Uppsala University

Modelling Complex Systems

1MA256

Lab4 report

Author:
Kristensen.Samuel
Place of publication: Uppsala

May 9, 2024

1 Task 1

1.1 Selection of dataset

We pick a dataset from Pajek datasets that does not have a too high number of vertices. The selected dataset is Football which contains vertices (n) = 35. The network describes the 22 soccer teams which participated in the World Championship in Paris, 1998. Players of the national team often have contracts in other countries. This constitutes a players market where national teams export players to other countries. Members of the 22 teams had contracts in altogether 35 countries.

1.2 Read and draw the dataset

We read the selected dataset, store it in a adjacency matrix and then draw it. The layout is a circle to help seeing each countries connection with each other and the thickness of the lines between them corresponds with the weight. The lines also have direction to help see how they are connected.

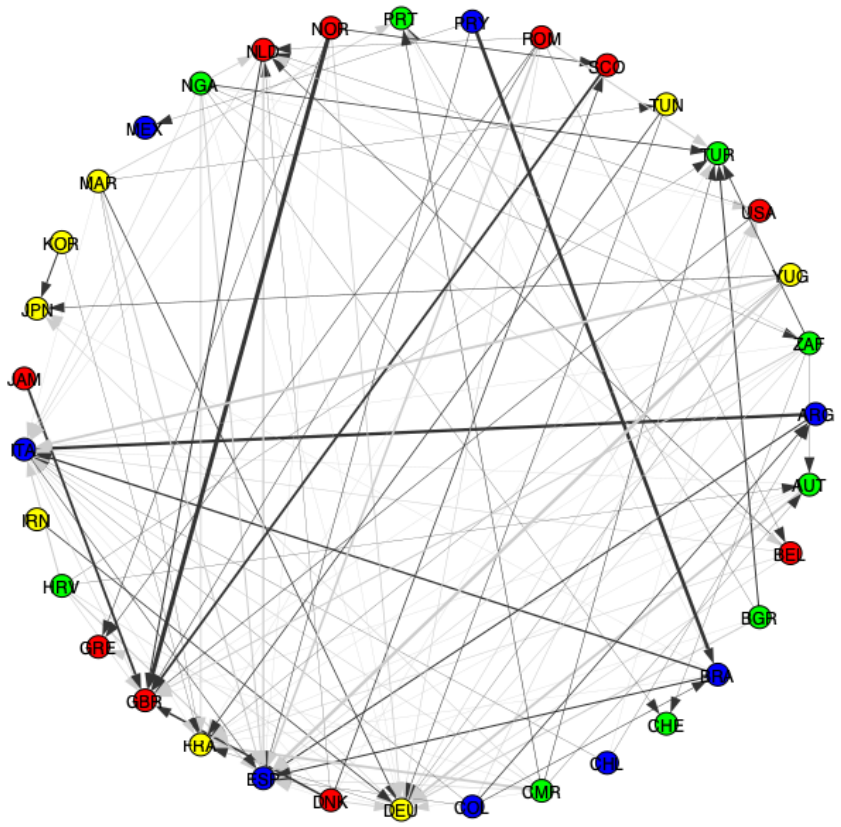


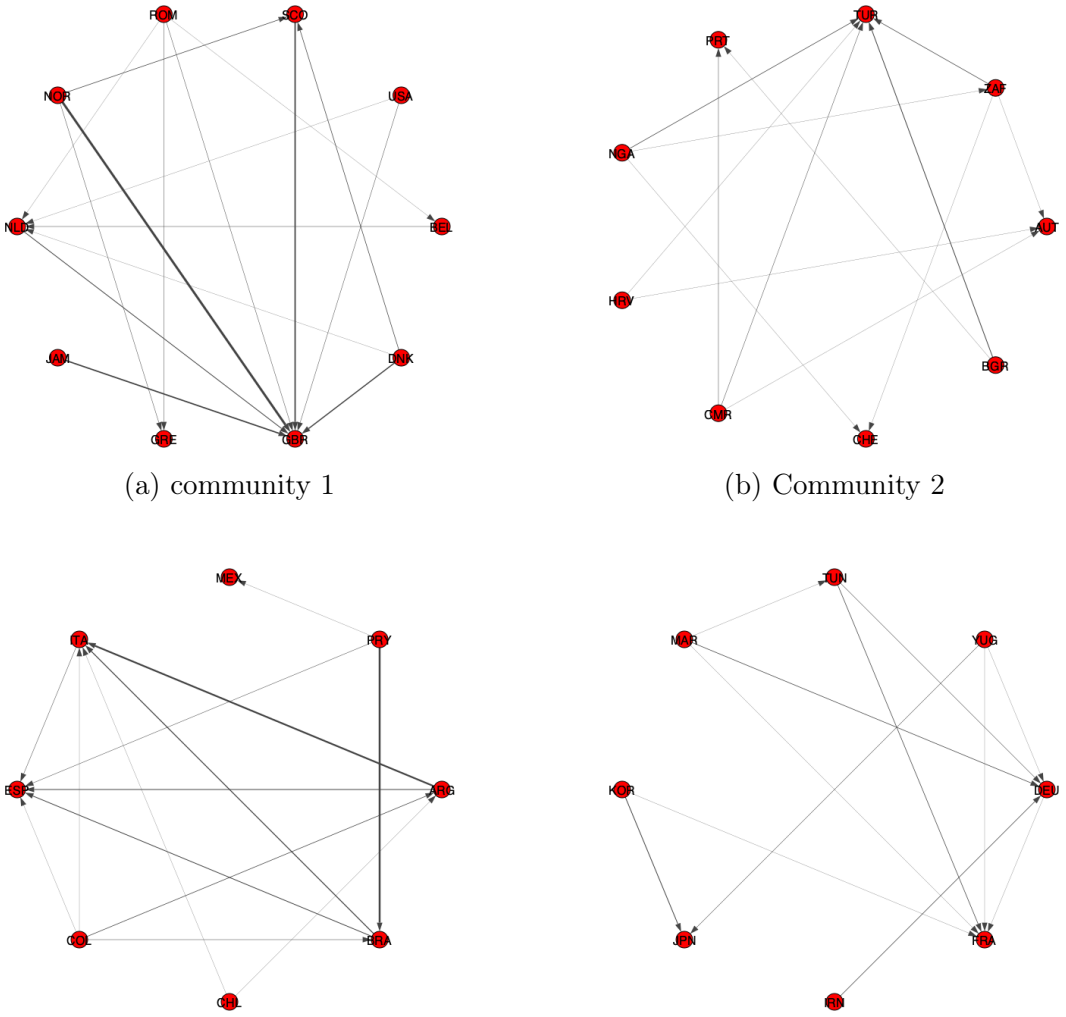
Figure 1: Graph of the network

Figure 1 depicts the network and how all the countries are connected to each other with weighted and directional arrows.

1.3 Communities

We can identify the communities by looking at figure 1 as they are colour coordinated there. The partitions is calculated using the Leiden algorithm. Doing that we can plot

the different communities by themself.



(a) community 1

(b) Community 2

(c) Community 3

(d) Community 4

Figure 2: The 4 communities found via Leiden algorithm

Figure 2 shows us the four communities found using the Leiden algorithm and how they are connected to each other.

Note: The partition using Leiden algorithm somehow yielded different results for multiple runs, sometimes with 4 communities and sometimes with 5. Therefore i choose the 4 communities and went onward from there.

1.4 Modularity

We now calculate the modularity of this network. The formula for modularity is

$$Q = \frac{1}{2m} \sum_{vw} \left[A_{vw} - \frac{k_v k_w}{2m} \right] \delta(c_v, c_w) \tag{1}$$

Where m is the number of total edges, A_{vw} is the number of edges between node v and w, and $\frac{k_v k_w}{2m}$ is the expected number of edges between node v and w.

The computation of modularity for this network came out to be 0.228 which is quite close to 0 and suggests a lack of community structure. An implementation of code for calculating modularity using the igraph functions were implemented and resulted in that $Q = 0.238$ which is close.

2 Task 2

We shall now write a code that generates a random Erdös-Rényi graph G(n, p). To do this we use the function erdos_renyi_graph from the networkx library.

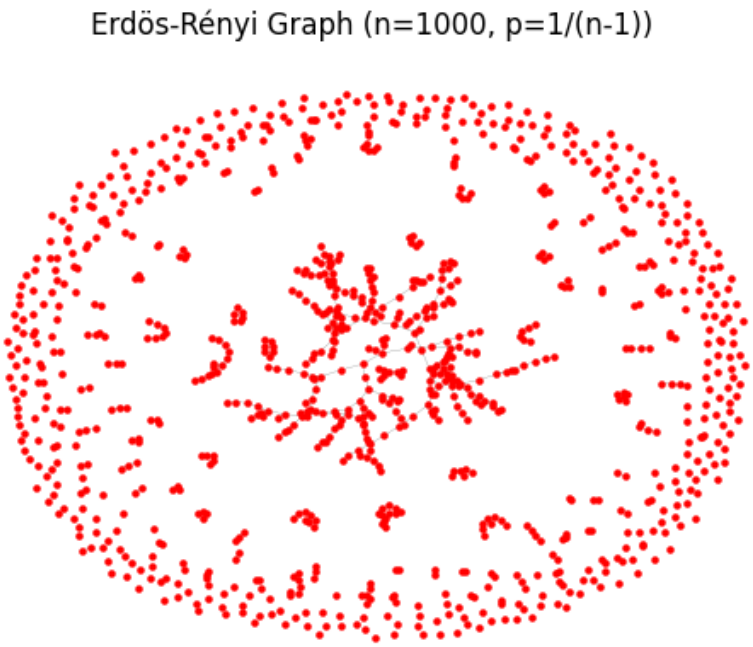


Figure 3: Erdös-Rényi graph

Figure 3 shows the Erdös-Rényi graph for when n=1000 (number of nodes) and $p = \frac{1}{n-1}$ (probability of edge existence) (same as wiki).

2.1 Isolated components

We will now check the threshold for the connectedness of G(n,p). We will do this by:

1. Verify that for the probability $p < \frac{(1-\epsilon)ln(n)}{n}$ the graph G(n, p) is almost surely disconnected: generate many such graphs with some large n, and count how many of them have an isolated component.
2. Verify that for $p > \frac{(1+\epsilon)ln(n)}{n}$ the graph G(n, p) is almost surely connected: : generate many such graphs with some large n, and count how many of them have no isolated component

We start with 1. For that we set $p = \frac{(1-\epsilon)ln(n)}{n} - \epsilon$, where $\epsilon = 0.005$ and n=1000. We use the erdos_renyi_graph for 1000 different graphs and to see if the graph contain an isolated component we use the networkx number_of_isolates to see if there are one or more isolated components. The result for the run were that the number of graph with at leas one isolated component were 1000 out of 1000.

For 2. we do the same thing but set $p = \frac{(1+\epsilon)ln(n)}{n} + \epsilon$, with $\epsilon = 0.005$, and n=1000. Since we count the number of graphs that are disconnected we just take 1000- what we get to count the number of graphs that are connected. From the simulation we got that Number of graphs with isolated component: 8 out of 1000 so that means that 992 graphs had no isolated components.

3 Task 3

We will now write a code that computes the global clustering coefficient C of G(n, p). C can be calculated as

$$C = \frac{\sum_{i,j,k} A_{ij}A_{jk}A_{ki}}{\sum_i n_i(n_i - 1)} \tag{2}$$

Where $n_i = \sum_j A_{ij}$. We fix a large n and plot expected C of G(n, p) as a function of p. That is after n has been fixed, we pick a p, generate many G(n, p)'s for that p, and compute the expected value of C. We repeat this for some discretization of [0, 1] $\ni$ p. In this way, we will obtain the plot of the expected value of C as a function of p.

The problem for this task was the computational time to run several G(n,p) for a number of p and for each one do a triple loop to calculate the numerator. I tried to find some other way to calculate this by for example using the equivelant formula

$$C = \frac{3 \times \text{number of triangles}}{\text{number of all triplets}} \tag{3}$$

However i could not find any good way to do this escpecially since the networkx function all_triples has been removed. So we did this in the way described by equation 2.

For the simulation we used n=100, number of graphs per p value = 10, and p is between 0 and 1 with 32 different equally spaced values. The number 32 was choosen because the code was implemented using multiprocessing and the max cores for my computer is 8 and 32 is a multiple of 8. Even though the code was parallel for this configuration it took about an hour to simulate for all the p values.

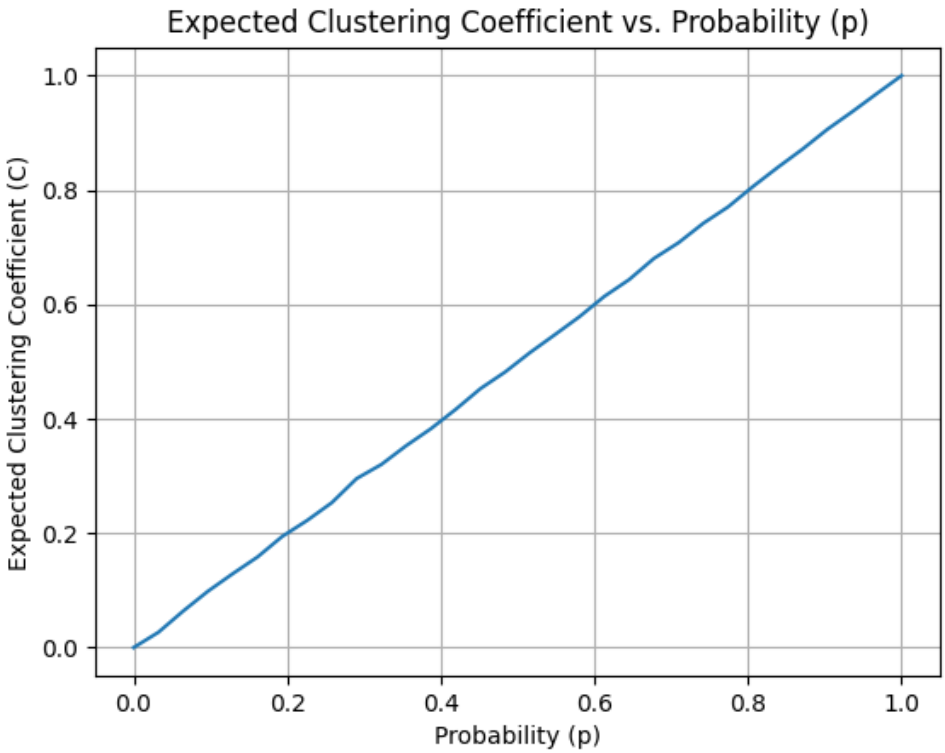


Figure 4: Expected C for different p values

The figure 4 shows how the expected clustering coefficient C depends on the probability p. We see that it is linear with p and comparing this with the theoretical results that $C = p + \mathcal{O}(n^{-1/2})$ we can conclude that it confirms the result we got.

4 Code

4.1 Code for Task 1

```
import numpy as np
import urllib.request
import re
from igraph import *
import leidenalg as la
import networkx as nx

import igraph as ig

# Read the data from the URL
url =
    urllib.request.urlopen("http://vlado.fmf.uni-lj.si/pub/networks/data/sport/football.net")
data_file = url.readlines()


G = nx.read_pajek('Football.net')
countries = list(G.nodes)

# Create Adjacency matrix
adjacency_matrix = np.zeros((35,35))
weightlist = []

# G.edges did not give us the weight so we have to do this by hand
for arcs in data_file[37:-1]:
    arc = arcs.decode('utf-8')
    words_list = re.split(r'\s+', arc)
    row = int(words_list[1])
    col = int(words_list[2])
    weight = int(words_list[3])
    weightlist.append(weight)
    adjacency_matrix[row-1, col-1] = weight

g = Graph.Weighted_Adjacency(adjacency_matrix.tolist())

# Set vertex and enge attributes
g.vs["label"] = countries
g.es["width"] = [x/4 for x in weightlist]
g.es["arrow_size"] = 0.8

# Use Leiden algorithm to find clusters
partition = la.find_partition(g, la.ModularityVertexPartition)

# Plot the graph with community structure
plot(partition, vertex_size=16, vertex_label_size=12, edge_width=g.es["width"],
    edge_arrow_size=g.es["arrow_size"], layout=g.layout_circle(),
    target="football_network.png")
```

```
ig.plot(partition,target="football_network2.png")

# Create subgraphs for each cluster in partition
clusters = [g.subgraph(cluster) for cluster in partition]

# Plot each subgraph containing the clusters
for idx, cluster_graph in enumerate(clusters):
    plot(cluster_graph, vertex_size=22, vertex_label_size=16,
        edge_width=cluster_graph.es["width"],
        edge_arrow_size=cluster_graph.es["arrow_size"],
        layout=cluster_graph.layout_circle(),
        target=f"cluster_{idx+1}_network.png")

communities = [[2, 10, 13, 14, 18, 24, 25, 28, 29, 32], [1, 3, 5, 7, 15, 23,
    26, 31, 34], [0, 4, 6, 8, 11, 17, 22, 27], [9, 12, 16, 19, 20, 21, 30, 33]]

# Initialize Q
Q = 0
m = np.sum(adjacency_matrix) / 2

# Iterate over each community
for i, community_i in enumerate(communities):
    for j, community_j in enumerate(communities):
        delta = 1 if i == j else 0
        Aij = 0
        sum_of_degrees_i = np.sum([np.sum(adjacency_matrix[node]) for node in
            community_i])
        sum_of_degrees_j = np.sum([np.sum(adjacency_matrix[node]) for node in
            community_j])
        for node_i in community_i:
            for node_j in community_j:
                Aij += adjacency_matrix[node_i][node_j]
        expected_edges = (sum_of_degrees_i * sum_of_degrees_j) / (2 * m)
        Q += (Aij - expected_edges) * delta

# Normalize Q by the total number of edges
Q /= (2 * m)

print("Modularity:", Q)


# Create an igraph Graph object from the adjacency matrix
g = ig.Graph.Adjacency((adjacency_matrix > 0).tolist())

# Assign community membership to each node
membership = [0] * len(adjacency_matrix)
for i, community in enumerate(communities):
    for node in community:
        membership[node] = i

# Compute Q using igraph's Q function
```

```
Q = g.modularity(membership)

print("Modularity (using igraph function):", Q)
```

4.2 Code for Task 2

```
import numpy as np
import networkx as nx
import matplotlib.pyplot as plt
from tqdm import tqdm

def is_graph_disconnected(graph):
    """
    Check if a graph is disconnected.

    Parameters:
        graph (nx.Graph): Input graph.

    Returns:
        bool: True if the graph is disconnected, False otherwise.
    """
    return nx.number_of_isolates(graph) > 0

def generate_erdos_renyi(n,p):
    """
    Generate many Erds- Rnyi  graphs with given parameters.

    Parameters:
        n (int): Number of nodes in the graph
        p (float): Probability of edge existence between any two nodes.

    Returns:
        erdos_renyi_graph
    """
    return nx.erdos_renyi_graph(n,p)

def count_disconnected_graphs(n, p, num_graphs):
    """
    Count how many graphs are disconnected.

    Parameters:
        n (int): Number of nodes in the graph.
        p (float): Probability of edge existence between any two nodes.
        num_graphs (int): Number of graphs to generate.

    Returns:
        int: Number of disconnected graphs.
    """
    num_disconnected = 0
    for _ in tqdm(range(num_graphs)):
```

```
        graph = generate_erdos_renyi(n,p)
        if is_graph_disconnected(graph):
            num_disconnected += 1
    return num_disconnected

def plot_graph(graph):
    """
    Plots a single Erds- Rnyi  graph with given parameters.

    Parameters:
        graph (nx.Graph): Input graph.
    """
    plt.figure()
    plt.title(f"Erds- Rnyi  Graph (n={n}, p=1/(n-1))")
    nx.draw(graph, with_labels=False, node_color='red', node_size=5, width=0.1)
    plt.show()


# Parameters
n = 1000 # Number of nodes
eps = 0.005 # Small value for epsilon
p = (1 - eps) * np.log(n)/n - eps # Probability of edge existence
num_graphs = 1000 # Number of graphs to generate

Erdos_Renyi = generate_erdos_renyi(n,p)
#plot_graph(Erdos_Renyi)

# Generate and count disconnected graphs
num_disconnected = count_disconnected_graphs(n, p, num_graphs)

# Print results
print(f"Number of graphs with isolated component: {num_disconnected} out of
    {num_graphs}")
```

4.3 Code for task 3

```
import numpy as np
import networkx as nx
import matplotlib.pyplot as plt
from tqdm import tqdm
import multiprocessing
from functools import partial

def compute_clustering_coefficient(graph):
    """
    Compute the global clustering coefficient of a graph.

    Parameters:
        graph (nx.Graph): Input graph.
```

```
    Returns:
        float: Global clustering coefficient.
    """
    adjacency_matrix = nx.adjacency_matrix(graph)
    number_of_nodes = graph.number_of_nodes()
    numerator = 0
    for i in range(number_of_nodes):
        for j in range(number_of_nodes):
            for k in range(number_of_nodes):
                numerator +=
                    adjacency_matrix[i,j]*adjacency_matrix[j,k]*adjacency_matrix[k,i]

    denominator = 0
    for row in adjacency_matrix:
        n_i = np.sum(row)
        denominator += n_i*(n_i-1)
    if denominator == 0:
        return 0
    return numerator/denominator

def expected_clustering_coefficient_single(p, n, num_graphs):
    """
    Compute the expected clustering coefficient for G(n, p) in a single process.

    Parameters:
        p (float): Probability of edge existence between any two nodes.
        n (int): Number of nodes in the graph.
        num_graphs (int): Number of graphs to generate.

    Returns:
        float: Expected clustering coefficient.
    """
    avg_clustering_coefficient = 0
    for _ in tqdm(range(num_graphs)):
        graph = nx.erdos_renyi_graph(n, p)
        avg_clustering_coefficient += compute_clustering_coefficient(graph)
    avg_clustering_coefficient /= num_graphs
    return avg_clustering_coefficient

def expected_clustering_coefficient_parallel(p_values, n, num_graphs):
    """
    Compute the expected clustering coefficient for G(n, p) in parallel.

    Parameters:
        p_values (list): List of probabilities of edge existence between any two
            nodes.
        n (int): Number of nodes in the graph.
        num_graphs (int): Number of graphs to generate.

    Returns:
        list: List of expected clustering coefficients.
```

```
    """
    # Create a pool of workers
    pool = multiprocessing.Pool()
    func = partial(expected_clustering_coefficient_single, n=n,
        num_graphs=num_graphs)
    expected_C_values = list(pool.map(func, p_values))
    pool.close()
    pool.join()
    return expected_C_values

def main():
    # Parameters
    n = 100 # Number of nodes
    num_graphs = 10 # Number of graphs to generate for each p
    p_values = np.linspace(0, 1, 32) # Discretization of [0, 1] for p

    # Compute expected clustering coefficient in parallel
    expected_C_values = expected_clustering_coefficient_parallel(p_values, n,
        num_graphs)

    # Plot expected C as a function of p
    plt.plot(p_values, expected_C_values)
    plt.xlabel('Probability (p)')
    plt.ylabel('Expected Clustering Coefficient (C)')
    plt.title('Expected Clustering Coefficient vs. Probability (p)')
    plt.grid(True)
    plt.show()

if __name__ == "__main__":
    main()
```