Project_SEMOGLOU

December 28, 2023

0.1 Business Mathematics - Data Analysis with Python, Project 3

0.2 PageRank Web Graph Analysis

0.2.1 Angelos Semoglou, s3332318

[1]:
```
# Import necessary libraries
import numpy as np
from numpy.linalg import norm
from scipy import sparse
import time

# Suppressing RuntimeWarning to avoid unnecessary warnings during execution
# ZeroDivisionError is expected during the computation and is handled␣
 ↪appropriately
import warnings
warnings.simplefilter("ignore", category=RuntimeWarning)
```

0.2.2 Methods

Data Retrieval and Modification Functions

[2]:
```
def get_data(data):
    """
    Extracts and processes data from the given file.

    Parameters:
    - data: The path to the data file/ Name of the file (str).

    Returns:
    - edges: The number of edges in the "graph".
    - nodes: The number of nodes in the "graph".
    - incoming_edges: Dictionary with webpages as keys and the count of their␣
 ↪outgoing edges as values.
    - outgoing_edges: Dictionary with webpages as keys and a list of incoming␣
 ↪edges as values.
    - source_pages: List of the origin pages.
    - target_pages: List of the destination pages.
    """
    # source_pages: list to store the source/origin pages.
```

```
    # target_pages: list to store the target/destination pages.
    source_pages, target_pages = [], []

    # incoming_edges: Dictionary with webpages as keys and the count of their␣
↪outgoing edges as values.
    # outgoing_edges: Dictionary with webpages as keys and a list of incoming␣
↪edges as values.
    incoming_edges, outgoing_edges = {}, {}

    # Open the data file for reading.
    with open(data, 'r') as f:
        # Iterate over each line in the initial data file.
        for line in f.readlines():
            # Extract the item from the first column and subtract one.
            # This adjustment is made because webpage IDs start from 1,
            # but matrix indexes start from 0 (Same for the 2nd column).
            source = int(line.split()[0]) - 1
            destination = int(line.split()[1]) - 1

            source_pages.append(source)
            target_pages.append(destination)

            # Fill the outgoing_edges dictionary by counting how many times
            # each source page has outgoing edges.
            if source not in outgoing_edges.keys():
                outgoing_edges[source] = 1
            else:
                outgoing_edges[source] += 1

            # Fill the incoming_edges dictionary by tracking the source pages
            # for each target/destination page
            if destination not in incoming_edges.keys():
                incoming_edges[destination] = [source]
            else:
                incoming_edges[destination].append(source)

    # Compute number of edges
    edges = len(source_pages)

    # Compute the number of nodes by taking the union of unique source and␣
↪target pages
    nodes = len(set(source_pages) | set(target_pages))

    # Ensure that all nodes have entries in the incoming_edges dictionary
    for node in range(nodes):
        if node not in incoming_edges.keys():
            incoming_edges[node] = []
```

```
    # Return the computed values
    return edges, nodes, incoming_edges, outgoing_edges, source_pages,␣
 ↪target_pages
```

[3]:
```
def get_data_add_X(data):
    """
    Load data from a file and add a node X without inlinks or outlinks.

    Parameters:
    - data: The path to the data file/ Data file name.

    Returns:
    - edges: The updated number of edges.
    - nodes: The updated number of nodes.
    - source_pages: List of origin pages.
    - target_pages: List of destination pages.
    """
    source_pages, target_pages = [], []

    with open(data, 'r') as f:
        # Extract source and destination pages from each line.
        for line in f.readlines():
            source = int(line.split()[0]) - 1
            destination = int(line.split()[1]) - 1
            source_pages.append(source)
            target_pages.append(destination)

    edges = len(source_pages)
    nodes = len(set(source_pages) | set(target_pages))
    # Increment the number of nodes to account for the new node X.
    nodes += 1

    return edges, nodes, source_pages, target_pages
```

[4]:
```
def get_data_add_XY(data):
    """
    Load data from a file and add two new nodes X, Y and a new edge Y -> X.
    (Y links to X)

    Parameters:
    - data: The path to the data file/ Data file name.

    Returns:
    - edges: Number of edges in the graph.
    - nodes: Number of nodes in the graph.
    - source_pages: List to store the origin pages.
```

```
    - target_pages: List to store the destination pages.
    """
    source_pages, target_pages = [], []

    with open(data, 'r') as f:
        for line in f.readlines():
            source = int(line.split()[0]) - 1
            destination = int(line.split()[1]) - 1
            source_pages.append(source)
            target_pages.append(destination)

    edges = len(source_pages)
    nodes = len(set(source_pages) | set(target_pages))

    # Add the new edge Y -> X
    source_pages.append(nodes)
    target_pages.append(nodes + 1)

    # Increase the number of edges by one
    edges += 1
    # Increase the number of nodes by two for X, Y
    nodes += 2

    return edges, nodes, source_pages, target_pages
```

[5]:
```
def get_data_add_XYZ(data):
    """
    Load data from a file and add three nodes X, Y, Z and two new edges (Y -> X␣
 ↪and Z -> X)
    to the graph, updating the number of nodes and edges accordingly.

    Parameters:
    - data: The path to the data file/ Data file name.

    Returns:
    - edges: The updated number of edges.
    - nodes: The updated number of nodes.
    - source_pages: List of origin pages.
    - target_pages: List of destination pages.
    """
    source_pages, target_pages = [], []

    with open(data, 'r') as f:
        for line in f.readlines():
            source = int(line.split()[0]) - 1
            destination = int(line.split()[1]) - 1
            source_pages.append(source)
```

```
            target_pages.append(destination)

    edges = len(source_pages)
    nodes = len(set(source_pages) | set(target_pages))

    # New edge Y -> X
    source_pages.append(nodes)
    target_pages.append(nodes + 2)

    # New edge Z -> X
    source_pages.append(nodes + 1)
    target_pages.append(nodes + 2)

    nodes += 3
    edges += 2

    return edges, nodes, source_pages, target_pages
```

[6]:
```
def get_data_add_XYZ_popular_1(data, popular):
    """
    Load data from a file, add nodes X, Y, Z, and links from X to popular pages.

    Parameters:
    - data: The path to the data file/ Data file name.
    - popular: List of popular nodes to link with node X.

    Returns:
    - edges: The updated number of edges.
    - nodes: The updated number of nodes.
    - source_pages: List of origin pages.
    - target_pages: List of destination pages.
    """
    source_pages, target_pages = [], []
    with open(data, 'r') as f:
        for line in f.readlines():
            source = int(line.split()[0]) - 1
            destination = int(line.split()[1]) - 1
            source_pages.append(source)
            target_pages.append(destination)

    edges = len(source_pages)
    nodes = len(set(source_pages) | set(target_pages))

    source_pages.append(nodes)
    target_pages.append(nodes + 2)

    source_pages.append(nodes + 1)
```

```
    target_pages.append(nodes + 2)

    # Add links from X to older, popular pages
    for node in popular:
        source_pages.append(nodes + 2)
        target_pages.append(node)

    # Increase number of nodes and edges as needed
    nodes += 3
    edges += len(popular) + 2

    return edges, nodes, source_pages, target_pages
```

[7]:
```
def get_data_add_XYZ_popular_2(data, popular):
    """
    Load data from a file, add nodes X, Y, Z, and links from Y or Z to popular␣
 ↪pages.

    Parameters:
    - data: The path to the data file/ Data file name.
    - popular: List of popular nodes.

    Returns:
    - edges: The updated number of edges.
    - nodes: The updated number of nodes.
    - source_pages: List of origin pages.
    - target_pages: List of destination pages.
    """
    source_pages, target_pages = [], []
    with open(data, 'r') as f:
        for line in f.readlines():
            source = int(line.split()[0]) - 1
            destination = int(line.split()[1]) - 1
            source_pages.append(source)
            target_pages.append(destination)

    edges = len(source_pages)
    nodes = len(set(source_pages) | set(target_pages))

    source_pages.append(nodes)
    target_pages.append(nodes + 2)

    source_pages.append(nodes + 1)
    target_pages.append(nodes + 2)

    # add links from X to older, popular pages
    for node in popular:
```

```
        source_pages.append(nodes + 2) # Assuming it's node Z, change to (nodes␣
↪+ 1) for Y
        target_pages.append(node)

    # increase number of nodes and edges as needed
    nodes += 3
    edges += len(popular) + 2

    return edges, nodes, source_pages, target_pages
```

Sparse Matrix Construction Functions

[8]:
```
def construct_sparse_matrix(data):
    """
    Construct a sparse matrix from the given connectivity data.
    (sparse matrix is used for memory efficiency)

    Parameters:
    - data: The path to the input data file/ Data file name.

    Returns:
    - sparse_matrix (sparse.csr_matrix): The constructed Sparse-CSR matrix -␣
↪representation of the graph.
    """
    edges, nodes, _, _, source_pages, target_pages = get_data(data)

    edge_weights = [1]*edges
    sparse_matrix = sparse.csr_matrix((edge_weights,
                                       (target_pages, source_pages)),
                                       shape=(nodes, nodes))
    return sparse_matrix
```

[9]:
```
def construct_sparse_matrix_X(data):
    """
    Construct a sparse matrix for the modified graph
    with an additional node X without inlinks or outlinks.

    Parameters:
    - data: The path to the input data file.

    Returns:
    - The sparse matrix representation of the graph.
    """
    # Get data for the modified graph with an additional node X
    edges, nodes, source_pages, target_pages = get_data_add_X(data)

    edge_weights = [1]*edges
    sparse_matrix = sparse.csr_matrix((edge_weights,
```

```
                                    (target_pages, source_pages)),
                                    shape=(nodes, nodes))
    return sparse_matrix
```

[10]:
```
def construct_sparse_matrix_XY(data):
    """
    Construct a sparse matrix for the modified graph
    with two additional nodes X, Y and an edge from Y to X.

    Parameters:
    - data: The path to the input data file.

    Returns:
    - The sparse matrix representation of the graph.
    """
    # Get data for the modified graph with an additional edge from Y to X
    edges, nodes, source_pages, target_pages = get_data_add_XY(data)

    edge_weights = [1]*edges
    sparse_matrix = sparse.csr_matrix((edge_weights,
                                    (target_pages, source_pages)),
                                    shape=(nodes, nodes))
    return sparse_matrix
```

[11]:
```
def construct_sparse_matrix_XYZ(data):
    """
    Construct a sparse matrix for the modified graph
    with three additional nodes X, Y ,Z and edges from Y and Z to X.

    Parameters:
    - data: The path to the input data file.

    Returns:
    -The sparse matrix representation of the graph.
    """
    # Get data for the modified graph with additional edges from Y and Z to X
    edges, nodes, source_pages, target_pages = get_data_add_XYZ(data)

    edge_weights = [1]*edges
    sparse_matrix = sparse.csr_matrix((edge_weights,
                                    (target_pages, source_pages)),
                                    shape=(nodes, nodes))
    return sparse_matrix
```

[12]:
```
def construct_sparse_matrix_popular_1(data, popular):
    """
```

```
    Construct a sparse matrix based on the provided data, emphasizing popular␣
↪nodes.
    (nodes X, Y, Z, and links from X to popular pages)

    Parameters:
    - data: The input data file.
    - popular: A list of popular nodes.

    Returns:
    - A sparse matrix representing the graph.
    """
    edges, nodes, source_pages, target_pages = get_data_add_XYZ_popular_1(data,␣
↪popular)
    edge_weights = [1]*edges
    sparse_matrix = sparse.csr_matrix((edge_weights, (target_pages,␣
↪source_pages)), shape=(nodes, nodes))
    return sparse_matrix
```

[13]:
```
def construct_sparse_matrix_popular_2(data, popular):
    """
    Construct a sparse matrix based on the provided data, emphasizing popular␣
↪nodes.
    (nodes X, Y, Z, and links from Z to popular pages)

    Parameters:
    - data: The input data file.
    - popular: A list of popular nodes.

    Returns:
    - A sparse matrix representing the graph.
    """
    edges, nodes, source_pages, target_pages = get_data_add_XYZ_popular_2(data,␣
↪popular)

    edge_weights = [1]*edges
    sparse_matrix = sparse.csr_matrix((edge_weights,
                                       (target_pages, source_pages)),
                                       shape=(nodes, nodes))
    return sparse_matrix
```

PageRank Algorithm Implementation with Power Method

[14]:
```
def PageRank_PowerMethod(G, alpha, tolerance = 1e-8):
    """
    Compute PageRank using the Power Method.

    Parameters:
```

```
    - G: The adjacency matrix of the graph.
    - alpha: The teleport probability.
    - tolerance: The convergence tolerance. Defaults to 1e-8.

    Returns:
    - ranks: The computed PageRank vector.
    - iterations: The number of iterations performed.
    - converged_nodes: List of nodes that converged in the first iteration.
    """
    n = G.shape[0]

    # Number of outlinks in each node-webpage
    d = G.sum(axis=0).T

    error = np.inf    # Initialize error to infinity
    iterations = 0    # Iteration counter
    converged_nodes = []    # List to store nodes that converged in the first␣
↪iteration

    ranks = np.ones((n,1))/n    # Initialize the vector for the algorithm


    while error > tolerance:
        try:
            new_ranks = G.dot(alpha * (ranks / d))  # G.dot(1/d) is the␣
↪stochastic matrix
        except (ZeroDivisionError):
            pass
        new_ranks += (1 - alpha) / n

        # Normalize using L1 Norm (Manhattan Distance) to ensure that the ranks␣
↪sum to 1.
        new_ranks = new_ranks / np.linalg.norm(new_ranks, ord=1)

        iterations += 1

        # Identify the nodes that converge in the first iteration
        if iterations < 2:
            for node in range(len(ranks)):
                if np.linalg.norm(ranks[node]-new_ranks[node]) < tolerance:
                    converged_nodes.append([node])

        # Check the stopping condition
        error = np.linalg.norm(ranks - new_ranks)/np.linalg.norm(ranks)
        ranks = new_ranks

    return new_ranks, iterations, converged_nodes
```

0.2.3 Question 1. PageRank Vector

[15]:
```
G = construct_sparse_matrix('stanweb.dat')
n = G.shape[0]
```

[16]:
```
start =  time.time()

# Compute PageRank using Power Method
ranks, iterations, fast = PageRank_PowerMethod(G, 0.85, 1e-8)

end = time.time()
duration = format(end - start, '.2f')

print('PageRank with alpha = 0.85')
print(f'Duration: {duration} seconds')
print(f'Number of iterations: {iterations}')
print(f'\nPageRank vector:\n\n {ranks}')

ranks = np.asarray(ranks).ravel()
nodes_alpha_85 = ranks.argsort()[-n:][::-1]
print(f'\n Top 100 ranked nodes are:\n\n {nodes_alpha_85[:100]}')
```

```
PageRank with alpha = 0.85
Duration: 4.60 seconds
Number of iterations: 95

PageRank vector:

 [[5.32288161e-07]
 [1.17232693e-04]
 [8.25934227e-07]
 …
 [5.35938215e-07]
 [1.80768199e-06]
 [1.47755921e-06]]

 Top 100 ranked nodes are:

 [ 89072 226410 241453 262859 134831 234703 136820  68888 105606  69357
  67755 225871 186749 272441 251795  95162 119478 231362  55787 167294
 179644  38341 117151 198089  60209 235495 132694 181700 247240 259454
  62477 120707 161889  77998  17780 176789 183003 221086 137631  96744
 112741 145891 151427  60439  81434 208541     90 258347 214127 222872
  27903 272761  96357  93777  34572 158567 192119 227978 245658 118243
  28599 104766  18545  13238 101160  65462  32103 279367  65579 185471
 170451  38948  93988 273988  84427 186901   2259  52021 205476 134374
  49101 151980  17566 173904  19188 137797 174664  84905 113672 208085
  67502 210161 153025 272960  47257  36368  89776 184124  59589  66693]
```

0.2.4 Question 2. PageRank Vectors with Different Alpha Values

[17]:
```
# List of alpha values to test
alpha_values = [0.75, 0.80, 0.90, 0.95]

# Lists to store indices/nodes of the ranks vector for each alpha
ind_nodes_list = []

# Iterate over each alpha value
for alpha in alpha_values:
    start_time = time.time()

    # Compute PageRank using Power Method
    ranks = PageRank_PowerMethod(G, alpha, 1e-8)[0]

    end_time = time.time()
    duration = format(end_time - start_time, '.2f')

    # Display results for the current alpha
    print(f'\nPageRank with alpha = {alpha}')
    print(f'Duration: {duration} seconds')
    print(f'Number of iterations: {iterations}')
    print(f'\nPageRank vector:\n\n {ranks}')

    # Get the indices of the ranked nodes
    ranks = np.asarray(ranks).ravel()

    alpha_nodes = ranks.argsort()[-n:][::-1]
    print('\nTop 100 ranked nodes are:\n\n', alpha_nodes[:100])

    # Save the indices - nodes for later comparison
    ind_nodes_list.append(alpha_nodes)

    print('\n' + '-'*71)  # Separator for better readability
```

```
PageRank with alpha = 0.75
Duration: 3.60 seconds
Number of iterations: 95

PageRank vector:

 [[8.87133439e-07]
 [1.02190903e-04]
 [1.20202886e-06]
 …
 [8.92501002e-07]
 [2.07445288e-06]
```

```
 [1.93160895e-06]]

Top 100 ranked nodes are:

 [226410  89072 241453 134831  67755  69357 225871 234703 186749 231362
 105606 136820  68888 167294 262859  38341 119478  95162 251795 272441
  55787 198089  81434 214127  93777  34572 245658  60209 117151 158567
 258347 132694 235495 101160 179644 181700 259454 247240 120707  62477
 137631 221086 176789 183003  77998  17780  96744  27903 272761 112741
 161889 170451 151427 145891  65462  60439 208541     90 222872  96357
 134374 227978  28599 104766  18545  13238  93988 118243 192119 151980
 279367  32103  67502 186901   2259 205476  84427 185471  19188 278082
  38948 273988  52021 137797 208085 133796  84905  98386 173904 177306
 113672  49101  17566  50784 267491  68294 166975  96386 203321   3407]

------------------------------------------------------------------------

PageRank with alpha = 0.8
Duration: 3.78 seconds
Number of iterations: 95

PageRank vector:

 [[7.09713890e-07]
 [1.09075249e-04]
 [1.01998003e-06]
 …
 [7.14294302e-07]
 [1.95675022e-06]
 [1.72541718e-06]]

Top 100 ranked nodes are:

 [ 89072 226410 241453 134831  69357  67755 234703 225871 186749 262859
 105606 136820  68888 231362 272441 251795  95162 167294 119478  38341
  55787 198089 179644  81434 214127 117151  93777  60209  34572 132694
 235495 245658 258347 158567 181700 259454 247240 120707  62477 176789
 183003 137631 221086  77998  17780  96744  27903 272761 101160 112741
 161889 145891 151427  60439 208541     90 222872  96357 227978  65462
  28599 104766 192119  18545  13238 170451 118243 279367  32103 134374
  93988 185471 151980 186901  84427   2259 205476  38948 273988  67502
  19188  52021 137797  49101  17566 173904  65579 278082 208085  84905
 113672 133796  98386 177306 210161 153025 272960  47257   3407 203321]

------------------------------------------------------------------------

PageRank with alpha = 0.9
Duration: 5.53 seconds
```

```
Number of iterations: 95

PageRank vector:

 [[3.54856905e-07]
 [1.27134963e-04]
 [6.14462944e-07]
 …
 [3.57433386e-07]
 [1.59036412e-06]
 [1.16279102e-06]]

Top 100 ranked nodes are:

 [ 89072 226410 241453 262859 136820  68888 134831 179644 234703  95162
 272441 251795 105606 119478  55787 225871 186749  69357  67755 167294
 231362 117151  38341 235495 161889 181700 112741 145891 151427  60209
 132694 259454 247240  60439  62477 120707 183003 137631 176789  17780
  77998 221086  96744     90 208541  65579 222872  96357 192119  27903
 272761 198089 118243 227978  32103 258347 174664  28599 104766  18545
  13238 279367 281771  81434  65462 214127 185471 158567  93777  38948
  34572  49101  17566 273988 173904  52021  77083 245658 186901 101160
  84427   2259 205476  93988  36368  19188 170451 210161 113672 123954
 184124  66693 272960  47257 153025  74360  89776 233601 247955  84905]

------------------------------------------------------------------------

PageRank with alpha = 0.95
Duration: 7.57 seconds
Number of iterations: 95

PageRank vector:

 [[1.77422132e-07]
 [1.36754855e-04]
 [3.75034721e-07]
 …
 [1.78781845e-07]
 [1.19103284e-06]
 [7.24616211e-07]]

Top 100 ranked nodes are:

 [ 89072 226410 241453 262859 179644 136820  68888 272441 251795  95162
 281771 174664 119478 234703  65579 105606  55787 134831 117151 235495
  77083 181700 247240 259454  62477 120707  77998  17780 183003 176789
 137631 221086  96744 161889 145891 225871 112741 151427  60439     90
 192119 186749 208541 222872  96357  60209  27903 272761 167294 132694
```

```
 32103 118243  69357  38341  67755 271408  14784 231362 227978 229579
 95365  28599 104766  18545  13238 173904 185471  49101  17566 152336
 36368 279367  38948  65462 123954 273988  52021  74360 198089 210161
258347  66693 184124 272960  47257 186901  89776   2259 233601 247955
205476 113672 153025  84427  19188  14355  59589 205331 161513 167817]
```

We observe that certain rankings remain consistent, while others show variations.

A comparison between the results obtained with alpha = 0.85 and those of other alpha values (0.75, 0.80, 0.90, 0.95) will be performed to identify any changes in the rankings.

[18]:
```
# Iterate over each alpha value and compare its top 100 nodes with alpha = 0.85
for i, nodes  in enumerate(ind_nodes_list):
    print(f'\nComparison with alpha = {alpha_values[i]} and alpha = 0.85:')

    # Compare two lists element-wise and print the number of differing elements
    counter = 0
    for j in range(100):
        if nodes_alpha_85[j] != nodes[j]:
            counter += 1

    if counter == 0:
        print('The top 100 ranked nodes are the same for all alpha values.')
    else:
        print('The number of different nodes in the top 100 rankings is:',␣
 ↪counter)
```

```
Comparison with alpha = 0.75 and alpha = 0.85:
The number of different nodes in the top 100 rankings is: 98

Comparison with alpha = 0.8 and alpha = 0.85:
The number of different nodes in the top 100 rankings is: 93

Comparison with alpha = 0.9 and alpha = 0.85:
The number of different nodes in the top 100 rankings is: 91

Comparison with alpha = 0.95 and alpha = 0.85:
The number of different nodes in the top 100 rankings is: 95
```

0.2.5 Question 3. Speed of Convergence

[19]:
```
ranks, iterations, converged_nodes = PageRank_PowerMethod(G, 0.85, 1e-8)

# Get indices of the top ranked nodes
ranks = np.asarray(ranks).ravel()
top_ranked_indices = ranks.argsort()[-G.shape[0]:][::-1]
```

```
print('Number of nodes that converged after the first iteration:',␣
 ↪len(converged_nodes))

# Check if any of the top 1000 highly ranked nodes converged in the first␣
 ↪iteration
flag = False
converged_in_top_1000 = [index for index in top_ranked_indices[:1000] if index␣
 ↪in converged_nodes]

if converged_in_top_1000:
    print(f'Number of highly ranked nodes (top 1000) that converged:␣
 ↪{len(converged_in_top_1000)}')
    print('Highly ranked nodes (top 1000) that converged after the first␣
 ↪iteration:', converged_in_top_1000)
else:
    print('\nNone of the top 100 highly ranked nodes converged after the first␣
 ↪iteration.')
```

```
Number of nodes that converged after the first iteration: 14534
Number of highly ranked nodes (top 1000) that converged: 5
Highly ranked nodes (top 1000) that converged after the first iteration: [35155,
1692, 169244, 188854, 116529]
```

0.2.6 Question 4. PageRank Analysis with Addition of New Web Page X

[20]:
```
# Construct the adjacency matrix for the original graph
G = construct_sparse_matrix('stanweb.dat')

# Construct the adjacency matrix for the graph with an additional node X (no␣
 ↪in-links or out-links)
G_X = construct_sparse_matrix_X('stanweb.dat')

# Compute PageRank for the original graph
ranks = PageRank_PowerMethod(G, 0.85, 1e-8)[0]
# Compute PageRank for the graph with an additional node X
ranks_X = PageRank_PowerMethod(G_X, 0.85, 1e-8)[0]

# Extract the PageRank of the added node X
pagerank_X = np.asarray(ranks_X[G_X.shape[0] - 1]).ravel()[0]
print('The PageRank of page X is:', pagerank_X)

# Trim ranks_X to match the length of ranks for comparison
ranks_X = ranks_X[:len(ranks)]
ranks = np.asarray(ranks).ravel()
ranks_X = np.asarray(ranks_X).ravel()
```

```
# Calculate the change in PageRank of the other pages due to the addition of␣
 ↪the new page X
difference = np.linalg.norm(ranks - ranks_X)
print('Change in PageRank of the other pages due to the addition of the new␣
 ↪page:', difference)

# Get the indices of the top-ranked nodes for both the original and modified␣
 ↪graphs
indices_ranks = ranks.argsort()[-len(ranks):][::-1]
indices_ranks_X = ranks_X.argsort()[-len(ranks_X):][::-1]

# Changes in ranks
print('Changes in:', np.sum(indices_ranks[:] != indices_ranks_X[:]), 'ranks.')
```

```
The PageRank of page X is: 5.32286513760914e-07
Change in PageRank of the other pages due to the addition of the new page:
1.8883261083181873e-08
Changes in: 66214 ranks.
```

We notice that the introduction of the new page has minimal impact on the PageRanks of the existing pages since they have no connections to the newly added page. It is crucial to consider that the total number of nodes is significantly large. The PageRank of the newly added page is influenced by the calculation formula for PageRank. Essentially, this contribution is solely derived from the term (1 - alpha)/n

0.2.7 Question 5. PageRank Impact Analysis with Addition of New Web Page Y and Link to X

[21]:
```
G = construct_sparse_matrix('stanweb.dat')
G_XY = construct_sparse_matrix_XY('stanweb.dat')

ranks = PageRank_PowerMethod(G, 0.85, 1e-8)[0]
ranks_XY = PageRank_PowerMethod(G_XY, 0.85, 1e-8)[0]

# Extract the PageRanks of pages X and Y after the addition of Y

pagerank_X = np.asarray(ranks_XY[G_XY.shape[0] - 1]).ravel()[0]
print('PageRank of page X after the addition of the new page Y is:', pagerank_X)

pagerank_Y = np.asarray(ranks_XY[G_XY.shape[0] - 2]).ravel()[0]
print('The PageRank of page Y is:', pagerank_Y)

# Trim ranks_XY for comparison
ranks_XY = ranks_XY[:len(ranks)]
ranks = np.asarray(ranks).ravel()
ranks_XY = np.asarray(ranks_XY).ravel()

difference = np.linalg.norm(ranks - ranks_XY)
```

```
print('Change in PageRank of the other pages due to the addition of the new␣
 ↪pages:', difference)

indices_ranks = ranks.argsort()[-len(ranks):][::-1]
indices_ranks_XY = ranks_XY.argsort()[-len(ranks_XY):][::-1]

print('Changes in:', np.sum(indices_ranks[:] != indices_ranks_XY[:]), 'ranks.')
```

```
PageRank of page X after the addition of the new page Y is:
9.848890761537594e-07
The PageRank of page Y is: 5.322848301781068e-07
Change in PageRank of the other pages due to the addition of the new pages:
4.2346278627566204e-08
Changes in: 69781 ranks.
```

We note a more significant change in the PageRanks of the existing pages resulting from the introduction of the new pages. It is crucial to consider the substantial number of nodes in the network. The PageRank of the newly added page aligns with the expected outcome from the PageRank calculation formula. Notably, the PageRank of X has visibly improved, while the PageRank of Y remains the same as that of page X in the previous question.

0.2.8 Question 6. Maximizing PageRank for Page X with links from Pages Y and Z

To maximize the PageRank of X, the links from both Y and Z are directed exclusively to X, aligning with the underlying logic of PageRank. The implementation of this approach is encapsulated in the get_data_add_XYZ method, which adds nodes Y and Z, strategically configuring links to enhance the PageRank of X. The observed results confirm a significant improvement in the PageRank of X.

[22]:
```
# Similar analysis to previous questions for the addition of new pages X, Y,␣
 ↪and Z and links

G = construct_sparse_matrix('stanweb.dat')
G_XYZ = construct_sparse_matrix_XYZ('stanweb.dat')

ranks = PageRank_PowerMethod(G, 0.85, 1e-8)[0]
ranks_XYZ = PageRank_PowerMethod(G_XYZ, 0.85, 1e-8)[0]

pagerank_X = np.asarray(ranks_XYZ[G_XYZ.shape[0] - 1]).ravel()[0]
print('The PageRank of page X is:', pagerank_X)

pagerank_Y = np.asarray(ranks_XYZ[G_XYZ.shape[0] - 2]).ravel()[0]
print('The PageRank of page Y is:', pagerank_Y)

pagerank_Z = np.asarray(ranks_XYZ[G_XYZ.shape[0] - 3]).ravel()[0]
print('The PageRank of page Z is:', pagerank_Z)

ranks_XYZ = ranks_XYZ[:len(ranks)]
ranks = np.asarray(ranks).ravel()
```

```
ranks_XYZ = np.asarray(ranks_XYZ).ravel()

difference = np.linalg.norm(ranks - ranks_XYZ)
print('Change in PageRank of the other pages due to the addition of the new␣
 ↪pages:', difference)

indices_ranks = ranks.argsort()[-len(ranks):][::-1]
indices_ranks_XYZ = ranks_XYZ.argsort()[-len(ranks_XYZ):][::-1]

print('Changes in:', np.sum(indices_ranks[:] != indices_ranks_XYZ[:]), 'ranks.')
```

```
The PageRank of page X is: 1.4374891233916276e-06
The PageRank of page Y is: 5.32283146606264e-07
The PageRank of page Z is: 5.32283146606264e-07
Change in PageRank of the other pages due to the addition of the new pages:
6.654224222608326e-08
Changes in: 78356 ranks.
```

The PageRank of X has improved, while the PageRank of Y and Z remains the same as that of page Y in the previous question.

0.2.9 Question 7. PageRank Improvement Strategies

Links from X to Older, Popular Pages and Links from Y or Z to Older, Popular Pages

[23]:
```
# Determine the top 100 most popular pages based on in-degree
incoming_edges = get_data('stanweb.dat')[2]
list_degree = np.asarray([len(incoming_edges[x]) for x in␣
 ↪sorted(list(incoming_edges.keys()))])

indices_degree = list_degree.argsort()[-len(list_degree):][::-1]
popular_100 = indices_degree[0:100]
print(f'100 most popular pages:\n{popular_100}')
```

```
100 most popular pages:
[226410 234703 105606 241453 167294 198089  81434 214127  38341 245658
  34572  89072  69357  67755 134831 231362 120707  62477 176789 259454
 137631 247240 183003 221086  77998 181700  17780  96744 186749 225871
  68888 136820 251795 272441  95162  84427 133796 119478 251989  93777
  27903 272761 186901 205476   2259  98386  19188 213897 192928 151938
 170451  67502 134374 117151 235495  60209 258347 151980  93988  84905
  27114 132694 267491  50784  55787 137689 265386 204568 130831 270770
  35083  58763  41116  65462  32103 271204 121420 248138  86239  63855
 226289 165188  92640  20532 117863 281567 247251 243179 244194  19469
 141369 116179 177013  14124  53054  82475   3163 153449  73529 262859]
```

[24]:
```
# Construct the adjacency matrix for the graph with links from X to the popular␣
 ↪pages
G_pop_1 = construct_sparse_matrix_popular_1('stanweb.dat', popular_100)
```

```
# Construct the adjacency matrix for the graph with links from Y to the popular␣
 ↪pages
G_pop_2 = construct_sparse_matrix_popular_2('stanweb.dat', popular_100)

# Compute PageRank for the graph with links from X to popular pages
ranksp1 = PageRank_PowerMethod(G_pop_1, 0.85, 1e-8)[0]

# Compute PageRank for the graph with links from Y to popular pages
ranksp2 = PageRank_PowerMethod(G_pop_2, 0.85, 1e-8)[0]

# Extract the PageRank of page X in the graph with links from X to popular pages
pagerank_Xpop_1 = np.asarray(ranksp1[G_pop_1.shape[0] - 1]).ravel()[0]

# Extract the PageRank of page X in the graph with links from Y to popular pages
pagerank_Xpop_2 = np.asarray(ranksp2[G_pop_2.shape[0] - 1]).ravel()[0]

print('If we add links from X to older, popular pages the PageRank of page X is:
 ↪', pagerank_Xpop_1)
print('If we add links from Y to older, popular pages the PageRank of page X is:
 ↪', pagerank_Xpop_2)
```

```
If we add links from X to older, popular pages the PageRank of page X is:
1.4374862565092927e-06
If we add links from Y to older, popular pages the PageRank of page X is:
1.4374862565092927e-06
```

It appears that introducing additional links to either X or Y does not alter the results obtained from the power method. The rankings persist without significant changes.

0.2.10 Question 8. New Node with Links to Top 10 Nodes and In-Links from 500 Nodes Below Average

[25]:
```
ranks_G = PageRank_PowerMethod(G, 0.85, 1e-8)[0]
ranks_G = np.asarray(ranks_G).ravel()
nodes_G = ranks_G.argsort()[-n:][::-1]

chosen_nodes = []

# Select 500 nodes with ranks below the mean rank
for x in range(0, 1000):
    chosen_nodes.append(list(ranks_G).index(ranks_G[ranks_G < np.
 ↪mean(ranks_G)][x]))

chosen_nodes = list(set(chosen_nodes))[:500]

# Create a dictionary to store the chosen and important nodes
d={}
```

```
d['chosen nodes'] = chosen_nodes

d['important nodes'] = list(nodes_G[:10].ravel())
```

[26]:
```
def get_data_8(data, d):
    """
    Extracts and processes data from the given file.

    Parameters:
    - data: The path to the data file/ Name of the file (str).
    - d: A dictionary containing information about the top 10 nodes and 500␣
 ↪nodes below average.

    Returns:
    - edges: The number of edges in the "graph".
    - nodes: The number of nodes in the "graph".
    - incoming_edges: Dictionary with webpages as keys and the count of their␣
 ↪outgoing edges as values.
    - outgoing_edges: Dictionary with webpages as keys and a list of incoming␣
 ↪edges as values.
    - source_pages: List of the origin pages.
    - target_pages: List of the destination pages.
    """
    source_pages, target_pages = [], []
    incoming_edges, outgoing_edges = {}, {}
    with open(data, 'r') as f:
        for line in f.readlines():
            source = int(line.split()[0]) - 1
            destination = int(line.split()[1]) - 1

            source_pages.append(source)
            target_pages.append(destination)

            if source not in outgoing_edges.keys():
                outgoing_edges[source] = 1
            else:
                outgoing_edges[source] += 1

            if destination not in incoming_edges.keys():
                incoming_edges[destination] = [source]
            else:
                incoming_edges[destination].append(source)

    for x in d['important nodes']:
        source = 281903
        destination = x
```

```
            source_pages.append(source)
            target_pages.append(destination)

            if source not in outgoing_edges.keys():
                outgoing_edges[source] = 1
            else:
                outgoing_edges[source] += 1

            if destination not in incoming_edges.keys():
                incoming_edges[destination] = [source]
            else:
                incoming_edges[destination].append(source)

        for x in d['chosen nodes']:
            source = x
            destination = 281903 # New node is the last node #281903
            source_pages.append(source)
            target_pages.append(destination)

            if source not in outgoing_edges.keys():
                outgoing_edges[source] = 1
            else:
                outgoing_edges[source] += 1

            if destination not in incoming_edges.keys():
                incoming_edges[destination] = [source]
            else:
                incoming_edges[destination].append(source)

        edges = len(source_pages)
        nodes = len(set(source_pages) | set(target_pages))

        for node in range(nodes):
            if node not in incoming_edges.keys():
                incoming_edges[node] = []

        return edges, nodes, incoming_edges, outgoing_edges, source_pages,␣
    ↪target_pages
```

[27]:
```
def construct_sparse_matrix_8(data,d):
    """
    Construct a sparse matrix from the given connectivity data.
    (sparse matrix is used for memory efficiency)

    Parameters:
    - data: The path to the input data file/ Data file name.
```

```
    - d: A dictionary containing information about the top 10 nodes and 500␣
 ↪nodes below average.

    Returns:
    - sparse_matrix (sparse.csr_matrix): The constructed Sparse-CSR matrix -␣
 ↪representation of the graph.
    """
    edges, nodes, _, _, source_pages, target_pages = get_data_8(data,d)

    edge_weights = [1]*edges
    sparse_matrix = sparse.csr_matrix((edge_weights,
                                       (target_pages, source_pages)),
                                       shape=(nodes, nodes))
    return sparse_matrix
```

[28]:
```
G_8 = construct_sparse_matrix_8('stanweb.dat',d)
n = G_8.shape[0]

ranks_G_8 = PageRank_PowerMethod(G_8, 0.85, 1e-8)[0]
print(f'\nPageRank vector:\n\n {ranks_G_8}')

ranks_G_8 = np.asarray(ranks_G_8).ravel()
nodes_G_8 = ranks_G_8.argsort()[-n:][::-1]

print('\nRank of the new node:',list(nodes_G_8).index(281903))
```

```
PageRank vector:

 [[5.32285116e-07]
 [1.17295980e-04]
 [8.25921488e-07]
 …
 [1.80765246e-06]
 [1.47360002e-06]
 [1.05010089e-04]]

Rank of the new node: 806
```

0.2.11 Question 9. New Node with Links to Bottom 10 Nodes and In-Links from Top 10 Nodes

[29]:
```
G = construct_sparse_matrix('stanweb.dat')
n = G.shape[0]

ranks_G = PageRank_PowerMethod(G, 0.85, 1e-8)[0]
ranks_G = np.asarray(ranks_G).ravel()
nodes_G = ranks_G.argsort()[-n:][::-1]
```

```
list(nodes_G[-10:].ravel())

# Store the least and most important nodes from the original graph
d1={}
d1['least important nodes'] = list(nodes_G[-10:].ravel())
d1['most important nodes'] = list(nodes_G[:10].ravel())
```

[30]:
```
def get_data_9(data, d):
    """
    Extracts and processes data from the given file.

    Parameters:
    - data: The path to the data file/ Name of the file (str).
    - d: A dictionary containing information about the least and most important␣
 ↪10 nodes.



    Returns:
    - edges: The number of edges in the "graph".
    - nodes: The number of nodes in the "graph".
    - incoming_edges: Dictionary with webpages as keys and the count of their␣
 ↪outgoing edges as values.
    - outgoing_edges: Dictionary with webpages as keys and a list of incoming␣
 ↪edges as values.
    - source_pages: List of the origin pages.
    - target_pages: List of the destination pages.
    """
    source_pages, target_pages = [], []
    incoming_edges, outgoing_edges = {}, {}
    with open(data, 'r') as f:
        for line in f.readlines():
            source = int(line.split()[0]) - 1
            destination = int(line.split()[1]) - 1
            source_pages.append(source)
            target_pages.append(destination)

            if source not in outgoing_edges.keys():
                outgoing_edges[source] = 1
            else:
                outgoing_edges[source] += 1

            if destination not in incoming_edges.keys():
                incoming_edges[destination] = [source]
            else:
                incoming_edges[destination].append(source)
```

```
    for x in d['most important nodes']:
        source = x
        destination = 281903
        source_pages.append(source)
        target_pages.append(destination)

        if source not in outgoing_edges.keys():
            outgoing_edges[source] = 1
        else:
            outgoing_edges[source] += 1

        if destination not in incoming_edges.keys():
            incoming_edges[destination] = [source]
        else:
            incoming_edges[destination].append(source)

    for x in d['least important nodes']:
        source = 281903
        destination = x
        source_pages.append(source)
        target_pages.append(destination)

        if source not in outgoing_edges.keys():
            outgoing_edges[source] = 1
        else:
            outgoing_edges[source] += 1

        if destination not in incoming_edges.keys():
            incoming_edges[destination] = [source]
        else:
            incoming_edges[destination].append(source)

    edges = len(source_pages)
    nodes = len(set(source_pages) | set(target_pages))

    for node in range(nodes):
        if node not in incoming_edges.keys():
            incoming_edges[node] = []

    return edges, nodes, incoming_edges, outgoing_edges, source_pages,␣
↪target_pages
```

[31]:
```
def construct_sparse_matrix_9(data,d):
    """
    Construct a sparse matrix from the given connectivity data.
    (sparse matrix is used for memory efficiency)
```

```
    Parameters:
    - data: The path to the input data file/ Data file name.
    - d: A dictionary containing information about the least and most important␣
 ↪10 nodes.

    Returns:
    - sparse_matrix (sparse.csr_matrix): The constructed Sparse-CSR matrix -␣
 ↪representation of the graph.
    """
    edges, nodes, _, _, source_pages, target_pages = get_data_9(data,d)

    edge_weights = [1]*edges
    sparse_matrix = sparse.csr_matrix((edge_weights,
                                       (target_pages, source_pages)),
                                       shape=(nodes, nodes))
    return sparse_matrix
```

[32]:
```
G_9 = construct_sparse_matrix_9('stanweb.dat',d1)
n = G_9.shape[0]

ranks_G_9 = PageRank_PowerMethod(G_9, 0.85, 1e-8)[0]

ranks_G_9 = np.asarray(ranks_G_9).ravel()
nodes_G_9= ranks_G_9.argsort()[-n:][::-1]

print('\nRank of the new node:',list(nodes_G_9).index(281903))
```

```
Rank of the new node: 13
```

[33]:
```
print('\n rank of most important nodes 1:',list(nodes_G_9).index(d1['most␣
 ↪important nodes'][0]))
print('\n rank of most important nodes 2:',list(nodes_G_9).index(d1['most␣
 ↪important nodes'][1]))
print('\n rank of most important nodes 3:',list(nodes_G_9).index(d1['most␣
 ↪important nodes'][2]))
print('\n rank of most important nodes 4:',list(nodes_G_9).index(d1['most␣
 ↪important nodes'][3]))
print('\n rank of most important nodes 5:',list(nodes_G_9).index(d1['most␣
 ↪important nodes'][4]))
print('\n rank of most important nodes 6:',list(nodes_G_9).index(d1['most␣
 ↪important nodes'][5]))
print('\n rank of most important nodes 7:',list(nodes_G_9).index(d1['most␣
 ↪important nodes'][6]))
print('\n rank of most important nodes 8:',list(nodes_G_9).index(d1['most␣
 ↪important nodes'][7]))
```

```
print('\n rank of most important nodes 9:',list(nodes_G_9).index(d1['most␣
 ↪important nodes'][8]))
print('\n rank of most important nodes 10:',list(nodes_G_9).index(d1['most␣
 ↪important nodes'][9]))
```

```
 rank of most important nodes 1: 1

 rank of most important nodes 2: 0

 rank of most important nodes 3: 2

 rank of most important nodes 4: 5

 rank of most important nodes 5: 3

 rank of most important nodes 6: 4

 rank of most important nodes 7: 11

 rank of most important nodes 8: 12

 rank of most important nodes 9: 6

 rank of most important nodes 10: 9
```

[34]:
```
print('\n rank of least important node 1:',list(nodes_G_9).index(d1['least␣
 ↪important nodes'][0]))
print('\n rank of least important node 2:',list(nodes_G_9).index(d1['least␣
 ↪important nodes'][1]))
print('\n rank of least important node 3:',list(nodes_G_9).index(d1['least␣
 ↪important nodes'][2]))
print('\n rank of least important node 4:',list(nodes_G_9).index(d1['least␣
 ↪important nodes'][3]))
print('\n rank of least important node 5:',list(nodes_G_9).index(d1['least␣
 ↪important nodes'][4]))
print('\n rank of least important node 6:',list(nodes_G_9).index(d1['least␣
 ↪important nodes'][5]))
print('\n rank of least important node 7:',list(nodes_G_9).index(d1['least␣
 ↪important nodes'][6]))
print('\n rank of least important node 8:',list(nodes_G_9).index(d1['least␣
 ↪important nodes'][7]))
print('\n rank of least important node 9:',list(nodes_G_9).index(d1['least␣
 ↪important nodes'][8]))
print('\n rank of least important node 10:',list(nodes_G_9).index(d1['least␣
 ↪important nodes'][9]))
```

```
rank of least important node 1: 408

rank of least important node 2: 409

rank of least important node 3: 407

rank of least important node 4: 403

rank of least important node 5: 401

rank of least important node 6: 402

rank of least important node 7: 405

rank of least important node 8: 406

rank of least important node 9: 404

rank of least important node 10: 410
```

0.2.12 PageRank Optimization Problem

In this problem, we delve into a challenging scenario faced by a company aiming to boost its website's prominence in Google searches by reaching the top 1% of the Pagerank ranking.

The primary challenge revolves around two key strategies:

1. Linking to High Pagerank Sites. Cost: (285000 - i + 1)^2 cents for each link added to a page with a score of i.

2. Creating New Satellite Sites. Cost: 100 euros per page for establishing new satellite sites.

Our objective is to analyze and propose a solution that strikes a balance between effectiveness and cost-efficiency.

0.2.13 First Strategy

We randomly selected 1000 pages with a rank near the mean and linked our website to one of these selected pages. If we didn't achieve the desired result, we continued to increase the number of linked pages until we reached a rank of 1%. This process required us to link to a total of 36 nodes for 2,892,276,729,591 cents. However, it's important to note that the choice of higher-ranked pages was random and this method can be quite costly.

[35]:
```
# Load the graph and compute PageRank
G = construct_sparse_matrix('stanweb.dat')
n = G.shape[0]
ranks = PageRank_PowerMethod(G, 0.85, 1e-8)[0]
ranks = np.asarray(ranks).ravel()
nodes_alpha_85 = ranks.argsort()[-n:][::-1]
```

```
# Select candidate nodes based on proximity to the mean PageRank value
chosen_nodes = []
for x in range(0, 1000):
    chosen_nodes.append(list(ranks).index(ranks[list(ranks<(np.mean(ranks)+np.
 ↪mean(ranks)/10)) and list(ranks>(np.mean(ranks)-np.mean(ranks)/10))][x]))

chosen_nodes=list(set(chosen_nodes))[:36]

# Create a dictionary to store "candidate" nodes
d2={}
d2['candidate'] = chosen_nodes
```

[36]:
```
def get_data_strategy_1(data, d):
    """
    Extracts and processes data from the given file, incorporating link␣
 ↪additions for the first strategy.

    Parameters:
    - data: The path to the data file/ Name of the file (str).
    - d: A dictionary containing information about the nodes selected for link␣
 ↪additions.

    Returns:
    - edges: The number of edges in the "graph".
    - nodes: The number of nodes in the "graph".
    - incoming_edges: Dictionary with webpages as keys and the count of their␣
 ↪outgoing edges as values.
    - outgoing_edges: Dictionary with webpages as keys and a list of incoming␣
 ↪edges as values.
    - source_pages: List of the origin pages.
    - target_pages: List of the destination pages.
    """
    source_pages, target_pages = [], []
    incoming_edges, outgoing_edges = {}, {}

    with open(data, 'r') as f:
        for line in f.readlines():
            source = int(line.split()[0]) - 1
            destination = int(line.split()[1]) - 1

            source_pages.append(source)
            target_pages.append(destination)

            if source not in outgoing_edges.keys():
                outgoing_edges[source] = 1
            else:
```

```
                outgoing_edges[source] += 1

            if destination not in incoming_edges.keys():
                incoming_edges[destination] = [source]
            else:
                incoming_edges[destination].append(source)

    for x in d['candidate']:
        source = x
        destination = 281903 # Our Website (Last Node - Website)
        source_pages.append(source)
        target_pages.append(destination)

        if source not in outgoing_edges.keys():
            outgoing_edges[source] = 1
        else:
            outgoing_edges[source] += 1

        if destination not in incoming_edges.keys():
            incoming_edges[destination] = [source]
        else:
            incoming_edges[destination].append(source)


    edges = len(source_pages)
    nodes = len(set(source_pages) | set(target_pages))

    for node in range(nodes):
        if node not in incoming_edges.keys():
            incoming_edges[node] = []

    # We only need the edges and nodes of the function to implement the strategy
    # We compute outgoing and incoming edges dictionaries for potential further␣
↪analysis
    return edges, nodes, incoming_edges, outgoing_edges, source_pages,␣
↪target_pages
```

[37]:
```
def construct_sparse_matrix_strategy_1(data,d):
    """
    Construct a sparse matrix from the given connectivity data.
    (sparse matrix is used for memory efficiency)

    Parameters:
    - data: The path to the input data file/ Data file name.
    - d: A dictionary containing information about the nodes selected for link␣
↪additions.
```

```
    Returns:
    - sparse_matrix (sparse.csr_matrix): The constructed Sparse-CSR matrix -␣
 ↪representation of the graph.
    """
    edges, nodes, _, _, source_pages, target_pages = get_data_strategy_1(data,d)

    edge_weights = [1]*edges
    sparse_matrix = sparse.csr_matrix((edge_weights,
                                       (target_pages, source_pages)),
                                       shape=(nodes, nodes))
    return sparse_matrix
```

[38]:
```
G_S1 = construct_sparse_matrix_strategy_1('stanweb.dat',d2)
n = G_S1.shape[0]

ranks_S1 = PageRank_PowerMethod(G_S1, 0.85, 1e-8)[0]
ranks_S1 = np.asarray(ranks_S1).ravel()
nodes_S1 = ranks_S1.argsort()[-n:][::-1]

print('\nRank of the Website:',list(nodes_S1).index(281903))
percentage_S1 = (list(nodes_S1).index(281903)/281903)*100
percentage_S1 = format(percentage_S1,'.2f')
print('\nOur Website is at the Top', percentage_S1,'% in PageRank Ranking.')
cost = 0
for x in chosen_nodes:
    cost += (285000 - x + 1)**2
print('\nTotal cost for the first strategy:', cost,'cents')
```

```
Rank of the Website: 2802

Our Website is at the Top 0.99 % in PageRank Ranking.

Total cost for the first strategy: 2892276729591 cents
```

0.2.14 Second Strategy

We created 75 new sites without using the first variation (If we didn't initially achieve the desired result, we continued to create additional sites until reaching the top 1%). We reached the objective for only 7500 euros.

[39]:
```
# Create a dictionary to store new nodes
d={}
number_new_nodes = 75
d['new node'] = []
for i in range(number_new_nodes):
    d['new node'].append(281903 + i + 1)
```

[40]:
```
def get_data_strategy_2(data,d):
    """
    Extracts and processes data from the given file, incorporating new nodes␣
 ↪for the second strategy.

    Parameters:
    - data: The path to the data file/ Name of the file (str).
    - d: A dictionary containing information about the new nodes.

    Returns:
    - edges: The number of edges in the "graph".
    - nodes: The number of nodes in the "graph".
    - incoming_edges: Dictionary with webpages as keys and the count of their␣
 ↪outgoing edges as values.
    - outgoing_edges: Dictionary with webpages as keys and a list of incoming␣
 ↪edges as values.
    - source_pages: List of the origin pages.
    - target_pages: List of the destination pages.
    """
    source_pages, target_pages = [], []
    incoming_edges, outgoing_edges = {}, {}

    with open(data, 'r') as f:
        for line in f.readlines():
            source = int(line.split()[0]) - 1
            destination = int(line.split()[1]) - 1

            source_pages.append(source)
            target_pages.append(destination)

            if source not in outgoing_edges.keys():
                outgoing_edges[source] = 1
            else:
                outgoing_edges[source] += 1

            if destination not in incoming_edges.keys():
                incoming_edges[destination] = [source]
            else:
                incoming_edges[destination].append(source)

    for x in d['new node']:
        source = x
        destination = 281903 # Our Website
        source_pages.append(source)
        target_pages.append(destination)

        if source not in outgoing_edges.keys():
```

```
            outgoing_edges[source] = 1
        else:
            outgoing_edges[source] += 1

        if destination not in incoming_edges.keys():
            incoming_edges[destination] = [source]
        else:
            incoming_edges[destination].append(source)

    edges = len(source_pages)
    nodes = len(set(source_pages) | set(target_pages))

    for node in range(nodes):
        if node not in incoming_edges.keys():
            incoming_edges[node] = []

    return edges, nodes, incoming_edges, outgoing_edges, source_pages,␣
↪target_pages

def construct_sparse_matrix_strategy_2(data,d):
    """
    Construct a sparse matrix from the given connectivity data.
    (sparse matrix is used for memory efficiency)

    Parameters:
    - data: The path to the input data file/ Data file name.
    - d: A dictionary containing information about the new nodes.

    Returns:
    - sparse_matrix (sparse.csr_matrix): The constructed Sparse-CSR matrix -␣
↪representation of the graph.
    """
    edges, nodes, _, _, source_pages, target_pages = get_data_strategy_2(data,d)

    edge_weights = [1]*edges
    sparse_matrix = sparse.csr_matrix((edge_weights,
                                       (target_pages, source_pages)),
                                       shape=(nodes, nodes))
    return sparse_matrix
```

[41]:
```
G_S2 = construct_sparse_matrix_strategy_2('stanweb.dat',d)
n = G_S2.shape[0]
ranks_S2 = PageRank_PowerMethod(G_S2, 0.85, 1e-8)[0]
ranks_S2 = np.asarray(ranks_S2).ravel()
nodes_S2 = ranks_S2.argsort()[-n:][::-1]

print('\nRank of the Website:',list(nodes_S2).index(281903))
```

```
percentage_S2 = (list(nodes_S2).index(281903)/281903)*100
percentage_S2 = format(percentage_S2,'.2f')
print('\nOur Website is at the Top', percentage_S2,'% in PageRank Ranking.')
print('\nTotal cost for the second strategy:', number_new_nodes * 100,'euros')
```

```
Rank of the Website: 2803

Our Website is at the Top 0.99 % in PageRank Ranking.

Total cost for the second strategy: 7500 euros
```

0.2.15 Conclusion and Recommended Strategy

After evaluating the cost-effectiveness of two strategies to boost the company's website to the top 1% of the Pagerank ranking, we found the following:

Linking to High Pagerank Sites: - Total Cost: 2,892,276,729,591 cents.

Creating New Affiliated Nodes: - Total Cost: 7,500 euros.

The analysis indicates that the second approach, creating new affiliated nodes, is more cost-effective. This approach aims to achieve the desired ranking while minimizing costs.

Further adjustments can be made based on budget constraints and specific goals.

(Also, a hybrid approach that combines linking to high Pagerank sites and creating new nodes/sites could also be used)