Homework #2

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 **April 13th** at 2:00pm, while late days expire on **April 17th** 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

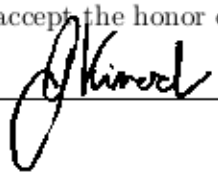
The students are strongly encouraged to discuss the homework with other classmates and form study groups. Yet, each student must then solve the homework by herself or himself without the help of others and should be able to redo the homework at a later time. In other words, the students are encouraged 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, 63160328

Study group: /

Date & time: 13.4.2020, 10:00

I acknowledge and accept the honor code.

Signature: 

Homework #2

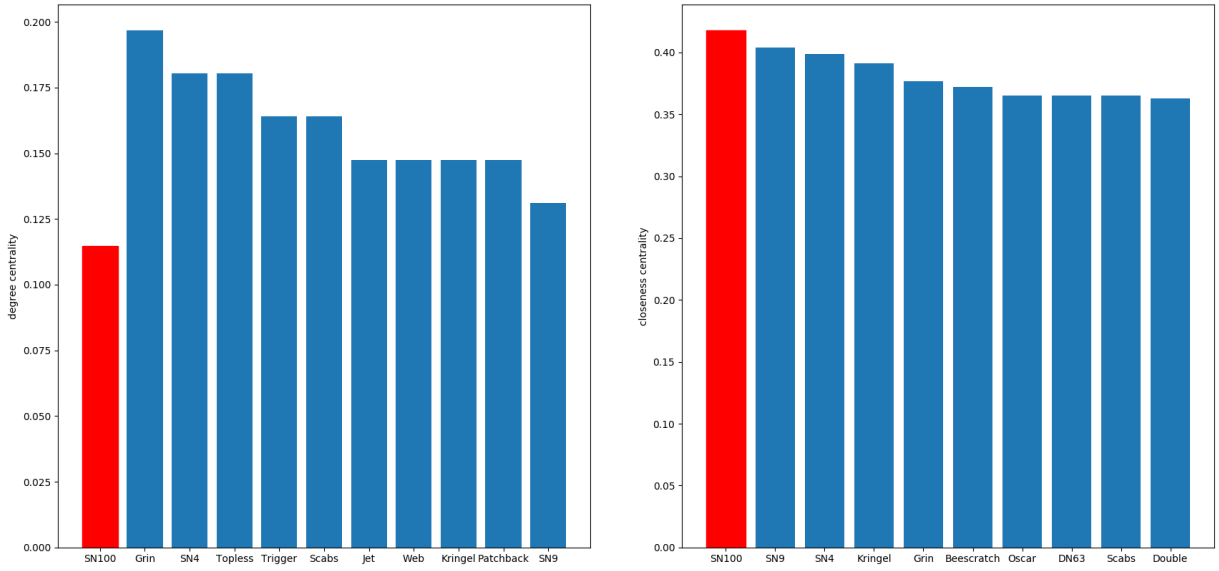
Jernej Vivod
Introduction to Network Analysis

April 13, 2020

Question 1

Rank the importance of a dolphin named SN100 in the provided social network of bottlenose dolphins. Consider at least the suggested measures of node importance and state the rank of SN100 for each measure of node importance and briefly discuss the results.

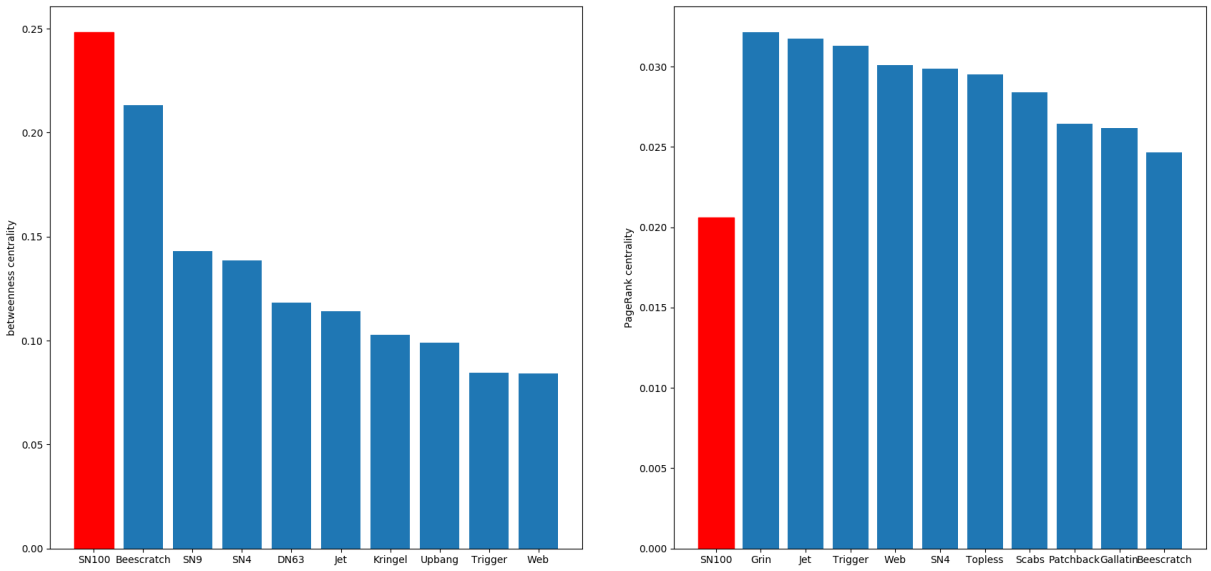
We evaluated the estimated importance of dolphins using degree centrality, closeness centrality, betweenness centrality and PageRank score. The comparison of 10 highest-ranked dolphins and dolphin SN100 is visualized on figures 1 (degree centrality and closeness centrality) and 2 (betweenness centrality and PageRank score).



(a) Degree centrality

(b) Closeness Centrality

Figure 1: Degree centrality and closeness centrality of SN100 compared to other highest-ranked dolphins.



(a) Betweenness centrality

(b) PageRank score

Figure 2: Betweenness centrality and PageRank centrality of SN100 compared to other highest-ranked dolphins.

	Rank	Percentile
Degree centrality	15	74.1935
Closeness centrality	1	100.0
Betweenness centrality	1	100.0
PageRank score	17	74.1935

Table 1

SN100 is ranked 15th for the degree centrality measure, 1st for the closeness centrality measure, 1st for the betweenness centrality measure and 17th for the PageRank measure.

Table 1 shows the rankings as well as the percentiles of the SN100's importance estimates. Thus the node representing the dolphin SN100 is estimated to be very close to other nodes and also to be part of a relatively large number of shortest paths between nodes.

The code used to provide the results is given below.

```
import networkx as nx
import parse_network


def node_importances(graph, measure):
    """
    Compute node importance scores according to specified measure.
    Author: Jernej Vivod

```

```
    Args:
        graph (obj): Networkx representation of the graph
        measure (str): Argument specifying the node importance measure to use.
    Returns:
        (dict): Dictionary mapping node indices to their estimated importances.
    """

    # Check if specified measure valid.
    if measure not in {'degree_centrality', 'PageRank', 'betweenness', 'closeness'}:
        raise(ValueError("the measure parameter can take the values of 'degree_centrality',
            'PageRank', 'betweenness' or 'closeness'"))

    # Compute node importances according to specified measure.
    if measure == 'degree_centrality':
        return nx.degree_centrality(graph)
    elif measure == 'PageRank':
        return nx.pagerank(graph)
    elif measure == 'betweenness':
        return nx.betweenness_centrality(graph)
    elif measure == 'closeness':
        return nx.closeness_centrality(graph)


def node_rank(graph, idx_node, measure):
    """
    Compute node importance rank according to specified measure.
    Author: Jernej Vivod

    Args:
        graph (obj): Networkx representation of the graph
        idx_node (str): Index of node for which to compute the rank.
        measure (str): Argument specifying the node importance measure to use.

    Returns:
        (int): Ranking of specified node according to specified importance measure.
    """

    # Check if specified measure valid.
    if measure not in {'degree_centrality', 'PageRank', 'betweenness', 'closeness'}:
        raise(ValueError("the measure parameter can take the values of 'degree_centrality',
            'PageRank', 'betweenness' or 'closeness'"))

    # Compute rank of node according to specified measure.
    if measure == 'degree_centrality':
        est_imp = nx.degree_centrality(graph)
        return [key for key, val in sorted(est_imp.items(), key=lambda x: x[1], reverse=True
            )].index(idx_node)
    elif measure == 'PageRank':
        est_imp = nx.pagerank(graph)
        return [key for key, val in sorted(est_imp.items(), key=lambda x: x[1], reverse=True
            )].index(idx_node)
    elif measure == 'betweenness':
        est_imp = nx.betweenness_centrality(graph)
        return [key for key, val in sorted(est_imp.items(), key=lambda x: x[1], reverse=True
            )].index(idx_node)
    elif measure == 'closeness':
        est_imp = nx.closeness_centrality(graph)
        return [key for key, val in sorted(est_imp.items(), key=lambda x: x[1], reverse=True
            )].index(idx_node)


def data_most_important(importance_dict, n_most_important, include_additional=None):
    """
    Compute data for making a bar plot of n nodes with highest estimated importance.
    Author: Jernej Vivod

    Args:
        importance_dict (dict): Dictionary mapping node indices to their estimated
```

```
            importances
        n_most_important (int): Number of most imporant nodes to keep
        include_additional (str): Index of additional node from graph to include in plot
            data.
        This node's data is appended to the front of the resulting lists.
    Returns:
        (tuple): tuple of lists of names of n most important nodes and importance scores of
            these nodes.
    """

    # Sort nodes by their evaluated importance.
    sorted_nodes = [(k, v) for k, v in sorted(importance_dict.items(), key=lambda el: el[1],
         reverse=True)]
    x = [nx.get_node_attributes(graph, 'name')[el[0]] for el in sorted_nodes[:n_most_
        important]]
    y = [el[1] for el in sorted_nodes[:n_most_important]]

    # If including additional specified node, add data for it if not yet present.
    if include_additional and nx.get_node_attributes(graph, 'name')[include_additional] not
        in x:
        x.insert(0, nx.get_node_attributes(graph, 'name')[include_additional])
        y.insert(0, list(filter(lambda x: x[0] == include_additional, sorted_nodes))[0][1])

    return x, y


if __name__ == '__main__':
    import matplotlib.pyplot as plt
    from scipy import stats

    # Set name of dolphin of interest.
    DOLPHIN_NAME = 'SN100'

    # Parse the bottlenose dolphin network as well as associated data.
    graph = parse_network.parse_network("../data/dolphins", create_using=nx.Graph)

    # Get index of node corresponding to the dolphin of interest.
    idx_dolphin = [el[0] for el in nx.get_node_attributes(graph, 'name').items() if el[1] ==
         'SN100'][0]

    ### Bar charts of centralities ###
    importances_degree_centrality = node_importances(graph, 'degree_centrality')
    rank1 = node_rank(graph, idx_dolphin, 'degree_centrality')
    x_bar1, y_bar1 = data_most_important(importances_degree_centrality, 10, include_
        additional=idx_dolphin)
    fig1, ax1 = plt.subplots()
    barlist1 = ax1.bar(x_bar1, y_bar1)
    barlist1[0].set_color('r')
    plt.ylabel("degree centrality")
    print("degree centrality for dolphin {0}: {1:.4f}".format(DOLPHIN_NAME, y_bar1[0]))
    print("degree centrality rank for dolphin {0}: {1}".format(DOLPHIN_NAME, rank1+1))
    print("percentile of degree centrality for dolphin {0}: {1:.4f}".format(DOLPHIN_NAME,
        stats.percentileofscore(list(importances_degree_centrality.values()), y_bar1[0])))

    importances_pagerank = node_importances(graph, 'PageRank')
    rank2 = node_rank(graph, idx_dolphin, 'PageRank')
    x_bar2, y_bar2 = data_most_important(importances_pagerank, 10, include_additional=idx_
        dolphin)
    fig2, ax2 = plt.subplots()
    barlist2 = ax2.bar(x_bar2, y_bar2)
    barlist2[0].set_color('r')
    plt.ylabel("PageRank centrality")
    print("PageRank centrality for dolphin {0}: {1:.4f}".format(DOLPHIN_NAME, y_bar2[0]))
    print("PageRank centrality rank for dolphin {0}: {1}".format(DOLPHIN_NAME, rank2+1))
    print("percentile of PageRank centrality for dolphin {0}: {1:.4f}".format(DOLPHIN_NAME,
        stats.percentileofscore(list(importances_pagerank.values()), y_bar2[0])))

    importances_betweenness_centrality = node_importances(graph, 'betweenness')
```

```
    rank3 = node_rank(graph, idx_dolphin, 'betweenness')
    x_bar3, y_bar3 = data_most_important(importances_betweenness_centrality, 10, include_
        additional=idx_dolphin)
    fig3, ax3 = plt.subplots()
    barlist3 = ax3.bar(x_bar3, y_bar3)
    barlist3[0].set_color('r')
    plt.ylabel("betweenness centrality")
    print("betweenness centrality for dolphin {0}: {1:.4f}".format(DOLPHIN_NAME, y_bar3[0]))
    print("betweenness centrality rank for dolphin {0}: {1}".format(DOLPHIN_NAME, rank3+1))
    print("percentile of betweenness centrality for dolphin {0}: {1:.4f}".format(DOLPHIN_
        NAME,
        stats.percentileofscore(list(importances_betweenness_centrality.values()), y_bar3
            [0])))

    importances_closeness_centrality = node_importances(graph, 'closeness')
    rank4 = node_rank(graph, idx_dolphin, 'closeness')
    x_bar4, y_bar4 = data_most_important(importances_closeness_centrality, 10, include_
        additional=idx_dolphin)
    fig4, ax4 = plt.subplots()
    barlist4 = ax4.bar(x_bar4, y_bar4)
    barlist4[0].set_color('r')
    plt.ylabel("closeness centrality")
    print("closeness centrality for dolphin {0}: {1:.4f}".format(DOLPHIN_NAME, y_bar4[0]))
    print("closeness centrality rank for dolphin {0}: {1}".format(DOLPHIN_NAME, rank4+1))
    print("percentile of closeness centrality for dolphin {0}: {1:.4f}".format(DOLPHIN_NAME,
        stats.percentileofscore(list(importances_closeness_centrality.values()), y_bar4[0]))
            )

    plt.show()

    ##############################################
```

Question 2

Plot degree, in-degree and out-degree distributions on a doubly-logarithmic plot. Compare all three degree distributions p_k of each network and highlight the differences between them. Do the distributions appear to be scale-free?

Figure 3 shows the degree, in-degree and out-degree distributions for the *Java Namespace of Java Language* network plotted on a doubly-logarithmic plot. The dashed line represents the power-law model deduced by maximum-likelihood estimation. Figure 4 shows the same information for the *Lucene search engine library* network.

We can see that for both networks, only the in-degree distribution features the characteristic shape of a power-law distribution. Both the degree and out-degree distributions have a notable peak at a domain value some distance away from 0. This can be explained by observing the fact that very few classes implement such specific functionalities as to only be useful as a dependency for a very small set of other classes - hence the peak is located at considerable distance from 0. Similar reasoning can be applied to the degree distribution. However, most classes depend only on a small subset of other classes with a notable quantity of them depending on an unusually high number of classes. Many of such classes are classes that implement some main functionality of a library or program and therefore require many functionalities from more specialized classes. The deduced power-law exponents are stated by table 2. The code used to provide the results is given below.

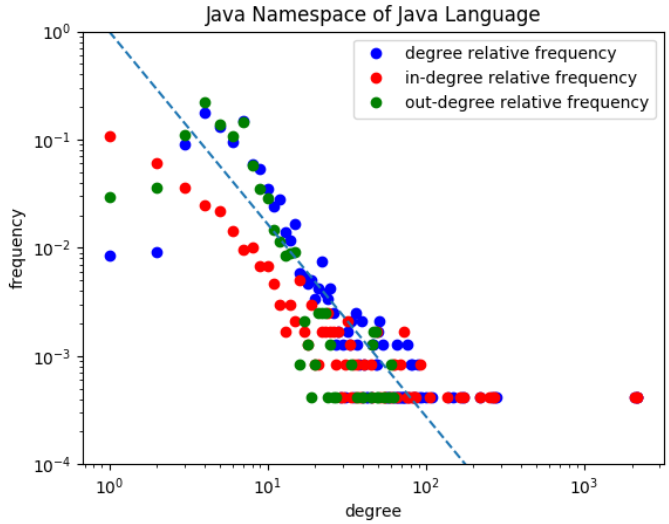


Figure 3: Degree, in-degree, out-degree and fitted power-law distribution model for the *Java Namespace of Java Language* network.

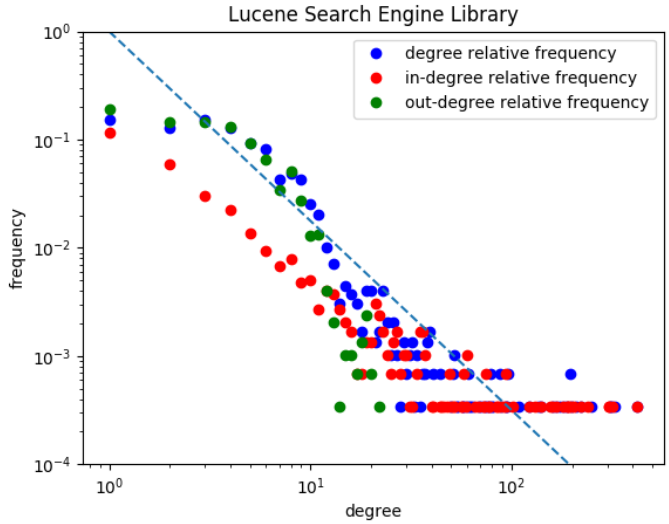


Figure 4: Degree, in-degree, out-degree and fitted power-law distribution model for the *Lucene search engine library* network.

	Power-law exponent
Java Namespace of Java Language	1.7789
Lucene Search Engine Library	1.7503

Table 2: Power-law exponents for the *Java Namespace of Java Language* and *Lucene Search engine* networks.

```
1 import networkx as nx
```

```
import math
import collections
import matplotlib.pyplot as plt


def plot_degree_distributions(graph):
    """
    Plot degree, in-degree and out-degree distribution on a doubly logartihmic plot.
    Author: Jernej Vivod

    Args:
        graph (obj): Networkx graph representation.
    """

    # Compute relative degree, in-degree and out-degree frequencies.
    degree_count = collections.Counter(dict(graph.degree()).values())
    sum_deg_count = sum(degree_count.values())
    degree_dist = {key:degree_count[key]/sum_deg_count for key in degree_count.keys()}
    assert sum(degree_dist.values()) - 1.0 < 1.0e-4

    in_degree_count = collections.Counter(dict(graph.in_degree()).values())
    sum_in_deg_count = sum(in_degree_count.values())
    in_degree_dist = {key:in_degree_count[key]/sum_in_deg_count for key in in_degree_count.
        keys()}
    assert sum(in_degree_dist.values()) - 1.0 < 1.0e-4

    out_degree_count = collections.Counter(dict(graph.out_degree()).values())
    sum_out_deg_count = sum(out_degree_count.values())
    out_degree_dist = {key:out_degree_count[key]/sum_out_deg_count for key in out_degree_
        count.keys()}
    assert sum(out_degree_dist.values()) - 1.0 < 1.0e-4

    # Plot relative degree frequencies on doubly-logarithmic plot.
    fig, ax = plt.subplots()
    ax.loglog(list(degree_dist.keys()), list(degree_dist.values()), 'bo', label="degree
        relative frequency")
    ax.loglog(list(in_degree_dist.keys()), list(in_degree_dist.values()), 'ro', label="in-
        degree relative frequency")
    ax.loglog(list(out_degree_dist.keys()), list(out_degree_dist.values()), 'go', label="out
        -degree relative frequency")
    ax.legend()

    return fig, ax


def power_law_exponent(degrees, min_degree):
    """
    Evaluate power-law exponent gamma using maximum-likelihood estimate.
    Author: Jernej Vivod

    Args:
        degrees (list): List of node degrees
        min_degree (int): Degree cut-off

    Returns:
        (float): maximum-likelihood estimate of the power-law exponent.
    """

    # Count number of nodes falling below the threshold.
    n = len(list(filter(lambda x: x >= min_degree, degrees)))

    # Compute maximum-likelihood estimate for gamma.
    return 1 + n*((sum([math.log(degree/(min_degree - 0.5)) for degree in degrees if degree
        >= min_degree]))**(-1.0))


if __name__ == '__main__':

```

```
    ### Parse graphs ###
    PATH1 = '../data/java'
    PATH2 = '../data/lucene'
    graph_java = nx.read_edgelist(PATH1, create_using=nx.DiGraph)
    graph_lucene = nx.read_edgelist(PATH2, create_using=nx.DiGraph)

    # Set plot titles and axis labels.
    title1 = "Java Namespace of Java Language"
    title2 = "Lucene Search Engine Library"
    xlabel = "degree"
    ylabel = "frequency"

    # Plot exponent estimate or not.
    PLOT_EXPONENT_ESTIMATE = True

    # Plot degree, in-degree and out-degree distributions.
    fig1, ax1 = plot_degree_distributions(graph_java)
    fig2, ax2 = plot_degree_distributions(graph_lucene)

    # Set titles and axis labels.
    ax1.set_title(title1)
    ax2.set_title(title2)
    ax1.set_xlabel(xlabel)
    ax1.set_ylabel(ylabel)
    ax2.set_xlabel(xlabel)
    ax2.set_ylabel(ylabel)

    # Compute power-law exponent using maximum-likelihood estimation.
    gamma1 = power_law_exponent(dict(graph_java.in_degree()).values(), min_degree=3)
    gamma2 = power_law_exponent(dict(graph_lucene.in_degree()).values(), min_degree=3)

    # If plotting exponent estimate on plot.
    if PLOT_EXPONENT_ESTIMATE:
        dom1 = range(1, max(dict(graph_java.in_degree()).values())+1)
        dom2 = range(1, max(dict(graph_lucene.in_degree()).values())+1)
        ax1.loglog(dom1, [el**(-gamma1) for el in dom1], '--')
        ax2.loglog(dom2, [el**(-gamma2) for el in dom2], '--')
        ylim = (1.0e-4, 1)
        ax1.set_ylim(ylim[0], ylim[1])
        ax2.set_ylim(ylim[0], ylim[1])

    # Show plot.
    plt.show()
```

Question 3

Consider the provided Internet overlay map represented by an undirected graph. Study how removing a fraction of random nodes affects the fraction of nodes in the largest connected component. Study how removing a fraction of nodes with the highest degrees affects the fraction of nodes in the largest connected component. Repeat the same computations on an Erdős–Rényi random graph model with parameters n and m taken from the Internet overlay map network. Briefly discuss the results.

Figure 5 shows the fraction of nodes remaining in the largest connected component after a fraction of either randomly selected nodes or nodes with highest degrees (hubs) were removed. We can see that removing nodes of very high degree quickly deteriorates the connectedness of the *Internet overlay map* network. This is due to these nodes acting as hubs

and forming vital bridges within the network. oppositely, removing fractions of randomly chosen nodes in the *Internet overlay map* network does not have a large detrimental effect on the connectedness of the network as the probability of choosing a vital hub that is important in ensuring the netowrk stays connected is relatively low.

On the other hand, the Erdős–Rényi model does not feature prominent hubs or nodes of high bridgeness and the fraction of nodes in the largest connected component is therefore very robust to removal of random as well as high-degree nodes.

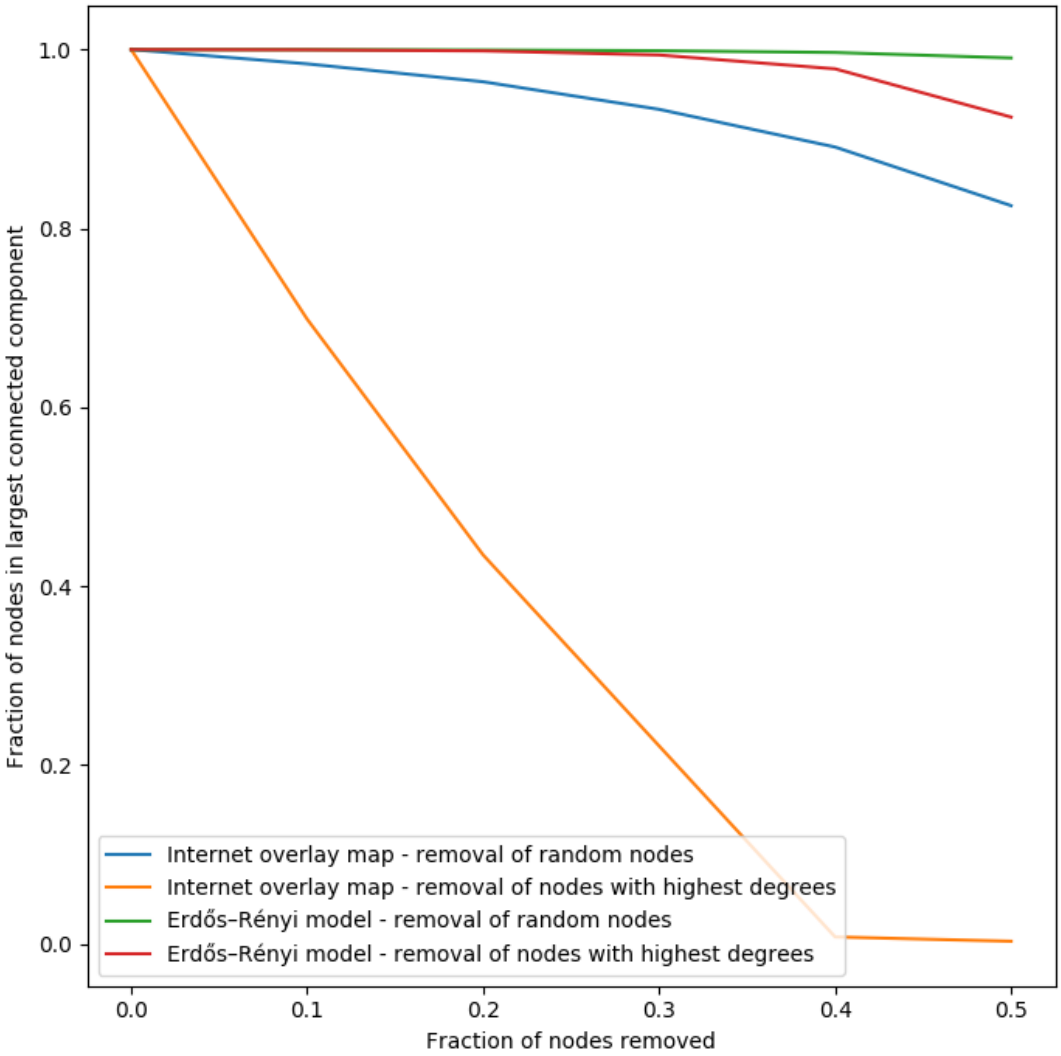


Figure 5: Fraction of nodes in the largest connected component after a fraction of either randomly selected nodes or nodes with highest degrees (hubs) were removed.

The code used to provide the results is given below.

```
import networkx as nx
import parse_network
import random

def remove_frac_nodes(graph, frac, remove_hubs):
    """
    Remove a fraction of nodes in specified graph.
    Author: Jernej Vivod

    Args:
        graph (obj): Networkx representation of a graph.
        frac (float): The fraction of nodes to remove from the graph.
        remove_hubs (bool): If set to true, remove specified fraction of
        nodes with highest degree. Else select nodes to be removed randomly.

    Returns:
        (obj): Networkx representation of a graph with specified
        fraction of nodes removed.
    """

    if frac < 0.0 or frac > 1.0:
        raise ValueError("Fraction must be between 0.0 and 1.0")
    else:
        if remove_hubs:
            # Remove fraction of nodes with highest degree.
            to_remove = [key for (key, val) in sorted(dict(graph.degree()).items(), key=
                lambda x: x[1], reverse=True)][:round(frac*graph.number_of_nodes())]
            graph.remove_nodes_from(to_remove)
            return graph
        else:
            # Remove fraction of randomly selected nodes from graph.
            graph.remove_nodes_from(random.sample(graph.nodes(), round(frac*graph.number_of_
                nodes())))
            return graph


def components(graph):
    """
    Find connected component in undirected graph.
    Author: Jernej Vivod

    Args:
        graph (obj): Networkx representation of a network

    Returns:
        (list): List of lists containing node IDs representing connected components.
    """

    # Empty list for storing the connected components
    connected_components = []

    # Perform DFS to find connected components.
    while graph.number_of_nodes() > 0:
        connected_components.append(component(graph))

    # Return connected components.
    return connected_components


def component(graph):
    """
    Find next connected component in graph.
    Author: Jernej Vivod

    Args:
```

```
        graph (obj): Networkx representation of a network

    Returns:
        (list): the list containing the node IDs constituting the connected component.
    """

    # Empty list for storing the next connected component
    component = []

    # Stack for implementing DFS
    stack = []

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

    # Perform DFS.
    while len(stack) > 0:

        # Pop next node from stack.
        node_nxt = stack.pop()

        # If node in graph.
        if node_nxt in graph:

            # Append index of current node to component list.
            component.append(node_nxt)

            # Go over neighbors of current node.
            for el in graph.neighbors(node_nxt):

                # If neighbor not yet visited, add to stack.
                if el in graph:
                    stack.append(el)

            # Remove current node from graph.
            graph.remove_node(node_nxt)

    # Return found connected component.
    return component


def frac_in_lcc(graph):
    """
    Compute fraction of nodes in largest connected component.
    Author: Jernej Vivod

    Args:
        graph (obj): Networkx representation of a graph.

    Returns:
        (float): Fraction of nodes in largest connected component.
    """

    cc = components(graph.copy())
    return max(map(lambda x: len(x), cc))/graph.number_of_nodes()


if __name__ == '__main__':
    import matplotlib.pyplot as plt

    # Parse network.
    PATH = "../data/nec"
    graph = parse_network.parse_network(PATH, create_using=nx.Graph)

    # Construct Erdos-Renyi model with same number of nodes and edges.
    graph_er_model = nx.gnm_random_graph(graph.number_of_nodes(), graph.number_of_edges())

```

```
    # Initialize list of fractions of nodes to remove.
    fracs = [0, 0.1, 0.2, 0.3, 0.4, 0.5]

    # Initialize lists for storing results.
    frac_lcc_graph_rm_rand = []
    frac_lcc_graph_rm_hubs = []
    frac_lcc_graph_er_rm_rand = []
    frac_lcc_graph_er_rm_hubs = []

    # Go over list of fractions.
    for frac in fracs:
        print(frac)

        # Remove fraction of nodes from graph and Erdos-Renyi model.
        graph_rm_rand = remove_frac_nodes(graph.copy(), frac, False)
        graph_rm_hubs = remove_frac_nodes(graph.copy(), frac, True)
        graph_er_rm_rand = remove_frac_nodes(graph_er_model.copy(), frac, False)
        graph_er_rm_hubs = remove_frac_nodes(graph_er_model.copy(), frac, True)

        # Add computed fractions of nodes in largest connected component to results lists.
        frac_lcc_graph_rm_rand.append(frac_in_lcc(graph_rm_rand))
        frac_lcc_graph_rm_hubs.append(frac_in_lcc(graph_rm_hubs))
        frac_lcc_graph_er_rm_rand.append(frac_in_lcc(graph_er_rm_rand))
        frac_lcc_graph_er_rm_hubs.append(frac_in_lcc(graph_er_rm_hubs))

    # Plot fractions of nodes in largest connected component with respect to fraction of
        nodes removed.
    fig, ax = plt.subplots()
    ax.plot(fracs, frac_lcc_graph_rm_rand, label="Internet overlay map - removal of random
        nodes")
    ax.plot(fracs, frac_lcc_graph_rm_hubs, label="Internet overlay map - removal of nodes
        with highest degrees")
    ax.plot(fracs, frac_lcc_graph_er_rm_rand, label="Erdos-Renyi model - removal of random
        nodes")
    ax.plot(fracs, frac_lcc_graph_er_rm_hubs, label="Erdos-Renyi model - removal of nodes
        with highest degrees")
    ax.legend()
    plt.xlabel("Fraction of nodes removed")
    plt.ylabel("Fraction of nodes in largest connected component")
    plt.show()
```

Question 4

Perform a random walk on the provided social network until you sample 10% of the nodes and take the induced graph constructed from such sampled nodes. Is the original social network small-world and/or seemingly scale-free? Is the sampled network small-world and/or seemingly scale-free? Reason why and support your answers with appropriate computations.

Figure 6 shows the node degree distribution for original graph and sampled graph. We can see that the distribution in the original plot follows a power-law distribution and the sampled graph also follows an approximate power-law distribution. When the fraction of nodes in the sample decreases, the frequency of low degree nodes decreases somewhat as there is a higher chance of the random walk sampling nodes with a higher degree and we notice a peak in the distribution somewhat distant from 0.

Table 3 shows the average shortest distance and average clustering of the original and sampled graphs. We can notice that the shortest paths are short especially when compared

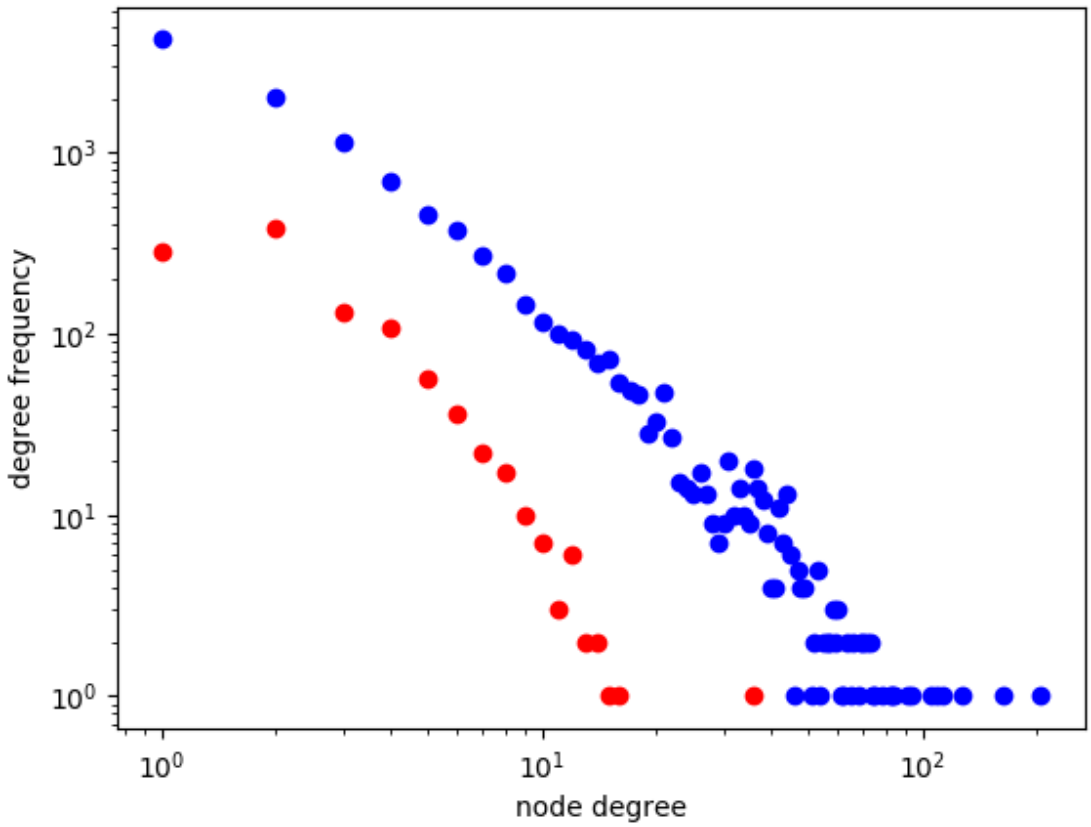


Figure 6: Degree distribution in original graph and in graph induced by random walk.

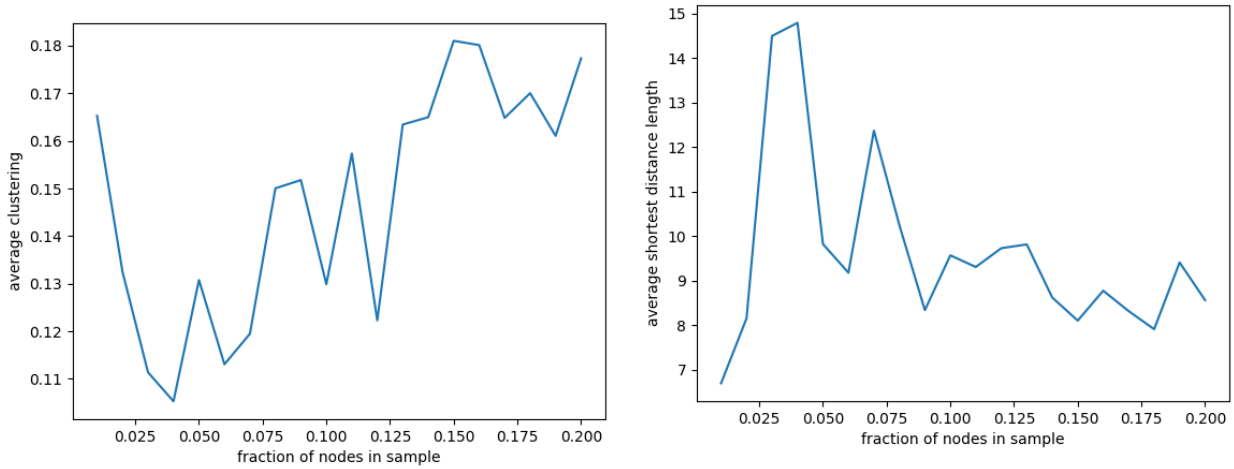
to the large number of nodes constituting the network. The clustering coefficient is also much higher than expected by chance. Both networks can therefore also be classified as small-world.

	Average shortest distance	Average clustering
Original graph	7.4855	0.2659
Sampled graph (10%)	8.5413	0.1353

Table 3: Average shortest path distance and average clustering for original and sampled graph.

It is also interesting to plot the average clustering and average shortest path lengths for different sample sizes as shown on figure 7. We can see that the average shortest path length soon settles in close vicinity of the average shortest path length in the original graph. The average clustering is somewhat lower but still much higher than would be expected by chance.

The code used to provide the results is given below.



(a) Average clustering for varying sample sizes. (b) Average shortest distance for varying sample sizes.

Figure 7: Average clustering and average shortest path length for varying sample sizes.

```python
import networkx as nx
import parse_network
import random
import collections

def random_walk(graph, frac_sample=0.1):
    """
    Perform random walk on specified graph until specified fraction
    of nodes have been covered. Return graph induced by such a random
    walk.

    Args:
        (obj): Networkx graph representation.
        frac_sample: Fraction of nodes in graph to cover.

    Returns:
        (obj): Networkx graph representation of the graph induced
        by the random walk.
    """

    # Intialize set of visited nodes.
    visited = set()

    # Initialize set of traversed edges.
    walk_edges = set()

    # Randomly choose starting node.
    start_node = random.choice(list(graph.nodes()))
    node_current = start_node

    # While specified fraction of network not covered, perform random walk.
    while len(visited) < frac_sample*graph.number_of_nodes():

        # Visit randomly chosen neighbor of current node and add to results sets.
        node_nxt = random.choice(list(graph.neighbors(node_current)))
        visited.add(node_nxt)
        walk_edges.add((node_current, node_nxt))
        node_current = node_nxt
```

```
39
40      # Construct graph from traversed edges and return it.
41      ind_graph = nx.Graph()
42      ind_graph.add_edges_from(walk_edges)
43      return ind_graph
44
45
46
47
48  if __name__ == '__main__':
49      import matplotlib.pyplot as plt
50
51      # Parse network.
52      PATH = "../data/social"
53      graph = parse_network.parse_network(PATH, create_using=nx.Graph)
54
55      # Get graph induced by random walk that covers 10% of the nodes.
56      ind_graph = random_walk(graph, 0.06)
57
58      # Print average distance and clustering in original and sampled graph.
59      # print("Average distance - original graph: {0}".format(nx.average_shortest_path_length(
            graph)))
60      print("Average distance - sampled graph: {0}".format(nx.average_shortest_path_length(ind
            _graph)))
61
62      # print("Average clustering - original graph: {0}".format(nx.average_clustering(graph)))
63      print("Average clustering - sampled graph: {0}".format(nx.average_clustering(ind_graph))
            )
64
65      # Plot degree distributions for original and sampled graphs.
66      degree_freq_original = collections.Counter(dict(graph.degree()).values())
67      x_original, y_original = list(degree_freq_original.keys()), list(degree_freq_original.
            values())
68
69      degree_freq_induced = collections.Counter(dict(ind_graph.degree()).values())
70      x_ind, y_ind = list(degree_freq_induced.keys()), list(degree_freq_induced.values())
71
72      fig, ax = plt.subplots()
73      ax.loglog(x_original, y_original, 'bo')
74      ax.loglog(x_ind, y_ind, 'ro')
75      plt.xlabel("node degree")
76      plt.ylabel("degree frequency")
77      plt.show()
```

Question 5

Probability that an individual would spread the disease through their social network is proportional to k^2, where k is the degree of the corresponding node. Consider two immunization schemes for preventing the spread of diseases. In the first scheme, you randomly select some number of individuals and vaccinate them. In the second scheme, you randomly select the same number of individuals, but then rather vaccinate a random acquaintance of theirs. Which of the two schemes do you expect to provide better immmunization? Why?

According to the so-called friendship paradox, which can be explained as a form of sampling bias, the friends or acquaintances of a person have, on average, a greater number of friends or acqaitances than that particular person. By this fact, the second scheme should be more successful as there is a higher change of selecting individuals with a large social

circle and therefore considerably higher probability of spreading the disease through their social network.

We can simulate this empirically by writing a short script that simulates the procedure some number of times and reports the normalized sum of squared degrees of nodes representing unvaccinated people as a measure of the effectiveness of the vaccination program.

Table 4 shows the results for 30 repetitions of both vaccination schemes where 10% of subjects are randomly selected.

	Measure
First scheme	77.3959
Second scheme	53.2773

Table 4: Sum of squared degrees of nodes representing unvaccinated persons for both vaccination schemes.

We can see a notable difference in the normalized sum of squared nodes representing unvaccinated persons that is stable when repetitions and fractions of selected sample are altered. As predicted by computations supporting the so-called friendship paradox, the second vaccination scheme is notably better in targeting individuals with a high probability of spreading the disease through their social circle.

The code used to simulate the vaccination schemes is given below.

```
import networkx as nx
import parse_network
import random


def mark_random_nodes(graph, num_to_mark):
    """
    Mark random specified number of nodes in specified graph by returning
    them in a list of node IDs.

    Args:
        graph (obj): Networkx representation of a graph.
        num_to_mark (int): number of nodes to mark (return in a list).

    Returns:
        (list): List of marked nodes.
    """

    # Randomly sample specified number of nodes.
    return random.sample(list(graph.nodes()), num_to_mark)


def mark_random_nodes_neighbors(graph, num_to_mark):
    """
    Select specified number of random nodes and then randomly mark/select
    a neighbor of each one and return a list of such marked nodes.

    Args:
        graph (obj): Networkx representation of a graph.
        num_to_mark (int): number of nodes to mark (return in a list).

    Returns:
        (list): List of marked nodes.
    """

```

```
36      # Initialize list for storing marked nodes.
37      marked = []
38
39      # Randomly sample specified number of nodes.
40      sample = random.sample(list(graph.nodes()), num_to_mark)
41
42      # Go over sampled nodes and mark randomly chosen neighbors.
43      for node in sample:
44          marked.append(random.choice(list(graph.neighbors(node))))
45
46      # Return list of marked nodes.
47      return marked
48
49
50  if __name__ == '__main__':
51
52      # Parse network.
53      PATH = '../data/social'
54      graph = parse_network.parse_network(PATH, create_using=nx.Graph)
55
56      # Fraction of nodes to mark and number of runs to perform.
57      FRAC_TO_MARK = 0.1
58      NUM_RUNS = 30
59
60      # Initialize results aggregate.
61      aggr1 = 0
62      aggr2 = 0
63
64      for idx in range(NUM_RUNS):
65
66          # Randomly mark specified fraction of nodes.
67          marked_scheme1 = mark_random_nodes(graph, int(round(graph.number_of_nodes()*FRAC_TO_
                MARK)))
68
69          # Randomly mark random neighbors of specified fraction of nodes.
70          marked_scheme2 = mark_random_nodes_neighbors(graph, int(round(graph.number_of_nodes
                ()*FRAC_TO_MARK)))
71
72          # Sum squared degrees of unmarked nodes normalized by number of nodes in graph and
                add to aggregate.
73          sum_norm_squared1 = sum([graph.degree()[el]**2/graph.number_of_nodes() for el in
                graph.nodes() if el not in marked_scheme1])
74          sum_norm_squared2 = sum([graph.degree()[el]**2/graph.number_of_nodes() for el in
                graph.nodes() if el not in marked_scheme2])
75          aggr1 += sum_norm_squared1
76          aggr2 += sum_norm_squared2
77
78      # Compute average sum of normalized squared node degrees for the two marking schemes.
79      avg_sum_norm_squared1 = aggr1/NUM_RUNS
80      avg_sum_norm_squared2 = aggr2/NUM_RUNS
81
82      # Print results.
83      print("Average sum of normalized squared node degrees for first marking scheme ({0}
            marked, {1} runs): {2:.4f}".format(NUM_RUNS, FRAC_TO_MARK, avg_sum_norm_squared1))
84      print("Average sum of normalized squared node degrees for second marking scheme ({0}
            marked, {1} runs): {2:.4f}".format(NUM_RUNS, FRAC_TO_MARK, avg_sum_norm_squared2))
```