EE 232E Graphs and Network Flows
Homework 3

Professor Roychowdhury

Prepared by:

Arunav Singh (304 760 844)
Eric Goldfien (603 887 003)
Steven Leung (304 777 142)

NOTE: This assignment was coded in R v2.15.2 with iGraph v0.7.0 and netrw v0.2.6.

Question 1

The graph generated from the edgelist in the file *sorted_directed_net.txt* is **not connected**. The Giant Connected Component (GCC) was found by using iGraph's *clusters()* function with "*strong*" as a parameter to find all strongly connected clusters.The largest one was chosen while all nodes not belonging to that cluster are deleted. The result is the GCC.

Question 2

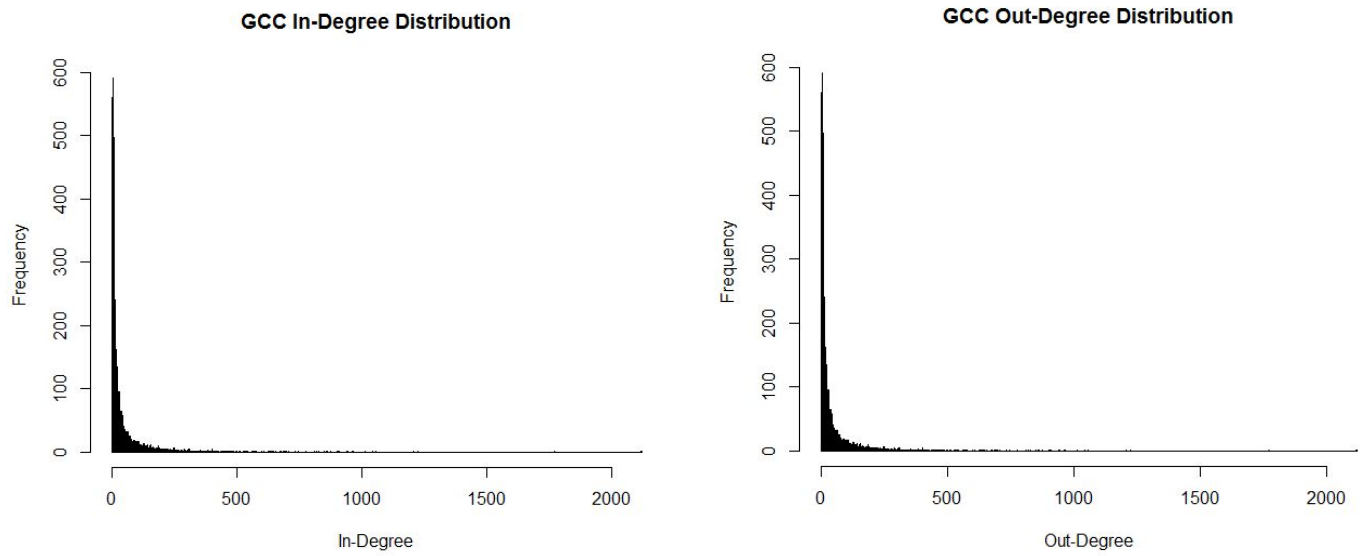

Figure 2.1 In-Degree and Out-Degree Distribution

The histograms of the in and out degrees of the GCC are virtually identical. This follows our intuition; since the GCC is strongly connected it makes sense that for every path from node *i* to node *j* there is a path from node *j* to node *i* ensuring that the for every node with out-degree *k* there must be an equivalent node with in-degree *k*.

Question 3

To convert the directed graph to undirected we use the *as.undirected()* function on the GCC. For *Option 1* the parameter "*each*" is used in the *as.undirected()* function which simply changes all edges from directed into undirected. For *Option 2* parameter "*collapse*" is used in the *as.undirected()* function which combines edges and results in a network without multiple edges between two nodes, *Option 2* also uses the parameter "*edge.attr.comb*" which sets the weight of the combined edge based on the function handle passed to it. Our function "*edge_weights*" was passed to *"edge.attr.comb"* will perform the following calculation:

$$\sqrt{w_{ij} * w_{ji}}$$

After the GCC is converted to undirected we wish to find it's community structure. For *Option 1* *label.propagation.community()* is used and for *Option 2* both *label.propagation.community()* and *fastgreedy.community()* are used independently to find the community structure. The results, based on *Option 1* and *2* and the choice of community finding algorithm are displayed below in *Tables 3.1 - 3.3.* Note that Option 1 and 2 are exactly the same when *label.propagation.community()* is used in option 2.

Option 1:

Modularity: 0.0001698

Community	1	2	3	4	5	6
Size	10469	4	3	3	3	5

Table 3.1 Community Structure for Option 1

Option 2: Using label.propagation.community()

Modularity: 0.0001698

Community	1	2	3	4	5	6
Size	10469	4	3	3	3	5

Table 3.2 Community Structure for Option 2 using *label.propagation.community()*

Option 2: Using fastgreedy.community()

Modularity: 0.328771

Community	1	2	3	4	5	6	7	8
Size	1836	791	1701	1213	2316	634	963	1033

Table 3.3 Community Structure for Option 2 using *fastgreedy.community()*

Question 4

To find the Sub-Communities of the GCC we used the community structure found by *fastgreedy.community()* on the network with collapsed edges. As seen from *Table 3.3*, the largest community of the GCC has 2316 nodes. All nodes that did not belong to that sub-community were deleted to isolate that largest community and

create a new network. The community structure of this new network is shown below in *Table 4.1* and is known as the sub-community structure of the largest community.

Sub-Community Structure of Largest Community:

Modularity: 0.362693

Community	1	2	3	4	5	6	7	8
Size	39	378	417	370	32	301	341	438

Table 4.1 Sub-Community Structure of Largest Community

Question 5

From *Table 3.3* it is seen that there are 8 communities of size larger than 100. We used a for loop to iterate through all 8 sub-communities performing the same analysis as in question 4 to find their community structures. Below are the community structures for all 8 sub-communities of size greater than 100. Note that the ordering of these sub-communities is not of importance, it is simply our method of numbering them.

Sub-community 1:

Modularity: 0.1925071

Community	1	2	3	4	5	6	7	8	9	10
Size	219	611	596	343	12	8	4	6	3	4

Community	11	12	13	14	15	16	17	18	19	20
Size	2	2	2	2	8	4	3	2	3	2

Table 5.1 Community Structure for Sub-Community 1

Sub-community 2:

Modularity: 0.3915805

Community	1	2	3	4	5	6	7	8
Size	224	165	201	78	23	20	19	11

Community	9	10	11	12	13	14	15
Size	10	11	8	4	4	7	6

Table 5.2 Community Structure for Sub-Community 2

Sub-community 3:

Modularity: 0.2727445

Community	1	2	3	4	5	6	7	8
Size	591	32	111	551	369	12	5	5

Community	9	10	11	12	13	14	15
Size	5	6	4	3	2	3	2

Table 5.3 Community Structure for Sub-Community 3

Sub-community 4:

Modularity: 0.3340657

Community	1	2	3	4	5	6	7	8	9	10	11
Size	86	274	320	21	377	91	13	19	4	5	3

Table 5.4 Community Structure for Sub-Community 4

Sub-community 5:

Modularity: 0.2836786

Community	1	2	3	4	5	6	7	8	9
Size	737	446	236	35	767	22	11	4	7

Community	10	11	12	13	14	15	16	17	18
Size	6	5	19	7	4	3	2	3	2

Table 5.5 Community Structure for Sub-Community 5

Sub-community 6:

Modularity: 0.4707233

Community	1	2	3	4	5	6	7	8
Size	159	23	86	75	150	33	28	15

Community	9	10	11	12	13	14	15	16
Size	9	8	25	8	5	4	3	3

Table 5.6 Community Structure for Sub-Community 6

Sub-community 7:

Modularity: 0.4374482

Community	1	2	3	4	5	6	7	8
Size	369	155	245	31	72	24	23	9

Community	9	10	11	12	13	14	15	16
Size	4	5	4	5	8	2	4	3

Table 5.7 Community Structure for Sub-Community 7

Sub-community 8:

Modularity: 0.4255128

Community	1	2	3	4	5	6	7	8	9	10	11
Size	241	256	291	49	42	67	13	6	11	6	4

Community	12	13	14	15	16	17	18	19	20	21	22
Size	5	6	4	6	6	4	3	3	4	3	3

Table 5.8 Community Structure for Sub-Community 8

Question 6:

To find examples of nodes that belong in different communities from using the *fastgreedy.community()* and *label.propagation.community()* algorithms, the following needs to be calculated:

- The visiting probability of all nodes with a random walker starting at a specific node *j* with a teleportation probability to node *j* with 15% probability (damping = 0.85). Only the top 30 visit probabilities will be used for each starting node to speed up calculations. Each of nodes used will be referred to v_j depending on the iteration (1:30). Note that v_j is a scalar and not a vector.

- The community membership (m_j) of each node based on the algorithm used (refer to Question 3). The format for this is that the corresponding index (of m_j) of the community that a specific node is in will be 1 with all other elements being 0.

The following equation can then be used to calculate a measurement of multiple memberships (M_i). Note that this (M_i) calculation is independent for each node in the original directed network.

$$M_i = \sum_j v_j m_j$$

Finally, a threshold can be set on the measurement of multiple memberships (M_i) to decide which nodes belong to multiple communities. Figures *6.1 - 6.3* summarize the results for the 2 different methods of community structure defined in Question 3. The number of nodes that belong to multiple communities are completely dependent on the threshold value set. The lower the threshold value, the more nodes are declared to be a part of multiple communities (which follows our intuition). The threshold value was selected so that there are not too many nodes that belong to multiple communities and this value was fixed for all options.

Option 1: Change all edges to undirected and use label.propagation.community ()

Number of Nodes	1	2	3
Nodes (Threshold = 0.4)	7372	7373	N/A
Nodes (Threshold = 0.3)	4966	7372	7373

Table 6.1 Summary of Nodes Belonging to Multiple Communities (Option 1)

Option 2: Merging two directed edges

Using *label.propagation.community()*:

Number of Nodes	1	2	3	4	5	6	7	8	9	10
Nodes (Threshold = 0.4)	6794	6795	8362	8363	10079	10080	N/A	N/A	N/A	N/A
Nodes (Threshold = 0.3)	6794	6795	7895	8301	8362	8363	8875	8887	10079	10080

Table 6.3 Summary of Nodes Belonging to Multiple Communities (Option 2 With *label.propagation.community*)

Using *fastgreedy.community():*

Number of Nodes	1	2
Nodes (Threshold = 0.4)	7372	7373
Nodes (Threshold = 0.3)	7372	7373

Table 6.3 Summary of Nodes Belonging to Multiple Communities (Option 2 With *fastgreedy.community*)