Discrete Mathematics Project 2 Report

Shubham Aggarwal
2023CSB1162

1 PageRank Algorithm

1.1 Introduction

The PageRank algorithm is a widely-used algorithm for ranking web pages in search engine results. Developed by Larry Page and Sergey Brin, the founders of Google, PageRank assigns a numerical value to each element of a hyperlinked set of documents, with the purpose of measuring its relative importance within the set.

1.2 Random Walk Algorithm

The Random Walk algorithm is a stochastic process where an entity moves randomly from one state to another within a defined set of states. In the context of networks, it's often used to model the behavior of a random surfer navigating through web pages. In each step, the surfer either moves to a neighboring page with a certain probability or jumps to a random page with another probability.

1.3 My Code

Listing 1: Python code implementing the random walk algorithm and PageRank

```python
import csv
import networkx as nx
import random

# Create a directed graph
G = nx.DiGraph()

# Define a function for random walk
def random_walk(steps):
    # Initialize a dictionary to store ratings for each node
    rating = dict.fromkeys(list(G.nodes), 0)
    # Choose a random starting node
    current_node = random.choice(list(G.nodes))
    # Perform random walk for the specified number of steps
    for i in range(steps):
```

```
        # With probability 0.15, randomly jump to any node
        if random.random() < 0.15:
            current_node = random.choice(list(G.nodes))
        else:
            # Select a neighbor of the current node to move to
            neighbors = list(G.neighbors(current_node))
            if neighbors:
                current_node = random.choice(neighbors)
            else:
                # If the current node has no neighbors, randomly jump to any node
                current_node = random.choice(list(G.nodes))
        # Increment the rating of the current node
        rating[current_node] += 1
    return rating

# Open the dataset file
with open("./dataset.csv") as f:
    reader = csv.reader(f)
    # Iterate over each row in the dataset
    for row in reader:
        # Skip the header row
        if reader.line_num == 1:
            continue
        # Get the parent node (source) from the first column
        parent = row[1].lower().split('@')[0]
        # Iterate over the child nodes (targets) in the row
        for entry in row[2:]:
            # Skip empty entries
            if entry == '':
                continue
            # Get the child node (target) from the last word in the entry
            child = entry.lower().split()[-1]
            # Add an edge from the parent to the child in the graph
            G.add_edge(parent, child)

# Define the total number of steps for the random walk
total = 10000000
# Perform random walk and get the ratings for each node
rating = random_walk(total)
# Sort nodes by their ratings in descending order
ranking = sorted(rating, key=rating.get, reverse=True)
# Print the top 10 leaders based on the random walk algorithm
print("The top 10 leaders -:")
for i in range(10):
    name = ranking[i]
```

```
    rate = rating[name] / total  # Normalize rating by total
        steps
    print(i+1, name, rate)

print("The top leader is", ranking[0],"with",rating[ranking
    [0]]/total, "points.")
```

1.4 Explanation of Code

The provided Python code implements a random walk algorithm and computes PageRank scores for a given network represented as a directed graph. Here's a breakdown of the code:

- The random walk algorithm simulates the behavior of a random surfer moving through the network. It randomly selects nodes to visit based on their neighbors and performs a certain number of steps to calculate ratings for each node.

1.5 Conclusion

In conclusion, both the Random Walk algorithm and the PageRank algorithm are powerful tools for analyzing and ranking nodes within networks. While the Random Walk algorithm provides a probabilistic model for navigating networks, PageRank offers a more sophisticated approach based on the concept of importance and connectivity. By understanding and implementing these algorithms, we can gain valuable insights into the structure and dynamics of various types of networks..

2 Missing Links Prediction Using Matrix Method

2.1 Introduction

Missing links prediction using the matrix method is a technique in network analysis where the goal is to predict the existence of edges (links) between nodes in a network by representing the network as an adjacency matrix and solving a system of linear equations. This approach leverages linear algebra concepts to infer the likelihood of connections between nodes based on the network's structure.

2.2 Matrix Method for Missing Links Prediction

The matrix method for missing links prediction involves the following steps:

1. **Adjacency Matrix Representation:** Represent the given network as an adjacency matrix, where each entry A_{ij} indicates the presence or absence of an edge between nodes i and j.

2. **Linear Combination:** Treat each row of the adjacency matrix as a vector and express one row as a linear combination of the other rows. This can be done by solving a system of linear equations.

3. **Prediction:** Once the coefficients of the linear combination are determined, use these coefficients with the corresponding column vectors to predict the missing entries in the adjacency matrix.

2.3 My Code

Listing 2: Python code implementing missing links prediction using the matrix method

```python
import csv
import networkx as nx
import numpy as np

# Create a directed graph
G = nx.DiGraph()

# Open the dataset file and create the graph
with open("./dataset.csv") as f:
    reader = csv.reader(f)
    # Iterate through each row in the dataset
    for row in reader:
        # Skip the header row
        if reader.line_num == 1:
            continue
        # Extract the parent node (source) from the second
            column
        parent = row[1].lower().split('@')[0]
        # Iterate through each entry in the row (child nodes
            )
        for entry in row[2:]:
            # Skip empty entries
            if entry == '':
                continue
            # Extract the child node (target) from the last
                word in the entry
            child = entry.lower().split()[-1]
            # Add an edge from the parent to the child in
                the graph
            G.add_edge(parent, child)

# Convert the graph to adjacency matrix
nodes = list(G.nodes())
n = len(nodes)
adj = np.zeros(shape=(n,n))
for i in range(n):
```

```
    for j in range(n):
        # Check if there is an edge between nodes i and j
        if G.has_edge(nodes[i], nodes[j]):
            # If there is an edge, set the corresponding
                entry in the adjacency matrix to 1
            adj[i][j] = 1

# Function to predict missing links using matrix method
def predict(i, j):
    # Remove the j-th column from the adjacency matrix
    without_col = np.delete(adj, j, axis=1)
    # Extract the i-th row from the modified adjacency
        matrix
    B = without_col[i]
    # Remove the i-th row from the modified adjacency matrix
    A = np.delete(without_col, i, axis=0)
    # Solve the system of linear equations Ax = B to find
        coefficients x
    X = np.linalg.lstsq(A.T, B.T, rcond=None)[0]
    # Remove the i-th row from the original adjacency matrix
    without_row = np.delete(adj, i, axis=0)
    # Extract the j-th column from the modified adjacency
        matrix
    C = without_row[:, j]
    # Calculate the expected value for the missing link
    expected = np.matmul(C, X)
    return expected

# Find missing links and add them to the graph
missing_links = []
for i in range(n):
    for j in range(n):
        # Check if there is no edge between nodes i and j
        if adj[i][j] == 0:
            # Predict the value of the missing link using
                the matrix method
            val = predict(i, j)
            # If the predicted value is greater than or
                equal to 1, consider it as a missing link
            if val > 0.5:
                missing_links.append((i, j))

for i, j in missing_links:
    # Print the missing link (node i to node j)
    print("Missing Link:", (nodes[i], nodes[j]))
    # Add the missing link to the graph
    G.add_edge(nodes[i], nodes[j])

# Calculate PageRank scores for the updated graph
rating = nx.pagerank(G)
```

```
# Sort nodes by their PageRank scores in descending order
ranking = sorted(rating, key=rating.get, reverse=True)
# Print the top 10 nodes with the highest PageRank scores
for i in range(10):
    name = ranking[i]
    rate = rating[name]
    print(name, rate)
```

2.4 Code Explanation

The Python code provided below performs missing links prediction using the matrix method. Here's a detailed explanation of each section:

- The code starts by importing necessary libraries including `csv`, `networkx`, `numpy`, and `pandas`.
- It creates an empty directed graph `G` using NetworkX.
- The code reads a dataset from a CSV file to populate the directed graph `G`.
- The graph is then converted to an adjacency matrix representation using NumPy.
- A function `predict` is defined to predict missing links using the matrix method. This function takes two indices `i` and `j` corresponding to the rows and columns of the adjacency matrix.
- Missing links are predicted based on the matrix method. If the predicted value is greater than or equal to 1, the missing links are added to the graph `G`.
- PageRank scores are calculated for the updated graph using NetworkX, and the top 10 nodes with the highest PageRank scores are printed.

2.5 Conclusion

In conclusion, missing links prediction using the matrix method offers a powerful approach to infer connections between nodes in a network based on its structural properties. By leveraging linear algebra techniques, we can accurately predict missing links and gain insights into the underlying relationships within the network. The code provided serves as a practical implementation of the matrix method for missing links prediction, demonstrating the application of discrete mathematics concepts in network analysis.

3 Dijkstra's Algorithm

3.1 Introduction

Dijkstra's algorithm is a popular algorithm in graph theory for finding the shortest paths between nodes in a graph.

Dijkstra's algorithm, named after Dutch computer scientist Edsger W. Dijkstra, is a graph search algorithm that finds the shortest path between nodes in a weighted graph.

3.2 Algorithm Steps

The algorithm works by repeatedly selecting the node with the shortest distance from the source node and updating the distances of its neighbors accordingly.

3.3 Code

Listing 3: Python code for Dijkstra's algorithm

```python
import csv
import networkx as nx
import numpy as np
import sys

# Create a directed graph
G = nx.DiGraph()

# Open the dataset file and create the graph
with open("./dataset.csv") as f:
    reader = csv.reader(f)
    for row in reader:
        # Skip the header row
        if reader.line_num == 1:
            continue
        # Extract the parent node (source) from the second column
        parent = row[1].lower().split('@')[0]
        # Iterate through each entry in the row (child nodes)
        for entry in row[2:]:
            # Skip empty entries
            if entry == '':
                continue
            # Extract the child node (target) from the last word in the entry
            child = entry.lower().split()[-1]
            # Add an edge from the parent to the child in the graph
            G.add_edge(parent, child)
```

```

# Convert the graph to an adjacency matrix
nodes = list(G.nodes())
n = len(nodes)
adj = np.zeros(shape=(n,n))
for i in range(n):
    for j in range(n):
        # If there is an edge between nodes i and j, set the corresponding entry in the adjacency matrix to 1
        # If there is no edge, set the entry to -1
        if G.has_edge(nodes[i], nodes[j]):
            adj[i][j] = 1
        else:
            adj[i][j] = -1

# Implementation of Dijkstra's algorithm
def dijkstra(adjacency_matrix, start_vertex):
    # Get the number of nodes in the graph
    n = len(adjacency_matrix[0])

    # Initialize an array to store the shortest distances from the start_vertex to all other vertices
    shortest_distances = [sys.maxsize] * n
    # Initialize a boolean array to track whether each vertex has been added to the shortest path tree
    added = [False] * n

    # Set the distance from start_vertex to itself as 0
    shortest_distances[start_vertex] = 0
    # Initialize an array to store the parent vertices of each vertex in the shortest path tree
    parents = [-1] * n
    parents[start_vertex] = -1

    # Iterate over all vertices
    for i in range(1, n):
        # Find the vertex with the minimum distance from the start_vertex that has not been added to the shortest path tree
        nearest_vertex = -1
        shortest_distance = sys.maxsize
        for vertex_index in range(n):
            if not added[vertex_index] and shortest_distances[vertex_index] < shortest_distance:
                nearest_vertex = vertex_index
                shortest_distance = shortest_distances[vertex_index]

        # Add the nearest_vertex to the shortest path tree
```

```
        added[nearest_vertex] = True

        # Update the shortest distances to all vertices adjacent to nearest_vertex
        for vertex_index in range(n):
            edge_distance = adjacency_matrix[nearest_vertex][vertex_index]
            # If there is an edge from nearest_vertex to vertex_index and the distance through nearest_vertex is shorter than the current shortest distance, update the shortest distance
            if edge_distance > 0 and shortest_distance + edge_distance < shortest_distances[vertex_index]:
                parents[vertex_index] = nearest_vertex
                shortest_distances[vertex_index] = shortest_distance + edge_distance

    return shortest_distances

# Calculate the shortest distances between all pairs of nodes using Dijkstra's algorithm
distance = np.zeros(shape=(n,n))
for i in range(n):
    arr = dijkstra(adj, i)
    for j in range(n):
        # If the shortest distance is infinite (no path exists), set the distance to -1
        if arr[j] == sys.maxsize:
            distance[i][j] = -1
        else:
            distance[i][j] = int(arr[j])

# Print the shortest distance matrix
print("The shortest distance matrix: ")
print(distance)

#Observations based on the algorithm

max_distance = int(distance.max())

print("Maximum distance between any 2 nodes:", max_distance)
#This verifies that we can reach from one person to another through less than logN steps.

unreachable = 0
for i in range(n):
    for j in range(n):
        if distance[i][j] == -1:
```

```
            unreachable += 1
print("Number of pairs of unreachable nodes:", unreachable)
#We can find out how many people were absent during the exercise through this
print("Number of students absent:", unreachable/142)

counts = [0]*(max_distance+1)
for i in range(n):
    for j in range(n):
        x = int(distance[i][j])
        if 0<=x<=max_distance:
            counts[x] += 1
for i,cnt in enumerate(counts):
    print("Count of", i, ":", cnt)
```

3.4 Code Explanation

The Python code provided below performs the calculation of the shortest distance matrix between all pairs of nodes in a directed graph using Dijkstra's algorithm. Here's a detailed explanation of each section:

- The code starts by importing necessary libraries including `csv` and `networkx`.
- It creates an empty directed graph `G` using NetworkX.
- The code reads a dataset from a CSV file to populate the directed graph `G`. Each row in the CSV file represents a directed edge from a parent node (source) to one or more child nodes (targets).
- The graph is then converted to an adjacency matrix representation using NumPy. The adjacency matrix represents the connections between nodes in the graph, where a value of 1 indicates the presence of an edge, and -1 indicates no edge.
- Dijkstra's algorithm is implemented to calculate the shortest distances between all pairs of nodes in the graph. The `dijkstra` function takes the adjacency matrix and a starting vertex as input and returns an array containing the shortest distances from the starting vertex to all other vertices.
- The shortest distance matrix is initialized as a 2D NumPy array of zeros with shape (n, n), where n is the number of nodes in the graph.
- The shortest distances between all pairs of nodes are calculated using Dijkstra's algorithm and stored in the matrix `distance`.
- If there is no path between two nodes, the distance is set to -1.
- Finally, the shortest distance matrix is printed to display the shortest distances between all pairs of nodes in the graph.

3.5 Observations

- We can reach from one person to another in less than log(N) steps in a network, where N are number of persons.
- All the diagonal entries in distance matrix are 0, as it represents distance from i to i.
- The count of 0 in distance entries represents number of persons.
- The count of 1 in distance entries represents the total number of entries filled in the survey.
- Number of students absent = $\frac{Numberofnonreachablenodepairs}{Numberofstudents-1}$
- Count of 2 is maximum.
- Count vs N follows bell curve with maxima at 2.

3.6 Conclusion

In conclusion, Dijkstra's algorithm is a powerful tool for finding the shortest paths between nodes in a graph. By efficiently traversing the graph and updating distances, it provides an optimal solution for various routing and navigation problems.