Homework 3 Report for EE232E

Lei Ding

Yi Ding

Sonu Mishra

In this assignment, we will study a real network and we are going to study the properties of this real network.

Question 1:

Is this network connected? If not, find out the giant connected component. And in the following, we will deal with this giant connected component.

Solution:

In this question, we first use read.graph() to read the given graph as directed. Then we use function is.connected() to detect whether this graph is connected and we find that it is not. After acquiring the largest component, we find that the size of the component' vertices is 10487 which is very close to the original network with 10501 vertices.

Question 2:

Measure the degree distribution of in-degree and out-degree of the nodes.

Solution:

In this question, we use the degree() function to get the giant connected component's in degree and out degree. And use hist to plot the degree distribution as in Figure 2.1 and Figure 2.2. We can find that both the in degree distribution and the out degree distribution are similar to half Gaussian distribution, where most nodes' degrees are very small (less than 100) and very few nodes' degrees are larger than 250.

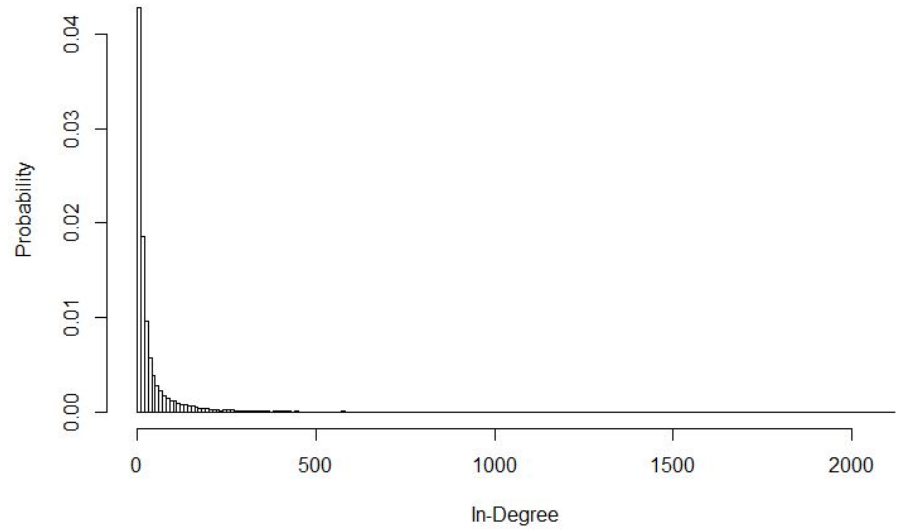

Figure 2.1 In-Degree distribution

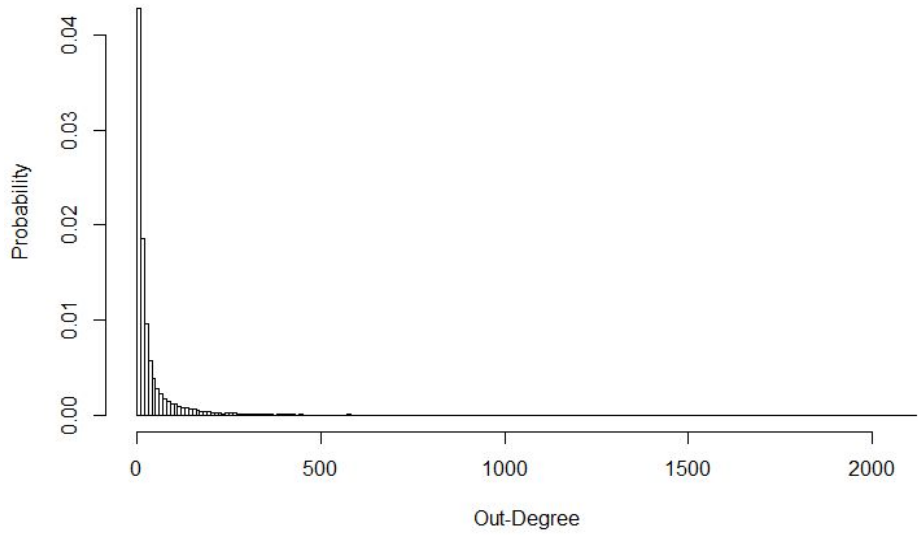

Figure 2.2 Out-Degree distribution

Question 3:

We would like to measure the community structure of the network. First, we need to convert it into an undirected network. Then we use fastgreedy.community and label.propagation.community to measure the community structure. Are the results of these two methods similar or not?

Solution:

There are two ways to convert the directed network into undirected:

(a)

In the first way, we simply remove the directions from the edges and thus form the undirected graph. The problem is that the graph now has parallel edges. For example, if there were two directed edges $u \rightarrow v$ and $v \rightarrow u$, now both of them have been converted to undirected edges u---v. Therefore, the resulting undirected graph is no longer simple.

To detect the community structure we have to use a community detection method that can handle non-simple graphs. One such method is *label propagation* community detection algorithm. This is available as `label.propagation.community` function in R.

Label propagation is a nearly linear complexity approach in which every node is assigned one of k labels. The method then proceeds iteratively and, in each iteration, each node takes the most frequent label of its neighbors in a synchronous manner. The method stops when the label of each node is one of the most frequent labels in its neighborhood i.e. no further change is possible.

We converted the graph into undirected using the method described above and to measure the community structure we used label.propagation.community.

Communities →	1	2	3	4	5	6
#Vertices →	10,469	4	5	3	3	3

We observe that there is a huge community that contains 10,469 / 10487 vertices in the graph, and various tiny communities. This can be seen from Figure 3.1. The modularity is **0.0001698002**.

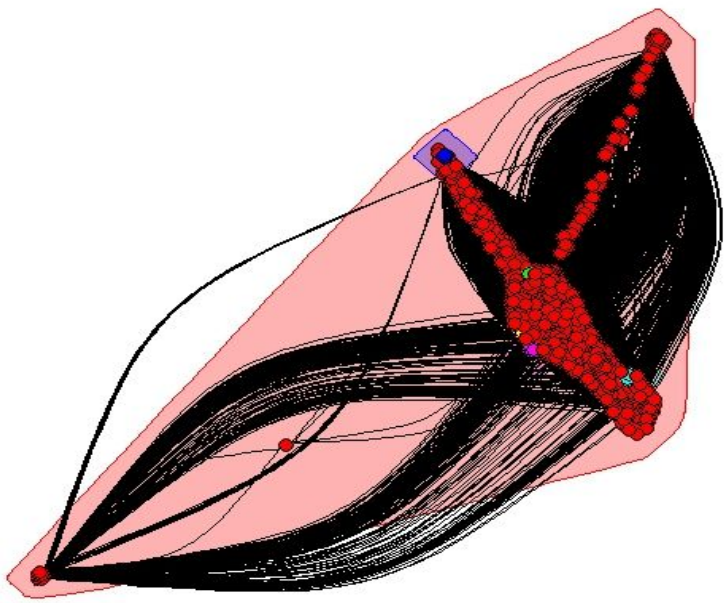

Figure 3.1 Community structure using Label Propagation method on non-simple graph

Although this is a very fast algorithm, it yields different results based on the initial configuration (which is decided randomly). We ran this for 5 times, and obtained different results.

Modularity: **0.0001698002**

Communities →	1	2	3	4	5	6
#Vertices →	10,469	4	5	3	3	3

Modularity: **0.0001698002**

Communities →	1	2	3	4	5	6
#Vertices →	10,469	4	5	3	3	3

Modularity: **0.0001494257**

Communities →	1	2	3	4	5
#Vertices →	10,472	4	5	3	3

Modularity:**0.0001698002**

Communities →	1	2	3	4	5	6
#Vertices →	10,469	4	5	3	3	3

Modularity: **0.0001024262**

Communities →	1	2	3	4	5
#Vertices →	10,474	4	3	3	3

Therefore, the method should be run a large number of times (say 1000) and then consensus labeling should be used to determine the most likely community structure. However, this will be a very tedious task. As in our case, the overall structure of the community remains more or less same (one huge community and several tiny communities) in each run, we can do with the result of any one run.

(b)

Another way to convert the directed graph into undirected is to merge the two directed edges between u and v so that the resulting network is simple. Suppose the weights are w_{uv} and w_{vu} respectively, and we set the weight for the merged undirected edge to be $\sqrt{w_{uv}w_{vu}}$. The resulting graph is simple and therefore both *fastgreedy* and *label propagation* can be used to measure the community structure.

Fastgreedy is a bottom-up hierarchical approach of community detection. This is a greedy approach that strives to maximize modularity. Initially, every vertex is assigned a separate community. In each iteration, the communities are merged in a way that each merge yields the largest increase in the modularity. The algorithm stops when the modularity converges. The method is fast and it has no parameters to tune. This is available as `fastgreedy.community` function in R.

Communities →	1	2	3	4	5	6	7	8
# Vertices →	1856	1666	1022	2226	731	1236	633	1077

As this is a hierarchical method, we obtain large number of modularity values corresponding to each merge. The mean modularity value is **0.2626252** and maximum modularity is **0.3287988**. The community structure is shown in Figure 3.2.

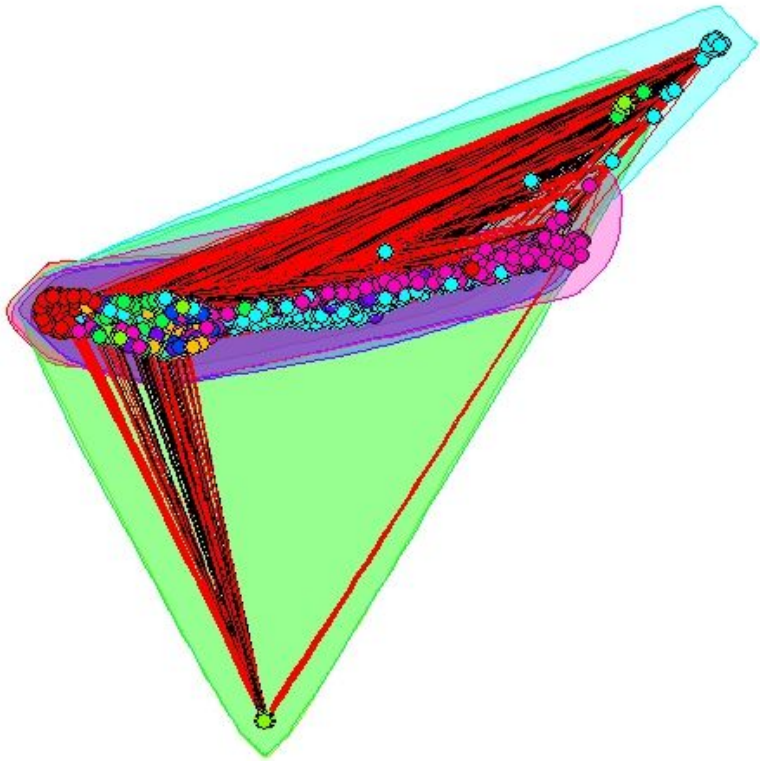

Figure 3.2 Community structure using Fast Greedy method on simple graph

This has a much higher value of modularity than the fast *label propagation*. This is because this detected a number of communities of similar sizes, unlike label propagation where most nodes belonged to just one community. Therefore the community structure detected by *label propagation* is more modular and the modularity value is higher.

Unlike *label propagation*, this does not give very different results each time and therefore one run is sufficient to get the optimal community structure. However, it is known to suffer from a resolution limit, i.e. communities below a given size threshold depending on the number of nodes and edges will always be merged with neighboring communities.

Now we try *label propagation* algorithm on the same undirected simple graph. Below are the results. Similar to earlier results, the different runs of this algorithm yields different results. Only one of them is shown here.

Modularity: **0.0001698002**

Communities →	1	2	3	4	5	6
#Vertices →	10,469	4	5	3	3	3

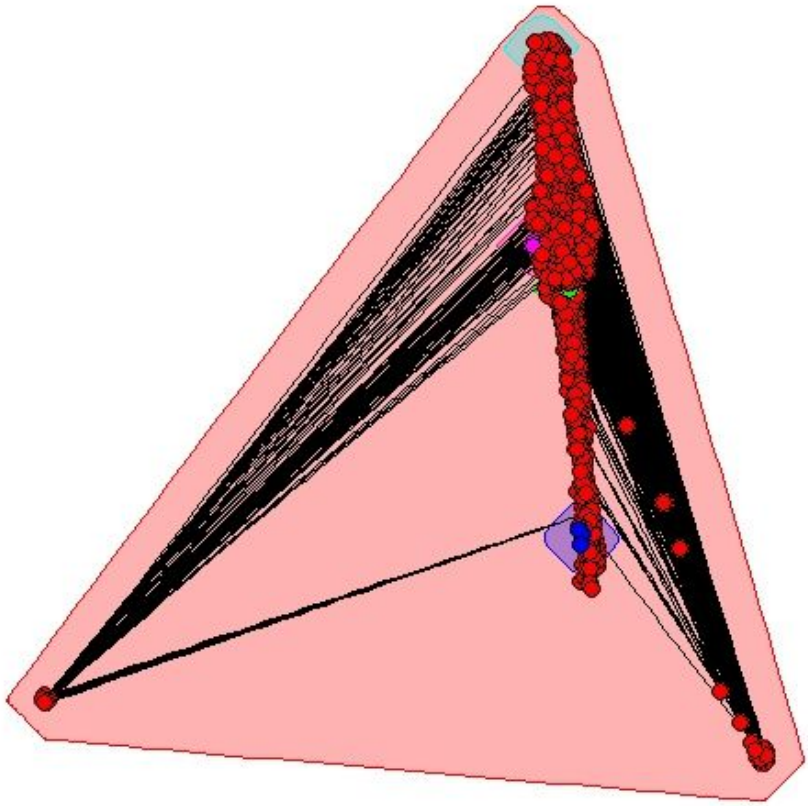

Figure 3.3 Community structure using Label Propagation method on simple graph

Question 4:

Find the largest community computed from fastgreedy.community. Isolate the community from other parts of the network to form a new network, and then find the community structure of this new network. This is the sub-community structure of the largest community.

Solution:

In this question, we are to process the largest community computed from fastgreedy.community. Based on question 3, we got a undirected graph from the giant connected component and then we use the function fastgreedy.community() to find the graph's structure which is shown in Table 4.1 and find that the largest community contains 2266 vertices. Then we isolate the largest part of this structure and detect its sub-community structure which is shown in Figure 4.1 and Table 4.2. And the modularity of this sub-community is 0.3595153, which is not large. It shows that we cannot divide this community into subcommunities very well.

Table 4.1 Structure of Giant Connected Component

ID	1	2	3	4	5	6	7	8
#Vertice	1856	1666	1022	2266	731	1236	633	1077

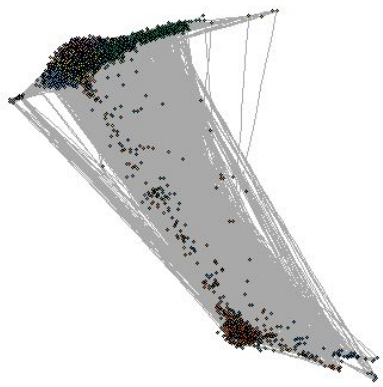

Figure 4.1 Structure of the largest community

Table 4.2 Structure of the largest community

SubCom	1	2	3	4	5	6	7	8
V_Num	306	457	313	365	426	347	47	5

Question 5:

Find all the sub-community structures of the communities with sizes larger than 100.

Solution:

In this question, based on the results from question 3, we first isolate communities whose sizes are larger than 100. And then we analyze each isolated communities respectively. The structures of these communities are shown in Table 5.1~5.8. To make them more clear, we barplot the community size of each subcommunity as shown in Figure 5.1 and 5.2.

We can find that generally, fastgreedy.community's modularity is larger than that of label.propagation and "fastgreedy" can generate more subcommunities than "label". "Label" tends to detect a very large subcommunity with other small communities while "fastgreedy" usually generates a more even set of subcommunities. So we may conclude that fastgreey.community method usually works better thatn the label.propagation.community.

Table 5.1 Community Structure for Community#1

Community#1 for fastgreedy.community modularity = 0.2249632							
SubCom	1	2	3	4	5	6	7
#Vertice	244	452	413	490	84	136	37
Community#1 label.propagation.community modularity = 0.0001301893							
SubCom	1			2			
#Vertice	1853			3			

Table 5.2 Community Structure for Community#2

Community#2 for fastgreedy.community modularity = 0.3711271									
SubCom	1	2	3	4	5	6	7	8	9
#Vertice	357	491	347	294	128	28	13	5	3
Community#2 label.propagation.community modularity = 0.0002311181									
SubCom	1		2		3		4		
#Vertice	1661		3		1		1		

Table 5.3 Community Structure for Community#

Community#3 for fastgreedy.community modularity = 0.5128581									
SubCom	1	2	3	4	5	6	7	8	9
#Vertice	318	209	30	193	118	44	41	15	10
Subcom	10	11	12	13	14	15	16	17	18

#Vertice	6	6	4	3	3	3	11	4	4
Community#3 label.propagation.community modularity = 0.4858320676									
SubCom	1	2	3	4	5	6	7	8	9
#Vertice	258	233	339	5	18	5	4	11	10
SubCom	10	11	12	13	14	15	16	17	18
#Vertice	8	14	6	15	6	8	4	6	6
SubCom	19	20	21	22	23	24	25	26	27
#Vertice	6	6	8	4	7	5	3	3	6
SubCom	28	29	30	31	32				
#Vertice	3	3	5	3	4				

Table 5.4 Community Structure for Community#4

Community#4 for fastgreedy.community modularity = 0.3595153								
SubCom	1	2	3	4	5	6	7	8
#Vertice	306	457	313	365	426	347	47	5
Community#4 label.propagation.community modularity = 0								
SubCom	1							
#Vertice	2266							

Table 5.5 Community Structure for Community#5

Community#5 for fastgreedy.community modularity = 0.3980826								
SubCom	1	2	3	4	5	6	7	8
#Vertice	60	247	138	56	55	50	74	16
SubCom	9	10	11	12	13	14	15	
#Vertice	11	4	4	2	7	3	4	
Community#5 label.propagation.community modularity = 0.2961031064								
SubCom	1	2	3	4	5	6	7	
#Vertice	589	109	4	15	6	5	3	

Table 5.6 Community Structure for Community#6

Community#6 for fastgreedy.community modularity = 0.3986121

SubCom	1	2	3	4	5	6	7	8	9
#Vertice	275	294	184	47	167	66	100	93	10
Community#6 label.propagation.community modularity = 0.0533002610									
SubCom	1	2	3	4	5	6	7	8	
#Vertice	1151	56	10	4	5	3	3	4	

Table 5.7 Community Structure for Community#7

Community#7 for fastgreedy.community modularity = 0.4796647								
SubCom	1	2	3	4	5	6	7	8
#Vertice	162	64	153	65	60	50	38	12
SubCom	9	10	11	12	13	14	15	16
#Vertice	7	3	3	4	3	3	3	3
Community#7 label.propagation.community modularity = 0.3600283900								
SubCom	1	2	3	4	5	6	7	8
#Vertice	401	148	10	13	3	15	14	5
SubCom	9	10	11	12	13	14	15	
#Vertice	3	4	4	4	3	3	3	

Table 5.8 Community Structure for Community#8

Community#8 for fastgreedy.community modularity = 0.5036454									
SubCom	1	2	3	4	5	6	7	8	9
#Vertice	190	253	145	153	75	49	88	80	16
SubCom	10	11	12	13					
#Vertice	6	11	4	7					
Community#8 label.propagation.community modularity = 0.4048298604									
SubCom	1	2	3	4	5	6	7	8	9
#Vertice	50	186	586	28	40	28	36	14	4
SubCom	10	11	12	13	14	15	16	17	18
#Vertice	32	7	5	5	6	7	3	4	4
SubCom	19	20	21	22	23	24	25	26	
#Vertice	3	6	5	4	3	4	4	3	

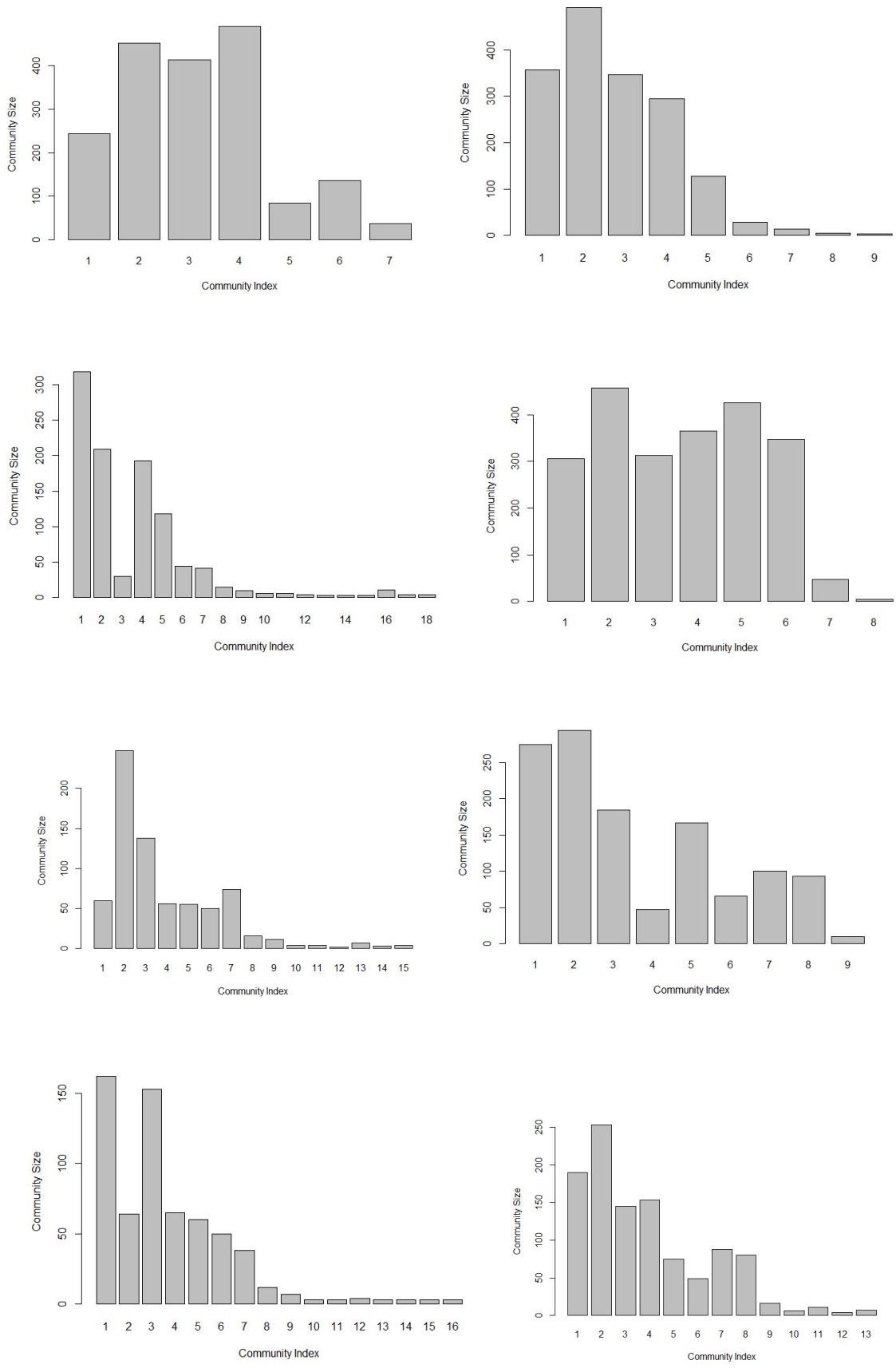

Figure 5.1 Subcommunities' structure for Fastgreedy.community (#1 to #8)

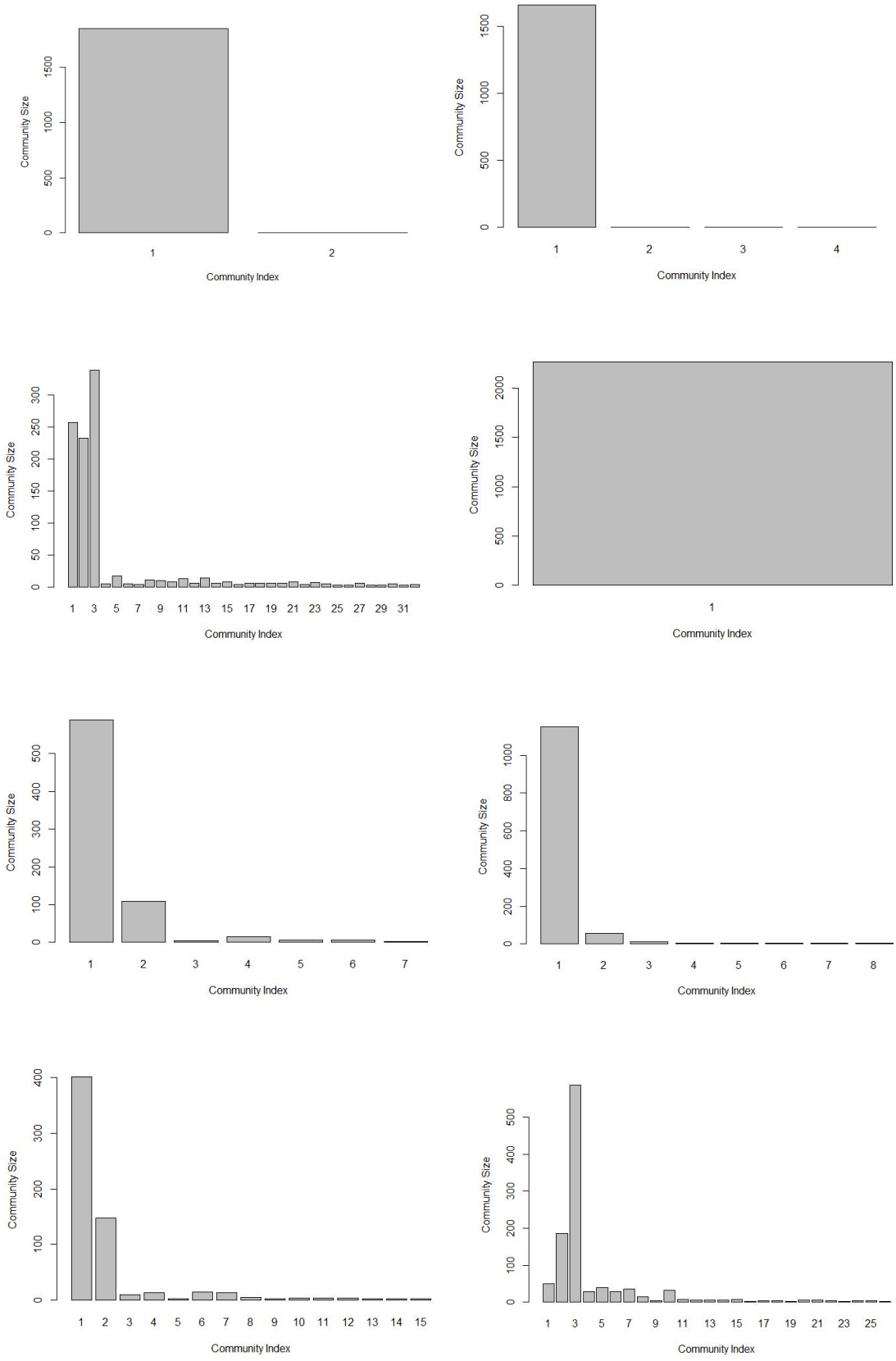

Figure 5.2 Subcommunities' structure for label.propagation.community (#1 to #8)

Question 6:

Both fastgreedy.community and label.propagation.community assume that each node belomgs to only one community. However, in practice, a node can belong to two or more communities at the same time. Thus, here we use personalized PageRank to study the overlapped community structures.

Solution:

To solve this problem, we first start the random walk from each node with a damping parameter equals to 0.85 in the original directed network. During the process, we get the visiting probabilities of all nodes in the network. Suppose we do random walk on node *i*, using the community structure we get from question 3, we compute:

$$\boldsymbol{M}_i = sum(v_j \boldsymbol{m}_j)$$

Where v_j is the visiting probability of node *j* and $\boldsymbol{m}_j$ is its community membership computed from question 3. Thus $\boldsymbol{M}_i$ is a measurement of the multiple memberships. Since to compute all nodes is costly, we just select top 30 nodes in computation. Once $\boldsymbol{M}_i$ is ready, we further select a threshold to see which node belongs to multiple communities.

First, we use *label.propagation.community* to find the nodes with multiple communities. In this case, we choose the threshold to be 0.1. Both nodes and it's probabilities in each communities are shown in table 6.1, where nodes with multiple communities are marked with color.

Table 6.1 Nodes with multiple communities using propagation

Node	Com. 1	Com. 2	Com. 3	Com. 4	Com. 5	Com. 6	Com. 7
1157	0.40	0.29	0.00	0.00	0.00	0.00	0.00
1158	0.42	0.29	0.00	0.00	0.00	0.00	0.00
10015	0.60	0.00	0.00	0.00	0.00	0.00	0.19
10176	0.36	0.00	0.39	0.00	0.00	0.00	0.00
10177	0.38	0.00	0.35	0.00	0.00	0.00	0.00
10178	0.35	0.00	0.50	0.00	0.00	0.00	0.00
10179	0.36	0.00	0.35	0.00	0.00	0.00	0.00
10348	0.40	0.00	0.00	0.40	0.00	0.00	0.00
10349	0.38	0.00	0.00	0.36	0.00	0.00	0.00
10350	0.34	0.00	0.00	0.50	0.00	0.00	0.00
10351	0.37	0.00	0.00	0.48	0.00	0.00	0.00
10352	0.36	0.00	0.00	0.49	0.00	0.00	0.00
10401	0.36	0.00	0.00	0.00	0.27	0.00	0.00

10402	0.37	0.00	0.00	0.00	0.34	0.00	0.00
10403	0.41	0.00	0.00	0.00	0.28	0.00	0.00
10464	0.43	0.00	0.00	0.00	0.00	0.48	0.00
10465	0.45	0.00	0.00	0.00	0.00	0.37	0.00
10466	0.44	0.00	0.00	0.00	0.00	0.38	0.00
10475	0.51	0.00	0.00	0.00	0.00	0.00	0.34
10476	0.46	0.00	0.00	0.00	0.00	0.00	0.44
10477	0.48	0.00	0.00	0.00	0.00	0.00	0.49

From the table, we can see that node 1157, 1158, 10015, 10176~10179, 10348~10352, 10401~10403, 10464~10466, and 10475~10477 have multiple communities.

As we can see in the above table, choosing a threshold is very important. Here we choose 0.1, but if we choose a small threshold, there will be hundreds of nodes that might belong to multiple communities with very small probability, which is not what we expect to see. On the other hand, if we choose a big threshold, like 0.2, than node 10015 will be missing. And if we choose an even larger threshold, we will kind of missing many important nodes.

Also, we can see that these nodes comes in group, such as 10176~10179, 10348~10352, 10401~10403, 10464~10466, and 10475~10477. These contiguous nodes all have multiple communities, which indicates that there exists spatial continuity of vertices in the network.

Then, we use *fastgreedy.community* to do the same thing as above. The result is similar and we show some example nodes in table 6.2. This time, we select threshold to be 0.2.

Table 6.2 Example nodes computed using fastgreedy with threshold 0.2

Node	Com. 1	Com. 2	Com. 3	Com. 4	Com. 5	Com. 6	Com. 7	Com. 8
961	0.03	0.07	0.00	0.27	0.00	0.02	0.01	0.27
2511	0.20	0.05	0.00	0.38	0.00	0.03	0.01	0.01
4031	0.08	0.25	0.02	0.21	0.01	0.03	0.00	0.03
7937	0.01	0.38	0.01	0.21	0.00	0.01	0.00	0.04
9528	0.10	0.04	0.22	0.20	0.00	0.01	0.00	0.00
10280	0.03	0.14	0.02	0.20	0.01	0.22	0.01	0.00

The reason we choose 0.2 for *fastgreedy.community* is because when we try 0.1, there are hundreds of nodes belong to multiple communities, which is not expected. Then we increase the threshold to 0.2 and compute the result. Then if we further increase the threshold to 0.3, we try it and only get one result shown in the following table.

Table 6.3 Nodes computed using fastgreedy with threshold 0.3

Node	Com. 1	Com. 2	Com. 3	Com. 4	Com. 5	Com. 6	Com. 7	Com. 8
7887	0.02	0.1	0.01	0.33	0.01	0.37	0.01	0.01

As we can also see from the table, the result is quite different from above. In *label.propagation.community*, we can see nodes belong to two communities only have probability values in corresponding two communities while in table 6.2 and 6.3, we find nodes have many small probabilities in communities rather than two main communities. Also, number of nodes with multiple communities with fastgreedy is much more than that of propagation even with a higher threshold. There are because that *fastgreedy.community* tends to make more sub-communities than *label.propagation.community*.