PageRank and its Applications on the Modern Web

Michael Gee and Jeremy Brennan

This paper begins with an explanation of the underlying mathematical principles of PageRank, including Markov Chains and stable steady state convergence. It then discusses the possibilities of ranking with a naive random walk and its shortcomings that form the basis of the PageRank algorithm.
It then explores the applications and modifications of PageRank that allow it to continue to perform effectively on the vast data sets and requirements of the modern web and its users and compares it with other various page-ranking algorithms such as the HITS algorithm.

Definitions

Item	Definition
$\vec{1}$	Vector with all entries 1

1. MARKOV CHAINS

1.1. About Markov Chains

A **Markov Chain** is a mathematical system that denotes transitions from one state to another, according to specified probabilities. No matter how the process arrived at its present state, the possible future states are fixed.

A *Markov matrix* or *stochastic matrix* is a Matrix A if and only if:

1. Every entry of A is positive: $a_{ij} \geq 0$.

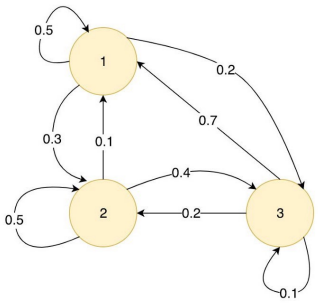
2. Every column of A adds to 1.

1.2. Constructing a Markov Matrix/Markov Chain

Mathematically we make the following definitions,
Let $W(j,i)$ be the be the weight of the link going from node j to i. Note that for a random walk $W(j,i) = \frac{1}{L(j)}$ where $L(j)$ is the number of outgoing links from j. Let N be the total number of nodes. We create an $N \times N$ Markov matrix M by:

$$M_{ij} = \begin{cases} W(j,i) & \text{if there is a link from j to i} \\ 0 & \text{otherwise} \end{cases}$$

For example[2], we create a transition Markov matrix from the directed graph below:



Markov process

	S_1	S_2	S_3
S_1	0.5	0.1	0.7
S_2	0.3	0.5	0.2
S_3	0.2	0.4	0.1

Transition matrix

This transition matrix is also called the Markov matrix. The element ij is the probability of moving from state j to i.

1.3. About Steady States

Because in a Markov matrix M, M_{ij} is the probability of moving from state j to i, we can repetitively multiply M by itself and arrive at the probability of moving from state j to i after some k moves ($k \in \mathbb{N}$).

Specifically, M^k_{ij} is the probability of moving from state j to i after some k moves.

We often call this a *random walk* performed by a *random surfer*. A random surfer is theoretical entity that is assigned a starting node in a directed graph and will randomly choose a path from its node with equal probability and move to the node in which the path points (ie performs a random walk).

As $k \to \infty$, M^k will converge and approach its steady state, indicating in each column of M^k the likely-hood of the random surfer ending on each node (indicated by row) after infinitely many iterations of its random walk.

This is called the *steady state* of M and will always be a scalar multiple of the eigenvector corresponding to eigenvalue $\lambda = 1$ of the Markov matrix (as proven in

Section 1.5).

1.4. Intermediate Proofs

Before proving that the steady state of a Markov matrix occurs for the eigenvalue corresponding to the eigenvalue $\lambda = 1$, we prove the following Lemmas:

$P(k) :=$ "$M^k\vec{x} = \lambda^k\vec{x}$ " $k \in \mathbb{N}^+$
Claim: $P(k)$ is true $\forall k$
Base Case: $k = 1 : M\vec{x} = \lambda\vec{x}$ (Markov matrix)
Induction Step: WTS $P(k) \implies P(k+1)$

$$\begin{aligned} & M^k\vec{x} = \lambda^k\vec{x} && \text{Induction Hypothesis} \\ \implies & M(M^k\vec{x}) = M(\lambda^k\vec{x}) \\ \implies & M^{k+1}\vec{x} = \lambda^k(M\vec{x}) \\ \implies & M^{k+1}\vec{x} = \lambda^k(\lambda\vec{x}) && (M\vec{x} = \lambda\vec{x}) \\ \implies & M^{k+1}\vec{x} = \lambda^{k+1}\vec{x} \end{aligned}$$

Claim: These transition Markov matrices that are constructed from incidence graphs will also have Eigenvalue $\lambda = 1$.

Proof. Consider some $N \times N$ Markov matrix M.
Then,

$$\begin{aligned} & M^T\vec{1} = \vec{1} && \text{(columns of } M \text{ sum to 1)} \\ \implies & M^T\vec{1} = (1)\vec{1} \end{aligned}$$

$\implies M^T$ has eigenvalue $\lambda = 1$
$\implies$ M has eigenvalue $\lambda = 1$ (M^T and M share same eigenvalues as they have the same characteristic polynomial) □

1.5. Proving Existence of Steady State

If a diagonalisable $N \times N$ Markov matrix M has eigenvectors $\vec{x}$ with first eigenvalue $\lambda_1 = 1$ and all other eigenvalues $\mid \lambda \mid < 1 \implies$ the first eigenvector $\vec{x}_1$ is an attracting steady state.

Proof. Consider some $N \times N$ Markov matrix M.
Then, let $\{\vec{x}_1, \vec{x}_2, \ldots, \vec{x}_n\}$ be the basis of eigenvectors for M.

Consider $\vec{v} \in \mathbb{R}^n$. Then, $\vec{v} = c_1\vec{x}_1 + \cdots + c_n\vec{x}_n$.
And by linearity,

$$\begin{aligned} M^k\vec{v} &= c_1\lambda_1^k\vec{x}_1 + \cdots + c_n\lambda_n^k\vec{x}_n \\ &\approx c_1\lambda_1^k\vec{x}_1 \text{ for very large } k \\ &\text{(all other } \lambda^k \text{ approach 0)} \end{aligned}$$

□

Hence Markov matrices that have an eigenvalue $\lambda_1 = 1$ and all other eigenvalues $\mid \lambda \mid < 1$, then it must have a stable steady state vector.

1.6. Confirmation of Steady State with Julia

We confirm that the steady state exists with an example from Dr. Offner[7].

We first define the Matrix $M = \begin{bmatrix} .95 & .1 \\ .05 & .9 \end{bmatrix}$.

```
M = [.95 .1 ; .05 .9 ]
```

We then determine the Eigenvalues and Eigenvectors:

```
using LinearAlgebra

valM,vecM = eigen(M)
```

```
Output:
eigenvalues:
2-element Array{Float64,1}:
1.0
0.85
eigenvectors:
2 * 2 Array{Float64,2}:
0.894427 -0.707107
0.447214 0.707107
```

We now show that M^k for some large k (we pick 50) will always converge to the stable steady state.

Testing $\vec{v} = (3, 0)$

```
M^50 * [3 ; 0]
```

```
Output:
2-element Array{Float64,1}:
2.0002957646637123
0.9997042353362875
```

Testing $\vec{v} = (0, 3)$

```
M^50 * [0 ; 3]
```

```
Output:
2-element Array{Float64,1}:
1.999408470672575
1.000591529327426
```

Testing $\vec{v} = (1.5, 1.5)$

```
M^50 * [ 1.5 ; 1.5]
```

```
Output:
2-element Array{Float64,1}:
1.9998521176681436
1.0001478823318566
```

Note that each of these vectors converge to the same vector: $\vec{x}_{50} = (2, 1)$ which is clearly a scalar multiple of the eigenvalue corresponding to $\lambda = 1$.

As discussed shortly, Markov chains form the basis for the PageRank algorithm.

2. ABOUT PAGERANK

PageRank is an algorithm developed by Google's Larry Page that ranks a given set of pages by using existing links between them. As stated by Google in their original PageRank paper[3], *"PageRank 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"*.

3. NAIVE APPROACH TO PAGERANK

Based on this goal of PageRank, the idea of Markov chains in *Section 1* can be applied.

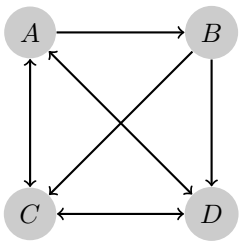
A naive PageRank algorithm can be applied as follows:

1. Construct a Markov process with web pages in the following manner: let the nodes be web pages and let hyperlinks from one web page to another other represent connections between these nodes.
2. Perform a random walk on this transition matrix until in converges to some steady state.

Hence we arrive at the definition of PageRank of node/web page p_i to be:

$$PR(p_i) = \sum_{p_j \to p_i} \frac{PR(p_j)}{L(p_j)}$$

For example,



Given the web pages $P = \{A, B, C, D\}$ with outgoing hyperlinks outlined in the incidence graph, we define the transition Markov matrix A, by the definition in Section 1:

$$A_{ij} = \begin{cases} \frac{1}{L(j)} & \text{if there is a link from j to i} \\ 0 & \text{otherwise} \end{cases}$$

$$\implies A = \begin{bmatrix} 0 & 0 & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{3} & 0 & 0 & 0 \\ \frac{1}{3} & 0 & 0 & \frac{1}{2} \\ \frac{1}{3} & 1 & \frac{1}{2} & 0 \end{bmatrix}$$

Note that this matrix A is a Markov matrix/column-stochastic by the definition in *Section 1*. By this definition, A will always be column-stochastic as long as each web page has outgoing links.

A steady state can now be determined as outlined in *Section 1* to determine the probability that a random walk will end up on this node after k steps as $k \to \infty$ which will be used to rank the pages.

For example,
We now define the probability of a 'random surfer' starting on each web page with equal probability.

$$\vec{x} = \begin{bmatrix} x_a \\ x_b \\ x_c \\ x_d \end{bmatrix} = \begin{bmatrix} .25 \\ .25 \\ .25 \\ .25 \end{bmatrix}$$

Now, we multiply by the Markov matrix to approach the steady state which will elucidate the probability of the random surfer to appear on a given web page assuming it takes a random path from each node each iteration:

$$A^k \vec{x}_0 = \vec{x}_k$$

We observe that this example converges very quickly, after 5 iterations clearly web page D has the highest rank.

k	x_a	x_b	x_c	x_d
0	.25	.25	.25	.25
1	.25	.08333	.20833	.45833
2	.33333	.08333	.31250	.27083
3	.29167	.11111	.24653	.35069
4	.29861	.09722	.27257	.33160
5	.30208	.09954	.26534	.33304

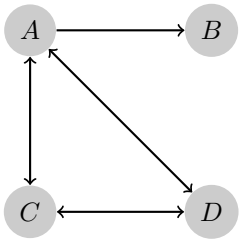
And so the ranks are as follows:

1. Web page D
2. Web page A
3. Web page C
4. Web page B

4. PROBLEMS WITH NAIVE PAGERANK

However, clearly there are several problems with this approach– specifically dangling nodes and independent nodes.

4.1. Dangling Nodes



Above: B is a dangling node.

Suppose that some pages do not have any out-links *(called dangling nodes)*. A random surfer will get stuck on these pages and the importance recieved by these pages will not be propagated.

Mathematically, consider the following example:

Example: Let M be the transition matrix of the incidence graph above:

$$M = \begin{bmatrix} 0 & 0 & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{3} & 0 & 0 & 0 \\ \frac{1}{3} & 0 & 0 & \frac{1}{2} \\ \frac{1}{3} & 0 & \frac{1}{2} & 0 \end{bmatrix}$$

As can be seen, the second column of M denoting the probability of a random surfer leaving B to the i^{th} node (where i denotes row) is 0. This makes it no longer a Markov matrix because the second column does not sum to 1.
Hence as we attempt to converge M to find the stable steady state, it will instead converge to the all-zero matrix, because random surfers that arrive at B will be lost. Empirically,

$$M^5 = \begin{bmatrix} 0.15278 & 0.00000 & 0.21181 & 0.21181 \\ 0.06481 & 0.00000 & 0.07639 & 0.07639 \\ 0.14120 & 0.00000 & 0.16667 & 0.19792 \\ 0.14120 & 0.00000 & 0.19792 & 0.16667 \end{bmatrix}$$

$$M^{10} = \begin{bmatrix} 0.08316 & 0.00000 & 0.10958 & 0.10958 \\ 0.03148 & 0.00000 & 0.04158 & 0.04158 \\ 0.07305 & 0.00000 & 0.09686 & 0.09588 \\ 0.07305 & 0.00000 & 0.09588 & 0.09686 \end{bmatrix}$$

$$M^{50} = \begin{bmatrix} 0.00048 & 0.00000 & 0.00063 & 0.00063 \\ 0.00018 & 0.00000 & 0.00024 & 0.00024 \\ 0.00042 & 0.00000 & 0.00056 & 0.00056 \\ 0.00042 & 0.00000 & 0.00056 & 0.00056 \end{bmatrix}$$

Observe that as $k \to \infty$, $M_{ij}^k \to 0$

4.2. Solution for Dangling Nodes

To solve this problem, we modify how the random surfer behaves. If the random surfer encounters a dangling node, it randomly jumps to one of the N pages. And we modify how we define transition Markov matrix A accordingly.

So mathematically,

$$A_{ij} = \begin{cases} \frac{1}{L(j)} & \text{if there is a link from j to i} \\ \frac{1}{N} & \text{if node j is a dangling node} \\ 0 & \text{otherwise} \end{cases}$$

Hence the definition of PageRank of node/web page p_i after this modification:

$$PR(p_i) = \sum_{p_j \to p_i} \frac{PR(p_j)}{L(p_j)} + \sum_{p_j \text{ has no links}} \frac{PR(p_j)}{N}$$

Example: Now observe how our previous example will now converge:

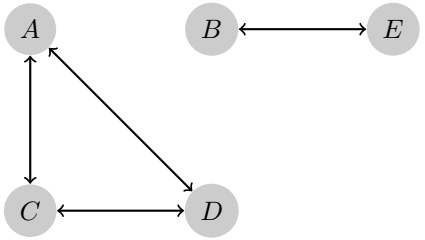
$$M = \begin{bmatrix} 0 & \frac{1}{4} & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{3} & \frac{1}{4} & 0 & 0 \\ \frac{1}{3} & \frac{1}{4} & 0 & \frac{1}{2} \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{2} & 0 \end{bmatrix}$$

And now,

$$M^{50} = \begin{bmatrix} 0.31034 & 0.31034 & 0.31034 & 0.31034 \\ 0.13793 & 0.13793 & 0.13793 & 0.13793 \\ 0.27586 & 0.27586 & 0.27586 & 0.27586 \\ 0.27586 & 0.27586 & 0.27586 & 0.27586 \end{bmatrix}$$

As required.

4.3. Independent Nodes/Networks



Above: B and E form an independent network from A, C, and D.

Suppose that the web's incidence graph of nodes is comprised of two independent networks. A random surfer starting from one of these networks will be unable to get to the other component. All pages in the other network will get zero importance.

Mathematically, consider the following example:

Example: Let M be the transition Markov matrix of the incidence graph above:

$$M = \begin{bmatrix} 0 & 0 & \frac{1}{2} & \frac{1}{2} & 0 \\ 0 & 0 & 0 & 0 & 1 \\ \frac{1}{2} & 0 & 0 & \frac{1}{2} & 0 \\ \frac{1}{2} & 0 & \frac{1}{2} & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

Now, we if we unevenly distribute our random surfers,

$$\vec{x}_0 = \begin{bmatrix} 0.20 \\ 0.10 \\ 0.15 \\ 0.40 \\ 0.15 \end{bmatrix}$$

Calculating $M^{50}\vec{x}_0$,

$$M^{50}\vec{x}_0 = \begin{bmatrix} 0.333 & 0.000 & 0.333 & 0.333 & 0.000 \\ 0.000 & 0.500 & 0.000 & 0.000 & 0.500 \\ 0.333 & 0.000 & 0.333 & 0.333 & 0.000 \\ 0.333 & 0.000 & 0.333 & 0.333 & 0.000 \\ 0.000 & 0.500 & 0.000 & 0.000 & 0.500 \end{bmatrix} \begin{bmatrix} 0.20 \\ 0.10 \\ 0.15 \\ 0.40 \\ 0.15 \end{bmatrix}$$
$$= \begin{bmatrix} 0.20 \\ 0.10 \\ 0.15 \\ 0.40 \\ 0.15 \end{bmatrix}$$

As discussed in *Section 1*, we stated that all initial vectors should converge to the same steady state.
However, this $M^{50}\vec{x}_0$ is not the same as if we had evenly distributed our surfers.

$$\text{Let } \vec{v} = \begin{bmatrix} 0.20 \\ 0.20 \\ 0.20 \\ 0.20 \\ 0.20 \end{bmatrix}$$

$$M^{50}\vec{v} = \begin{bmatrix} 0.333 & 0.000 & 0.333 & 0.333 & 0.000 \\ 0.000 & 0.500 & 0.000 & 0.000 & 0.500 \\ 0.333 & 0.000 & 0.333 & 0.333 & 0.000 \\ 0.333 & 0.000 & 0.333 & 0.333 & 0.000 \\ 0.000 & 0.500 & 0.000 & 0.000 & 0.500 \end{bmatrix} \begin{bmatrix} 0.20 \\ 0.20 \\ 0.20 \\ 0.20 \\ 0.20 \end{bmatrix}$$
$$= \begin{bmatrix} 0.20 \\ 0.20 \\ 0.20 \\ 0.20 \\ 0.20 \end{bmatrix}$$

4.4. Solution for Independent Nodes

To solve this problem, Page and Brin introduced a positive constant d between 0 and 1 (Google used 0.85) called the *damping factor*[3].
To prevent the random surfer from being caught on a node at any time, the damping factor defines the probability that the random surfer will not teleport to another random node. ie. $(1-d)$ defines the probability that the random surfer will not teleport to another random node.

So mathematically,
We define:

$$P_0 = A$$

$$P_k = (1-d)P_{k-1} + dR \ (k \in \mathbb{N})$$

where all entries of R are $\frac{1}{N}$.

Lemma: P_k will be a Markov matrix. (This is required for a stable steady state to exist).

Proof. Base case: We know that A is a Markov matrix (by definition of how we construct it outlined in *Section 1*).
$\implies P_0^T\vec{1} = A^T\vec{1} = \vec{1}$ (by definition of Markov matrix)

Induction Hypothesis: Now assume all P_l are positive for all $0 \leq l \leq k$
Consider P_{k+1}:

$$\begin{aligned} P_{k+1}^T\vec{1} &= ((1-d)P_k + dR)^T\vec{1} \\ &= ((1-d)P_k^T + dR^T)\vec{1} \\ &= (P_k^T - dP_k^T + dR^T)\vec{1} && \text{(Distribution)} \\ &= P_k^T\vec{1} - dP_k^T\vec{1} + dR^T\vec{1} && \text{(Distribution)} \\ &= \vec{1} - d\vec{1} + dR^T\vec{1} && \text{(Induction hypothesis)} \\ &= \vec{1} - d\vec{1} + d\vec{1} \\ &= \vec{1} \end{aligned}$$

So, if P_l is a Markov matrix for $0 \leq l \leq k$, then P_{k+1} must also be a Markov matrix.
We know P_0 is a Markov matrix, hence by the principle of mathematical induction, P_g $(g \in \mathbb{N})$ must also be a Markov matrix. □

It is interesting to note here that if $d = 0$, then we have the naive PageRank in *Section 3* and if $d = 1$, then we have the PageRank of every page equal to $\frac{1}{n}$. Hence, $0 < d < 1$.

This leads to the following PageRank for some node/web page p_i:

$$PR(p_i) = \frac{1-d}{N} + d\Big(\sum_{p_j \to p_i} \frac{PR(p_j)}{L(p_j)} + \sum_{p_j \text{ has no links}} \frac{PR(p_j)}{N}\Big)$$

4.5. PageRank Solution

Hence, we consider the solution to the PageRank of a given node/web page to be:

$$PR(p_i) = \frac{1-d}{N} + d\Big(\sum_{p_j \to p_i} \frac{PR(p_j)}{L(p_j)} + \sum_{p_j \text{ has no links}} \frac{PR(p_j)}{N}\Big)$$

And we arrive at this PageRank by:

$$\text{Letting } P_0 = A$$

$$\text{Letting } P_k = (1-d)P_{k-1} + dR \ (k \in \mathbb{N})$$

where all entries of R are $\frac{1}{N}$.

And finding the stable steady state of:

$$P_k \vec{x}_0 = \vec{x}_k$$

5. TESTING CONVERGENCE

5.1. Defining a Margin of Error

Finally, the question arises– how is convergence tested in this iterative approach to calculating the PageRank of a network of web pages?

A *margin of error* ϵ is defined such that the PageRank algorithm halts iteration when each page's previous PageRank differs from its current PageRank by less than or equal to the margin of error:

Mathematically, for the k^{th} iteration of the PageRank algorithm, where $PR_k(p_i)$ is the k^{th} iteration of the PageRank algorithm on some node p_i, it will cease to iterate if for all nodes p:

$$\mid PR_k(p) - PR_{k-1}(p) \mid < \epsilon$$

5.2. Operating for Speed, Defining Maximum Iterations

Although Page and Brin claimed in their original paper[3] that the PageRank algorithm will converge within a tolerable limit after 52 iterations (ie meet the margin of error condition). In the same paper, they also stated that the convergence in a network of half the above size took approximately 45 iterations. Through this data, they concluded the algorithm's scaling factor for extremely large networks would be roughly linear in $\log n$, where N is the size of the network.

However, modern data sets can often far exceed this (there are approximately 1.7 billion web pages hosted currently[5]) and the PageRank algorithm must produce results quickly. Hence, many implementations of PageRank will also define the maximum iterations that the algorithm may perform before halting if the margin of error condition is not achieved.

6. IMPLEMENTING PAGERANK IN JULIA

We program the PageRank algorithm in the `Julia` language in 3 parts:

1. Pre-processing the websites indexed by natural numbers and the corresponding websites they link to into a Markov matrix.

2. Processing the Markov matrix into rankings using PageRank.

3. Printing the results into a readable format.

The rationale behind preforming the calculation in steps is: readability and abstraction. Although our code will work for all inputs (any number of websites, etc.), when moving algorithms into production it is easier to modify a subroutine than it is to modify an entire algorithm.

Although we step through the implementation step-by-step here, we also attach a copy of the entire function-set at the conclusion of the paper.

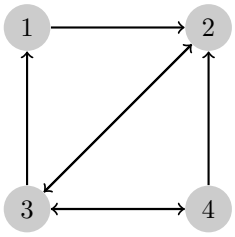
6.1. Pre-processing Data

We accept input in the following structure:
Array $(2 \times n)$ where N is number of websites
- Array[x][1]: String: Website URL
- Array[x][2]: Array: $(1 \times X)$ where X is number of websites linked in website Array[x].

Data structures like this are often better illustrated by an example, like the following:

```
DATA = [
    ["https://instagram.com", ["https://www.pokemon.com"]],
    ["https://www.pokemon.com", ["https://bulbapedia.bulbagarden.net/"]],
    ["https://bulbapedia.bulbagarden.net/", ["https://www.pokemon.com", "https://facebook.com", "https://instagram.com"]],
    ["https://facebook.com", ["https://www.pokemon.com", "https://bulbapedia.bulbagarden.net/"]]
]
```

This represents the following graph:



Above: 1: Instagram, 2: Pokemon, 3: Bulbapedia, 4: Facebook.

We then perform the following operations on this data structure to produce the Markov transition matrix:

1. Get the number of web pages N and initialise an $N \times N$ matrix of zero.

```
    # Get N: size of matrix
    N = size(webpages)[1]

    # Define M: an empty N x N matrix with values 0
    M = zeros((N, N))
```

2. For efficient processing, load each website and its index into a temporary dictionary (reindexing the websites by the website URL rather than natural numbers).

```
    # For faster lookup processing, reindex
     websites from naturals to
    # the websites themselves

    # Define lookup dictionary
    lookup_dict = Dict{String,Int64}()

    # Load lookup dictionary
    for website in 1:N
        lookup_dict[webpages[website][1]] =
     website;
    end
```

3. We now loop through each website and define a variable in preparation to check how many valid connections the web page has:

```
    # Insert each website into the
    for website in 1:N

        # Get number of valid links
        # Needs to loop through instead of
    calling size()
        # because if link not defined in
    webpages we discard it

        number_of_edges = 0
```

4. In this loop, we then loop through each link once to disregard any invalid hyperlinks.

```
    for link in webpages[website][2]
        # Check that valid link and link is
     not the web page itself
        if (haskey(lookup_dict, link) &&
    webpages[website][1] != link)
            number_of_edges =
    number_of_edges + 1
        end
    end
```

5. After obtaining the correct amount of edges for this website, we then write the corresponding probabilities to our matrix M.

```
    # Write the probability of moving from
    node j to i
    # for each M_{ij}
    for link in webpages[website][2]
        if (haskey(lookup_dict, link) &&
    webpages[website][1] != link)
            M[lookup_dict[link], website] =
     1/number_of_edges
        end
    end
```

   Special attention is payed to the usage of the `lookup_dict` to get the correct position for the row:

```
    M[lookup_dict[link], website] = 1/
    number_of_edges
```

6. We then return M,

```
    return M
```

Example: Running this code on the example `DATA` we defined earlier, we get the following:

```
In [5]: M = pre_process_webpages(DATA)

Out[5]: 4×4 Array{Float64,2}:
         0.0  0.0  0.333333  0.0
         1.0  0.0  0.333333  0.5
         0.0  1.0  0.0       0.5
         0.0  0.0  0.333333  0.0
```

This corresponds to the transition matrix of the graph above too, as desired.

6.2. PageRank

Our goal of implementing the PageRank algorithm will be as based on the discussion in *Section 4.5*, particularly the changes to the naive PageRank that we discuss in *Section 4*.

As described in *Section 4*, we will be focused on calculating the PageRank of each node iteratively through the following equation:

$$PR(p_i) = \frac{1-d}{N} + d\Big(\sum_{p_j \to p_i} \frac{PR(p_j)}{L(p_j)} + \sum_{p_j \text{ has no links}} \frac{PR(p_j)}{N}\Big)$$

We will define our PageRank algorithm to take the following parameters:

```
function pagerank(M, max_iterations, d)
```

1. M (Array): Transition Matrix of Directed Graph of Websites
2. max_iterations (int): Maximum iterations for PageRank
3. d (float ($0 < d < 1$)): damping coefficient. The probability that a random surfer will not jump to an arbitrary node

We then take the following steps to construct the PageRank algorithm:

1. Define the margin of error

```
    e = 1.0e-6
```

2. Define number of web pages

```
    N = size(M)[1]
```

3. Evenly distributing random surfer

```
    v = fill(1.0/N, N)
```

4. Create copy of last v to margin of error

```
    v_last = copy(v)
```

5. Replace zero columns from transition matrix to deal with dangling nodes, as mentioned in *Section 4.1*.

```
    # Replace zero columns from transition
    matrix (replace with 1/N)
    for col in 1:N
        if (M[:,col] == zeros(N))
            M[:,col] = fill(1.0/N, N)
        end
    end
```

6. We then loop through until we either reach the max_iterations or meet the margin of error requirement

```
    for loop in 1:max_iterations
```

In this stage, we perform our PageRank calculation defined in *Section 4.5*:

$$\text{Letting } P_0 = A$$

$$\text{Letting } P_k = (1-d)P_{k-1} + dR \ (k \in \mathbb{N})$$

where all entries of R are $\frac{1}{N}$.

Which we derived from the following PageRank expression:

$$PR(p_i) = \frac{1-d}{N} + d\Big(\sum_{p_j \to p_i} \frac{PR(p_j)}{L(p_j)} + \sum_{p_j \text{ has no links}} \frac{PR(p_j)}{N}\Big)$$

```
    v = d * M * v .+ (1 - d) / N
```

Then, we calculate the error and compare it to the margin of error

```
    # Calculate the error between current
    and last state vectors
    difference = abs.(v_last - v)

    # Sum all error values
    error = sum(difference)

    # Compare to margin of error
    if (error < N * e)
        return v
    end
```

And finally, set the last vector to be the current for future margin of error calculations

```
    v_last = v
```

7. We return the PageRank of each page

Example: We run the transition Markov matrix M produced in the last example through our PageRank function with `max_iterations = 100` and `d = 0.85` (as originally used by Google).

```
In [6]: result = pagerank(M, 100, 0.85)

Out[6]: 4-element Array{Float64,1}:
         0.14509120716073326
         0.3300833885110279
         0.37973419716750556
         0.14509120716073326
```

Here, we arrive at a ranked PageRank of each page as desired.

6.3. Printing Results

We must now take these PageRanks and convert them into meaningful data that would be used in our search engine.

To accomplish this, we define the following function:

```
function print_output(webpages, result)
```

That takes in the following parameters:

1. webpages (Array): A 2D array describing webpages and any existsing links to other webpages
2. result (Vector): The PageRank ($0 < ranking < 1$) of each web page

This function works in the following order:

1. Julia's `sortperm` function is first used to order the vector into order of descending PageRank.

```
    ranked_result = sortperm(result, rev=
    true)
```

Using the `sortperm` function on the PageRank vector above would yield the following:

```
In [10]: sortperm(result, rev=true)

Out[10]: 4-element Array{Int64,1}:
          3
          2
          1
          4
```

2. We use this ranking to loop through each rank and print out the corresponding website by performing a lookup (`findall`) on the original `DATA` array data structure.

```
    # Initialise counter to print rankings
    counter = 1

    # Print out rankings
    for webpage_index in 1:size(result)[1]

        # Get index of this rank:
        index = findall(x->x==webpage_index
    , ranked_result)[1]

        # Print the web page
        print(string(counter), ". ",
    webpages[index][1], "\n")
        counter = counter + 1
    end
```

Example: We run this function on the `DATA` and `result` from above and get the ranked web pages:

```
In [9]: print_output(DATA, result)

1. https://bulbapedia.bulbagarden.net/
2. https://www.pokemon.com
3. https://instagram.com
4. https://facebook.com
```

And we have our ranked web pages as desired!

7. PAGERANK ON THE MODERN WEB AND LARGE DATASETS

7.1. Introduction to the Modern Web

Now, that we have illustrated how PageRank works, it is important to highlight how PageRank remains relevant on the modern web.
Brin and Page invented PageRank while doing research for Stanford University in 1996[3]. Back in 1996, there were only 100,000 websites[11] hosted publicly on the World Wide Web. Nowadays, there are 162 million[11]. In order to remain relevant, there are several modifications that have been made to the PageRank algorithm to adapt to modern hardware such as distributed/-parallel computing and the sheer volume of websites available in the 21st Century.

We explore improvements on PageRank in the following[8] domains.

1. Memory Efficiency. Loading millions of web pages into RAM is expensive and slow. We discuss a block-based strategy to computing PageRank faster and on more computers (distributed computing).

2. Computational Efficiency. Being an iterative algorithm, PageRank consumes a lot of computational power (especially considering it is often performed on million-by-million sized matrices). We discuss pre-processing techniques to decrease the load required to calculate such rankings.

3. Computational Usage Efficiency: Because PageRank is such a computationally heavy, iterative algorithm, we discuss how many iterations to allow on our transition Markov matrix before a ranking of passable accuracy is achieved.

7.2. Pre-processing Graph

Realistically, PageRank does not need to rank all millions of pages– no human will realistically browse through every page. Hence, significant pre-processing can be performed on the incidence graph.

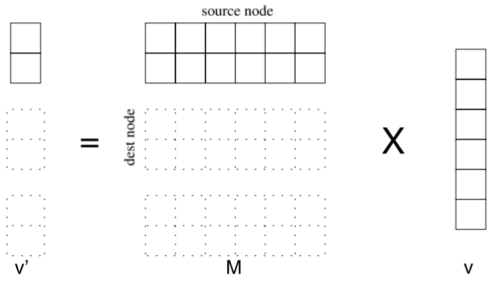
We perform two passes on the graph, each time remove all dangling nodes (all pages/nodes that do not point to any other pages/nodes). Contrary to our approach in *Section 4.2* of sending the random surfer to a random node, removing these dangling nodes is more beneficial when performing calculations on large data sets.

Dr. Haveliwala from Stanford University states [8] an 81-million-node graph will be compressed down to approximately a 19-million-node graph after two passes. Performing (19-million operations additions × 19-million row calculations) each iteration of PageRank is far less than performing (81-million × 81-million) calculations.
This pre-processing will therefore result in:

1. Less RAM requried to hold entire graph.

2. Less computational power required to execute each iteration of PageRank algorithm.

7.3. Block Multiplication



Another approach to decrease RAM usage is to break each matrix and vector up into blocks and perform *block multiplication* on them and then reform them to compare results and rank pages. This concept is illustrated above on the naive PageRank. This enables us to:

1. Perform PageRank on any computer regardless of RAM.

2. Utilise capabilities of distributive computing more easily.

7.4. Accuracy in Floating Points

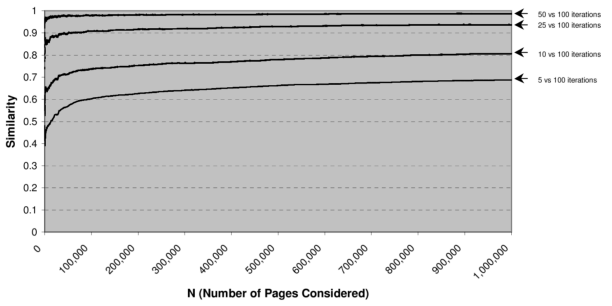
Dr. Haveliwala from Stanford University states [8] that double-precision floating points are not required within matrices and vectors. However, states double-precision floating points should be used for all scalars.

Not utilising double-precision floating point numbers for the matrices and vectors nearly halves the amount of RAM required to load each matrix and vector.

7.5. Convergence Analysis

Dr. Haveliwala has also performed a fascinating study [8] on the similarity in rankings between 5, 10, 25, 50, and 100 iterations of the PageRank algorithm.

The purpose of the PageRank algorithm is to rank a set of given web pages. Ultimately, the end-user wants a ranking and does not care how close it is to the steady state vector.



Above, Haveliwala has compared the similarity of ranks of 5 and 100, 10 and 100, 25 and 100, and 50 and 100. It is important to not he has only sampled the 100 million most significant nodes each time.

As can be seen in the diagram, Haveliwala has empirically shown that a fairly accurate PageRank can be derived after only 10 iterations. This informs how we:

1. Set our margin or error ϵ
2. Set the maximum iterations allowed

And therefore, decreases the amount of computation required as it has been shown that less iterations can be performed.

8. CUSTOMISING SEARCH RESULTS

We continue to discuss the problems that arise from the sheer volume of webpages available on the modern web, but in this section, take a different approach. Instead of dealing with scale or compute, we investigate the quality of ranking delivered to the individual consumer.

For a quick *example*, consider a *consumer A* that lives in Australia and another *consumer B* that lives in America. When each of these consumers type *'local news'* into their search engine, *consumer A* would prefer news from Australia whereas *consumer B* is rather looking for the American news.

We will now discuss how to give each consumer what they desire, given we know what they like. *Note: collecting information on what they like involves applications of machine learning that are beyond the scope of this paper.*

In *Section 4.5* we discussed that we would calculate PageRank by:

$$\text{Letting } P_0 = A$$

$$\text{Letting } P_k = (1-d)P_{k-1} + dR \ (k \in \mathbb{N})$$

where all entries of R are $\frac{1}{N}$.

And finding the stable steady state of:

$$P_k \vec{x}_0 = \vec{x}_k$$

We can now introduce a bias on the random walk such that the *random surfer* is more likely to arrive at a page more preferred by the user.

We accomplish this by altering the matrix R that determines which page the *random surfer* will 'jump' to in accordance with the damping factor (see *Section 4.4* for damping factor). By giving pages the user would prefer a higher probability, we will introduce a bias that will increase the likelihood of a random surfer ending up on the preferred web page. *Note: The exact values that compose each consumer's personal matrix R are often determined by processes such as deep reinforcement learning and it is important to note again that these processes are beyond the scope of the paper.*

This idea is best represented by an *example*, we will continue our example of *Australian Consumer A* and *American Consumer B* use Julia.
We begin by defining the following data structure of news websites:

```
DATA = [
    ["https://www.australian_news.com.au",[]],
    ["https://www.american_news.com", []],
    ["https://www.botswana_news.bt", ["https://
    www.nihon_news.jp"]],
    ["https://www.nihon_news.jp", []]
]
```

By running this data structure through the PageRank algorithm and accompanying code we implemented in *Section 6*, we arrive at the following ranking:

```
In [5]: M = pre_process_webpages(DATA)

Out[5]: 4×4 Array{Float64,2}:
         0.0  0.0  0.0  0.0
         0.0  0.0  0.0  0.0
         0.0  0.0  0.0  0.0
         0.0  0.0  1.0  0.0

In [6]: result = pagerank(M, 100, 0.85)

Out[6]: 4-element Array{Float64,1}:
         0.20618552829813133
         0.20618552829813133
         0.20618552829813133
         0.381443415105606

In [7]: print_output(DATA, result)

        1. https://www.american_news.com
        2. https://www.botswana_news.bt
        3. https://www.nihon_news.jp
        4. https://www.australian_news.com.au
```

However, *Consumer A* prefers to see the Australian news.
Hence we can alter their matrix R to be:

$$R = \begin{bmatrix} 0.997 & 0.997 & 0.997 & 0.997 \\ 0.001 & 0.001 & 0.001 & 0.001 \\ 0.001 & 0.001 & 0.001 & 0.001 \\ 0.001 & 0.001 & 0.001 & 0.001 \end{bmatrix}$$

We define R such that if a *random surfer* decides to 'jump' to another node randomly, it now has a 99.7% chance of jumping to the Australian news of 0.1% chance of jumping to any other news source.

In Julia, we alter PageRank such that:

```
R = [0.997 0.997 0.997 0.997; 0.001 0.001 0.001
     0.001;  0.001 0.001 0.001 0.001; 0.001
    0.001 0.001 0.001]
```

and redo our calculation, taking the first column of R:

```
v = d * M * v .+ (1 - d) * R[: , 1]
```

And we arrive at the following result:

```
In [58]: result = pagerank(M, 100, 0.85)
Out[58]: 4-element Array{Float64,1}:
          0.3247813773698529
          0.17538137736985288
          0.17538137736985288
          0.3244558678904414

In [59]: print_output(DATA, result)
         1. https://www.australian_news.com.au
         2. https://www.nihon_news.jp
         3. https://www.american_news.com
         4. https://www.botswana_news.bt
```

Where clearly, the Australian news is ranked first!

Again, code can be found at the conclusion of the paper.

9. OTHER RANKING METHODOLOGIES: HITS ALGORITHM

We just discussed several methodologies that alter the PageRank algorithm to perform more effectively on the modern web. It is also worth investigating other page-ranking methods. While PageRank is a *link-based*[12] ranking algorithm that ranks based on hyperlinked connections between websites, there also exist other *link-based* ranking algorithms that rather rank based on the *authority* of webpages. The **HITS Algorithm** is an example of this.

A large weakness of PageRank as discussed earlier is its large consumption of processing power and memory. While there are techniques so minimize these, first using a keyword based algorithm to get a smaller number of results and working from there is likely more efficient. HITS solves this by first preforming a keyword rank search, scanning the text in web pages and ranking them based on the number of occurrences of a keyword. It is then up to the implementation how many of these results we keep moving forward, but it will likely only be in the hundreds, compared to assessing millions.

The graph that results from this keyword search is the Root Query Graph (R_Q) where Q is our query/keyword. The problem with this is that these sites are not necessarily well connected just because they contain many occurrences of Q. To solve this problem we expand our graph by including all sites linked to by the sites in R_Q. This is much more likely to form a web of interconnected sites, and we call this new graph S_Q.

We cannot move forward however without knowing our goal. When a keyword is searched, a user is likely wishing to find authoritative sources on the subject. However, the page that repeats Q the most times is not necessarily the most authoritative, but it can help us lead us there. The goal of HITS is to find both **hubs** and **authorities**. Hubs are web pages that, while not authorities themselves, link many authorities to provide an abundance of information. Authorities contain the desired information themselves and are thus pointed to by hubs. This paired definition of hubs and authorities leads us into the implementation of HITS.

A site p is assigned both an authority ranking and a hub ranking. It's authority ranking a_p is equal to the sum of the hub rankings for all sites linking p. This is done because its expected for a good authoritative source to be cited by good hubs. Alternatively, a site's hub ranking h_p is equal to the sum of the authority rankings of the sites it points to, as a good hub should point to authoritative sources.

$$AR(p_i) = \sum_{p_j \to p_i} HR(p_j)$$

$$HR(p_i) = \sum_{p_i \to p_j} AR(p_j)$$

We begin to implement this mathematically through the use of vectors and matrices. $\vec{h}$ represents the hub ranks of all sites while $\vec{a}$ represents the authority ranks of all sites. We make use of the adjacency matrix A where

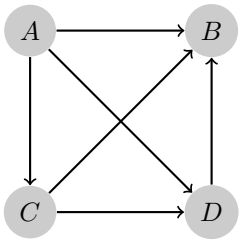
$$A_{ij} = \begin{cases} 1 & \text{if there is a link from i to j} \\ 0 & \text{otherwise} \end{cases}$$

Since hub ranks are determined based on the sums of authority ranks for sites they point to, $\vec{h} = A\vec{a}$, and since authority ranks are determined based on the sums of hub ranks for sites that point to them,

$\vec{a} = A^T\vec{h}$. We can then substitute one formula into the other to get an iterative definition for calculating both hub ranks and authority ranks. For some $k \in \mathbb{N}$, $\vec{h}_k = AA^T\vec{h}_{k-1}$ and $\vec{a}_k = A^TA\vec{a}_{k-1}$.

This iterative computation is very similar to PageRank, but instead relies on the simpler applications of primitive matrices rather than Markov matrices. Primitive matrices are square with all non-negative entries. We know via the Perron Frobenius Theorem[13] that if A is a primitive matrix, it has a largest positive eigenvalue and corresponding positive eigenvector. This leads into the property that as $k \to \infty$, for a vector $\vec{v}_k = A^k\vec{v}$ where A is a primitive matrix, $\frac{\vec{v}_k}{\sum_{i=1}^{N} v_i}$ becomes the normalized eigen vector for the largest eigen value in the matrix A. We must normalize it because repeated applications of A do not maintain the magnitude of vectors like Markov matrices do. Since AA^T and A^TA matrices are both the products of adjacency matrices containing 0s and 1s, they are both non-negative. They are square due to being the product of a matrix and its transpose, making them both primitive matrices.

Here is an example graph representing S_Q.



The above graph corresponds to an adjacency matrix of

$$A = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

And A^TA / AA^T matrices of

$$A^TA = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 3 & 1 & 2 \\ 0 & 1 & 1 & 1 \\ 0 & 2 & 1 & 2 \end{bmatrix}, AA^T = \begin{bmatrix} 3 & 0 & 2 & 1 \\ 0 & 0 & 0 & 0 \\ 2 & 0 & 2 & 1 \\ 1 & 0 & 1 & 1 \end{bmatrix}$$

For both A^TA and AA^T, $\lambda \approx 5.05$. $\vec{x}_{A^TA} = \begin{bmatrix} 0 \\ .44504 \\ .19806 \\ .35689 \end{bmatrix}$ and $\vec{x}_{AA^T} = \begin{bmatrix} .44504 \\ 0 \\ .35689 \\ .19806 \end{bmatrix}$. These are the expected values for $\vec{h}_k$ and $\vec{a}_k$

For our $\vec{h}_0$ and $\vec{a}_0$ vectors, we will just start with the $\vec{1}$ vector.

$$(A^TA)^{20}\vec{1} = \vec{a}_{20} = \begin{bmatrix} 0 \\ 141.4 \times 10^{12} \\ 62.9 \times 10^{12} \\ 113.4 \times 10^{12} \end{bmatrix} \to \begin{bmatrix} 0 \\ .44504 \\ .19806 \\ .35689 \end{bmatrix}$$

$$(AA^T)^{20}\vec{1} = \vec{h}_{20} = \begin{bmatrix} 141.4 \times 10^{12} \\ 0 \\ 113.4 \times 10^{12} \\ 62.9 \times 10^{12} \end{bmatrix} \to \begin{bmatrix} .44504 \\ 0 \\ .35689 \\ .19806 \end{bmatrix}$$

After many iterations, both normalized vectors reflected the normalized eigenvectors as predicted. The numbers show that authority rankings go from highest to lowest B, D, C, A and the hub rankings go A, C, D, B. This makes sense as B is receiving the most connections and A is sending out the most. Since in this case all sites had a total of three outgoing and incoming connections, the authority and hub rankings were opposite orders. However, in a large system of sites where the number of total outgoing and incoming links varies greatly depending on the site, this is not necessarily true.

ACKNOWLEDGEMENTS

REFERENCES

[1] Brilliant.org. (2019). Markov Chains — Brilliant Math Science Wiki. [online] Available at: https://brilliant.org/wiki/markov-chains/ [Accessed 24 Nov. 2019].

[2] Medium. (2019). Machine Learning.—.Hidden Markov Model (HMM). [online] Available at: https://medium.com/@jonathan_hui/machine-learning-hidden-markov-model-hmm-31660d217a61 [Accessed 24 Nov. 2019].

[3] Ilpubs.stanford.edu. (2019). [online] Available at: http://ilpubs.stanford.edu:8090/422/1/1999-66.pdf [Accessed 27 Nov. 2019].

[4] Courses.cs.washington.edu. (2019). Project 3, part 3: Implementing PageRank. [online] Available at: https://courses.cs.washington.edu/courses/cse373/17au/project3/project3-3.html [Accessed 29 Nov. 2019].

[5] Internetlivestats.com. (2019). Total number of Websites - Internet Live Stats. [online] Available at: https://www.internetlivestats.com/total-number-of-websites/ [Accessed 29 Nov. 2019].

[6] Cs.brown.edu. (2019). [online] Available at: http://cs.brown.edu/courses/cs016/static/files/assignments/projects/GraphHelpSession.pdf [Accessed 29 Nov. 2019].

[7] Offner, D. (2019). Lecture 22 Julia Notes. Carnegie Mellon University, pp.Code Snippets 20-25.

[8] Haveliwala, T. (2019). Efficient Computation of PageRank. Stanford University.

[9] Raluca Tanase, R. (2019). PageRank Algorithm - The Mathematics of Google Search. [online] Pi.math.cornell.edu. Available at: http://pi.math.cornell.edu/ mec/Winter2009/

RalucaRemus/Lecture3/lecture3.html [Accessed 2 Dec. 2019].

[10] Online.sfsu.edu. (2019). [online] Available at: http://online.sfsu.edu/meredith/Linear_Algebra/725_F2010/PDF/Dynneson_FinalDraft_Linear-Algebra-Project.pdf [Accessed 2 Dec. 2019].

[11] Pingdom Royal. (2008). How we got from 1 to 162 million websites on the internet - Pingdom Royal. [online] Available at: https://royal.pingdom.com/how-we-got-from-1-to-162-million-websites-on-the-internet/ [Accessed 3 Dec. 2019].

[12] Review on Web Page Ranking. (2017). International Journal of Recent Trends in Engineering and Research, 3(2), pp.113-115.

[13] Raluca Tanase, R. (2019). HITS Algorithm - Hubs and Authorities on the Internet. [online] Pi.math.cornell.edu. Available at: http://pi.math.cornell.edu/ mec/Winter2009/RalucaRemus/Lecture4/lecture4.html [Accessed 7 Dec. 2019].

10. CODE

Please see following pages for code describing in this paper.

In [1]:

```
function pre_process_webpages(webpages)
    """Pre-process Webpages: Define an N x N matrix from webpages data

    Parameters
    ----------
    webpages : Array
        A 2D array describing webpages and any existsing links to other webpages

    Returns
    -------
    Matrix (N x N)
        The corresponding transition matrix to the input webpage array

    """
    # Get N: size of matrix
    N = size(webpages)[1]

    # Define M: an empty N x N matrix with values 0
    M = zeros((N, N))

    # For faster lookup processing, reindex websites from naturals to
    # the websites themselves

    # Define lookup dictionary
    lookup_dict = Dict{String,Int64}()

    # Load lookup dictionary
    for website in 1:N
        lookup_dict[webpages[website][1]] = website;
    end

    # Insert each website into the
    for website in 1:N

        # Get number of valid links
        # Needs to loop through instead of calling size()
        # because if link not defined in webpages we discard it

        number_of_edges = 0

        for link in webpages[website][2]
            # Check that valid link and link is not the web page itself
            if (haskey(lookup_dict, link) && webpages[website][1] != link)
                number_of_edges = number_of_edges + 1
            end
        end

        # Write the probability of moving from node j to i
        # for each M_{ij}
        for link in webpages[website][2]
            if (haskey(lookup_dict, link) && webpages[website][1] != link)
                M[lookup_dict[link], website] = 1/number_of_edges
            end
        end

    end

    return M
end
```

In [2]:

```
function pagerank(M, max_iterations, d)
    """PageRank: Rank the input pages

    Parameters
    ----------
    M : Array
        Transition Matrix of Directed Graph of Websites

    max_iterations: int
        Maximum iterations for PageRank

    d: float (0 < d < 1)
        damping coefficient. The probability that a random surfer will not
        jump to an arbitrary node

    Returns
    -------
    Vector (size N)
        The ranking (0 < ranking < 1) of each webpage

    """
    # Define Margin of Error
    ϵ = 1.0e-6

    # Define number of web pages
    N = size(M)[1]

    # Evenly distributing random surfer
    v = fill(1.0/N, N)

    # Create copy of last v to margin of error
    v_last = copy(v)

    # Remove zero columns from transition matrix (replace with 1/N)
    for col in 1:N
        if (M[:,col] == zeros(N))
            M[:,col] = fill(1.0/N, N)
        end
    end

    for loop in 1:max_iterations

        # Iterate on v
        v = d * M * v .+ (1 - d) / N

        # Calculate the error between current and last state vectors
        difference = abs.(v_last - v)

        # Sum all error values
        error = sum(difference)

        # Compare to margin of error
        if (error < N * ϵ)
            return v
        end

        # Set last state vector to current state
        v_last = v
```

```
    end

    return v
end
```

Out[2]:

```
pagerank (generic function with 1 method)
```

In [3]:

```
function print_output(webpages, result)
    """Print Output: Rank and output the ranks in plan text

    Parameters
    ----------
    webpages : Array
        A 2D array describing webpages and any existsing links to other webpages

    result: Vector (size N)
        The PageRank (0 < ranking < 1) of each webpage

    Returns
    -------
    None

    Side Effects
    ------------
    Prints out each website in plain text in order of ranking

    """
    # Set the ranks of each result
    ranked_result = sortperm(result, rev=true)


    # Initialise counter to print rankings
    counter = 1

    # Print out rankings
    for webpage_index in 1:size(result)[1]

        # Get index of this rank:
        index = findall(x->x==webpage_index, ranked_result)[1]

        # Print the web page
        print(string(counter), ". ", webpages[index][1], "\n")
        counter = counter + 1
    end

end
```

Out[3]:

```
print_output (generic function with 1 method)
```

In [4]:

```
DATA = [
    ["https://instagram.com", ["https://www.pokemon.com"]],
    ["https://www.pokemon.com", ["https://bulbapedia.bulbagarden.net/"]],
    ["https://bulbapedia.bulbagarden.net/", ["https://www.pokemon.com", "http
s://facebook.com", "https://instagram.com"]],
    ["https://facebook.com", ["https://www.pokemon.com", "https://bulbapedia.bul
bagarden.net/"]]
]
```

Out[4]:

```
4-element Array{Array{Any,1},1}:
 ["https://instagram.com", ["https://www.pokemon.com"]]
 ["https://www.pokemon.com", ["https://bulbapedia.bulbagarden.ne
t/"]]
 ["https://bulbapedia.bulbagarden.net/", ["https://www.pokemon.com",
"https://facebook.com", "https://instagram.com"]]
 ["https://facebook.com", ["https://www.pokemon.com", "https://bulba
pedia.bulbagarden.net/"]]
```

In [5]:

```
M = pre_process_webpages(DATA)
```

Out[5]:

```
4×4 Array{Float64,2}:
 0.0  0.0  0.333333  0.0
 1.0  0.0  0.333333  0.5
 0.0  1.0  0.0       0.5
 0.0  0.0  0.333333  0.0
```

In [6]:

```
result = pagerank(M, 100, 0.85)
```

Out[6]:

```
4-element Array{Float64,1}:
 0.14509120716073326
 0.3300833885110279
 0.37973419716750556
 0.14509120716073326
```

In [7]:

```
print_output(DATA, result)
```

```
1. https://bulbapedia.bulbagarden.net/
2. https://www.pokemon.com
3. https://instagram.com
4. https://facebook.com
```

Australia Example

In [1]:
```
function pre_process_webpages(webpages)
    """Pre-process Webpages: Define an N x N matrix from webpages data

    Parameters
    ----------
    webpages : Array
        A 2D array describing webpages and any existsing links to other
 webpages

    Returns
    -------
    Matrix (N x N)
        The corresponding transition matrix to the input webpage array

    """
    # Get N: size of matrix
    N = size(webpages)[1]

    # Define M: an empty N x N matrix with values 0
    M = zeros((N, N))

    # For faster lookup processing, reindex websites from naturals to
    # the websites themselves

    # Define lookup dictionary
    lookup_dict = Dict{String,Int64}()

    # Load lookup dictionary
    for website in 1:N
        lookup_dict[webpages[website][1]] = website;
    end

    # Insert each website into the
    for website in 1:N

        # Get number of valid links
        # Needs to loop through instead of calling size()
        # because if link not defined in webpages we discard it

        number_of_edges = 0

        for link in webpages[website][2]
            # Check that valid link and link is not the web page itself
            if (haskey(lookup_dict, link) && webpages[website][1] != lin
k)
                number_of_edges = number_of_edges + 1
            end
        end

        # Write the probability of moving from node j to i
        # for each M_{ij}
        for link in webpages[website][2]
            if (haskey(lookup_dict, link) && webpages[website][1] != lin
k)
                M[lookup_dict[link], website] = 1/number_of_edges
            end
```

```
        end

    end

    return M
end
```

Out[1]: pre_process_webpages (generic function with 1 method)

In [60]:

```
function pagerank(M, max_iterations, d)
    """PageRank: Rank the input pages

    Parameters
    ----------
    M : Array
        Transition Matrix of Directed Graph of Websites

    max_iterations: int
        Maximum iterations for PageRank

    d: float (0 < d < 1)
        damping coefficient. The probability that a random surfer will n
ot
        jump to an arbitrary node

    Returns
    -------
    Vector (size N)
        The ranking (0 < ranking < 1) of each webpage

    """
    # Define Margin of Error
    ϵ = 1.0e-6

    # Define number of web pages
    N = size(M)[1]

    # Evenly distributing random surfer
    v = fill(1.0/N, N)

    # Create copy of last v to margin of error
    v_last = copy(v)

    # Define R for consumer that wishes to see the Australian news
    R = [0.997 0.97 0.97 0.97; 0.001 0.001 0.001 0.001; 0.001 0.001 0.00
1 0.001; 0.001 0.001 0.001 0.001]

    # Remove zero columns from transition matrix (replace with 1/N)
    for col in 1:N
        if (M[:,col] == zeros(N))
            M[:,col] = fill(1.0/N, N)
        end
    end

    for loop in 1:max_iterations

        # Iterate on v
        v = d * M * v .+ (1 - d) * R[: , 1]

        # Calculate the error between current and last state vectors
        difference = abs.(v_last - v)

        # Sum all error values
        error = sum(difference)
```

```
        # Compare to margin of error
        if (error < N * ϵ)
            return v
        end

        # Set last state vector to current state
        v_last = v

    end

    return v
end
```

Out[60]:
```
pagerank (generic function with 1 method)
```

In [57]:
```
function print_output(webpages, result)
    """Print Output: Rank and output the ranks in plan text

    Parameters
    ----------
    webpages : Array
        A 2D array describing webpages and any existsing links to other
 webpages

    result: Vector (size N)
        The PageRank (0 < ranking < 1) of each webpage

    Returns
    -------
    None

    Side Effects
    ------------
    Prints out each website in plain text in order of ranking

    """
    # Set the ranks of each result
    ranked_result = sortperm(result, rev=true)


    # Initialise counter to print rankings
    counter = 1

    # Print out rankings
    for webpage_index in 1:size(result)[1]

        # Get index of this rank:
        index = findall(x->x==webpage_index, ranked_result)[1]

        # Print the web page
        print(string(counter), ". ", webpages[index][1], "\n")
        counter = counter + 1
    end

end
```

Out[57]:
```
print_output (generic function with 1 method)
```

In [50]:
```
DATA = [
    ["https://www.australian_news.com.au",[]],
    ["https://www.american_news.com", []],
    ["https://www.botswana_news.bt", ["https://www.nihon_news.jp"]],
    ["https://www.nihon_news.jp", []]
]
```

Out[50]:
```
4-element Array{Array{Any,1},1}:
 ["https://www.australian_news.com.au", Any[]]
 ["https://www.american_news.com", Any[]]
 ["https://www.botswana_news.bt", ["https://www.nihon_news.jp"]]
 ["https://www.nihon_news.jp", Any[]]
```

In [51]:
```
M = pre_process_webpages(DATA)
```

Out[51]:
```
4×4 Array{Float64,2}:
 0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0
 0.0  0.0  1.0  0.0
```

In [58]:
```
result = pagerank(M, 100, 0.85)
```

Out[58]:
```
4-element Array{Float64,1}:
 0.3247813773698529
 0.17538137736985288
 0.17538137736985288
 0.3244558678904414
```

In [59]:
```
print_output(DATA, result)
```

```
1. https://www.australian_news.com.au
2. https://www.nihon_news.jp
3. https://www.american_news.com
4. https://www.botswana_news.bt
```

In [ ]:
```

```