BrunoFBessa_5881890_P9_code

July 5, 2021

0.1 SFI5904 - Complex Networks

Project 9: Communities 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.

Generate networks with 2 or 3 communities, using libraries and also the software for synthetic geographic networks with initial nodes positions following modular distribution.

Appy at least 2 methods for community detection, including accessibility and compare the results qualitatively using visualizations of the networks with communities identified with different colors.

0.2 Code

[111]:
```
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
from community import community_louvain
from IPython import display
```

[77]:
```
# 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="spring_layout",
               title: str="Graph",
               with_labels: bool=False) -> 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 plot_graph_borders(G: nx.classes.graph.Graph,
               layout: str,
               title: str,
               with_labels: bool) -> None:

    """
    Plots the generated graph on the console with borders.
    """
    if layout == "circular_layout":
```

```python
        pos = nx.circular_layout(G)
    else:
        pos = nx.spring_layout(G)

    graph_access = graph_accessibility(G, h=2)
    threshold = np.percentile(graph_access, 50)
    graph_access = [1 if x >= threshold else 0 for x in graph_access]

    fig_net = nx.draw(G,
                      pos,
                      node_color=graph_access,
                      cmap=plt.get_cmap("coolwarm"),
                      node_size=100,
                      with_labels=with_labels)
    plt.suptitle(title, fontsize=15)
    plt.show(fig_net)


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

    """
    Plots the generated graph on the console with borders.
    """
    if layout == "circular_layout":
        pos = nx.circular_layout(G)
    else:
        pos = nx.spring_layout(G)

    fig_net = nx.draw(G,
                      pos,
                      node_color=values,
                      cmap=plt.get_cmap("coolwarm"),
                      node_size=100,
                      with_labels=with_labels)
    plt.suptitle(title, fontsize=15)
    plt.show(fig_net)

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

```
    """
    Plots the generated graph on the console with borders.
    """
    if layout == "circular_layout":
        pos = nx.circular_layout(G)
    else:
        pos = nx.spring_layout(G)

    graph_access = graph_accessibility(G, h=2)
    threshold = np.percentile(graph_access, 50)
    graph_access = [1 if x >= threshold else 0 for x in graph_access]

    fig_net = nx.draw(G,
                      pos,
                      node_color=graph_access,
                      cmap=plt.get_cmap("coolwarm"),
                      node_size=100,
                      with_labels=with_labels)
    plt.suptitle(title, fontsize=15)
    plt.savefig("images/"+file_name)
    plt.close(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)


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)])


class SpatialPointConfined():

    """
    Defines a two dimensional point, confined in a region center_x, center_y.
    Spacial/Geographical networks will be built from a set of objects of this␣
↪class.
    """

    def __init__(self, index: str, box_size: int, center_x: float, center_y:␣
↪float, conf_size: float) -> None:
        self.index = index
        self.x = random.uniform(center_x - conf_size/2, center_x + conf_size/2)␣
↪* box_size
        self.y = random.uniform(center_y - conf_size/2, center_y + conf_size/2)␣
↪* 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()
```

```python
        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_radius_community(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 = []

    spatial_points_community_a = [SpatialPointConfined(i, 1, 0.2, 0.2, 0.2) for␣
↪i in range(0, int(N/2))]
    spatial_points_community_b = [SpatialPointConfined(i, 1, 0.8, 0.8, 0.2) for␣
↪i in range(int(N/2), N)]

    spatial_points = spatial_points_community_a + spatial_points_community_b

    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_community(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 = []

    spatial_points_community_a = [SpatialPointConfined(i, 1, 0.2, 0.2, 0.2) for␣
 ↪i in range(0, int(N/2))]
    spatial_points_community_b = [SpatialPointConfined(i, 1, 0.8, 0.8, 0.2) for␣
 ↪i in range(int(N/2), N)]

    spatial_points = spatial_points_community_a + spatial_points_community_b


    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
```

[105]:

```
def node_accessibility(G: nx.classes.graph.Graph,
                       current_node: int=0,
                       visited_nodes: list=[],
                       h: int=1) -> float:

    """
    Defines the accessibility for one node of a network
    """

    neighbours = list(G.adj[current_node])
    H_entropy = 0
    if G.degree[current_node] > 1:
        visited_nodes.append(current_node)
        for neighbour in neighbours:
            p = 1/len(neighbours)
            H_entropy += -1 * p * math.log(p, 2)
        if h <= 1:
            return H_entropy
        else:
            for distance in range(0, h-1):
                if neighbour not in visited_nodes:
                    H_entropy += node_accessibility(G,
                                                    current_node=neighbour,
                                                    visited_nodes=visited_nodes,
                                                    h=distance)
    accessibiity = math.exp(H_entropy)
```

```
    return accessibiity

def graph_accessibility(G: nx.classes.graph.Graph, h) -> list:

    """
    Returns a list of respective accessibilities for the nodes of a network.
    """

    acc_list = []
    for node in list(G.nodes()):
        acc_node = node_accessibility(G, current_node=node, visited_nodes=[],␣
 ↪h=h)
        acc_list.append(acc_node)

    return acc_list
```

0.3 Results

Code for results is below:

```
[ ]: WX = spatial_network_waxman_community(N=300, box_size=1, alpha=0.12)

# Detect communities using Louvain Method
values_louvain_WX = list(community_louvain.best_partition(WX).values())
plot_graph_communities(WX, "spring_layout", "Louvain Community Detection for␣
 ↪WX", False, values=values_louvain_WX)

# Detect communities using Accessibility Metric
plot_graph_borders(WX, "spring_layout", "Accessibility Detection for WX", False)

SC = nx.random_partition_graph([100, 100, 100], 0.75, 0.020)
plot_graph(G, title="Synthetic Network with 3 communities")

# Detect communities using Louvain Method
values_louvain_SC = list(community_louvain.best_partition(SC).values())
plot_graph_communities(SC, "spring_layout", "Louvain Community Detection for␣
 ↪Synth. Communities", False, values=values_louvain_SC)
```