The University of Texas at Dallas

PROJECT 2

IMPLEMENTATION OF NAGAMOCHI IBARAKI ALGORITHM

CS 6385

Debaspreet Chowdhury **UTD ID - 2021211268**

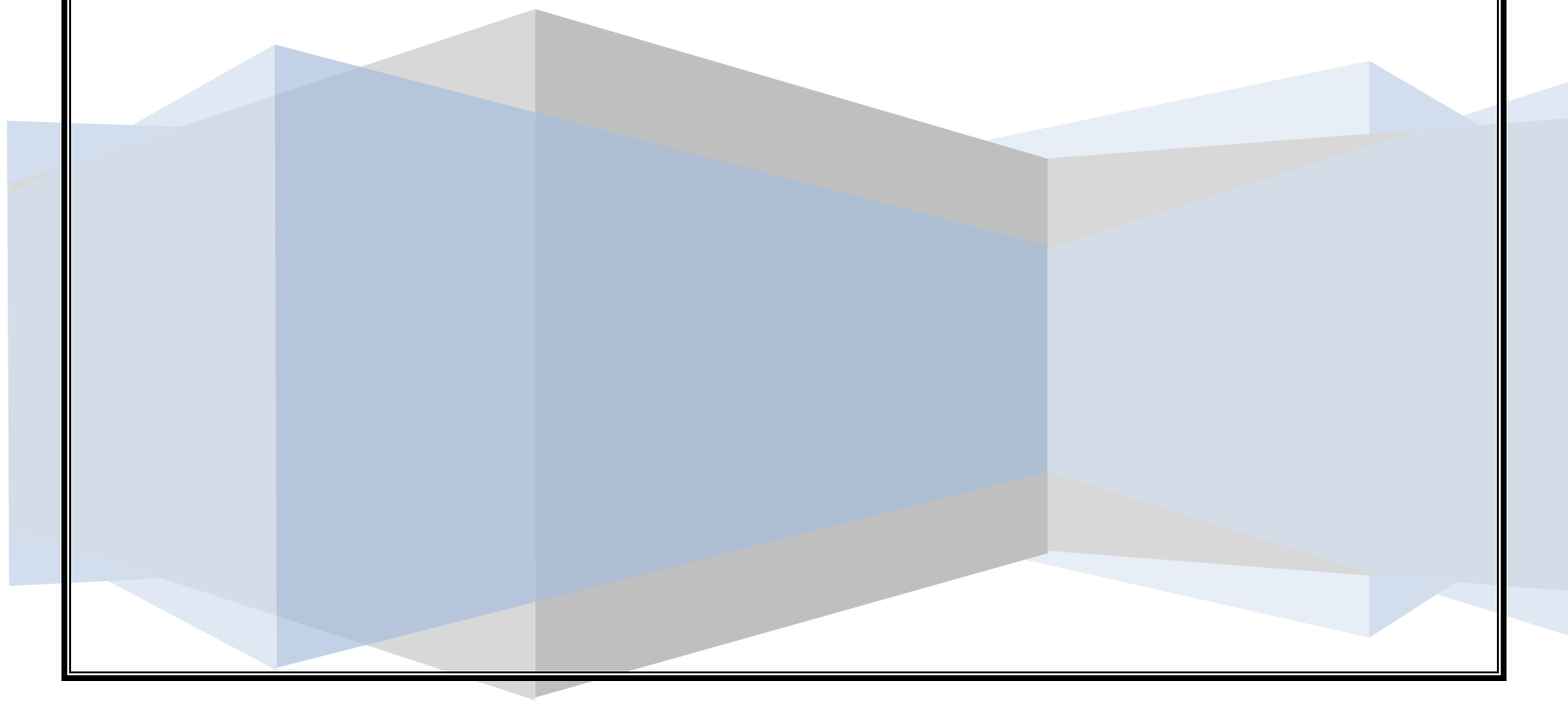


TABLE OF CONTENTS

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 network package 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.neighbors_iter(n) / G[n]**

   This returns the neighbors of node 'n' when used in this format [n for n in G[n]]. Source code available at https://networkx.github.io/documentation/latest/_modules/networkx/classes/graph.html#Graph.neighbors_iter

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

1) **nagamochi(G)**

   This function takes graph ‘G’ as input and implements the nagamochi ibaraki algorithm on it to find the minimum cut in the graph by using maximum adjacency orderings.

2) **contraction(G,n,m)**

   This function takes the graph ‘G’, node1 ‘n’ and node2 ‘m’ as inputs to contract the graph ‘G’ between the two nodes without self loops. The function returns the graph G after doing the operation.

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.
   5) #now we have the Maximum adjacency order ‘S’ comprising of nodes.

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) To run a 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.
8) I also use Gephi to display the graphs my programs generate. Gephi can be found at - https://gephi.github.io/users/download/
9) 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

Nagamochi Ibaraki Algorithm

It's a way to find the minimum cut in an undirected graph. L(G) denotes the edge connectivity of a graph G. For x and y to be nodes, L(x,y) denotes the connectivity between those two nodes.

At the heart of Nagamochi Ibaraki algorithm lays the maximum adjacency ordering vector. An MA ordering $V_1,\ldots,V_n$ of the nodes is generated recursively by the following algorithm.

1) Take any random node V_1
2) Once $V_1\ldots\ldots V_i$ is already chosen, take a node for V_{i+1} that has the maximum number of edges connecting it with the set $\{V_1,\ldots,V_i\}$.
3) Nagamochi and Ibaraki proved that this ordering has the following property:
   $L(Vn-1,Vn) = d(Vn)$ where $d(Vn)$ denotes the degree of the node Vn. i.e. the connectivity between the last two pairs of nodes is equal to the degree of the last node.

Steps and Pseudo Code are:

1) A node V_1 is picked at random and all the nodes connected to this node are then listed.
2) Of all the nodes in the list, the one having maximum number of edges connected to the node V_1 (or group of nodes) is picked and arranged in the order in which its picked with the previous node or group of nodes. The order in which such nodes are added to the group of nodes is of utmost importance as it is this order that'll decide the value of L(x,y)
3) This order of nodes in the list is called Maximum Adjacency ordering.
4) As the group size increases with the addition of a node to the group, there comes a point when the group size can't be increased any further. This situation comes when there are no adjacent nodes to this group of nodes. At this point let 'y' be the last node and 'x' be the second last node which were added to the group of nodes most recently. Then the following holds true -
   $L(x,y)=d(y)$
   We store the value d(y) in an array.
5) Now we contract nodes x and y and compute $L(G_{xy})$ on the remaining graph-which is done by repeating the above steps from 1 on the newly created graph after contraction.
   This process of contraction is repeated on and on till the point when we get 1 node after contraction.
6) The above steps will create a series of L(x,y), namely $L_1(x,y)$, $L_2(x,y)$, $L_3(x,y)$, $L_4(x,y)$, $L_5(x,y)$, $L_6(x,y)$ etc..
7) We thus get that $L(G) = \min\{L_1(x,y), L_2(x,y), L_3(x,y), L_4(x,y), L_5(x,y), L_6(x,y)\ldots\ldots\}$, which we also get from $L(G) = \min\{L(x,y),L(G_{xy})\}$

The program at a glance:

Create graph ‘G’ with n nodes and m edges using the function ***dense_gnm_random_graph(n,m)*** which creates an undirected graph with no self loops.

1) Create an empty list ‘S’ which will hold our nodes as we add nodes on the fly to this list thus creating our maximum adjacency order.
2) Assume initially for the first 2 iterations, each of the nodes will have exactly one edge connecting to its neighbor.
3) We add the seed node V1 to S. Then we find the neighbors of V1 using ***G.neighbors_iter(n) / G(n)*** which returns an array of nodes connected to V1. We choose the last node V2 from this array and add it to S. So now ‘S’ has the nodes V1,V2 in order. **We are thus forming our maximum adjacency vector.**
4) For each node in ‘S’, we create a neighbor array and join that array to the next nodes neighbor array. The loop continues till all the nodes in ‘S’ are covered. Let this array,which we initialize as empty be ‘N’
5) This new array ‘N’ has all the nodes which connect to our cluster of nodes in ‘S’. We now have to add the nodes with maximum edges connected to this cluster ‘S’
6) The logic is to find the maximum occurrence of a node in ‘N’. This node has the property that it occurs the maximum number of times in ‘N’ which implies that this node is a common neighbor to all the nodes of ‘S’ and thus has the maximum number of edges to the cluster ‘S’.
7) ***K=Counter(N).most_common(1)*** gives the nodes with the maximum occurrence and hence maximum number of edges. K[0][0] chooses the node with maximum edges to the cluster ‘S’. Let this node be V3.
8) V3 is added to ‘S’.
9) Steps 4 to 8 are iterated until len(K)==0.
10) Thus now we get a cluster ‘S’ that consists of $\{V_1,V_2,V_3 \ldots V_{n-1},V_n\}$.
11) Applying formula, $L_m(G)=d(V_n)$. We store this value in an array L.
12) We run the ***contraction(G,V_{n-1},V_n)*** to get a new graph.
13) Step 1 to 12 are iterated till len(G)==2 where len(G) returns the number of nodes in thegraph.
14) We apply min function on L to get L(G). This gives us the minimum cut.

TASK 2

L(G) is Lambda(G) which is the connectivity of the graph G

Ill now present the 2 programs I wrote for calculation of L(G). The contraction(G,n,m) is a subroutine which is called in the nagamochi(G,seed=n) function. Also for Task3,Task4 and Task5, the same function nagamochi(G,seed=n) has been called.

contraction(G,n,m)

The concept for contracting two nodes is that when we join/contract 2 nodes say x and y, we copy all the neighbors of y to x or vice versa and then delete all the edges of 'y' to its neighbors and then finally deleting the node 'y' itself.

```
#graph contraction that prevents self loops
def contraction(G,x,y):              #function input parameters: graph 'G', nodes 'x' and 'y'
  M=[m for m in G[x]]                #extract the neighbors of node 'x' and put them in an array.
  N=[n for n in G[y]]                #extract the neighbors of node 'y' and put them in an array.
  if G.has_edge(x,y):                #if there is an edge between 'x' and 'y', do the following.
    G.remove_edge(x,y)               #if above true, then remove edge between 'x' and 'y'
    M=[m for m in G[x]]              #extract the new neighbors of node 'x' and put them in an array
    N=[n for n in G[y]]              #extract the new neighbors of node 'x' and put them in an array
    for m in M:                      #for each node in array M
     for n in N:                     #for each node in array N
       if m==n:                      #if node 'n' == node 'm'
         if G.has_edge(y,n):         #if there is an edge between node 'y' and its neighbor 'n'
           G.remove_edge(y,n)        #if the above is true, then remove edge between  'y' and 'n'
       if m!=n:                      #if node 'n' != node 'm'
         if G.has_edge(y,n):         #if there is an edge between node 'y' and 'n'
           G.remove_edge(y,n)        #if above true, delete the edge
         G.add_edge(x,n)             #add edge between 'x' and 'n'
    G.remove_node(y)                 #remove node 'y' after contraction
  else:                              #this part executes if initially there was no edge between 'x' and 'y'
   for m in M:                       #for each node in array M
    for n in N:                      #for each node in array N
     if m==n:                        #if node 'n' == node 'm'
      if G.has_edge(y,n):            #if there is an edge between node 'y' and its neighbor 'n'
       G.remove_edge(y,n)            #if the above is true, then remove edge between  'y' and 'n'
     if m!=n: :                      #if node 'n' != node 'm'
       if G.has_edge(y,n):           #if there is an edge between node 'y' and 'n'
         G.remove_edge(y,n) )        #if above true, delete the edge
       G.add_edge(x,n) )             #add edge between 'x' and 'n'
    G.remove_node(y)                 #remove node 'y' after contraction
```

nagamochi(G)

The algorithm has already been described in the previous pages under program at a glance. Ill comment each line however to augment the readability of the program.

```
#nagamochi algorithm using maximum adjacency ordering and graph contraction
#takes graph 'G' as input parameter
G=nx.dense_gnm_random_graph(n,m)          #creating a random graph with 'n' nodes and 'm' edges

def nagamochi(G):                         #defining the funtion
  L=[]                                    #defining an empty list/array 'L' that holds
                                           graph connectivity values.
 while len(G)>=2:                         #while number of nodes in the
                                           modified graph 'G' is >=2, execute code below.
    S=[]                                  #define maximum adjacency list as an empty array
    Num1=[]                               #define array to hold the nodes of graph 'G'
    Num2=[]                               #define an empty array to hold the
                                           neighbors of  the first node in Num1
    N=[]                                  #define an empty array to hold
                                            all the neighbors cluster 'S'
    Num1=G.nodes()                        #assigning Num1 for the nodes of the graph 'G'
    S.append(Num1[0])                     #add the first node of Num1 to cluster 'S'
    Num2=[n for n in G[Num1[0]]           #Num2 holds neighbors of first node in Num1
    S.append(Num2[len(Num2)-1])           #add the last node of Num2 to cluster 'S'
    while len(K) != 0:                    #K holds the most commonly
                                           occurring nodes in the array 'N'
      for n in S:                         #for each node in cluster 'S'
        N=N+[m for m in G[n]]             #add the neighbors of each node in 'S' to list 'N'
      K=Counter(N).most_common(1)         #list the most commonly occurring nodes.
      S.append(K[0][0])                   #select the first node from
                                           the above list and add it to 'S'
   #repeat the above process till K==0 upon which exit the inner while loop.
   #now we have the Maximum adjacency order 'S' comprising of nodes.
   L.append(G.degree(S[len(S)-1]))        #calculate the degree of the
                                           last node in 'S' and append it to 'L'
   contraction(G,S[len(S)-1],S[len(S)-2]) #contract the last and 2nd last nodes of 'S'
  #the above contraction creates a new graph 'G'
  #repeat the process till we get a graph where there are only 2 nodes.
 return min(L)                            #find the minimum value of 'L' and return it.
```

TASK 3

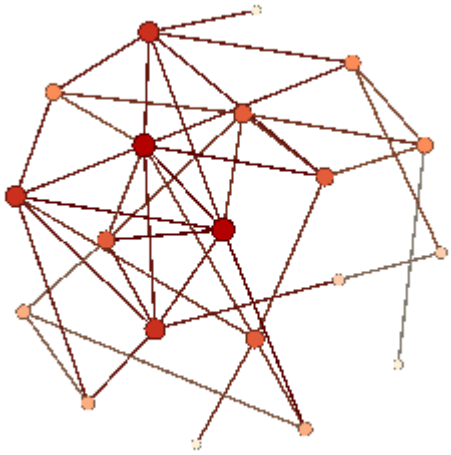
L(G) is Lambda(G) which is the connectivity of the graph G

```
import networkx as nx
#initialize a container array 'num' to hold the graphs
#and then generate graphs(n,m) with randomly selected edges
#and stuff the graphs into the container
del num
num=[]                                  #initialize graph container
n=20                                    #number of nodes is 20 fixed
for m in range(40,401,5):               #varying the edges in size of 5
 G=nx.dense_gnm_random_graph(n,m)       #generate graph with n nodes and m random edges
 num.append(G)                          #add to container
 print nagamochi(G),m                   #print graph connectivity v/s #edges
```

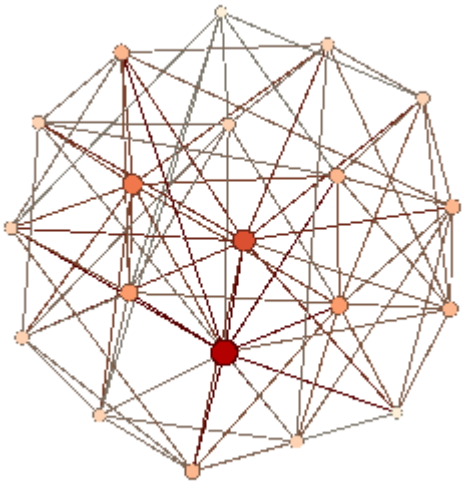
Number of Edges	L(G)	Number of Edges	L(G)	Number of Edges	L(G)
40	1	165	13	290	40
45	1	170	13	295	40
50	1	175	14	300	40
55	1	180	15	305	40
60	2	185	17	310	42
65	4	190	19	315	42
70	4	195	19	320	42
75	4	200	19	325	42
80	6	205	20	330	44
85	6	210	20	335	44
90	6	215	24	340	44
95	7	220	24	345	44
100	7	225	24	350	46
105	7	230	27	355	46
110	7	235	27	360	46
115	8	240	30	365	48
120	8	245	30	370	48
125	8	250	35	375	48
130	8	255	35	380	50
135	12	260	35	385	50
140	11	265	38	390	50
145	12	270	38	395	50
150	12	275	38	400	50
155	12	280	38	400	50
160	14	285	38	400	50

GRAPHS (Plotted using Gephi)

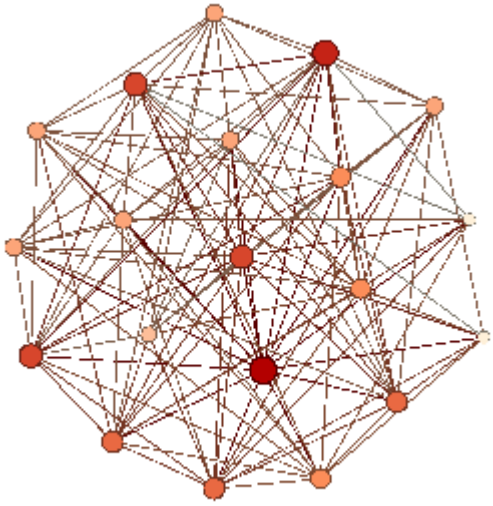
N=number or nodes M=number of edges L(G)=graph connectivity.



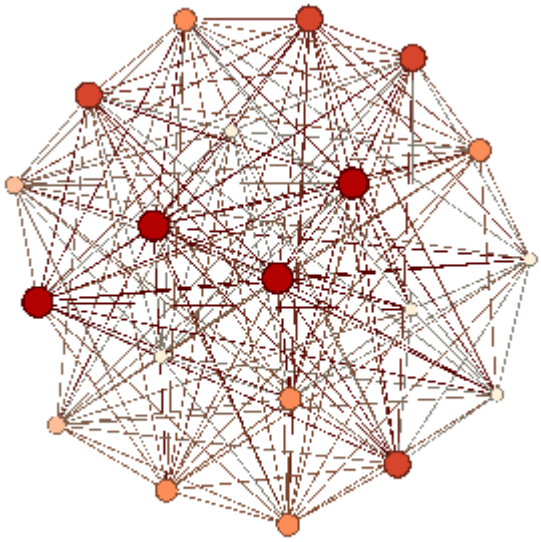
N=20, M=40, L(G)=1



N=20, M=80, L(G)=6



N=20, M=120, L(G)=8



N=20, M=160,L(G)=14

NOTE :

a) **As the average degree of the node increases, L(G) increases too.**

b) **In the above graphs, nodes with bigger and darker shapes have higher connectivity compared to the smaller nodes.**

c) **The source code for the function 'dense_gnm_random_graph' which I used in this program to generate graph is given in the Project description towards the beginning of this documentation.**

TASK 4

L(G) is Lambda(G) which is the connectivity of the graph G

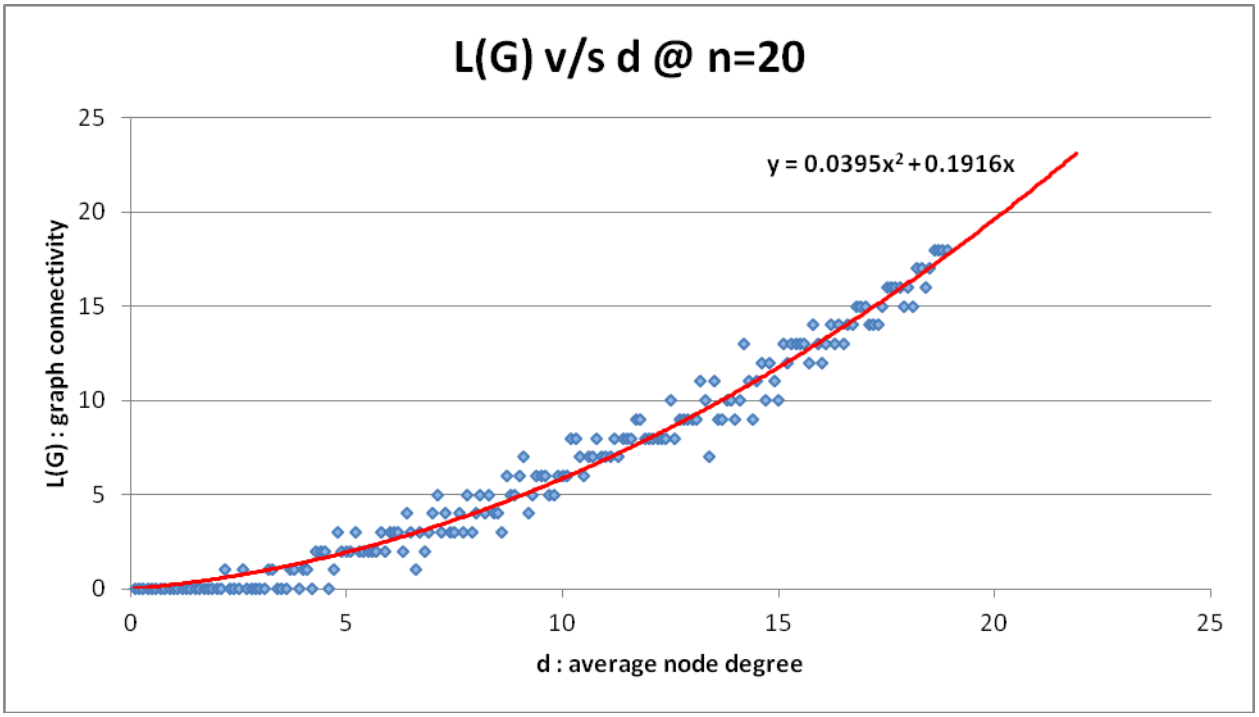
d=(2*m)/(n) where 'd' is the average degree of a node in the graph G. This degree is an integer. However I would first consider this degree to be floating point to give a clearer picture of the resulting graph plot and then consider another graph where only integral values of'd' have been considered and which is actually the true case.

```
#the program calculates L(G) and 'd' values for a graph with fixed number of nodes but varying
#number of edges between nodes.
import networkx as nx
n=20                                    #number of nodes is 20 fixed
for m in range(1,(n*n-n)/2):            #edges vary from 1 to 190
  G=nx.dense_gnm_random_graph(n,m) #creating a graph with 'n' nodes and 'm' edges.
  d=(2*m)/float(n)                      #calculating average degree of nodes in the graph
  k= nagamochi(G)                       #finding L(G) of the graph.
  print k,d                             #printing L(G) and average degree values
```

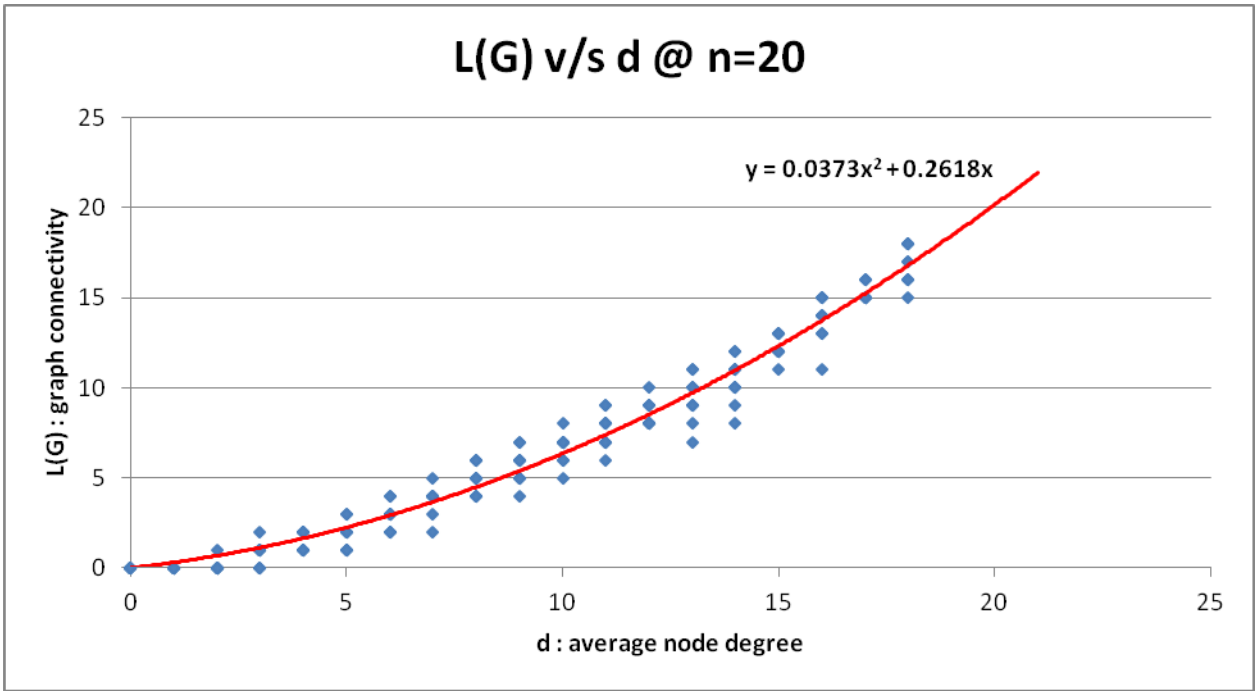
L(G)	d	L(G)	d	L(G)	d	L(G)	d	L(G)	d
0	0.1	1	3.8	3	7.5	8	11.2	11	14.9
0	0.2	0	3.9	4	7.6	7	11.3	10	15
0	0.3	1	4	3	7.7	8	11.4	13	15.1
0	0.4	1	4.1	5	7.8	8	11.5	12	15.2
0	0.5	0	4.2	3	7.9	8	11.6	13	15.3
0	0.6	2	4.3	4	8	9	11.7	13	15.4
0	0.7	2	4.4	5	8.1	9	11.8	13	15.5
0	0.8	2	4.5	4	8.2	8	11.9	13	15.6
0	0.9	0	4.6	5	8.3	8	12	12	15.7
0	1	1	4.7	4	8.4	8	12.1	14	15.8
0	1.1	3	4.8	4	8.5	8	12.2	13	15.9
0	1.2	2	4.9	3	8.6	8	12.3	12	16
0	1.3	2	5	6	8.7	8	12.4	13	16.1
0	1.4	2	5.1	5	8.8	10	12.5	14	16.2
0	1.5	3	5.2	5	8.9	8	12.6	13	16.3
0	1.6	2	5.3	6	9	9	12.7	14	16.4
0	1.7	2	5.4	7	9.1	9	12.8	13	16.5
0	1.8	2	5.5	4	9.2	9	12.9	14	16.6
0	1.9	2	5.6	5	9.3	9	13	14	16.7
0	2	2	5.7	6	9.4	9	13.1	15	16.8
0	2.1	3	5.8	6	9.5	11	13.2	15	16.9
1	2.2	2	5.9	6	9.6	10	13.3	15	17

L(G)	d	L(G)	d	L(G)	d	L(G)	d	L(G)	d
0	2.3	3	6	5	9.7	7	13.4	14	17.1
0	2.4	3	6.1	5	9.8	11	13.5	14	17.2
0	2.5	3	6.2	6	9.9	9	13.6	14	17.3
1	2.6	2	6.3	6	10	9	13.7	15	17.4
0	2.7	4	6.4	6	10.1	10	13.8	16	17.5
0	2.8	3	6.5	8	10.2	10	13.9	16	17.6
0	2.9	1	6.6	8	10.3	9	14	16	17.7
0	3	3	6.7	7	10.4	10	14.1	16	17.8
0	3.1	2	6.8	6	10.5	13	14.2	15	17.9
1	3.2	3	6.9	7	10.6	11	14.3	16	18
1	3.3	4	7	7	10.7	9	14.4	15	18.1
0	3.4	5	7.1	8	10.8	11	14.5	17	18.2
0	3.5	3	7.2	7	10.9	12	14.6	17	18.3
0	3.6	4	7.3	7	11	10	14.7	16	18.4
1	3.7	3	7.4	7	11.1	12	14.8	17	18.5
18	18.6	18	18.7	18	18.8	18	18.9		

Graph showing L(G) v/s d (with non integral values of d)



Graph showing L(G) v/s d (with integral values of d)



NOTE :

a) **As the average degree of the node increases, L(G) increases too.**

b) **In both cases (of integral and non integral values of d) , the increase wrt L(G) is quadratic in nature as depicted by the trendline in the graph with coefficient 0.03773.**

c) **Conclusion is as average degree increases so does the graph connectivity.**

L(G)	d	L(G)	d	L(G)	d	L(G)	d
0	0	1	4	5	8	8	13
0	0	2	4	6	8	9	13
0	0	2	4	5	8	7	13
0	0	1	4	4	9	9	13
0	0	2	4	5	9	11	13
0	0	1	4	6	9	9	13
0	0	3	5	6	9	10	13
0	0	2	5	7	9	10	13
0	0	1	5	7	9	11	13
0	1	1	5	6	9	10	14
0	1	1	5	5	9	11	14
0	1	3	5	5	9	8	14
0	1	1	5	6	9	11	14
0	1	2	5	7	10	10	14

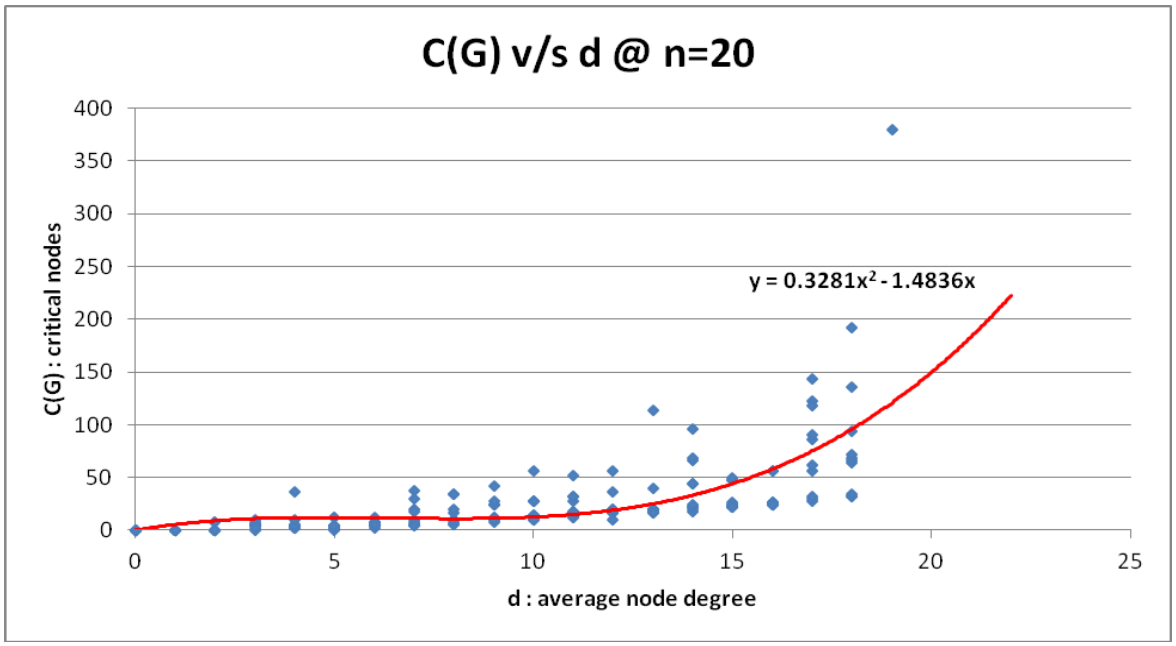
L(G)	d	L(G)	d	L(G)	d	L(G)	d
0	1	2	5	5	10	10	14
0	1	2	5	6	10	11	14
0	1	3	6	7	10	12	14
0	1	3	6	7	10	9	14
0	1	3	6	7	10	12	14
0	2	4	6	7	10	12	15
0	2	2	6	6	10	11	15
0	2	3	6	8	10	13	15
0	2	4	6	7	10	12	15
0	2	2	6	8	11	12	15
0	2	3	6	7	11	13	15
0	2	2	6	8	11	13	15
0	2	4	7	7	11	13	15
1	2	5	7	8	11	12	15
0	2	2	7	7	11	12	15
0	3	4	7	8	11	11	16
1	3	3	7	6	11	14	16
1	3	4	7	9	11	13	16
1	3	4	7	9	11	14	16
1	3	4	7	8	12	14	16
0	3	4	7	8	12	14	16
1	3	4	7	8	12	15	16
0	3	6	8	8	12	13	16
2	3	4	8	9	12	15	16
1	3	4	8	9	12	15	16
1	4	5	8	8	12	15	17
2	4	4	8	9	12	16	17
2	4	6	8	8	12	15	17
1	4	5	8	10	12	16	17
17	18	15	18	10	13	15	17
18	18	17	18	16	17	15	17
18	18	16	18	15	17	15	17
18	18	16	18	15	17	15	17
18	18	16	18	15	17	13	11

TASK 5

L(G) is Lambda(G) which is the connectivity of the graph G and C(G) is the number of critical edges in the graph.

A node is critical if its deletion cause the graph connectivity to decrease. Thus for any 'e', if L(G-e) < L(G), then that e(edge) is critical. For my algorithm to find the number of critical edges in the graph and in turn plot this number C(G) v/s the average node degree, I implement the following code which is commented.

```
#for each increment in 'm' while keeping 'n' constant at 20, we calculate C(G) and 'd' and print the values.
import networkx as nx
for m in range(1,191):                    #for edges varying from 1 to 190
 n=20
 G=nx.dense_gnm_random_graph(n,m) #create a random graph
 T = nagamochi(G,1)                       #fetch the initial graph connectivity.
 c=0                                      #initially in any graph, number of critical edges is 0
 num1=G.nodes()                           #initialize num1 array with all the nodes in the graph
 num2=num1                                #initialize num2 array with all the nodes in the graph
 for i in num1:                           #compare each element of num1 with num2
  for j in num2:
    if G.has_edge(i,j):                   #check if there is any edge between nodes.
     G.remove_edge(i,j)                   #if yes then remove the edge
     if nagamochi(G)<T:                   #check if the graph connectivity < initial connectivity.
      c=c+1                               #increment counter
     G.add_edge(i,j)                      #add the removed edge back to the graph.
 d=(2*m)/n                                #find degree of node.
 print c,d
 G.clear()                                #clear existing graph to make space for the next grap
```



NOTE :

a) As the average degree of the node increases,C(G) increases too as evident from the graph.

TASK 6

Degree of a node is the number of edges connected to the node. Average degree is the average of this number over the entire graph. Numerically this degree, d = 2*m/n where m=number of edges in the graph and n is the number of nodes in the graph. **So degree can also be termed as 'number of edges per node' which is inductively a measure of density.**

An intuitive approach to why L(G) increases with average degree or the expected behavour ?

Intuitively, if the number of edges per node is more (ie degree is more) then the network will be more dense. If the network is more dense, that means its more connected. So we will need to cut more number of edges to disconnect the node. This by definition is the connectivity of the graph because graph connectivity is defined as the minimum number of edges that need to be broken in order to disconnect the graph. So with a denser graph/network, we get a greater value of graph connectivity – ie we will need to cut more number of edges down to disconnect the graph. Thus fortifying our observation from the graph we obtained in Task3 (more the average degree, greater is L(G)).

However 'd' is the average degree. So some nodes may have degree more than the average and some less than the average. In case the average goes up, that means more nodes have greater degrees and lesser nodes have less degree. This inductively points to the fact that we have to disconnect relatively more number of edges to disconnect the graph.

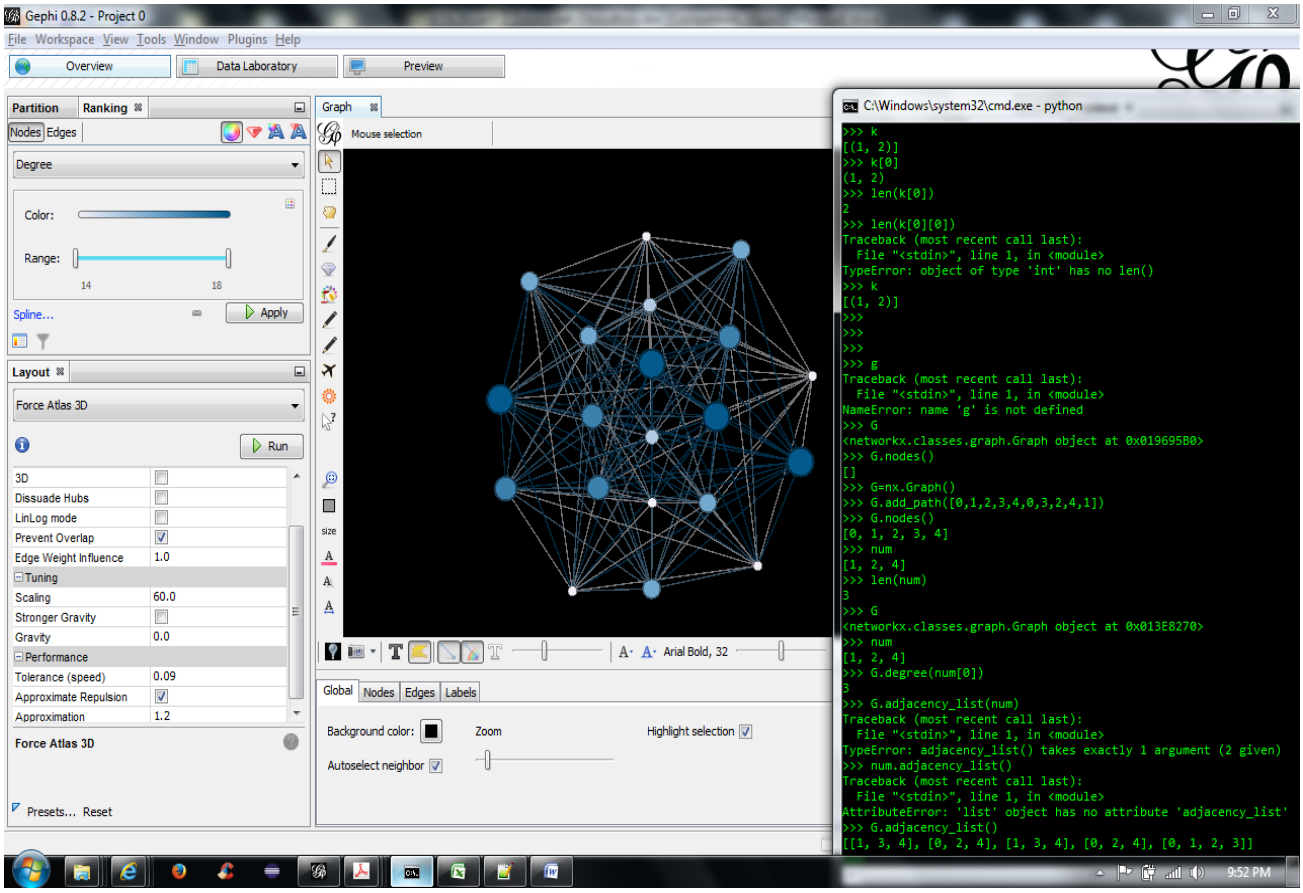
An intuitive approach to why C(G) increases with average degree or the expected behavour ?

Critical edge is an edge which if removed decreases the connectivity of the graph. Intuitively, this edge is more likely to be towards the edge between the two most connected nodes in the graph or relatively heavily connected nodes in the graph. If this link goes down, the network will get disconnected. In a way, the connectivity of the network/graph depends on this node to some extent. And thus the number of such critical links can be greater only in dense graphs or networks. And because dense networks are a result of nodes existing with a higher degree, we deduce the fact that as the average degree increases, so does the number of such critical edges. This fact is supported by the result we obtain in Task4.

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