Homework #1

This homework is complete and will not be changed. The homework does not require a lot of writing, but may require a lot of thinking. It does not require a lot of processing power, but may require efficient programming. It accounts for 12.5% of the course grade. All questions and comments regarding the homework should be directed to Piazza.

Submission details

This homework is due on March 23rd at 2:00pm, while late days expire on March 27th at 1:00pm. The homework must be submitted as a hard-copy in the submission box in front of R 2.49 and also as an electronic version to eUcilnica. It can be prepared in either English or Slovene and either written by hand or typed on a computer. The hard-copy should include (1) this cover sheet with filled out time of the submission and signed honor code, (2) short answers to the questions, which can also demand proofs, tables, plots, diagrams and other, and (3) a printout of all the code required to complete the exercises. The electronic submission should include only (1) answers to the questions in a single file and (2) all the code in a format of the specific programming language. Note that hard-copies will be graded, while electronic submissions will be used for plagiarism detection. The homework is considered submitted only when both versions have been submitted. Failing to include this honor code in the submission will result in 10% deduction. Failing to submit all the developed code to eUcilnica will result in 50% deduction.

Honor code

The students are strongly encouraged to discuss the homework with other classmates and form study groups. Yet, each student must then solve the homework by herself or himself without the help of others and should be able to redo the homework at a later time. In other words, the students are encouraged to collaborate, but should not copy from one another. Referring to any solutions obtained from classmates, course books, previous years, found online or other, is considered an honor code violation. Also, stating any part of the solutions in class or on Piazza is considered an honor code violation. Finally, failing to name the correct study group members, or filling out the wrong date or time of the submission, is also considered an honor code violation. Honor code violation will not be tolerated. Any student violating the honor code will be reported to faculty disciplinary committee and vice dean for education.

Name & SID: JERNEJ VIVOD

Study group: /

Date & time: 22. 3. 2020 , 13:50

I acknowledge and accept the honor code.

Signature:

Homework #1

Jernej Vivod
Introduction to Network Analysis

March 15, 2020

Question 1.1

What can you say about the degree sequences $\{k\}$ and degree distributions p_k of the shown graphs?

The second graph follows a power law distribution as there exist some nodes with an unusual number of connections (hubs). The first network has nodes that are generally well-connected. There is an absence of hubs. Plotting the degree distribution would produce a characteristic bell-shaped curve (binomial distribution). Unlike the second network, this network contains some nodes with degree 1.

Question 1.2

Assuming a simple undirected network with n nodes, m links and c connected components. Show that the given two inequalities hold. Using these inequalities give a criterion for m that ensures a connected network. Is the criterion practically useful? Why?

Let us start the proof by proving that if $n - c \leq m$ holds for a graph with n nodes, then it also holds for a graph with $n + 1$ nodes. Note that the upper bound for the number of links in a fully connected network with n nodes is $m_{max} = \frac{n(n-1)}{2}$. Using this, we can show:

$$
\begin{aligned}
(n+1) - c &\leq \frac{(n+1)n}{2} \\
2n + 2 - 2c &\leq n^2 + n \\
n + 2 - 2c &\leq n^2 \\
-2c + 2 &\leq n^2 - n
\end{aligned}
$$

Since the right side of the inequality is strictly positive for $n > 0$ and the left side of the inequality is non-positive for $c > 0$, the equality hold for all graphs with more than one node and more than one connected component.

We now only need to show, that the equality holds for a base case. Assume a graph with a single node (and one connected component). We can see that:

$$
\begin{aligned}
n - c &\leq m \\
1 - 1 &\leq 0 \\
0 &\leq 0
\end{aligned}
$$

By the induction step, we now know that the inequality holds for any graph with $n > 0$ nodes.

The reasoning for the second inequality can be given as follows. Suppose we have a network with n nodes and c connected components. The upper bound for the number of links in such a network is achieved by having the network split into one giant fully connected component and $c - 1$ other "trivial" connected components consisting of single nodes with no links. We know that the number of links in a fully connected network is given by the number of unique pairs of nodes in the network $\binom{n}{2}$. By having the aforementioned network configuration, we "lose" $c-1$ nodes to the trivial connected components. By this reasoning, $m \leq \binom{n-c+1}{2}$ must hold.

To ensure a connected network, the inequality $m \geq n - 1$ must hold. To link n nodes into a connected network, we need to place exactly one link between a node with no links and another node that is part of a connected component. To create such a tree graph, we need to place $n - 1$ links.

This criterion is useful as it gives the minimum number of connections needed to form a path from each node to every other node which is useful in many real-world applications (certain infrastructure projects, etc.).

Question 1.3

What would the mentioned algorithm for finding connected components of undirected networks find in a directed network if one could follow the links in any direction? What would it find in a directed network if one could follow the links only in the proper direction? What would it find in a directed network if one could follow the links only in the opposite direction? Design an efficient algorithm for finding strongly connected components in directed networks.

If one could follow the links in any direction, the algorithm would find the weakly connected components of the directed graph. If one could follow the links only in the proper direction, the algorithm would find all the nodes reachable from the starting node by following the directed paths in their proper directions as well as the nodes reachable from the nodes not in the component of the initial node but without the nodes in the previously found components. Similarly, the algorithm would find the same types of components in the transpose graph (with the edge directions reversed). In the original graph, that would be equivalent to finding the nodes from which the to starting node can be reached by following the directed edges.

The algorithm for finding the strongly connected components can be formulated by observing the mentioned properties as well as the behaviour of depth-first search. We can

design an algorithm that will find all nodes reachable from root node as well as to find the subset of the found nodes that can reach the root node by using depth-first search on the transpose graph.

Running the algorithm we see that the Enron e-mail communication network has 78058 strongly connected components. The size of the largest one is 9164. There are relatively many (especially single node) strongly connected components in the network (32.15% the number of nodes) This is due to a large number of people that only received e-mail communication but did not reply. The people who did reply overwhelmingly became part of the largest strongly connected component. The results are surprising in terms of the large number of single-node strongly connected components (one-way communication).

The pseudocode for the algorithm used to find the strongly connected components is given below:

```
Data: Directed graph
Result: List of strongly connected components
Initialize stack S ;
Put root node on stack S ;
while S not empty do
    Pop node from stack S. ;
    if popped node not yet visited then
        if node has unvisited neighbors then
            push popped node back on stack S. ;
            Add unvisited neighbors to stack S. ;
        else
            Add node on DFS finish stack. ;
            Pop node from stack. ;
        end
    else
        Add node on DFS finish stack. ;
        Pop node from stack. ;
    end
end
Get transpose graph G' ;
Initialize empty list SCC ;
while DFS finish stack not empty do
    Pop node from DFS finish stack ;
    Perform DFS starting from popped node to get next SCC and add it to SCC ;
end
```

Algorithm 1: Algorithm used to find the strongly connected components.

The implementation of the algorithm used to find the strongly connected is given below:

```python
import networkx as nx


def strongly_connected_components(graph):
    """
    Find strongly connected components in directed graph.
    Author: Jernej Vivod (vivod.jernej@gmail.com)

    Args:
        graph (networkx.classes.digraph.DiGraph): directed graph.

    Returns:
        (list): List of lists containing nodes in connected components.
    """

    def get_finish_stack(graph):
        """
        Get stack of nodes based on their DFS finish times.

        Author:
            Jernej Vivod (vivod.jernej@gmail.com)

        Args:
            graph (networkx.classes.digraph.DiGraph): directed graph.

        Returns:
            finish_stack (list): stack of nodes ordered by their DFS finish times.
        """

        # Initialize starting finish time enumeration value.
        step_nxt = 1

        # Initialize dictionary for storing start and finish times.
        finish_dict = {el:[0, 0] for el in graph.nodes()}

        # Initialize stack for storing the nodes in finish time order.
        finish_stack = []

        # Enumerate while there are nodes in the graph.
        while graph.number_of_nodes() > 0:
           step_nxt = dfs_enumerate(graph, finish_dict, finish_stack, step_nxt)

        # Return stack of nodes based on their finish time.
        return finish_stack


    def dfs_enumerate(graph, finish_dict, finish_stack, step):
        """
        Enumerate component based on DFS finish times (auxiliary function)
        Author: Jernej Vivod (vivod.jernej@gmail.com)

        Args:
            graph (networkx.classes.digraph.DiGraph): directed graph.
            finish_dict (dict): dictionary mapping nodes to their DFS start and finish times.
            finish_stack (list): stack for keeping nodes based on their DFS finish times.
            step (int): DFS enumeration start value.

        Returns:
            (int): next value to use in enumeration of nodes based on DFS finish times.
        """

        # Initialize stack for performing DFS.
        stack = []

        # Add starting node to stack.
        node_start = list(graph.nodes())[0]
```

```python
        stack.append(node_start)

        # While stack not empty, perform DFS and enumerate nodes
        # based on finish time.
        while len(stack) > 0:

            # Get node on top of stack.
            node_current = stack[-1]

            # If node in graph ...
            if graph.has_node(node_current):

                # Set start time and increment enumeratio value.
                finish_dict[node_current][0] = step
                step += 1

                # Get neighbors and remove node from graph.
                neighbors = graph.neighbors(node_current)
                graph.remove_node(node_current)
                has_unvisited_neighbors = False

                # Go over neighbors and add unvisited to stack.
                for neighbor in neighbors:
                    if graph.has_node(neighbor):
                        has_unvisited_neighbors = True
                        stack.append(neighbor)

                # If node has no unvisited neighbors ...
                if not has_unvisited_neighbors:

                    # If not yet finished, set finish time and pop from stack.
                    if finish_dict[node_current][1] == 0:
                        finish_dict[node_current][1] = step
                        finish_stack.append(node_current)
                        step += 1
                    stack.pop()
            else:
                # If not yet finished, set finish time and pop from stack.
                if finish_dict[node_current][1] == 0:
                    finish_dict[node_current][1] = step
                    finish_stack.append(node_current)
                    step += 1
                stack.pop()

        # Return last enumeration value.
        return step


    def get_strongly_connected_components(graph_trans, finish_stack):
        """
        Get strongly connected components by performing DFS on transposed graph
        and utilizing the computed stack of nodes ordered by their DFS finish
        times on the original graph.
        Author: Jernej Vivod (vivod.jernej@gmail.com)

        Args:
            graph_trans (networkx.classes.digraph.DiGraph): The transpose of the original
                graph.
            finish_stack (list): stack of nodes ordered by their DFS finish times.

        Returns:
            (list): list of lists of nodes constituting the strongly connected components.
        """

        # Initialize list for sotring connected components.
        components = []

        # Initialize stack for performing DFS.
```

```python
        stack = []

        # While stack of nodes ordered by finish time not empty ...
        while len(finish_stack) > 0:

            # Add starting node for DFS to stack.
            node_start = finish_stack.pop()
            stack.append(node_start)

            # Initialize list for storing the next
            # strongly connected component.
            component = []

            # While stack not empty, perform DFS.
            while len(stack) > 0:
                current_node = stack.pop()

                # If node in graph ...
                if graph_trans.has_node(current_node):

                    # Add node to component.
                    component.append(current_node)

                    # Get neighbors and remove node from graph.
                    neighbors = graph_trans.neighbors(current_node)
                    graph_trans.remove_node(current_node)

                    # Go over neighbors and add unvisited to stack.
                    for neighbor in neighbors:
                        if graph_trans.has_node(neighbor):
                            stack.append(neighbor)

            # If found non-empty component, add to list of
            # strongly connected components.
            if len(component) > 0:
                components.append(component)

        # Return list of strongly connected components.
        return components


    # Get stack of nodes ordered by their DFS finish times.
    finish_stack = get_finish_stack(graph.copy())

    # Get transpose of graph.
    graph_trans = graph.reverse()

    # Compute strongly connected components.
    return get_strongly_connected_components(graph_trans, finish_stack)
```

Question 1.4

Think of another example of a network for which $\langle C \rangle \to$ const and $C \to 0$ when $n \to \infty$.

Another example of a network with such properties would be the "petal" network shown on figure 1. To prove that the network has such properties, let us denote with k the number of the petals in the network. In a network with 2 petals, there are 4 connected triples that extend between the petals as well as 3 connected triples on each on the petals themselves. Now consider a network with k petals. To find the number of connected triplets that extend

between pairs of petals, we need to count such triplets in $\frac{n(n-1)}{2}$ unique pairs of petals. We also need to count the $3 \cdot k$ triplets found in each individual petal. As the number of petals increases, the growth of the term capturing the increase in the number of connected triplets is clearly quadratic as more and more petals are added, where as the number of triangles in the graph increases by 1 for each added petal (linear growth). The clustering coefficient of each node in the petals is equal to 1. As more and more petals are added, the average clustering converges to the constant value of 1.

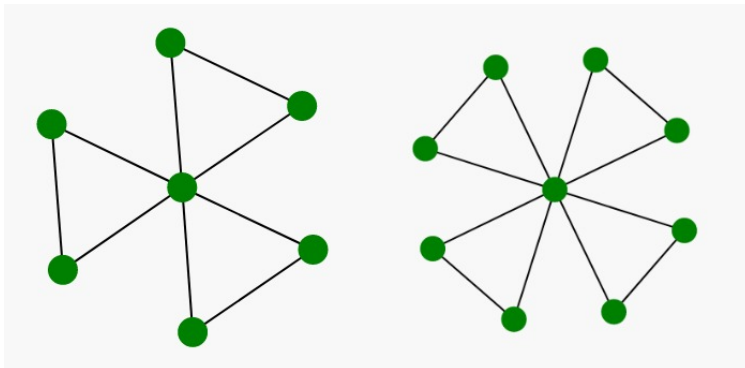

Figure 1: Examples of a petal network with 3 and 4 petals.

Question 1.5

Design an efficient algorithm for computing E_{90} of a connected undirected network. Implement the algorithm and compute E_{90} for citation networks of physics papers. Are the results surprising? Why? Compute also the number of nodes and the average degree $\langle k \rangle$ of all three networks and discuss the results.

We can formulate the algorithm by using two similar but distinct definitions of the effective diameter. We can either compute the lengths of paths between every unique pair of connected vertices or take the average minimal distance required to reach 90% of nodes from each node. Both methods were implemented for this task. Table 1 summarizes the results.

	all-pairs E_{90}	unique-pairs E_{90}	number of nodes	average degree $\langle k \rangle$
Citations for 2010-2011	14.3792	15.0	18985	4.3694
Citations for 2010-2012	11.9346	13.0	38356	5.7302
Citations for 2010-2013	10.4929	12.0	56473	7.0955

Table 1: 90-percentile effective diameter, number of nodes and average degree for citations networks of physics papers for increasing time intervals.

We can see that analyzing citations over a larger interval produces a lower 90-percentile effective diameter and a higher average node degree. The decrease in the 90-percentile effective diameter can seem surprising at first but can be interpreted by observing that taking a larger time interval increases the chances of early papers being cited by papers

towards the end of the time interval which in turn increases the average node degree. These citations by later papers often then form links that reduce distances between earlier papers as well as later ones and thus decrease the 90-percentile effective diameter of the network.

The pseudocode below presents the algorithm used to solve this task using distances between unique pairs of nodes. To compute the distances using all pairs of nodes, a slightly modified algorithm which computes the 90th percentile for distances between sampled node and all other nodes in the graph was used. The latter algorithm obtains the results by averaging the 90th percentile distances from each node to every other node in the graph.

Data: Graph G
Result: 90-percentile effective diameter of graph G
idx = 1 ;
for *each node in sorted nodes (by ID) in G* **do**
  dists := get next distances to sorted G. ;
  `/* select relevant distances (not yet computed for that pair of nodes). */`
  dists := dists[idx:end] ;
  idx := idx + 1 ;
end
return value at 90th percentile in dists. ;

Algorithm 2: Algorithm used to find the 90-percentile effective diameter.

The code written to find the solutions to this question is given below:

```python
import networkx as nx
import numpy as np

def effective_diameter(graph, mode, percentile):
    """
    Compute nth-percentile effective diameter of graph.
    Author: Jernej Vivod (vivod.jernej@gmail.com)

    Args:
        graph (networkx.classes.digraph.DiGraph): Graph for which to compute the nth-percentile effective diameter.
        mode (str): Method of computing the results. If equal to 'unique_pairs', compute
        result as nth-percentile of distances between unique node pairs.
        percentile (int): Percentile used in the computations.

    Returns:
        (float): The nth-percentile effective diameter of specified graph.
    """

    def pairwise_distances(graph):
        """
        Compute vector of distances between unique pairs of nodes in graph.

        Author:
            Jernej Vivod (vivod.jernej@gmail.com)

        Args:
            graph (networkx.classes.digraph.DiGraph): Graph for which to compute the nth-percentile effective diameter.

        Returns:
```

```python
            (numpy.ndarray): Vector of distances between unique pairs of nodes in specified graph.

        """

        # Allocate vector for storing unique pairwise distances.
        num_nodes = graph.number_of_nodes()
        dists = np.empty(int((num_nodes*(num_nodes-1))/2), dtype=int)

        # Set start position for selecting relevant pairwise distances.
        # Set index for relevant pairwise distances vector.
        start_pos = 1
        dists_idx = 0

        # Go over nodes and compute pairwise distances.
        for node in sorted(map(int, graph.nodes())):

            # Get distances from next node to all other nodes.
            dists_nxt = get_distances(graph, str(node))

            # Get relevant pairwise distances
            dists[dists_idx:dists_idx+len(dists_nxt[start_pos:])] = dists_nxt[start_pos:]
            start_pos += 1
            dists_idx += len(dists_nxt[start_pos:]) + 1
        return dists


    def distances_percentile(graph, percentile):
        """
        Compute vector of nth-percentile distances to every node from each node.

        Author:
            Jernej Vivod (vivod.jernej@gmail.com)

        Args:
            graph (networkx.classes.digraph.DiGraph): Graph for which to compute the nth-percentile effective diameter.
            percentile (int): Percentile used in the computations.

        Returns:
           (numpy.ndarray): Vector of n-th percentile distances to every node from each node.

        """

        # Allocate vector for storing 90th percentile distances.
        num_nodes = graph.number_of_nodes()
        dists_perc = np.empty(num_nodes, dtype=float)

        # Go over nodes and compute distances at percentiles.
        for (idx, node) in enumerate(sorted(map(int, graph.nodes()))):

            # Get distances from next node to all other nodes.
            dists_nxt = get_distances(graph, str(node))

            # Compute distance representing the percentile.
            dists_perc[idx] = np.percentile(dists_nxt, percentile)

        # Return vector of distances at percentiles for each node.
        return dists_perc


    def get_distances(graph, node):
        """
        Compute distances from node to every other node in graph.

        Args:
            graph (networkx.classes.digraph.DiGraph): Graph for which to compute the nth-
```

```python
                percentile effective diameter.
            node (str): Node for which to compute distances to every other node.

        Returns:
            (numpy.ndarray): array of distances from specified node to every other node in the graph.

        Author:
            Jernej Vivod (vivod.jernej@gmail.com)
        """

        # Initialize array for storing distances.
        dists = np.full(graph.number_of_nodes(), -1, dtype=int)

        # Set distance of current node to itself to zero.
        dists[int(node)-1] = 0

        # Initialize queue and add starting node.
        queue = []
        queue.append(node)

        # While queue not empty, perform BFS.
        while len(queue) > 0:
            node_current = queue.pop(0)
            for neighbor in graph.neighbors(node_current):

                # Compute distances to neighbors.
                if dists[int(neighbor)-1] == -1:
                    dists[int(neighbor)-1] = dists[int(node_current)-1] + 1
                    queue.append(neighbor)

        # Return array of distances of node to all the other nodes.
        return dists


    # Compute nth-percentile effective diameter.
    if mode == 'unique_pairs':
        dists_vec = pairwise_distances(graph)
        return np.percentile(dists_vec, percentile)
    elif mode == 'all_pairs':
        dists_perc = distances_percentile(graph, percentile)
        return np.mean(dists_perc)
```

Question 2.1

Design an algorithm that does not select nodes uniformly at random, but proportional to their degrees. Thus, node i is selected with probability $\frac{k_i}{2m}$, where k_i is its degree ad m is the number of links. The algorithm should run in constant time $O(n)$, whereas you can assume any standard network representation.

Since the probabilities are expressed as fractions, we do not need to resort to algorithms such as the alias method for sampling from weighted distributions. Instead, we can represent the network as a list of nodes (their IDs) where each node is repeated k_i times. The probability of a node being selected by random uniform sapling from such a list will be equal to the desired probability. The sampling can then simply be done arbitrary many times by generating a random integer in the interval $[1, \text{list\_length}]$ which can be done in constant time. The pseudocode for the algorithm as well as the code used for the implementation are given below.

Data: Graph G
Result: Node selected with probability proportional to its degree.
L := list of node ID's where the i-th node ID is repeated k_i times. ;
`/* Selection can now be done in O(1) (constant) time. */`
for *number of nodes to select* **do**
  idx := random integer from $[0, \text{length}(L)]$;
  yield L[idx] ;
end

Algorithm 3: Algorithm used to select N nodes with probability proportional to their degree in constant time.

```python
import random

def select_preferential(node_to_degree, n):
    """
    Select n nodes in specified graph with probability proportional
    to their degrees (no replacement).
    Author: Jernej Vivod

    Args:
        node_to_degree (dict): dictionary mapping node IDs to their degrees.
        n (int): number of nodes to select.

    Returns:
        (list): IDs of selected nodes.
    """

    # Create list of node IDs where number of occurrences is equal to the node's degree.
    sel_list = [key for key in node_to_degree.keys() for _ in range(node_to_degree[key])]

    # Allocate list for storing results.
    res = ['']*n

    # Sample n nodes without replacement.
    for idx_sel in range(n):
        sel = sel_list[random.randint(0, len(sel_list)-1)]
        res[idx_sel] = sel
        sel_list = list(filter(lambda x: x != sel, sel_list))

    # Return sample as list of IDs.
    return res
```

Question 2.2

Consider a random graph model in which links are placed independently between each pair of nodes i and j with probability p_{ij} proportional to $v_i v_j$, where v_i is some non-negative number associated with node i. First show that the expected node degree k_i is proportional to v_i. Next, derive an exact expression for p_{ij} in terms of the degree sequence $\{k_i\}$ and discuss the result.

The expected node degree $E(k_i)$ can be written as:

$$E(k_i) = \sum_{j \neq i} p_{ij}$$
$$E(k_i) = \alpha \sum_{j \neq i} v_i v_j$$
$$E(k_i) = \alpha v_i \sum_{j \neq i} v_j$$

The last line clearly shows that the expected degree is proportional to the value of v_i.

The probability of a link between node i and node j can be expressed by dividing the number of ways to choose one half-edge from node i and one half-edge from node j by the number of ways to connect a half-edge to another half-edge. We can write this as:

$$p_{ij} = \frac{k_i k_j}{2m - 1}$$

We know that p_{ij} is proportional to $v_i v_j$. We can therefore write:

$$p_{ij} = \frac{k_i k_j}{2m - 1}$$
$$v_i v_j \propto \frac{k_i k_j}{2m - 1}$$
$$v_i v_j \propto \frac{k_i k_j}{2m - 1}$$
$$v_i v_j \propto k_i k_j$$
$$v_i v_j = \alpha k_i k_j$$

This shows that the product of the values $v_i v_j$ is directly proportional to the product of the degrees of nodes i and j.

Question 2.3

Represent a small part of the Facebook social network as an undirected graph and compute its degree distribution p_k. Construct an Erdős–Rényi random graph with parameters n and m and again compute p_k. Compute also the theoretical degree distribution of the Erdős–Rényi random graph (Poisson distribution). Finally construct a random graph according to the preferential attachment model using the specified procedure. Again compute p_k. Plot the results on a doubly logarithmic (log-log) plot. Compare all four degree distributions p_k and highlight the differences among them.

The results obtained by randomly sampling 30000 nodes from the Facebook social network graph are shown on figure 2.

We can see that both the Facebook social network network as well as the Barabási–Albert preferential attachment model follow a power-law node degree distribution, whereas the

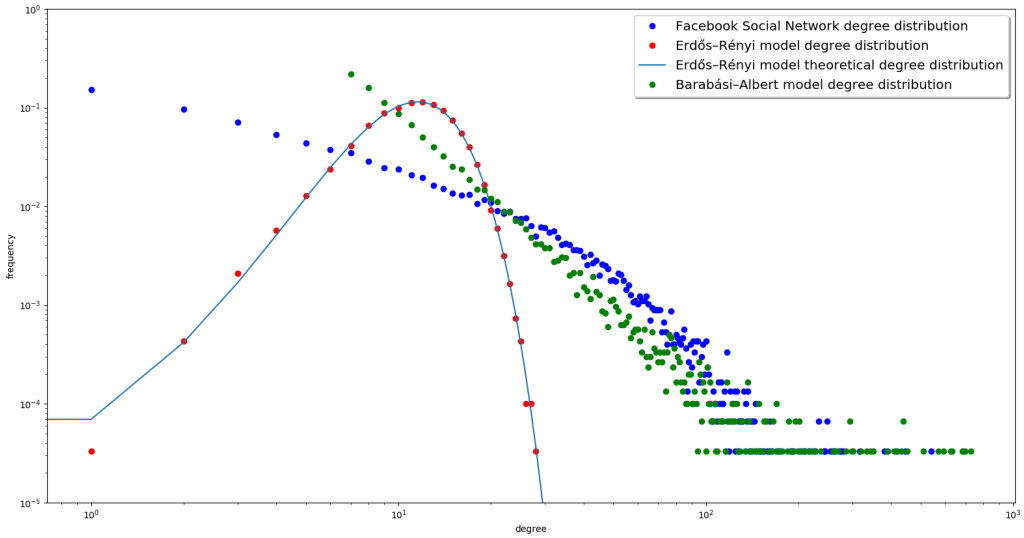

Figure 2: Degree distributions of a sample of the Facebook social network graph and its different models.

Erdős–Rényi model follows the binomial degree distribution which sharply contrasts with the more natural Barabási–Albert model that features hubs as well as a multitude of nodes with relatively low degrees. We can also note that the actual, empirically compute degree distribution of the Erdős–Rényi model closely follows the theoretically predicted distribution.

The code written to find the solutions to this question is given below:

```
import math
import random
import networkx as nx
from collections import Counter
import matplotlib.pyplot as plt

from select_preferential import select_preferential

# Parse Facebook Social Network graph from file.
GRAPH_PATH = '../data/facebook'
graph_fb_full = nx.read_edgelist(GRAPH_PATH, create_using=nx.Graph)

# Create a subgraph of the Facebook Social Network graph by randomly sampling nodes.
SAMPLE_SIZE_FB = 30000
sampled_nodes = random.sample(graph_fb_full.nodes, SAMPLE_SIZE_FB)
graph_fb = graph_fb_full.subgraph(sampled_nodes)

# Compute histogram of node degrees.
hist_fb = Counter([degree for n, degree in graph_fb.degree()])

# Compute mean node degree.
mean_k = sum([degree*hist_fb[degree]/graph_fb.number_of_nodes() for degree in hist_fb.keys()
    ])

# Plot degree distribution for Facebook Social Network graph.
degrees = hist_fb.keys()
plt.loglog(list(degrees), [hist_fb[degree]/graph_fb.number_of_nodes() for degree in degrees
```

```
    ],
        'bo', label="Facebook Social Network degree distribution")

# Create Erdos-Renyi model with same number of nodes and links.
graph_model_er = nx.gnm_random_graph(graph_fb.number_of_nodes(), graph_fb.number_of_edges())

# Compute histogram of node degrees.
hist_model_er = Counter([degree for n, degree in graph_model_er.degree()])

# Plot degree distribution for model
degrees_model_er = hist_model_er.keys()
plt.loglog(list(degrees_model_er), [hist_model_er[degree]/graph_model_er.number_of_nodes() for degree in degrees_model_er],
        'ro', label="Erdos-Renyi model degree distribution")


# Plot theoretical distribution of the Erdos-Renyi model degree distribution.
degrees_model_er_sorted = list(range(71))
hist_model_er_theoretical = [(mean_k**k)*math.exp(-mean_k)/math.factorial(k) for k in degrees_model_er_sorted]
plt.loglog(degrees_model_er_sorted, hist_model_er_theoretical, '-', label="Erdos-Renyi model theoretical degree distribution")


# Construct preferential attachment model (Barabasi-Albert model).

# Start with fully connected graph.
graph_model_ba = nx.complete_graph(math.ceil(mean_k)+1, create_using=nx.Graph)
node_nxt = max(list(graph_model_ba.nodes())) + 1

for idx in range(SAMPLE_SIZE_FB - math.ceil(mean_k) - 1):

    graph_model_ba.add_node(node_nxt)

    # Select ceil(mean_k/2) existing nodes with probability proportional to their degrees and link to them.
    for node in select_preferential(dict(graph_model_ba.degree()), n=math.ceil(mean_k/2)):
        graph_model_ba.add_edge(node_nxt, node)

    node_nxt += 1
    print("done {0}/{1}".format(idx, SAMPLE_SIZE_FB-math.ceil(mean_k)-2))

# Compute histogram of node degrees.
hist_model_ba = Counter([degree for n, degree in graph_model_ba.degree()])

# Plot degree distribution for model.
degrees_model_ba = hist_model_ba.keys()
plt.loglog(list(degrees_model_ba), [hist_model_ba[degree]/graph_model_ba.number_of_nodes() for degree in degrees_model_ba],
        'go', label="Barabasi-Albert model degree distribution")


# Finish plotting.
plt.legend(loc='upper right', shadow=True, fontsize='x-large')
plt.xlabel("degree")
plt.ylabel("frequency")
plt.ylim(1e-5, 1)
plt.show()
```

Question 3

You are given the Slovenian highway network from 2010 with traffic loads at each location. Find out which measure of node position could be utilized to

best predict the traffic loads. You should consider at least node degree k_i, node clustering coefficient $c_i = \frac{2t_i}{k_i(k_i-1)}$ and the node harmonic mean distance $\ell_i^{-1} = \frac{1}{n-1}\sum_j \frac{1}{d_{ij}}$. Compute the Pearson or Spearman correlation coefficient between the values returned by some node measure and the actual traffic loads. Compute the correlation coefficient for each of the three measures. Are the results expected? Why? List also the top ten locations according to the best node measure along with the computed values and the actual traffic loads.

Table 2 shows the values of the Pearson correlation coefficient for each of the three measures. Table 3 lists the top ten locations according to the best node measure along with

Node measure	Pearson correlation coefficient value
node degree measure	0.2757
node clustering measure	undefined (0 clustering for all nodes)
harmonic mean distance measure	0.6178

Table 2: Values of the Pearson correlation coefficient for used node measures.

the computed values and the actual traffic loads.

Node measure	Pearson correlation coefficient value	Traffic load
Kozarje	0.1230	35759.1561
Koseze	0.1222	50232.0277
Zadobrova	0.1200	33779.1600
Malence	0.1195	55297.8647
Brdo	0.1183	35759.1561
Slivnica	0.1155	20031.1456
Vič	0.1137	20686.5315
Celovška	0.1133	28813.9702
Zaloška	0.1132	20686.5315

Table 3: Top ten locations according to the best node measure (harmonic mean distance) and the actual traffic load.

The predictions made by using the harmonic mean distance measure (closeness centrality of a node measure by the reciprocal of farness) are relatively good. We can see that the top rated nodes are located close to important junctions in the ring around Ljubljana which itself serves as a sort of a junction between the extending branches. The results are therefore expected.

The code written to find the solutions to this question is given below:

```
import networkx as nx
import scipy.stats as sps
import re

def load_with_attributes(path):
    """
    Load graph from specified file and add listed node attributes to graph.
```

```python
    Author: Jernej Vivod (vivod.jernej@gmail.com)

    Args:
        path (str): Path to file containing graph data.

    Returns:



    """

    # Parse graph.
    graph = nx.read_edgelist(GRAPH_PATH, create_using=nx.Graph)

    # Initialize dictionaries for parsing data.
    load = dict()
    names = dict()

    # Flag indicating whether next line contains data to be parsed.
    parse_data = False
    with open(path, 'r') as f:
        for line in f:
            if line[0] == "#":
                if not parse_data:
                    if len(line.split(" ")) == 1:
                        parse_data = True
                elif parse_data and len(line.split(" ")) > 1:
                    data_raw_nxt = line.split(" ")
                    names[data_raw_nxt[1]] = re.findall(r'"([^"]*)"', line)[0]
                    load[data_raw_nxt[1]] = float(data_raw_nxt[-1].strip())
            else:
                nx.set_node_attributes(graph, load, 'load')
                nx.set_node_attributes(graph, names, 'name')
                return graph


# Parse Slovenian highways network dataset.
GRAPH_PATH = '../data/highways'
graph = load_with_attributes(GRAPH_PATH)

# Compute node degree, node clustering coefficient and the node harmonic mean distance.
degrees = graph.degree()
clustering_coefficients = nx.clustering(graph)
harmonic_mean_distances = {node : sum(map(lambda x: 1.0/x, filter(lambda x: x != 0,
    nx.single_source_shortest_path_length(graph, node).values())))/(graph.number_of_nodes()
        -1) for node in graph.nodes()}


# Compute correlation of measure values with actual loads.
nodes = graph.nodes()
nodes_to_data = graph.nodes(data=True)
loads_data = [nodes_to_data[node]['load'] for node in nodes]

# Compute pearson correlation coefficient for the node degree measure.
degree_measure_data = [degrees[node] for node in nodes]
degree_measure_pearson = sps.pearsonr(degree_measure_data, loads_data)
print("Pearson correlation coefficient for the node degree measure: {0}, two-sided p-value:
    {1}".format(*degree_measure_pearson))

# Compute pearson correlation coefficient for the node clustering measure.
clustering_measure_data = [clustering_coefficients[node] for node in nodes]
clustering_measure_pearson = sps.pearsonr(clustering_measure_data, loads_data)
print("Pearson correlation coefficient for the node clustering measure: {0}, two-sided p-
    value: {1}".format(*clustering_measure_pearson))

# Compute pearson correlation coefficient for the harmonic mean distance measure.
harmonic_mean_distances_measure_data = [harmonic_mean_distances[node] for node in nodes]
harmonic_mean_distances_measure_pearson = sps.pearsonr(harmonic_mean_distances_measure_data,
```

```
     loads_data)
print("Pearson correlation coefficient for the harmonic mean distance measure: {0}, two-sided p-value: {1}".format(*harmonic_mean_distances_measure_pearson))


# List top 10 nodes according to the best measure. List the value of the computed measure and the actual load.

# Select best measure and get top rated nodes.
best_measure = max(((degree_measure_pearson[0], degrees, 'node degree'), (clustering_measure_pearson[0], clustering_coefficients, 'node clustering'),
    (harmonic_mean_distances_measure_pearson[0], harmonic_mean_distances, 'node harmonic mean distance')))
top_nodes = list(map(lambda x: x[0], sorted(best_measure[1].items(), key=lambda x: x[1], reverse=True)))[:10]

# Print results.
print("Best measure: '{0}'".format(best_measure[2]))
print("Top-rated nodes:")
for idx, node in enumerate(top_nodes):
    print("({0}) {1}: measure={2:.4f}, load={3:.4f}".format(idx+1, nodes_to_data[node]['name'], best_measure[1][node], nodes_to_data[node]['load']))
```