A

PROJECT REPORT

ON

Analysis and Efficient Implementation of Google's PageRank Algorithm

Prepared By: Sujay C Sharma (2017A7PS0012G)

Under the Supervision of: Professor J.K.Sahoo

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE, PILANI

GOA CAMPUS

NOVEMBER 2019

Acknowledgements

I would like to thank Professor J.K.Sahoo for giving me the opportunity to work on this project and assisting me throughout the course of this semester by clarifying my doubts and pointing me in the right direction whenever I needed help on this project.

Table of Contents

1. Introduction 4
2. The PageRank Algorithm
   - 2.1 A Brief History of the PageRank Algorithm 6
   - 2.2 Motivation 6
   - 2.3 The Model 7
   - 2.4 The Algorithm 8
3. Implementations of the PageRank Algorithm
   - 3.1 Iterative Method 11
   - 3.2 Matrix Power Method 12
   - 3.3 Inner-Outer Iteration Method 14
   - 3.4 Results 17
4. Damping Factor Analysis 18
5. Approaches to Efficiently Implement the PageRank Algorithm
   - 5.1 Altering the Personalisation Vector 21
   - 5.2 Divide and Conquer Approach Matrix Multiplication 22
   - 5.3 Compressed Sparse Row Matrix Representation 22
6. Future Scope and Conclusion 26
7. Bibliography 27

Introduction

The World Wide Web (WWW) is an information system where documents and other web resources are identified by Uniform Resource Locators (URLs) and are accessible over the internet which is a global system of interconnected computer networks that links devices worldwide. There has been a rapid growth in the World Wide Web since it was developed in 1989 by Tim Berners-Lee. This has led the current age of mankind to be known as the 'Information Age'.[1]

According to a survey by the internet services company Netcraft, there are over a billion different web sites(1,518,207,412) on the World Wide Web as of 2019 with this number regularly fluctuating even as we speak. This has led to almost 2 zettabytes (2 billion-billion bytes) of information being available on the Internet. With over 50% of the human population relying on the Internet for information on a vast variety of topics, organizing and ranking webpages on the Internet is now a task of utmost importance to ensure that users get access to quality content in their desired domain on the Internet.[2]

This has led to a number of webpage ranking algorithms such as the HITS algorithm used by ask.com, the IBM CLEVER project, the TrustRank and Hummingbird algorithms. However, the most famous and important of these webpage ranking algorithms is without a doubt Google's PageRank algorithm which is used to rank web pages in their search engine results.

Owing to the fact that Google's search engine accounts for 76% and 86% of desktop and mobile search traffic respectively, Google's PageRank algorithm is the topic of a lot of discussion and analysis from an academic perspective as well as a commercial perspective.[3]

This report will be taking a look at the history and the basic idea of Google's PageRank algorithm followed by 3 different implementations of the algorithm which each have their own pros and cons. This will then be followed by an analysis of the algorithm and finally cover some approaches implemented in order to improve the efficiency of the algorithm.

The PageRank Algorithm

A Brief History of the PageRank Algorithm:

PageRank is an algorithm used by Google Search to rank webpages in their search engine results. The algorithm was designed by Larry Page and Sergey Brin in 1996 at Stanford University as a part of a research project about a new kind of search engine. The system was developed with the help of Scott Hassan and Alan Steremberg who were both critical to the development of Google. The algorithm's name is a play on the name of developer Larry Page. Shortly after the publishing of this research paper in 1998, Page and Brin founded Google.[4]

Motivation:

In the field of academic research, it is well known that the most popular research papers are the ones that tend to have the most citation counts. Webpages on the Internet are connected to each other via a set of hyperlinks. This was the motivation that led to the idea that webpages could be ranked based on number and quality of hyperlinks similar to citation counts for research papers.

Therefore, the study of citation analysis of academic research papers was the primary motivation for coming up with the PageRank model.

The Model:

Larry Page and Sergey Brin came up with a graph based model of all the webpages on the Internet which is now termed as a webgraph.
Webgraphs are constructed based on a very simple logic:

(i) Each webpage on the Internet becomes a node of the corresponding webgraph.

(ii) Each hyperlink between two webpages becomes an edge of the corresponding webgraph.

(iii) Since these hyperlinks can be both to or from a particular webpage, it is necessary to capture the direction as well. This leads to each edge having a direction associated with it (in-edges and out-edges).

(iv) No self loops are allowed in a webgraph.

Therefore, a webgraph is nothing but a directed graph where each node is a webpage and each edge is a hyperlink.

A numerical weightage is now assigned to all edges of the webgraph. The individual PageRank of a particular webpage in a webgraph would be the sum of all of its in-edges, denoting its relative importance among all the webpages under consideration in the webgraph. The algorithm models the behaviour of an idealised random web surfer who chooses a webpage randomly to go to next from a list of available webpages. Therefore, the output of the PageRank algorithm is a probability distribution used to represent the likelihood that a person randomly clicking on links will arrive

at a particular webpage. The sum of the PageRanks of all the webpages in a webgraph will therefore always be 1.[7]

The Algorithm:

Let N denote the total number of nodes in the webgraph.
Let u denote a particular node.
Let PR(u) denote the PageRank of a node u.
Let F(u) denote the set of nodes to which u has an out-edge to.
Let N(u) denote the set of nodes to which u has an in-edge from.
Let d denote the damping factor of the algorithm.

The PageRanks for all the nodes of the webgraph are initialised to the same value which is equal to the 1/N.
The PageRank of a particular node u is then distributed evenly among all of its out-edges i.e each out-edge gets a value of (PR(u)/|F(u)|) assigned to it.
This step is repeated for all nodes of the webgraph.
Now, the PageRank of a node u in the next iteration of the algorithm will be the sum of all of the values assigned to its in-edges.
This step is repeated for all nodes of the webgraph.
This entire procedure described so far is then repeated till a steady state solution is achieved. Achieving a steady state solution is determined by calculating the difference between PageRanks of the current and previous iteration and comparing if they are less than a threshold value. If they are less

than the threshold value, then we say that the algorithm has converged and the PageRanks of all nodes in the current iteration are their actual PageRanks. One more issue that the algorithm has to take care of is the problem of dangling nodes.

A dangling node is a node which has a number of in-edges and which does not have any out-edges. The presence of dangling nodes can be very detrimental to the PageRank algorithm as they keep accumulating PageRank via their in-edges but they do not redistribute the PageRank as they do not have any out-edges. Therefore, such dangling nodes are termed as rank sinks as they accumulate PageRank as the number of iterations increase. However, this does not reflect the actual importance of the webpage represented by this dangling node. Therefore, the behaviour of the algorithm must be modified in this instance. This is done by defining an N-dimensional dangling vector which represents the probabilities of moving to a particular node from a dangling node. Since, we are trying to model an ideal random web surfer, we usually make it equally likely to go to any node from a dangling node.

A random web surfer would not keep surfing the Internet infinitely. This led to the addition of a term known as the damping factor which ensures faster convergence of the PageRank algorithm and helps model the random web surfer much better. Different applications of the PageRank algorithm would require different values of the damping factor.

Another modification to the algorithm would be to set the initial PageRanks based on an N-dimensional personalisation vector rather than just assigning all pages equal PageRanks. This can be used to model an individual user's webgraph where he is more likely to start from a particular webpage rather than a webpage at random.[7]

The entire algorithm can be concisely written as:

Iterative

At $t = 0$, an initial probability distribution is assumed, usually

$$PR(p_i; 0) = \frac{1}{N}.$$

At each time step, the computation, as detailed above, yields

$$PR(p_i; t+1) = \frac{1-d}{N} + d \sum_{p_j \in M(p_i)} \frac{PR(p_j; t)}{L(p_j)},$$

Here, $M(p_i)$ denotes set of all webpages from which an out-edge exists to p_i. $L(p_j)$ denotes number of out-edges from webpage p_j.

Implementations of the PageRank Algorithm

Iterative Method:

The iterative method is one of the simplest methods to implement the PageRank algorithm. The advantage of using the iterative method is that it takes up very less memory as we do not use matrix representations and operations. Memory usage is an important aspect of the algorithm given the size of modern webgraphs. The lack of large memory usage makes this the ideal method to implement the PageRank algorithm on a personal computer given that most personal computers do not have a lot of RAM.

The disadvantage of using the iterative method is that it is computationally expensive due to the presence of multiple looping constructs where calculations are done repeatedly to compensate for the lack of a matrix storage structure.

Therefore, this would be the method of choice if time is not of the essence and memory is limited.

The iterative method is implemented as shown in the code snippet below. It mainly follows the description of the algorithm in the previous section and therefore makes it the easiest to understand and implement.

```python
# Function to return the PageRank of all nodes in a given webgraph G using the basic iterative method
def PageRankIterative(G, damping_factor = 0.85, error_tolerance = 0.1, max_iterations = 100):
# Initialise number of iterations of algorithm to 0
    number_iterations = 0
# Initialise PageRank of all nodes to be initially the same value i.e 1/number_of_nodes
    input_ranks = []
    number_of_nodes = G.number_of_nodes()
    for i in range(0,number_of_nodes):
        input_ranks.append(1/number_of_nodes)
# Iterations of PageRank Algorithm
    while number_iterations <= max_iterations:
        number_iterations = number_iterations + 1
        output_ranks = []
# Initialise PageRanks of next iteration to 0 for all nodes
        for i in range(0,number_of_nodes):
            output_ranks.append(0)
# Compute PageRank iteratively node by node
        for i in range(0,number_of_nodes):
            number_out_edges = G.out_degree(i+1)
# Take care of dangling nodes by distributing its PageRank evenly to all nodes(randomly move to any node)
            if number_out_edges == 0:
                for j in range(0,number_of_nodes):
                    output_ranks[j] = output_ranks[j] + (input_ranks[i]/number_of_nodes)
# Find list of nodes which have a link to the current node
            number_in_edges = G.in_degree(i+1)
            predecessors = list(G.predecessors(i+1))
# Increment PageRank of current node to sum of PageRanks of all nodes linked to current node divided by number of outward links for each of those nodes
            for j in range(0,number_in_edges):
                output_ranks[i] = output_ranks[i] + (input_ranks[predecessors[j]-1]/G.out_degree(predecessors[j]))
# Introduce the damping factor to increase convergence rate and ensure sum of all PageRanks is 1 at all iterations
        for i in range(0,number_of_nodes):
            output_ranks[i] = (damping_factor * output_ranks[i]) + ((1-damping_factor)/number_of_nodes)
# Calculate difference between PageRanks of current and next iteration to check for convergence
        difference = 0.0
        for i in range(0,number_of_nodes):
            difference = difference + abs(output_ranks[i] - input_ranks[i])
        #print(difference)
# Update PageRanks
        input_ranks = output_ranks
# Check for convergence against a tolerance value
        if(difference < error_tolerance):
            #print(number_iterations)
            break
# Return PageRanks of all the nodes
    return output_ranks
```

Matrix Power Method:

The matrix power method uses matrices, vectors and the operations associated with them to implement the PageRank algorithm. The primary difference from the iterative method is the creation of a square matrix whose dimensions are the same as that of the number of nodes in the webgraph.

This matrix is popularly termed as the Google matrix. The entries of the Google matrix are filled up based on the following rules:

(i) G[i][j] = 0 if no out-edge exists from node j to node i.

(ii) G[i][j] = 1/|F(j)| if an out-edge exists from node j to node i.

(iii) If case of a dangling node j, G[i][j] = 1/N for all i.

The constructed Google matrix is then multiplied by a vector representing the current PageRanks.

The method can be represented as:

PR(Next) = [G * PR(Current) * d] + (1-d)/N

Here, d is a scalar representing the damping factor.

The advantage of using the matrix power method is that efficient matrix operations can be utilised to make the algorithm much more faster. The drawback of this method is the vast amount of memory which would be required to construct the Google matrix in a real world scenario with a lot of nodes. This would be infeasible on a personal computer.

The algorithm is implemented as shown in the code snippet below.

```python
# Function to return the PageRank of all nodes in a given webgraph G using the matrix power method
def PageRankPowerMethod(G, damping_factor = 0.85, error_tolerance = 0.1, max_iterations = 100):
# Initialise number of iterations of algorithm to 0 and initialise a square matrix of size (number_of_nodes * number_of_nodes)
    number_iterations = 0
    number_of_nodes = G.number_of_nodes()
    GoogleMatrix = np.zeros((number_of_nodes,number_of_nodes))
# Initialise PageRank of all nodes to be initially the same value i.e 1/number_of_nodes
    InputRanks = np.full((number_of_nodes,1),1/number_of_nodes)
# Procedure to fill in values of matrix GoogleMatrix:
    # GoogleMatrix[i][j] = 0 if no link exists from node j to node i
    # GoogleMatrix[i][j] = 1/number of links from node j if link exists from node j to node i
    # In case of a dangling node j, randomly move to any of the nodes i.e GoogleMatrix[i][j] = 1/number_of_nodes for all i
    for i in range(0,number_of_nodes):
        number_out_edges = G.out_degree(i+1)
        if number_out_edges == 0:
            for j in range(0,number_of_nodes):
                GoogleMatrix[j][i] = 1/number_of_nodes
        else:
            successors = list(G.successors(i+1))
            for j in range(0,number_out_edges):
                GoogleMatrix[successors[j]-1][i] = 1/number_out_edges
# Iterations of PageRank Algorithm
    while number_iterations < max_iterations:
        number_iterations = number_iterations + 1
# Pagerank in next iteration = d(G * R) + (1-d)/N
        OutputRanks = np.matmul(GoogleMatrix,InputRanks) * damping_factor + ((1-damping_factor)/(number_of_nodes))
# Calculate difference between PageRanks of current and next iteration to check for convergence
        difference = sum(sum(abs(OutputRanks - InputRanks)))
        #print(difference)
# Update PageRanks
        InputRanks = OutputRanks
# Check for convergence against a tolerance value
        if(difference < error_tolerance):
            #print(number_iterations)
            break
# Return PageRanks of all the nodes
    return OutputRanks
```

Inner-Outer Iteration Method:

This method works based on the observation that the smaller the damping factor, the easier it is for the algorithm to converge. This fact is combined with representing the algorithm as a system of linear equations to come up with a technique that is faster than the matrix power method while also maintaining the same level of memory usage. The method works based on an iterative scheme where each iteration requires solving a linear system similar in

algebraic structure to the original one but with a lower damping factor thus converging faster.

The method is described in the image below.

Algorithm 1 basic inner-outer iteration

Input: $P, \alpha, \beta, \tau, \eta, \mathbf{v}$
Output: $\mathbf{x}$
$\mathbf{x} \leftarrow \mathbf{v}$
$\mathbf{y} \leftarrow P\mathbf{x}$
while $\|\alpha\mathbf{y} + (1-\alpha)\mathbf{v} - \mathbf{x}\|_1 \geq \tau$
    $\mathbf{f} \leftarrow (\alpha - \beta)\mathbf{y} + (1-\alpha)\mathbf{v}$
    repeat
        $\mathbf{x} \leftarrow \mathbf{f} + \beta\mathbf{y}$
        $\mathbf{y} \leftarrow P\mathbf{x}$
    until $\|\mathbf{f} + \beta\mathbf{y} - \mathbf{x}\|_1 < \eta$
end while
$\mathbf{x} \leftarrow \alpha\mathbf{y} + (1-\alpha)\mathbf{v}$

x represents a vector with the PageRanks of all the nodes.

v represents the personalisation vector used to initialise x.

P represents the Google matrix.

α represents the desired larger damping factor.

β represents the smaller damping factor used during inner iterations.

τ represents the outer iteration tolerance factor.

η represents the inner iteration tolerance factor.

(I − αP)x = (1 − α)v represents the linear equation which is being solved.

β must be smaller than α for this implementation to work efficiently.
Typically values of $\alpha = 0.85$ and $\beta = 0.5$ are used.[8]

The algorithm is implemented as shown in the code snippet below.

```
def PageRankInnerOuterMethod(G, damping_factor_alpha = 0.85, damping_factor_beta = 0.5, error_tolerance_outer = 0.1,error_tolerance_inner = 0.1, max_iterations_outer = 100, max_iterations_inner = 100):
# Inner Outer iteration solves for a smaller damping factor first and then uses that result to solve for the given damping factor
    if (damping_factor_alpha < damping_factor_beta):
        return "Error: Alpha must be greater than Beta"
# Initialise number of iterations of algorithm to 0 and initialise a square matrix of size (number_of_nodes * number_of_nodes)
    number_iterations = 0
    number_of_nodes = G.number_of_nodes()
    GoogleMatrix = np.zeros((number_of_nodes,number_of_nodes))
# Initialise PageRank of all nodes to be initially the same value i.e 1/number_of_nodes
    InputRanks = np.full((number_of_nodes,1),1/number_of_nodes)
# Procedure to fill in values of matrix GoogleMatrix:
# GoogleMatrix[i][j] = 0 if no link exists from node j to node i
# GoogleMatrix[i][j] = 1/number of links from node j if link exists from node j to node i
# In case of a dangling node j, randomly move to any of the nodes i.e GoogleMatrix[i][j] = 1/number_of_nodes for all i
    for i in range(0,number_of_nodes):
        number_out_edges = G.out_degree(i+1)
        if number_out_edges == 0:
            for j in range(0,number_of_nodes):
                GoogleMatrix[j][i] = 1/number_of_nodes
        else:
            successors = list(G.successors(i+1))
            for j in range(0,number_out_edges):
                GoogleMatrix[successors[j]-1][i] = 1/number_out_edges
# Personalisation Vector set to be equal to 1/number_of_nodes for all nodes i.e all nodes are equally likely to be visited in the case of a random choice
    personalisation_vector = np.full((number_of_nodes,1),1/number_of_nodes)
# Initialise PageRanks to be equal to the Personalisation Vector i.e all equal
    InputRanks = personalisation_vector
# y vector used during Inner-Outer Iteration
    y = np.matmul(GoogleMatrix,InputRanks)
# Pagerank in next Outer iteration = alpha(G * R) + (1-alpha)/N
    OutputRanks = (damping_factor_alpha * y + (1-damping_factor_alpha) * personalisation_vector)
# Calculate difference between PageRanks of current and next Outer iteration to check for convergence
    difference_outer = sum(sum(abs(OutputRanks - InputRanks)))
# Repeat Outer iterations until convergence
    while (difference_outer > error_tolerance_outer):
# Calculate f as (alpha-beta)y + (1-alpha)/N
# f used in Inner iterations
        f = (damping_factor_alpha - damping_factor_beta) * y + (1 - damping_factor_alpha) * personalisation_vector
# Set difference_inner to 100 to ensure atleast 1 iteration of inner loop
        difference_inner = 100
# Repeat Inner iterations until convergence
        while (difference_inner > error_tolerance_inner):
# Update InputRanks and y in each Inner iteration till the InputRanks converge for smaller damping factor beta
            InputRanks = f + (damping_factor_beta * y)
            y = np.matmul(GoogleMatrix,InputRanks)
# Calculate difference between PageRanks of current and next Inner iteration to check for convergence
            difference_inner = sum(sum(abs(f + (damping_factor_beta * y) - InputRanks)))
```

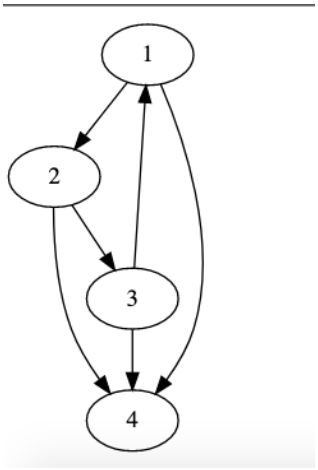
```
# Update OutputRanks in each Outer iteration till the OutputRanks and InputRanks converge for damping factor alpha
        OutputRanks = (damping_factor_alpha * y + (1-damping_factor_alpha) * personalisation_vector)
        difference_outer = sum(sum(abs(OutputRanks - InputRanks)))
# Set final PageRanks of all nodes in OutputRanks
    OutputRanks = (damping_factor_alpha * y + (1-damping_factor_alpha) * personalisation_vector)
# Return PageRanks of all the nodes
    return OutputRanks
```

Results:

```
H = nx.DiGraph()
H.add_edge(1,2)
H.add_edge(2,3)
H.add_edge(3,1)
H.add_edge(1,4)
H.add_edge(2,4)
H.add_edge(3,4)
damping_factor = 1
error_tolerance = 0.1
max_iterations = 100
pageranks_power = PageRankPowerMethod(H, damping_factor, error_tolerance, max_iterations)
pageranks_iterative = PageRankIterative(H, damping_factor, error_tolerance, max_iterations)
pageranks_inner_outer = PageRankInnerOuterMethod(H,damping_factor)
print(pageranks_power)
print(pageranks_iterative)
print(pageranks_inner_outer)
```

```
[[0.203125]
 [0.203125]
 [0.203125]
 [0.390625]]
[0.203125, 0.203125, 0.203125, 0.390625]
[[0.203125]
 [0.203125]
 [0.203125]
 [0.390625]]
```

As seen in the code snippet given above, the 3 different methods of implementing the PageRank algorithm have all been run on a simple toy webgraph which looks like this:



The purpose of using this toy webgraph is to make it easy for us to also manually compute the PageRanks and verify that all the methods are giving the right results. Note that a damping factor of 1 has been used here as the model is fairly small and won't take long to converge.

Damping Factor Analysis

As mentioned in the section above, the damping factor plays a crucial role in the convergence of the PageRank algorithm and to the PageRanks themselves.

Iterative

At $t = 0$, an initial probability distribution is assumed, usually

$$PR(p_i; 0) = \frac{1}{N}.$$

At each time step, the computation, as detailed above, yields

$$PR(p_i; t+1) = \frac{1-d}{N} + d \sum_{p_j \in M(p_i)} \frac{PR(p_j; t)}{L(p_j)},$$

As illustrated above, PageRanks are determined by a combination of a term computed at each iteration which will end up being different multiplied by the damping factor and a non-zero constant multiplied by one minus the damping factor. The damping factor determines the weightage given towards the PageRank computations. Therefore, a high damping factor would lead to more importance given towards the PageRank computations and a lower damping factor would lead to less importance given towards the PageRank computations.

It can be observed that as the value of the damping factor decreases, the algorithm converges much faster. This is due to the fact that all the PageRanks are forced towards the non-zero constant rather than the actual

computation of the PageRank at each iteration. This can be illustrated by taking the two extreme cases:

Case1:

If the damping factor is 0, then all nodes are assigned the same PageRank in iteration 1 i.e 1/N. Therefore, the algorithm converges in just a single iteration in this case as this is the same as the initial PageRanks. However, this would actually not be representative of the true ranks of the webpages in this webgraph.

Case2:

If the damping factor is 1, then all nodes are assigned PageRank purely based on the computation done at each iteration. The non-zero constant term would be having no effect whatsoever. The convergence for this case would be highly dependent on the structure of the particular webgraph and could take a large number of iterations to actually converge. However, the PageRanks obtained in this case would be perfectly representative of the actual ranks of the webpages of this webgraph.

Therefore, the choice of the damping factor must be chosen optimally and very carefully after considering what exactly is desired for that particular task. Setting too high a damping factor could lead to very slow convergence with large webgraphs. Setting too low a damping factor could lead to PageRanks not being actually representative of the underlying webpages. After careful analysis, most researchers have arrived at the conclusion that a

damping factor of 0.85 is optimal for most applications related to web analysis. However, in some applications of the PageRank algorithm to biological data, a damping factor of around 0.31 has been found to optimal.[4]

The code snippet below illustrates the impact of the damping factor.

```
# Function to analyze PageRank accuracy and computation time for different damping factor values
def DampingFactorAnalysis():
# Initially set damping factor to 1 and other parameters for the PageRank Algorithm
    curr_damping_factor = 1
    error_tolerance = 0.001
    max_iterations = 100
    difference = 0
    true_pageranks = 0
    max_time = 0
# Run PageRank Algorithm for different values of damping factor and find which value leads to fastest convergence
    for i in range(0,11):
        curr_damping_factor = 1 - (i/10)
# Load a random graph with 20000 nodes and 1/1000th probability of an edge being added between 2 nodes
        G = nx.fast_gnp_random_graph(20000,1/1000).to_directed()
        G = nx.convert_node_labels_to_integers(G,1)
        number_of_nodes = G.number_of_nodes()
# Measure time taken for the PageRank to be computed using the iterative method
        start_time = t.process_time()
        pageranks = PageRankPowerMethod(G, curr_damping_factor, error_tolerance, max_iterations)
        end_time = t.process_time()
        total_time = end_time - start_time
# Set maximum time to time if i is 0
        if (i == 0):
            max_time = total_time
            true_pageranks = pageranks
        error = 0.0
        for j in range(0,number_of_nodes):
            error = difference + abs(true_pageranks[j] - pageranks[j])
        print("Damping Factor: ",(10-i)/10, "Time Taken: ", total_time, "Error compared to True PageRank: ", error)
```

```
# Analyze different damping factors
DampingFactorAnalysis()
```

```
Damping Factor:  1.0 Time Taken:  4.700951138000164 Error compared to True PageRank:  [0.]
Damping Factor:  0.9 Time Taken:  3.035951218000264 Error compared to True PageRank:  [1.31736264e-05]
Damping Factor:  0.8 Time Taken:  2.8101013580003382 Error compared to True PageRank:  [3.00174591e-06]
Damping Factor:  0.7 Time Taken:  2.8382574340002975 Error compared to True PageRank:  [8.05481481e-06]
Damping Factor:  0.6 Time Taken:  2.6750187079996977 Error compared to True PageRank:  [2.06254881e-08]
Damping Factor:  0.5 Time Taken:  2.6565456550001727 Error compared to True PageRank:  [7.94722487e-06]
Damping Factor:  0.4 Time Taken:  2.464489378000053 Error compared to True PageRank:  [5.20967198e-06]
Damping Factor:  0.3 Time Taken:  2.5023031830000946 Error compared to True PageRank:  [2.98449149e-06]
Damping Factor:  0.2 Time Taken:  2.5237939290000213 Error compared to True PageRank:  [4.99381086e-06]
Damping Factor:  0.1 Time Taken:  2.340845690000151 Error compared to True PageRank:  [3.47027975e-06]
Damping Factor:  0.0 Time Taken:  2.1020001020001473 Error compared to True PageRank:  [2.19005726e-06]
```

As observed in the output of the above cell, as the damping factor decreases the algorithm converges much faster.

Approaches to Efficiently Implement the PageRank Algorithm

This section describes various approaches taken to make the PageRank algorithm converge faster for a given damping factor. The algorithm must also converge to the same PageRank values with and without these efficient implementation techniques.

Altering the Personalisation Vector:

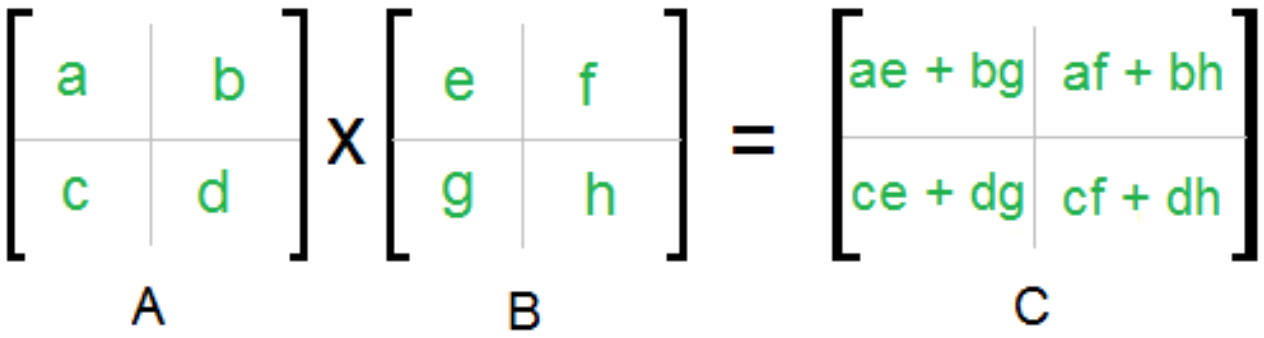
This approach is based on the idea that initialising the PageRanks to a specific value closer to their true value rather than making them equal initially might lead to faster convergence. In order to test this idea, the PageRank of a particular node was initialised to the ratio of the number of in-edges to that particular node and the total number of edges.

However, the results were not what was desired as the algorithm now converged to a set of PageRanks which were different from the PageRanks obtained without this approach. The approach was tested using the recommended damping factor of 0.85.

Divide and Conquer Approach Matrix Multiplication:

This approach is based on an implementation of matrix multiplication where a square matrix of even order is split up into four smaller square matrices and this procedure is repeatedly recursively till the order of the matrix obtained is small enough to calculate efficiently. This divide and conquer method is then used in the PageRank algorithm so that the algorithm may converge sooner.[5]

However, the results were not what was desired in this case as well. Although the algorithm converged to the same PageRanks with and without this technique, this technique actually turned out to be more time consuming and converged slower than the original implementation. Looking into the matrix multiplication technique employed by the Numpy library explains these results. Numpy already uses divide and conquer approach during matrix multiplication internally and also performs a variant of this for square matrices of odd order as well. Therefore, implementing the divide and conquer approach for even order square matrices explicitly lead to worse results than just utilising the matrix multiplication in the Numpy library.



A, B and C are square metrices of size N x N
a, b, c and d are submatrices of A, of size N/2 x N/2
e, f, g and h are submatrices of B, of size N/2 x N/2

Compressed Sparse Row Matrix Representation:

The failure of the previous approach led to some reading regarding sparse matrices and the realisation that the matrix operation need not be faster for all cases but just for the specific case of a sparse matrix as typically all webgraphs are represented by sparse matrices i.e the Google matrix is sparse for all webgraphs.

A matrix is defined to be sparse if most of its entries are zero. The number of zero valued elements divided by the total number of elements is called the sparsity of a matrix. A matrix is considered sparse when it has a sparsity value of greater than 0.5. This is usually the case for a typical webgraph. An analysis of some publicly available webgraphs shows that the ratio of number of nodes to edges ranges from 1:10 to 1:20.

All of the zero entries of a matrix can be skipped over rather than actually using them for matrix multiplication as anything multiplied by zero results in zero. This useless calculation can be particularly costly given the large size of webgraphs and their very sparse nature.

A Compressed Sparse Row (CSR) format is a way to represent matrices which is specifically designed for efficient operations on sparse matrices. CSR format is based on the fact that there are very few non-zero entries and it is better to just store these non-zero entries as (Row-Column-Data) tuples. Apart from this the dimensions of the matrix are also stored. Only the information about these non-zero entries are now used during matrix operations.[6]

Utilising the CSR format in the PageRank algorithm leads to much faster convergence and also leads to accurate results with the PageRanks with and without this efficient implementation technique being the same.

The code snippets below illustrate the results of running the Matrix Power implementation of the PageRank algorithm with and without the CSR format.

```python
# Function to return the PageRank of all nodes in a given webgraph G using the improved matrix power method
def PageRankPowerMethodSparse(G, damping_factor = 0.85, error_tolerance = 0.1, max_iterations = 100):
# Initialise number of iterations of algorithm to 0 and initialise 3 lists for rows,columns and data of the sparse matrix
    number_iterations = 0
    number_of_nodes = G.number_of_nodes()
    rows = []
    columns = []
    data = []
# Initialise PageRank of all nodes to be initially the same value i.e 1/number_of_nodes
    InputRanks = np.full((number_of_nodes,1),1/number_of_nodes)
# Procedure to fill in values of matrix GoogleMatrix:
    # GoogleMatrix[i][j] = 0 if no link exists from node j to node i
    # GoogleMatrix[i][j] = 1/number of links from node j if link exists from node j to node i
    # In case of a dangling node j, randomly move to any of the nodes i.e GoogleMatrix[i][j] = 1/number_of_nodes for all i
    for i in range(0,number_of_nodes):
        number_out_edges = G.out_degree(i+1)
        if number_out_edges == 0:
            for j in range(0,number_of_nodes):
                rows.append(j)
                columns.append(i)
                data.append(1/number_of_nodes)
        else:
            successors = list(G.successors(i+1))
            for j in range(0,number_out_edges):
                rows.append(successors[j]-1)
                columns.append(i)
                data.append(1/number_out_edges)
# Construct GoogleMatrixSparse in Compressed Sparse Matrix(CSR) format using the data,rows and columns lists.
    GoogleMatrixSparse = csr_matrix((data,(rows,columns)), shape=(number_of_nodes,number_of_nodes))
# Iterations of PageRank Algorithm
    while number_iterations < max_iterations:
        number_iterations = number_iterations + 1
# Pagerank in next iteration = d(G * R) + (1-d)/N
        OutputRanks = GoogleMatrixSparse.dot(InputRanks) * damping_factor + ((1-damping_factor)/(number_of_nodes))
# Calculate difference between PageRanks of current and next iteration to check for convergence
        difference = sum(sum(abs(OutputRanks - InputRanks)))
        #print(difference)
# Update PageRanks
        InputRanks = OutputRanks
# Check for convergence against a tolerance value
        if(difference < error_tolerance):
            #print(number_iterations)
            break
# Return PageRanks of all the nodes
    return OutputRanks
```

```python
I = nx.fast_gnp_random_graph(20000,1/1000).to_directed()
I = nx.convert_node_labels_to_integers(I,1)
print(I.number_of_nodes())
print(I.number_of_edges())
damping_factor = 0.85
error_tolerance = 0.1
max_iterations = 100
start_time = t.process_time()
pageranks = PageRankPowerMethod(I,damping_factor, error_tolerance, max_iterations)
end_time = t.process_time()
time_before_sparse_matrix_vector_multiplication = end_time - start_time
print("PageRanks: ",pageranks)
print("Sum of PageRanks: ",sum(pageranks))
print("Time Taken: ",time_before_sparse_matrix_vector_multiplication)
```

```
20000
399462
PageRanks:  [[5.36079068e-05]
 [4.77602779e-05]
 [4.17043258e-05]
 ...
 [4.27085053e-05]
 [6.17615053e-05]
 [6.50482313e-05]]
Sum of PageRanks:  [1.]
Time Taken:  2.340573243000108
```

```
print(I.number_of_nodes())
print(I.number_of_edges())
damping_factor = 0.85
error_tolerance = 0.1
max_iterations = 100
start_time = t.process_time()
pageranks = PageRankPowerMethodSparse(I,damping_factor,error_tolerance)
end_time = t.process_time()
time_after_sparse_matrix_vector_multiplication = end_time - start_time
print("PageRanks: ",pageranks)
print("Sum of PageRanks: ",sum(pageranks))
print("Time Taken: ",time_after_sparse_matrix_vector_multiplication)
```

```
20000
399462
PageRanks:  [[5.36079068e-05]
 [4.77602779e-05]
 [4.17043258e-05]
 ...
 [4.27085053e-05]
 [6.17615053e-05]
 [6.50482313e-05]]
Sum of PageRanks:  [1.]
Time Taken:  0.7806611370001519
```

As seen in the 2 above snippets, running the algorithm with the same damping factor of 0.85 leads to the same set of PageRanks in both cases.

It can also be observed that the usage of the CSR format leads to a runtime of 0.78 seconds which is around 3 times as fast as the original runtime of 2.34 seconds for the same webgraph.

Future Scope and Conclusion

There is a lot of scope for studies looking into some of the other webpage ranking algorithms mentioned in the introduction section such as the HITS algorithm used by ask.com, the IBM CLEVER project, the TrustRank and Hummingbird algorithms.

Studies can also be done into some of the other implementations of the PageRank algorithm apart from the three mentioned in this report and how to make them more efficient.

Studies can also be done about the emergence of webpages that try to artificially inflate their PageRank by paying other webpages to provide links to theirs. Research can also be done about the attempts made by Google to identify such webpages.

Further, a thorough study can also be done about the applications of ranking algorithms outside the domain of web analysis such as biological data, citation analysis and social network analysis.

Therefore, the importance of the research paper published in 1998 can still be seen after more than 2 decades. With the exponential growth of the World Wide Web, providing users with quality information is important now more than ever. The presence of much bigger computational resources gives the opportunity for almost any individual to analyse and experiment with this algorithm. The webpage ranking problem will continue to be a very crucial problem for years to come and there will certainly be more ground breaking research in this domain in the years to come.

Bibliography

1.“World Wide Web.” 2019. *Wikipedia*. Wikimedia Foundation. https://en.wikipedia.org/wiki/World_Wide_Web.

2.*. “How Many Websites Are There Around the World?” *Mill For Business*, Netcraft, 25 Sept. 2019, https://www.millforbusiness.com/how-many-websites-are-there/.

3.“2019's Search Engine Statistics: 99firms - November 2019.” *99firms.Com*, 24 Oct. 2019, https://99firms.com/blog/search-engine-statistics/.

4.“PageRank.” 2019. *Wikipedia*. Wikimedia Foundation. https://en.wikipedia.org/wiki/PageRank.

5.“Matrix Multiplication Algorithm.” *Wikipedia*, Wikimedia Foundation, 8 Nov. 2019, https://en.wikipedia.org/wiki/Matrix_multiplication_algorithm#Divide_and_conquer_algorithm.

6.“Sparse Matrix.” *Wikipedia*, Wikimedia Foundation, 7 Nov. 2019, https://en.wikipedia.org/wiki/Sparse_matrix.

7.Page, Lawrence & Brin, Sergey & Motwani, Rajeev & Winograd, Terry. (1998). The PageRank Citation Ranking: Bringing Order to the Web.

8.Gleich, David & Gray, Andrew & Greif, Chen & Lau, Tracy. (2010). An Inner-Outer Iteration for Computing PageRank. SIAM J. Scientific Computing. 32. 349-371. 10.1137/080727397.