Exercise set #3 (31 pts)

- The deadline for handing in your solutions is Nov 28th 2016 23:55.
- Return your solutions (one `.pdf` file and one `.zip` file containing Python code) in MyCourses (Assignments tab). Additionally, submit your pdf file also to the Turnitin plagiarism checker in MyCourses.
- Check also the course practicalities page in MyCourses for more details on writing your report.

All data and code necessary for this exercise set is provided in the accompanying `.zip` file.

1. Degree correlations and assortativity (7 pts)

In this problem, we consider degree correlations and assortativity of two real-world networks: the Zachary karate club network (`karate_club_network_edge_file.edg`) [1] and a snowball-sampled subgraph of a Facebook friendships network (`facebook-wosn-links_subgraph.edg`) [2, 3]. To get started, you can use the accompanying Python template (`degree_corrs_assort.py`).

For both networks, perform the following analyses:

a) (2 pts) **Create a scatter plot** of the degrees of nodes incident to each edge (to refresh your memory: a node is said to be incident to the edges connected to it - for each edge, there are 2 incident nodes). The scatter plot should have degree values present in the network on both axes and the coordinates of each point in the scatter correspond to (start node degree, end node degree) of an edge in the network. Note that, as our network is undirected, each edge should be considered twice and the scatter thus should be symmetrical.

b) (2 pts) **Produce a heat map**[1] of the degrees of incident nodes of each edge. This heat map should be a 2D histogram, where the x and y coordinates of each element correspond to a possible combination of incident degree values in the network. Thus, the value of each (2D) bin in the 2D-histogram tells the number of edges with the combination of degrees of the incident nodes falling to the 2D-bin. As the scatter above, also the heat map should be symmetrical. Finally, add a colorbar next to the map that shows which color corresponds to which value of the histogram.

c) (1 pts) Assortativity coefficient is defined as the Pearson correlation coefficient between the degrees of incident nodes. **Calculate the assortativity coefficient** of the network using `scipy.stats.pearsonr` and compare your result with the output of NetworkX function `degree_assortativity_coefficient`. As mentioned in the lecture, social networks typically are assortative. **Does this hold for these two social networks? What could explain this result?**

d) (2 pts) For each node, **compute the average nearest neighbour degree** k_{nn} and **make a scatter plot** of k_{nn} as a function of k. In the same plot, **plot also the curve** of $\langle k_{nn} \rangle(k)$ as a function of k, *i.e.* the averaged k_{nn} for each k. **Comment the result** from the viewpoint of assortativity.

[1] `http://en.wikipedia.org/wiki/Heat_map`

2. Centrality measures for undirected networks (7 pts)

In this exercise, we get familiar with some common centrality measures by applying them to undirected networks (although these measures can all be generalized also to directed networks). Below, we list and define the measures used in this exercise:

1. degree $k(i)$:
   Number of neighbors of node i
2. betweenness centrality $bc(i)$:
   Number of shortest paths between other nodes of the network that pass through node i. However, if there exist several shortest paths between a given pair of nodes, then the contribution of that node pair to the betweenness of i is given by the fraction of those paths that contain i. The betweenness scores are also normalized by $(N-1)(N-2)$, i.e. the number of all node-pairs of the network, excluding pairs that contain i (because paths starting or ending in node i do not contribute to the betweenness of i), which is the maximum possible score. Formally, if σ_{st} is the number of shortest paths from s to t and σ_{sit} the number of such paths that contain i, then
   $$bc(i) = \frac{1}{(N-1)(N-2)} \sum_{s \neq i} \sum_{t \neq i} \frac{\sigma_{sit}}{\sigma_{st}}.$$
3. closeness centrality $C(i)$:
   Inverse of the average shortest path distance to all other nodes than i:
   $$C(i) = \frac{N-1}{\sum_{v \neq i} d(i,v)}.$$
4. k-shell $k_s(i)$:
   Node i belongs to the k-shell, if it belongs to the k-core of the network but does not belong to the $k+1$-core. The k-core is the maximal subnetwork (i.e. the largest possible subset of the network's nodes, and the links between them) where all nodes have at least degree k. In other words, the 1-core is formed by removing nodes of degree 0 (isolated nodes) from the network, the 2-core is formed by removing nodes of degree 1 and iteratively removing the nodes that become degree 1 or 0 because of the removal, and so on. The 1-shell is then the set of nodes that was removed from the 1-core to obtain the 2-core.
5. eigenvector centrality $e(i)$:
   Eigenvector centrality is a generalization of degree that takes into account the degrees of the node's neighbors, and recursively the degrees of the neighbors of neighbors, and so on. It is defined as the eigenvector of the adjacency matrix that corresponds to the largest eigenvalue.

a) (2 pts) Your first task is to **compute/reason without a computer the first 4 centrality measures** of the above list for the network shown in Fig. 1 (i.e. using pen-and-paper; note that you do not need to compute the eigenvector centrality, as one then would need to compute the eigenvalues of a 4×4 matrix which can be a bit painful). In your computations, use the definitions given above and show also intermediate steps where necessary.

b) (2 pts) **Use NetworkX to compute all five centrality measures** for the networks studied in exercise 1.3 (`small_cayley_tree.edg, larger_lattice.edg, small_ring.edg`;

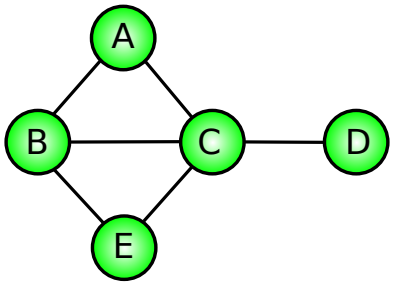

Figure 1: A small undirected network.

see Fig 2) as well as for the Karate club network (`karate_club_network_edge_file.edg`). Then, **visualize betweenness, closeness, k-shell, and eigenvector centrality as a function of degree in a scatter plot** for each of the networks. For easier visual comparison of the measures, you should normalize the k-shell values by dividing them by the maximal k-shell value. For this and the following tasks, you can use the Python template `basic_centrality_measures.py`.

Hint: For some of the networks, the power iteration algorithm used by NetworkX to calculate eigenvector centrality may not converge. In this case, increase the value of the tolerance (`tol`) parameter of `eigenvector_centrality()` until the iteration converges.

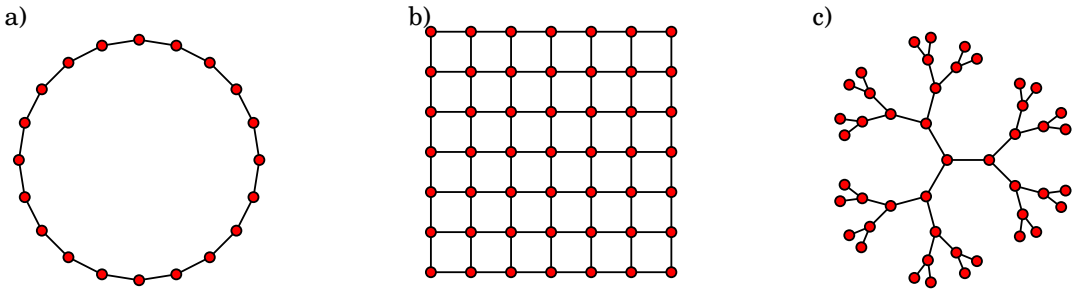

Figure 2: The model networks of exercise 1.3. **a)** A simple ring lattice with $N = 20$. **b)** A simple 2D-lattice with $k = 4$, $L = 7$, ($N = 49$). Note that L used here differs from exercise 1.3. **c)** A Cayley tree with $k = 3$, and $l = 4$

c) (1 pts) To highlight the differences between the centrality measures, **plot five visualizations** of the Karate club network, **each time using one of the centrality measures to define the colors** of the network nodes. To make the visualization easier, coordinates of the nodes are provided in `.pkl` files (`small_cayley_tree_coords.pkl, larger_lattice_coords.pkl, small_ring_coords.pkl, karate_club_coords.pkl`).

d) (2 pts) Based on the results of a) and b), **how do these centralities differ** from each other? Would you say that some of them do a better or worse job than others in identifying central nodes? To answer the questions, you can for example pick some representative nodes and try to explain why different centrality measures rank these node differently regarding its centrality. In your answer, briefly **cover all the networks** visualized in c).

3. PageRank (directed network) (17 pts)

PageRank, a generalization of eigenvector centrality for directed networks, is used by *e.g.* Google to determine the centrality of web pages. If we consider a random walker that with probability d moves to one of the neighbors of the current node and with probability $1 - d$ teleports to a random node, PageRank of each node equals the fraction of time the random walker spent in that node. In this exercise, we investigate the behavior of PageRank in both a simple directed model network (see fig. 3) and an extract from the Wikipedia hyperlink network. To get started, you can use the provided Python template `pagerank.py`.

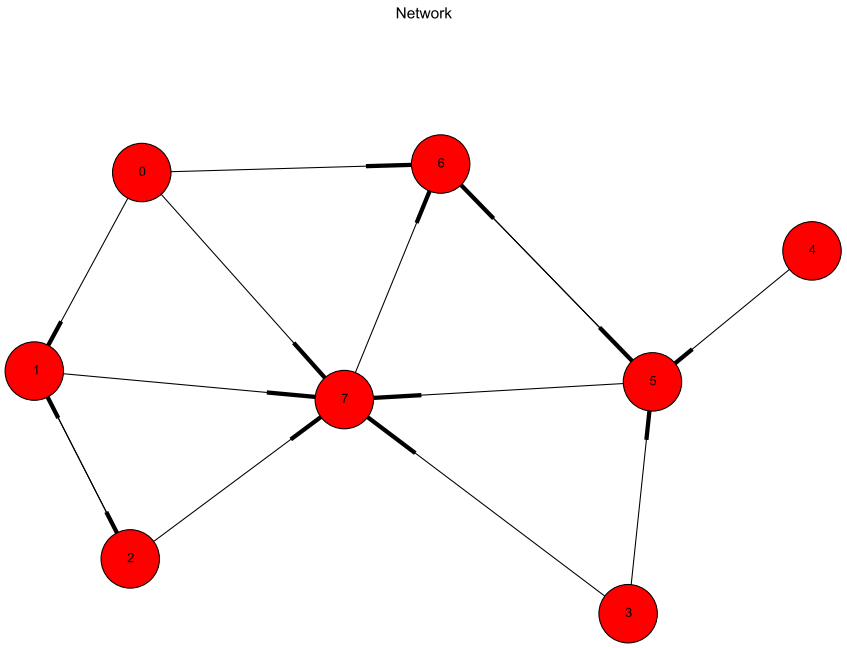

Figure 3: A simple directed network.

a) (1 pt) **Load the network** given in file `pagerank_network.edg` and, as a sanity check, **visualize it** with `nx.draw`.

Hint: To load the directed network, use parameter `create_using=nx.DiGraph()` when reading the edge list. NetworkX visualization of directed graphs is somewhat ugly but sufficient for the present purposes. In fact, the spring layout algorithm in Networkx, which is its default algorithm for computing node positions, works only well with undirected graphs, so for computing the layout, it's better to feed the algorithm the undirected version of the network. Also, the algorithm can give different results on different runs, so it may be useful to plot the network a few times until the result looks good.

b) (4 pts) **Write a function that computes the PageRank** on a network by simulating a random walker. In more detail,

1. Initialize the PageRank of all nodes to 0.
2. Pick the current node (the starting point of the random walker) at random.

3. Increase the PageRank of the current node with 1.
4. Select the node, to which the random walker will move next:
   * Draw a random number $p \in [0, 1]$.
   * If $p < d$, the next node is one of the successors of the current one (i.e. nodes linked *to* by the current node). Pick it randomly.
   * Else, the random walker will teleport. Pick the next node randomly from all the network nodes.
5. Repeat 3-4 N_{steps} times.
6. Normalize the PageRank values by N_{steps}.

Use your function to compute PageRank in the example network. **Visualize the result** on the network: update your visualization from a) by using the PageRank values as node color values. Compare your results with `nx.pagerank` by plotting both results as a function of node index.

Hints: The damping factor is normally set to $d = 0.85$. $N_{steps} = 10000$ is a reasonable choice.

c) (4 pts) The above algorithm is a naive way of computing PageRank. The actual algorithm behind the success of Google, introduced by its founders, Larry Page and Sergey Brin, is based on power iteration [4]. The power iteration can be shown to find the leading eigenvector for the "Google matrix" (or other matrices) very fast under certain conditions. An intuitive way of thinking about the power iteration algorithm is to think that at time $t - 1$ you have a vector $x(t - 1)$ where each element gives the probability of finding the walker. You use the rules of the random walk/teleportation process to find out what are the probabilities of finding the random walkers at each node at time t. That is you increase the time t and calculate $x(t)$ based on $x(t-1)$ until the vector x doesn't change any more. **Write a function that computes the PageRank** by using power iteration. In more detail,

1. Initialize the PageRank of all nodes to $\frac{1}{n}$, where n is the number of nodes in the network. That is, at the iteration $t = 0$ your PageRank vector contains the same value for each node, and it is equally likely to find the walker in each node. (Any other initialization strategy is possible as long as the sum of all elements is one, and the closer the initial vector is to the final vector the faster you will find the final PageRank values.)
2. Increase the iteration number t by one and create a new empty PageRank vector $x(t)$.
3. Fill in each element of the new vector PageRank vector $x(t)$ using the old PageRank vector $x(t - 1)$ and the formula: $x_i(t) = (1 - d)\frac{1}{n} + d\sum_{j\in\nu_i} \frac{x_j(t-1)}{k_j^{\text{out}}}$, where ν_i is the set of nodes that have a directed link ending at i. That is, for each node i you need to calculate their entry in the new PageRank vector $x(t)$ as a sum of two parts:
   * probability that the walker will teleport into the node $(1 - d)\frac{1}{n}$ and
   * probability that the walker will move from a neighbor j to node i. Iterate over each in-neighbor j of the node i (i.e., there is a link from i to j) and add the neighbors contribution $d\frac{x_j(t-1)}{k_j^{\text{out}}}$ to the entry of the node i in the new PageRank vector $x(t)$.
4. Repeat 2-3 $N_{\text{iterations}}$ times.

Use your function to compute PageRank in the example network and **visualize the result** on the network as in b).

Hints:

- The damping factor is normally set to $d = 0.85$.
- You can monitor the progress of the power iteration by printing out the change in the PageRank vector $\Delta(t) = \sum_i |x_i(t) - x_i(t-1)|$ after each iteration step. The change $\Delta(t)$ should be decreasing function of t. $N_{\text{iterations}} = 10$ should be more than enough in most cases.
- You can list the incoming edges to node i with the function `net.in_edges(i)`, where `net` is the network object.
- The sum of all elements in the PageRank vector should always equal to one. There might be slight deviations from this due to numerical errors, but much larger or smaller values is an indication that something is wrong with the code.

d) (2 pts) The Google search engine indexes billions of websites and the algorithm for calculating the PageRank needs to be extremely fast. In the original paper about PageRank [4], by Google founders Larry Page and Sergey Brin, they claim that their "iterative algorithm" is able to calculate the PageRank for 26 million webpages in a few hours using a normal desktop computer (in 1998). **Come up with a rough estimate** of how long it would take for your power iteration algorithm (part c) and naive random walker algorithm (part b) to do the same. You can assume that the average degree of the 26 million node network is small and that the power iteration converges in the same number of steps as it does for your smaller networks. For the random walk you can assume that you need to run enough steps that the walker visits each node on average 1000 times. You can also omit any considerations of fitting the large network in memory or the time it takes to read it from the disk etc. With these assumption you can simply calculate the time it takes to run the algorithm in a reasonable size network and multiply the result by the factor that the 26 million node network is bigger than your reasonable sized network.

Hints:

- There are several ways of timing your code. In Linux you can run your script using the command `time python myscript.py` instead of `python myscript.py` and read out the "user" value. Even better is to use IPython and run a function calculating everything with the command `%timeit calculate_everything()`, or a script with command `%timeit %run myscript.py`. You can also use the Python `timeit` module.
- The small example network is probably going to be to small to test out the speed of your function especially if you measure the time it takes to run a Python script. (In this case your function might take milliseconds to run but running the whole script might still take a second or so because of starting Python and loading various modules.) You should aim for a network for which it takes several seconds to run the PageRank function. You might find it useful to use network model in networkx to run your code. For example,
  `net=nx.directed_configuration_model(10**4*[5],10**4*[5],create_using=nx.DiGraph())`
  will produce network with 10000 nodes where each node has in and out degrees of 5 using the configuration model.
- Don't feel bad if you cannot beat Larry and Sergey in speed when using Networkx and Python. These tools are not meant for speed of computation and even modern computers might not be enough to help. Also, your competition invented Google.

e) (2 pts) **Describe** how the network's structure relates to PageRank. What is the connection between degree k or in-degree k_{in} and PageRank? How does PageRank change if the node belongs to a strongly connected component? How could this information be used in improving the power iteration algorithm given in part c)?

f) (2 pts) **Investigate the role of the damping factor** d. Repeat the PageRank calculation with *e.g.* 5 different values of $d \in [0, 1]$ and plot the PageRank as a function of node index (plots of all values of d in the same figure). How does the change of d affect the rank of the nodes and the absolute PageRank values?

g) (2 pts) Now, let's see how PageRank works in a real network. File `wikipedia_network.edg` contains the strongly connected component of the Wikipedia hyperlink network around the page Network Science[2]. Load the network and **list the five most central nodes and their centrality score in terms of PageRank, in-degree, and out-degree**. Here you should use `nx.pagerank`, as the naive algorithm implemented in (a) converges very slowly for a network of this size. Comment the differences and similarities between the three lists of most central pages.

Feedback (1 pt)

To earn one bonus point, give feedback on this exercise set and the corresponding lecture latest two day after the report's submission deadline.
Link to the feedback form: `https://goo.gl/forms/9ue4OVozyBwMVkuu2`.

References

[1] W. W. Zachary, An information flow model for conflict and fission in small groups, Journal of Anthropological Research 33, 452-473 (1977)

[2] Facebook friendships network dataset - KONECT, August 2014. `http://konect.uni-koblenz.de/networks/facebook-wosn-links`

[3] B. Viswanath, A. Mislove, M. Cha, K. P. Gummadi, On the evolution of user interaction in Facebook. Workshop of Online Social Networks, 37-42 (2009)

[4] S. Brin and L. Page, Reprint of: The anatomy of a large-scale hypertextual web search engine. Computer networks, 56.18 (2012), pp.3825-3833.

[2] extracted on May 2nd 2012