ST2: MATHEMATICAL MODELING OF PROPAGATION PHENOMENA
PROPAGATION ON GRAPHS

Instructor: Fragkiskos Malliaros
TA: Abdulkadir Çelikkanat

Monday, December 10, 2018

Description

The first lab mainly aims at familiarizing the students with fundamental elements and methods commonly used in network analysis with the *NetworkX*[1] Python package.

Part I: Analysis of a real-world network

Exercise 1: Basic properties of the network

In this exercise, the collaboration network `NetScience` will be analyzed by examining several structural properties. The `NetScience` co-authorship network has been compiled from the bibliographies of two review articles and covers the network of researchers working on the field of network science. If an author i has co-authored a paper with an author j, the graph contains an undirected edge from i to j. Here we consider that the graph is unweighted.

The graph is stored in the `NetScience.edgelist` file[2], as an edge list:

```
# Undirected graph: NetScience.edgelist
# Collaboration network of researchers working on network theory  (there
is an edge if authors coauthored)
# Nodes: 1461 Edges: 2742
# FromNodeId ToNodeId
344 342
344 343
...
```

For the following questions, it is recommended to use the *NetworkX* package of Python.

1. Load the network data into an undirected graph G, using the `read_edgelist()` function. Note that, the delimeter used to separate values is the tab character $\backslash t$ and additionally, the text that follows the # character correspond to comments. The general syntax of the function is the following:

```
read_edgelist(path, comments='#', delimiter=None, create_using=None,
    nodetype=None, data=True, edgetype=None, encoding='utf-8')
```

2. Complete the required fields in the `compute_network_characteristics()` method in order to find the following network characteristics: (1) number of nodes, (2) number of edges, (3) minimum degree, (4) maximum degree, (5) mean degree, (6) median degree of nodes in the graph and

[1] https://networkx.github.io/

[2] The data can be also downloaded from the following link: `http://vlado.fmf.uni-lj.si/pub/networks/data/collab/netscience.htm`.

(7) density of the graph. For this task, you can use the built-in functions `min, max, median, mean` of the NumPy library, and the following method provides an iterator for $(node, degree)$ pairs:

```
graph.degree()
```

Note that, the density of an undirected graph is defined as $2m/n(n-1)$, where n is the number of nodes and m is the number of edges.

Exercise 2: Connected components of the graph

A *connected component* of an undirected graph is a subgraph where every pair of nodes is connected and there is no any connection between the subgraph and the rest of the graph. In this exercise, we will consider the largest connected component of the network (also called Giant Connected Component – GCC).

1. A graph can be composed by many connected components, so we will firstly check whether the network is connected or not using the following method:

```
is_connected(G)
```

2. If it contains more than one connected components, we will extract the largest connected component (GCC). A generator for each connected component of G can be obtained using the following function:

```
connected_component_subgraphs(G, copy=True)
```

3. Now, we can find the number of nodes and edges of the largest connected component (GCC) and examine in what fraction of the whole graph they correspond to.

4. Let's visualize the giant connected component of the network, with different node colors indicating their degrees. What do you observe?

```
visualize(graph, values, node_size)
```

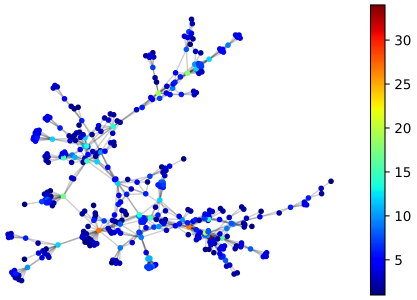

Figure 1: A visualization of the largest connected component of the network.

Part II: Analysis of clustering structures in the graph

Exercise 3: Triangle subgraphs

A triangle is a clique of three nodes, i.e., all nodes are connected to each other. Triangle subgraphs play a crucial role in the area of graph mining and social network analysis, since they are closely related to

the existence of clustering structures in the graph. Let's now compute the total number of triangles in the *NetScience* collaboration network.

1. Firstly, we will compute the number of triangles that a node participates to, by filling the missing code from the following function:

```
def count_triangles_of(graph, node):
    count = 0
    ...
    ...
    return count
```

2. Now, we can compute the total number of triangles in the graph using the previous function `count_triangles_of(graph, node)`:

```
def count_all_triangles(grap):
    count = 0
    ...
    ...
    return count
```

   Finally, we can compare the results that our implementation gives to the one obtained by the built-in function of Python `triangles(G, nodes=None)`. Note that, after summing up the number of triangles that a node participates to, this number is divided by 3 (why?) in order to compute the total number of triangles.

3. *Spectral computation of the total number of triangles*. Recall that, the (i, j)-th element of the $\boldsymbol{A}^l$ counts the number of paths of length l that start from node i and end at node j, where $\boldsymbol{A}$ is the adjacency matrix of the graph. A triangle in a graph is defined as a path of length 3 that starts and ends at the same node. Thus, the i-th diagonal element of $\boldsymbol{A}^3$ counts the number of triangles that node i participates to. Taking into account that each triangle is counted twice for each of the three participating nodes (clockwise and anticlockwise paths starting from node i), the total number of distinct triangles in the graph will be given by the following formula:

$$\Delta(G) = \frac{1}{6} tr(\boldsymbol{A}) = \frac{1}{6} \sum_{i=1}^{|V|} A_{ii}^3, \tag{1}$$

   where $tr(\boldsymbol{A})$ is the trace of a matrix. Since matrix $\boldsymbol{A}$ is a real, square matrix, then $tr(\boldsymbol{A}) = \sum_{i=1}^{|V|} \lambda_i$, and more generally, $tr(\boldsymbol{A}^l) = \sum_{i=1}^{|V|} \lambda_i^l$, where $\lambda_1, ..., \lambda_{|V|}$ are the eigenvalues of $\boldsymbol{A}$. Thus, the total number of triangles in G can be computed by the spectrum of the adjacency matrix as follows:

$$\Delta(G) = \frac{1}{6} \sum_{i=1}^{|V|} \lambda_i^3. \tag{2}$$

   This formulation offers a spectral computation of the total number of triangles $\Delta(G)$. In general, such a method for computing the number of triangles is of similar complexity as the method that is applied directly on the graph (e.g., function `triangles()` of Python), which is generally high (it can be $\mathcal{O}(|V|^3)$ for dense graphs).

   In this exercise, we will compute the number of triangles using the spectral properties of the network, defining the following function:

```
def spectral_num_of_triangles_counting(graph):

    A =  # adjacency matrix of graph
    eigval, eigvec = eigh(A) # compute eigenvalues
    count =

    return count
```

Here, it is important to note that `adjacency_matrix(G,nodelist=None,weight=weight)` returns the sparse representation of the adjacency matrix but `eigh(A)` does not support sparse matrices – so the adjacency matrix should be firstly converted into dense representation.

Exercise 4: Clustering coefficient

Next we will compute the average clustering coefficient of the graph, which is a measure of the degree to which nodes in a graph tend to cluster together, i.e., to create tightly knit groups characterized by a relatively high density of ties. The *clustering coefficient* is based on triplets of nodes. A triplet consists of three nodes that are connected by either two (open triplet) or three (closed triplet) undirected ties. A triangle consists of three closed triplets, one centered on each of the nodes.

1. The clustering coefficient of a node v is the number of triangles connected to it divided by the total number of triplets centered around node v. In other words, the clustering coefficient of node v can be expressed as

$$C(v) = \frac{2 \times number\ of\ triangles}{deg(v)(deg(v) - 1)}. \tag{3}$$

   By completing the following partially filled method, we will compute the clustering coefficient for any given node:

```
def clustering_coefficient_of(graph, node):
    cc = 0.0

    d = # get the degree of the node
    if d > 1: # if the degree is greater or equal to 2
        cc = # Compute clustering coefficient

    return cc
```

2. The *average clustering coefficient* is simply the average of the clustering coefficients of all vertices in the graph. More formally, it can be defined as

$$\bar{C} = \frac{1}{N} \sum_{v \in V} C(v), \tag{4}$$

   where N is the total number of nodes. Similarly, we can compute the average clustering coefficient of a graph using the following function:

```
    def average_clustering_coefficient(graph):
    avcc = 0.0
    ...
    return avcc
```

Part III: Degree and triangle participation distributions

Exercise 5: Analysis of the degree and triangle participation distributions of the network

The degree distribution of a network is a key property used in network analysis, and most real-world networks obey a *scale-free* property with respect to their degrees. In other words, the majority of the nodes have low degrees and only a small fraction of nodes correspond to high degree nodes.

Additionally, we will compute and plot the triangle participation distribution, i.e., a distribution that shows the number of triangles that each node participates to (how many nodes participate to one triangle, how many nodes participate to two triangles, etc).

1. Now, by filling in the required fields in `plot_histograms` function, the degree and triangle participation distributions of the network can be visualized:

```
def plot_histograms(graph):
    # degree sequence
    degree_sequence =
    degree_count_values = sorted(set(degree_sequence))
    degree_hist = [degree_sequence.count(val) for val in
                                          degree_count_values]

    # the sequence of number of triangles that each node participates
    triangle_participation =
    tr_count_values = sorted(set(triangle_participation))
    tri_hist = [triangle_participation.count(val) for val in
                                              tr_count_values]
    ...
    ...
```

2. Lastly, here we will examine how the degree distribution of a random graph looks like. That way, we will be able to examine to what extend the degree distribution observed in a real network deviates from randomness. To do so, we will generate an undirected Erdös-Rényi[3] random graph $\mathcal{G}_{n,p}$, using the following built-in function of *NetworkX*:

```
    erdos_renyi_graph(n, p, seed=None, directed=False)
```

   Here, n is the total number of nodes in the graph and $p \in (0, 1)$ is the probability of adding an edge between a pair of node. In other words, each edge among any pair of nodes is included in the graph with probability p – independent from every other edge. What do you observe? Is there any difference in the structure of random vs. real graphs with respect to the degree distribution (e.g., the *NetScience* collaboration network)?

Part IV: Graph diameter

Exercise 6: Analysis of the diameter of the graph

The *diameter* of a network is defined as the longest of all shortest paths between any pair of nodes in the network. In this exercise, we will compute the *diameter* of giant connected component (GCC) of the `NetScience` network.

1. Fill in the missing code in the following `compute_diameter(graph)` function to find the diameter of a given connected graph:

[3]Wikipedia's lemma for the Erdös-Rényi model: `https://en.wikipedia.org/wiki/ErdosRenyi_model`

```
def compute_diameter(graph):
# It is assumed that the graph is connected
diameter = 0
...
...
return diameter
```

You can use the `single_source_shortest_path_length(G,source,cutoff=None)` method in order to compute the shortest path lengths from a *source* node to all the other nodes.