The Google PageRank Algorithm

James Ngai, Maxwell Feng

November 2022

1 Introduction

When searching on Google, Google outputs a list of websites. The websites at the top of the page are accessed more frequently than websites near the bottom.

This idea of ordering websites is called Google PageRank. The orderering is based on the likelihood of visiting each website. Links that are more likely to be visited are placed higher than links that are less likely to be visited.

To implement PageRank, we create a matrix that predicts which link a person will click given that a person is on a certain website. Then, we will be able to find a steady-state vector that lists the destinations and the probability of ending on them during a random walk. The most probable destinations will be ranked highest.

This paper describes algorithms and computations used to create PageRank and end with an implementation of PageRank. We will also test our PageRank algorithm on smaller test cases.

We also will implement the Hyperlink-Induced Topic Search (HITS) algorithm, which is another algorithm that rates web pages. In the HITS algorithm, there is a notion of both Hubs and Authorities. A website has a high Hub score if the website points to many credible websites. Similarly, a website has a high Authority score if the website is pointed to by many credible websites. The HITS algorithm produces a ranking for Hubs and Authorities based on how many edges point to or from a node. Note that the ranking for Authorities in HITS should be similar to the rankings for websites in the PageRank algorithm.

After testing with a small amount of websites, we will compare the results of the HITS algorithm and PageRank.

2 Mathematical Background

The linear algebra background required to understand Google PageRank and HITS are primarily matrices, row/column space, Markov matrices and steady-state vectors. Some basic graph theory, including vertices, edges, indegrees, outdegrees, and undirected/directed graphs, are needed to understand PageRank and HITS.

Matrices are a 2-dimensional array of elements. An $n \times m$ matrix consists of n rows and m columns.

The row/column space is the span of the rows or columns of a particular matrix.

For a given graph, a node N has indegree i if there are i other nodes that point to N.

For a given graph, a node N has outdegree j if N points to j other nodes.

Markov matrices are matrices that have all non-negative entries, and the sum of each column vector is equal to 1. These matrices are also called probability matrices.

Adjacency matrices are often used to represent undirected graphs, where there is a 1 in entry ij if there exists an edge between node i and node j and 0's everywhere else. These matrices are symmetric since if there is an edge between nodes i and j, there will be an entry at both ij and ji.

After normalizing the columns of an adjacency matrix, the resulting matrix becomes a Markov matrix. However, we assumed that the graph was undirected. Representing a directed graph as a Markov matrix has issues. In a directed graph, if a node has indegrees but no outdegrees, then there will be a zero column in the probability matrix. Since all Markov matrices have columns that sum to 1, the resulting matrix cannot be represented as a Markov matrix. This issue will be addressed in the "Algorithms and Computations" section.

If we are able to represent an undirected graph as a Markov matrix, then we can find steady-state vectors, which is the long-term probability that the system will be in each state. Steady-state vectors are a scalar multiple of the eigenvector that corresponds to eigenvalue 1.

We can see that the steady-state is this eigenvector since all eigenvalues for a Markov matrix are less than or equal to 1. Also note that for some $n \times n$ Markov matrix A, $A^k\vec{x} = c_1\lambda_1^k\vec{v_1} + c_2\lambda_2^k\vec{v_2} + ...$ for eigenvalues $\lambda_1, \lambda_2, ...$ and eigenvectors $\vec{v_1}, \vec{v_2}, ...$ As k grows to infinity, for some $1 \leq i \leq n$, $\lambda_i^k \to 0$ if $\lambda_i < 1$. Then, $A^k\vec{x}$ approximates to $c_1\lambda_j^k\vec{v_j}$ where $\lambda_j = 1$, so $A^k\vec{x} \approx c_1\vec{v_j}$.

For PageRank, the entries in v_j are websites, and since we only care about the magnitudes of the entries relative to one another, the constant c_1 is irrelevant. The mathematics behind the PageRank relies on this long-term behavior since we want to find where someone will end up after a long random walk. If we find the long-term behavior of a random surfer, then we can find the most likely website that the random surfer will end up at.

3 Algorithms and Computations

3.1 PageRank

As mentioned before, to generate a PageRank, we try to answer the following question: if a random surfer randomly clicks on links, after a long period of time, where will the random surfer end up?

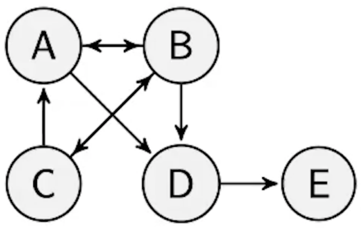
We cannot guarantee which website the random surfer will end up, but we can give probabilities of which websites the random surfer will end up.

To create a matrix that predicts which links will be visited, we must make the assumption that each link that is listed on a particular website has equal probability of being visited.

Based on this assumption, if there are n websites, we can create an $n \times n$ matrix (we will call this matrix P) to predict which link a random surfer will click.

In the matrix, we can index each website as an integer from 1 to n, and we can represent the probability of going from index i to index j as the entry P_{ij}. Note that all entries on the diagonals are 0 since we assume that a person does not stay on the same website.

For example, suppose we have the following graph, where each node represents a website.



We can create the matrix

$$\begin{pmatrix} 0 & \frac{1}{3} & \frac{1}{2} & 0 & 0 \\ \frac{1}{2} & 0 & \frac{1}{2} & 0 & 0 \\ 0 & \frac{1}{3} & 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{3} & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

However, this naive representation has a problem. Since E does not point to any nodes, after a long period of time of random walking, the random surfer will always end up at node E, and the random surfer will not be able to move to any other websites. This will make node E the highest value on our PageRank even though realistically, node E is not very likely to be visited since it is pointed to by only one node.

To solve this issue, we can assume that once the person reaches a dead end, like website E, the person has an equal probability of visiting any website.

Thus, our new matrix becomes

$$\begin{pmatrix} 0 & \frac{1}{3} & \frac{1}{2} & 0 & \frac{1}{5} \\ \frac{1}{2} & 0 & \frac{1}{2} & 0 & \frac{1}{5} \\ 0 & \frac{1}{3} & 0 & 0 & \frac{1}{5} \\ \frac{1}{2} & \frac{1}{3} & 0 & 0 & \frac{1}{5} \\ 0 & 0 & 0 & 1 & \frac{1}{5} \end{pmatrix}$$

Now, this matrix is a Markov matrix.

Finding the steady-state vector of this matrix assumes that the random surfer keeps clicking on links. However, we should also consider if the random surfer decides to settle on a website. To solve this problem, we introduce a variable called the damping factor.

The damping factor is the probability that a random surfer on the Internet will eventually stop clicking on links on a given webpage and eventually click on a random link. We will denote the damping factor as d.

Since d is the probability that the person stops and clicks on a random link, $1-d$ is the probability that the person continues clicking on links.

Thus, if we have n nodes, we can construct an $n \times n$ probability matrix P, as we did above, and multiply it by the probability that people continue clicking links. Then, we can add it to the probability that people settle on a page. The probability that people settle on a page is represented by the matrix dK, where $K = \frac{1}{n}\mathbf{1}$ (note that $\mathbf{1}$ is the $n \times n$ matrix where all entries are 1; this is because we assume each website has equal probability of being clicked on).

Our matrix then becomes

$$G = (1-d)P + dK$$

Google typically uses $d = 0.15$, which is based on the frequency that people use bookmarks, so for the rest of this project, we will use $d = 0.15$. Our matrix then becomes:

$$G = 0.85P + 0.15K$$

We guarantee all entries are positive, so G is a Markov matrix. To compute the steady-state vector, we could either find the eigenvector with eigenvalue 1 through computation or iteratively (we continuously multiply the probability matrix). We choose to find the eigenvector iteratively since this algorithm is in a lower complexity class and thus faster.

The method works by continuously multiplying our resulting vector by G until the norm of our vector is less than some specified ϵ. Since we are trying to find the long-term behavior of $G^k\vec{x}$, we can simply continue multiplying on the left by G until we find a "satisfactory" vector.

To determine if a vector is "satisfactory," we choose an ϵ that measures the difference between the norms of the previous two vectors. Since $G^k\vec{x}$ converges to some vector, after a long time, the vector will be changing by a very small amount. If the ϵ is small enough, then our resulting vector will be approximately equal to the actual steady-state vector.

3.2 HITS

The purpose of HITS is to rank websites based off of two factors, Authority and Hub. Hub is defined as a measurement of the quality of a website's outgoing links. If a website were to rank to many credible sites(sites where lots of links point to), the website is considered to have a high Hub.

Authority is defined as a measurement of the number of websites that link to a website. If a website were to have many links directed to it's site from credible sites(websites that have a high volume of inbound links), that website has a high Authority score.

HITS works by iteratively updating the scores for Authority and Hub through repeated matrix multiplication. The authority score is explicitly calculated by summing the Hub score of its parents. The Hub score is explicitly calculated by summing the Authority score of its children.

To calculate the Authority using matrices, we can initialize a vector $\vec{v_0} = (1, ..., 1)$, $\vec{w_0} = (1, ..., 1)$ to be multiplied to our Adjacency matrix. After the first iteration $A\vec{v_0}$, the resulting vector $\vec{w_1}$ is the sum of all websites linking to the index i website.

$$\vec{w_1} = \left(\sum_{i=1}^{n} A_{1i}, \sum_{i=1}^{n} A_{2i}, .., \sum_{i=1}^{n} A_{ni} \right) \text{ or } \vec{w} = A\vec{v_0}$$

To calculate the Hub scores, we now need to take the vertical sum of the columns of the Adjacency matrix. Thus, we can follow the same process for authority but with A^T to take column sums.

$$\vec{v_1} = \left(\sum_{i=1}^{n} A_{1i}^T, \sum_{i=1}^{n} A_{2i}^T, .., \sum_{i=1}^{n} A_{ni}^T \right) \text{ or } \vec{v_1} = A^T\vec{w_0}$$

On the second iteration, the $\vec{w}$ is now multiplied to $A^T\vec{w}$ and $\vec{v}$ is now multiplied to $A\vec{v}$. This swap occurs because the new Authority and Hub rankings now creates a weighted calculation for Authority and Hub. With the next iteration, sites with a greater Hub score which link to sites will be given greater Authority. This intuitively makes sense as sites which credibly link to other sites will promote the Authority of the sites it links to. Similarly, a site with greater Authority, if linked to, promotes the Hub score of the website linking to the Authority. Thus, as more iterations occur, the quality of the links are increasingly considered as opposed to just the number of links.

This results in

$$\text{Authority} \quad \vec{v_2} = A^T A\vec{v_0}$$

$$\text{Hub} \qquad \vec{w_2} = AA^T\vec{w_0}$$

After n iterations, the largest eigenvalue σ_1^2 dominates, leading to the corresponding eigenvector with Authority/Hub rankings.

This leads to two possible approaches where HITS can iterate through language optimized matrix multiplications $A^T A\vec{v_o}$ and $A^T A\vec{w_o}$ or swap the vectors each iteration. In modern languages like Julia, the Power Method is likely preferred as matrix multiplications are optimized as opposed to for loops.

$$\text{Authority} \quad \vec{v_{2n}} = \left(A^T A\right)^n \vec{v_0}$$

$$\text{Hub} \qquad \vec{w_{2n}} = \left(AA^T\right)^n \vec{w_0}$$

4 Coding and Algorithm Design Decisions

We will now divide the code into sections and briefly explain the purpose of each section. To view the full code without interruptions, see the Appendix at the end of the paper.

4.1 PageRank

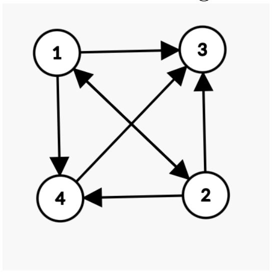
We first import some packages that allow us to use linear algebra concepts, like matrices.

```
using LinearAlgebra
using Distributions
```

Here is a small example of inputs that we can use:

```
websites = ["https://shp31337.github.io/",
"https://jamesn3.github.io/JamesN3/",
"https://www.amazon.com/", "https://www.desmos.com/"]

website_links = [["https://jamesn3.github.io/JamesN3/",
"https://www.amazon.com/", "https://www.desmos.com/"],
["https://shp31337.github.io/", "https://www.desmos.com/",
"https://www.amazon.com/"], [], ["https://www.amazon.com/"]]
```

In the first list, we have 4 websites, so our graph will include 4 nodes. Here is a graph to help visualize the nodes and edges:



1: https://shp31337.github.io/
2: https://jamesn3.github.io/JamesN3/
3: https://www.amazon.com/
4: https://www.desmos.com/

In the second list, the first node (https://shp31337.github.io/) will point to nodes 2, 3, and 4. The directed graph can thus be created by these two inputs.

Note that the 3rd element in the list is an empty list, which denotes that the 3rd website does not point to any other node (some mathematicians call this a sink).

The first list is parsed through so that we can represent each website as its index in the list. For each list at index i in the second list, the websites inside the respective list are nodes that website i will point to.

Implementation-wise, we can create a directed graph using adjacency matrices.

To create a directed graph, we first create a dictionary so that each website name is linked to its respective index. The dictionary takes in a website name as an input and outputs the index each website is assigned to.

Now that we have assigned an index to each website, we create an adjacency matrix to represent the edges.

The function list_to_adj creates such an $n \times n$ adjacency matrix. It iterates through the website links list for every website and places a 1 at index ij if there exists an edge from website i to j. All other entries contain 0.

```
A = zeros(length(websites), length(websites))
dict = Dict()
for i in 1:length(websites)
    dict[websites[i]] = Int(i)
end

# Implement into Adjacency matrix
function list_to_adj(A, websites, website_links)
    for i in 1:size(website_links, 1)
        for k in 1:size(website_links[i], 1)
            name = website_links[i][k]
            A[dict[name], i] = Int(floor(1))
        end
    end
    return Int.(A)
end

A = list_to_adj(A, websites, website_links)
```

Here is the adjacency matrix for our ongoing example:

```
display(A)

4 by 4 Matrix{Int64}:
 0  1  0  0
 1  0  0  0
 1  1  0  1
 1  1  0  0
```

However, since we are dealing with directed graphs, we may run into the problem discussed above: some columns may be zero columns if they have no outdegrees. Thus, we create a copy of our adjacency matrix, and we iterate through every column in the adjacency matrix and check that every entry in the column is 0 or not. If we find a zero column, then we change all entries in that column to $\frac{1}{n}$.

```
A_adjusted = copy(A)
A_adjusted = float.(A_adjusted)
for i in 1:size(A, 1)
    good = false
    for k in 1:size(A, 1)
        if (A[k,i] != 0)
            good = true
        end
    end
    if (good == false)
        for k in 1:size(A, 1)
            A_adjusted[k,i] = 1/ (size(A, 1))
        end
    end
end
```

In our example, after adjusting for zero columns, we get the following adjacency matrix:

```
display(A_adjusted)
```

```
4 by 4 Matrix{Float64}:
 0.0  1.0  0.25  0.0
 1.0  0.0  0.25  0.0
 1.0  1.0  0.25  1.0
 1.0  1.0  0.25  0.0
```

Now that we have all non-zero columns, we want to normalize all columns to the l1 norm so that they become Markov matrices.

```
# filled adjacency matrix with norm
```

```
sums = sum(A_adjusted, dims=1)
# A[:, 1]/5
D = zeros(size(A_adjusted, 1), size(A_adjusted, 1))
M = zeros(size(A_adjusted, 1), size(A_adjusted, 1))
for i in 1:size(A_adjusted, 1)
    D[i, i] = sums[i]
end

M = A_adjusted*inv(D)
```

Here is our example matrix normalized:

```
4 by 4 Matrix{Float64}:
 0.0       0.333333  0.25  0.0
 0.333333  0.0       0.25  0.0
 0.333333  0.333333  0.25  1.0
 0.333333  0.333333  0.25  0.0
```

Now, we implement the Power Method. We first create the matrix G (as described in the "Algorithms and Computations" section) and set it equal to trans. Then, we multiply the resulting vector v by trans continuously until we get that the difference in norms is less than ϵ.

Also note that the input vector does not matter since the long-term behavior will always converge to the same vector.

```
function eigen_steady_state(M, num_iters, d, epsilon, l_norm)
    trans = zeros(size(M, 1), size(M, 1))
    trans = (1-d)M + (d/size(M, 1)) * ones(size(M, 1), size(M, 1))
    # return matrix with count for each site
    v = ones(size(M, 1))

    for i in 1:num_iters
        v_last = v
        v = trans*v
        if norm(v - v_last, l_norm) < epsilon
            return v
        end
    end
    return v
end
```

Now, we are ready to implement the random walk. We simply call our steady_state function and rank the websites based on which value in v (the steady-state vector) is larger.

```
function randomwalk(M, num_iters, d, epsilon, l_norm)
    v = eigen_steady_state(M, num_iters, d, epsilon, l_norm)
    v_copy = copy(v)
    v_ranking = zeros(size(M, 1))
    for i in 1:length(v_copy)
        tuple_v = findmax(v_copy)
        v_ranking[i] = Int(tuple_v[2])
        v_copy[tuple_v[2]] = -10
    end
    return v_ranking
end
```

For our ongoing example, here are the website indices in order.

```
results = randomwalk(M, 1000, .15, 0.01, 2)

4-element Vector{Float64}:
 3.0
 4.0
 1.0
 2.0
```

Since we want to output websites in the end, we finish the algorithm by converting the indices back into websites by using our dictionary.

```
function arr_2_string(results, websites, dict)
    final_ranking = copy(websites)
    count = 1
    for num in results
        for tuple in dict
            if (tuple[2] == num)
                final_ranking[count] = tuple[1]
            end
        end
        count += 1
    end
    return final_ranking
end
```

Thus, we get the finalized order for our PageRank Algorithm:

```
display(arr_2_string(results, websites, dict))
```

```
4-element Vector{String}:
 "https://www.amazon.com/"
 "https://www.desmos.com/"
 "https://shp31337.github.io/"
 "https://jamesn3.github.io/JamesN3/"
```

4.2 HITS

In the HITS algorithm, we use the adjacency matrix we generated in PageRank. We simply implement the algorithm described in the "Algorithms and Computations" section.

```
function HITS(A, num_iters)
    x = ones(size(A, 1)) / size(A, 1)
    y = ones(size(A, 1)) / size(A, 1)
    for i in 1:num_iters
        x2 = A'*y
        y2 = A*x
        x = x2 / norm(x2)
        y = y2 / norm(y2)
    end
    x_copy = copy(x)
    y_copy = copy(y)
    x_ranking = zeros(length(x_copy))
    y_ranking = zeros(length(y_copy))
    for i in 1:length(x_copy)
        tuple_x = findmax(x_copy)
        x_ranking[i] = Int(tuple_x[2])
        x_copy[tuple_x[2]] = -10
        tuple_y = findmax(y_copy)
        y_ranking[i] = Int(tuple_y[2])
        y_copy[tuple_y[2]] = -10
    end
    # x is authorities
    #y is rankings
    return y_ranking, x_ranking
end
```

For our example, we list out the Authorities and Hubs in order of highest to lowest ranking.

```
authority, hub = HITS(A, 50)
println("Authority in order\n")
display(arr_2_string(authority, websites, dict))
println("Hub in order\n")
display(arr_2_string(hub, websites, dict))

Authority in order:

4-element Vector{String}:
 "https://www.amazon.com/"
 "https://www.desmos.com/"
 "https://shp31337.github.io/"
 "https://jamesn3.github.io/JamesN3/"

Hub in order:

4-element Vector{String}:
 "https://shp31337.github.io/"
 "https://jamesn3.github.io/JamesN3/"
 "https://www.desmos.com/"
 "https://www.amazon.com/"
```

4.3 Further Exploration into Lp spaces

The purpose is to compare the convergence rate for the $L1$ norm and $L2$ norm when doing RandomWalk in PageRank. This code tests values of ϵ and how many iterations of our randomwalk code is required to terminate.

```
    arr = []
    for i in 1:200
        push!(arr, eigen_steady_state_iterations(M, 100000000000000,
        .15, i/1000, 2))
    end
    Plots.plot(arr, xlabel = "epsilon in thousandths",
    ylabel = "Randomwalk Iterations")
    #.001 increments of epsilon
```

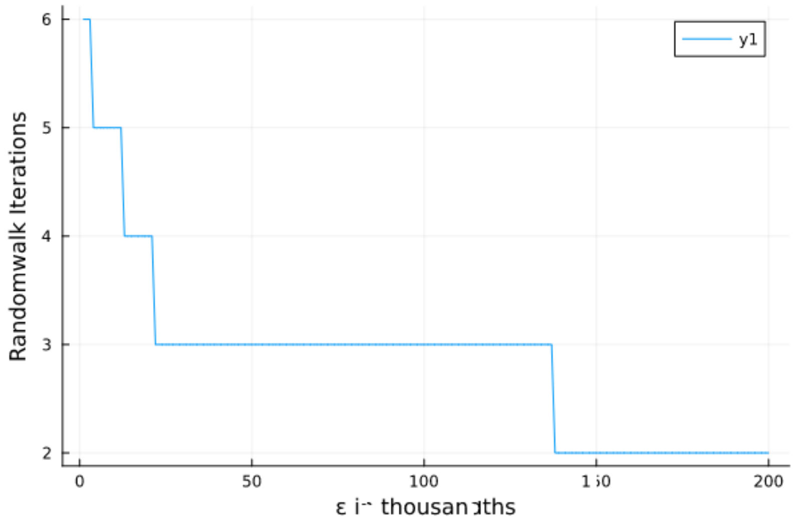


Figure 1: At low ϵ, (less than .025), the convergence rate in terms of iterations is seen to dramatically change, while when epsilon increases, the rate of convergence is increasingly less affected.

```
arr = []
for i in 1:200
    push!(arr, eigen_steady_state_iterations(M, 100000000000000,
    .15, i/1000, 1))
end
Plots.plot(arr, xlabel = "epsilon in thousandths",
ylabel = "Randomwalk Iterations")
```

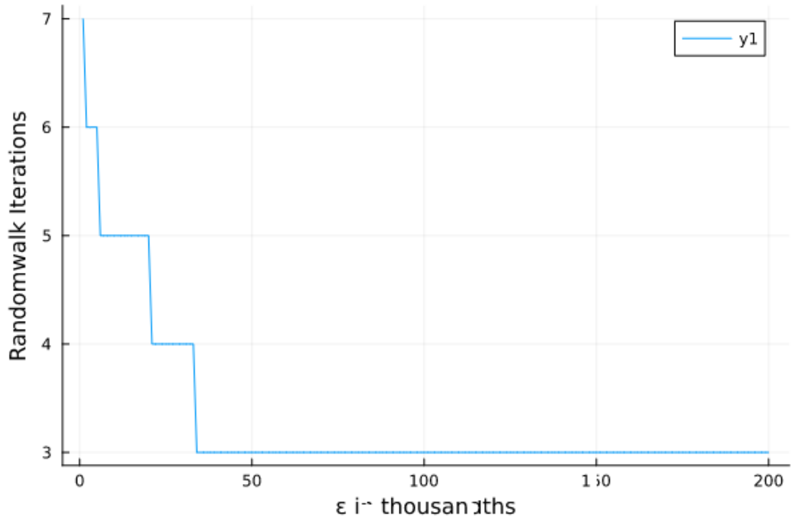


Figure 2: At low ϵ, (less than .025), the convergence rate in terms of

iterations is seen to dramatically change, while when epsilon increases, the rate of convergence is increasingly less affected.

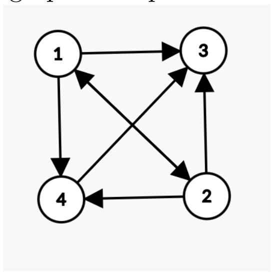
Conclusion: The rate of convergence for small datasets is rather insignificant, so the use of larger datasets will be required. Our findings do match a study from IIT Jodhpur which prove $l2$ norms converge faster than $l1$ norms.

5 Results and Interpretation of Computations

General Results:

First we will analyze our small example.

Here is a reminder of the graph to help us visualize the small example easily:



Index Correspondence:
1: https://shp31337.github.io/
2: https://jamesn3.github.io/JamesN3/
3: https://www.amazon.com/
4: https://www.desmos.com/

As shown in the graphic, node 3 is pointed to very often, so intuitively, node 3 should be ranked the highest. Also, both nodes 1 and 2 are pointed to by only one other node, so intuitively, these two nodes should be ranked lower.

Our final ranking using PageRank was 3, 4, 1, 2, and our final Authority ranking using HITS was 3, 4, 1, 2 as well. Note that the Authority ranking from HITS matches our ranking from PageRank. Based on intuition, our results seem valid.

Also, in the HITS algorithm, we have the Hub ranking as 1, 2, 4, 3. Both nodes 1 and 2 point to 3 other nodes, and node 3 points to 0 nodes. Thus, the Hub ranking also makes sense.

Both HITS and PageRank seem to work on examples with just a few nodes. With smaller examples, it seems impossible to determine if one algorithm is more efficient or accurate than the other. However, HITS provides more information since it outputs both a list of Hubs and Authorities. This information could potentially be used to implement an algorithm to rank credibility of websites since a higher Hub score usually correlates with higher credibility. Since the HITS algorithm give slightly more information, the HITS algorithm is more preferable.

Interesting Finding:
An interesting result from the power method involves the implementation of Julia. It was found that the runtime from stopping once convergence was achieved was slower than doing 10000000 iterations (a potentially larger number of iterations) of matrix multiplication without convergence. The following runtime was comapred with the two functions

1.2 second runtime with convergence

```
eigen_steady_state(M, 10000000, .15, 0.01, 2)
```

0.4 second runtime without convergence and a predetermined number of iterations

```
eigen_steady_state2(M, 10000000, .15, 0.01, 2)
```

This can be inferred to be a result of Julia's optimization as opposed to a difference in the number of iterations. Thus, consideration of language optimzations must be considered when designing efficent PageRank algorithms for large datasets.

6 Conclusion

In this project, we implemented Google's PageRank algorithm and the HITS algorithm using Markov chains. Although we were not able to find which algorithm was more efficient and accurate using small test cases, we deduced that both algorithms produced logically sound results for small examples.

With a larger graph of websites, we hypothesize that the results of HITS and PageRank will still be similar since both algorithms correlate with indegrees and outdegrees. Also, since our code finds the steady-state vector iteratively rather than algebraically, our algorithm is more time-efficient, so there should not be many issues with regard to runtime.

Another potentital future research topic includes optimizing Google PageRank when nodes have different weights (if some websites are more likely to be visited). We suspect that the probability matrix will have to be modified, but the algorithm to find the steady-state vector will remain the same.

7 Appendix

Full Code:

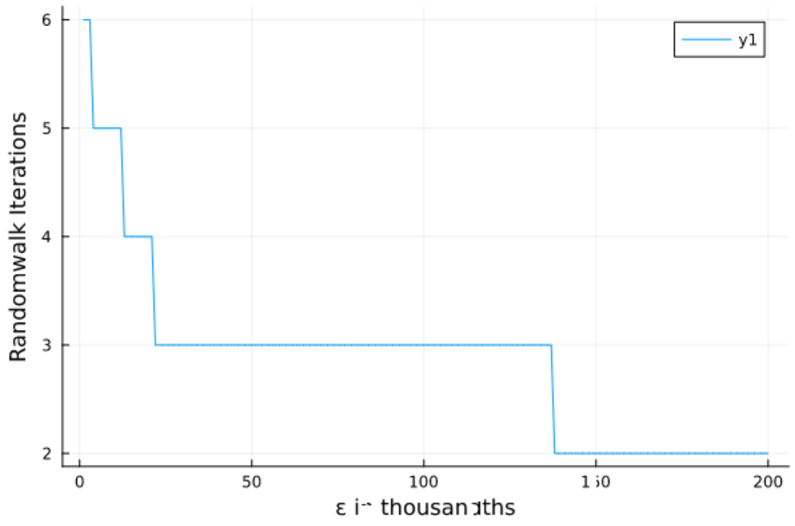
```
using LinearAlgebra
using Distributions

websites = ["https://shp31337.github.io/",
"https://jamesn3.github.io/JamesN3/",
"https://www.amazon.com/", "https://www.desmos.com/"]

website_links = [["https://jamesn3.github.io/JamesN3/",
"https://www.amazon.com/", "https://www.desmos.com/"],
["https://shp31337.github.io/", "https://www.desmos.com/",
"https://www.amazon.com/"], [], ["https://www.amazon.com/"]]

A = zeros(length(websites), length(websites))
dict = Dict()
for i in 1:length(websites)
    dict[websites[i]] = Int(i)
end

# Implement into Adjacency matrix
function list_to_adj(A, websites, website_links)
    for i in 1:size(website_links, 1)
        for k in 1:size(website_links[i], 1)
            name = website_links[i][k]
            A[dict[name], i] = Int(floor(1))
        end
    end
    return Int.(A)
end

A = list_to_adj(A, websites, website_links)

display(A)

4 by 4 Matrix{Int64}:
 0  1  0  0
 1  0  0  0
 1  1  0  1
 1  1  0  0

 A_adjusted = copy(A)
A_adjusted = float.(A_adjusted)
```

```
for i in 1:size(A, 1)
    good = false
    for k in 1:size(A, 1)
        if (A[k,i] != 0)
            good = true
        end
    end
    if (good == false)
        for k in 1:size(A, 1)
            A_adjusted[k,i] = 1/ (size(A, 1))
        end
    end
end

display(A_adjusted)

4 by 4 Matrix{Float64}:
 0.0  1.0  0.25  0.0
 1.0  0.0  0.25  0.0
 1.0  1.0  0.25  1.0
 1.0  1.0  0.25  0.0

 # filled adjacency matrix with norm

sums = sum(A_adjusted, dims=1)
# A[:, 1]/5
D = zeros(size(A_adjusted, 1), size(A_adjusted, 1))
M = zeros(size(A_adjusted, 1), size(A_adjusted, 1))
for i in 1:size(A_adjusted, 1)
    D[i, i] = sums[i]
end

M = A_adjusted*inv(D)

4 by 4 Matrix{Float64}:
 0.0       0.333333  0.25  0.0
 0.333333  0.0       0.25  0.0
 0.333333  0.333333  0.25  1.0
 0.333333  0.333333  0.25  0.0

 function eigen_steady_state(M, num_iters, d, epsilon, l_norm)
    trans = zeros(size(M, 1), size(M, 1))
    trans = (1-d)M + (d/size(M, 1)) * ones(size(M, 1), size(M, 1))
    # return matrix with count for each site
    v = ones(size(M, 1))
    for i in 1:num_iters
        v_last = v
        v = trans*v
        if norm(v - v_last, l_norm) < epsilon
            return v
        end
    end
    return v
end

function randomwalk(M, num_iters, d, epsilon, l_norm)
    v = eigen_steady_state(M, num_iters, d, epsilon, l_norm)
    v_copy = copy(v)
    v_ranking = zeros(size(M, 1))
    for i in 1:length(v_copy)
        tuple_v = findmax(v_copy)
        v_ranking[i] = Int(tuple_v[2])
        v_copy[tuple_v[2]] = -10
    end
    return v_ranking
end

results = randomwalk(M, 1000, .15, 0.01, 2)

4-element Vector{Float64}:
 3.0
 4.0
 1.0
 2.0

 function arr_2_string(results, websites, dict)
    final_ranking = copy(websites)
    count = 1
    for num in results
        for tuple in dict
            if (tuple[2] == num)
                final_ranking[count] = tuple[1]
            end
        end
        count += 1
    end
    return final_ranking
end

display(arr_2_string(results, websites, dict))
```

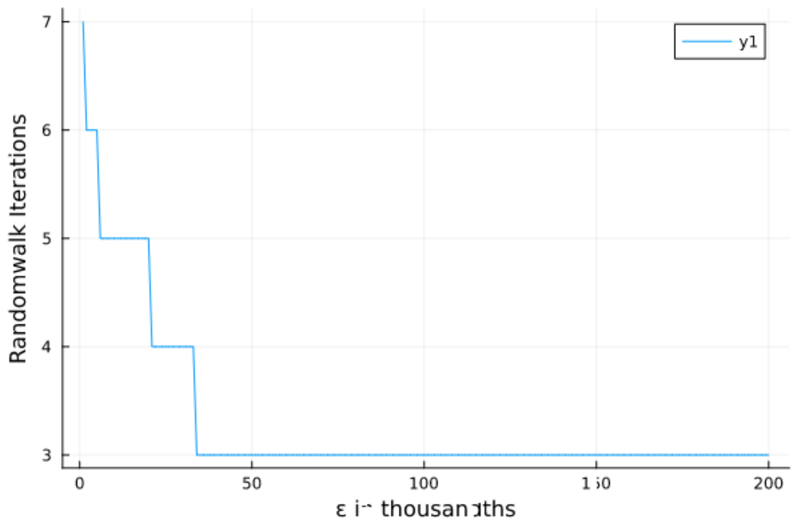
```
4-element Vector{String}:
 "https://www.amazon.com/"
 "https://www.desmos.com/"
 "https://shp31337.github.io/"
 "https://jamesn3.github.io/JamesN3/"

 function HITS(A, num_iters)
    x = ones(size(A, 1)) / size(A, 1)
    y = ones(size(A, 1)) / size(A, 1)
    for i in 1:num_iters
        x2 = A'*y
        y2 = A*x
        x = x2 / norm(x2)
        y = y2 / norm(y2)
    end
    x_copy = copy(x)
    y_copy = copy(y)
    x_ranking = zeros(length(x_copy))
    y_ranking = zeros(length(y_copy))
    for i in 1:length(x_copy)
        tuple_x = findmax(x_copy)
        x_ranking[i] = Int(tuple_x[2])
        x_copy[tuple_x[2]] = -10
        tuple_y = findmax(y_copy)
        y_ranking[i] = Int(tuple_y[2])
        y_copy[tuple_y[2]] = -10
    end
    # x is authorities
    #y is rankings
    return y_ranking, x_ranking
end

authority, hub = HITS(A, 50)
println("Authority in order\n")
display(arr_2_string(authority, websites, dict))
println("Hub in order\n")
display(arr_2_string(hub, websites, dict))

Authority in order:

4-element Vector{String}:
 "https://www.amazon.com/"
 "https://www.desmos.com/"
 "https://shp31337.github.io/"
 "https://jamesn3.github.io/JamesN3/"
```

```
Hub in order:

4-element Vector{String}:
 "https://shp31337.github.io/"
 "https://jamesn3.github.io/JamesN3/"
 "https://www.desmos.com/"
 "https://www.amazon.com/"
```

```
    arr = []
    for i in 1:200
        push!(arr, eigen_steady_state_iterations(M, 100000000000000,
        .15, i/1000, 2))
    end
    Plots.plot(arr, xlabel = "epsilon in thousandths",
    ylabel = "Randomwalk Iterations")
    #.001 increments of epsilon
```



```
    arr = []
    for i in 1:200
        push!(arr, eigen_steady_state_iterations(M, 100000000000000,
        .15, i/1000, 1))
    end
    Plots.plot(arr, xlabel = "epsilon in thousandths",
    ylabel = "Randomwalk Iterations")
```



7
6
5
4
3
Randomwalk Iterations
0
50
100
150
200
ε in thousandths
y1

References

[1] Chonny *HITS Algorithm: Link Analysis Explanation and Python Implementation from Scratch* 2021.

[2] Naiko Saito *MAT 167: Applied Linear Algebra Lecture 24: Searching by Link Structure I* 2017.

[3] Shion Honda *PageRank Explained: Theory, Algorithm, and Some Experiments* 2020.

[4] Gilbert Strang *Introduction to Linear Algebra. 5th Edition* 2001.

[5] Subhajit Sahu, Kishore Kothapalli, Dip Sankar Banerjee *Adjusting PageRank parameters and Comparing results* 2021.