NIT Silchar

Department of Computer Science and Engineering

Social Network Analysis

CS 331

B.Tech (CSE) Sixth Semester

(January 2022-June 2022)

<u>MINI – PROJECT REPORT</u>

Mini-Project Title

Using information diffusion for incremental community detection in dynamic networks

Research Papers Implemented:

1. Identifying communities in dynamic networks using information dynamics (2020)
2. Community detection in dynamic networks via a spreading process (2019) (DC)

Team-Member Details

Team Leader: Abhishek Bharadwaz **1912106**

Sourabh Shah **1912158**

Ayesha Nasim **1912177**

Aditya Agarwal **1912104**

Aditya Soni **1912101**

Individual Contribution

Sourabh Shah (1912158):

- Implementation and coding “Community detection in dynamic networks via a spreading process”
- Implementation and coding “Identifying communities in dynamic networks using information dynamics”
- Report Formation

Ayesha Nasim (1912177):

- Implementation and coding “Community detection in dynamic networks via a spreading process”:
- Implementation and coding “Identifying communities in dynamic networks using information dynamics”
- Report Formation

Abhishek Bharadwaz (1912106):

- Implementation and coding “Identifying communities in dynamic networks using information dynamics”
- Report Formation

Aditya Agarwal (1912104):

- Implementation and coding “Identifying communities in dynamic networks using information dynamics”
- Report Formation

Aditya Soni (1912101):

- Implementation and coding “Identifying communities in dynamic networks using information dynamics”
- Report Formation

Abstract/Summary

1. Community detection in dynamic networks via a spreading process

1.1 Introduction:

In the algorithm, an SIR-like spreading process has been used. The Susceptible-Infectious-Recovered-like process has been chosen on the observation that during epidemic spreading, similarly behaving agents tend to self-organize into the same cluster. Hence, where nodes that easily infect one another are considered closer together, and are similar**.**

1.2 The Community Detection Technique:

The technique for community detection comprises three components:

i) A spreading process to quantify the similarity between any pair of nodes in the

Network

ii) A greedy clustering algorithm to partition the network into communities

iii) An extension to account for the network's temporal evolution in an effective

manners.

1.3 The spreading Process

A SIR model is adopted in which each individual might be in one of the following

three states: susceptible S, infectious I, or recovered R.

S individuals may get infected due to the physical contact with their I neighbors, and this occurs with infection probability λ. Individuals may recover with probability μ.

A Markov model for the SIR spreading process is characterized by the following transition probabilities:

$$
\begin{aligned}
p_i^{[t]}(S,\tau+1) &= p_i^{[t]}(S,\tau)\prod_{j=1}^{N}(1-\lambda A_{ij}^{[t]}p_j^{[t]}(I,\tau)),\\
p_i^{[t]}(I,\tau+1) &= p_i^{[t]}(S,\tau)\\
&\quad\times\left[1-\prod_{j=1}^{N}(1-\lambda A_{ij}^{[t]}p_j^{[t]}(I,\tau))\right]\\
&\quad+(1-\mu)p_i(I,t),\\
p_i^{[t]}(R,\tau+1) &= \mu p_i^{[t]}(I,\tau)+p_i^{[t]}(R,\tau),
\end{aligned}
$$

$p^{[t]}_i$ (s, τ) denotes the probability that node vi is in state S ∈ {S, I,R} at time τ.

Similarly,

$P^{[t]}_I$ (I, τ) denotes the probability that node vi is in state I ∈ {S, I, R} at time τ.

And

$P^{[t]}_I$ (R, τ) denotes the probability that node vi is in state R ∈ {S, I, R} at time τ.

1.4 The Clustering:

Every network node vi with corresponding vector p[t]i, the nodes have been clustered,

first by collecting all the information into an N × N partitioning matrix, P[t], at time t whose elements are determined by the equation below:

$$
P_{ij}^{[t]} = \begin{cases} \mathbf{p}_i^{[t]}(j), & i \neq j,\\ \dfrac{1}{N-1}\sum_{j\neq i}\mathbf{p}_i^{[t]}(j), & i = j. \end{cases}
$$

The similarity was determined by Pearson Correlation

$$
\mathrm{Cor}_{\mathrm{Pearson}}(v_i,v_j) = \frac{1}{N}\sum_{k=1}^{N}\frac{(P_{ik}^{[t]}-\mu_i^{[t]})(P_{jk}^{[t]}-\mu_j^{[t]})}{\sigma_i^{[t]}\sigma_j^{[t]}},
$$

where

$$
\begin{aligned}
\mu_i^{[t]} &= \frac{1}{N}\sum_{k=1}^{N}P_{ik}^{[t]},\\
\sigma_i^{[t]} &= \sqrt{\frac{1}{N}\sum_{k=1}^{N}(P_{ik}^{[t]}-\mu_i^{[t]})^2}.
\end{aligned}
$$

We finally calculate a distance function, which essentially turns the community detection problem into a clustering problem.

$$d(v_i, v_j) = 1 - \mathrm{Cor}_{\mathrm{Pearson}}(v_i, v_j).$$

For Clustering purposes, a K Means algorithm has been used.

We start clustering using no. of clusters, k=2 and calculate the clusters for k-1 and k+1. We then calculate the modularity that we obtain for the given clusters.

We stop our search when we find the local maxima of modularity and use that k value in the K Means algorithm.

2. Identifying communities in dynamic networks using information dynamics

2.1 Introduction:

DCDID(Dynamic Community Detection Based on Information Dynamics) uses the information dynamics model to mimic information transmission among nodes, with the goal of improving community recognition efficiency by filtering out the unchanging sub-graph. This model is based on the features to automatically display the

community structure by simulating information exchange between members. The authors devised a novel technique based on information dynamics for gaining insights regarding community division in dynamic networks, with the core premise being to examine an accommodative dynamical system and investigate its information dynamics over time.

In particular, in an interpersonal network, people with similar interests or features are more likely to interact with others, and the propagation of information between them tends to be more frequent. With the diffusion and interaction of information, people in the same community have almost the equivalent amount of information, whereas those in diverse communities have different amounts of information. Over time, the information dynamics on the network reaches a steady state, and the communities can be naturally uncovered by counting the amount of information of nodes in the network.

2.2 Dynamic community detection:

DCDID uses a batch processing technique to incrementally uncover communities in dynamic networks. It employs the information dynamics model to simulate the exchange of information among nodes and aims to improve the efficiency of community detection by filtering out the unchanged subgraph.

The three phases of DCDID are: initial community structure detection, computation of altered subgraphs, and incremental community identification.

a. **Initial Community Structure Detection:** The community partition of the network at time slice T0 is the original community structure. Because there is no prior knowledge about community structure in the initial time slice, community detection must be performed over the whole network. To determine the community structure of the starting network at time slice T0, we employ community detection based on information dynamics (CDID).
b. **Changed Subgraphs:** The authors classify the events that modify the network into four categories based on the actions that may create changes in the community structure: adding nodes, removing nodes, adding edges, and deleting edges.
c. **Incremental Community Identification:** The authors use an incremental batch-based community detection approach. They use the information dynamics model to progressively discover the communities based on the produced subgraphs that may change.

2.3 Jaccard similarity coefficient:

The Jaccard similarity index (also known as the Jaccard similarity coefficient) analyses members from two sets to determine which are common and which are unique.

The Jaccard similarity coefficient of two nodes v and u is defined as follows:

$$JS_{vu} = \frac{|\ \tau(v) \bigcap \tau(u)\ |}{|\ \tau(v) \bigcup \tau(u)\ |}$$

where $u \in V, v \in V, \tau(u) = N(u) \cup \{u\}$ *, and* $N(u)$ *is the set of adjacent nodes of node* u *.*

2.4 Contact strength:

Let Gt = (Vt, Et) be an undirected network at time step t, and the contact

strength of vertex v on vertex u is defined as follows:

$$CS_{uv} = \frac{|N(u) \bigcap N(v)|}{T_u}$$

where Tu denotes the number of triangles for vertex u, and the intersection between N(u) and N(v) represents the amount of triangles shared by two nodes u and v.

2.5 Information:

Let Gt = (Vt, Et) be an undirected network at time step t, and the information of vertex u is defined as follows:

$$I_u = \frac{D_u}{D_{max}}$$

where Du represents the degree of vertex u, and Dmax denotes the maximum degree of the network Gt**.**

2.6 Information Dynamic Model:

To reveal communities in dynamic model , we begin to to build the information dynamic model, which consists of three components: information propagation volume, information loss, and propagation model

a. **Propagation Volume:** The local topology of a network, such as the degree of a node, similarities, and connection strengths of nodes, has a significant impact on information dispersion. To represent the amount of information diffusion, the authors use node similarity, connection strength, and information difference to simulate the amount of information diffusion in a more realistic way.
Formally, let Iu->v represent the information that a node v obtains from its neighbor u, which is defined as follows:

$$I_{u \to v} = f(I_u - I_v) JS_{uv} CS_{uv}$$

where JSuv denotes the Jaccard similarity coefficient between node u and node v, and CSuv represents the contact strength of node v on node u. The coupling function f() denotes the information that can be disseminated from the u to v, which is defined as follows:

$$f(I_u - I_v) = \begin{cases} e^{(I_u - I_v)} - 1 & I_u - I_v \geq 0 \\ 0 & I_u - I_v < 0. \end{cases}$$

b. **Information Loss:** In the actual world, loss can occur during information propagation due to the effect of complex settings.The authors employ the volume of information and topological aspects to characterize the loss of information in a more realistic and accurate manner. Let I(u->v)_cost denote the loss of information, which is defined as follows:

$$I_{(u \to v)\_cost} = \frac{Avg_{s(v)}}{Avg_{d(v)}} f(I_u - I_v) * (1 - JS_{uv})$$

where the first item of the formula characterizes the local topological feature, which consists of the local average similarity and local average degree. I(u->v)_cost is positively correlated with coupling function f () and negatively correlated with JSuv.

As a result, the more the information volume to be disseminated, the greater the information loss, and the lower the information loss, the more comparable the communication objects are.

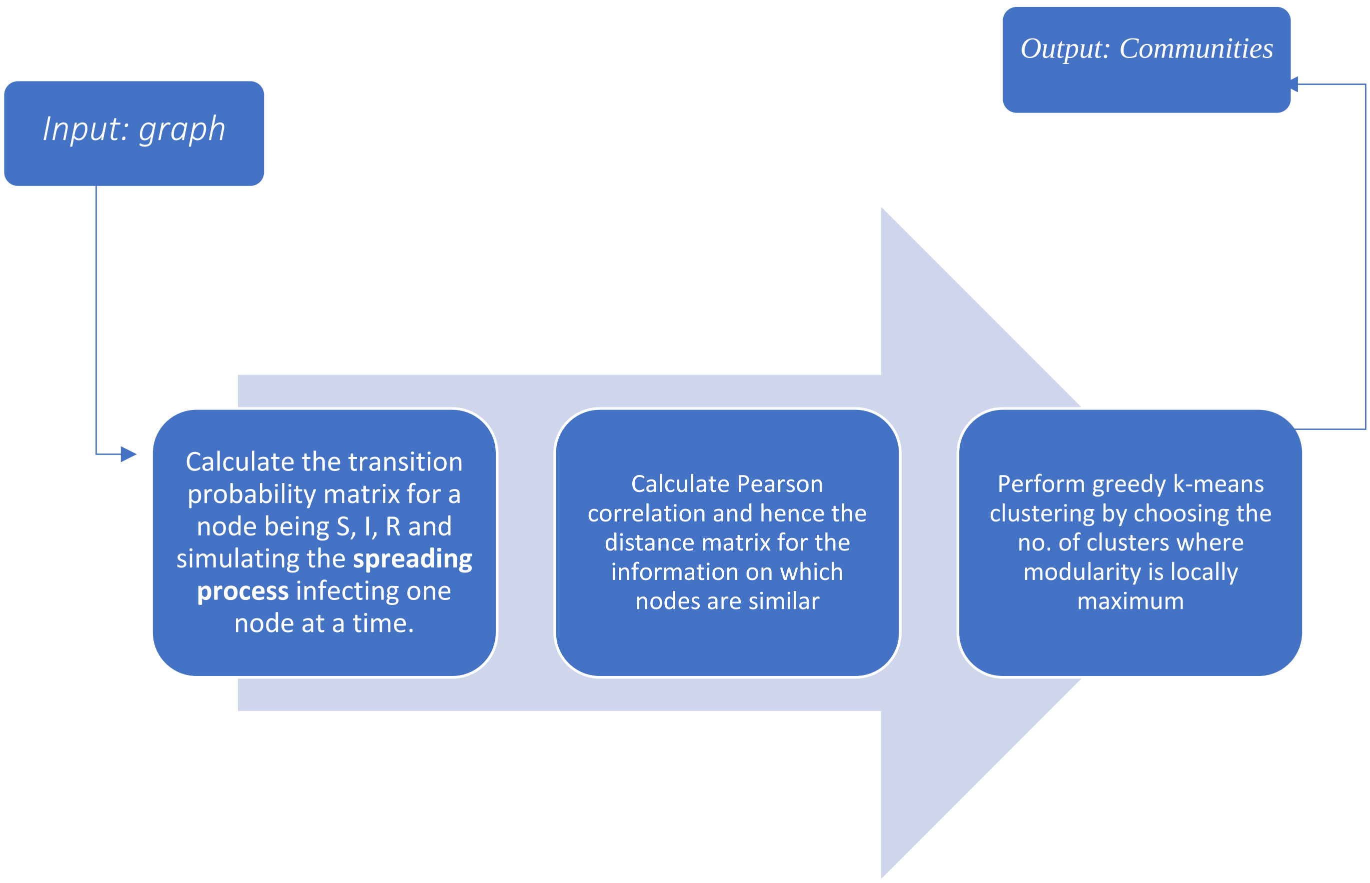
c. **Information Propagation:** By considering the information volume and the information loss described above, the information dynamics of a node v over time is provided by the following:

$$I_{v(t+1)} = I_{v(t)} + \sum_{u \in N(v)} (I_{u \to v} - I_{(u \to v)\_cost})$$

where Iv(t) represents the information of node v at time step t, and the second term of this formula denotes the information that is acquired from its neighbors.

Methodology

1. Community detection in dynamic networks via a spreading process



2. Identifying communities in dynamic networks using information dynamics

In this dynamic artificial network, we take a company as an example to present the process of dynamic community detection based on information dynamics,

which comprises the following stages:

1. First, everyone possesses his/her own knowledge as initial information because of distinct occupations (i.e., v = 0.66, u1 = 0.5), as shown in Figure 1a.
2. Then, the information spreads through the topological structure of the network.
3. Next, the communities are naturally revealed by computing the different information in the network
4. Building upon the information of communities detected at the time slice T0, an incremental community discovery framework is adopted for the subsequent snapshot networks, which includes adding nodes, deleting nodes, adding edges, and deleting edges events.
5. Figure 1c–f demonstrates that the addition and deletion of nodes and edges may lead to changes in the network structure.

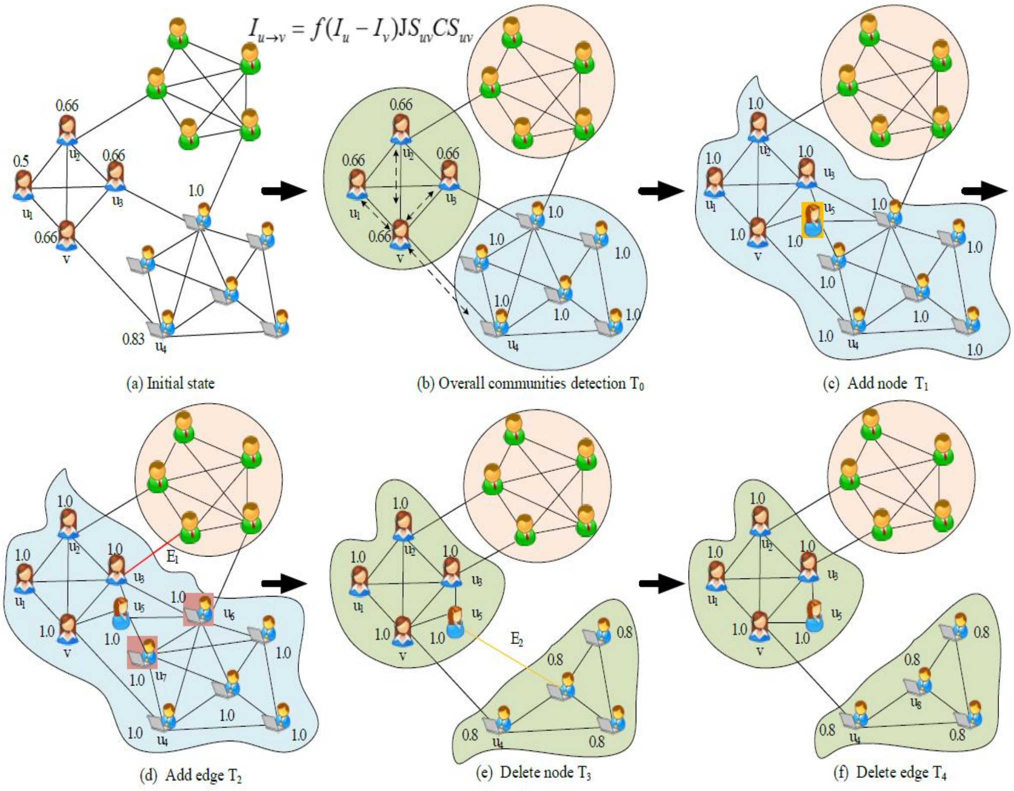
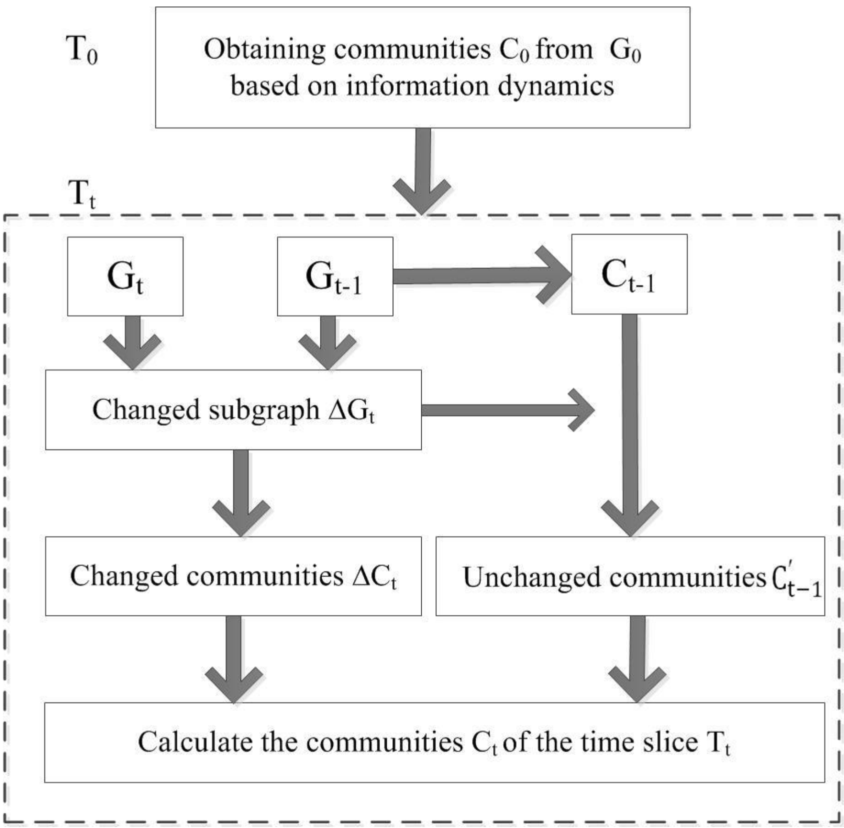


Fig: Flow Diagram

The algorithm methodology:



Experiments and Coding

1. Community detection in dynamic networks via a spreading process

1.1 Importing the libraries

Here, we import the libraries required for the implementation of the algorithm.

```
In [1]: import networkx as nx
        import networkx.algorithms.community as nx_comm
        import matplotlib.pyplot as plt
        import math
        import pandas as pd
        import seaborn as sns
        from sklearn.cluster import KMeans
        from sklearn.cluster import AgglomerativeClustering
        import warnings; warnings.simplefilter('ignore')
        from csv import reader
```

1.2 Importing and using the dataset according to the need

The datasets used are in csv format, and we extract the list of edges from it.

```
In [2]: '''
        Getting the edges from the csv formatted dataset
        '''

        #file=r"C:/Users/ayesh/Desktop/dolphin.csv"
        file = r"C:/Users/ayesh/Desktop/football.csv"
        #file = r"C:/Users/ayesh/Desktop/karate.csv"
        nodes = []
        edge_list = []
        with open(file, 'r') as read_obj:
            csv_reader = reader(read_obj)
            count = 0
            for row in csv_reader:
                if count != 0:
                    xx, yy = row
                    x = int(xx) - 1
                    y = int(yy) - 1
                    edge_list.append((int(x),int(y)))
                    if x not in nodes:
                        nodes.append(x)
                    if y not in nodes:
                        nodes.append(y)
                count += 1
```

1.3 Creating a graph from the dataset

Here, we have created a graph of the dataset using the list of edges extracted earlier.

In [3]:
```
'''
Create Graph from list of  edges
'''

G = nx.Graph()
G.add_edges_from(edge_list)
len(G)
```
Out[3]: 115

1.4 Plotting the original graphs

The original graph has been plotted using networkx library functions, in normal and circular format.

In [4]:
```
'''
Plotting the original graph
'''

plt.figure(figsize = (20, 10))
plt.subplot(122)
nx.draw(G, with_labels=True, font_weight='bold')
```

In [5]:
```
'''
Plotting the graph in circular form
'''

plt.figure(figsize=(40,40))
nx.draw_circular(G, with_labels=True)
```

1.5 Getting the adjacency matrix from the graph

We have then obtained the adjacency matrix from the graph.

In [6]:
```
'''
Getting the adjacency matrix from the graph
'''

Adj_matrix = nx.adjacency_matrix(G).todense().tolist()
```

1.6 Setting the parameters for SIR-like simulation

We've initialized the parameters to be used for the SIR-like simulation.

ntimes is the number of times we run the simulation

Lambda is the probability of a susceptible node turning Infected.

Mu is the probability of an infected node being recovered.

```
In [7]: '''
Defining the parameters for SIR simulation
'''

ntimes = 10
lamda = 0.9
mu = 0.8
```

1.7 Simulating the spreading process

```
In [8]: '''
Simulation of the spreading process with single node infected
'''

def run_simulation(G, times, Prob_R, Prob_S, Prob_I):
    temp_Prob_R, temp_Prob_S, temp_Prob_I = Prob_R, Prob_S, Prob_I
    for t in range(1, times+1):
        # calculate Prob_S for every node
        for node in nodes:
            Prob_S[node] = temp_Prob_S[node]
            mult = 1
            for n in nodes:
                mult *= (1 - lamda * Adj_matrix[n][node] * temp_Prob_I[n])
            Prob_S[node] *= mult
        # calculate Prob_I for every node
        for node in nodes:
            Prob_I[node] = temp_Prob_S[node]
            mult = 1
            for n in nodes:
                mult *= (lamda * Adj_matrix[n][node] * temp_Prob_I[n])
            Prob_I[node] *= (1 - mult)
            Prob_I[node] += (1 - mu) * temp_Prob_I[node]
        # calculate Prob_R for every node
        for node in nodes:
            Prob_R[node] = mu * temp_Prob_I[node] + temp_Prob_R[node]
        temp_Prob_S, temp_Prob_I, temp_Prob_R = Prob_S, Prob_I, Prob_R
    return temp_Prob_S, temp_Prob_I, temp_Prob_R
```

1.8 Filling the transition probability matrix

Here, we fill in the transition probability matrix considering one node to be infected and the other nodes

```
In [9]: '''
Infecting every node and filling the transition probability matrix
'''

nodes = list(G.nodes())

Prob_matrix = [[0 for i in range(len(nodes))] for j in range(len(nodes))]

for node in nodes:
    Prob_R = [0 for _ in range(len(nodes))]
    Prob_S = [mu for _ in range(len(nodes))]
    Prob_I = [0 for _ in range(len(nodes))]

    Prob_I[node] = 1

    Prob_S, Prob_I, Prob_R = run_simulation(G, ntimes, Prob_R, Prob_S, Prob_I)

    for n in nodes:
        Prob_matrix[n][node] = Prob_R[n]
```

1.9 Calculating the partition matrix

The partition matrix is calculated.

```
In [10]: '''
Calculating the partition matrix from the probability matrix
'''

Part_matrix = [[0 for i in range(len(nodes))] for j in range(len(nodes))]
for i in range(len(nodes)):
    for j in range(len(nodes)):
        if i != j:
            Part_matrix[i][j] = Prob_matrix[i][j]
        else:
            x = 0
            for jj in range(len(nodes)):
                if jj != i:
                    x += Prob_matrix[i][jj]
            Part_matrix[i][j] = x / (len(nodes)-1)
```

1.10 Calculating the mean and standard deviation for Pearson Correlation

```
In [11]: '''
Initialising the mean and standard deviation for Pearson Correlation
'''
means = [0 for _ in range(len(nodes))]
sds = [0 for _ in range(len(nodes))]
```

```
In [12]: '''
Calculating the Mean
'''
for i in range(len(nodes)):
    means[i] = sum(Part_matrix[i])/(len(nodes))
```

```
In [13]: '''
Calculating the Standard Deviation
'''

for i in range(len(nodes)):
    x = 0
    for k in range(len(nodes)):
        x += (Part_matrix[i][k] - means[i])*(Part_matrix[i][k] - means[i])
    x /= len(nodes)
    sds[i] = math.sqrt(x)
```

1.11 Calculating Pearson Correlation

```
In [14]: '''
Calculating the Pearson Correlation from the Partition Matrix
'''

Pear_cor = [[0 for i in range(len(nodes))] for j in range(len(nodes))]
for i in range(len(nodes)):
    for j in range(len(nodes)):
        x = 0
        for k in range(len(nodes)):
            x += (Part_matrix[i][k]-means[i])*(Part_matrix[j][k]-means[j])
        x /= (len(nodes)*sds[i]*sds[j])
        Pear_cor[i][j] = x
```

1.12 Calculating the distance matrix

```
In [15]: '''
Calculating the distance matrix from Pearson Correlation
'''

Distance_matrix = [[0 for i in range(len(nodes))] for j in range(len(nodes))]
for i in range(len(nodes)):
    for j in range(len(nodes)):
        Distance_matrix[i][j] = 1 - Pear_cor[i][j]
```

1.13 Performing Greedy K-means clustering

```
In [16]: '''
         Performing clustering based on the distance matrix
         '''

         cluster = KMeans(2)
         cluster.fit(Distance_matrix)
         identified_clusters= cluster.fit_predict(Distance_matrix)
```

```
In [17]: '''
         USing k-means greedy algorithm greedily
         '''
         def calcModularity(n):
             cluster = KMeans(n)
             cluster.fit(Distance_matrix)
             identified_clusters=cluster.fit_predict(Distance_matrix)
             print(identified_clusters)
             clustMat=[[0 for i in range(0)] for j in range(n+1)]
             i=0
             for id in identified_clusters:
                 clustMat[id+1].append(i)
                 i+=1

             print(clustMat)
             mods = nx_comm.modularity(G,clustMat,weight='weight',resolution=1)
             print(mods)
             return mods
```

```
In [18]: fv=5
         for i in range(3,len(nodes)-1):
             prv_mod=calcModularity(i-1)
             cur_mod=calcModularity(i)
             nxt_mod=calcModularity(i+1)
             if(prv_mod<=cur_mod  and cur_mod>=nxt_mod):
                 fv=i
                 break
         cluster = KMeans(fv)
         cluster.fit(Distance_matrix)
         identified_clusters= cluster.fit_predict(Distance_matrix)
```

1.14 Accuracy measures

```
In [31]: '''
         Ground truth for Football Network
         '''
         graph = nx.read_gml ('C:/Users/ayesh/Downloads/football.gml', label = 'id')
         gt_membership = [graph.nodes[v]['value'] for v in G.nodes()]

         '''
         Calculating NMI and ARI measures
         '''
         print(gt_membership)
         print('NMI value: ', normalized_mutual_info_score(gt_membership, colors))
         print('ARI value: ', adjusted_rand_score(gt_membership,colors))
```

We have calculated NMI and ARI using the ground truth values.

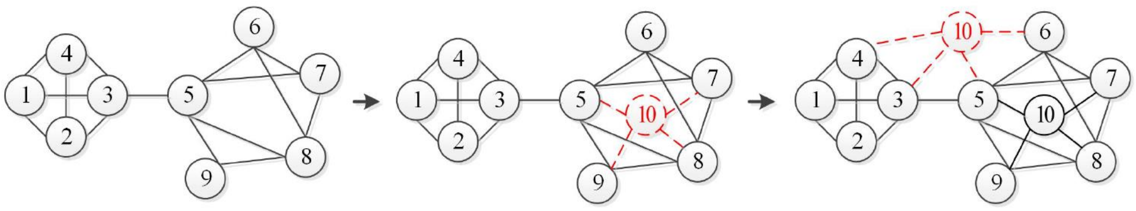
2. Identifying communities in dynamic networks using information dynamics

2.1 Add Nodes:

When compared to the prior time slice network Gt-1, adding nodes refers to the addition of additional nodes to the current time slice network Gt.Let AN denotes the set of added nodes,which is defined as follows:

$$AN = \{v|v \in V_t, v \notin V_{t-1}\}$$

where Vt and Vt-1 represent the set of nodes in the networks Gt and Gt-1, respectively. The connection density of a community is increased by adding a node within it, while the number of communities remains the same. As a result, all that is required is to split the new nodes into the existing community. When the additional node is not part of the community, however, the community structure may alter.



(a) Original graph (b) Adding nodes in a community (c) Adding nodes between communities

Fig: adding nodes in a community

2.1.1 Algorithm:

Algorithm 3 Add_nodes

Input: AN, G, C
Output: Δg

for each node $v \in AN$ **do**
  if $N(v)$ in the same community $C_{N(v)}$ **then**
    $C_{N(v)} \leftarrow v$
  else
    $\Delta g \leftarrow v$
    for each node $u \in N(v)$ **do**
      $\Delta g \leftarrow C_u$
    end for
  end if
end for

2.1.2 Code:

```
def node_addition(G, addnodes, communitys):

    change_comm = set()  # Can be unrestricted.

    processed_edges = set()  # Processed edge



    for u in addnodes:

        neighbors_u = G.neighbors(u)

        neig_comm = set()  # Label of the company where you live

        pc = set()

        for v in neighbors_u:
```

```
            neig_comm.add(communitys[v])

            pc.add((u, v))

            pc.add((v, u))  # undirected

        if len(neig_comm) > 1:  # Explanation Inside the company district 
where this joining point is absent

            change_comm = change_comm | neig_comm

            lab = max(communitys.values()) + 1

            communitys.setdefault(u, lab)

            change_comm.add(lab)

        else:

            if len(neig_comm) == 1:  # Explanation Concluding point Inside 
the company district, or connecting with one company district

                communitys.setdefault(v, neig_comm[0])

                processed_edges = processed_edges | pc

            else:

                communitys.setdefault(v, max(communitys.values())+1)


    # Returnable development transformational company district, processing 
transitional Bien Hoa latest company district structure.

    return change_comm, processed_edges, communitys
```

2.2 Delete Nodes:

The deleted node refers to a node that is removed in the current time slice network Gt compared with the previous time slice network Gt-1. Let AN represent the set of deleted nodes, which is given by the following:

$$DN = \{v | v \notin V_t, v \in V_{t-1}\}.$$

The deleted nodes can be computed by solving the difference set between sets Vt and Vt-1. We can observe that the deletion of a node within one community or between The community has caused changes in the structure of the community. Therefore, when deleting a node, we need to add the deleted node and the connected communities to the subgraph ΔGt.

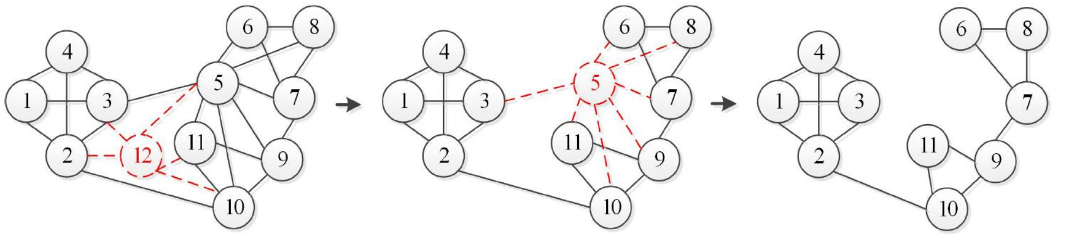


Fig: deleting nodes in a community

2.2.1 Algorithm:

Algorithm 4 Del_nodes

Input: DN, G, C
Output: Δg
1: **for** each node $v \in DN$ **do**
2: **for** each node $u \in N(v)$ **do**
3: $\Delta g \leftarrow C_u$
4: **end for**
5: **end for**

2.2.2 Code:

```
def node_deletion(G, delnodes, communitys):  # tested, correct
    change_comm = set()  # Can be unrestricted.
    processed_edges = set()  # Processed edge
    for u in delnodes:
        neighbors_u = G.neighbors(u)
        neig_comm = set()  # Label of the company where you live
        for v in neighbors_u:
            neig_comm.add(communitys[v])
            processed_edges.add((u, v))
            processed_edges.add((v, u))
        del communitys[u]
        change_comm = change_comm | neig_comm
```

```
    # Returnable development transformational company district, processing transitional Bien Hoa latest company district structure.

    return change_comm, processed_edges, communitys
```

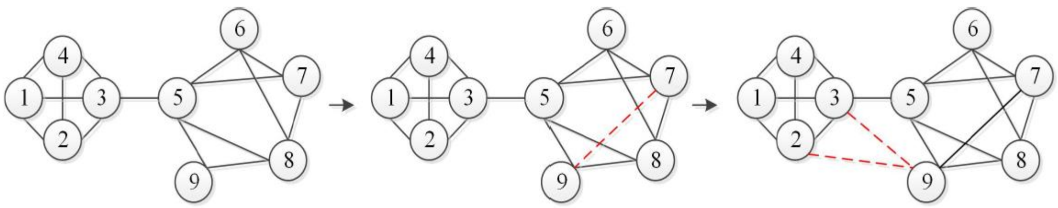
2.3 Add Edges:

Similarly, the added edges correspond to the new edges in the current time slice network Gt compared to the previous time slice network Gt-1. Formally, we define the added edges as follows:

$$AE = \{e|e \in E_t, e \notin E_{t-1}\}.$$

where Et and Et-1 represent the set of edges in the networks Gt and Gt-1, respectively.

It is not necessary to deal with the new edges added. However, the addition of edges between communities may lead to changes in community structure.



(a) Original graph (b) Adding edges in a community (c) Adding edges between communities

Fig: adding edges in a community

2.3.1 Algorithm:

Algorithm 5 Add_edges

Input: AE, C
Output: Δg
1: **for** each edge $e \in AE$ **do**
2: $//(u,v) \in e$
3: **if** $C_u \neq C_v$ **then**
4: $\Delta g \leftarrow C_u$
5: $\Delta g \leftarrow C_v$
6: **end if**
7: **end for**

2.3.2 Code:

```
def edge_addition(addedges, communitys):

    change_comm = set()  # Can be unrestricted.

# print addedges

# print communitys
```

```
    for item in addedges:

        neig_comm = set()  # Label of the company where you live

        neig_comm.add(communitys[item[0]])  # Judgment

        neig_comm.add(communitys[item[1]])

        if len(neig_comm) > 1:  # Explanation Inside the company district
where this member is absent

            change_comm = change_comm | neig_comm

    return change_comm  # Returnable Develop transformational company
district,
```

2.4 Delete Edges:

The deleted edge refers to the edge removed in the current time slice networkGt compared to the previous time slice network Gt-1. Let DE denote the set of deleted edges, which is defined as follows:

$$DE = \{e | e \notin E_t, e \in E_{t-1}\}.$$

Deleting the inner edge causes the community to split. Therefore, we need to add the current edge and the community involved to the subgraph ΔGt.In contrast, deleting the links between the communities weakens the connection between them, which does not cause changes of the original communities.

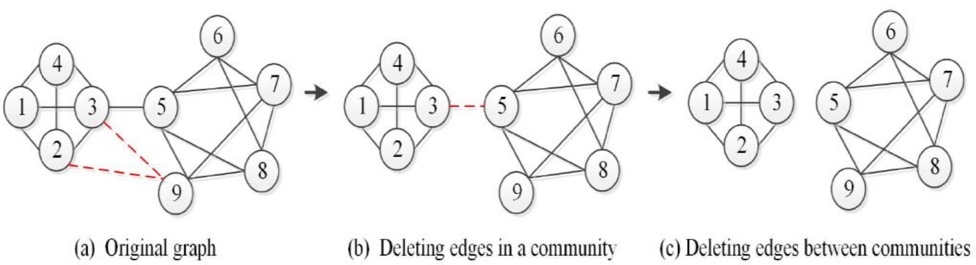


Fig: deleting edges in a community

2.4.1 Algorithm:

Algorithm 6 Del_edges

Input: DE, C
Output: Δg

for each edge $e \in AE$ **do**
  $//(u, v) \in e$
  if $C_u = C_v$ **then**
    $\Delta g \leftarrow C_u$
  end if
end for

2.4.2 Code:

```
def edge_deletion(deledges, communitys):

    change_comm = set()  # Can be unrestricted.

    for item in deledges:

        neig_comm = set()  # Label of the company where you live

        neig_comm.add(communitys[item[0]])  # Judgment

        neig_comm.add(communitys[item[1]])

        if len(neig_comm) == 1:  # Explanation Inside the company district
where this member is absent

            change_comm = change_comm | neig_comm

    return change_comm  # Returnable
```

2.5 Calculate Changed Communities :

ΔCt. After obtaining the subgraph ΔGt that may change,we need to redetect the communities in the subgraph. Here, we employ the information dynamics to discover the subgraph ΔGt incrementally and obtain the corresponding community structure ΔCt.

Code:

```
def getchangegraph(all_change_comm, newcomm, Gt):

    Gte = nx.Graph()

    com_key = newcomm.keys()

    for v in Gt.nodes():

        if v not in com_key or newcomm[v] in all_change_comm:
```

```
            Gte.add_node(v)

            neig_v = Gt.neighbors(v)

            for u in neig_v:

                if u not in com_key or newcomm[u] in
all_change_comm:

                    Gte.add_edge(v, u)

                    Gte.add_node(u)


    return Gte
```

Compute Unchanged Communities C’t-1: Based on the acquired networks Gt-1 and ΔGt,we can calculate all the communities and the communities that may change at the t-1 time slice.Therefore, the unchanged communities can be obtained by calculating the difference set of the two sets.

Compute Communities Ct: Communities in the network at time slice t are composed of the unchanged communities at the previous time slice t-1 and the changed communities at the time slice t. Let Ct denote the communities of network Gt at time slice t, which is given as follows:

$$C_t = C'_{t-1} + \Delta C_t$$

where C’t-1 represents the unchanged communities at the previous time slice t-1, and ΔCt denotes the changed communities at the time slice t.

2.6 Community Detection based on Information Dynamics:

We identify the community structure using information dynamics models by modeling the exchange of information on the network, which entails multiple processes. Each node is given initial information based on the local topological features at the start. The information then spreads across the network, and each node is continually communicating with its neighbors. Information is exchanged more often between nodes in the same community than between nodes in different communities. Each node changes its information depending on the information dynamics models at each step. The exchange of information between nodes tends to zero as time passes, and the information dynamics of each node in the network converge. Finally, the quantity of information available for each node in the same community is essentially the same, however the amount of information available for each node in different communities varies. As a result, by examining the quantity of information for each node, we may automatically expose the communities.

2.6.1 Algorithm:

Algorithm 2 CDID

Input: $G_t = (V_t, E_t)$
Output: C_t

//Initialization of information
for each node $v \in V_t$ **do**
    for each node $u \in N(v)$ **do**
        compute the JS_{vu}, CS_{uv} using Equation (1)–(2)
    end for
    compute the I_v using Equation (3)
end for
//Information dynamic interaction.
while *true* **do**
    $I_{max} = 0$
    for each node $v \in V_t$ **do**
        for each node $u \in N(v)$ **do**
            compute $I_{u \to v}$ using Equation (4)–(5)
            compute $I_{(u \to v)\_cost}$ using Equation (6)
        end for
        compute $I_{v(t+1)}$ using Equation (7)
        $I_{in} = I_{u \to v} - I_{(u \to v)\_cost}$
        if $I_{in} > I_{max}$ **then**
            $I_{max} = I_{in}$
        end if
    end for
    // the balanced state
    if $I_{max} < Threshold$ **then**
        Break
    end if
end while
// Find communities C_t
for each node $v \in V_t$ **do**
    if $v \notin C_t$ **then**
        for each node $u \in N(v)$ **do**
            if $|I_v - I_u| < Threshold$ **then**
                $u -> C_v$
            else
                $u -> C_u$
            end if
        end for
    end if
end for

2.6.2 Code:

```
def CDID(Gsub, maxlabel):  # G_sub is a subgraph, run information dynamics on subgraphs that may change the structure, maxlabel is the maximum label that does not change the community structure



    # initial information
```

```
    Neigb = {}

    info = 0

    # average degree, maximum degree

    avg_d = 0

    max_deg = 0

    N = Gsub.number_of_nodes()

    deg = Gsub.degree()

    max_deg = max(dict(deg).values())

    avg_d = sum(dict(deg).values()) * 1.0 / N


    ti = 1

    list_I = {}  # Store the information of each node, the initial is the
degree of each node, and each iteration continues to change dynamically

    maxinfo = 0

    starttime = datetime.datetime.now()

    for v in Gsub.nodes():

        if deg[v] == max_deg:

            info_t = 1 + ti * 0

            ti = ti + 1

# print v,max_deg,info_t

            maxinfo = info_t

        else:

            info_t = deg[v] * 1.0 / max_deg

            # info_t=round(random.uniform(0,1),3)

        # info_t=deg[v]*1.0/max_deg

        list_I.setdefault(v, info_t)

        Neigb.setdefault(v, Gsub.neighbors(v))  # The neighbor node of node
v

        info += info_t

    node_order = sorted(list_I.items(), key=lambda t: t[1], reverse=True)

    node_order_list = list(zip(*node_order))[0]
```

```
    # Calculate the similarity between nodes, the Jaccard coefficient


    def sim_jkd(u, v):

        list_v = Gsub.neighbors(v)

        list_v.append(v)

        list_u = Gsub.neighbors(u)

        list_u.append(u)

        t = set(list_v)

        s = set(list_u)


        return len(s & t) * 1.0 / len(s | t)

    # Calculate the number of hop2 between nodes


    def hop2(u, v):

        list_v = (Neigb[v])

        list_u = (Neigb[u])

        t = set(list_v)

        s = set(list_u)

        return len(s & t)


    st = {}  # store the similarity

    hops = {}  # store hop2 number

    hop2v = {}  # Store the ratio of hop2 numbers

    sum_s = {}  # Store the sum of the neighbor similarity of each node

    avg_sn = {}  # Store the local average similarity of each node, local
refers to the neighbor nodes

    avg_dn = {}  # Store the local average degree of each node


    for v, Iv in list_I.items():

        sum_v = 0
```

```
        sum_deg = 0
        tri = nx.triangles(Gsub, v) * 1.0
        listv = Neigb[v]
        num_v = len(list(listv))
        sum_deg += deg[v]

        for u in listv:
            keys = str(v) + '_' + str(u)
            p = st.setdefault(keys, sim_jkd(v, u))
            h2 = hop2(v, u)
            hops.setdefault(keys, h2)
            if tri == 0:
                if deg[v] == 1:
                    hop2v.setdefault(keys, 1)
                else:
                    hop2v.setdefault(keys, 0)
            else:
                hop2v.setdefault(keys, h2 / tri)

            sum_v += p
            sum_deg += deg[u]

        sum_s.setdefault(v, sum_v)
        avg_sn.setdefault(v, sum_v * 1.0 / num_v)
        avg_dn.setdefault(v, sum_deg * 1.0 / (num_v + 1))
#    print('begin loop')

#    oldinfo = 0
    info = 0
    t = 0
```

```
while 1:
    info = 0
    t = t + 1
    Imax = 0

    for i in range(len(node_order_list)):
        v = node_order_list[i]
        Iv = list_I[v]
        for u in Neigb[v]:
            # p=sim_jkd(v,u)
            keys = str(v) + '_' + str(u)

            Iu = list_I[u]
            if Iu - Iv < 0:
                #                          It=It*1.0/E
                It = 0
            else:
                It = (math.exp(Iu - Iv) - 1)
            # It=It*1.0*deg[u]/(deg[v]+deg[u])
            if It < 0.0001:
                It = 0  #
            fuv = It
            #                       print(fuv)
            p = st[keys]
            p1 = p * hop2v[keys]
            Iin = p1 * fuv  #
            Icost = avg_sn[v] * fuv * (1 - p) / avg_dn[v]
            #                Icost=avg_s*fuv*avg_c/avg_d
            #                Icost=(avg_sn[v])*fuv/avg_dn[v]
```

```
                Iin = Iin - Icost
                if Iin < 0:
                    Iin = 0
                Iv = Iv + Iin
                #                        print(v,u,Iin,Icost,Iv,Iu,It)
                if Iin > Imax:
                    Imax = Iin

            if Iv > maxinfo:
                Iv = maxinfo
            list_I[v] = Iv
            # print(v,u,Iin,Iv,Iu,tempu[0],pu,tempu[1],fuv)
            info += list_I[v]
        # if v==3:
        #                print(v,Iv)

        if Imax < 0.0001:
            break

    endtime = datetime.datetime.now()
    # print ('time:', (endtime - starttime).seconds)
    # Group division ***********************************************
****************


    queue = []
    order = []
    community = {}
    lab = maxlabel
    number = 0
    for v, Info in list_I.items():
```

```
    if v not in community.keys():
        lab = lab + 1
        queue.append(v)
        order.append(v)
        community.setdefault(v, lab)
        number = number + 1
        while len(queue) > 0:
            node = queue.pop(0)
            for n1 in Neigb[node]:
                if (not n1 in community.keys()) and (not n1 in queue):
                    if abs(list_I[n1] - list_I[node]) < 0.001:
                        queue.append(n1)
                        order.append(n1)
                        community.setdefault(n1, lab)
                        number = number + 1
    if number == N:
        break

        # print (order)
        # print(community)
order_value = [community[k] for k in sorted(community.keys())]
commu_num = len(set(order_value))   # number of communities
endtime1 = datetime.datetime.now()
print('Social division ends')
print(list_I)
#print('community number:', commu_num)
print('alltime:', (endtime1 - starttime).seconds)
return community
```

2.7 Dynamic Community Detection:.

Initial community structure detection, calculation of changing subgraphs, and incremental community identification are the three steps of dynamic community detection (DCDID).

Initial Detection of Community Structure: The original community structure is the network's community split at time slice T0. Because the first slice has no past information about community structure, community detection must be done over the whole network. We use community detection based on information dynamics to establish the community structure of the initial network at time slice T0 (CDID).

Changed Subgraphs: The authors divide network modifications into four categories depending on the acts that may cause changes in the community structure: adding nodes, removing nodes, adding edges, and deleting edges.

Incremental Community Identification: The authors employ an incremental batch-based community identification technique to detect communities. They employ the information dynamics concept to gradually uncover communities based on the subgraphs that are formed and may alter.

Algorithm 1 DCDID

Input: $DG = \{G_0, G_1, ..., G_k\}$
Output: $DC = \{C_0, C_1, ..., C_k\}$

//Initial community detection
$C_0 = CDID(G_0)$
//Incremental community detection
for $t = 1$ to k **do**
  compute AN,DN,AE,DE using Equation (8)–(11)
  $\Delta G_t \leftarrow Add\_nodes(AN, G_t, C_{t-1})$
  $\Delta G_t \leftarrow Del\_nodes(DN, G_{t-1}, C_{t-1})$
  $\Delta G_t \leftarrow Add\_edges(AE, C_{t-1})$
  $\Delta G_t \leftarrow Del\_edges(DE, C_{t-1})$
  compute the unchanged communities C'_{t-1}
  $\Delta C_t \leftarrow CDID(\Delta G_t)$
  compute C_t using Equation (12)
end for

Code:

```
edges_added = set()

edges_removed = set()

nodes_added = set()

nodes_removed = set()
```

```
G = nx.Graph()


# Edge Path
edge_file = '15node_t01.txt'
# Path to the directory
path = 'DCDID1/data/test1/'
# Adding nodes to the graph with edges
with open(path+edge_file, 'r') as f:


    edge_list = f.readlines()
    for edge in edge_list:
        edge = edge.split()
        G.add_node(int(edge[0]))
        G.add_node(int(edge[1]))
        G.add_edge(int(edge[0]), int(edge[1]))
G = G.to_undirected()


# initial graph


print('Time Slice NetworkG0******************************************** *')
nx.draw_networkx(G)
fpath = 'DCDID1/data/pic/G_0.png'
plt.savefig(fpath)
# Output method 1: save the image as a png format image file
plt.show()
# print G.edges()
# comm_file='switch.t01.comm'
comm_file = '15node_comm_t01.txt'
with open(path+comm_file, 'r') as f:
    comm_list = f.readlines()
```

```
    comm_list = str_to_int(comm_list)

comm = {}  # Used to store the detected community structure in the format
{node: community label}

comm = CDID(G, 0)  # initial community

# drawing community

print('Community C0 of T0 time
slice****************************************** ****')

print(comm)

drawcommunity(G, comm, 'DCDID1/data/pic/community_0.png')

initcomm = conver_comm_to_lab(comm)

comm_va = list(initcomm.values())

getscore(comm_va, comm_list)

start = time.time()

G1 = nx.Graph()

G2 = nx.Graph()

G1 = G

# filename='switch.t0'

filename = '15node_'

for i in range(2, 5):

    print('begin loop:', i-1)

    # comm_new_file=open(path+'output_new_'+str(i)+'.txt','r')

# comm_new_file=open(path+filename+str(i)+'.comm','r')

    comm_new_file = open(path+filename+'comm_t0'+str(i)+'.txt', 'r')

    if i < 10:

        # edge_list_old_file=open(path+'switch.t0'+str(i-1)+'.edges','r')

        # edge_list_old=edge_list_old_file.readlines()

        # edge_list_new_file=open(path+filename+str(i)+'.edges','r')

        edge_list_new_file = open(path+filename+'t0'+str(i)+'.txt', 'r')

        edge_list_new = edge_list_new_file.readlines()

        comm_new = comm_new_file.readlines()

    elif i == 10:
```

```
        # edge_list_old_file=open(path+'switch.t09.edges','r')
        # edge_list_old=edge_list_old_file.readlines()
        edge_list_new_file = open(path+'switch.t10.edges', 'r')
        edge_list_new = edge_list_new_file.readlines()
        comm_new = comm_new_file.readlines()
    else:
        # edge_list_old_file=open(path+'switch.t'+str(i-1)+'.edges','r')
        # edge_list_old=edge_list_old_file.readlines()
        edge_list_new_file = open(path+'switch.t'+str(i)+'.edges', 'r')
        edge_list_new = edge_list_new_file.readlines()
        comm_new = comm_new_file.readlines()
    comm_new = str_to_int(comm_new)


# for line in edge_list_old:
# temp = line.strip().split()
#
# G1.add_edge(int(temp[0]),int(temp[1]))
    for line in edge_list_new:
        temp = line.strip().split()
        G2.add_edge(int(temp[0]), int(temp[1]))
    print('T'+str(i-1)+'time slice network G'+str(i-1) +
          '********************* ***********************')
    nx.draw_networkx(G2)
    fpath = 'DCDID1/data/pic/G_' + \
        str(i-1)+'.png'
    # Output method 1: save the image as a png format image file
    plt.savefig(fpath)
    plt.show()
# total_nodes = previous_nodes.union(current_nodes)#The total number of
nodes in the current time slice and the previous time slice, the two sets
are related
```

```
    total_nodes = set(G1.nodes()) | set(G2.nodes())
# current_nodes.add(1002)
# previous_nodes.add(1001)

    nodes_added = set(G2.nodes())-set(G1.nodes())
    print('Add node set to: ', nodes_added)
    nodes_removed = set(G1.nodes())-set(G2.nodes())
    print('Remove node set as:', nodes_removed)
# print ('G2', G2.nodes())
# print ('G1', G1.nodes())
# print ('add node',nodes_added)
# print ('remove node',nodes_removed)
    edges_added = set(G2.edges())-set(G1.edges())
    print('Added edge set is: ', edges_added)
    edges_removed = set(G1.edges())-set(G2.edges())
    print('Delete edge set: ', edges_removed)
# print ('add edges',edges_added)
# print ('remove edges',edges_removed)
# print len(G1.edges())
# print len(edges_added), len(edges_removed)
    all_change_comm = set()
    #Add node processing ##############################################
################
    addn_ch_comm, addn_pro_edges, addn_commu = node_addition(
        G2, nodes_added, comm)
# print ('addnode_community',addn_commu)
# print edges_added
# print addn_pro_edges
    edges_added = edges_added-addn_pro_edges  # remove processed edges
# print edges_added
```

```
    all_change_comm = all_change_comm | addn_ch_comm
# print('addn_ch_comm',addn_ch_comm)


    #Delete node processing ##############################################
################
# print('nodes_removed',nodes_removed)
    deln_ch_comm, deln_pro_edges, deln_commu = node_deletion(
        G1, nodes_removed, addn_commu)
    all_change_comm = all_change_comm | deln_ch_comm
    edges_removed = edges_removed-deln_pro_edges
# print('deln_ch_comm',deln_ch_comm)
# print ('delnode_community',deln_commu)
    #Add edge processing ##############################################
###############
# print('edges_added',edges_added)
    adde_ch_comm = edge_addition(edges_added, deln_commu)
    all_change_comm = all_change_comm | adde_ch_comm
# print('all_change_comm',all_change_comm)
    #delete edge processing ##############################################
###############
    dele_ch_comm = edge_deletion(edges_removed, deln_commu)
    all_change_comm = all_change_comm | dele_ch_comm
# print('all_change_comm',all_change_comm)
    unchangecomm = ()  # Unchanged community tag
    newcomm = {}  # The format is {node:community}
    # Add edges and delete edges, just process on existing nodes, no new
nodes will be added, nodes will be deleted (previously processed)
    newcomm = deln_commu
    unchangecomm = set(newcomm.values())-all_change_comm
    unchcommunity = {key: value for key, value in newcomm.items(
    ) if value in unchangecomm}  # unchanged community : tags and nodes
```

```
    # Find the subgraph corresponding to the changed community, then use
information dynamics on the subgraph to find the new community structure,
add the unchanged community structure, and get the new community structure.

# print('change community:',all_change_comm)

    Gtemp = nx.Graph()

    Gtemp = getchangegraph(all_change_comm, newcomm, G2)

    unchagecom_maxlabe = 0

    if len(unchangecomm) > 0:

        unchagecom_maxlabe = max(unchangecomm)

# print('subG', Gtemp.edges())

    if Gtemp.number_of_edges() < 1:  # community has not changed

        comm = newcomm

    else:

        getnewcomm = CDID(Gtemp, unchagecom_maxlabe)

        print('T'+str(i-1)+'time slice delta_g'+str(i-1) +

              '************************************************')

        nx.draw_networkx(Gtemp)

        fpath = 'DCDID1/data/pic/delta_g' + \

            str(i-1)+'.png'

        plt.savefig(fpath)

        plt.show()


# print('newcomm', getnewcomm)

        # Merge community structure, unchanged plus newly acquired

# mergecomm=dict(unchcommunity, **getnewcomm )#The format is
{node:community}

        d = dict(unchcommunity)

        d.update(getnewcomm)

        # Take the currently obtained community structure as the next
community input

        comm = dict(d)
```

```
        print('T'+str(i-1)+'time slice network community structure 
C'+str(i-1) +
              '***********************************************')
        drawcommunity(G2, comm, 'DCDID1/data/pic/community_'+str(i-
1)+'.png')
# print ('getcommunity:',conver_comm_to_lab(comm))
    getscore(list(conver_comm_to_lab(comm).values()), comm_new)
    print('community number:', len(set(comm.values())))
    print(comm)
    G1.clear()
    G1.add_edges_from(G2.edges())
    G2.clear()
print('all done')


# Convert community format to, label as primary key, node as value
def conver_comm_to_lab(comm1):
    overl_community = {}
    for node_v, com_lab in comm1.items():
        if com_lab in overl_community.keys():
            overl_community[com_lab].append(node_v)
        else:
            overl_community.update({com_lab: [node_v]})
    return overl_community


def getscore(comm_va, comm_list):
    actual = []
    baseline = []
    # groundtruth, j represents each community, j is the community name
    for j in range(len(comm_va)):
```

```
        for c in comm_va[j]:  # Each node in the community, representing each node
            flag = False
            # The detected community, k is the community name
            for k in range(len(comm_list)):
                if c in comm_list[k] and flag == False:
                    flag = True
                    actual.append(j)
                    baseline.append(k)
                    break
    print('nmi', metrics.normalized_mutual_info_score(actual, baseline))
    print('ari', metrics.adjusted_rand_score(actual, baseline))



def drawcommunity(g, partition, filepath):
    pos = nx.spring_layout(g)
    count1 = 0
    t = 0
    node_color = ['#66CCCC', '#FFCC00', '#99CC33', '#CC6600', '#CCCC66',
                  '#FF99CC', '#66FFFF', '#66CC66', '#CCFFFF', '#CCCC00', '#CC99CC', '#FFFFCC']

    print("partition")
    print(partition)
    for com in set(partition.values()):
        count1 = count1 + 1.
        list_nodes = [nodes for nodes in partition.keys()
                      if partition[nodes] == com]

        def r(): return random.randint(0, 255)
        color = '#{:02x}{:02x}{:02x}'.format(r(), r(), r())
```

```
        nx.draw_networkx_nodes(
            g, pos, list_nodes, node_size=220, node_color=color)
        nx.draw_networkx_labels(g, pos)
        t = t+1

    labels = nx.get_edge_attributes(g, 'weight')
    nx.draw_networkx_labels(g, pos)
    nx.draw_networkx_edges(g, pos, alpha=0.5)
    plt.savefig(filepath)
    plt.show()
```

Result Analysis

1. Community detection in dynamic networks via a spreading process

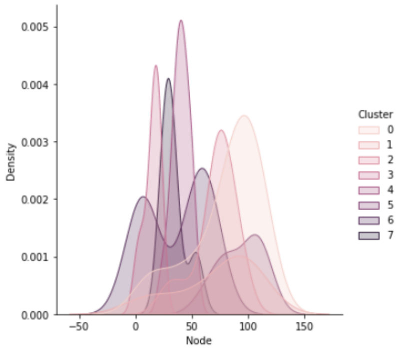
Data visualization

(The below results are for the Football Network)

We've used various plotting techniques to visualize the cluster and node data we obtain from the algorithm.

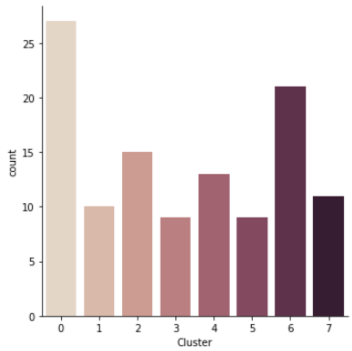
In [23]:
```
'''
Density plot for clusters
'''
sns.displot(df, x="Node", hue="Cluster", kind="kde", fill=True)
```

Out[23]: <seaborn.axisgrid.FacetGrid at 0x2344a67e6d0>



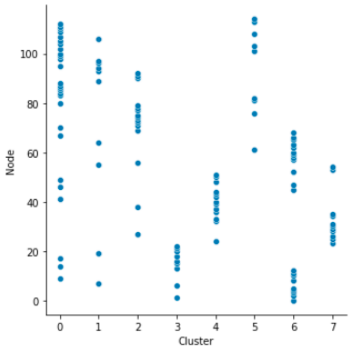
In [24]:
```
'''
Nodes in each cluster
'''
sns.catplot(x="Cluster", kind="count", palette="ch:.25", data=df)
```

Out[24]: <seaborn.axisgrid.FacetGrid at 0x2344a5a87c0>

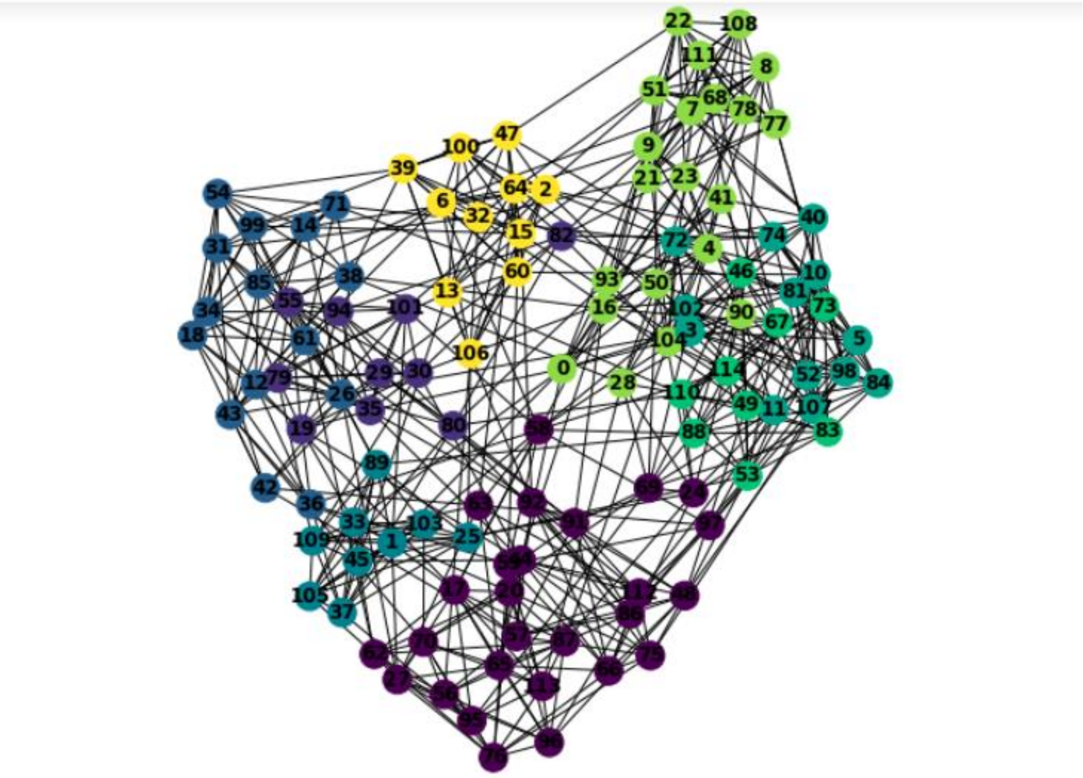


In [25]:
```
'''
Cluster contains which node
'''
sns.relplot(data=df, x="Cluster", y="Node")
```

Out[25]: <seaborn.axisgrid.FacetGrid at 0x2344a5b0070>



Graph Visualization:



From the NMI and ARI values that we obtain, we observe that the values for both the accuracy measures are very high and hence, indicate good clustering.

In [32]:
```
'''
Ground truth for Football Network
'''
graph = nx.read_gml ('C:/Users/ayesh/Downloads/football.gml', label = 'id')
gt_membership = [graph.nodes[v]['value'] for v in G.nodes()]

'''
Calculating NMI and ARI measures
'''
#print(gt_membership)
print('NMI value: ', normalized_mutual_info_score(gt_membership, colors))
print('ARI value: ', adjusted_rand_score(gt_membership,colors))
```

```
NMI value:  0.8171749298888212
ARI value:  0.6264013643740304
```

2. Identifying communities in dynamic networks using information dynamics

We have made a custom 15 node dynamic data set in text file to implement this algorithm.

The figures contain the communities formed at each time stamp, and also the change in the community subgraph (shown in red)

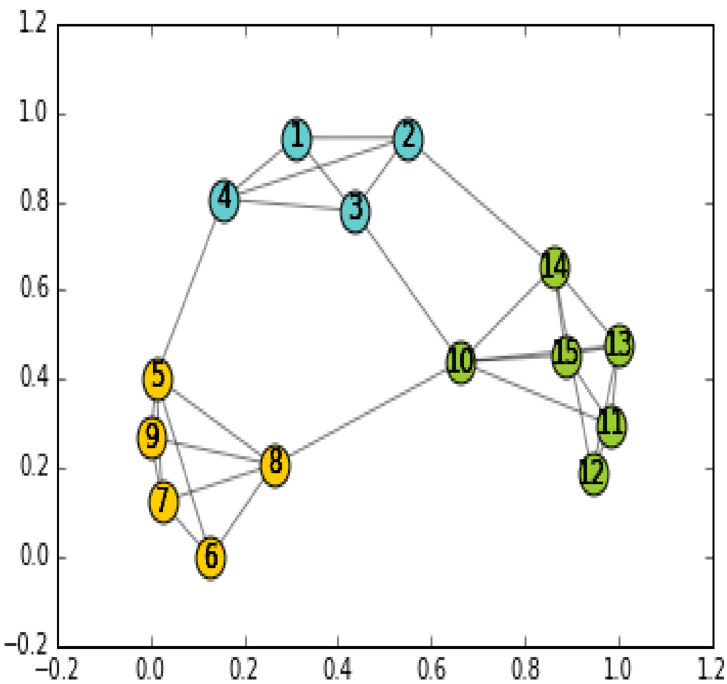


Fig: Community T0

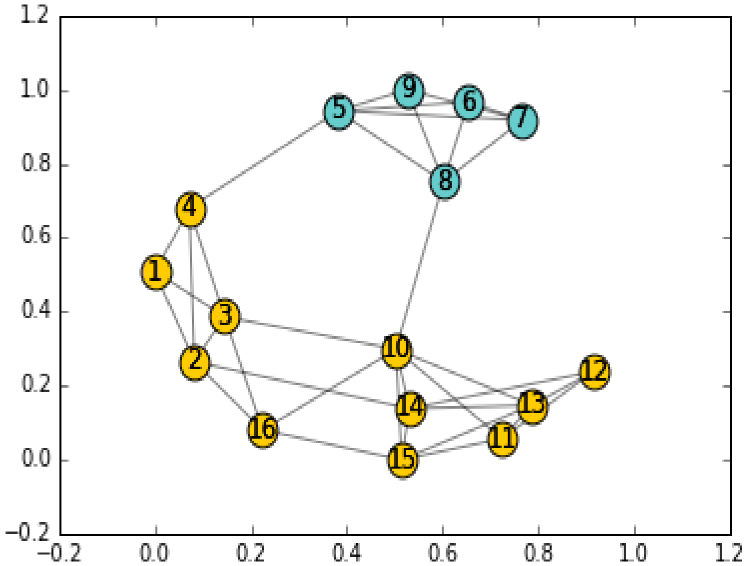


Fig: Community T1

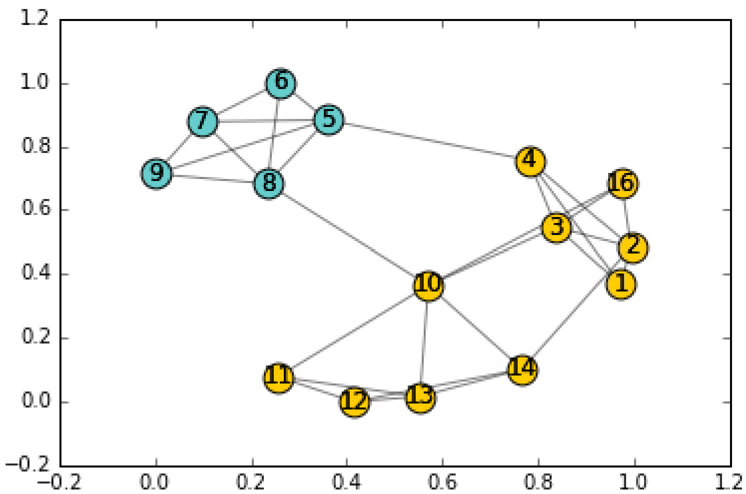


Fig: Community T2

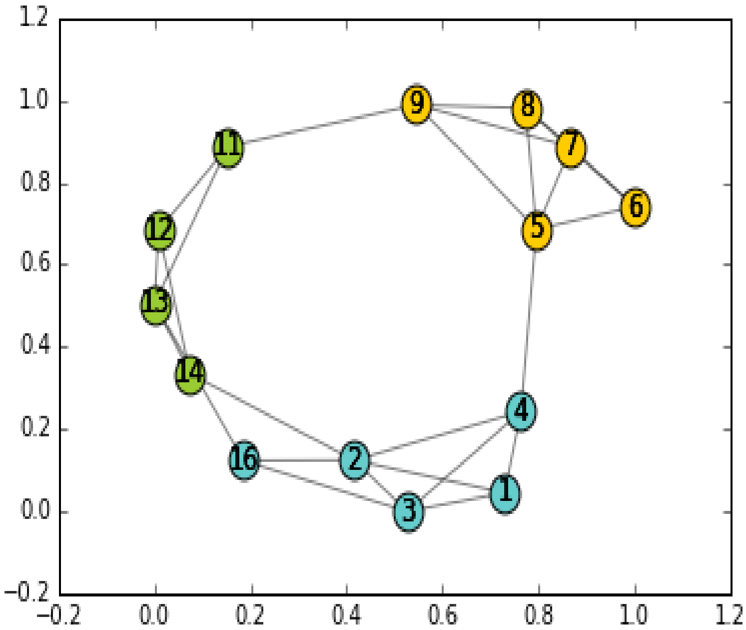


Fig: Community T3

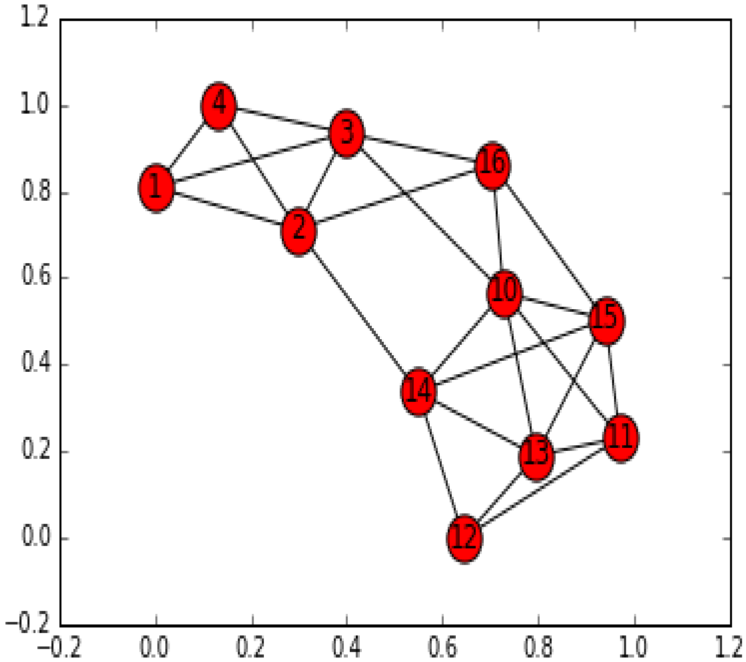


Fig: Delta g1

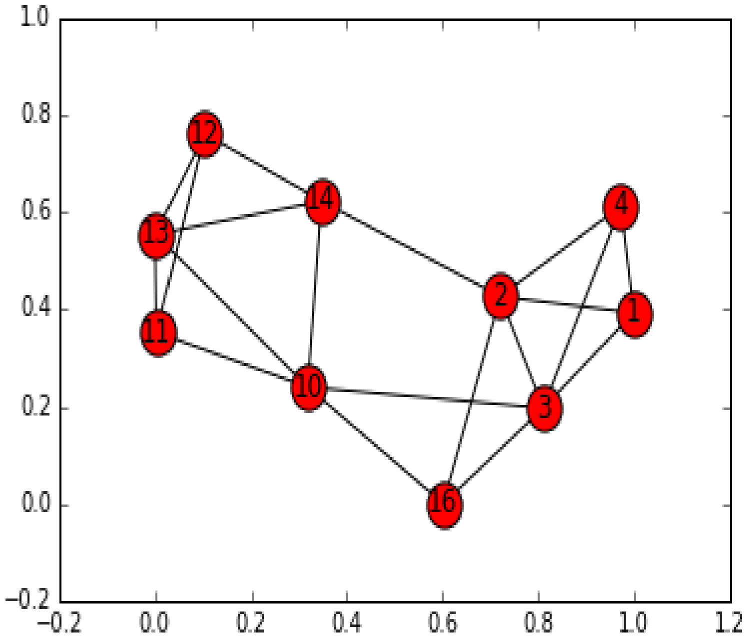


Fig: Delta g2

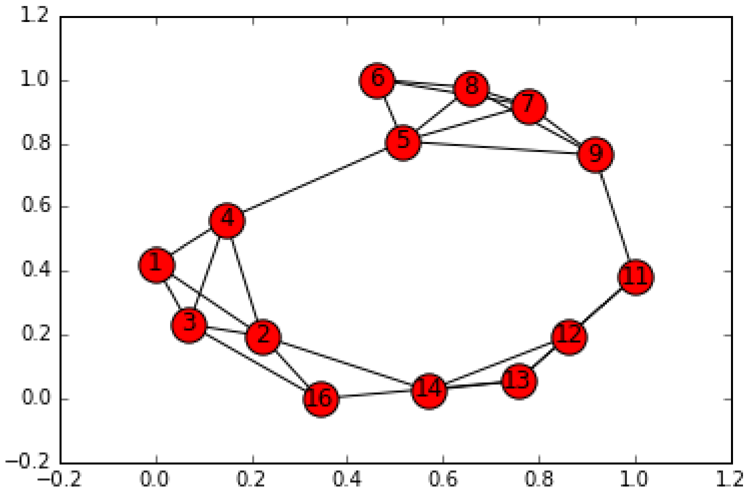

1.2
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
−0.2
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1
2
3
4
5
6
7
8
9
11
12
13
14
16

Fig: Delta g3

Conclusion

For the first research paper, Community detection using spreading process- the algorithm works well and gives a high value of NMI and ARI, so we conclude that the clustering is efficient. It was based on the fact that two nodes from the same community more easily infect one another than two nodes from different communities do.

For the second research paper, identifying communities using information dynamics, the algorithm is very efficient because at every time stamp, it calculates the changed subgraph instead of going through the network as a whole again. The NMI and ARI values are high for this algorithm too, indicating strong clustering.