Redes complejas

Práctica 3:

Introducción a NetworkX para medidas y métricas

MUSI - Máster Universitario en Sistemas Inteligentes

López Muñoz, Iván - 43221427B

Índice de contenidos

Introducción

En el presente documento se van a computar diferentes conceptos del capítulo 7 de la asignatura, que van a servir como introducción y preparación para la realización de la práctica final.

Para ello, se va a utilizar una base de datos ya conocida, los Quakers, un conjunto de personas que disentían de la Iglesia oficial de Inglaterra y promovían una amplia tolerancia religiosa. Aunque el motivo de agrupación de estas personas nos sea indiferente, dada la efectividad de sus redes, los datos utilizados en esta introducción son una lista de nombres y relaciones entre los primeros Quakers del siglo XVII.

Las redes son una estructura de datos fundamental que se utiliza para modelar una amplia gama de sistemas y fenómenos del mundo real. La capacidad de analizar y extraer información de las redes se ha vuelto cada vez más importante para comprender las complejas interconexiones que subyacen a muchos sistemas tanto naturales como diseñados.

Una herramienta muy útil para realizar análisis de redes es NetworkX, una biblioteca de Python que proporciona un conjunto de herramientas para el estudio de redes complejas. NetworkX ofrece una amplia gama de algoritmos gráficos, estructuras de datos y capacidades de visualización que permiten obtener una comprensión más profunda de los datos de la red.

En primer lugar, se va a seguir un tutorial de guía (https://shorturl.at/oqxEQ) para preparar la red con NetworkX en Python y al acabar se añadirán a este tutorial conceptos relacionados con medidas y métricas que hemos visto en clase en el capítulo 7.

Creación de la red

Esta sección del código lee un archivo CSV llamado 'quakers_nodelist.csv' y otro llamado 'quakers_edgelist.csv' utilizando el módulo csv de Python y se almacenan en la lista ‘nodes’ y ‘edges’. Cada fila en ‘nodes’ representa un nodo en la red, con varias propiedades, y ‘edges’ almacena los datos de las aristas.

```
# Read in the nodelist file
with open('quakers_nodelist.csv', 'r') as nodecsv:
    nodereader = csv.reader(nodecsv)
    nodes = [n for n in nodereader][1:]

# Get a list of just the node names (the first item in each row)
node_names = [n[0] for n in nodes]

# Read in the edgelist file
with open('quakers_edgelist.csv', 'r') as edgecsv:
    edgereader = csv.reader(edgecsv)
    edges = [tuple(e) for e in edgereader][1:]

G = nx.Graph() # Initialize a Graph object
G.add_nodes_from(node_names) # Add nodes to the Graph
G.add_edges_from(edges) # Add edges to the Graph
print(G) # Print information about the Graph
```

Se inicializa un objeto de grafo de NetworkX ‘G’ y se imprime información sobre el grafo, que incluye el número de nodos y aristas.

```
Graph with 119 nodes and 174 edges
```

A continuación se crean 5 diccionarios vacíos que se utilizarán para almacenar información específica relacionada con los nodos de la red. Cada diccionario de atributos se asigna a un nombre de atributo específico, como 'historical_significance', 'gender', 'birth_year', 'death_year', y 'sdfb_id’.

```
# Create an empty dictionary for each attribute
hist_sig_dict = {}
gender_dict = {}
birth_dict = {}
death_dict = {}
id_dict = {}

for node in nodes: # Loop through the list of nodes, one row at a time
    hist_sig_dict[node[0]] = node[1] # Access the correct item, add it
    gender_dict[node[0]] = node[2]
    birth_dict[node[0]] = node[3]
    death_dict[node[0]] = node[4]
    id_dict[node[0]] = node[5]

# Add each dictionary as a node attribute to the Graph object
nx.set_node_attributes(G, hist_sig_dict, 'historical_significance')
nx.set_node_attributes(G, gender_dict, 'gender')
nx.set_node_attributes(G, birth_dict, 'birth_year')
nx.set_node_attributes(G, death_dict, 'death_year')
nx.set_node_attributes(G, id_dict, 'sdfb_id')
```

Por último, se utiliza la función nx.connected_components(G) para encontrar los grupos de nodos conectados en el grafo G. Esto es útil en redes que pueden estar fragmentadas en múltiples componentes aislados, como es el caso del grafo de los Quakers

Después, se utiliza la función max() para encontrar el componente más grande basada en la cantidad de nodos que contiene. Se crea un "subgrafo" que contiene sólo el componente más grande del grafo original G para realizar análisis específicos.

```
components = nx.connected_components(G)
largest_component = max(components, key=len)

# Create a "subgraph" of just the largest comp
# Then calculate the diameter of the subgraph,
subgraph = G.subgraph(largest_component)
diameter = nx.diameter(subgraph)
```

A partir de aquí vamos a computar los diferentes conceptos del capítulo 7 de medidas y métricas que no se encuentran en la página de la introducción.

PageRank Centrality

En este fragmento del código, se calcula la centralidad de PageRank para los nodos en el grafo G y sirve para medir la importancia de los nodos en una red. Una vez calculada la PageRank centrality, se asigna como un atributo de los nodos con nx.set_node_attributes.

```
# PageRank centrality

pagerank_centrality = nx.pagerank(G, alpha=0.85)  # You can adjust the alpha value if needed
# Assign PageRank centrality as a node attribute
nx.set_node_attributes(G, pagerank_centrality, 'pagerank_centrality')
# Sort the nodes by PageRank centrality
sorted_pagerank = sorted(pagerank_centrality.items(), key=itemgetter(1), reverse=True)
```

Finalmente, los nodos se ordenan en orden descendente.

```
------------------------------------------------------------
Top 20 nodes by PageRank centrality:

Node George Fox: PageRank Centrality = 0.04607655014262055
Node William Penn: PageRank Centrality = 0.043189785507042895
Node James Nayler: PageRank Centrality = 0.03269788744927486
Node George Whitehead: PageRank Centrality = 0.02929823716820768
Node Margaret Fell: PageRank Centrality = 0.029052242449373974
Node Benjamin Furly: PageRank Centrality = 0.02179017735640094
```

Closeness Centrality

En este fragmento de código, se calcula la Closeness Centrality para los nodos en el subgrafo de G que mide la cercanía de un nodo a otros nodos en la red.

```
# Closeness Centrality

closeness_centrality = nx.closeness_centrality(subgraph)
# Assign Closeness Centrality as a node attribute
nx.set_node_attributes(subgraph, closeness_centrality, 'closeness_centrality')
# Sort the nodes by Closeness Centrality
sorted_closeness = sorted(closeness_centrality.items(), key=itemgetter(1), reverse=True)
```

Finalmente, los nodos se ordenan en orden descendente:

```
------------------------------------------------------------
Top 20 nodes by Closeness Centrality:

Node George Fox: Closeness Centrality = 0.48717948717948717
Node William Penn: Closeness Centrality = 0.4634146341463415
Node George Whitehead: Closeness Centrality = 0.4398148148148148
Node Margaret Fell: Closeness Centrality = 0.42035398230088494
Node James Nayler: Closeness Centrality = 0.41125541125541126
Node George Keith: Closeness Centrality = 0.40425531914893614
```

Cliques

En este fragmento de código, se calcula el número de cliques máximos en el subgrafo de G. Los cliques son conjuntos de nodos en los que cada miembro está conectado por un vértice entre sí.

```
# Clique Number (Maximal Cliques)
cliques = list(nx.find_cliques(subgraph))
clique_number = max(len(clique) for clique in cliques)
print(f"Clique Number (Maximal Cliques): {clique_number}")
```

El número máximo de cliques en el subgrafo de G es:

```
-----------------------------------
Clique Number (Maximal Cliques): 4
-----------------------------------
```

K-Core numbers

En este fragmento de código, se calcula el número del K-Core para cada nodo en el subgrafo y representa el nivel al que un nodo se encuentra en un subconjunto densamente conectado de la red.

```
# K-Core Number
k_core_number = nx.core_number(subgraph)
print("K-Core Numbers:",'\n')
for node, k_core in k_core_number.items():
    print(f"Node {node}: K-Core Number = {k_core}")
```

Podemos ver una lista de todos los nodos con su K-Core correspondiente:

```
-----------------------------------------
K-Core Numbers:

Node Joseph Wyeth: K-Core Number = 1
Node James Logan: K-Core Number = 2
Node Dorcas Erbery: K-Core Number = 1
Node William Mucklow: K-Core Number = 1
Node Thomas Salthouse: K-Core Number = 2
Node William Dewsbury: K-Core Number = 3
```

Clustering coefficient

En este fragmento de código, se calcula el coeficiente de agrupamiento promedio para el subgrafo de la red y mide la tendencia de los nodos en una red a formar grupos o cliques.

```
# Clustering Coefficient
clustering_coefficient = nx.average_clustering(subgraph)
print(f"Average Clustering Coefficient: {clustering_coefficient}")
```

El promedio del subgrafo es:

```
--------------------------------------------------
Average Clustering Coefficient: 0.28125381392844623
--------------------------------------------------
```

Redundancy

En este fragmento de código, se calcula la redundancia para cada nodo del subgrafo de la red. La redundancia se refiere a la cantidad promedio de conexiones desde los vecinos de un nodo a otros vecinos del mismo nodo. Es decir, mide cuántos amigos en común tienen los amigos de un nodo en relación con ese nodo en particular.

```python
# Redundancy
# Initialize a dictionary to store the redundancy values for each node
redundancy_dict = {}

# Calculate redundancy for each node
for node in subgraph.nodes():
    neighbors = list(subgraph.neighbors(node))
    total_redundancy = 0
    for neighbor in neighbors:
        neighbor_neighbors = list(subgraph.neighbors(neighbor))
        neighbor_neighbors.remove(node)  # Exclude the current node itself
        total_redundancy += len(set(neighbor_neighbors) & set(neighbors))
    redundancy = total_redundancy / len(neighbors) if len(neighbors) > 0 else 0
    redundancy_dict[node] = redundancy

# Print the redundancy values for each node
print("Redundancy Values:",'\n')
for node, redundancy in redundancy_dict.items():
    print(f"Node {node}: Redundancy = {redundancy}")
```

Podemos ver la redundancia de cada nodo de la red del subgrafo:

```
--------------------------------------------------
Redundancy Values:

Node Joseph Wyeth: Redundancy = 0.0
Node James Logan: Redundancy = 1.0
Node Dorcas Erbery: Redundancy = 0.0
Node William Mucklow: Redundancy = 0.0
Node Thomas Salthouse: Redundancy = 1.0
Node William Dewsbury: Redundancy = 2.0
Node John Audland: Redundancy = 1.0
Node Richard Claridge: Redundancy = 0.0
Node William Bradford: Redundancy = 1.3333333333333333
```

Reciprocity

La reciprocidad mide en qué medida las conexiones en la red son bidireccionales o recíprocas. Es decir, cuántas de las conexiones de ida tienen una conexión de regreso en la red.

```python
# Reciprocity

reciprocity = nx.reciprocity(subgraph)
print(f"Reciprocity: {reciprocity}")
```

En este caso no hay ningún bucle:

```
--------------------
Reciprocity: 0.0
--------------------
```

Structural equivalences

La equivalencia estructural se refiere a la similitud en la forma en que los nodos están conectados a sus vecinos.

```python
# Similarity - structural equivalence (simplest version)
structural_equivalence = {}
for node in G.nodes():
    neighbors = set(G.neighbors(node))
    equivalent_nodes = [n for n in G.nodes() if set(G.neighbors(n)) == neighbors and n != node]
    structural_equivalence[node] = equivalent_nodes

# Print nodes with structural equivalences
print("Nodes with structural equivalences:", '\n')
for node, equivalent_nodes in structural_equivalence.items():
    if equivalent_nodes:
        print(f"Node {node} is structurally equivalent to nodes: {equivalent_nodes}")
```

Los nodos que tienen alguna equivalencia estructural son los siguientes:

```
Nodes with structural equivalences:

Node William Mucklow is structurally equivalent to nodes: ['Ellis Hookes', 'Elizabeth Hooten', 'William Coddington', 'Leonard
Fell']
Node Thomas Salthouse is structurally equivalent to nodes: ['William Mead']
Node Isabel Yeamans is structurally equivalent to nodes: ['Isaac Norris', 'Edward Haistwell', 'Thomas Story']
Node George Fox the younger is structurally equivalent to nodes: ['Thomas Lower']
Node Thomas Lower is structurally equivalent to nodes: ['George Fox the younger']
Node William Mead is structurally equivalent to nodes: ['Thomas Salthouse']
Node Ellis Hookes is structurally equivalent to nodes: ['William Mucklow', 'Elizabeth Hooten', 'William Coddington', 'Leonard
Fell']
Node Elizabeth Hooten is structurally equivalent to nodes: ['William Mucklow', 'Ellis Hookes', 'William Coddington', 'Leonard
Fell']
Node John Whiting is structurally equivalent to nodes: ['Thomas Taylor']
Node William Coddington is structurally equivalent to nodes: ['William Mucklow', 'Ellis Hookes', 'Elizabeth Hooten', 'Leonard
Fell']
Node Thomas Taylor is structurally equivalent to nodes: ['John Whiting']
Node Leonard Fell is structurally equivalent to nodes: ['William Mucklow', 'Ellis Hookes', 'Elizabeth Hooten', 'William
Coddington']
Node Isaac Norris is structurally equivalent to nodes: ['Isabel Yeamans', 'Edward Haistwell', 'Thomas Story']
Node Edward Haistwell is structurally equivalent to nodes: ['Isabel Yeamans', 'Isaac Norris', 'Thomas Story']
Node Thomas Story is structurally equivalent to nodes: ['Isabel Yeamans', 'Isaac Norris', 'Edward Haistwell']
------------------------------------------------------------
```

Conclusiones

Visto que en la red inicial había nodos aislados, se ha tenido que realizar las diferentes medidas y métricas respecto al subgrafo con mayor número de componentes.

Se han obtenido, viendo los diferentes resultados como algunos nodos son mucho más importantes que otros. Es el caso del nodo ‘George Fox’ o ‘William Penn’ que destacan tanto por el número de grados como en la betweenness, PageRank o closeness centrality

NetworkX es una herramienta fundamental para analizar redes en Python debido a su facilidad de uso permitiendo obtener información valiosa sobre la estructura y dinámica de las redes sin hacer uso de las complicadas y confusas ecuaciones matemáticas de todos los conceptos