BrunoFBessa_5881890_P7_code

July 4, 2021

0.1 SFI5904 - Complex Networks

Project 7: Relationship between topolody and dynamics in complex netwoks First Semester of 2021

Professor: Luciano da Fontoura Costa (luciano@ifsc.usp.br)

Student: Bruno F. Bessa (num. 5881890, bruno.fernandes.oliveira@usp.br) Universidade de São Paulo, São Carlos, Brazil.

A: Obtain scatterplots and respective Pearson correation coefficients between the degree and the activation rate for each node of at least 300 nodes with average degree between 5 and 7. Use one network for each one of the models ER, BA, WS (with 3 probabilities of reconnection) as well as the three geometric models seen during the course.

B: Make the networks used in (A) more and more directed, removing individual connections from them and check the effect on the correlations.

0.2 Code

[354]:
```
import random
import numpy as np
import pandas as pd
from scipy import signal
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_theme()
import networkx as nx
import scipy
import secrets
import math
```

[388]:
```
# Definition of network models

class Point():

    """
    Defines nodes of a graph.
    The Erdos-Renyi network model is built from a set of objects of this class.
    """
```

```python
    def __init__(self, index: str) -> None:
        self.index = index
        self.neighbors = []

    def __repr__(self) -> None:
        return repr(self.index)

def main_component(G: nx.classes.graph.Graph) -> nx.classes.graph.Graph:

    """
    To calculate the distances we need to have only the main connected component
    """
    G = G.to_undirected()
    G.remove_edges_from(nx.selfloop_edges(G))
    Gcc = sorted(nx.connected_components(G), key=len, reverse=True)
    G = G.subgraph(Gcc[0])
    G = nx.convert_node_labels_to_integers(G, first_label=0)

    return G

def plot_graph(G: nx.classes.graph.Graph,
               layout: str,
               title: str,
               with_labels: bool) -> None:

    """
    Plots the generated graph on the console.
    """
    if layout == "circular_layout":
        pos = nx.circular_layout(G)
    else:
        pos = nx.spring_layout(G)
    fig_net = nx.draw(G, pos, node_color='w', node_size=1,␣
 ↪with_labels=with_labels, arrows=True)
    plt.suptitle(title, fontsize=15)
    plt.show(fig_net)

def save_plot_graph(G: nx.classes.graph.Graph,
                    layout: str,
                    title: str,
                    with_labels: bool,
                    file_name: str) -> None:

    """
    Saves generated graph plot on file system.
    """
```

```
    if layout == "circular_layout":
        pos = nx.circular_layout(G)
    else:
        pos = nx.spring_layout(G)
    fig_net = nx.draw(G, pos, node_color='w', node_size=1,␣
 ↪with_labels=with_labels)
    plt.suptitle(title, fontsize=15)
    plt.savefig("images/"+file_name)
    plt.close(fig_net)


def erdos_renyi(N: int,
                p: float,
                plot: bool = True,
                file_name: str = None,
                with_labels=False) -> nx.classes.graph.Graph:

    """
    Defines unidirected connections (i, j) = (j, i) for all pairs of nodes␣
 ↪based on a
    random event with probability less than p, given as a parameter for the␣
 ↪constructor.
    """

    G = nx.Graph()
    nodes = [Point(i) for i in range(N)]
    edges = [(i, j) for i in range (N) for j in range(i) if random.random() < p]

    # Adds edges
    for (i, j) in edges:
        nodes[i].neighbors.append(nodes[j])
        nodes[j].neighbors.append(nodes[i])
    for edge in list(edges):
        G.add_edge(list(edge)[0], list(edge)[1])

    # To calculate the distances we need to have only the main connected␣
 ↪component
    G = main_component(G)

    # Option of visualization. Do not use in the case of series of exeperiments
    title = "Erdos-Renyi network (N={} , p={})".format(N, p)
    if plot:
        plot_graph(G, "spring_layout", title, with_labels)

    if file_name != None:
        save_plot_graph(G, "spring_layout", title, file_name, with_labels)
```

```
    return G


class SpatialPoint():

    """
    Defines a two dimensional point.
    Spacial/Geographical networks will be built from a set of objects of this␣
 ↪class.
    """

    def __init__(self, index: str, box_size: int) -> None:
        self.index = index
        self.x = random.uniform(0, 1) * box_size
        self.y = random.uniform(0, 1) * box_size

    def get_coordinates(self) -> np.array:
        return np.array([(self.x, self.y)])

    def get_distance(self, other) -> float:
        p1_coord = self.get_coordinates()
        p2_coord = other.get_coordinates()

        dist = scipy.spatial.distance.cdist(p1_coord, p2_coord, 'euclidean')
        return dist[0][0]

    def __repr__(self) -> None:
        return repr([(self.x, self.y)])


def spatial_network_voronoi(N: int,
                            box_size: int = 1,
                            plot: bool = True,
                            file_name: str = None,
                            with_labels=False) -> nx.classes.graph.Graph:

    """
    Defines undirected connections (i, j) = (j, i) for all pairs of points if␣
 ↪they share adjacent frontiers
    in the Voronoi tesselation.
    """

    G = nx.Graph()

    spatial_points = [SpatialPoint(i, box_size) for i in range(N)]
    points2d_aux = [point_arr.get_coordinates() for point_arr in spatial_points]
```

```python
    points2d = []
    for point_arr_aux in points2d_aux:
        points2d.append(list(point_arr_aux[0]))

    # Frontiers of Voronoi tesselation:
    vor = scipy.spatial.Voronoi(points2d)

    # Network from the adjascent cells
    edges = vor.ridge_points
    for edge in list(edges):
        G.add_edge(list(edge)[0], list(edge)[1])

    # To calculate the distances we need to have only the main connected component
    G = main_component(G)

    # Option of visualization. Do not use in the case of series of exeperiments 
    title = "Network from Voronoi Tesselaton (N={})".format(N)
    if plot:
        plot_graph(G, "spring_layout", title, with_labels)

    if file_name != None:
        save_plot_graph(G, "spring_layout", title, file_name, with_labels)


    return G


def spatial_network_radius(N: int,
                           box_size: int = 1,
                           radius = 0.3,
                           plot: bool = True,
                           file_name: str = None,
                           with_labels=False) -> nx.classes.graph.Graph:

    """
    Defines a network in which the connections between random nodes are performed if the distance
    from one to the other is less than a radius r, given as a parameter to the constructor.
    """

    G = nx.Graph()

    spatial_points = [SpatialPoint(i, box_size) for i in range(N)]
    points2d_aux = [point_arr.get_coordinates() for point_arr in spatial_points]
```

```
    points2d = []
    for point_arr_aux in points2d_aux:
        points2d.append(list(point_arr_aux[0]))


    # Adds edge between (i, j) if distance (i, j) <= radius
    edges = [(node_i.index, node_j.index) for node_i in spatial_points for␣
 ↪node_j in spatial_points if node_i.get_distance(node_j) > 0 and node_i.
 ↪get_distance(node_j) <= radius]
    for edge in list(edges):
        G.add_edge(list(edge)[0], list(edge)[1])

    # To calculate the distances we need to have only the main connected␣
 ↪component
    G = main_component(G)

    # Option of visualization. Do not use in the case of series of exeperiments
    title = "Geographic newtork (N={} , radius={})".format(N, radius)
    if plot:
        plot_graph(G, "spring_layout", title, with_labels)

    if file_name != None:
        save_plot_graph(G, "spring_layout", title, file_name, with_labels)


    return G

def spatial_network_waxman(N: int,
                            box_size: int = 1,
                            alpha = 0.3,
                            plot: bool = True,
                            file_name: str = None,
                            with_labels=False) -> nx.classes.graph.Graph:
    """
    Defines a network in which the connections between random nodes are␣
 ↪performed due to a random
    event with probability p < exp(-distance/alpha), given as a parameter to␣
 ↪the constructor.

    """
    G = nx.Graph()

    spatial_points = [SpatialPoint(i, box_size) for i in range(N)]
    points2d_aux = [point_arr.get_coordinates() for point_arr in spatial_points]
    points2d = []
    for point_arr_aux in points2d_aux:
        points2d.append(list(point_arr_aux[0]))
```

```
    # Adds edge between (i, j) if p (random) is less than exp^(distance(i,j)/
↪alpha)
    edges = [(node_i.index, node_j.index) for node_i in spatial_points for␣
↪node_j in spatial_points if random.random() < np.exp(-1*node_i.
↪get_distance(node_j)/alpha)]
    for edge in list(edges):
        G.add_edge(list(edge)[0], list(edge)[1])

    # To calculate the distances we need to have only the main connected␣
↪component
    G = main_component(G)

    # Option of visualization. Do not use in the case of series of exeperiments
    title = "Waxman network (N={} , alpha={})".format(N, alpha)
    if plot:
        plot_graph(G, "spring_layout", title, with_labels)

    if file_name != None:
        save_plot_graph(G, "spring_layout", title, file_name, with_labels)


    return G

def regular_reticulated(N: int,
                        plot: bool = True,
                        file_name: str = None,
                        with_labels=False) -> nx.classes.graph.Graph:
    """
    Defines a regular grid network:
     - each node is connected to its horizontal and vertical neighbours
     - the borders of the network are "folded" (connected to each other)

    Choice of representation: given the simmetry of a square matrix of size L,
    all elements can be represented as elements of a list:

     A_ij = List[i+j*L]

     Example (N=10, 3x3 matrix):
     list_edges = [ (0,1), (1,2), (2,0),
                    (3,4), (4,5), (5,3),
                    (6,7), (7,8), (8,6),
                    (0,3), (3,6), (6,0),
                    (1,4), (4,7), (7,1),
                    (2,5), (5,8), (8,2)]
    """
```

```python
    G = nx.Graph()
    L = math.floor(N**(0.5))

    # Horizontal connections
    for i in range(L*L):
        x = i % L
        y = i // L
        if x+1 < L and y < L:
            G.add_edge(i, i+1)
        else:
            G.add_edge(i, L*y)
    # Vertical connections
    for i in range(L*L):
        y = i % L
        x = i // L
        if x+1 < L and y < L:
            G.add_edge(i, i+L)
        else:
            G.add_edge(i, y)

    # To calculate the distances we need to have only the main connected component
    G = main_component(G)

    # Option of visualization. Do not use in the case of series of exeperiments
    title = "Regular grid newtork (N={})".format(N)
    if plot:
        plot_graph(G, "spring_layout", title, with_labels)

    if file_name != None:
        save_plot_graph(G, "spring_layout", title, file_name, with_labels)


    return G

def reconnect_regular(G: nx.classes.graph.Graph,
                      p: float,
                      plot: bool = True,
                      file_name: str = None,
                      with_labels=False) -> nx.classes.graph.Graph:
    """
    Recconectios of a regular network with probability p' < p given as a parameter to the function.
    """
    N = len(G)
    nodes = list(G.nodes())
```

```
    for node in nodes:
        edges_node = list(G.edges(node))
        for edge in edges_node:
            if random.random() < p:
                random_node = secrets.choice(nodes)
                new_edge = (node, random_node)
                # Remove selected edge. Add new edge
                G.remove_edge(list(edge)[0], list(edge)[-1])
                G.add_edge(node, random_node)

    # Option of visualization. Do not use in the case of series of exeperiments
    G = main_component(G)

    # Option of visualization. Do not use in the case of series of exeperiments␣
↪

    title = "Network with {} nodes. Reconnection prob.: {}".format(N, p)
    if plot:
        plot_graph(G, "spring_layout", title, with_labels)

    if file_name != None:
        save_plot_graph(G, "spring_layout", title, file_name, with_labels)

    return G

def reconnect_directed(G: nx.classes.graph.Graph,
                       p: float,
                       plot: bool = True,
                       file_name: str = None,
                       with_labels=False) -> nx.classes.graph.Graph:
    """
    Recconectios of a regular network with probability p' < p given as a␣
↪parameter to the function.
    """
    N = len(G)
    nodes = list(G.nodes())
    for node in nodes:
        edges_node = list(G.edges(node))
        for edge in edges_node:
            if random.random() < p:
                random_node = secrets.choice(nodes)
                new_edge = (node, random_node)
                # Remove selected edge. Add new edge
                G.remove_edge(list(edge)[0], list(edge)[-1])
                G.add_edge(node, random_node)

    # Option of visualization. Do not use in the case of series of exeperiments␣
↪
```

```python
    title = "Network with {} nodes. Reconnection prob.: {}".format(N, p)
    if plot:
        plot_graph(G, "spring_layout", title, with_labels)

    if file_name != None:
        save_plot_graph(G, "spring_layout", title, file_name, with_labels)

    return G

def watts_strogatz(N: int,
                   avg_deg: float,
                   p: float,
                   plot: bool = True,
                   file_name: str = None,
                   with_labels=False) -> nx.classes.graph.Graph:

    """
    Defines Watts-Strogatz network for given p an d average degree.
    """

    k = int(avg_deg)
    G = nx.watts_strogatz_graph(N, k, p, seed=None)

        # Option of visualization. Do not use in the case of series of␣
 ↪exeperiments
    title = "Network with {} nodes. Reconnection prob.: {}".format(N, p)
    if plot:
        plot_graph(G, "spring_layout", title, with_labels)

    if file_name != None:
        save_plot_graph(G, "spring_layout", title, file_name, with_labels)

    return G

def random_graph(N_random:int = 10) -> nx.classes.graph.Graph:

    """
    Defines a graph with N nodes and random connections between them.
    """

    G = nx.Graph()
    for node in range(N_random+1):
        G.add_node(node)

    nodes = list(G.nodes())
    for node in nodes:
        random_node = secrets.choice(nodes)
```

```python
        G.add_edge(node, random_node)

    return G

def ba_graph(N: int,
             m: int = 3,
             plot: bool = True,
             file_name: str = None,
             with_labels=False) -> nx.classes.graph.Graph:

    """
    Defines the Barbási-Albert network model with preferencial attachment.
    """

    G = random_graph()
    dict_degree = dict(G.degree())
    list_k_nodes = []
    for k_value, k_freq in dict_degree.items():
        for freq in range(k_freq):
            list_k_nodes.append(k_value)

    for node in range(len(G), N-1):
        for conn in range(m):
            random_node = secrets.choice(list_k_nodes)
            G.add_edge(node, random_node)

    # Option of visualization. Do not use in the case of series of exeperiments
    G = main_component(G)

    # Option of visualization. Do not use in the case of series of exeperiments
    title = "Barbási-Albert newtork (N={} , m={})".format(N, m)
    if plot:
        plot_graph(G, "spring_layout", title, with_labels)

    if file_name != None:
        save_plot_graph(G, "spring_layout", title, file_name, with_labels)

    return G
```

[450]:
```python
def correlation_plot(x: list,
                     y: list,
                     x_label: str = "x",
                     y_label: str = "y",
                     title: str = None,
                     file_name: str = None) -> None:
    """
    Dispersion graph for lists x and y.
```

```
    """

    pearson_corr = np.corrcoef(x, y)[0,1]
    spearman_corr, spearman_pval = scipy.stats.spearmanr(x, y)

    fig, ax = plt.subplots()
    ax.scatter(x, y)
    ax.set_xlabel(x_label)
    ax.set_ylabel(y_label)
    if title == None:
        ax.set_title("Dispersion for {} and {} Pearson Coef.: {:.2f}".
↪format(x_label, y_label, pearson_corr, fontsize=15))
    else:
        pc_title = " Pearson Coef.: {:.2f}".format(pearson_corr)
        title = title + pc_title
        ax.set_title(title.format(fontsize=15))

    plt.show(True)
    if file_name != None:
        fig.savefig("images/"+file_name)

def steering_coefficient(G: nx.classes.graph.Graph,
                         x_label: str = "Degree",
                         y_label: str = "Activation Rate",
                         title: str = None,
                         file_name: str = None) -> None:

    """
    Automation for steering coefficient plots.
    """

    x = list(dict(nx.degree(G)).values())
    y = items_frequency(random_walk(G))

    delta = len(x) - len(y)
    if delta > 0:
        for element in range(delta):
            x.pop()

    correlation_plot(x, y, x_label, y_label, title, file_name)

def items_frequency(x: list) -> list:

    """
    Calculates frequency of elements in a list
    """
```

```
    dict_freq = {}
    for item in x:
        if (item in dict_freq):
            dict_freq[item] += 1
        else:
            dict_freq[item] = 1

    list_values = list(dict(sorted(dict_freq.items())).values())
    list_values = [value/len(x) for value in list_values]

    return list_values

def random_walk(G: nx.classes.graph.Graph) -> list:

    """
    Performs a random walk on the graph and returns a list of visited nodes.
    """
    walk_length = 50000 * len(G)
    graph_nodes = list(nx.nodes(G))
    current_node = random.choice(graph_nodes)
    visited_nodes = []

    for i in range(walk_length):
        neighbours = list(G.adj[current_node])
        current_node = random.choice(neighbours)
        visited_nodes.append(current_node)

    return visited_nodes
```

0.3 Results

Code for the results is below.

[459]:
```
N = 300
ER = erdos_renyi(N, p=0.02, plot=False)             # <k>=6.14
BA = ba_graph(N, 4, plot=False)                     # <k>=6.71
VO = spatial_network_voronoi(N, 1, plot=False)      # <k>=5.90
RA = spatial_network_radius(N, 1, 0.079, plot=False) # <k>=6.20
WX = spatial_network_waxman(N, 1, 0.048, plot=False) # <k>=6.62
WS_1 = watts_strogatz(N, 6, 0.1, plot=False)        # <k>=6.00
WS_2 = watts_strogatz(N, 6, 0.2, plot=False)        # <k>=6.00
WS_3 = watts_strogatz(N, 6, 1.0, plot=False)        # <k>=6.00

ER = ER.to_directed()
BA = BA.to_directed()
VO = VO.to_directed()
```

```
RA = RA.to_directed()
WX = WX.to_directed()
WS_1 = WS_1.to_directed()
WS_2 = WS_2.to_directed()
WS_3 = WS_3.to_directed()
```

[ ]:

```
steering_coefficient(ER, title="Erdos-Renyi, reciprocity={:.1f}".format(nx.
 ↪reciprocity(ER)), file_name="ER_r1.jpg")
steering_coefficient(BA, title="Barabás-Albert, reciprocity={:.1f}".format(nx.
 ↪reciprocity(BA)), file_name="BA_r1.jpg")
steering_coefficient(VO, title="Voronoi Tesselation, reciprocity={:.1f}".
 ↪format(nx.reciprocity(VO)), file_name="VO_r1.jpg")
steering_coefficient(RA, title="Spatial (radius=0.079), reciprocity={:.1f}".
 ↪format(nx.reciprocity(RA)), file_name="RA_r1.jpg")
steering_coefficient(WX, title="Waxman (alpha=0.048), reciprocity={:.1f}".
 ↪format(nx.reciprocity(WX)), file_name="WX_r1.jpg")
steering_coefficient(WS_1, title="Watts-Strogatz (prob=0.1), reciprocity={:.
 ↪1f}".format(nx.reciprocity(WS_1)), file_name="WS_1_r1.jpg")
steering_coefficient(WS_2, title="Watts-Strogatz (prob=0.2), reciprocity={:.
 ↪1f}".format(nx.reciprocity(WS_2)), file_name="WS_2_r1.jpg")
steering_coefficient(WS_3, title="Watts-Strogatz (prob=1.0), reciprocity={:.
 ↪1f}".format(nx.reciprocity(WS_3)), file_name="WS_3_r1.jpg")
```