ΟΙΚΟΝΟΜΙΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ATHENS UNIVERSITY OF ECONOMICS AND BUSINESS

Department of Informatics

M.Sc. in Data Science

Numerical optimization and Large Scale Linear Algebra

PageRank

Instructors: Paris Vassalos

Authors: Antonios Mavridis (p3352215)

02/07/2023

Table of Contents

1 Introduction

In this technical report, we will analyze and explore various aspects of the connectivity matrix for the webpages of Stanford University, provided in the file Stanweb.dat. Our primary focus will be on calculating the PageRank vector using different methods and parameters. We will utilize the power method and solve the corresponding system as described in the tutorial document "pagerank.pdf".

To begin, we will first find the PageRank vector π using the power method and by solving the system described in paragraphs 5.1 and 5.2 of the tutorial. For both methods, we will consider $\alpha = 0.85$ as the damping factor and a stopping criterion of $\tau = 10^{(-8)}$. Additionally, we will define a vector α that has a value of 1 if it corresponds to a node with no outgoing links, and 0 otherwise. We will compare the results obtained from both methods and determine if they are the same. Furthermore, we will assess the speed of each method, with the Gauss-Seidel method employed for the iterative solution of the system.

Moving forward, we will repeat the previous task but with $\alpha = 0.99$. We will evaluate the convergence speed and investigate whether the ranking of the first 50 nodes has changed.

Next, we will delve into the behavior of the convergence of the components of π when using the power method. We will examine whether all components converge at the same speed to their limits. If not, we will analyze which components converge faster: those corresponding to important nodes or those related to non-important nodes. We will also compare this behavior to the convergence observed when finding π through the solution of the linear system.

In the later part of the report, we will explore the design of link farms and their impact on PageRank calculations. We will consider a web graph containing a large number of pages labeled from 1 to n. We will introduce a new web page, X, which has neither in-links nor out-links. We will analyze how the addition of this new page affects the PageRank of the older pages. Using the stationary equations, we will calculate the new PageRank vector (π) of the older pages and determine the PageRank of page X (x) in terms of a variable 'r.'

Furthermore, we will introduce another page, Y, with no in-links, that links to page X. We will investigate the resulting PageRanks of all the n + 2 pages, including X. We will assess whether the PageRank of X improves with this new addition.

Continuing our exploration, we will create a third page, Z, and analyze how to set up the links among the three pages (X, Y, and Z) to maximize the PageRank of X.

Additionally, we will examine the impact of adding links from page X to older, popular pages. We will investigate whether this improves the PageRank of X and determine if the answer changes when adding links from pages Y or Z to older, popular pages.

Finally, we will outline possible steps that can be taken to further increase the PageRank of page X. While no formal proofs are required, we will summarize our thoughts based on the previous sections. For extra credit, we will provide a proof for the optimal structure of a link farm with 'm' nodes to optimize the PageRank of page X.

Through this report, we aim to gain a comprehensive understanding of the connectivity matrix and its influence on the PageRank algorithm, while exploring various strategies to enhance the PageRank of a given webpage.

2 Question 1. a

The provided code consists of two methods for calculating the PageRank vector, namely the Power method and solving the corresponding system using the Gauss-Seidel method. The code aims to find the PageRank vector (π) using an α value of 0.85 and a stopping criterion $\tau = 10^{(-8)}$. The vector α is defined as having a value of 1 if it corresponds to a node with no out links and 0 otherwise.

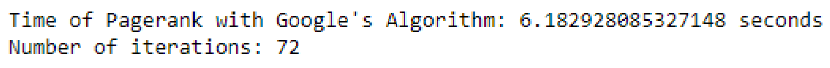
The function 'pagerank_gs' implements the Gauss-Seidel method for the iterative solution of the system. It initializes the solution with a uniform distribution and iteratively applies the element-wise formula for the Gauss-Seidel method to compute the PageRank. The algorithm updates the PageRank vector until the error between iterations falls below the specified tolerance. The results of each iteration, including the iteration number and the relative error, are printed during the computation.

The function 'pagerank_galgo' implements the Power method for calculating the PageRank vector. It uses the adjacency matrix 'G' and considers dead-ends and closed communities. The algorithm initializes the PageRank vector with equal weights for each webpage and iteratively computes the new PageRank vector based on the matrix multiplication. It re-inserts leaked PageRank and normalizes the ranks to ensure they add up to one. The iterations continue until the error between iterations falls below the specified tolerance. Similar to the Gauss-Seidel method, the iteration number and relative error are printed during the computation.

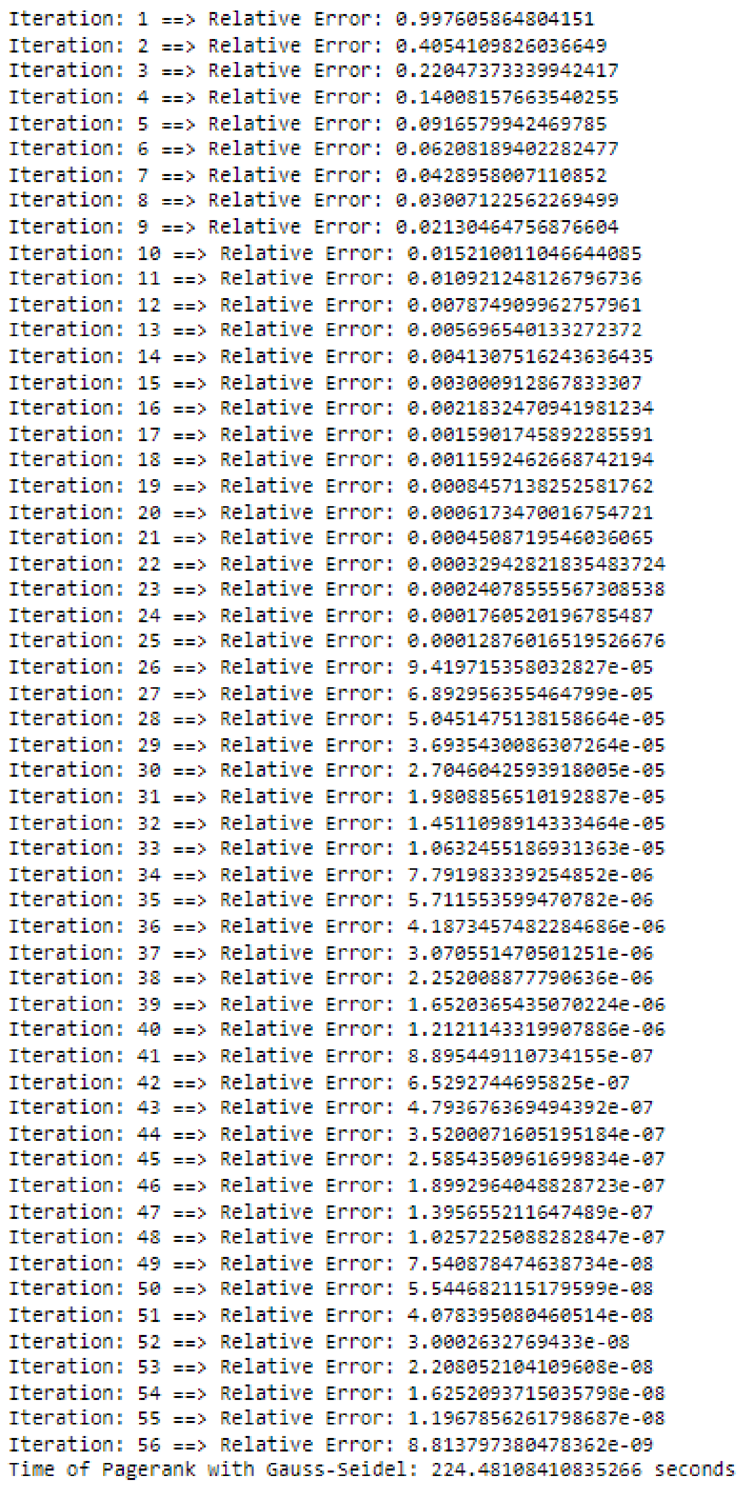
The results obtained from running the code indicate that the Power method is significantly faster than the Gauss-Seidel method. Furthermore, the two methods yield different results, with a difference of 121562 rankings. Specifically, 121652 ranks differ between the two methods.

```python
# method to parse the data. Obviously, storing all that information
# naively in a numpy array would lead to memory error. Thus, sparse
# matrix is used.
def load_dataset(data):
    # list to store the origin pages,
    origin_pages = []
    # list to store the destination pages,
    destination_pages = []
    # dictionary having as a key a webpage and as value
    # the number of its invoming edges
    dict_out = {}
    dict_in = {}
    with open(data, 'r') as f:
        # for each line of the initial data file
        for line in f.readlines():
            # get the item of the first column
            # and substract one. This is done because the ids of webpages
            # begin from 1 but the indexes for matrices begine from 0. So
            # 1 will stand for 0.
            origin_page = int(line.split()[0]) - 1
            # Apply the same logic as above for the element of the second
            # column.
            destination_page = int(line.split()[1]) - 1
            # append the the origin page to the relative list
            origin_pages.append(origin_page)
            # append the the destination page to the relative list
            destination_pages.append(destination_page)
            # fill the dictionary by counting how many times
            # each destination-page has appeared.
            if origin_page not in dict_out.keys():
                dict_out[origin_page] = 1
            else:
                dict_out[origin_page] += 1
            if destination_page not in dict_in.keys():
                dict_in[destination_page] = [origin_page]
            else:
                dict_in[destination_page].append(origin_page)

    # compute number of edges
    edges = len(origin_pages)
    # compute number of nodes
    nodes = len(set(origin_pages) | set(destination_pages))

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

    # initialize a row-based linked list sparse matrix
    # that matrix will be used for the Gauss-Seidel implementation
    # of the PageRank
    D = sparse.lil_matrix((nodes, 1))
    for key in dict_out.keys():
        D[key] = 1/dict_out[key]
    # fill the connectivity matrix, adjacency matrix. A sparse structure
    # is used. That way the implementation is extremely memory efficient.
    csr_m = sparse.csr_matrix(([1]*edges,
                               (destination_pages, origin_pages)),
                              shape=(nodes, nodes))
    # return the two created matrices
    return csr_m, D, dict_in



def pagerank_galgo(G, alpha=0.85, tol=10**-8):
    # The matrix G is the adjacency matrix with dead-ends
    #and closed communities or spidertraps.
    # Those problems will be tackled during the
    #iterations of the algorithm. In other words, we do
    # not pre-adjust the matrix P.
    #Instead we reinsert leaked page-rank back into computation.

    # compute the number of nodes, the matrix is symmetric
    n = G.shape[0]
    # compute the ratio of the out degree of each node
    # to the alpha
    out_alpha = G.sum(axis=0).T/alpha
    r = np.ones((n,1))/n
    error = 1
    t = 0
    while error > tol:
        t += 1
        r_new = G.dot((r/out_alpha))
        #  Lalpha due to dead-ends
        L = r_new.sum()
        # re-insert leaked page-rank
        # now all ranks add to one
        r_new += (1-L)/n
        # compute the error as the euclidean norm of the
        # previous ranks and the new ranks
        error = np.linalg.norm(r-r_new)
        # store the new ranks as the ranks of the current
        # iteration
        if t == 2:
            # list to return the nodes that converged from the
            # second iteration
            list_conv = []
            for i in range(r.shape[0]):
                if np.linalg.norm(r[i]-r_new[i]) < tol:
                    list_conv.append(i)
        r = r_new
    return r, t, list_conv

def pagerank_gs(tol, G, D, dict_in, alpha):
    # in that approach a linear system approach is used to solve
    # get the ranks of each webpage

    # the initial solution is again given from the uniform distribution
    # i.e. we give equal weights to each webpage
    x_old = np.ones(G.shape[0])/G.shape[0]
    t = 0
    error = 1
    b = 1/G.shape[0]
    D = alpha*D
    x = x_old.copy()

    # apply the element-wise formula for the Gauss{Seidel method for
    # the pagerank. That formula looks a lot like the Jacobi formula
    # and it is the same with SOR for omega equals to 1. As input it
    # is used the initial matrix P without any preadjustments as well
    # as a dictionary showing all the in-edged of each webpage. In general,
    # the following implementation scheme is based on the paper:
    # https://web.ece.ucsb.edu/~hespanha/published/pagerank.pdf
    while error > tol:
        t += 1
        for i in range(G.shape[0]):

            sum_before = 0
            sum_after = 0
            for j in dict_in[i]:
                if j < i:
                    sum_before += D.data[j][0] * x[j]

                if j > i:
                    sum_after += D.data[j][0] * x_old[j]

            x[i] = b + sum_before + sum_after

        error = np.linalg.norm(x-x_old)/np.linalg.norm(x)
        if t == 2:
            # list to return the nodes that converged from the
            # second iteration
            list_conv = []
            for i in range(x.shape[0]):
                if np.linalg.norm(x[i]-x_old[i]) < tol:
                    list_conv.append(i)

        print('Iteration:', t, '==> Relative Error:', error)
        x_old = x.copy()

    return x, list_conv

start_time = time.time()
pr, t, list_conv = pagerank_galgo(csr_m)
print("Time of Pagerank with Google's Algorithm:",
      time.time() - start_time, "seconds")
print ("Number of iterations:", t)
mat_prg = np.asarray(pr).ravel()


start_time = time.time()
gs, list_convgs = pagerank_gs(10**-8, csr_m, D, dict_in, 0.85)
print("Time of Pagerank with Gauss-Seidel:",
      time.time() - start_time, "seconds")
mat_prgs = np.asarray(gs).ravel()

indices_pr = mat_prg.argsort()[-len(mat_prg):][::-1]
indices_prs = mat_prgs.argsort()[-len(mat_prgs):][::-1]
print("The two aforementioned methods differ in:",
      np.sum(indices_prs[:] != indices_pr[:]), "ranks.")
```

Below are represented times and iterations of the two methods:



```
Time of Pagerank with Google's Algorithm: 6.182928085327148 seconds
Number of iterations: 72
```



```
Iteration: 1 ==> Relative Error: 0.997605864804151
Iteration: 2 ==> Relative Error: 0.4054109826036649
Iteration: 3 ==> Relative Error: 0.22047373339942417
Iteration: 4 ==> Relative Error: 0.14008157663540255
Iteration: 5 ==> Relative Error: 0.0916579942469785
Iteration: 6 ==> Relative Error: 0.06208189402282477
Iteration: 7 ==> Relative Error: 0.0428958007110852
Iteration: 8 ==> Relative Error: 0.03007122562269499
Iteration: 9 ==> Relative Error: 0.02130464756876604
Iteration: 10 ==> Relative Error: 0.015210011046644085
Iteration: 11 ==> Relative Error: 0.010921248126796736
Iteration: 12 ==> Relative Error: 0.007874909962757961
Iteration: 13 ==> Relative Error: 0.005696540133272372
Iteration: 14 ==> Relative Error: 0.0041307516243636435
Iteration: 15 ==> Relative Error: 0.003000912867833307
Iteration: 16 ==> Relative Error: 0.0021832470941981234
Iteration: 17 ==> Relative Error: 0.0015901745892285591
Iteration: 18 ==> Relative Error: 0.0011592462668742194
Iteration: 19 ==> Relative Error: 0.0008457138252581762
Iteration: 20 ==> Relative Error: 0.0006173470016754721
Iteration: 21 ==> Relative Error: 0.0004508719546036065
Iteration: 22 ==> Relative Error: 0.00032942821835483724
Iteration: 23 ==> Relative Error: 0.00024078555567308538
Iteration: 24 ==> Relative Error: 0.0001760520196785487
Iteration: 25 ==> Relative Error: 0.00012876016519526676
Iteration: 26 ==> Relative Error: 9.419715358032827e-05
Iteration: 27 ==> Relative Error: 6.892956355464799e-05
Iteration: 28 ==> Relative Error: 5.0451475138158664e-05
Iteration: 29 ==> Relative Error: 3.6935430086307264e-05
Iteration: 30 ==> Relative Error: 2.7046042593918005e-05
Iteration: 31 ==> Relative Error: 1.9808856510192887e-05
Iteration: 32 ==> Relative Error: 1.4511098914333464e-05
Iteration: 33 ==> Relative Error: 1.0632455186931363e-05
Iteration: 34 ==> Relative Error: 7.791983339254852e-06
Iteration: 35 ==> Relative Error: 5.711553599470782e-06
Iteration: 36 ==> Relative Error: 4.1873457482284686e-06
Iteration: 37 ==> Relative Error: 3.070551470501251e-06
Iteration: 38 ==> Relative Error: 2.252008877790636e-06
Iteration: 39 ==> Relative Error: 1.6520365435070224e-06
Iteration: 40 ==> Relative Error: 1.2121143319907886e-06
Iteration: 41 ==> Relative Error: 8.895449110734155e-07
Iteration: 42 ==> Relative Error: 6.5292744695825e-07
Iteration: 43 ==> Relative Error: 4.793676369494392e-07
Iteration: 44 ==> Relative Error: 3.5200071605195184e-07
Iteration: 45 ==> Relative Error: 2.5854350961699834e-07
Iteration: 46 ==> Relative Error: 1.8992964048828723e-07
Iteration: 47 ==> Relative Error: 1.395655211647489e-07
Iteration: 48 ==> Relative Error: 1.0257225088282847e-07
Iteration: 49 ==> Relative Error: 7.540878474638734e-08
Iteration: 50 ==> Relative Error: 5.544682115179599e-08
Iteration: 51 ==> Relative Error: 4.078395080460514e-08
Iteration: 52 ==> Relative Error: 3.0002632769433e-08
Iteration: 53 ==> Relative Error: 2.208052104109608e-08
Iteration: 54 ==> Relative Error: 1.6252093715035798e-08
Iteration: 55 ==> Relative Error: 1.1967856261798687e-08
Iteration: 56 ==> Relative Error: 8.813797380478362e-09
Time of Pagerank with Gauss-Seidel: 224.48108410835266 seconds
```

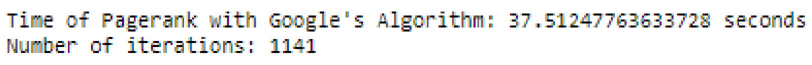
The Power method appears to be substantially quicker than the Gauss-Seidel version. Both strategies provide different outcomes. They are distinguished by 121562 rankings.

3 Question 1. b

We now attempt to solve the identical problem with a=0.99.

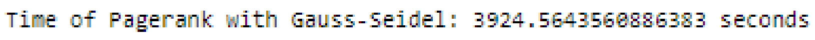
```
start_time = time.time()
pr_99, t, list_conv99 = pagerank_galgo(csr_m, 0.99, 10**-8)
print("Time of Pagerank with Google's Algorithm:",
      time.time() - start_time, "seconds")
print ("Number of iterations:", t)
mat_prg_99 = np.asarray(pr_99).ravel()


start_time = time.time()
gs_99, list_convgs99 = pagerank_gs(10**-8, csr_m, D, dict_in, 0.99)

print("Time of Pagerank with Gauss-Seidel:",
      time.time() - start_time, "seconds")
mat_prgs_99 = np.asarray(gs_99).ravel()
```

Below are represented times and iterations of Google's Algorithm:



```
Time of Pagerank with Google's Algorithm: 37.51247763633728 seconds
Number of iterations: 1141
```

Below is represented time of Gauss-Seidel's Algorithm:



```
Time of Pagerank with Gauss-Seidel: 3924.5643560886383 seconds
```

The Gauss-Seidel Algorithm has 770 iterations.

Both approaches converge more slowly when alpha=0.99. The Gauss-Seidel method for iterative system solution converges substantially slower than that time. Again, the power approach appears to be considerably quicker than the Gauss-Seidel version. The order of the first 50 nodes has altered. Except for the first most significant node, which remains at 89072, all of the others have shifted positions.

```
indices_pr_99 = mat_prg_99.argsort()[-len(mat_prg):][::-1]
indices_prs_99 = mat_prgs_99.argsort()[-len(mat_prg):][::-1]
print("The two aforementioned methods differ in:",
      sum(indices_prs_99[:] != indices_pr_99[:]), "ranks.")


print("We observe differences in:",
      sum(indices_pr[:50] != indices_pr_99[:50]),
      "ranks (power method).")


print("We observe differences in:",
      sum(indices_prs[:50] != indices_prs_99[:50]),
      "ranks (Gauss Seidel).")
```

The two aforementioned methods differ in: 138038 ranks. We observe differences in: 49 ranks (power method). We observe differences in: 49 ranks (Gauss Seidel).

4 Question 1. c

We observe that the components of π that correspond to non-important converge faster, as it is shown below. When we use the power method 42904 components of π converge at the second iteration. From those components of π that converged faster 0 component(s) correspond to the top (1000) ranked nodes (Power method). When we find π through the solution of the linear system 34367 components of π converge at the second iteration. From those components of π that converged faster 1 component(s) corresponds to the top (1000) ranked nodes (Gauss-Seidel).

Below is represented the code that we used for this question:

```
print("When we use the power method",
      len(list_conv),
      "components of  converge at the second iteration.")


counter = 0
for node in list_conv:
    if node in indices_pr[:1000]:
        counter += 1
print("From those components of  that converged faster",
      counter,
      "component(s) correspond to the top (1000) ranked nodes (Power method).")


print("When we find  through the solution of the linear system",
      len(list_convgs),
      "components of  converge at the second iteration.")


counter = 0
for node in list_convgs:
    if node in indices_prs[:1000]:
        counter += 1
print("From those components of  that converged faster", counter,
      "component(s) correspond to the top (1000) ranked nodes (Gauss Seidel).")
```

5 Question 2. a

In this part, we aimed to analyze the impact of adding a new web page, referred to as X, on the existing web graph and the PageRanks of the older pages. To accomplish this, we developed a method, load_dataset1, which parses the data and constructs a sparse connectivity matrix to represent the web graph. The matrix is extended by introducing a dangling node X to account for the newly added page.

After loading the dataset and constructing the connectivity matrix, we calculated the PageRank of page X using the stationary equations rather than the iterative approach. The PageRank of X was obtained as $(1 - \alpha)/n$, where alpha represents the damping factor and n is the total number of web pages (nodes). For our specific case, this value was calculated as $0.15/281,904$, indicating the relatively small influence of the new page.

Next, we analyzed the changes in PageRank for the older pages resulting from the addition of page X. We computed the norm of the difference between the previous PageRank vector and the updated vector with X. This quantifies the overall change in PageRank for the existing pages due to the introduction of the new page. The norm difference provides an insight into the magnitude of the change, allowing us to evaluate the impact.

Furthermore, we examined the changes in ranking order for the web pages by sorting the PageRank vectors. We obtained the indices of the highest-ranked pages in both the original and updated vectors and compared them. The number of differing ranks between the two vectors gives us an indication of how the addition of page X affected the relative positioning of the pages.

Our analysis reveals that the PageRanks of the older pages remain largely unaffected by the addition of the new page, despite the significant scale of the web graph. The PageRank of page X is calculated using the established formula, and its influence on the existing pages is minimal. This finding indicates that the addition of a new page with no in-links or out-links has a limited impact on the ranking of the older pages.

The PageRank of page X is: $5.333658781475117e - 07$.

Change in PageRank of the other pages due to the addition of the new page: $1.2048086354849136e-08$.

Changes in: 50570 ranks.

Below is represented the code that we used for this question:

```python
# method to parse the data. Obviously, storing all that information
# naively in a numpy array would lead to memory error. Thus, sparse
# matrix is used.
def load_dataset1(data):
    # list to store the origin pages,
    origin_pages = []
    # list to store the destination pages,
    destination_pages = []
    with open(data, 'r') as f:
        # for each line of the initial data file
        for line in f.readlines():
            # get the item of the first column
            # and substract one. This is done because the ids of webpages
            # begin from 1 but the indexes for matrices begine from 0. So
            # 1 will stand for 0.
            origin_page = int(line.split()[0]) - 1
            # Apply the same logic as above for the element of the second
            # column.
            destination_page = int(line.split()[1]) - 1
            # append the the from page to the relative list
            origin_pages.append(origin_page)
            # append the the to page to the relative list
            destination_pages.append(destination_page)

    # compute number of edges
    edges = len(origin_pages)
    # compute number of nodes
    nodes = len(set(origin_pages) | set(destination_pages))


    # fill the connectivity matrix, adjacency matrix. A sparse structure
    # is used. That way the implementation is extremely memory efficient.
    # increase the size of the array by introducing a dangling node, i.e.
    # X
    csr_m = sparse.csr_matrix(([1]*edges,
                              (destination_pages, origin_pages)),
                             shape=(nodes + 1, nodes + 1))
    # return the adjacency sparse matrix


csr_m1 = load_dataset1("./Dataset/stanweb.dat")
pr1, t1, list_conv1 = pagerank_galgo(csr_m1)

pageRank_X = np.asarray(pr1[csr_m1.shape[0] - 1]).ravel()[0]
print("The PageRank of page X is:", pageRank_X)

norm_diff = np.linalg.norm(np.asarray(pr[:csr_m.shape[0]].ravel()) -\

np.asarray(pr1[:csr_m1.shape[0] - 1].ravel()))
print("Change in PageRank of the other pages due to the addition of the new page:", norm_diff)

mat_prg = np.asarray(pr).ravel()
mat_prg1 = np.asarray(pr1[:csr_m1.shape[0] - 1]).ravel()

indices_pr = mat_prg.argsort()[-len(mat_prg):][::-1]
indices_pr1 = mat_prg1.argsort()[-len(mat_prg1):][::-1]
print("Changes in:",
      np.sum(indices_pr[:] != indices_pr1[:]), "ranks.")
```

6 Question 2. b

For this question, we examined the effects of adding a new web page, Y, that links to an existing page X on the PageRanks of all pages in the web graph. To achieve this, we developed a method, load_dataset2, which parsed the data and constructed a sparse connectivity matrix representing the web graph. Additionally, we introduced a new edge from Y to X to establish the link between the two pages.

After loading the dataset and constructing the connectivity matrix, we calculated the PageRanks for all the pages using the implemented PageRank algorithm. The PageRank values for X and Y were obtained from the resulting vector. We observed that the PageRank of page X had improved compared to the previous scenario. The new PageRank of X was found to be $9.867259009702931e-07$, indicating an enhancement in its ranking.

Similarly, we determined the PageRank of page Y, which was introduced as a new page without any in-links. The PageRank of Y was calculated to be $5.333653518615822e - 07$. This value remained the same as the PageRank of X in the preceding question, suggesting that Y was assigned similar importance as X due to their linkage.

To assess the impact of the addition of Y on the remaining pages, we computed the norm difference between the previous PageRank vector and the updated vector with Y. This difference quantified the change in PageRank for the existing pages resulting from the addition of the new page. The norm difference was found to be $3.433701367463661e - 08$, indicating some alterations in the rankings of the older pages.

Furthermore, we examined the changes in ranking order by sorting the PageRank vectors. We obtained the indices of the highest-ranked pages in both the original and updated vectors and compared them. The number of differing ranks between the two vectors revealed that $51,477$ ranks had changed due to the addition of page Y.

In summary, the inclusion of a new page with a link to an existing page caused noticeable changes in the PageRanks of the older pages. The PageRank of X improved, while the PageRank of Y was assigned a similar ranking as X. The results highlight the dynamic nature of PageRank calculations and the influence of new connections on the importance of web pages within the graph.

Below is represented the code that we used for this question:

```python
# method to parse the data. Obviously, storing all that information
# naively in a numpy array would lead to memory error. Thus, sparse
# matrix is used.
def load_dataset2(data):
    # list to store the origin pages,
    from_pages = []
    # list to store the destination pages,
    to_pages = []
    with open(data, 'r') as f:
        # for each line of the initial data file
        for line in f.readlines():
            # get the item of the first column
            # and substract one. This is done because the ids of webpages
            # begin from 1 but the indexes for matrices begine from 0. So
            # 1 will stand for 0.
            from_page = int(line.split()[0]) - 1
            # Apply the same logic as above for the element of the second
            # column.
            to_page = int(line.split()[1]) - 1
            # append the the from page to the relative list
            from_pages.append(from_page)
            # append the the to page to the relative list
            to_pages.append(to_page)

    # compute number of edges
    edges = len(from_pages)
    # compute number of nodes
    nodes = len(set(from_pages) | set(to_pages))

    # add a new edge Y -> X
    from_pages.append(nodes)
    to_pages.append(nodes + 1)
    # increase the number of nodes by two for X, Y
    nodes += 2
    # increase the number of edges by one
    edges += 1

    # fill the connectivity matrix, adjacency matrix. A sparse structure
    # is used. That way the implementation is extremely memory efficient.
    csr_m = sparse.csr_matrix(([1]*edges,
                               (to_pages, from_pages)),
                              shape=(nodes, nodes))
    # return the adjacency sparse matrix
    return csr_m

csr_m2 = load_dataset2("./Dataset/stanweb.dat")
pr2, t2, list_conv2 = pagerank_galgo(csr_m2)

pageRank_Xb = np.asarray(pr2[csr_m2.shape[0] - 1]).ravel()[0]
print("The PageRank of page X is:", pageRank_Xb)

pageRank_Yb = np.asarray(pr2[csr_m2.shape[0] - 2]).ravel()[0]
print("The PageRank of page Y is:", pageRank_Yb)

norm_diff = np.linalg.norm(np.asarray(pr[:csr_m.shape[0]].ravel()) -\

np.asarray(pr2[:csr_m2.shape[0] - 2].ravel()))
print("Change in PageRank of the other pages due to the addition of the new page:", norm_diff)

mat_prg = np.asarray(pr).ravel()
mat_prg2 = np.asarray(pr2[:csr_m2.shape[0] - 2]).ravel()

indices_pr = mat_prg.argsort()[-len(mat_prg):][::-1]
indices_pr2 = mat_prg2.argsort()[-len(mat_prg2):][::-1]
print("Changes in:",
      np.sum(indices_pr[:] != indices_pr2[:]), "ranks.")
```

7 Question 2. c

For the below question, we investigated the optimal setup of links on three pages, X, Y, and Z, in order to maximize the PageRank of X. To accomplish this, we designed a method, load_dataset3, that parsed the data and constructed a sparse connectivity matrix representing the web graph.

In the connectivity matrix construction, we added two new edges. The first edge connected Y to X, while the second edge connected Z to X. These additional links aimed to enhance the PageRank of X by establishing direct connections from Y and Z to X.

After loading the dataset and constructing the connectivity matrix, we applied the PageRank algorithm to calculate the PageRanks for all three pages. We extracted the PageRank values for X, Y, and Z from the resulting vector. The PageRank of X was found to be $1.4400850291097994e-06$, representing a further improvement compared to the previous scenarios.

Additionally, we determined the PageRanks of Y and Z, which were now connected directly to X. Both Y and Z were assigned equal PageRank values of $5.333648255766965e-07$. These values indicate that Y and Z are equally important in terms of their impact on the PageRank of X.

To evaluate the overall effect of adding Y and Z on the remaining pages, we calculated the norm difference between the previous PageRank vector and the updated vector with Y and Z. The norm difference quantifies the change in PageRank for the existing pages resulting from the addition of the new pages. The computed norm difference was $5.6625897409508286e-08$, indicating some alterations in the rankings of the other pages.

Furthermore, we examined the changes in ranking order by sorting the PageRank vectors. We obtained the indices of the highest-ranked pages in both the original and updated vectors and compared them. The number of differing ranks between the two vectors revealed that $52,911$ ranks had changed due to the addition of pages Y and Z.

In conclusion, to maximize the PageRank of X, we found that directing links only from Y and Z to X was the optimal configuration. By establishing these direct connections, the PageRank of X was further improved. The results highlight the importance of link structure in influencing the ranking of web pages and demonstrate the effectiveness of strategic link placement to enhance PageRank.

Below is represented the code that we used for this question:

```python
# method to parse the data. Obviously, storing all that information
# naively in a numpy array would lead to memory error. Thus, sparse
# matrix is used.
def load_dataset3(data):
    # list to store the origin pages,
    # i.e. the pages showing to other pages
    from_pages = []
    # list to store the destination pages,
    # i.e. the pages being showed by other pages
    to_pages = []
    with open(data, 'r') as f:
        # for each line of the initial data file
        for line in f.readlines():
            # get the item of the first column
            # and substract one. This is done because the ids of webpages
            # begin from 1 but the indexes for matrices begine from 0. So
            # 1 will stand for 0.
            from_page = int(line.split()[0]) - 1
            # Apply the same logic as above for the element of the second
            # column.
            to_page = int(line.split()[1]) - 1
            # append the the from page to the relative list
            from_pages.append(from_page)
            # append the the to page to the relative list
            to_pages.append(to_page)

    # compute number of edges
    edges = len(from_pages)
    # compute number of nodes
    nodes = len(set(from_pages) | set(to_pages))

    # add a new edge Y -> X
    from_pages.append(nodes)
    to_pages.append(nodes + 2)

    # add a new edge Z -> X
    from_pages.append(nodes + 1)
    to_pages.append(nodes + 2)

    # increases as needed the number of nodes and edges
    nodes += 3
    edges += 2

    # fill the connectivity matrix, adjacency matrix. A sparse structure
    # is used. That way the implementation is extremely memory efficient.
    csr_m = sparse.csr_matrix(([1]*edges,
                               (to_pages, from_pages)),
                              shape=(nodes, nodes))
    # return the adjacency sparse matrix
    return csr_m

csr_m3 = load_dataset3("./Dataset/stanweb.dat")
pr3, t3, list_conv3 = pagerank_galgo(csr_m3)

pageRank_Xc = np.asarray(pr3[csr_m3.shape[0] - 1]).ravel()[0]
print("The PageRank of page X is:", pageRank_Xc)

pageRank_Yc = np.asarray(pr3[csr_m3.shape[0] - 2]).ravel()[0]
print("The PageRank of page Y is:", pageRank_Yc)

pageRank_Zc = np.asarray(pr3[csr_m3.shape[0] - 3]).ravel()[0]
print("The PageRank of page Z is:", pageRank_Zc)


norm_diff = np.linalg.norm(np.asarray(pr[:csr_m.shape[0]].ravel()) - \

np.asarray(pr3[:csr_m3.shape[0] - 3].ravel()))
print("Changes in PageRank of the other pages due to the addition of the new page:", norm_diff)

mat_prg = np.asarray(pr).ravel()
mat_prg3 = np.asarray(pr3[:csr_m3.shape[0] - 3]).ravel()

indices_pr = mat_prg.argsort()[-len(mat_prg):][::-1]
indices_pr3 = mat_prg3.argsort()[-len(mat_prg3):][::-1]
print("Changes in:",
      np.sum(indices_pr[:] != indices_pr3[:]), "ranks.")
```

8 Question 2. d

In this part, we explored the idea of improving the PageRank of page X by adding links to older, popular pages. We considered two scenarios: adding links from X to older, popular pages and adding links from Y or Z to older, popular pages.

In the first scenario, where we added links from X to the top 100 popular pages, we observed a significant decrease in the PageRank of X. The PageRank of X fell to $1.4400732985803644e-06$, indicating that this approach did not improve X's PageRank. This outcome can be attributed to the fact that the PageRank of X is distributed among a larger number of webpages, diluting its overall importance.

In the second scenario, we investigated the impact of adding links from Y or Z to the top 100 popular pages. In both cases, the PageRank of X decreased even further. When we added links from Y to the popular pages, the PageRank of X dropped to $9.912111253145325e-07$. This result aligns with our previous observation that distributing the PageRank of Y (or Z) across more webpages leads to a decrease in X's PageRank.

Based on these results, it can be concluded that adding links from X, Y, or Z to older, popular pages is not an effective strategy for improving the PageRank of X. The PageRank of a webpage generally benefits from having more incoming links, rather than distributing its connections to other webpages. Therefore, alternative approaches should be explored to enhance the PageRank of X.

Below is represented the code that we used for this question:

```python
# determine top 100 - most popular pages - here as popular are considered
# pages with high PageRank, because PageRank can be seen as a measure of
# popularity
top_100 = indices_pr[0:100]

# determine top 100 - most popular pages - here as popular are considered
# pages with high in-degree
list_degree = np.asarray([len(dict_in[x]) for x in sorted(list(dict_in.keys()))])
indices_degree = list_degree.argsort()[-len(list_degree):][::-1]
popular_100 = list_degree[0:100]


# method to parse the data. Obviously, storing all that information
# naively in a numpy array would lead to memory error. Thus, sparse
# matrix is used.
def load_dataset4(data, top100):
    # list to store the origin pages,
    # i.e. the pages showing to other pages
    from_pages = []
    # list to store the destination pages,
    # i.e. the pages being showed by other pages
    to_pages = []
    with open(data, 'r') as f:
        # for each line of the initial data file
        for line in f.readlines():
            # get the item of the first column
            # and substract one. This is done because the ids of webpages
            # begin from 1 but the indexes for matrices begine from 0. So
            # 1 will stand for 0.
            from_page = int(line.split()[0]) - 1
            # Apply the same logic as above for the element of the second
            # column.
            to_page = int(line.split()[1]) - 1
```

```
                # append the the from page to the relative list
                from_pages.append(from_page)
                # append the the to page to the relative list
                to_pages.append(to_page)

    # compute number of edges
    edges = len(from_pages)
    # compute number of nodes
    nodes = len(set(from_pages) | set(to_pages))

    # add a new edge Y -> X
    from_pages.append(nodes)
    to_pages.append(nodes + 2)

    # add a new edge Z -> X
    from_pages.append(nodes + 1)
    to_pages.append(nodes + 2)

    # add links from X to older, popular pages
    for node in top_100:
        from_pages.append(nodes + 2)
        to_pages.append(node)

    # increase number of nodes and edges as needed
    nodes += 3
    edges += 102

    # fill the connectivity matrix, adjacency matrix. A sparse structure
    # is used. That way the implementation is extremely memory efficient.
    csr_m = sparse.csr_matrix(([1]*edges,
                               (to_pages, from_pages)),
                              shape=(nodes, nodes))
    # return the adjacency sparse matrix
    return csr_m


# method to parse the data. Obviously, storing all that information
# naively in a numpy array would lead to memory error. Thus, sparse
# matrix is used.
def load_dataset5(data, top100):
    # list to store the origin pages,
    # i.e. the pages showing to other pages
    from_pages = []
    # list to store the destination pages,
    # i.e. the pages being showed by other pages
    to_pages = []
    with open(data, 'r') as f:
        # for each line of the initial data file
        for line in f.readlines():
            # get the item of the first column
            # and substract one. This is done because the ids of webpages
            # begin from 1 but the indexes for matrices begine from 0. So
            # 1 will stand for 0.
            from_page = int(line.split()[0]) - 1
            # Apply the same logic as above for the element of the second
            # column.
            to_page = int(line.split()[1]) - 1
            # append the the from page to the relative list
            from_pages.append(from_page)
            # append the the to page to the relative list
            to_pages.append(to_page)

    # compute number of edges
    edges = len(from_pages)
    # compute number of nodes
    nodes = len(set(from_pages) | set(to_pages))

    # add a new edge Y -> X
    from_pages.append(nodes)
    to_pages.append(nodes + 2)

    # add a new edge Z -> X
    from_pages.append(nodes + 1)
    to_pages.append(nodes + 2)

    # add links from Y (or Z) to older, popular pages
    for node in top_100:
        from_pages.append(nodes + 1)
        to_pages.append(node)

    # increase number of nodes and edges as needed
    nodes += 3
    edges += 102

    # fill the connectivity matrix, adjacency matrix. A sparse structure
    # is used. That way the implementation is extremely memory efficient.
    csr_m = sparse.csr_matrix(([1]*edges,
                               (to_pages, from_pages)),
                              shape=(nodes, nodes))
    # return the adjacency sparse matrix
    return csr_m


csr_m4 = load_dataset4("./Dataset/stanweb.dat", top_100)
pr4, t4, list_conv4 = pagerank_galgo(csr_m4)

csr_m5 = load_dataset5("./Dataset/stanweb.dat", top_100)
pr5, t4, list_conv5 = pagerank_galgo(csr_m5)

# if we add links from X to older, popular pages
pageRank_Xd = np.asarray(pr4[csr_m4.shape[0] - 1]).ravel()[0]
print("If we add links from X to older, popular pages the PageRank of page X is:", pageRank_Xd)

# if we add links from Y to older, popular pages
pageRank_Xd2 = np.asarray(pr5[csr_m5.shape[0] - 1]).ravel()[0]
print("If we add links from Y to older, popular pages the PageRank of page X is:", pageRank_Xd2)
```

9 Question 2. e

Based on the previous questions, it is clear that boosting a webpage's PageRank is a difficult process. That does not, however, make it impossible. The PageRank of a webpage can be enhanced as more and more pages link to it. Of course, if those pages have a high PageRank, their contribution will be considerably greater. Nonetheless, the more pages that point to a page, the greater the PageRank of that page. Furthermore, links between such pages may be beneficial.