RUTGERS
UNIVERSITY | NEW BRUNSWICK

Harshal Sinha (NetID - hs1030)

Param Amit Shah (NetID - ps1169)

Vedant Gupta (NetID - vg374)

Master of Science in Computer Science

Page Rank Algorithms

December 2021

The School of Graduate Studies

Contents

Chapter 1

Introduction

1.1 Page Rank

The PageRank algorithm or Google algorithm was introduced by Lary Page, one of the founders of Google. It was first used to rank web pages in the Google search engine. It works by counting the number and quality of links to a page to determine a rough estimate of how important the website is. The underlying assumption is that more important websites are likely to receive more links from other websites.

PageRank is designed and developed to prioritize the results of web keyword searches. In this algorithm, citations or backlinks to a webpage is counted. This provides some information about the quality and importance of a given webpage. PageRank does not only count all the number of links back to a given page equally, it normalizes an inward link by the total number of links going out of a given page, which can be used to assess the importance of a link to a webpage. PageRank defines a probability for each webpage as the probability that a user may randomly open that webpage. Then by sorting the webpages in the decreasing order of their probability, the most important and relevant webpages can be found.

1.2 Page Rank Theory

1.2.1 Page Rank(PR)

The web can be represented like a directed graph where nodes represent the web pages and edges form links between them. Typically, if a node (web page) i is linked to a node j, it means that i refers to j. PageRank uses the following formula to calculate and update PageRank PR for each webpage:

$$PR(p_i) = (1-\beta)/N + \beta \sum_{P_j \epsilon M(p_i)} \frac{PR(p_j)}{L(p_j)} \tag{1.1}$$

Here p_i are the pages being ranked, M(p_i) is the set of pages that link to page p_i, L(p_j) is the number of outbound links on page p_j, N is the total number of pages, and β is the damping factor. This formula can be written in matrix form as well, which makes the calculations easier.

We define matrix R, which contains the Page Rank values of all web pages. This matrix is called Eigen vector.

$$R = \begin{bmatrix} PR_(p_1) \\ PR_(p_2) \\ . \\ . \\ PR_(p_n) \end{bmatrix} \tag{1.2}$$

The R matrix gets updated using the following formula:

$$\begin{bmatrix} (1-\beta)/N \\ (1-\beta)/N \\ . \\ . \\ (1-\beta)/N \end{bmatrix} + \beta \begin{bmatrix} l_(p_1,p_1)...l_(p_1,p_N) & & \\ . & . & . \\ . & . & . \\ . & . & . \\ l_(p_1,p_1)...l_(p_1,p_N) & & \end{bmatrix} R \tag{1.3}$$

where the adjacency function $l(p_i, p_j)$ is the ratio between number of links outbound from page j to page i to the total number of outbound links of page j. The adjacency function is 0 if page p_j does not link to p_i, and normalized such that for each j $\sum_{i=1}^{N} l(p_i, p_j) = 1$ i.e. the elements of each column sum up to 1.

PageRank algorithms can be computed using iterative (also known as Power Iterative Mehtod)

or algebraic method. We are using iterative method to calculate the PageRank values of all the web pages. In the algorithm, we are initially giving uniform probability to all the web pages. Given by:

$$PR(p_i, 0) = \frac{1}{N} \tag{1.4}$$

Then, the PageRank of each page will be updated iteratively according to:

$$PR(p_i, t+1) = \frac{1-\beta}{N} + \beta \sum_{P_j \epsilon M(p_i)} \frac{PR(p_j)}{L(p_j)} \tag{1.5}$$

We continue the iterations until the pagerank values converge. Figure 1.1 shows a simple page rank representation of a network, the webpages in the figure are connected to each other via hyperlinks. The pagerank value for each node is shown on the node, for example page C has pagerank value of 34.3 and page D has a value of 3.9, so page C will be ranked higher than page D and E even though page E has a lot of incoming edges page C has been cited by page B, which is a high ranked page in this graph, causing the bump in ranking of page C. The percentage value of a pagerank can be interpreted as the probability that a random web surfer at random will choose that page.

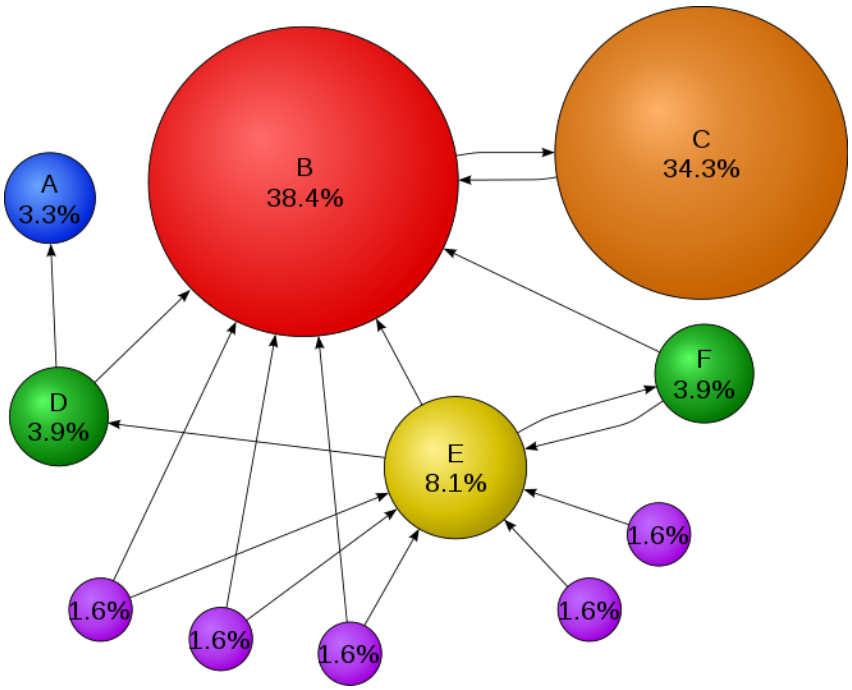

Figure 1.1: Example of Page Rank algorithm applied to a sample graph

1.2.2 Weighted Page Rank (WPR)

In our PageRank algorithm, each page's rank value is evenly divided among its out-link edges but in Weighted PageRank algorithm , we assign larger rank values to more relevant web pages. The importance of each link is dependent on the number of incoming links $W_{in}^{v,u}$ and outgoing links $W_{out}^{v,u}$. These incoming and outgoing weights are used to calculate the PageRank (PR) value for a given web page.

$W_{in}^{v,u}$: weight of link from v to u (v,u). It is calculated as the ratio of number of incoming links to page u to the total number of incoming links of all the web pages from v.

$W_{out}^{v,u}$: weight of link from v to u (v,u). It is calculated as the ratio of number of outgoing links to page u to the total number of outgoing links of all the web pages from v.

$$W_{v,u}^{in} = \frac{I_u}{\sum_{p \epsilon R_{(v)}} I_p} \tag{1.6}$$

$$W_{out}^{v,u} = \frac{O_u}{\sum_{p \epsilon R_{(v)}} O_p} \tag{1.7}$$

Where O_u represents the number of out links to page u and I_p represents the number of out links of all pages cited by reference R(v).

Where I_u represents the number of in links to page u and I_p represents the number of in links of all pages cited by reference R(v).

We can calculate the rank value of each given page using the above two formula (1.6) and (1.7) by:

$$PR_{(u)} = (1 - \beta) + \beta \sum (v \epsilon B(u) PR_{(v)} W_{(v,u)}^{in} W_{(v,u)}^{out} \tag{1.8}$$

1.2.3 Power Iteration Method(PI)

The power iteration method is based on the original PageRank algorithm but the way in which we calculate and update the page rank is quite different. The proposed method makes it much easier and faster, The R matrix containing the PageRanks are updated using the following formulas:

$$R = (\beta M + \frac{(1 - \beta)E}{N} R) =: \hat{M} R \tag{1.9}$$

$$R = (\beta M + \frac{(1-\beta)E}{N}R) =: MR \tag{1.10}$$

Where M was defined earlier in PageRank algorithm and E is the matrix of all ones. The R matrix generated is the eigenvector of $\hat{M}$. Power method is a fast and easy way to calculate R, We start with an initial guess for R and iterate until R converges. If we start with an arbitrary vector x(0), the operator Mˆ is applied in succession:

$$x(t+1) = \hat{M}x(t) \tag{1.11}$$

The iterations will be continued until:

$$|x(t+1) - x(t)| < \epsilon \tag{1.12}$$

1.2.4 Power Extrapolation Method (PE)

In order to speed up the calculations of Page Rank, the Power iteration method has been modified and this modified method is called Power Extrapolation method. In this method, it is assumed that the iterate $x(k-1)$ can be written as a linear combination of eigen vectors e1 and e2, where e2 has the eigen value of c.

$$x(k-1) = e_1 + \alpha_2 e_2 \tag{1.13}$$

Now, the current iterate x(k) can be calculated by the power method, which iterates by successive multiplication by $\hat{M}$.

$$x(k) = \hat{M}x(k-1) = M(e_1 + \alpha_2 e_2) = e_1 + \alpha_2 \lambda_2 e_2 \tag{1.14}$$

Since the eigen value of e_2 was c, therefore $\lambda 2 = c$ and we get:

$$x(k) = e_1 + \alpha_2 c e_2 \tag{1.15}$$

Now we can solve for e_1

$$e_1 = (x(k) - \frac{cx(k-1)}{1-c}) \tag{1.16}$$

1.3 Data Collection

Dataset statistics	
Nodes	875713
Edges	5105039
Nodes in largest WCC	855802 (0.977)
Edges in largest WCC	5066842 (0.993)
Nodes in largest SCC	434818 (0.497)
Edges in largest SCC	3419124 (0.670)
Average clustering coefficient	0.5143
Number of triangles	13391903
Fraction of closed triangles	0.01911
Diameter (longest shortest path)	21
90-percentile effective diameter	8.1

Figure 1.2: Google Web Graph statistics

The data used in this project is called "Google Web Graph" and it was collected from "Stanford Network Analysis Platform". This dataset represents a directed graph containing 875713 nodes with 5105039 edges connecting them. In the web graph data, the nodes represent web pages and edges are hyperlinks.

The dataset is stored in a text file, where each webpage is represented by a web id. The web ids are stored in two columns in the text file and each hyperlink is represented as an edge from the first column to the second column. We created an adjacency matrix out of this dataset and used it to test the implementation of the different page rank algorithms.

In order to implement the algorithms mentioned earlier and compare their performance, we needed to acquire a data base that contains information about the hyperlinks between web pages. It was necessary to find a large dataset in order to make sure that our algorithm can work in real applications, where a huge amount of data is being processed.

1.4 Data Processing

Firstly, we read the data from the text file and create an adjacency list from it. Each node consists of a list of nodes representing the directed edges to other nodes. In order to use this data to test our algorithms we need to transform it into an adjacency matrix. We need two adjacency matrices i.e. one for page rank and the other for weighted page rank algorithm. Values of the matrices would be different for different algorithms.

As we are taking a large web graph with 875713 nodes and 5105039 edges, the adjacency matrices would be very sparse. For a robust implementation, we use a row sparse matrix from *Scipy* python library.

For creating the adjacency matrix, each node is picked from the adjacency list and iterated over its adjacency list to input the values in sparse matrix for page rank and weighted page rank. Refer appendix section 1.7 for the code to read the data from the text file, to create the initial adjacency list and, to create the sparse matrix from the adjacency list.

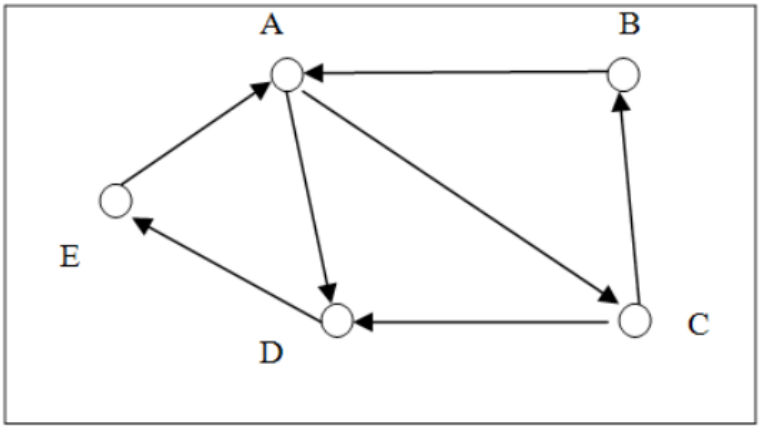

Figure 1.3: Example Web Graph

The two types of sparse matrices that we will be using are:

- Sparse matrix for Page Rank: For page rank algorithm, sparse matrices depend only on the in-link weight of the edges. For example, say we have three nodes - A,B and C. There is an edge from A to B and A to C. The rank of A is divided between both the edges AB and AC. If there are multiple edges between a pair of nodes then only one of the node is considered and weight for that one node is doubled.
- Sparse matrix for Weighted Page Rank: For weighted page rank values of the sparse matrix

are both dependent on Weight_IN and Weight_OUT. Weight_IN is calculated using in-link weights of a node and its affiliated nodes in-links. For the graph shown in figure 1.2 we show the calculation of In - Weight and Out - Weight for the edges AC and AD.

$$Weight_{AC}^{IN} = \frac{I_C}{I_B + I_E} = \frac{1}{1+1} = \frac{1}{2} \tag{1.17}$$

$$Weight_{AD}^{IN} = \frac{I_D}{I_B + I_E} = \frac{2}{1+1} = \frac{1}{1} = 1 \tag{1.18}$$

$$Weight_{AC}^{OUT} = \frac{O_C}{O_B + O_E} = \frac{1}{1+1} = \frac{1}{2} \tag{1.19}$$

$$Weight_{AD}^{OUT} = \frac{O_D}{O_B + O_E} = \frac{1}{1+1} = \frac{1}{2} \tag{1.20}$$

1.5 Algorithm Implementation Details

For the purpose of this project we have implemented two algorithms normal page rank and weighted page rank, for each of those we implemented them using power iteration method and power extrapolation method i.e. we have a total of 4 algorithms, page rank with power iteration, page rank with power extrapolation, weighted page rank with power iteration and weighted page rank with power extrapolation.

We use all these algorithms to calculate the rank of vector r. Both the techniques (power iteration and power extrapolation) for all the algorithms are used to reach the convergence based on the tolerance and all these algorithms are finally compared. For power extrapolation there is one added parameter namely residuals is also taken in account. We stop the algorithm when convergence is reached.

Below is the high - level pseudo code for calculation of Page Rank:

- Set initial values for tolerance and damping factors.
- Create adjacency matrices for Page Rank and Weighted page rank.
- Calculate rank vector for Page Rank algorithm using power iteration method.
- Calculate rank vector for Page Rank algorithm using power extrapolation method.
- Calculate rank vector for weighted Page Rank algorithm using power iteration method.
- Calculate rank vector for weighted Page Rank algorithm using power extrapolation method.

1.6 Analysis of performance of the algorithms

1.6.1 Time and Space Complexity Analysis

For Page Rank using Power Iteration the time complexity is O(k * N) where k is the number of iterations and N is the number of pages(nodes in the graph). At each iteration, the page rank value of all the nodes in the graph is computed.

The space complexity is O(N) because we have to store the page rank value for all the N nodes in the graph.

1.6.2 Performance analysis by changing the Error Tolerance

Figure 1.3 shows the comparison of performance of different page rank algorithms in terms of number of iterations required for convergence as the error tolerance is increased. From the plot, we notice that as we increase the error tolerance of the algorithm, the number of iterations required for convergence of both Page Rank Power Iteration and Power Extrapolation algorithm decreases very sharply while for Weighted Page Rank it decreases gradually. We can see that as the tolerance factor increases beyond 0.01 all four algorithms require almost similar number of iterations. This is because Weighted Page Rank assigns more page rank value to popular pages hence that results in less error while in Normal Page Rank there is more error.

Figure 1.4: Plot of Number of Iterations vs Tolerance

1.6.3 Performance analysis by changing the Damping Factor(β)

From the figure (1.4), we compare the performances of different PageRank algorithms that we have implemented in this project in terms of the number of iterations required for the algorithm to converge as we keep on increasing the damping factor. From the graph , we can see that as the

damping factor increases , number of iterations required for convergence increases exponentially for both Power Iteration and Power Extrapolation PageRank algorithm but increases linearly for Weighted PageRank algorithms. We further see that for lower values of damping factor (less than 0.2) , all the four algorithms converge at similar number of iterations as first half of the equation (1.1) overpowers the other part for all the algorithms. However, as we increase the damping factor, the weights of forward links increases the performance of the algorithm and thus the weighted PageRank algorithm requires less number of iterations to converge as compared to standard PageRank algorithm.

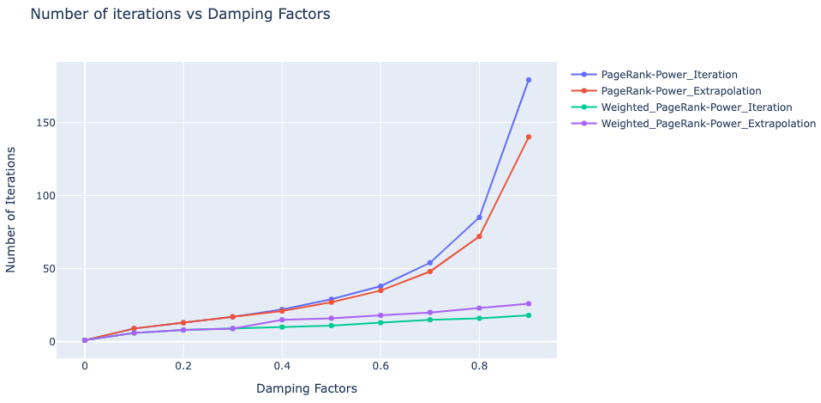

Figure 1.5: Plot of Number of Iterations vs Damping Factor

1.7 Conclusion

Page Rank Algorithm is used to rank the relevant pages by treating all links equally when distributing rank scores. Thus, its most obvious application is for search engines. PageRank is able to rank websites in order to provide more relevant search results faster. Its also commonly used in Information Retrieval algorithms which is used to extract information from user's past data. In this project, we have also implemented the Weighted Page Rank algorithm, which takes into account the importance of both the inlinks and the outlinks of the pages and distributes rank scores based on the popularity of the pages. When we compare the performances (number of iterations) of both the normal PageRank and weighted PageRank, we see that weighted PageRank is better than the normal one.

1.8 Future Work

The current project has been implemented using a data set that was released by Google in 2002. In the future, we can make this page ranking dynamic by using a web crawler which continuously indexes the pages of the web graph. This way the page ranking can run on dynamic data.

1.9 Appendix

This section contains the code snippets

1.9.1 Importing required libraries setting the base parameters

```
import numpy as np
from scipy import sparse
import plotly.graph_objects as go
import plotly.offline as po
from plotly.offline import iplot
import time as t

# Parameters for the various algorithms for page rank
beta = 0.8 # damping factor
tolerance = 10**-10 # tolerance
TOP_K = 7 # Return to the TOP_K related pages
dValueForExtraPolation = [8] # d value for extrapolation for accelarating page
                                        rank
```

1.9.2 Generating graph data

```
s = set() # Set used to hold the nodes in the dataset, its length is the number
                                        of nodes in the dataset.
def getGraphData(file):
    '''This function returns a dictionary of edges between the graph nodes read
                                        from the file i.e. the adjacency
    list of the graph data.'''
    graphEdges = dict() # Dictionary to store graph edges
    with open(file) as f: # open the file containing the graph data.
        # Read all lines from the file, a line contains tab seperated source
                                        node and dest node.
        lines = f.readline()
```

```
        while lines:
            line = [int(node) for node in lines.strip().split('\t')] # get a
                                        single line from lines
            # line[0] is the source node, line[1] is the destination node.
            # Storing the edge between line[0] and line[1] in graphEdges
                                        dictionary.
            # Adding the unique nodes in set s.
            if graphEdges.get(line[0]):
                s.add(line[1])
                graphEdges[line[0]].append(line[1])
            else:
                graphEdges[line[0]] = [line[1]]
                s.add(line[0])
                s.add(line[1])
            lines = f.readline()
    # len(s) is number of nodes in the data
    return graphEdges, len(s)
```

1.9.3 Method to create sparse matrix from graph.

```
def createSparseMatrixFromGraph(graph, size):
    '''This function creates a sparse matrix from the adjacency list.'''
    row = [] # the row index of the matrix
    col = [] # the col index of the matrix
    data = [] # The value to be stored at the row index and col index
    for node in graph.keys(): # iterate over all the nodes
        nodeVal = graph.get(node) # the list of nodes to which the edge goes to
                                        from nodeKey
        # We assign equal node values to all the nodes which is 1/number of
                                        nodes the edges goes to
        Count = len(nodeVal)
        eachNodeVal = 1 / Count
        # for handling duplicates, for duplicates we assign more weights
        countDict = dict()
        for val in nodeVal:
            if countDict.get(val):
                countDict[val] = countDict.get(val) + 1
            else:
                countDict[val] = 1
```

```
        for node_value in list(set(nodeVal)):
            count = countDict.get(node_value) # get the count of the node from
                                              val dict
            row.append(node_value) # append key to the row matrix
            col.append(node) # append the node to which the connection goes to
                                              in the column matrix
            # multiply with count the decided weight to get the final value.
            data.append(count * float(eachNodeVal))
    # using sparse from scipy to generate the matrix
    matrix = sparse.csr_matrix((data, (row, col)), shape=(size, size))
    return matrix
```

1.9.4 Method to create sparse matrix from weighted graph

```
def createSparseWeightedMatrixFromGraph(graph, size):
    row = [] # row index of the matrix
    col = [] # column index of the matrix
    data = [] # The value to be stored at the row index and col index
    for node in graph.keys(): # iterate over all the nodes
        nodeVal = graph.get(node) # the list of nodes to which the edge goes to
                                          from nodeKey
        # We assign equal node values to all the nodes which is 1/number of
                                          nodes the edges goes to
        Count = len(nodeVal)
        eachNodeVal = 1 / Count
        # for handling duplicates, for duplicates we assign more weights
        countDict = dict()
        for val in nodeVal:
            if countDict.get(val):
                countDict[val] = countDict.get(val) + 1
            else:
                countDict[val] = 1
        for node_value in list(set(nodeVal)):
            count = countDict.get(node_value) # get the count of the node from
                                              val dict
            row.append(node_value) # append key to the row matrix
            col.append(node) # append the node to which the connection goes to
                                              in the column matrix
            data.append(count) # append the count to the data list.
```

```
    matrix = sparse.csr_matrix((data, (row, col)), shape=(size, size))

    row_sum = matrix.sum(axis=1).T.tolist()[0] # calculate the sum of rows,
                                        used in assigning weights
    col_sum = matrix.sum(axis=0).tolist()[0] # calculate the sum of columns,
                                        used in assigning weights
    row = [] # row index of the matrix
    col = [] # column index of the matrix
    data = [] # The value to be stored at the row index and col index
    for node in graph.keys(): # iterate over all the nodes
        nodeVal = graph.get(node) # the list of nodes to which the edge goes to
                                         from nodeKey
        Count = len(nodeVal)
        eachNodeVal = 1 / Count
        # for handling duplicates, for duplicates we assign more weights
        countDict = dict()
        for val in nodeVal:
            if countDict.get(val):
                countDict[val] = countDict.get(val) + 1
            else:
                countDict[val] = 1
        #Incoming weight for children node
        sum_In = 0
        for node_value in list(set(nodeVal)):
            sum_In = sum_In + row_sum[node_value]
        #Outgoing weight for children node
        sum_Out = 0
        for node_value in list(set(nodeVal)):
            sum_Out = sum_In + col_sum[node_value]
        # Calculating the incoming and outgoing weight, assigning weight as the
                                         product of two.
        for node_value in list(set(nodeVal)):
            node_in = row_sum[node_value]
            incoming_weight = float(node_in/sum_In)

            node_out = col_sum[node_value]
            outgoing_weight =float(node_out/sum_Out)
            row.append(node_value)
            col.append(node)
```

```
            data.append(incoming_weight*outgoing_weight)
    # return the matrix
    matrix = sparse.csr_matrix((data, (row, col)), shape=(size, size))
    return matrix
```

1.9.5 Implementation of Power Iteration algorithm

```
def power_iteration(sparse_matrix,beta,tolerance,number_of_nodes):
    # Given, 1 is one vector with nx1 entries of value 1
    one = np.ones((number_of_nodes,1))
    # initialising the r0
    r = 1/ float(number_of_nodes) * one
    iteration = 0 # number of iterations for the algo to get the output.
    while True:
        iteration += 1 # increment the count
        rnew = ((1-beta)/number_of_nodes) * one + beta * sparse_matrix.dot(r) #
                                        new r value for all the nodes
        l1Norm = np.linalg.norm(abs(rnew - r), ord=1) # calculate the error,
                                        used l1norm
        if l1Norm < tolerance: # If error is less than tolerance break
            break
        r = rnew # return the new r and the number of iterations it required
                                        the algo to0 converge.
    return r,iteration
```

1.9.6 Implementation of Power Extrapolation

```
def power_extrapolation(transition_matrix, number_of_nodes,
                                        dValueForExtraPolation, beta, tolerance
                                        ):
    dValues = dValueForExtraPolation # D values to accelarate page rank
    d_0 = dValues[0] # initial d_value
    pageRank = {}  # Dictionary to store page rank
    residual = {} # dictionary to store errors
    for d_val in dValues: # iterate over dvalue array
        page_rank_iteration = []
        initial_pagerank = 1 / number_of_nodes * np.ones((number_of_nodes, 1))
                                        # calculate initial page rank
        iteration = 0 # initial iteration
```

```
        r_error = {} # dict to store erorr
        page_rank_iteration.append(initial_pagerank)
        while True: # loop till error becomes less than tolerance
            iteration += 1 # increment the count
            # calculate the next page rank
            next_pagerank = (1 - beta) / number_of_nodes * np.ones((
                                        number_of_nodes, 1)) + beta *
                                        transition_matrix.dot(initial_pagerank)
            if iteration == d_val + 2:
                next_pagerank = (next_pagerank - (beta**d_val)*
                                        page_rank_iteration[iteration - d_val])
                                         / (1 - beta**d_val)
            error = np.linalg.norm(abs(next_pagerank - initial_pagerank), ord=1
                                        ) # calculate the error
            r_error[iteration] = error
            if error < tolerance: # if error becomes less than the tolerance
                                        break
                page_rank_iteration.append(next_pagerank)
                break
            page_rank_iteration.append(next_pagerank)
            initial_pagerank = next_pagerank
        pageRank[d_val] = page_rank_iteration
        residual[d_val] = r_error
    # return the final page rank and the iteration.
    return pageRank.get(d_0)[-1], iteration
```

1.10 References

1. https://en.wikipedia.org/wiki/PageRank
2. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.3.6130&rep=rep1&type=pdf
3. https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.454.5022&rep=rep1&type=pdf
4. https://medium.com/@sahirnambiar/a-simplified-implementation-of-pagerank-b8b5d282dc42
5. https://indjst.org/download-article.php?Article_Unique_Id=INDJST7111&Full_Text_Pdf_Download=True