STUDY ON CONTROLLABILITY OF COMPLEX NETWORKS

BY

TAPISH RATHORE

UG201010037

CH. SIVA PRATHEEK

UG201011005

SRIKANTH M

UG201011035

Dr. GANESH BAGLER

Introduction

The ability to control a natural or technological system is of paramount importance in science and engineering because it enables us to get a more detail picture of our understanding of those systems. Over the years various analytical tools have been developed to provide a solid framework for controllability of complex networks. A complex network, in context of network theory is a graph with non-trivial topological features. Complex Networks are ubiquitous, they are found in natural, social and man-made systems, such as gene-regulatory networks and mobile sensor networks. In our work we apply the various tools developed to real networks and calculate the number of driver nodes, which is mainly dependent on degree distribution of the network.

According to control theory, a system is said to be controllable if the state of a system can be driven from any initial state to any final state within finite time. The difficulty about this is rooted in the fact that two independent factors influence controllability which are listed below, each with its own layer of unknown.

1. The architecture of the system, represented by network which contains information regarding the interacting components.
2. The dynamical rules that capture the time dependent interactions between the components.

Thus progress has been made only on those networks where these layers are well mapped such as small biological circuits and rate control for communication networks. To ultimately develop a framework to control complex and non-linear networks, the first step is to investigate the controllability using linear time-invariant dynamics

$$\frac{dx}{dt} = Ax(t) + Bu(t)$$

Where the vector **x(t)**=$(x_1(t)\ldots.x_N(t))^T$ captures the state of a system of N nodes at time ‘t’. The state of a node may denote the amount of traffic passing through that node in a communication network. The matrix ‘A’ is a square matrix of dimensions ‘N’ and describes

wiring of the network as well as interaction strength between the components and ‘B’ is a matrix with dimensions (M,N) and it represents the nodes controlled by an outside controller. The system is controlled using a time-dependent input vector u(t). Therefore, in order to control a system we must first identify the set of nodes, which when given a control input can fully control the network. We call these nodes as the driver nodes.

The mathematical condition for controllability of a system is given by Kalman’s controllability condition which states that the rank of the controllability matrix should be full. Since we need complete information regarding the weights of links of a network which is not possible in most of the cases and moreover if possible, a brute force search requires repeated computation of matrix ‘C’ which is computationally expensive. Thus the concept of structural controllability is used to overcome this limitation of weights and to avoid brute force search of driver nodes, a relatively efficient algorithm called the “Maximum Matching Algorithm” is used to compute the driver nodes. The maximum matching rule states that a node is said to be matched if a link in the maximum matching points at it.

Salient highlights of the work

1. Re-evaluation of the driver node calculations in many of the networks analyzed by [1].
2. Key directions for ranking driver nodes.
3. Quantitative analysis of certain networks which anomalous fraction of driver nodes. An attempt was done to find the effect of motifs present in those networks on the required fraction of driver nodes.

ALGORITHMS USED FOR CREATING RANDOM CONTROLS

ER-Model [6]:

In graph theory, there are two closely related models for generating random graphs. One of these models is the ER Model which produces random graphs with N nodes and K edges. The ER model was first proposed with the original purpose of studying the network by using probabilistic methods, the properties of graph as a function of increasing number of random connections. The ER graphs are generated by connecting couples of randomly selected nodes, prohibiting multiple connections, until the resulting graph has K edges. For complete description of the ER model, the entire statistical ensemble of possible realizations need to be described. An alternative model of ER graphs consists of connecting each couple of nodes with a probability ‘p’. This alternative model for constructing the ER model involves easier analytical computations. The ER models are among the best studied graph models even though they do not reproduce most of the properties of real networks.

Degree preserved Random Network Generation [7]:

Another popular random control that can be applied on networks is to create a degree conserved random network corresponding to the real world network. This type of control helps us to create degree conserved models that are particularly useful in the algorithms where degree distribution of the network is found to have a complete control on some property of network. Therefore, by using this network, the conclusions can be validated for these algorithms, where the concerned feature of original network will remain invariant under its degree conserved random control.

To create a real world network, we simply take pair of edges and interchange their destination nodes. This rewiring is subject to a verification that there is no edge that is already existing between the corresponding pair of nodes, which helps in avoiding duplicate edges. This process of rewiring is done for the order of number of edges in the network, thereby randomizing the resulting network significantly.

Initial strategies used for finding driver nodes

The study of complex networks helps in the analysis of various problems modelled as networks. Our first attempt was to analyse biological systems which had been modelled as networks. The protein –protein interactome (PPI) is a network which represents individual proteins inside the cells as nodes and the bonds between them as edges. The whole set of interactions existing in a cell is termed interactome.

We had hoped to identify biologically relevant features by studying controllability of PPI networks.

Network controllability deals with controlling the dynamics of the system by taking the system from a particular state to a desired output state. By applying network controllability on protein-protein interactomes we could not only identify essential genes, but also identify potential drug targets for curing diseases.

Driver nodes are the minimum number of nodes, which must be controlled (given an input) to change the state of the system. Calculation of driver nodes for a large network is computationally difficult and involves certain approximations (in the calculation of the rank).

The number of driver nodes for any particular network is fixed. But the driver nodes themselves are not unique. This eliminated the possibility of finding a meaningful result associated with biological networks, which required the identification of unique nodes that could be mapped to essential genes (genes that are pivotal in the formation of a particular PPI network).

So, we concentrated towards developing a strategy to identify driver nodes in complex networks, setting aside the biological interaction networks.

The strategy initially developed made use of the standard maximal cardinal matching algorithm on directed as well as undirected graphs. However, the strategy failed to take into account self-loops. This error was identified and corrected. The corrected strategy identifies driver nodes in directed graphs by converting them first to undirected bipartite graphs and then applying maximum cardinal matching.

Actual procedure for finding driver nodes [1]

The final and correct strategy used to identify driver nodes is applicable only for directed graphs. It involves –

1. Converting the directed graph to a bipartite graph
2. Finding the maximum matching in the converted bipartite graph

The directed graph is converted into its bipartite counterpart by applying the following algorithm –

```
[m,n] = size(adj);
```

```
for i = 1:m
   for j = 1:n
      if adj(i,j)~=0
         bipartite(i,n+j) = adj(i,j);
         bipartite(n+j,i) = adj(j,i);
      end
   end
end
```

An example from [1]–

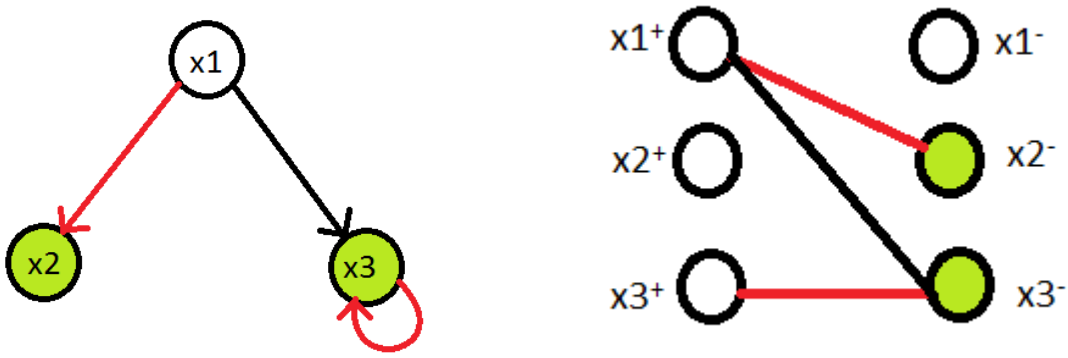


Figure 1: Conversion of Directed Graph to Bipartite Graph

The maximum matching in the bipartite graph is calculated using the MATE function [2].

The MATE function calculates the maximum cardinality matching on a graph. Mate(i) = j would mean that edge (i,j) belongs to the maximum matching.

The nodes which do not appear in the maximum matching are driver nodes. In the above example, x1 is unmatched, hence a driver node.

Ranking of Driver nodes

The controllability of complex networks is relevant to many areas of science and engineering such as in biology, mechanical systems etc. In order to control a complex network, some amount of energy has to be spent so that the system achieves the desired state. Moreover, we have also seen that by controlling special nodes in network called as the ‘Driver’ nodes, we are able to control the entire network.

In using complex systems such as mechanical systems and biological systems, an important and unavoidable issue is the energy required to control such a complex network i.e. the cost of controlling the network. Hence, we proposed the problem of ranking the driver nodes in order of their energy consumption for controlling the network [5]. Results for the energy required to control a complex network with a single driver node were already available, although assumptions were made at arriving at the simplified result, one of such assumptions being that the network must be a weighted undirected network. We therefore tried to improve

the functionality of the above method by extending it to multiple driver node case as this would be helpful in ranking the driver nodes.

The algorithm which can be used for ranking of driver nodes provided the above method is feasible is given below:

1. Assume that a complex network has N driver nodes.
2. We replace the m^{th} driver node, which is a part of the N driver node systems with an alternative set of node(s) by making changes to the maximum matching algorithm i.e. by changing the path of traversal near that node.
3. We then calculate energy required to control the system. This is the energy required to control the system in absence of the m^{th} node.
4. Similarly, we calculate energy required to control the system in a similar manner for all N driver nodes.
5. We, then rank them in increasing order of their energy.

This concludes the process of ranking the driver nodes. The energy calculation for multiple driver node case is the key for the above algorithm to work.

As mentioned above we tried to improve the functionality of the energy algorithm by extending it to multiple node case. The energy algorithm involved the use of Grammian matrix which was simplified for the single driver node case. But the same simplification is not possible for multiple driver node case i.e. a generic formula of energy for N driver nodes could not be calculated because of the limited analytical tools available and moreover computation of Grammian for N driver node case id very expensive. Hence we could not proceed further with this approach.

Study on role of motifs on controllability of network

Certain real world networks are found to predominantly possess certain kinds of sub graphs in them, as compared to their random counterparts [4]. These sub-graphs are called as motifs. Among them, commonly found motifs in real world networks are:

1. 3-Node Feed-Forward loops (FFLs)

2. Bi-Fan loops (FFLs)

3. Bi-parallel loops

The motivation of this study was to study the effect of these micro sub-networks on the minimum number of driver nodes required to control the entire network. We hypothesize that presence of certain kind of motif in the real world network that is absent in the random counterpart, contributes to the discrepancy in number of driver nodes required, between real world network, and its random counterpart. We also hypothesize that presence of certain kind of motif would increase the fraction of required driver nodes, and certain other motifs would bring down the fraction of required driver nodes. Furthermore, the effect of these motifs is expected to be predominant in cases where their level of significance is higher.

In our experiment, we used Bi-Fan, and feedback motifs for performing the study.

The algorithm used to create Bi-Fan loop is as follows:

bflgen (number of loops required, edge list/ Adjacency Matrix of original network) :

nn = number of nodes in the original network;

ne = number of edges in the original network;

Ad = ER random control corresponding to original network

c1 = edgelist (Ad);

while (number of BFLs are less than required number)

1. Randomly pick 2 Edges e1, and e2
2. Randomly pick another edge e3 that has same source as e1 (v1), but different from e1
3. Randomly pick another edge e4 that has same destination as e3 (v3) such that
    - Source of e4 (v4) is not source to another edge that ends at destination of e1 (v2)
    - It should also be ensured that v1, v2, v3, v4 are distinct
4. Ensure that source and destinations of e2 have a degree greater than 1
5. If all the above conditions are satisfied, then create an edge between v4 and v2.
6. Check for the number of BFLs in the current network.
    - If it is less than the required number, perform next iteration.

return edgelist

Similarly, the algorithm used to create four node feedback loop is as follows:

ffblgen (number of loops required, edge list/ Adjacency Matrix of original network) :

nn = number of nodes in the original network;

ne = number of edges in the original network;

Ad = ER random control corresponding to original network

while (number of FBLs are less than required number)

1 Randomly pick 2 Edges e1, and e2 such that
    - e2 is not part of edges belonging to feedback loops
2 Randomly pick another edge e3 that has source as destination of e1 (v2)
3 Randomly pick another edge e4 that has source as destination of e3 (v3) such that
    - v1 (source of edge e1), v2, v3, v4 (destination of e4) are distinct
    - v1 and v4 do not have an edge present (with v1 as source, and v4 as destination).
4 Ensure that source and destinations of e2 have a degree greater than 1
5 If all the above conditions are satisfied, then create an edge between v4 and v2.
6 Check for the number of FBLs in the current network.
    - If it is less than the required number, perform next iteration.

return edgelist of newly created network

The MATLAB implementation of the above algorithm is annexed to this report. Similar algorithms can be developed for generating other motifs into the network manually.

For checking the number of BFLs in the network, the procedure is as follows:

countbfl(edgelist c) :

num = 0;

for i from 1 to length(edgelist)

for j from 1 to length(edgelist)

idx1 = index list of edges which have their source same as source of ith edge in c

idx2 = index list of edges which have their source same as source of jth edge in c

for k from 1 to length(idx1)

for l from 1 to length(idx2)

if destination of idx1(k,1)th edge is same as destination of jth edge and

destination of idx2(k,1)th edge is same as destination of ith edge

- Check if this quartet of these nodes is already part of BFLs list. If not, add to that list, and increment num;

return num;

For checking the number of four node FBLs in the network, the procedure is as follows:

countffbl(edgelist c) :

num = 0;

for i from 1 to length(edgelist)

for j from 1 to length(edgelist)

idx1 = index list of edges which have their source same as destination of ith edge in c

idx2 = index list of edges which have their source same as destination of jth edge in c

for k from 1 to length(idx1)

for l from 1 to length(idx2)

if destination of idx1(k,1)th edge is same as source of jth edge and

destination of idx2(k,1)th edge is same as source of ith edge

- Check if this quartet of these nodes is already part of FBLs list. If not, add to that list, and increment num;

return num;

Similar procedures have been implemented for counting number of three node feedback loops, three node feed-forward loops, and bi-parallel loops. The code for these procedures has been annexed to this report.

Results

Table 1 shown below provides numerical data for computing minimum number of driver nodes required to control the network for different types of networks for three different controls: one for real world network itself, one for a random control in which degree for each node is preserved, and the other for an ER based random control. Real world network data was obtained from [3]. Figure 2 depicts the results with bar plots corresponding to the real world network, ER random network, and degree preserved random networks, respectively.

Name of the Network	Type of the Network	N	L	Nd Real	Nd ER	Nd dgprsv
Ythan	Food web	135	601	0.5111	0.0169	0.4344
Seagrass	Food web	49	226	0.2653	0.0228	0.1957
Little Rock	Food web	183	2494	0.5410	0.0055	0.2017
Grassland	Food web	88	137	0.5227	0.0169	0.4344

c. Elegans	Neuronal	297	2345	0.1650	0.0034	0.0975
s838	Electronic Circuits	512	819	0.2324	0.2955	0.1930
s420 (11)	Electronic Circuits	252	399	0.2341	0.1953	0.2984
s208 (10)	Electronic Circuits	122	189	0.2377	0.3018	0.1980
Prison Inmate	Trust	67	182	0.1340	0.1067	0.1470
Manufacturing	Intra Organizational	77	2228	0.0130	0.0130	0.0130
Consulting	Intra Organizational	46	879	0.0435	0.0435	0.0218
Freeman-I	Intra Organizational	34	695	0.0294	0.0294	0.0294
Freeman-II (4)	Intra Organizational	34	830	0.0294	0.0294	0.0294
Escherichia coli	Metabolic	2275	5763	0.3824	0.1298	0.2176
Saccharomyces cerevisiae (18)	Metabolic	1511	3833	0.3289	0.1291	0.2065
Caenorhabditis elegans (19)	Metabolic	1173	2864	0.3018	0.2004	0.1435
Yeast-2 (15)	Regulatory	688	1079	0.8212	0.8110	0.3034
EC-2 (16)	Regulatory	418	519	0.7512	0.3834	0.7521

Table 1: Summary of Driver node results for real world network and

the corresponding random controls

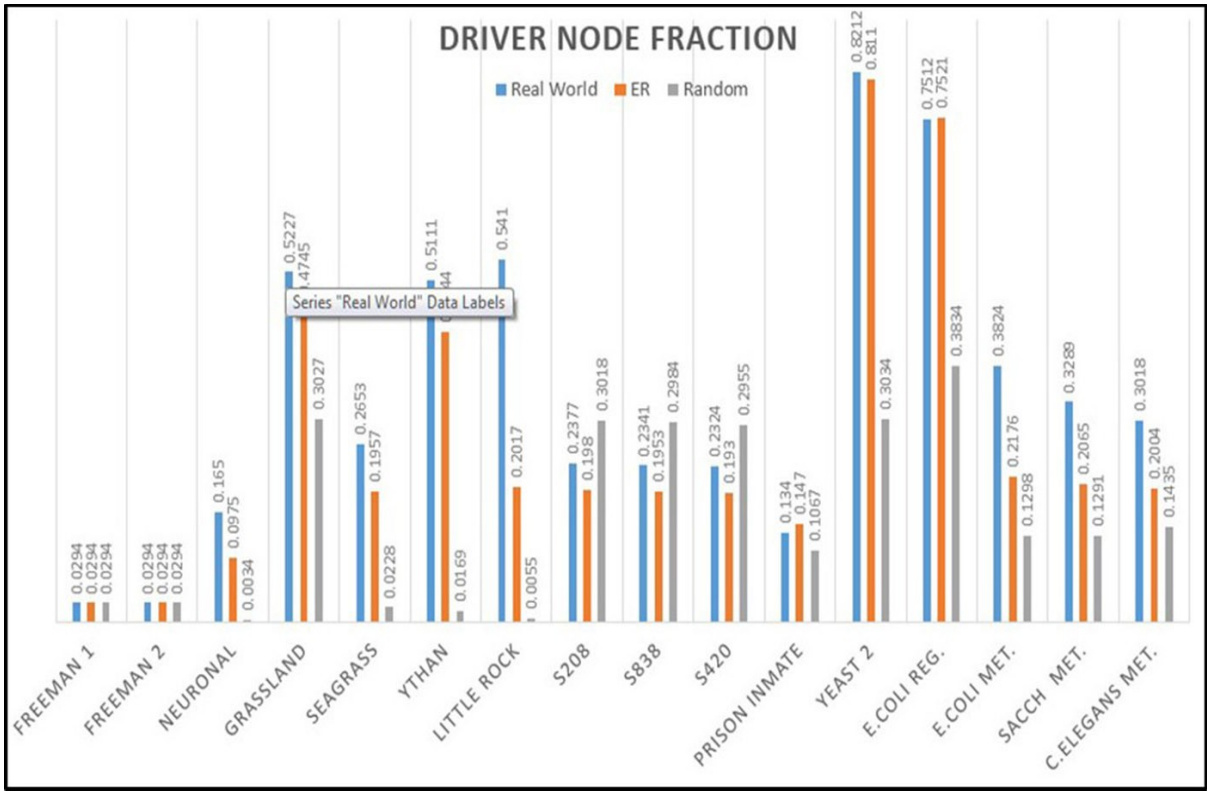


Figure 2: Summary of the driver node result

The results that are reproduced in our work is significant when it comes to drawing key conclusions about the dependency of number and nature of driver nodes on the network architecture. First, the average degree of driver nodes for a given network is always found to be less than the average degree of the entire network. This shows that driver nodes tend to avoid hubs, and prefer peripheral nodes. This result is shown in figure 3. Second, the minimum number of driver nodes required to control a real world network, Ndreal is always greater than its ER network counterpart. This implies that a heterogeneous, and scale free real world network will have a higher value of minimum number of required driver nodes, compared to a more-homogeneously distributed ER networks (NdER). This is illustrated in figure 4 a. The final and the most important remark on the driver nodes is that the number of driver nodes required to control a given network is only dependent on degree distribution of the network, and is independent of structure of the network, i.e., the arrangement of individual edges. This is depicted in figure 4 b, where we can observe that Ndreal is almost equal to minimum number of driver nodes required in degree preserved random control (Ndrand-degree) for all the networks. Note that the data for random networks is an average taken from an ensemble of thousand networks in case of ER network, and four hundred in case of degree preserved random networks. Also, in each case of degree preserved random network, edges were rewired for 1.5 times the total number of edges present in the network.

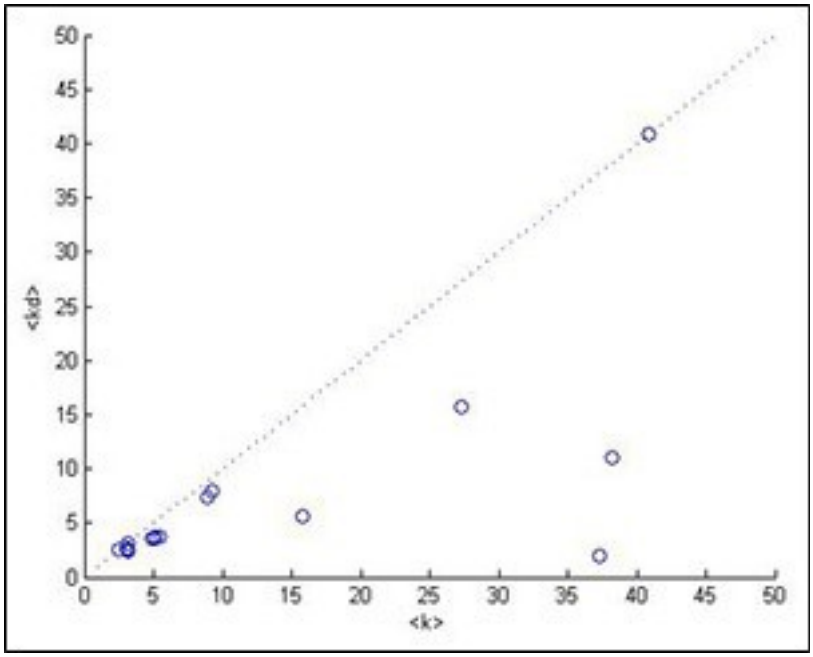
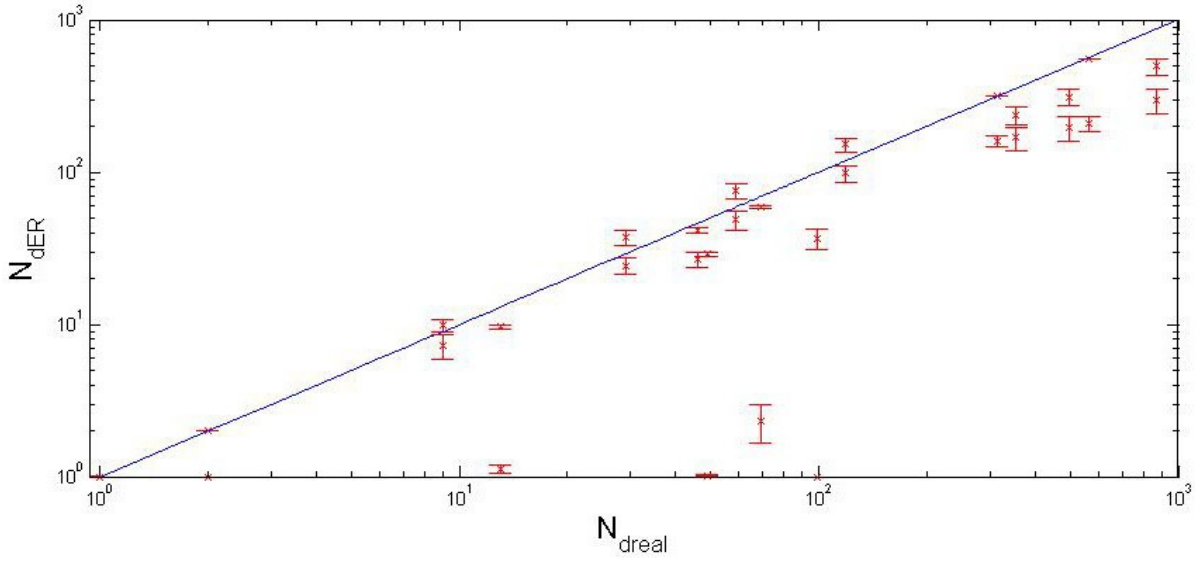


Figure 3: Plot between average degree of driver nodes, and that of entire network for various Networks

\

A



B

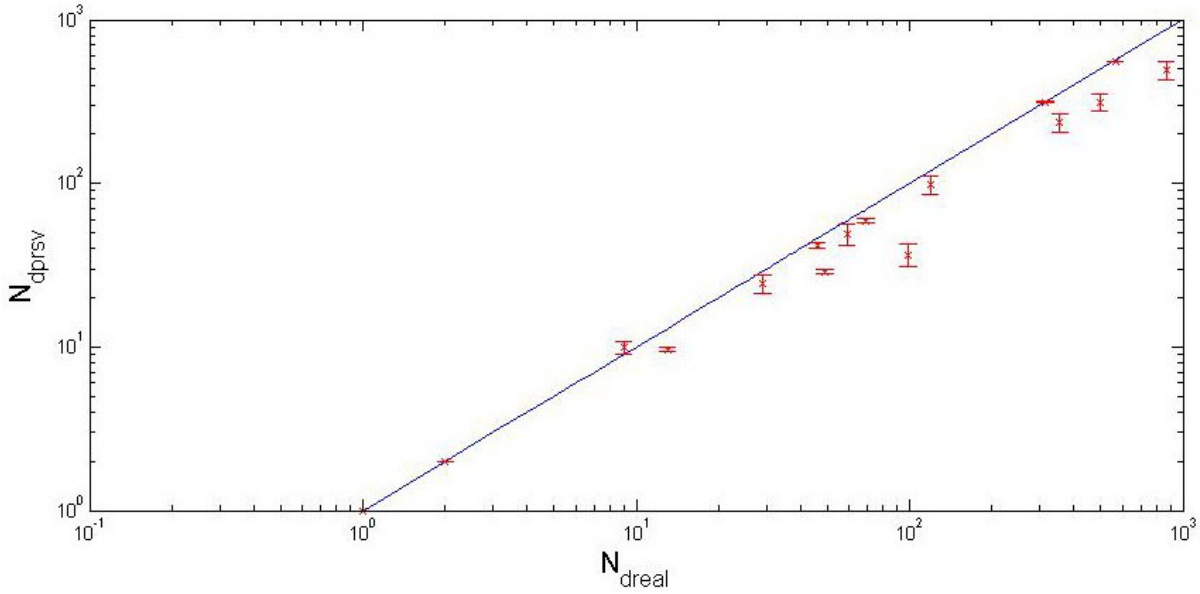
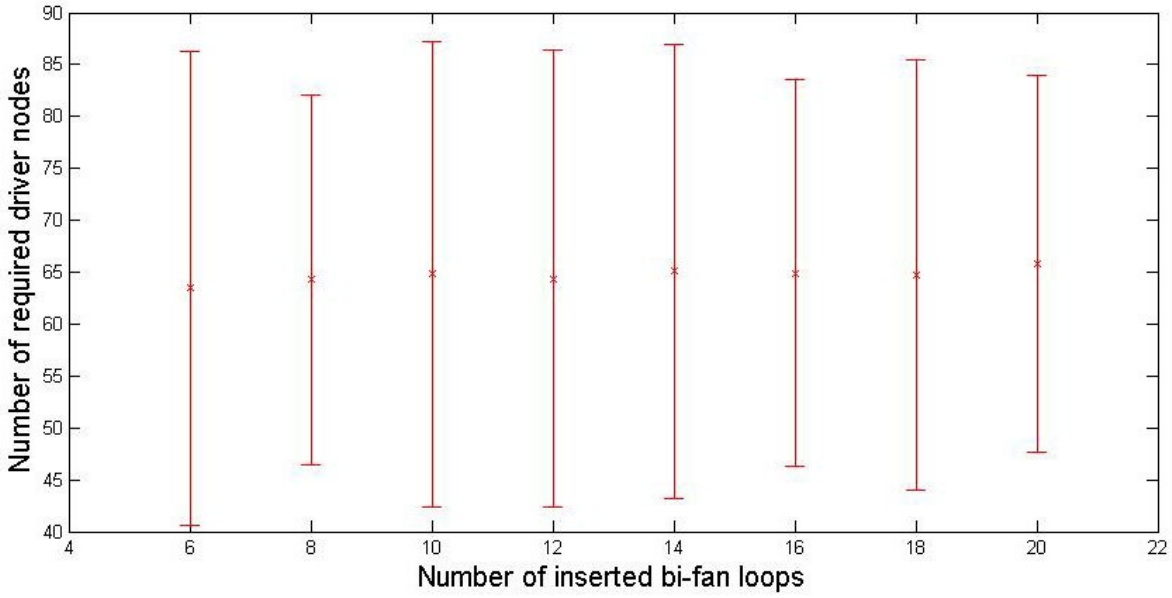


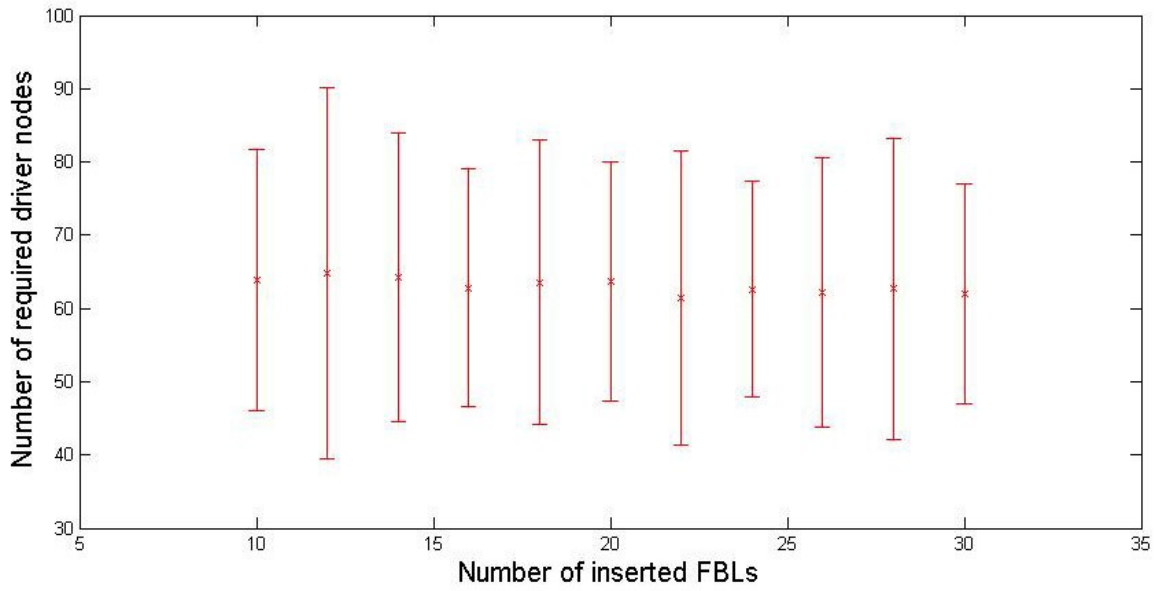
Figure 4: Plot of Number of driver nodes for real world network versus ER Network, and degree preserved random network

For studying the effect of driver nodes on controllability, initially, we create a random control corresponding to real world network, and calculate minimum number of driver nodes required to control this network. After this, we insert a number of particular type of motifs into this network, and calculate the minimum number of driver nodes required for this network. This experiment is repeated for number of times over a range of number of inserted motifs. One significant assumption that we made during this experiment is that, other types of motifs are not created as a side effect when we insert one particular kind of motif. This assumption might have had a significant implication on the obtained results.

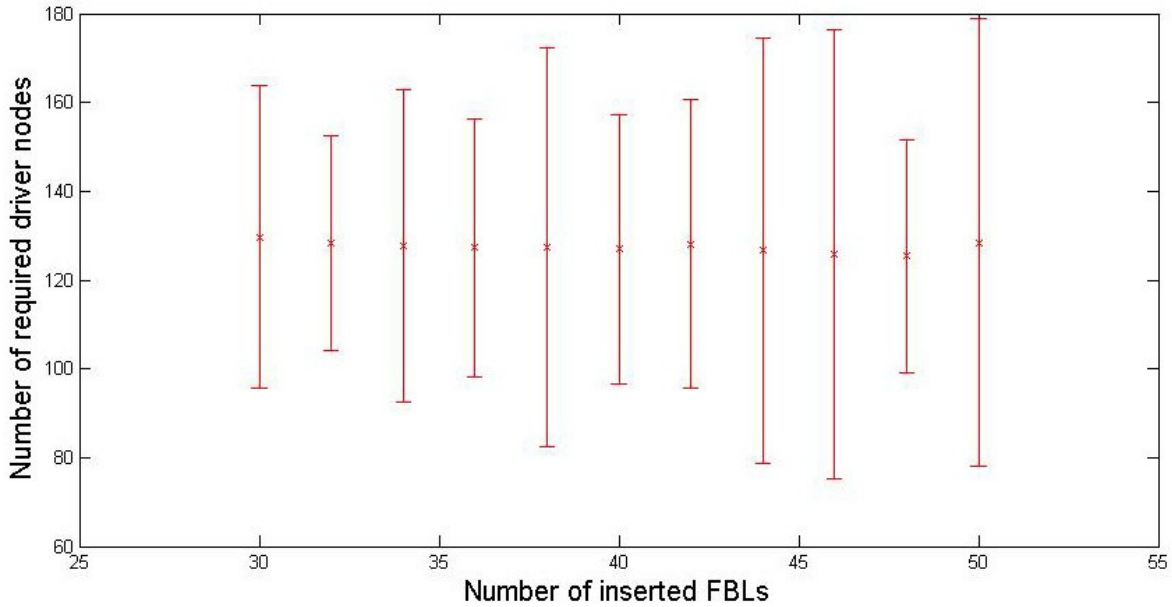
A



B


Number of required driver nodes
Number of inserted FBLs
100
90
80
70
60
50
40
30
5
10
15
20
25
30
35

C


Number of required driver nodes
Number of inserted FBLs
180
160
140
120
100
80
60
25
30
35
40
45
50
55

D

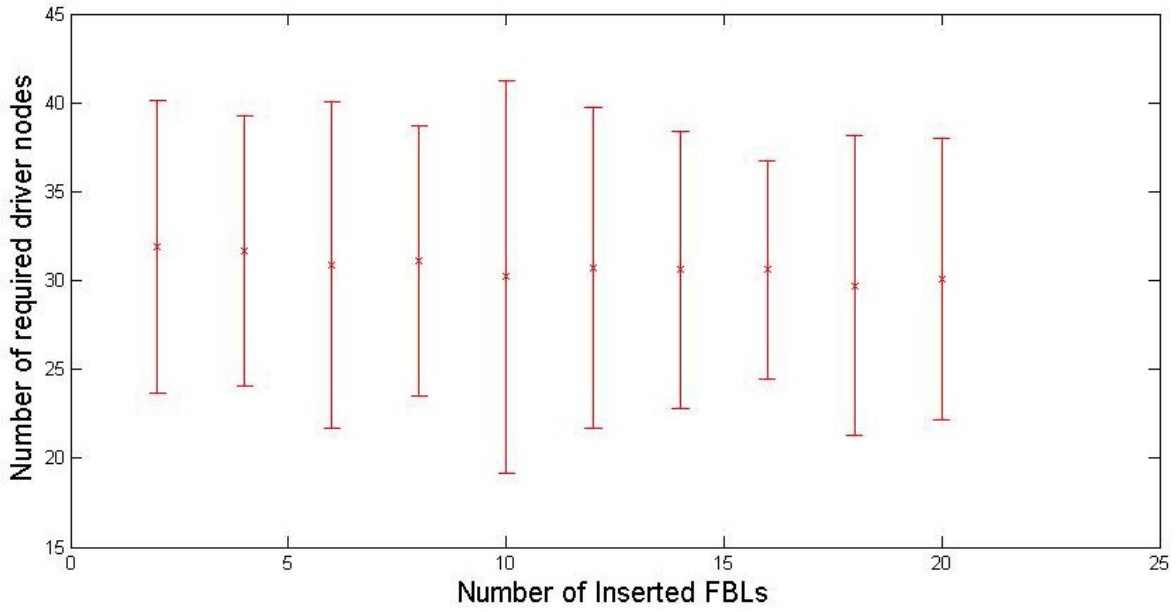


Figure 5: The first plot corresponds to minimum number of required driver nodes versus number of inserted bi-fan loops in an ER network corresponding to s420 network. Plots in b, c, and d correspond to minimum number of required driver nodes versus number of inserted three node feedback loops in an ER network corresponding to s420, s838 and s208 networks respectively. Note that the obtained results were average of 50 networks, and the error bar for each individual points is also depicted in the plots.

Figure 5 a shows the result of inserting bi-fan loop into the ER network of s420 (electronic circuit network). The number of bi-fan loop in the original network is 10. Therefore, 6 to 20 (both inclusive) bi-fan loops were inserted in the corresponding ER network, with a sampling of 2. The curve between minimum number of required nodes and number of inserted bi-fan loops was plotted, and the minima was obtained at Nbfl = 6, though the curve obtained was having insignificant variation in the required number of driver nodes. Nonetheless, it is worth to note that the value of minimum number of driver nodes required in the ER network with inserted bi-fan loops is closer to real world network, compared to ER network with no feedback loop.

Figure 5 b shows the result of inserting four node feedback loops into the ER network of s420 (electronic circuit network). The number of feedback loop in the original network is 40. Therefore, 30 to 50 (both inclusive) feedback loops were inserted in the corresponding ER network, with a sampling of 2. The curve between minimum number of required nodes and number of inserted feedback loops was plotted, and the minima was obtained at Nffl = 36, though the curve obtained was having insignificant variation in the required number of driver nodes. Nonetheless, it is worth to note that the value of minimum number of driver nodes required in the ER network with inserted feedback loops is closer to real world network, compared to ER network with no feedback loop.

Similarly, Figure 5 c shows the result of inserting four node feedback loops into the ER network of s420 (electronic circuit network). The number of feedback loop in the original network is 20. Therefore, 10 to 50 (both inclusive) feedback loops were inserted in the corresponding ER network, with a sampling of 5. The curve between minimum number of required nodes and number of inserted feedback loops was plotted, and the minima was obtained at Nffl = 25, though the curve obtained was having insignificant variation in the required number of driver nodes. Nonetheless, it is worth to note that the value of minimum number of driver nodes required in the ER network with inserted feedback loops is closer to real world network, compared to ER network with no feedback loop. Similar result obtained with s208 electronic circuit network is shown in Figure 5 d.

Conclusion

The minimum number of driver nodes required to control a real world network depends on the degree of homogeneity of the network. The more heterogeneous the degree distribution is, more is the proportion of required minimum driver nodes. This is evident from the results where we can observe that a more homogeneous ER network required lesser number of driver nodes to be controlled than its real world counterpart. Thus, the degree distribution plays an important role in control of the entire network.

The results also provides some significant lead in the role of motifs in the control, which can be taken forward in a promising direction. Though the current results are obscure and insufficient for concluding anything about the role of motifs, it is sufficient with the current status for us to conclude that motifs do affect the controllability of the entire network.

Future work

The results produced by [1] is significant, and can be used in many directions. First, from the computational analysis, we can observe that in general, random ER networks require lesser number of driver nodes needed to control the entire network. But, there are exceptions where the original network has significantly lesser number of minimum driver nodes required to control the entire network. Therefore, a potentially significant heuristics for guiding study of driver nodes is to analyse the network architecture in these kinds of network, and identify the rationale for occurrence of such exceptional cases. The second possible direction (which is slightly complex) is to explore networks that are part of discrete and finite space. In these cases, it might happen that the controllability matrix used for testing whether network is controllable need not be full rank, since we are concerned only about discrete points in the space. The possibility of reducing number of driver nodes further down could be realizable in these cases.

We can also perform analysis of driver nodes on the given real world networks to interpret its functional properties in the network. For example, we can find the set of driver nodes in a protein-protein interactome network, and examine whether they form a subset of critical proteins in the network. This kind of study would possibly reduce significant overhead that occurs when we study property of each individual node in the network for some applications e.g. drug target discovery. Finally, we can explore a means for developing a procedure to rate the importance of each driver node used to control the network. This would be important, especially when the network is subject to deletion of nodes (random or targeted), and deletion of a driver node might impact on control of the entire network.

Another interesting direction of research on driver nodes is to perform comprehensive study on role of motifs on the required minimum number of driver nodes. For instance, all the networks can be experimented by insertion of motifs reported in [4]. Furthermore, the effect of inserting a combination of distinct motifs on the minimum required number of driver nodes can be explored.

REFERENCES:

[1] Liu, Yang-Yu, Jean-Jacques Slotine, and Albert-László Barabási. "Controllability of complex networks." Nature 473.7346 (2011): 167-173.

[2] "Structure induction by lossless graph compression", Leonid Peshkin, In Proc. of Data Compression Conf. DCC, 2007

[3] The real world network data used in the project is available at www.boseinst.ernet.in/soumen/Network_Controllability_Datasets.html

[4] Milo, Ron, et al. "Network motifs: simple building blocks of complex networks." Science 298.5594 (2002): 824-827.

[5] Yan, Gang, et al. "Controlling complex networks: How much energy is needed?" Physical review letters 108.21 (2012): 218703.

[6] Bollobás, Béla. Random graphs. Springer New York, 1998.

[7] Maslov, Sergei, and Kim Sneppen. "Specificity and stability in topology of protein networks." Science 296.5569 (2002): 910-913.

MATLAB IMPLEMENTATION OF ALGORITHMS

1. **Code for Computing driver nodes**

```
function[driver,avg_val] = undirected_driverNodes_adj(Ad)
% driver is the set of driver nodes
% avg_val is 1X2 matrix, with average degree of whole network as its,
% first entry, and average degree of driver nodes as its
% second entry. Input is the Adjacency matrix of original
% network.
[old,~] = size(Ad);
badj = convert2bipartite(Ad);
a = card_match(badj);
sum1 = length(nonzeros(Ad));
pos = length(Ad);
avg = 2/pos*(sum1);
[~,n]=size(a);
driver = zeros(n);
for i=old+1:n
    if a(1,i)==0
        driver(1,i)=(i - old);
    end
end
driver=driver(driver~=0);
if ~isempty(driver)
  sum1 = 0;
for i = 1:length(driver)
 sum1=sum1+length(nonzeros(Ad(driver(i,1),:))) +
length(nonzeros(Ad(:,driver(i,1))));
```

```
end
pos = length(driver);
avg_driver = 1/pos * sum1;
avg_val = [avg avg_driver];
else
    avg_val = [avg avg];
end

end


function [ badj ] = convert2bipartite( adj )
% Input of the function is Adjacency of original network.
% Output of the function is a bi-partite network corresponding
% to the original graph
[m,n] = size(adj);
badj = zeros(2*m,2*n);
for i = 1:m
    for j = 1:n
        if adj(i,j) ~= 0
            badj(i,n+j) = adj(i,j);
            badj(n+j,i) = adj(i,j);
        end
    end
end

end
```

2. Code for counting motifs

2.1 Feed forward loops

```
function[num,list1] = countffl(c)
    % Input is edge list of original network
    % If a set of three nodes have connections v1-->v2-->v3
    % and in addition, if there is an edge from v1 to v3, then
    % these nodes form a feed forward loop
    % Output 'num' corresponds to number of FFLs in the network
    % Output 'list1' corresponds to set of edges v1-->v2,v2-->v3
    % and v1-->v3 (of all nodes that form FFLs)
        num = 0;
        list1 = zeros(1,2);
        d = c(:,1);
        for i = 1:length(c)
          idx = find(d == c(i,2));
           for j = 1:length(idx)
              for k = 1:length(c)
                if c(k,1) == c(i,1) && c(k,2) == c(idx(j,1),2) && k ~= i
                   num = num + 1;
                   list1(3*(count-1) + 1,1) = c(i,1);
                   list1(3*(count-1) + 1,2) = c(idx(j,1),2);
                   list1(3*(count-1) + 2,1) = c(i,1);
                   list1(3*(count-1) + 2,2) = c(i,2);
                   list1(3*(count-1) + 3,1) = c(i,2);
                   list1(3*(count-1) + 3,2) = c(idx(j,1),2);
               end
             end
           end
```

```
            end
     end
```

2.2 Three node Feedback loop

```
function[num,list2] = countfb(c)
% Input is edge list of original network
% If a set of three nodes have connections v1-->v2-->v3
% and in addition, if there is an edge from v3 to v1, then
% these nodes form a feed forward loop
% Output 'num' corresponds to number of FFLs in the network
% Output 'list1' corresponds to set of edges v1-->v2,v2-->v3
% and v3-->v1 (of all nodes that form FFLs)
    num = 0;
    d = c(:,1);
    list2 = zeros((num+1),3);
    for i = 1:length(c)
    idx = find(d == c(i,2));
    for j = 1:length(idx)
        for k = 1:length(c)
            if c(k,2) == c(i,1) && c(k,1) == c(idx(j,1),2)
               ch = check1(sort([c(i,2),c(idx(j,1),2),c(i,1)]),list2);
               if ch == 0
                   num = num + 1;
                   list = sort([c(i,2),c(idx(j,1),2),c(i,1)]);
                   list2(num,:) = list;
                   break;
               end
            end
        end
     end
     end
end
```

2.3 Four node Feedback loop

```
function[num,list,list1] = countffbl(c)
% Input is edge list of original network
% If a set of three nodes have connections v1-->v2-->v3-->v4
% and in addition, if there is an edge from v4 to v1, then
% these nodes form a four node feed forward loop
% Output 'num' corresponds to number of FFLs in the network
% Output 'list' corresponds to sorted order of  v1,v2,v3,v4
% Output 'list1' corresponds to v1,v2,v3,v4
   num = 0;
   list = zeros(1,4);
   list1 = zeros(1,4);
   d = c(:,1);
   for i = 1:length(c)
       for j = 1:length(c)
            idx1 = find(d == c(i,2));
            idx2 = find(d == c(j,2));
            for k = 1:length(idx1)
                for l = 1:length(idx2)
                    if c(idx1(k,1),2) == c(j,1) && c(idx2(l,1),2) == c(i,1)
&& c(i,1) ~= c(j,1) && c(i,2) ~= c(j,2) && c(i,1) ~= c(j,2) && c(j,1) ~=
c(i,2)
                        ch =
check1(sort([c(i,1),c(i,2),c(j,1),c(j,2)]),list);
```

```
                        if ch == 0
                            num = num + 1;
                            list(num,:) =
sort([c(i,1),c(i,2),c(j,1),c(j,2)]);
                            list1(num,1) = c(i,1);
                            list1(num,2) = c(i,2);
                            list1(num,3) = c(j,1);
                            list1(num,4) = c(j,2);
                        end
                    end
                end
            end
        end
    end
end
```

- Following procedure outputs edges corresponding to Feedback loops:

```
function[edg] = fbedl(list)
    [len1,len2] = size(list);
    edg = zeros(len1*(len2-1),1);
        for i = 1:len1
            k = (i-1)*len2;
            for j = 1:len2
                l = rem(j+1,len2);
                edg(k + j,1) = list(i,j);
                if l == 0
                    edg(k + j,2) = list(i,len2);
                else
                    edg(k + j,2) = list(i,l);
                end
            end
        end
end
```

2.4 Bi-parallel loops

```
function[num,list,list1] = countbp(c)
% Input is edge list of original network
% If a set of three nodes have connections v1-->v2-->v4, and v1-->v3-->v4
% and in addition, if there is an edge from v4 to v1, then
% these nodes form a four node feed forward loop
% Output 'num' corresponds to number of FFLs in the network
% Output 'list' corresponds to sorted order of  v1,v2,v3,v4
% Output 'list1' corresponds to v1,v2,v3,v4
   num = 0;
   list = zeros(1,4);
   list1 = zeros(1,4);
   d = c(:,1);
   for i = 1:length(c)
       for j = 1:length(c)
            idx1 = find(d == c(i,1));
            idx2 = find(d == c(i,2));
            for k = 1:length(idx1)
                for l = 1:length(idx2)
                    if c(idx1(k,1),2) == c(j,1) && c(idx2(l,1),2) == c(j,2)
&& c(i,1) ~= c(j,1) && c(i,2) ~= c(j,2) && c(i,1) ~= c(j,2) && c(j,1) ~=
c(i,2)
                        ch =
check1(sort([c(i,1),c(i,2),c(j,1),c(j,2)]),list);
```

```
                        if ch == 0
                            num = num + 1;
                            list(num,:) =
sort([c(i,1),c(i,2),c(j,1),c(j,2)]);
                            list1(num,1) = c(i,1);
                            list1(num,2) = c(i,2);
                            list1(num,3) = c(j,1);
                            list1(num,4) = c(j,2);
                        end
                    end
                end
            end
        end
    end
end
```

Note that separate procedure has to be written for extracting edges belonging to bi-parallel loops. This can be done by using extracted node quartets.

2.5 Bi-Fan loops

```
function[num,list,list1] = countbp(c)
% Input is edge list of original network
% If a set of three nodes have connections v1-->v2, v1-->v3 and v2-->v3,
v2-->v4
% and in addition, if there is an edge from v4 to v1, then
% these nodes form a four node feed forward loop
% Output 'num' corresponds to number of FFLs in the network
% Output 'list' corresponds to sorted order of  v1,v2,v3,v4
% Output 'list1' corresponds to v1,v2,v3,v4
    num = 0;
    list = zeros(1,4);
    list1 = zeros(1,4);
    d = c(:,1);
    for i = 1:length(c)
        for j = 1:length(c)
             idx1 = find(d == c(i,1));
             idx2 = find(d == c(i,2));
             for k = 1:length(idx1)
                 for l = 1:length(idx2)
                     if c(idx1(k,1),2) == c(j,1) && c(idx2(l,1),2) == c(j,2)
&& c(i,1) ~= c(j,1) && c(i,2) ~= c(j,2) && c(i,1) ~= c(j,2) && c(j,1) ~=
c(i,2)
                         ch =
check1(sort([c(i,1),c(i,2),c(j,1),c(j,2)]),list);
                         if ch == 0
                             num = num + 1;
                             list(num,:) =
sort([c(i,1),c(i,2),c(j,1),c(j,2)]);
                             list1(num,1) = c(i,1);
                             list1(num,2) = c(i,2);
                             list1(num,3) = c(j,1);
                             list1(num,4) = c(j,2);
                         end
                     end
                 end
             end
         end
    end
end
```

```
function[list1] = bfedl(list)
% bfedl outputs edge list corresponding to
% bi-fan loops.
% Input is the node quartet(output named list1(of countbfl))
   [m,~] = size(list);
   list1 = zeros(4*m,2);
   for i = 1:m
       list1(4*(i-1) + 1, 1) = list(i,1);
       list1(4*(i-1) + 1, 2) = list(i,2);
       list1(4*(i-1) + 2, 1) = list(i,3);
       list1(4*(i-1) + 2, 2) = list(i,4);
       list1(4*(i-1) + 3, 1) = list(i,1);
       list1(4*(i-1) + 3, 2) = list(i,4);
       list1(4*(i-1) + 4, 1) = list(i,3);
       list1(4*(i-1) + 4, 2) = list(i,2);
   end
end
```

3. Random network generation

3.1 ER Network

```
function [mUnd] = gAdjER_weights(nn,ne,nw,clen)
% Here nn stands for number of nodes in the network
% ne stands for number of edges(zero for weighted networks)
% nw,clen corresponds to number of weights, and the column corresponding
% to weights
% For generation of unweighted graph, set nw, and clen to be zero
%
m = zeros(nn);
[a,~] = size(nw);
if a ~= 1
    [d,~] = size(nw);
    len = zeros(d,1);
    for i = 1:d
        [len(i,1),~] = size(find(clen == nw(i,1)));
    end
    ident = eye(nn);
    for i = 1:d
        nd_idx = find(~ident);
        con = randperm(numel(nd_idx),len(i,1));
        m (nd_idx(con)) = nw(i,1);
        ident = ident + m;
    end
    mUnd = m;
else
    ident = eye(nn);
    nd_idx = find(~ident); % Indices of non-diag elements
    con = randperm(numel(nd_idx),ne); % Pick random elements
    m( nd_idx(con) ) = 1;
    mUnd = m;
end
end
```

3.2 Degree preserved random network generation

```
function[c1] = deg_prsv(c)
```

```
% creates a degree preserved network
% c1 is the edge list corresponding to a degree
% preserved random network.
% checks whether selected nodes have an edge already present
% between them, and rewires the existing edge to these two nodes
% if they do not have an edge. The selected edge is removed as
% long as it does not disconnect one of the edges from rest of the
% network.
   [len,~] = size(c(:,1));
   c1 = c;
   i = 0;
   while i < 1.5*len
    idx = randperm(len,2);
           val1 = [c1(idx(1),1),c1(idx(2),2)];
           val2 = [c1(idx(2),1),c1(idx(1),2)];
           l = c1(:,1:2);
           check = 0;
           for a = 1:length(l)
               for b = 1:2
                  if (l(a,1) == val1(1,1) && l(a,2) == val1(1,2)) ||
(l(a,1) == val2(1,1) && l(a,2) == val2(1,2))
                      check = 1;
                  end
               end
           end
           if check == 0
              temp = c1(idx(1),1);
              c1(idx(1),1) = c1(idx(2),1);
              c1(idx(2),1) = temp;
              i = i+1;
           end
   end
```

- **Code for finding average number of driver nodes over an ensemble of degree preserved random control**

```
% This is the script for calculating average number of driver
% nodes in case of degree preserved random control.
% Each time, the random control operates for 400 times,
% and number of driver nodes are calculated for each network
% This code outputs the average fraction of driver nodes (given by frac),
% and the variance of the fraction of driver nodes among the computed
% networks.
len1 = zeros(400,1);
for k = 1:400
c1 = deg_prsv(c);
c1 = deg_prsv(c);
Ad = code1(c1);
[driver1,~] = undirected_driverNodes_adj(Ad);
driver1 = driver1';
len1(k,1) = length(driver1);
end
sum1 = 0;
for i = 1:length(len1)
    sum1 = sum1 + len1(i,1);
end
sum1 = sum1/length(len1);
frac = sum1/length(Ad);
if frac == 0
    frac = 1/length(Ad);
```

```
end
avg = sum1;
dpr_var = var(len1);
clear sum1;
clear driver1;
clear i;
clear k;
clear n;
clear Ad;
clear c1;
clear len1;
```

Code for manual creation of Feedback loops

```
function[c,cur_val] = fblgen(val,c)
%c is input/output edgelist
% cur_val is the number of feedback loops generated
% this might be different from val, the input
% corresponding to required number of feedback
% loops because addition of an edge may result
% in generation of more than one feedback loop
% simultaneously.
nn = length(unique(c));
ne = length(c);
Ad = gAdjER_weights(nn,ne,0,0);
c = edggen(Ad);
c = c(:,1:2);
[n,~] = countfb(c);
if n > val
    vals = (n + 1);
else
    vals = val;
end
[cur_val,list] = countfb(c);
list1 = fbedl(list);
    while cur_val < vals
        ind = randperm(ne,2);
        ch1 = check1(c(ind(1,2),:),list1);
        if ch1 == 1
            continue;
        end
        d = c(:,1);
        idx = find(d == c(ind(1,1),2));
        [len,~] = size(idx);
        if len == 0
            continue;
        end
        ind1 = randperm(len,1);
        ch =
check1(sort([c(ind(1,1),1),c(ind(1,1),2),c(idx(ind1),2)]),list);
        if ch == 1
            continue;
        end
        v = [c(idx(ind1),2),c(ind(1,1),1)];
        ch = check1(v,c);
        if ch == 0 && c(ind(1,1),1) ~= c(idx(ind1),2)
          c(ind(1,2),1) = c(idx(ind1),2);
          c(ind(1,2),2) = c(ind(1,1),1);
        end
        [cur_val,list] = countfb(c);
        list1 = fbedl(list);
```

```
    end
end
```

Code for generation of Bi-Fan loops

```
function[c,cur_val] = bflgen(val,c)
%c is input/output edgelist
% cur_val is the number of feedback loops generated
% this might be different from val, the input
% corresponding to required number of Bi-Fan
% loops because addition of an edge may result
% in generation of more than one Bi-Fan loop
% simultaneously.
nn = length(unique(c));
ne = length(c);
Ad = gAdjER_weights(nn,ne,0,0);
c = edggen(Ad);
c = c(:,1:2);
n = countffbl(c);
if n > val
    vals = (n + 1);
else
    vals = val;
end
[cur_val,list,list2] = countbfl(c);
list1 = bfedl(list2);
if cur_val < vals
    while cur_val < vals
        ind = randperm(ne,2);
        ch1 = check1(c(ind(1,2),:),list1);
        if ch1 == 1
            continue;
        end
        d = c(:,1);
        e = c(:,2);
        idx = find(d == c(ind(1,1),1));
        [len,~] = size(idx);
        ind1 = randperm(len,1);
        if ind(1) == idx(ind1(1))
            continue;
        end
        idx1 = find(e == c(idx(ind1(1,1)),2));
        [len,~] = size(idx1);
        ind4 = randperm(len,1);
        if idx1(ind4(1)) == idx(ind1(1))
            continue;
        end
        cur_list =
[c(ind(1,1),1),c(ind(1,1),2),c(idx(ind1(1)),2),c(idx1(ind4(1),1))] ;
        if length(unique(cur_list)) < 4
            continue;
        end
        ch =
check1(sort([c(ind(1,1),1),c(ind(1,1),2),c(idx(ind1(1)),2),c(idx1(ind4(1),1
))]),list);
        if ch ~= 0
            continue;
        end
        ind2 = find(c == c(ind(1,2),2));
        ind3 = find(c == c(ind(1,2),1));
        if length(ind2) == 1 || length(ind3) == 1
```

```
            continue;
        end
        c(ind(1,2),1) = c(idx1(ind4(1)),1);
        c(ind(1,2),2) = c(ind(1,1),2);
        [cur_val,list] = countbfl(c);
        list1 = fbedl(list);
    end
end
end
```