Social Computing [CS60017]

Assignment 1

Instructions

1. This assignment is meant to use SNAP (`http://snap.stanford.edu/`) which is a library for handling very large graphs efficiently, and provides a large number of functions for analyzing graphs. It has two versions, one for C++ and another for Python. You are free to use either C++ or Python language.

2. At the beginning of each source code file, set the seed of the random number generator of SNAP to **42**.

3. Your codes should print out *exactly* what is asked, and in the specified format. There will be a **penalty of 20 marks** if the specified output format is not followed.

4. **How and what to submit:** Solutions should be uploaded via the Piazza link for the course (see course website). Submit one .zip or .tar.gz file containing a compressed folder that should contain all source codes, all files to be submitted (as per the task descriptions given below). Name the compressed file the same as your roll number. Example: name the compressed file 19CS60R00.zip or 19CS60R00.tar.gz if your roll number is 19CS60R00.

5. Also submit an additional text file **instructions.txt** where you should state how to run your codes as well as any additional information you want to convey, such as the version of Python or C++ compiler. The instructions.txt file should also contain your name and roll number.

6. We should be able to run your submitted code in a computer with a reasonable configuration (for instance 2GB or more RAM) by following your submitted instructions. If any part of your code takes a long time to run (e.g., more than 10 minutes) report that in the instruction file with an estimate of time required.

7. The assignment should be done individually by each student. You should not copy any code from one another, or from any Web source. Anyone who is found to have copied any of the codes will be awarded zero for the whole assignment.

8. **Deadline for submission: September 7th, 2019 (firm deadline)**

Problem 1 [50 points]

In this problem, we will get familiar with SNAP and use it to programmatically compute structural properties of some real world networks.

Datasets

Download the networks from Table 1 (the gzipped files in the Files section). Each graph is represented as an edge list. For this exercise assume that the networks are undirected.

Network	**Description**	**Network download location**
soc-Epinions1	Epinions network	`https://snap.stanford.edu/data/soc-Epinions1.html`
cit-HepPh	Physics citation network	`https://snap.stanford.edu/data/cit-HepPh.html`
email-Enron	Enron email	`https://snap.stanford.edu/data/email-Enron.html`
p2p-Gnutella04	Gnutella p2p network	`https://snap.stanford.edu/data/p2p-Gnutella04.html`

Table 1: Name, description and download location for networks for this task

Task 1.1 [10 points]

Once you have downloaded the above networks, *generate the sub graphs* of each using the rules in Table 2. The table also contains the files to be submitted for this task. These subgraphs will be used throughout this assignment. Generate the edge list files and save them in a folder called **subgraphs**.

Task 1.2 [40 points]

For each of the subgraphs given in Table 2, write a program **gen_structure.py** (or in cpp) to generate following structural metrics below. Your code should take the name of the subgraph (specified in Table 2) as its argument, also note what your code should do when we run your code with a <subgraph name> as argument :

1. **Size of the network** [4 points]
   (a) Number of nodes
   Your code should print a line in stdout "Number of nodes in <subgraph name>: value"

Subgraph name	**Rule to extract the subgraph**	**Submission file name containing the subgraph as edgelist**
soc-Epinions1-subgraph	Consider only the nodes with odd numbered node ids in soc-Epinions1 network.	**soc-Epinions1-subgraph.elist** with one edge per line (each edge is two node ids separated by white space/tab)
cit-HepPh-subgraph	Consider only the nodes with even numbered node ids in cit-HepPh network.	**cit-HepPh-subgraph.elist** with one edge per line (each edge is two node ids separated by white space/tab)
email-Enron-subgraph	Consider only the nodes with node ids divisible by 3 in email-Enron network.	**email-Enron-subgraph.elist** with one edge per line (each edge is two node ids separated by white space/tab)
p2p-Gnutella04-subgraph	Consider the full graph of p2p-Gnutella04 network.	**p2p-Gnutella04-subgraph.elist** with one edge per line (each edge is two node ids separated by white space/tab)

Table 2: Rules for extracting subgraph for each of the networks and submission instructions

(b) Number of edges
Your code should print a line in stdout "Number of edges in <subgraph name>: value"

2. **Degree of nodes in the network** [6 points]

   (a) Number of nodes which have degree = 7
   Your code should print a line in stdout "Number of nodes with degree=7 in <subgraph name>: value"

   (b) Node id(s) for the node with the highest degree. Note that there might be multiple nodes with highest degree
   Your code should print a line in stdout "Node id (s) with highest degree in <subgraph name>: comma (i.e. ",") separated id(s) of nodes"

   (c) Plot of the Degree distribution
   Your code should create the plotted image in the same directory as your code and print on stdout "Degree distribution of <subgraph name> is in: <filename> "

3. **Paths in the network** [12 points]

   (a) Approximate full diameter (maximum shortest path length) computed starting from 10, 100, 1000 random test nodes. Also calculate the average and variance across these 3 estimates of the diameter.
   Your code should print lines in stdout "Approximate full diameter in <subgraph name> with sampling <number of test nodes > nodes: diameter value" and "Approximate full diameter in <subgraph name> with sampling nodes (mean and variance): mean diameter value, variance value"

(b) Approximate effective diameter computed starting from 10, 100, 1000 random test nodes. Also calculate the average and variance across these 3 estimates of the diameter.
Your code should print lines in stdout "Approximate effective diameter in <subgraph name> with sampling <number of test nodes > nodes: diameter" and in the next line "Approximate effective diameter in <subgraph name> with sampling nodes (mean and variance): mean diameter value, variance value"

(c) Plot of the distribution of the shortest path lengths in the network.
Your code should create the plotted image in the same directory as your code and print on stdout "Shortest path distribution of <subgraph name> is in: <filename> "

4. **Components of the network** [8 points]

(a) Fraction of nodes in the largest connected component
Your code should print a line in stdout "Fraction of nodes in largest connected component in <subgraph name>: value"

(b) Number of edge bridges. An edge is a bridge if, when removed, increases the number of connected components.
Your code should print a line in stdout "Number of edge bridges in <subgraph name>: value"

(c) Number of articulation points. A node is a articulation point if, when removed, increases the number of connected components.
Your code should print a line in stdout "Number of articulation points in <subgraph name>: value"

(d) Plot of the distribution of sizes of connected components
Your code should create the plotted image in the same directory as your code and print on stdout "Component size distribution of <subgraph name> is in: <filename> "

5. **Connectivity and clustering in the network** [10 points]

(a) Average clustering coefficient of the network (briefly explained here `https://en.wikipedia.org/wiki/Clustering_coefficient#Network_average_clustering_coefficient`). For the average clustering coefficient round to 4 decimal places while reporting.
Your code should print a line in stdout "Average clustering coefficient in <subgraph name>: value"

(b) Number of triads
Your code should print a line in stdout "Number of triads in <subgraph name>: value"

(c) Clustering coefficient of a randomly selected node. Also report the selected node id.
Your code should print a line in stdout "Clustering coefficient of random node < node id > in <subgraph name>: value"

(d) Number of triads a randomly selected node participates in. Also report the selected node id.
Your code should print a line in stdout "Number of triads random node < node id > participates in <subgraph name>: value"

(e) Number of edges that participate in at least one triad
Your code should print a line in stdout "Number of edges that participate in at least one triad in <subgraph name>: value"

(f) Plot of the distribution of clustering coefficient
Your code should create the plotted image in the same directory as your code and print on stdout "Clustering coefficient distribution of <subgraph name> is in: <filename> "

Problem 2 [50 points]

In this problem we will calculate some centrality measures of networks. For this problem too, use the edge lists from Table 2

Task 2.1 [35 points]

Write a code to compute the following centrality metrics for a graph:

1. **Degree centrality** [7 points]
2. **Closeness centrality** for node i, given by $C_i = \frac{n}{\sum_j d_{ij}}$, where d_{ij} is the length of the shortest path from i to j, and n is the number of nodes in the graph. [12 points]
3. **Betweenness centrality** for node i, given by $C_i = \sum_{st} \frac{n^i_{st}}{g_{st}}$, where n^i_{st} is the number of shortest paths between nodes s and t which pass through i, and g_{st} is the total number of shortest paths between nodes s and t. [16 points]

Your code should take the name of elist file as an input argument and should output a text file each for the centrality measures, which contains a line for each node and has the format: *nodeID <white space> centrality value*. Each file should be sorted by the centrality value.

Please note the following carefully:

1. Your code file should be named **gen_centrality.py** (or in cpp).
2. You can build your code on SNAP (using graph classes etc.).
3. There are already functions in SNAP (cpp version) called "GetBetweennessCentr" and "GetClosenessCentr". However, you should **NOT** use these functions. You need to implement these centralities yourself, as extensions to SNAP's graph framework.
4. In fact you can implement more efficient ways of computing centralities in your code. There are multiple algorithms, like variation of Floyd Warshall algorithm, Johnson's algorithm and Brandes' algorithm. You can read more about them here: `https://en.wikipedia.org/wiki/Betweenness_centrality#Algorithms` .
5. Please also mention in your **instructions.txt** file which algorithm you used for computing each centrality.
6. Before each function in your code, write (in comments) any functions or data structures (that are written by you) that this function uses. This is to ensure that you **only include functions or classes in your file that your code actually uses**.

Task 2.2 [15 points]

Calculate the **Closeness centrality** and **Betweenness centrality** of all nodes using the in built functions available in SNAP library. While calculating **Betweenness centrality**, set *NodeFrac* parameter to 0.8 to calculate approximate values (but faster!)

Your code should be in a file: **analyze_centrality.py** (or in cpp)

Finally use the output files generated in Task 2.1 and do the following for each of the two centralities:

1. Take the top 10 nodes generated by SNAP for the centrality.
2. Take the top 10 nodes generated by your implementation of the centrality, in Task 2.1.
3. Calculate how many nodes among the top 10 are same.

Your code should print a line in stdout "Number of overlaps for Closeness Centrality: <value>"

NOTE:

(1) You can use in-built modules of SNAP for all parts of the assignment, except Task 2.1 where you should NOT use the SNAP modules for computing Closeness centrality and Betweenness centrality.

(2) For any queries regarding the assignment, mail TA Soham Poddar (sohampoddar26 [at] gmail.com).