NetworkX

Course Instructor: Dr.V.S.Felix Enigo

Gray Cardinal Nodes

- A Person X did not have many strong ties. He was a man of few words, yet he could make an offer you can’t refuse

- Positions of X can have an immense power; by knowing well-connected people, they can exploit this information and information asymmetry to further their own plans, while staying largely in the shadows

Eigen Vector Centrality

- Eigenvector Centrality is **an algorithm that measure of the influence of a node in a network**

- A high eigenvector score means that a node is connected to many nodes who themselves have high scores

- Relative scores are assigned to all nodes in the network based on the connections to high-scoring nodes which contributes more to the score of the node in question than equal connections to low-scoring nodes

- Eigenvector centrality differs from in-degree centrality: **a node receiving many links does not necessarily have a high eigenvector centrality**

Eigen Vector Centrality Algorithm

1. Start by assigning a centrality score of 1 to all nodes (`v_i` = 1 for all *i* in the network).
2. Recompute the scores of each node as a weighted sum of centralities of all nodes in a node's neighborhood:

$$v_i = \sum_{j \in N} x_{i,j} * v_j$$

3. Normalize **v** by dividing each value by the largest value.
4. Repeat steps 2 and 3 until the values of **v** stop changing.

NetworkX provides an implementation of eigenvector centrality:

```
>>> eigenvector_centrality(g)
```

Cond..

- Eigen vector centrality is an iterative algorithm, where for each node one must iterate through its neighbors to compute the weighted degree
- Every iteration of the algorithm *O(nodes*average_degree)* operations
- Requires large no. of iterations, not realistic to compute on very

  large networks.

Example

- >>> eigenvector_centrality(g)

LiveJournal Russian Network

- 'valerois' 0.250535826
- 'bagira' 0.222453253
- 'azbukivedi' 0.215904343
- 'kpoxa_e' 0.207785523
- 'boctok' 0.164058289
- 'yelya' 0.160704177
- 'mamaracha' 0.159064962
- 'karial' 0.15127215
- 'angerona' 0.146023845
- 'marinka' 0.127491521

Inference:

- mamaracha and valerois have low degree, but high betweenness, and high eigenvector centrality. This largely means that they are in a position called *Boundary Spanners*

Google - PAGE RANK ALGORITHM

- Google use PageRank algorithm to rank and display pages

- Instead of centrality “radiating forward” from a node and being one of the node’s properties, PageRank centrality is determined through incoming links

- PageRank was originally developed for indexing web pages, but can be applied to social networks as well, as long as they are directed graphs

- Example, a retweet network on Twitter is an excellent candidate.

Simplified Page Rank Algorithm

- Assume four web pages: A, B, C, and D
- Ignoring Links from a page to itself, or multiple outbound links from one single page to another single page, are ignored
- PageRank is initialized to the same value for all pages
- Assume a probability distribution between 0 and 1
- Hence the initial value for each page in this example is 0.25
- The PageRank transferred from a given page to the targets of its outbound links upon the next iteration is divided equally among all outbound links.
- If the only links in the system were from pages B, C, D -> A
- Each link would transfer 0.25 PageRank to A upon the next iteration, for a total of 0.75.
- PR(A) = PR(B) + PR (C) + PR (D) = 0.25+0.25+0.25 = 0.75

Cond…

- Suppose instead that page B has link to C & A (B -> C and B-> A)
- Upon the first iteration, page B would transfer half of its existing value, or 0.125, to page A and the other half, or 0.125, to page C
- Suppose page D had links to all three pages
- It would transfer one-third of its existing value, or approximately 0.083, to A
- At the completion of this iteration, page A will have a PageRank of approximately 0.458.
- PR(A) = PR(B)+ PR(C) + PR(D) = 0.125+0.25+ 0.083 = 0.458

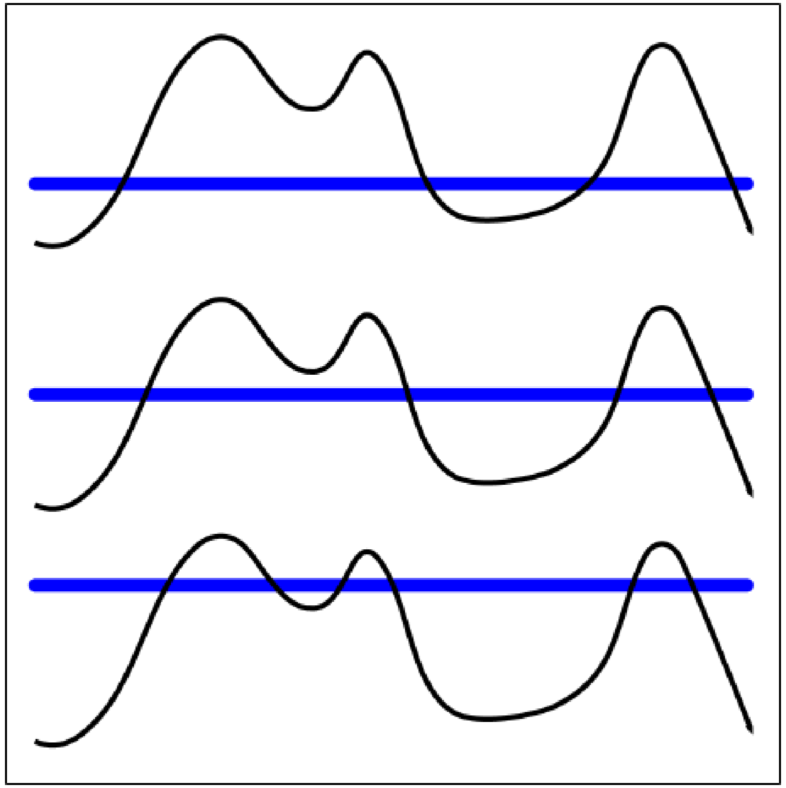
Components and Subgraphs

- A *subgraph* is a subset of the nodes of a network, and all of the edges linking these nodes
- Any group of nodes can form a subgraph
- Component subgraphs (or simply *components*) are portions of the network that are disconnected from each other
- >>> e=nx.read_pajek("egypt_retweets.net")
- >>> len(e)
- 25178
- >>> len(nx.connected_components(e))
- 3122

Cond…

>>> [len(c) for c in net.connected_component(g) if len(c) > 10]

- [17762, 64, 16, 16, 14, 13, 11, 11]



- Island in the Net

Islands in the Net

- Consider an island with a complex terrain, height of each point on the terrain is defined by the value of a node
- Value of the node could be degree centrality or edge (e.g., number of retweets)
- Now slowly rising the water level leaves the portions of landscape underwater
- When valleys of island are flooded, island splits into smaller islands revealing the highest peaks
- Increasing water level further, leaves peak smaller and subsequently disappears the peak

Cond…

- Method needs to be applied judiciously to reveal meaningful results
- In terms of networks, giant component gets split up into smaller components
- Areas with the strongest amount of activity E.g. retweeting in Egyptian network (subcores) become their own components that can be analyzed separately

NetworkX Implementation

- For island method, first implement a function to virtually raise the water level
- The function applies a threshold ("water level"), letting all edges above a certain value through, and removing all others

```
def trim_edges(g, weight=1):
        g2=net.Graph()
        for f, to, edata in g.edges(data=True):
                if edata['weight'] > weight:
                        g2.add_edge(f,to,edata)
        return g2
```

Cond…

```
def island_method(g, iterations=5):
    weights= [edata['weight'] for f,to,edata in g.edges(data=True)]

    mn=int(min(weights))
    mx=int(max(weights))
    #compute the size of the step, so we get a reasonable step in iterations
    step=int((mx-mn)/iterations)

    return [[threshold, trim_edges(g, threshold)] for threshold in range(mn,mx,step)]
```

The above code computes evenly spaced thresholds and produce a list of networks at each water level

Code Cond…

```
>>> cc=net.connected_component_subgraphs(e)[0]
>>> islands=island_method(cc)
>>> for i in islands:
...  # print the threshold level, size of the graph, and number of connected components
...  print i[0], len(i[1]), len(net.connected_component_subgraphs(i[1]))
```

```
1 12360 314
62 27 11
123 8 3
184 5 2
245 5 2
```

```python
import networkx as nx

# Read the Pajek file
e = nx.read_pajek("egypt_retweets.net")

# Calculate the number of nodes in the graph
num_nodes = len(e)

# Get the connected component subgraphs
connected_components = list(nx.connected_component_subgraphs(e))

# Calculate the number of connected components with more than 10 nodes
component_sizes = [len(c) for c in connected_components if len(c) > 10]

# Define a function to trim edges based on weight
def trim_edges(g, weight=1):
    g2 = nx.Graph()
    for f, to, edata in g.edges(data=True):
        if edata['weight'] > weight:
            g2.add_edge(f, to, edata)
    return g2

# Define the island method to find meaningful components
def island_method(g, iterations=5):
    weights = [edata['weight'] for f, to, edata in g.edges(data=True)]
    mn = int(min(weights))
    mx = int(max(weights))
    step = int((mx - mn) / iterations)
    return [[threshold, trim_edges(g, threshold)] for threshold in range(mn, mx, step)]

# Select the first connected component subgraph
cc = connected_components[0]

# Apply the island method
islands = island_method(cc)

# Iterate through the islands and print information
for i in islands:
    threshold_level = i[0]
    graph_size = len(i[1])
    num_connected_components = len(nx.connected_component_subgraphs(i[1]))
    print(threshold_level, graph_size, num_connected_components)
```