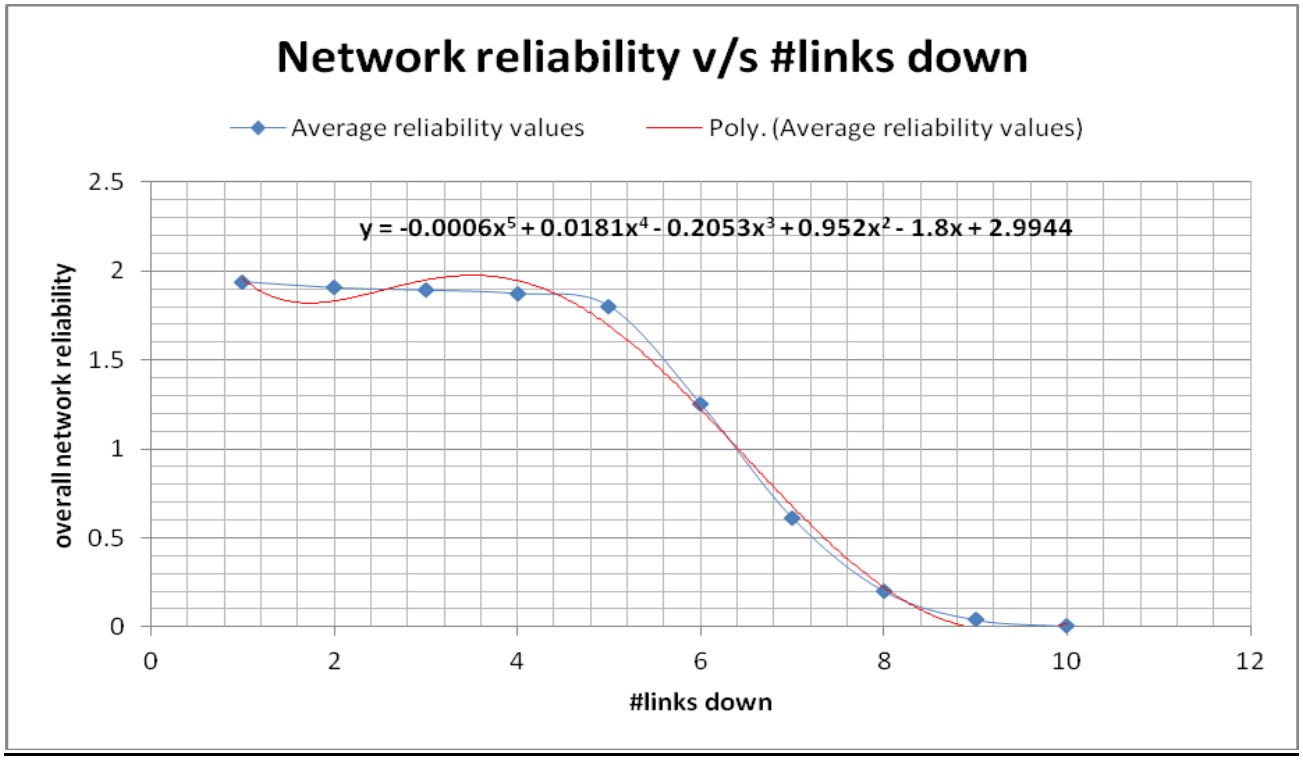
For each selection of 'k' links, we bring all the links down and calculate the reliability and print the reliability value corresponding to the number of links brought down.

```
 #define a function reliabilityonlinks which is not taking any arguments.
def reliabilityonlinks():
 from itertools import combinations                          #import itertools library
 import networkx as nx                                       #import network library
 import random                                               #import random library
 #generate graph with 5 nodes and 10 edges
 G=nx.dense_gnm_random_graph(5,10)
 #returns a tuple of all edges in graph G
 total_Edges=G.edges()
 probability=0.8                                             #set probability 0.8
 for i in xrange(0,11):                                      #for 'i' varying from 0 to 10
 #set reliability=0 initially whenever the previous loop starts
 reliability=0
 #for all possible edge combinations taken 'i' at a time from total_Edges
  for subset in combinations(total_Edges,i):
    count=0                                                  #set count=0
 #for j varying in the each of the above mentioned combinations of edge
    for j in subset:
 #if  the above combination is not empty, ie if there are edges
      if len(subset)!=0:
 #increment count=>count the edges of that particular subset.
        count=count+1
      G.remove_edge(j[0],j[1])                               #remove that particular edge
 #if there are path connecting source with target node
    if (len(list(nx.all_simple_paths(G,0,2))) > 0):
 #if the above is true then include the reliability value in the sum of the overall network reliability
      reliability+=(pow(1-probability,count))*(pow(probability,(10-count)))
 #again for each edge in the above subset
    for k in subset:
 #add the edge that was removed in the previous step.
      G.add_edge(k[0],k[1])
    print reliability/float(i),i            #print reliability and corresponding number of links brought down.
if __name__=='__main__':                    #if this block of the code is 'main', then execute the following.
  reliabilityonlinks()                                       #this will execute the function described above.
```

Table showing the overall network reliability corresponding to the number of down links in the network:

#links down	overall network reliability
1	1.938435456
2	1.90497472
3	1.890928614
4	1.871498445
5	1.801285796
6	1.250837572
7	0.61185065
8	0.200250982
9	0.039391004
10	0.003523215

Graph based on the above table showing how reliability degrades as we bring down more and more number of healthy links in the network:

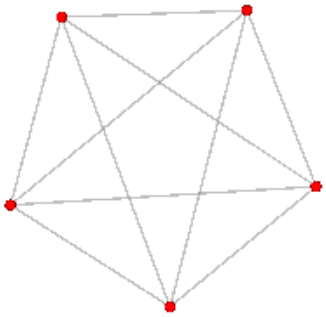


NOTE:

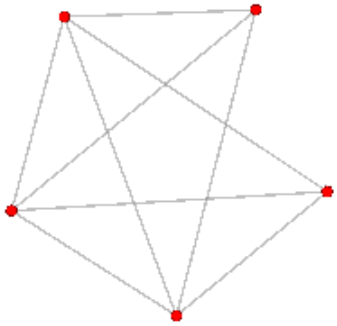
1) **As we bring more and more number or links down, the network becomes more unreliable.**
2) **As we bring more links down, we lose connectivity of 'S' node with 'T' node.**
3) **As evident from the graph, the degradation in reliability of the network is more in the region 5 to 8. This means when the #links down is less, we have more flexibility in the network and thus reliability doesn't degrade much.**

GRAPHS (Plotted using Gephi)

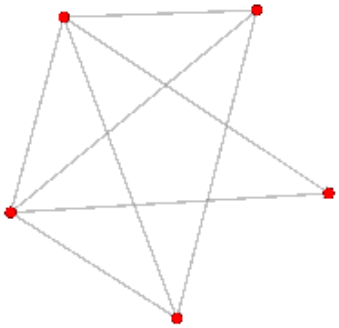
#links down = L reliability = R



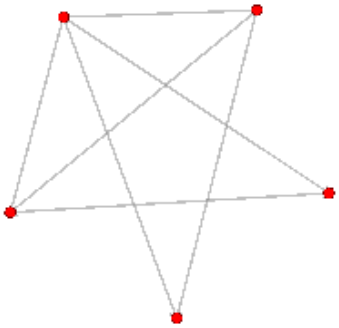
L=0 , R= 1.938435456



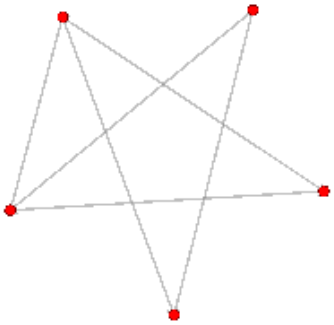
L=1 , R= 1.938435456



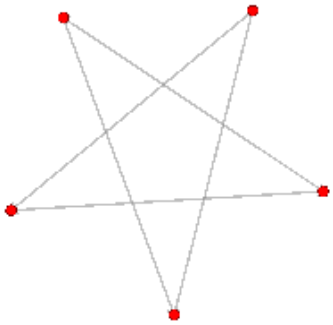
L=2 , R= 1.90497472



L=3 , R= 1.890928614



L=4 , R= 1.871498445



L=5 , R= 1.801285796

NOTE:

We see as the network becomes more disconnected, reliability decreases too.

TASK 3

Copy paste the following code from the first line of comment in the 'Task3.py' named file and run it from command prompt in windows using python interpreter. Example: python task3.py

probability_reliability():

```
def probability_reliability():                                  #defining the function
  from itertools import combinations                            #import itertools
  import networkx as nx                                         #import networkx
  import random                                                 #import random
  G=nx.dense_gnm_random_graph(5,10)                             #generate graph G
  total_Edges=G.edges()                    #extract all edges of graph G in a tuple total_Edges
  probability=0                                  #start with individual link probability p =0
  while probability<1.02:                                       #repeat the following till p till p=1
    reliability=0                                               #set initial reliability =0
    for i in xrange(0,11):                  #for a combination of 'I'edges from  total_Edges
      for subset in combinations(total_Edges,i):                #for each such combination of the above
      count=0                                                   #set initial count to 0
        for j in subset:                                        #for each element in subset
          if len(subset)!=0:                                    #if length of subset is not 0
            count=count+1                                       #increment count by 1
          G.remove_edge(j[0],j[1])                  # and then remove the edge from the graph
#calculate f the S andT nodes still connected
        if (len(list(nx.all_simple_paths(G,0,2))) > 0):
#if S and T connected, include this configuration in network reliability calculation
          reliability+=(pow(1-probability,count))*(pow(probability,(10-count)))
       for k in subset:                          #for each k in the same subset as before for this loop
         G.add_edge(k[0],k[1])                  #revert back the deleted edges from the previous steps
      print probability,reliability                             #print the reliability and probability values
#increment probability value by 0.02 and repeat the loop
      probability+=0.02

#if this block of the code is 'main', then execute the following
if __name__=='__main__':
#this will execute the function described above.
   probability_reliability()
```

Table showing network reliability corresponding to link probability:

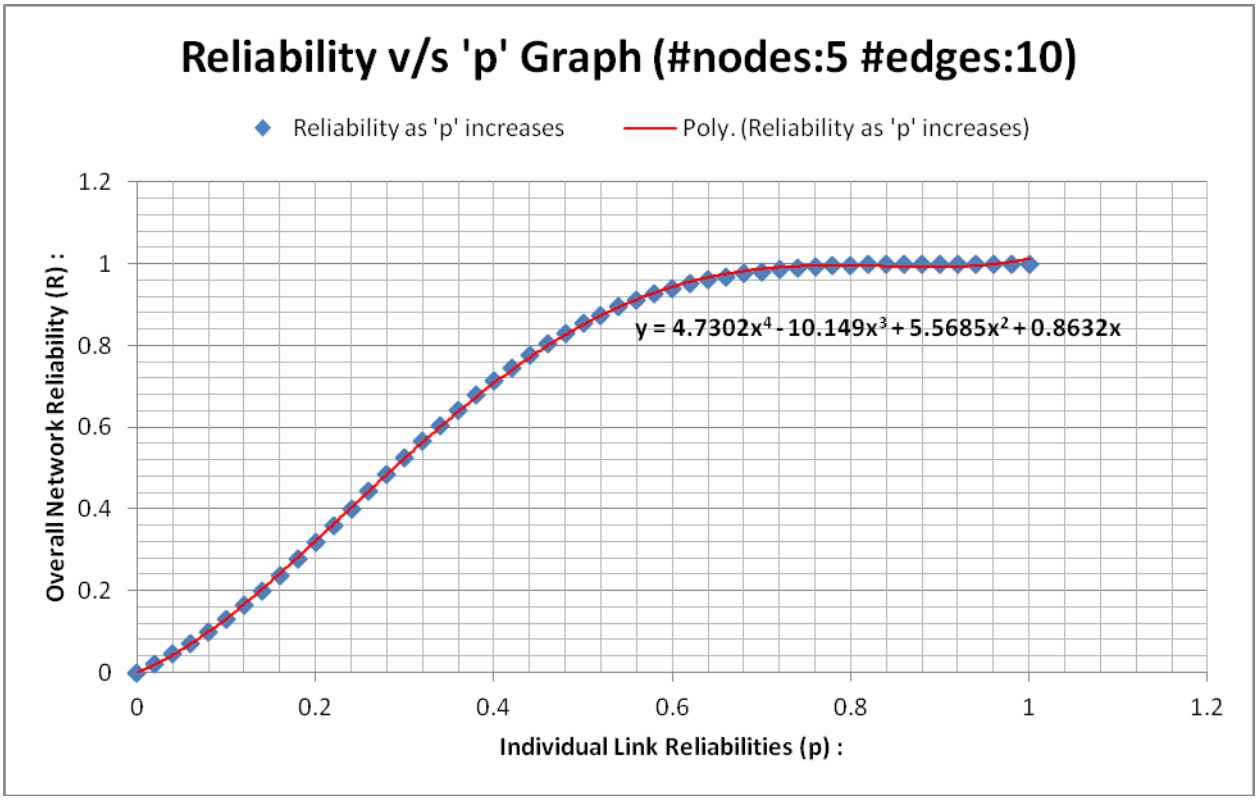
Individual Link Reliability = P Overall network reliability = R

P	R	P	R
0	0	0.5	0.853515625
0.02	0.021221522	0.52	0.874906663
0.04	0.044951327	0.54	0.894115291
0.06	0.07123806	0.56	0.91121316
0.08	0.100062944	0.58	0.926293552
0.1	0.131340659	0.6	0.939467674
0.12	0.164923215	0.62	0.950860906
0.14	0.200606112	0.64	0.9606091
0.16	0.238136109	0.66	0.968854997
0.18	0.277220056	0.68	0.97574486
0.2	0.31753431	0.7	0.981425355
0.22	0.358734314	0.72	0.986040765
0.24	0.40046402	0.74	0.989730543
0.26	0.442364876	0.76	0.992627257
0.28	0.484084151	0.78	0.994854919
0.3	0.525282443	0.8	0.996527718
0.32	0.565640253	0.82	0.99774913
0.34	0.604863529	0.84	0.99861139
0.36	0.642688165	0.86	0.999195298
0.38	0.67888342	0.88	0.999570299
0.4	0.713254298	0.9	0.999794803
0.42	0.74564291	0.92	0.999916672
0.44	0.775928896	0.94	0.999973821
0.46	0.804028975	0.96	0.999994857
0.48	0.829895716	0.98	0.99999968
0.48	0.829895716	1	1

NOTE:

1) **A direct observation from the table is that as individual link reliability increases, the overall network reliability increases too.**
2) **The increase in network reliability is steep initially but later the increase slacks down.**
3) **The above fact is corroborated by the fact that since the link reliability can at most be 1, the network reliability has to reach a saturation.**

Graph showing network reliability corresponding to link probability:



NOTE:

1) **A direct observation from the graph is that as individual link reliability increases, the overall network reliability increases too.**

2) **The increase in network reliability is steep initially but later the increase slacks down.**

3) **The above fact is corroborated by the fact that since the link reliability can at most be 1, the network reliability has to reach a saturation eventually as we can see in the graph where the slope stabilizes and runs parallel to the horizontal axis.**

TASK 4

Copy paste the following code from the first line of comment in the 'Task4.py' named file and run it from command prompt in windows using python interpreter. Example: python task4.py

```
from itertools import combinations                    #import itertools library
import networkx as nx                                 #import network library
import random                                         #import random library
G=nx.dense_gnm_random_graph(5,10)                     #create a graph G with 5 nodes and 10 edges
t=G.edges()                                           #store the number of edges in tuple 't'
p=0.9                                                 #set individual link probabilities to 0.9
a=0                                                   #set a to 0
r=0                                                   #set initial reliability to 0
s='nothing'                                           #set network state s to 'nothing'
def main():                                           #define a function main()
num=[]                                #initialize an multidimensional array to hold 'r' and 's' values.
#num will then be 1024 in length and contain all possible network states
#and link state combinations. That is 2^10=1024
for i in xrange(0,11):                                #for I varying between 0 to 10
#for each value of I from above, return the possible combinations on t
  for subset in combinations(t,i):
    count=0                                           #set count to 0
    for j in subset:                                  #for each element in subset(combination)
     if len(subset)!=0:                  #check if the subset has length >0 that is the subset is not empty
#increment the count. This gives the number of links that will be brought down
       count=count+1
     G.remove_edge(j[0],j[1])                         #remove the links from the graph
#after the prevous step, check if there is still connectivity between S and T nodes
   if (len(list(nx.all_simple_paths(G,0,2))) > 0):
      s='up'                                          #if connectivity is there then network is up.
   else:                                                                          #else
      s='down'                                        #network is down
#irrespective of the fact hether the network is up or down, calculate reliability value r
    r=(pow(1-p,count))*(pow(p,(10-count)))
    for k in subset:                                  #for each k value
      G.add_edge(k[0],k[1])                           #add back the removed links to the graph
      num.append([r,s])                               #add values of r and s to num.
#at the end of the las step, num will contain 1024 elements of the form [r,s], that is itll contain the
#reliability value and the network state for each combination of link states.
#print num
```

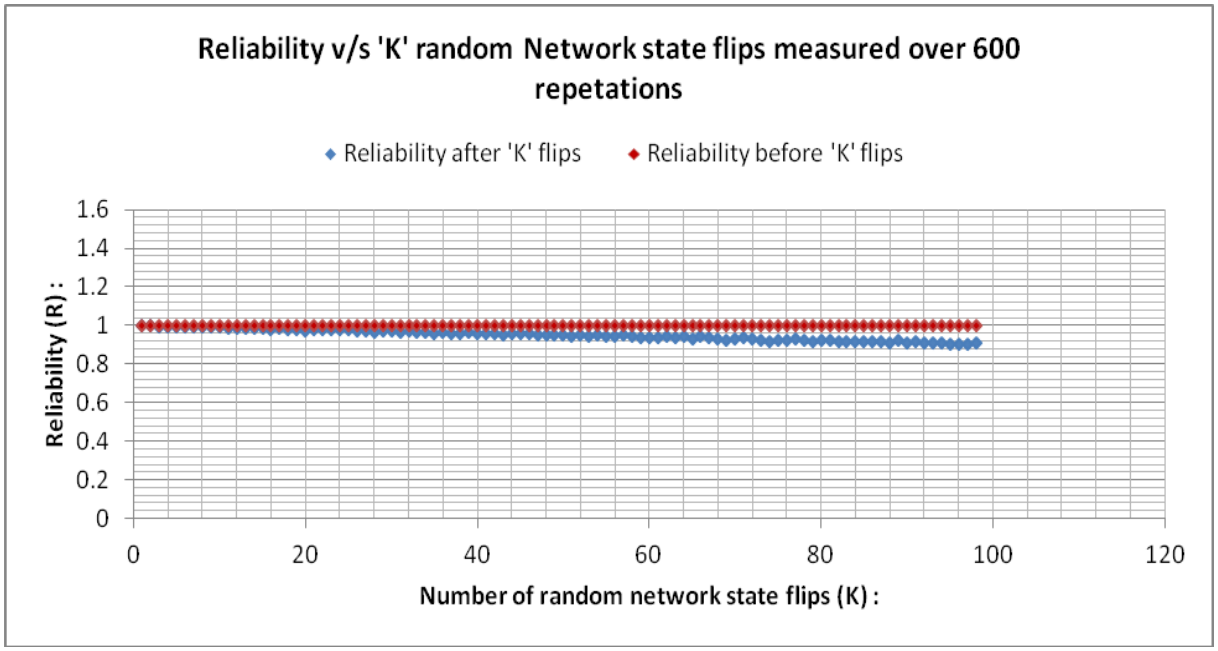
```
#the above part of the code was essential to finding the array num.
#in the below part we use num array for our flipping experiment whereby
#we will randomly flip a given number of edges from up->down
#and down->up state and calculate the reliability of the network and compare it
#to the original reliability if the network wasn't flipped.
for i1 in range(1,99):                                  #for number of flips in the range 1 to 99
  r1=0                                                  #initialize the reliability value after the flip to 0
  r2=0                                                  #initialize the reliability value before the flip to 0
#we run the experiment 600 times. That is we for any given value of number of flips in one
#experiment, we repeat the experiment 600 times to average out the reliability value.
  for freq in range(1,600):
#index is a tuple of i1 (from previous loop) number of index values picked from num randomly
    index=random.sample(range(1024), i1)
     for a1 in range(0,len(index)):                     #for each a1 from 0 to length of index tuple.
       if num[index[a1]][1]=='up':                      #if num[index[a1]][1] value is up
         num[index[a1]][1]='down'                       #flip the above value to down
       else:
         num[index[a1]][1]='up'                         #flip the above value to up
#after running likewise through the entire index tuple, we should be having all the flips
#calculate the reliability of the flipped network by calling the reliability function.
    r1=r1+reliability(num)
#after the flipped network reliability has been calculated, we need to revert the network state to what
#was previously there.
      for a2 in range(0,len(index)):                    #for each a1 from 0 to length of index tuple.
        if num[index[a2]][1]=='up':                     #if num[index[a1]][1] value is up
          num[index[a2]][1]='down'                      #flip the above value to down
        else:
          num[index[a2]][1]='up'                        #flip the above value to up
#calculate the reliability of the original network by calling the reliability function
     r2=r2+reliability(num)
     print r1/float(600),r2/float(600)      #print the average reliability values before and after the flips

def reliability(num=[[]]):              #define the function reliability that takes num array as an input
  r=0                                                   #initialize the reliability sum to be 0
  for i in range(0,len(num)):                           #for each I from 0 to length of num
  if num[i][1]=='up':               #if the second element of each num element is 'up do the following'
  r=r+num[i][0]      #add the value of the first element of each num element. i.e. add the reliabilities.
  return r                                              #return the final reliability value.
#if this block of the code is 'main', then execute the following
if __name__=='__main__':
#this will execute the function described above.
      main()
```

Table showing number of flips, network reliability after the flip and network reliability before the flips:

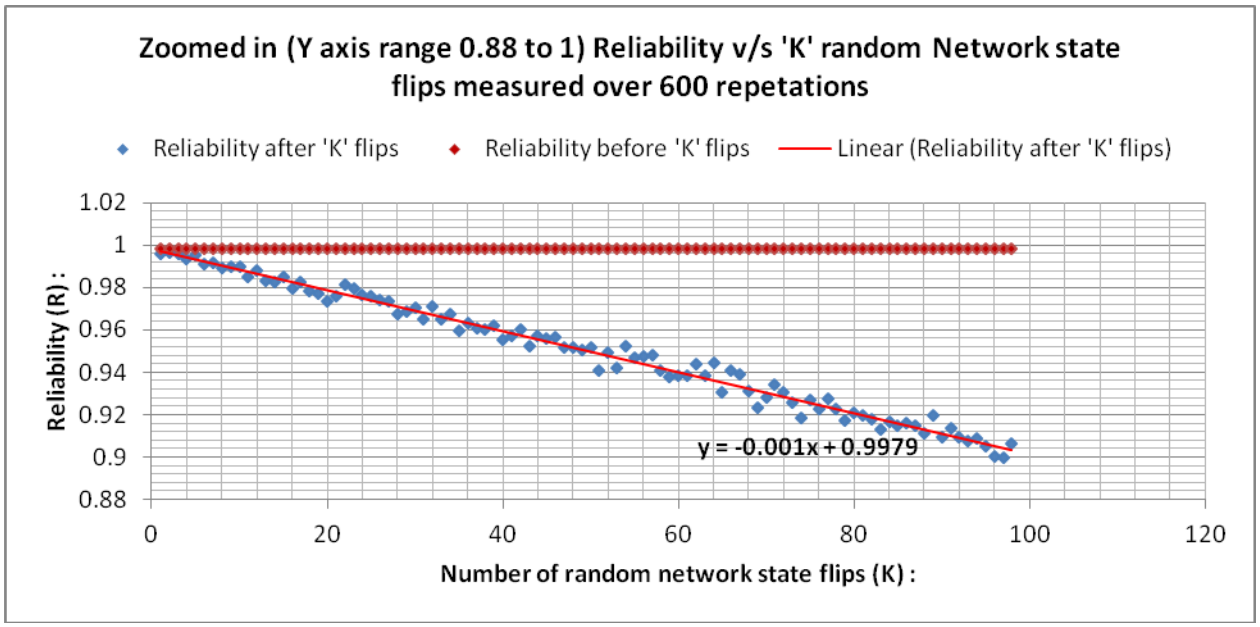
Reliability after Flipping = Fa , Reliability before flipping=Fb

K Flips	Fa	Fb	K Flips	Fa	Fb	K Flips	Fa	Fb
1	0.995934	0.998128	49	0.950239	0.998128	36	0.963322	0.998128
2	0.996723	0.998128	50	0.951708	0.998128	37	0.960699	0.998128
3	0.995632	0.998128	51	0.940959	0.998128	38	0.959908	0.998128
4	0.993665	0.998128	52	0.949562	0.998128	39	0.961987	0.998128
5	0.995172	0.998128	53	0.941938	0.998128	40	0.955204	0.998128
6	0.990988	0.998128	54	0.952175	0.998128	41	0.957415	0.998128
7	0.991642	0.998128	55	0.946993	0.998128	42	0.95992	0.998128
8	0.989225	0.998128	56	0.947394	0.998128	43	0.952564	0.998128
9	0.98975	0.998128	57	0.948018	0.998128	44	0.957326	0.998128
10	0.98982	0.998128	58	0.940635	0.998128	45	0.955666	0.998128
11	0.985037	0.998128	59	0.938004	0.998128	46	0.956512	0.998128
12	0.987872	0.998128	60	0.938382	0.998128	47	0.951629	0.998128
13	0.983437	0.998128	61	0.938562	0.998128	48	0.951511	0.998128
14	0.98239	0.998128	62	0.944122	0.998128	98	0.906636	0.998128
15	0.984707	0.998128	63	0.93828	0.998128	84	0.916526	0.998128
16	0.979316	0.998128	64	0.944448	0.998128	85	0.91499	0.998128
17	0.982485	0.998128	65	0.930271	0.998128	86	0.915888	0.998128
18	0.97813	0.998128	66	0.940791	0.998128	87	0.915043	0.998128
19	0.976927	0.998128	67	0.938871	0.998128	88	0.911099	0.998128
20	0.97355	0.998128	68	0.93136	0.998128	89	0.919832	0.998128
21	0.975747	0.998128	69	0.923571	0.998128	90	0.909657	0.998128
22	0.981621	0.998128	70	0.927924	0.998128	91	0.913627	0.998128
23	0.979819	0.998128	71	0.934449	0.998128	92	0.909099	0.998128
24	0.976724	0.998128	72	0.930622	0.998128	93	0.907815	0.998128
25	0.975885	0.998128	73	0.925923	0.998128	94	0.909028	0.998128
26	0.974388	0.998128	74	0.918567	0.998128	95	0.904845	0.998128
27	0.973741	0.998128	75	0.926648	0.998128	96	0.900466	0.998128
28	0.967373	0.998128	76	0.92251	0.998128	97	0.899694	0.998128
29	0.968919	0.998128	77	0.927411	0.998128	81	0.919849	0.998128
30	0.970707	0.998128	78	0.922404	0.998128	82	0.917718	0.998128
31	0.964952	0.998128	79	0.917321	0.998128	83	0.913036	0.998128
32	0.971172	0.998128	80	0.920907	0.998128	34	0.967519	0.998128
33	0.965278	0.998128	35	0.959389	0.998128	1	0.967519	0.998128

Graph showing network reliability corresponding to link probability:



NOTE FOR ABOVE GRAPH: (Focus on the range of Y axis values) we see that the network reliability after k flips does not deviate much form the reliability before the flips. This makes sense intuitively. As the number of network states possible are 1024. Flipping the network brings it back to one of its other states. So effectively on the whole there is not much appreciable change in the the reliability of the network. The network should also behave like this. Any state flips should not affect the reliability of the network. The network should be robust to changes in the network states which is also the behavour shown by the graph.



NOTE FOR ABOVE GRAPH: (a zoomed in version of the first graph on this page), we see that the network reliability after k flips deviates from the reliability curve before the flip by a maximum of 0.1 units meausred along R (vertical axis) at a value of K=99. This is **NOT AN APPRECIABLE** change as the slope is only -0.001 and the change of 0.1 is very trivial.

TASK 5

Intuitive explanation for graphs obtained in Task2:

The graph in Task2 shows the plot between number of links down and the corresponding network reliability. As we pull down more and more links in a network, the reliability of the network decreases. This means that with more number of links down, it'll get more difficult to keep the Source node connected to the target node. This behavior of the network is confirmed by the graph where we see a sheer decrease in the reliability as the numbers of nodes go down. The decrease is more prominent in the region where number of links is between 5 and 8. This can be explained by the fact that if the number of links that fail is towards a larger value, the reliability will still decrease but the decrease won't be steep. Pulling down 900 links out of 1000 available links and pulling down 800 out of 1000 links has almost the same effect. But the contrast is more in case we pull down 100 out of 1000 links and see the difference with respect to 900 links pulled down.

Intuitive explanation for graphs obtained in Task3:

We know that the overall network is made up of individual links. So the overall network reliability depends on individual link reliability. So if the individual links reliability is low, the overall network reliability is also low because the network reliability is the sum of products of link reliability. This is very well confirmed by the graph which shows a steady increase in network probability.

Also it's worth noting that overall network reliability saturates at 1. This is because the link probability can be at most 1.

Intuitive explanation for graphs obtained in Task4:

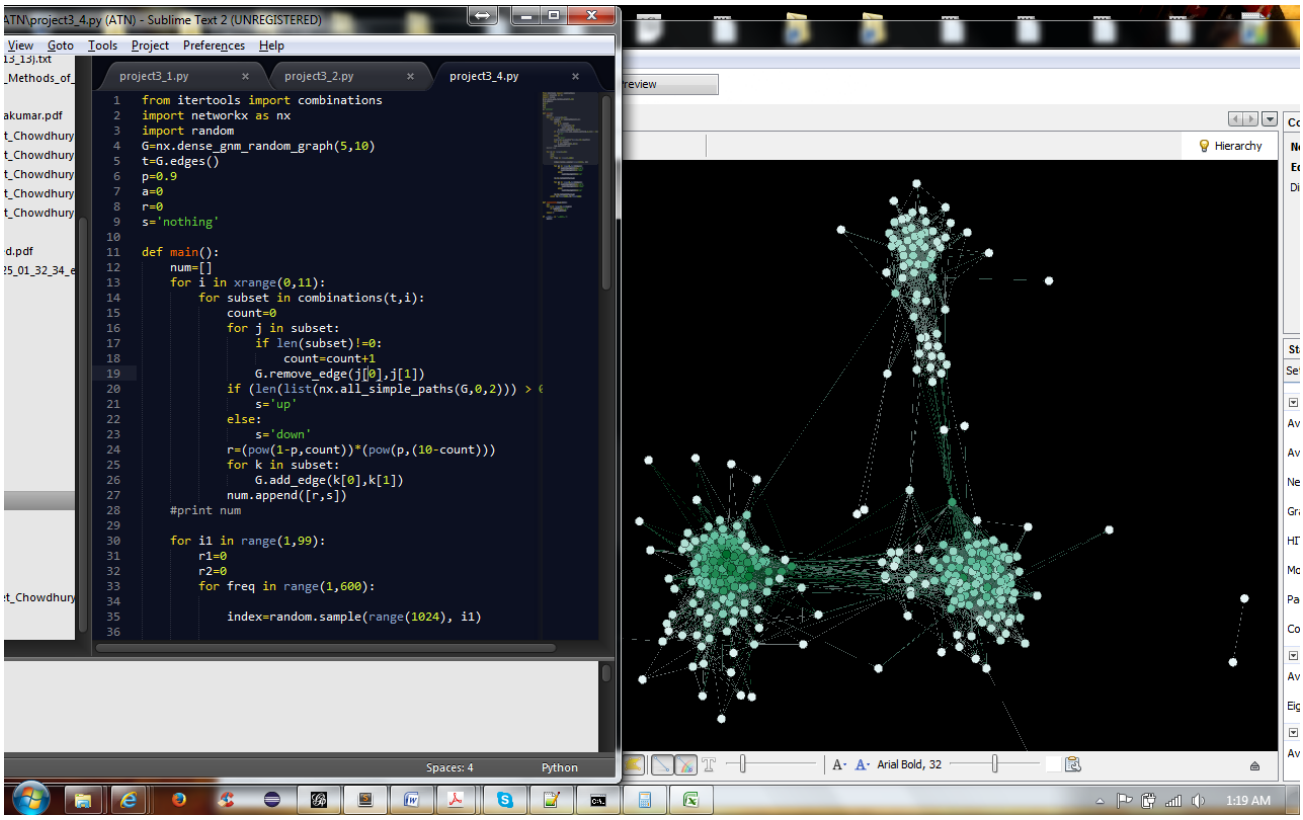
Ideally the network should be robust to any state flips. So the reliability should not change appreciably in case the states are flipped from up tp down or down to up. This desired behavior of the network is very well shon in the graph. From the first graph on page 18 we see that network reliability after k flips does not deviate much form the reliability before the flips. This makes sense intuitively. As the number of network states possible are 1024. Flipping the network brings it back to one of its other states thus balancing out the previos state. So effectively on the whole there is not much appreciable change in the the reliability of the network.

From the second graph, which is a zoomed in version of the first graph, we get another confirmation with respect to the change. The slope of the line is -0.001 which is almost equal to 0 thus running parallely and horizontally to the line which depicts reliability values before the flip all in the range k=0 to k=99.

REFERENCES

1) https://www.python.org/doc/
2) https://networkx.github.io/
3) https://www.youtube.com/
4) http://stackoverflow.com/

MY WORKSPACE (GEPHI+PYTHON) ☺



END