Course Project Report: Impression Network Analysis

Jyoti
2022CSB1319

April 25, 2024

1 Experiment 1: Identifying Top Leaders

To identify the top leader in the impression network, I employed a random walk algorithm with teleportation. The Python code snippet used for this experiment is provided below:

Below is the Python code for assigning ranks to nodes in a graph based on a random walk algorithm, along with explanations and comments:

```
import pandas as pd
import networkx as nx
import numpy as np
import random
import matplotlib.pyplot as plt

# Read data from Excel sheet
df = pd.read_excel('impressions.xlsx')

# Drop duplicate rows from the DataFrame
df.drop_duplicates(inplace=True)

# Create an empty directed graph
G = nx.DiGraph()

# Add nodes from the first column
nodes = df.iloc[:, 0]  # Selecting the first column
unique_nodes = nodes.unique()
G.add_nodes_from(unique_nodes)  # Add unique nodes


# Add edges from each row
for _, row in df.iterrows():
    source_node = row[0]  # Source node is in the first column
    for target_node in row[1:]:
        if pd.notnull(target_node):  # Check if the cell is not empty
            G.add_edge(source_node, target_node)

nx.draw(G, with_labels = True)
plt.show()                          # Display the final graph G using matplotlib


def rank_assign(G):
  points = 10000000   # Total initial points/coins to be distributed among the no
  points_box = {}
  for node in G.nodes():
          points_box[node] = 0   # Initializing the points of all the nodes in G t

  random_node = random.choice(list(G.nodes()))
  while(points >= 0 ):
          adjacency_list = list(G.neighbors(random_node))
          if adjacency_list:
              neighbour = random.choice(adjacency_list)
          else:
              neighbour = random.choice(list(set(G.nodes()) - {random_node}))
# If the current node does not have a neighbour then choose any other node at ra
          points_box[neighbour]=points_box[neighbour]+1
# increment the points of the randomly chosen neighbour
          points=points-1
# decrement the Total points by 1
          random_node = neighbour
# make the neighbour the new random_node
  rank_1 = max(points_box, key=points_box.get)
# The top leader is the one with the maximum points after uniformity has been ac

  print("Top leader :",rank_1)
  return points_box

rank_assign(G)
```

The random walk algorithm successfully identified the top leader to be **2022CSB1091**.
The result can be verified by running the code multiple times to get the same result.

1.1 Libraries Used

The Python code utilizes the following libraries:

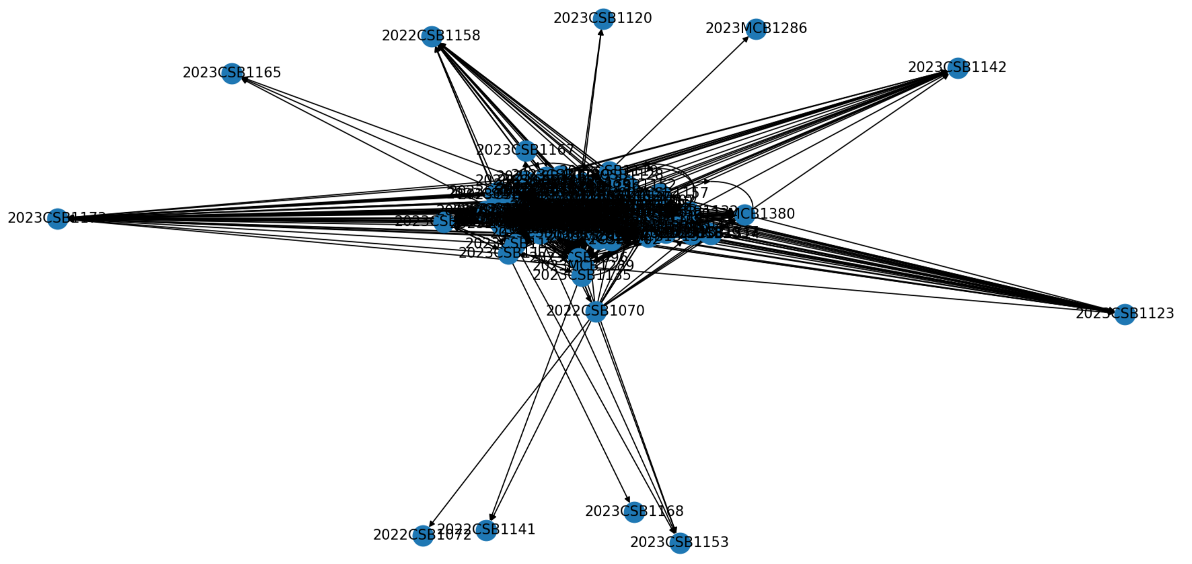


Figure 1: Graph of the impressions data

2 Experiment 2: Link Recommendation

For link recommendation,the matrix method using Linear Algebra as discussed in class is used. The details of this method and its implementation can be found in the accompanying code.

```
import pandas as pd
import networkx as nx
import numpy as np
import matplotlib.pyplot as plt
```

Library	**Purpose**
`pandas`	Data manipulation, specifically for reading data from an Excel sheet and performing data cleaning.
`networkx`	Graph manipulation, used to create and analyze the graph structure.
`random`	Generation of random numbers, used in the random walk algorithm.
`matplotlib.pyplot`	Visualization of the graph.

Table 1: Libraries Used in the Python Code

```
# Read data from Excel sheet
df = pd.read_excel('impressions.xlsx')

# Drop duplicate rows from the DataFrame
df.drop_duplicates(inplace=True)

# Create an empty directed graph
G = nx.DiGraph()

# Add nodes from the first column
nodes = df.iloc[:, 0]  # Selecting the first column
unique_nodes = nodes.unique()
G.add_nodes_from(unique_nodes)  # Add unique nodes


# Add edges from each row
for _, row in df.iterrows():
    source_node = row[0]  # Source node is in the first column
    for target_node in row[1:]:
        if pd.notnull(target_node):  # Check if the cell is not empty
            G.add_edge(source_node, target_node)


class CustomError(Exception):
    pass

# Generate adjacency matrix
adj_matrix = nx.adjacency_matrix(G)

# Convert to NumPy array
adj_matrix = adj_matrix.toarray()

# Write the adjacency matrix to a file
np.savetxt('adj_matrix.txt', adj_matrix, fmt='%d')

#Create the transposed matrix too to be used later
transposed_adj_matrix = np.transpose(adj_matrix)

# Write the adjacency matrix to a file
np.savetxt('transposed_adj_matrix.txt', transposed_adj_matrix, fmt='%d')

n, m = transposed_adj_matrix.shape


def find_link(A,i,j,X,M):
    X=np.transpose(X)
    link = np.dot(X,M)
    if(link.item()<=0):
        x=0
    else:
        x=1
    return x


def find_unknowns_1(A, i, j):
    try:
        if ((i-1)<=0) or ((j-1)<=0):
            raise CustomError("Custom error occurred.")
        # Select submatrix of A and column matrix X
        A_sub = A[:i-1, :j-1]
        B = np.array([A[i, 0:j-1]])  # B is the transpose of the ith row of A fr
        A_sub =np.transpose(A_sub)
        B=np.transpose(B)
        X, residuals, rank, s = np.linalg.lstsq(A_sub, B, rcond=None)
        M = np.array([A[0:i-1, j]])
        M =M.reshape(-1, 1)
        return find_link(A,i,j,X,M)

    except (np.linalg.LinAlgError, IndexError, CustomError) as e:
        #Exception occurred in find_unknowns_1
        return 0


for i in range(n):
    for j in range(m):
        if transposed_adj_matrix[i, j] == 0 and adj_matrix[i, j] == 0 and i!=j:
            link = find_unknowns_1(transposed_adj_matrix, i, j)
            transposed_adj_matrix[i, j] = link
            adj_matrix[i, j] = link

np.savetxt('new_adj_matrix.txt',adj_matrix, fmt='%d')

name_to_index = {node: i for i, node in enumerate(G.nodes)}

# Add edges to the predefined graph based on the adjacency matrix
num_nodes = len(adj_matrix)
for i in range(num_nodes):
    for j in range(num_nodes):
        if adj_matrix[i][j] != 0:  # If non-zero entry in the adjacency matrix
            node_i = list(G.nodes())[i]  # Get node name corresponding to index
            node_j = list(G.nodes())[j]  # Get node name corresponding to index
            G.add_edge(node_i, node_j)
```

```
nx.draw(G, with_labels = True)
plt.show()
```

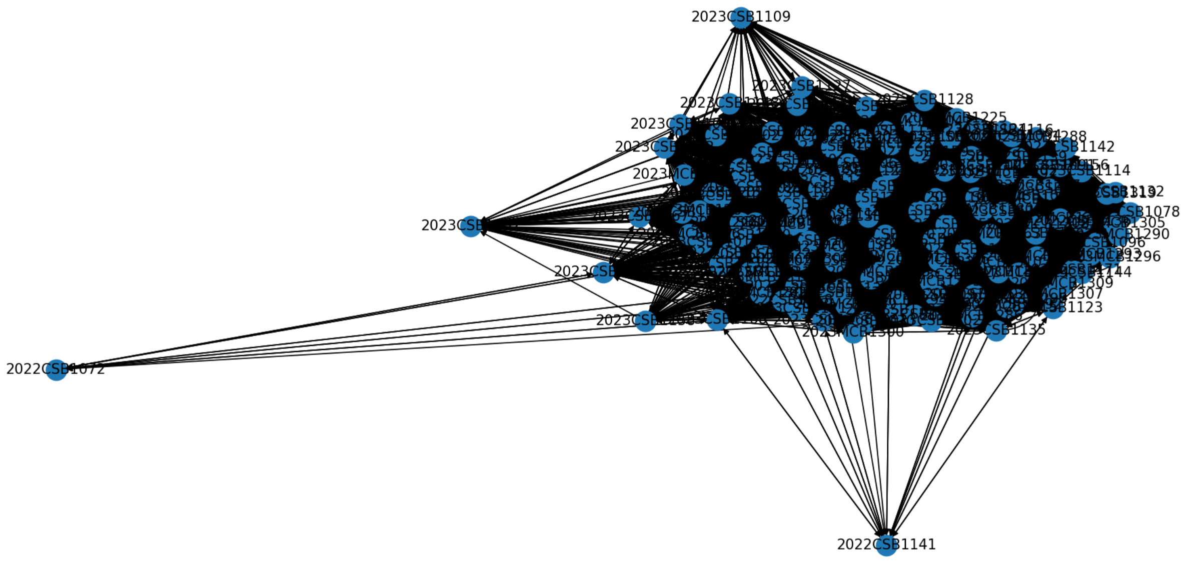


Figure 2: Graph after predicting the missing links.

2.1 Explanation

Understanding Missing Links:

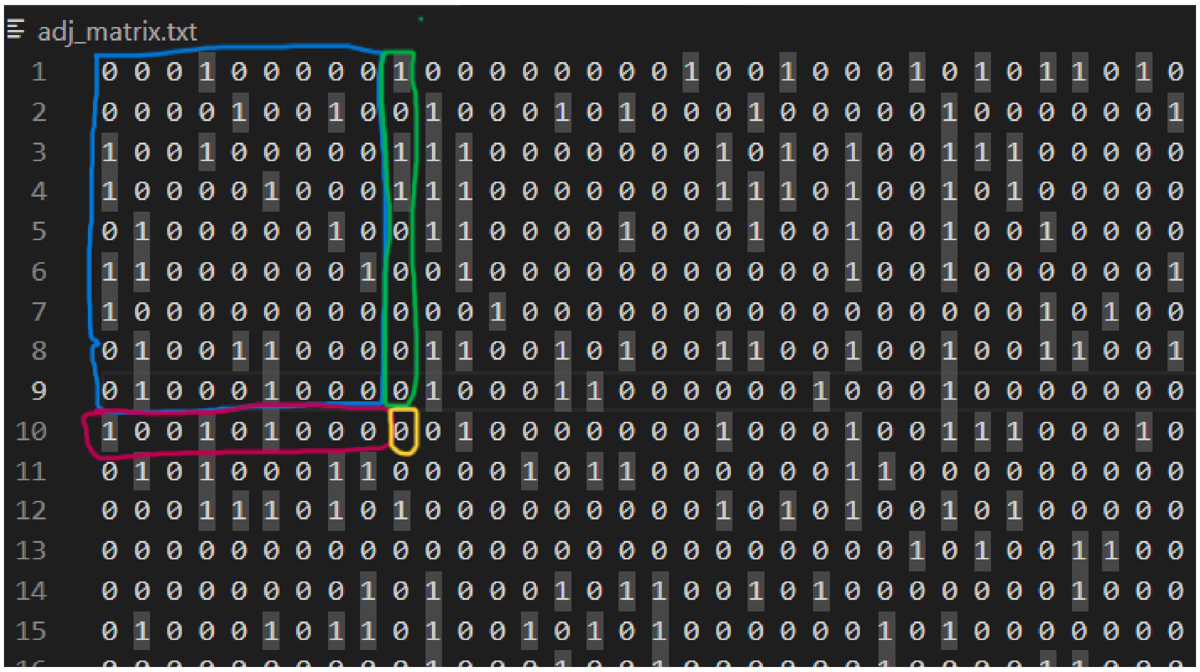
- In the adjacency matrix of the graph stored in "adj_matrix.txt", numerous nodes lack connections, represented by zeros.
- Certain nodes—let's denote them as A and B—lack any direct link between them, suggesting missing links.
- Missing links indicate nodes that have not yet interacted, and our goal is to infer the potential connection type between such nodes.

Identifying Missing Links:

- We identify missing links when both $adj\_matrix[i][j]$ and $adj\_matrix[j][i]$ are zero.
- Alternatively, when $adj\_matrix[i][j] = transposed\_adj\_matrix[i][j]$, it suggests nodes without reciprocal links.

Predictive Approach:

- Our predictive model relies on fundamental linear algebra concepts.
- By constructing matrices A, B, M, and X, we discern missing link potential by examining the relationship between elements in A and the respective row.



adj_matrix.txt

```
0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 0 1 0 0 1 0 0 0 1 0 1 0 1 1 0 1 0
0 0 0 0 1 0 0 1 0 0 1 0 0 0 1 0 1 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 1
1 0 0 1 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 1 0 1 0 1 0 0 1 1 1 0 0 0 0 0
1 0 0 0 0 1 0 0 0 1 1 1 0 0 0 0 0 0 0 1 1 1 0 1 0 0 1 0 1 0 0 0 0 0
0 1 0 0 0 0 0 1 0 0 1 1 0 0 0 0 1 0 0 0 1 0 0 1 0 0 1 0 0 1 0 0 0 0
1 1 0 0 0 0 0 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0 0 0 0 0 1
1 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0
0 1 0 0 1 1 0 0 0 0 1 1 0 0 1 0 1 0 0 1 1 0 0 1 0 0 1 0 0 1 1 0 0 1
0 1 0 0 0 1 0 0 0 0 1 0 0 0 1 1 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0
1 0 0 1 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 1 0 0 1 1 1 0 0 0 1 0
0 1 0 1 0 0 0 1 1 0 0 0 0 1 0 1 1 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0
0 0 0 1 1 1 0 1 0 1 0 0 0 0 0 0 0 0 0 1 0 1 0 1 0 0 1 0 1 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 1 1 0 0
0 0 0 0 0 0 0 0 1 0 1 0 0 0 1 0 1 1 0 0 1 0 1 0 0 0 0 0 0 0 1 0 0 0
0 1 0 0 0 1 0 1 1 0 1 0 0 1 0 1 0 1 0 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0
```

Figure 3: A_sub (Blue), B(Pink) , Missing link (Yellow) , M (Green)

- First X is Calculated using A_sub.X = B .
- After this X is multiplied with M to find the missing link.
- This methodology leverages existing node relationships to forecast potential interactions with previously unconnected nodes.

Addressing Potential Errors:

- An error may arise while solving for X in $AX = B$ due to a singular matrix A (i.e., $\det(A) = 0$), indicating an inability to invert the matrix.
- Complications may arise when the missing link is situated at the matrix's edges, necessitating appropriate boundary handling.
- Boundary conditions, such as index out-of-bound errors at the matrix's edges, necessitate careful boundary management to avoid computational errors.

- In the event of any of these errors occurring, the missing link prediction may be inconclusive, and a value of 0 is returned due to insufficient information for accurate forecasting.

3 Experiment 3: Proposed Problem

The problem proposed aims to identify communities within a graph, representing densely connected sections, and subsequently locate bridge nodes that facilitate connections between these communities. This problem is of practical significance, as communities are prevalent in various social networks, spanning across diverse interests, beliefs, and fields of study. These communities are not mutually exclusive, with individuals often belonging to multiple groups, serving as conduits for information dissemination.

```
import pandas as pd
import networkx as nx
import numpy as np
import random
import matplotlib.pyplot as plt
from collections import defaultdict

class CustomError(Exception):
    pass
# Read data from Excel sheet
df = pd.read_excel('impressions.xlsx')

# Drop duplicate rows from the DataFrame
df.drop_duplicates(inplace=True)

# Create an empty directed graph
G = nx.DiGraph()

# Add nodes from the first column
nodes = df.iloc[:, 0]  # Selecting the first column
unique_nodes = nodes.unique()
G.add_nodes_from(unique_nodes)  # Add unique nodes


# Add edges from each row
for _, row in df.iterrows():
    source_node = row[0]  # Source node is in the first column
    for target_node in row[1:]:
        if pd.notnull(target_node):  # Check if the cell is not empty
            G.add_edge(source_node, target_node)
```

```
def rank_assign(G):
 points = 10000000   # Total initial points/coins to be distributed among the no
 points_box = {}
 for node in G.nodes():
        points_box[node] = 0   # Initializing the points of all the nodes in G t

 random_node = random.choice(list(G.nodes()))
 while(points >= 0 ):
        adjacency_list = list(G.neighbors(random_node))
        if adjacency_list:
            neighbour = random.choice(adjacency_list)
        else:
            neighbour = random.choice(list(set(G.nodes()) - {random_node}))
# If the current node does not have a neighbour then choose any other node at ra
        points_box[neighbour]=points_box[neighbour]+1
# increment the points of the randomly chosen neighbour
        points=points-1
# decrement the Total points by 1
        random_node = neighbour
# make the neighbour the new random_node
 return points_box

rank_assign(G)

# Generate adjacency matrix
adj_matrix = nx.adjacency_matrix(G)

# Convert to NumPy array
adj_matrix = adj_matrix.toarray()

# Write the adjacency matrix to a file
np.savetxt('adj_matrix.txt', adj_matrix, fmt='%d')

points_box=rank_assign(G)

# Step 1: Identify the communities
def label_propagation(G):
    # Initialize each node with a unique label
    for node in G.nodes():
        G.nodes[node]['label'] = node
    while True:
        # Shuffle the nodes to avoid bias in label propagation order
        nodes = list(G.nodes())
        random.shuffle(nodes)

        # Flag to track label changes
```

```
            labels_changed = False

            # Update labels based on neighbor labels
            for node in nodes:
                neighbor_labels = [G.nodes[neighbor]['label'] for neighbor in G.neig
                if neighbor_labels:
                    most_common_label = max(set(neighbor_labels), key=neighbor_label
                    # If all labels occur once, choose the label of the neighbor wit
                    if neighbor_labels.count(most_common_label) == 1:
                        max_rank_neighbor = max(G.neighbors(node), key=lambda x: poi
                        most_common_label = G.nodes[max_rank_neighbor]['label']
                    if G.nodes[node]['label'] != most_common_label:
                        G.nodes[node]['label'] = most_common_label
                        labels_changed = True

            # Check for convergence
            if not labels_changed:
                break
        # Create a dictionary to store community memberships
        communities = {}
        for node in G.nodes():
            label = G.nodes[node]['label']
            if label not in communities:
                communities[label] = []
            communities[label].append(node)

        return communities

    # Apply Label Propagation Algorithm
    communities = label_propagation(G)
    print(communities)


    # Step 2: Determine boundary nodes
    boundary_nodes = defaultdict(set)  # Dictionary to store boundary nodes for each
    for community_label, community_nodes in communities.items():
        for node in community_nodes:
            # Check if the node has neighbors outside its community
            if any(neighbor not in community_nodes for neighbor in G.neighbors(node)
                boundary_nodes[community_label].add(node)

    # Step 3: Find bridge nodes (nodes that connect to the maximum number of commun
    bridge_nodes = set()
    for community_label, nodes in boundary_nodes.items():
        if len(nodes) == 1:  # If there's only one boundary node in the community, a
            bridge_nodes.update(nodes)
        else:
            max_connections = 0
            bridge_node = None
            for node in nodes:
                # Count the number of communities connected to the current node
                connected_communities = {label for neighbor in G.neighbors(node)
                                         for label, community_nodes in bounda
                                         if neighbor in community_nodes}
                num_connections = len(connected_communities)
                if num_connections > max_connections:
                    bridge_node = node
                    max_connections = num_connections
            if bridge_node is not None:
                bridge_nodes.add(bridge_node)

    print("Bridge nodes connecting to the most communities:", bridge_nodes)
    for node in bridge_nodes:
        connected_communities = set()

        # Add the node's own community label
        for label, community_nodes in communities.items():
            if node in community_nodes:
                connected_communities.add(label)

        # Add the communities of the node's neighbors
        for neighbor in G.neighbors(node):
            for label, community_nodes in communities.items():
                if neighbor in community_nodes:
                    connected_communities.add(label)

        print("Node", node, "is connected to the following communities:", connected_
```

To address this problem, the following method is proposed:

1. **Community Identification:** Using the Label Propagation method, each node in the graph is initially assigned a unique label. Starting from a random node, labels propagate through the graph by adopting the most common label among neighboring nodes. This process continues until community saturation is reached.
2. **Boundary Node Detection:** Boundary nodes, which connect to other communities, are identified from the established communities. These nodes serve as transition points between different community clusters.
3. **Bridge Node Localization:** Bridge nodes are then identified as nodes connected to the maximum number of communities outside their own. These pivotal nodes play a vital role in bridging disparate communities and facilitating communication and information flow across the network.

By employing this method, we aim to gain insights into the community structure of the graph and identify key nodes that bridge different communities, thereby enhancing our understanding of network dynamics and facilitating targeted interventions for information dissemination and network optimization.

4 Conclusion

In conclusion, the fusion of linear algebra and graph theory presents a potent method for inferring missing links in networks. By adapting techniques akin to the PageRank algorithm and analyzing adjacency matrices, we systematically identify potential missing connections. Through matrix operations and thresholding, this approach offers a mathematically grounded framework for uncovering hidden relationships. By applying these methods, we deepen our understanding of network structures, enabling the discovery of previously unseen connections, and advancing the field of network analysis.