PageRank and HITS: An Exploration

Rohini Das, Carson Piehl

December 2022

1 Introduction

The invention of the modern internet has caused disruption to nearly every facet of our lives. Be it in, entertainment, education, communication, healthcare, or commerce, the modern world would be unrecognizable without the internet. At least part of this vast transformation must be attributed to the creation of search engines, the tools that give the users of today's internet the ability to find the needles they search for in a haystack that includes billions of websites. But how do these search engines function? In this paper, we seek to examine several of the algorithms that power search engines and the modern internet from the perspective of their grounding in Linear Algebra. We attempt to construct PageRank from Linear Algebra principles. We then discuss its slightly earlier cousin, HITS, and how each seek to model random walk behavior through the internet using key insights from Markov Theory. We provide a Julia implementation of each algorithm as well. We then seek to extend the discussion through a small case study of a research method for experimental determination of the damping factor.

2 Page Rank Mechanism

2.1 Naive Markov Model

To motivate our discussion of Page Rank, we consider a simple model of the internet. This internet takes the form of a directed graph, with each website represented by a node, and the links between websites being represented as a directed edge. We want to rank these websites, with those that a random internet user is most likely to spend time on ranked highest.

We can think of this graph as a Markov Chain. In this case, the states of the Markov Chain are the pages of our internet. We can model a user traversing our internet as a random walk on this Markov chain, with transition probabilities based on hypertext links. To formalize this slightly, we can call our pages $p_1, p_2, .., p_N$ for our internet with N total pages. Each of our pages under consideration has some transition probability to the other pages which we find using a general linking function[1]:

$$\ell(p_i, p_j) = \left\{ \frac{\textbf{links to } p_i}{\textbf{total links in } p_j} \quad \text{if total links in } p_j \neq 0 \right. \tag{1}$$

We have a slight caveat, and also have to define a linking function in the case that our denominator is 0:

$$\ell(p_i, p_j) = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases} \tag{2}$$

These functions give the transition probability from p_j to p_i that we need to define our Markov Matrix. We show a Julia implementation constructing this Matrix as the first part of our code section in the appendix. Our Markov matrix $\mathcal{M}$ is[1]:

$$\mathcal{M} = \begin{bmatrix} \ell(p_1,p_1) & \ell(p_1,p_2) & \cdots & \ell(p_1,p_N) \\ \ell(p_2,p_1) & \ddots & & \vdots \\ \vdots & & \ell(p_i,p_j) & \\ \ell(p_N,p_1) & \cdots & & \ell(p_N,p_N) \end{bmatrix} \tag{3}$$

We quickly show that all matrices of this form are Markov Matrices.

First a quick lemma. We consider the sums of the columns of $\mathcal{M}$ of form $\ell(p_1,p_j)+\ell(p_2,p_j)+...+\ell(p_N,p_j)$. Since all of our links from p_j must link to some p_i, we know that this sum will be 1, since all of the links in the denominator must eventually be accounted for in the numerator. Thus:

$$\sum_{i=1}^{N} \ell(p_i,p_j) = 1 \tag{4}$$

We consider $\mathcal{M}^T\vec{1}$ where each entry is equal to a sum of the form above. Thus $\mathcal{M}^T\vec{1} = \vec{1}$ and $\mathcal{M}$ is a Markov matrix.

We now review some of the defining characteristics of Markov matrices to help illuminate their relevance to our problem of ranking websites. As we described above, the entry $\mathcal{M}_{ij}$ is the transition probability from j to i. In our matrix, it is the probability that a user clicks on a link from p_j to p_i. The key insight, however, is that our matrix, like all Markov matrices has a "steady state attracting" eigenvector, which we call $\vec{x_1}$ with eigenvalue $\lambda_1 = 1$. All other eigenvectors of a Markov matrix have eigenvalues smaller than 1 which leads to this eigenvector eventually dominating the others. Therefore, the "steady state attracting" eigenvector pulls vectors multiplied by the Markov matrix towards it, causing the eventual result of $\mathcal{M}^k\vec{v}$ for large k to be $\vec{x_1}$.

We will use this fact to find the long term behavior of users on our internet and thus which "states," or in reality, websites, users are most likely to be at. Using this information, we can rank the websites accordingly. To compute this steady state vector we will use a simple iterative calculation. We repeatedly multiply the Markov matrix by some initial distribution until the difference between the last two calculations is falls below some arbitrarily small ϵ, and let the final distribution be the steady state eigenvector. Our Julia implementation for this naive Markov model calculation is in the second part of the code section of our appendix.

2.2 Problems of the Naive Markov Model

While our model is accurate for our simplified internet which only exists as a directed graph, in reality, our model has several glaring issues, stemming partly from issues caused by our directed graph model. For example, when we have "sinks" (websites with no outgoing links), our probability distribution will only consist of them. This is clearly incorrect, since users do not simply pool into websites with no outgoing links. Another issue is that in real life, users do not simply navigate the internet by clicking links from one website to the next. Users can use their search feature, bookmarks bar, or simply type in a link.

These two restrictions hold our model back from being a truly good simulation, and thus from creating an accurate ranking of pages. We will have to address these limitations.

2.3 PageRank

Our first, and most major issue is the creation of sinks. Instead of naively assuming that users will simply never leave a website with no links, we can make a different and more accurate assumption,

that they are in fact very likely to leave the website. While we have no information about our users or their preferences, we will simply say that they have an equal chance to go to any website (or stay on their current website) if the website has no links.

Recall that N is the total number of websites. We can modify our linking function in the edge case **(2)** in order to prevent sink formation:

$$\ell(p_i, p_j) = \frac{1}{N} \textbf{ otherwise} \tag{5}$$

Our second problem is slightly more complicated. We need to model users acting outside of our Markov model. Again, we have no information about what users are searching for, so we can simply assume that they choose to go to a random website, for a probability distribution vector of:

$$\vec{B} = \frac{1}{N}\vec{1} \tag{6}$$

To find the ultimate probability distribution using these two cases, we need some likelihood that each case will occur. We will simply call this d (as it is also known as a damping factor.) With our new damping factor, we can finally write our complete iterative equation for our PageRank probability vector using the Law of Total Probability. Let $PR(t)$ be the probability distribution vector after t steps. Let $\mathcal{M}$ be our Markov matrix defined above. Let d be our probability that the user follows our Markov model (exclusively clicks links) and $1 - d$ be the probability that the user does not follow our model (uses searchbar, bookmarks, etc.) then[1]:

$$PR(t+1) = d\mathcal{M}PR(t) + \frac{1-d}{N}\vec{1} \tag{7}$$

Does our new formulation still have the important characteristic of convergence for large t? Note that for a given d and N, the base vector is constant. Thus it will not affect the convergent behavior of the Markov matrix in the long term.

We can now use this iterative equation similarly to how we used it previously. We will iteratively calculate our probability vectors until their difference is below a certain ϵ, and call that the steady state vector.

Our Julia implementation of the iterative equation **(7)** and our use of it to find the steady state vector are in the 3rd and 4th parts of the code appendix respectively. We run the code on a few examples and give their outputs, using a damping factor of 0.85.

3 HITS Mechanism

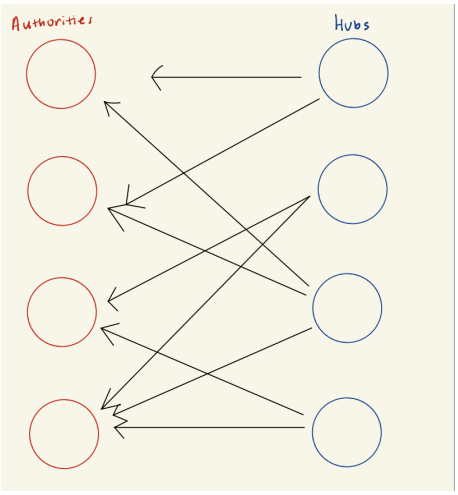
3.1 Problems of the Page Rank algorithm

While the PageRank algorithm successfully ranks web pages in search engine results, its design results in the score of pages being primarily determined by the scores of incoming links. As a result, websites with several outgoing links may still not be given a high score if they have fewer incoming links. While these web pages may not contain much information itself, they essentially served as large directories that led uses to other pages that were more informative. Yet, Page Rank underestimates these websites because of their fewer incoming links.

3.2 HITS Algorithm

To combat this issue, Jon Kleinberg developed HITS or Hyperlink-Induced Topic Search, a link analysis algorithm that rates Web pages based on two measures of scoring: authority and hub value

As mentioned above, certain web pages act as directories. While they may not contain much information, they contain information that direct users to other pages that are more informative. These types of web pages are called hubs. While hubs may not be linked by many other pages, they point to several other pages that are authoritative. On the other hand, other web pages contain more information. These types of pages are called authoritative. While hubs are points to other pages, web pages with good authority are linked by many hubs. Thus, a web page's authority score is based on the value of its content and its incoming links while a web page's hub value is based on its outgoing links. It is important to note that these two metrics are not mutually exclusive. Certain web pages can have a high authority as well as a high hub value. An example would be Wikipedia. The image below shows a diagram of the HITS algorithm



Suppose k represents the number of websites in the current network. Let A represent the adjacency matrix for the current network. Let us represent the authority weight vector by $\vec{v}$ and the hub weight vector by $\vec{u}$. While $\vec{v_t}$ where $t \in \mathbb{N}$ represents the authority score for the k websites at time t, $\vec{u_t}$ where $t \in \mathbb{N}$ represents the hub score for the k websites at time t pre-normalization. $\vec{v_t} = \begin{pmatrix} a_1 \\ a_2 \\ ... \\ a_k \end{pmatrix}$ and $\vec{u_t} = \begin{pmatrix} h_1 \\ h_2 \\ ... \\ h_k \end{pmatrix}$. For some value n, a_n = the sum of h_m where all nodes m point to n.

Essentially, the authority value for node n is equal to the sum of the hub values for all nodes m where m points to n. Similarly, for some value n, h_n = the sum of a_m where all nodes m are pointed to by n. Essentially, the hub value for node n is equal to the sum of the authority values for all nodes m where m is pointed to by n.

At $t = 0$, $\vec{v_0} = 1 = \begin{pmatrix} 1 \\ 1 \\ 1 \\ ... \\ 1 \end{pmatrix}$ Similarly, at $t = 0$, $\vec{u_0} = 1 = \begin{pmatrix} 1 \\ 1 \\ 1 \\ ... \\ 1 \end{pmatrix}$

$$\forall t \in \mathbb{N}, (\vec{a}_{t+1}, \vec{h}_{t+1}) = (A^T * \vec{h}_t, A * \vec{a}_t) \tag{8}$$

This process of calculating the authority and hub values for a certain website in the network is completed k times.

$$\forall k \in \mathbb{N}, \vec{a}_k = \lim_{t \to k}((\vec{a}_t)_k) \tag{9}$$

$$\forall k \in \mathbb{N}, \vec{h}_k = \lim_{t \to k}((\vec{h}_t)_k) \tag{10}$$

To guarantee convergence, we can normalize the hub and authority values after each iteration. Thus, for every iteration that we calculate the hub and authority values, we perform normalization as well.

$$\forall t \in \mathbb{N}, \vec{a}_t^1 = \frac{\vec{a}_t}{||\vec{a}_t||} \tag{11}$$

$$\forall t \in \mathbb{N}, \vec{h}_t^1 = \frac{\vec{h}_t}{||\vec{h}_t||} \tag{12}$$

By normalizing the authority and hub values, the sum of the entries in each vector is equal to 1

$$\forall t \in \mathbb{N}, ||\vec{a}_t^1|| = ||\vec{h}_t^1|| = 1 \tag{13}$$

We will iteratively calculate the authority and hub values until their difference is below a certain epsilon. When this occurs, the hub and authority values converge. As mentioned earlier, higher authority values signal that the page is pointed to by several hub pages while lower authority values signal that the page is not pointed to by several hub pages. Similarly, higher hub values signal that the page points to several good authorities while lower hub values signal that the page does not point to several good authorities.

Since $\vec{h}$ and $\vec{a}$ represent the vectors of the hub and authority scores respectively, $\vec{a} = A^T\vec{h}$ and $\vec{h} = A\vec{a_i}$. This means that $\vec{a} = A^TA\vec{a}$ and $\vec{h} = AA^T\vec{h}$. These equations are similar to a pair of eigenvector equations and we can rewrite $\vec{h} = AA^T\vec{h}$ as the equation for the eigenvectors of AA^T while we can rewrite $\vec{a} = A^TA\vec{a}$ as the equation for the eigenvectors of A^TA. Thus, $\vec{h} = \frac{1}{\lambda_h}AA^T\vec{h}$ and $\vec{a} = \frac{1}{\lambda_a}A^TA\vec{a}$.

If $v_1...v_k$ represents the authority values, then these values converge to a probabilistic eigenvector with the largest eigenvalue of A^TA. Thus, the authority value vector is the probabilistic eigenvector that corresponds to the largest eigenvalue of A^TA

On the other hand, if $u_1...u_k$ represents the hub values, then these values converge to a probabilistic eigenvector with the largest eigenvalue of AA^T. Thus, hub value vector is the probabilistic eigenvector that corresponds to the largest eigenvalue of AA^T. When looking at HITS, there are few theorems that we should keep in mind. Since A^TA and AA^T are symmetric and real, their eigenvectors are real as well. Secondly, suppose A is a non-negative matrix that is symmetric. Suppose x is its largest eigenvector where the sum of its entries equal to 1. If y represents a column vector with non-negative entries, by normalizing y, as we approach some large value k, A^ky converges at x. Essentially, for large values of k, the entries in the column vector y_k are similar to the entries of the vector x. Since we only have one largest eigenvalue, the power method works.

Our Julia implementation of iterative equations **(8)** and our use of it to find the authority and hub values are in the 5th section of the code appendix. We ran our Julia implementation on a few examples and gave their respective outputs

4 HITS vs. PageRank

To conclude our examination of the two algorithms, we test our implementations on some toy "internets" and compare the differences in results. We provide a small Julia REPL displaying our results:

```
# Running our implementations on a

# semi dense graph
```

```
webs1 = [[2,3],[1,3,4],[2,4],[1,3]]

# simple graph
webs2 = [[3], [3], [], [3]]

# maximal density graph
webs3 = [[2, 3, 4], [1, 3, 4], [1, 2, 4], [1, 2, 3]]

link_matrix1 = construct_link_matrix(webs1);
link_matrix2 = construct_link_matrix(webs2);
link_matrix3 = construct_link_matrix(webs3);

page_rank1 = get_final_page_rank(link_matrix1, .85, .00000001);
page_rank2 = get_final_page_rank(link_matrix2, .85, .00000001);
page_rank3 = get_final_page_rank(link_matrix3, .85, .00000001);

hubs1, auth1 = update_hits(webs1);
hubs2, auth2 = update_hits(webs2);
hubs3, auth3 = update_hits(webs3);

println(page_rank1);
println(pr_vector_to_ranks(page_rank1));
println(hubs1);
println(pr_vector_to_ranks(hubs1));
println(auth1);
println(pr_vector_to_ranks(auth1));
# considering the average of the hub and authority vector
println(pr_vector_to_ranks((auth1 + hubs1) / 2))

println("_________________");

println(page_rank2);
println(pr_vector_to_ranks(page_rank2));
println(hubs2);
println(pr_vector_to_ranks(hubs2));
println(auth2);
println(pr_vector_to_ranks(auth2));

println("_________________");

println(page_rank3);
println(pr_vector_to_ranks(page_rank3));
println(hubs3);
println(pr_vector_to_ranks(hubs3));
println(auth3);
println(pr_vector_to_ranks(auth3));

# Output follows:
[0.21005029249458826, 0.253971829775535, 0.2993208569846849, 0.23665702074519168]
[4, 2, 1, 3]
[0.4230888805908032, 0.6845558438576168, 0.3120973001130712, 0.5049498698117849]
[3, 1, 4, 2]
[0.5049442388246835, 0.3121064108961896, 0.684553188874647, 0.4230931759752355]
[2, 4, 1, 3]
```

```
# ranks using average of hub and authority scores
[4, 1, 2, 3]
----------------
[0.15266412926758666, 0.15266412926758666, 0.5420076121972399, 0.15266412926758666]
[2, 2, 1, 2]
[0.5773502691896257, 0.5773502691896257, 0.0, 0.5773502691896257]
[1, 1, 4, 1]
[0.0, 0.0, 1.0, 0.0]
[2, 2, 1, 2]
----------------
[0.25, 0.25, 0.25, 0.25]
[1, 1, 1, 1]
[0.5, 0.5, 0.5, 0.5]
[1, 1, 1, 1]
[0.5, 0.5, 0.5, 0.5]
[1, 1, 1, 1]
```

From this exercise we note several important results. First is that our implementations of HITS and PageRank perform as expected on two extreme graphs, the minimal and maximal density graphs. In the maximal density example, all websites were ranked equally, and in the minimal example, the linked to website was a clear favorite of both algorithms. Examining the semi-dense graph provides us the most insight.

While HITS and PageRank agreed on the least important pages, pages one and four, they had a slight disagreement on the most important page of the small network. PageRank values the links from all pages to page three, and names it the most important. HITS, however, values page two's importance as a hub, and decides that it is the most important page of the set. Overall, both of these interpretations seem valid, and could be easily used to supplement the other, or simply used in different contexts to greater effect.

5 Experimental Damping Factor Determination

5.1 Introduction

Earlier, we discussed the damping factor. One way to improve our PageRank algorithm is to find more accurate values of the damping factor, the proportion of times that pages are changed by clicking through links over the total number of page changes. Determining more accurate values of the damping factor for the population as a whole, or even for individual users may be able to result in better or in the second case, more personalized page ranking algorithms.

For example, if we know that one person has a high likelihood of clicking through links as their primary mode of internet traversal, weighting the Markov model more heavily would allow us to more accurately rank and recommend pages that may be more important to them, and vice versa.

5.2 Methodology

To this end, we have written a small Chrome extension the goal of which is to experimentally determine the damping factor. The code is listed at the end of the code appendix along with some discussion of the algorithmic and implementation choices. Here we will explain the broad strokes of the implementation, along with some discussion of the limitations and advantages of the methodology.

A Chrome extension is a small piece of "piggyback" code running inside of a user instance of the Chrome web browser. From Google, "Extensions are software programs, built on web technolo-

gies (such as HTML, CSS, and JavaScript) that enable users to customize the Chrome browsing experience."[c*ite* They have some storage capability and access to the user's browsing actions. A key advantage of this approach is that it allows us to capture user behavior in a minimally We use this capability to essentially log two values, the number of total page changes p, and the number of links clicked l and simply calculate the damping factor as the ratio:

$$d = \frac{l}{p} \tag{14}$$

This Chrome extension was then used in an anecdotal case study, with one of the paper's authors as the only participant. The author attempted to then simply use their browser as normally as possible, while the extension logged the data. After a day, the data was recovered from the options page of the extension.

5.3 Limitations

5.3.1 Sampling

As described in the methodology subsection, the Chrome extension was only used as a research tool in an anecdotal case study with two participants. This is not a statistically significant study and should be used to draw no wider conclusions. While we discuss some interesting directions for research and possible implications, the speculation in this paper should by no means serve as a substitute for a well-designed statistically representative study.

5.3.2 Architectural Issues

There are several small architectural issues with the inconsistency of the Chrome extension as a platform for confirmed data logging. The architecture of the extension injects a script into a website which leads to occasional naming conflicts with existing functions, and causes l to not be incremented. Similarly, our implementation had occasional issues with accessing storage on websites that already use local storage. In these cases, neither l nor p are incremented.

Finally, and perhaps most grossly, is the issue of script injection timing. In all cases, our extension takes time to establish and run itself on a web page, from one half to a full second on average. However, the page's content, including links, may load and be clickable. Unfortunately, during this time, we can never increment l, which creates another source of error in the methodology.

These three issues may lead to an undercount of the damping factor in the study. In anecdotal experience, these issues affected around 5% of the websites I visited, and so the error is hopefully minimal.

5.3.3 Chrome's onUpdated Event

In our architecture, we must select one of Google's predefined events to trigger the increment of p. Unfortunately, Google does not provide us with fine grained control over the parameters that allow us to trigger the increment. In our implementation, we used Chrome's onUpdated event, which seems to be triggered whenever a page's URL or content is loaded or reloaded, including tab changes. Tab changes occur when a user switches from one tab to another in their browser, while no new content is truly being loaded. This seems to mostly fit our criteria for incrementing p since a tab change does move pages in a way, but it is hard to be sure, since Google's official documentation is simply: "Fired when a tab is updated."[1] While it seems to provide the desired behavior, incidental triggers of this event may cause artificial inflation of p.

5.3.4 Link Clicks

The way that our methodology measures link clicks is taking all clicks on a page, then checking if our click's "target" has an "href" attribute, which causes an HTML element to act as a link. However, this method is non-exhaustive. While "href"[3] is generally the most popular method to create a link, there are also custom JavaScript links and other methods that are difficult to account for. Non-"href" links do not increment l, which may lead to an undercount of the damping factor.

5.4 Results

Over the day that it was used, the extension recorded 31 link clicks and 231 total page changes for a damping factor of $d \approx .134$, which is significantly lower than currently accepted experimental determinations of the damping factor of $d \approx .85$[1]. This is certainly at least in part due to the limitations described above, all of which serve to artificially deflate the damping factor. However, it seems unlikely that the entirety of the decrease is due to the factors above.

To speculate without merit on possible causes, we consider the increased incidence of tab switching in modern browsing. Rather than consistently navigating to new websites, modern browsers allow users to construct a small sub-graph of tabs that they frequently change pages inside of. This implies that future ranking algorithms should focus instead on finding the correct sub-graphs, rather than the correct singular pages.

More importantly, however, this research has provided a proof of concept for the methodology of using a Chrome extension. The advantages of low-interference user observation in this respect are not to be understated, especially since the extension avoids infringing on user privacy by collecting any data about their browsing history, and only uses minimal storage resources.

5.5 Future Research

Future research in this area should focus on minimizing the limitations of the research method and creating more truly rigorous studies using the research tool created. In this way, we can investigate if there has been a true deflation of the damping factor from tab based browsing in order to suggest insights into future ranking algorithms. Our research also suggests that future research should center on finding subgraphs rather than single nodes.

6 Code Appendix

6.1 PageRank Implementation

6.1.1 Naive Markov Construction

```
# Constructs and returns the naive Markov linking matrix using equations (1) and (2)
# Example input: websites = [[4], [4], [4], []]
function construct_link_matrix_naive_markov(websites)
    N = size(websites)[1]; # the number of websites we run pagerank on
    link_matrix = zeros((N, N)); # initialize the linking matrix to zeros
    for website_no in eachindex(websites) # consider the links from each website
        total_links = size(websites[website_no])[1]
        if total_links == 0
            # if a website has no links, we assume that users have no exits
            # and stay at the website
            link_vector = zeros(N);
            link_vector[website_no] = 1;
            link_matrix[:, website_no] = link_vector;
        else
            # if the website does have links, initialize the markov vector
            # with zeros
            link_vector = zeros(N);
            # all the non zero entries are the websites that it links to
            for link in websites[website_no]
                link_vector[link] += 1 / total_links;
            end
            # copy the link vector into our matrix
            link_matrix[:, website_no] = link_vector
        end
    end
    return link_matrix
end
# Example output:
# The linking matrix for this example list, defined as:
# [0 0 0 0;       # M_ij =
#  0 0 0 0;       # 1 / the total number of links (directional edges) from j if j links to i
#  0 0 0 0;       # 0 otherwise
#  1 1 1 0 ]      # Note that this is not a Markov Matrix! All of its columns do not sum to 1.
```

This function takes an input list of enumerated "websites" which are lists containing their "links" to websites as the index of the website. Note the use of the naive linking probability function, which leads to non-Markov matrices as results.

6.1.2 Naive Markov Iteration

```
# A naive PageRank that only uses the Markov Model
function page_rank_naive_markov(markov_matrix, epsilon)
    N = size(markov_matrix)[1]; # finding N
    # an initial distribution vector where we assume the user is equally likely to start on any page
    pr_last = ones(N)/N;
    pr_next = markov_matrix * pr_last;
    # dot the vectors for the sum of squares difference
```

```
    while ( dot((pr_next - pr_last), (pr_next - pr_last)) > epsilon )
        pr_last = pr_next;
        # Simple multiplication to get powers of the Markov Matrix
        pr_next = markov_matrix * pr_last;
    end
    return pr_next;
end
```

This function iteratively computes probability distributions from a Markov matrix using multiplication. It stops when the sum of squares difference between the two most recent distributions falls below an arbitrary epsilon.

6.1.3 PageRank Matrix Construction

```
function construct_link_matrix(websites)
    N = size(websites)[1]; # the number of websites we run pagerank on
    link_matrix = zeros((N, N)); # initialize the linking matrix to zeros
    for website_no in eachindex(websites) # consider the links from each website
        total_links = size(websites[website_no])[1]
        if total_links == 0
            # if a website has no links, we assume that users are equally
            # likely to navigate to any other website (including staying)
            link_matrix[:, website_no] = ones(N)/N;
        else
            # if the website does have links, initialize the markov vector
            # with zeros
            link_vector = zeros(N);
            # all the non zero entries are the websites that it links to
            for link in websites[website_no]
                link_vector[link] += 1 / total_links;
            end
            # copy the link vector into our matrix
            link_matrix[:, website_no] = link_vector
        end
    end
    return link_matrix
end
```

A similar function to the naive matrix constructor above, however, it uses the improved linking function to avoid sinks.

6.1.4 PageRank Step Functions

```
# THE EQUATION:

# Let PR(t) be the page rank vector at time t
# Let M be the markov matrix constructed from the earlier function
# Let d, the damping factor be the probability that an internet user
# simply uses the markov path, instead of acting using other behavior
# Let N be the dimension of the square matrix M, the number of websites
# Let 1> be the ones vector
# Then:
# PR(t+1) = d * M * PR(t) + (1 - d)/N * 1>
```

```
function step_page_rank(link_matrix, prev_pr, damping)
    N = size(link_matrix)[1];

    # This simply calculates the distribution of likelihoods
    # that a user is at a specific website simply following the
    # link_matrix markov model
    linkage_product = damping * (link_matrix * prev_pr);

    # The base vector is a simple model of user behavior assuming
    # that the user doesnt follow the Markov Model.
    # It assumes that the user is equally likely to be on any website.
    base_vector = ((1 - damping)/N) * ones(N);

    return linkage_product + base_vector;
end

# Since the base_vector is constant for a given damping factor and matrix size,
# we dont want to have to recalculate it every time
function step_page_rank_using_constants(link_matrix, prev_pr, damping, base_vector)
    return (damping * (link_matrix * prev_pr)) + base_vector;
end
```

A set of two functions that implement equation **(7)**. The step page rank function computes the next distribution in a one shot manner, following the equation described in the paper. The second takes advantage of the base vector being constant and uses it as a parameter to save on computation.

6.1.5 PageRank Iteration

```
# Calculates the final page rank, after the difference in vectors converges
# below some epsilon.
# link matrix is a matrix constructing using the function above
# the damping factor is the probability that users follow the markov behavioral
# model.
# the epsilon is an arbitrary constant, below which we deem that the PR vectors
# have converged
function get_final_page_rank(link_matrix, damping, epsilon)
    N = size(link_matrix)[1]; # finding N
    pr_last = ones(N)/N; # an initial vector where we assume the user is equally likely to start on any
    base_vector = ((1 - damping)/N) * ones(N); # calculating the base vector
    pr_next = step_page_rank_using_constants(link_matrix, pr_last, damping, base_vector);
    while ( dot((pr_next - pr_last),(pr_next - pr_last)) > epsilon ) # sum of squares difference
        pr_last = pr_next;
        pr_next = step_page_rank_using_constants(link_matrix, pr_last, damping, base_vector);
    end
    return pr_next;
end
```

A function that performs the iterative calculation of the PageRank distribution vector until differences fall below some epsilon. At that point, the algorithm decides it has found the steady state vector and returns the most recent PageRank vector.

6.1.6 Ranking Function

```
function pr_vector_to_ranks(pr_vector)
    # Sort sorts the largest values to the end of the vector,
    # while we want the opposite.
    sorted = reverse(sort(pr_vector));

    # An integer ranking vector
    # (using Int64 in case we actually do run this on the entire internet)
    ranks = zeros(Int64, size(pr_vector));
    for i in eachindex(pr_vector)
        # Find where in the sorting vector each value of the original vector is
        ranks[i] = findfirst(x->x == pr_vector[i], sorted);
    end

    # Ranks [i] is the rank of website i
    return ranks
end
```

A small function that takes the PageRank distribution vector and outputs the rank of website i as a value at array index i. Note that this function is also used to ingest and rank HITS hub and authority vectors.

6.2 HITS Julia Implementation

6.2.1 HITS Transition Matrix Constructor

```
#Constructs the transition matrix. The transition matrices in the hits algorithm
#are the adjacency matrix as well as the transpose
function construct_hits_matrix(webs)
    N = length(webs) # the number of websites we run hits algorithm on
    hits_matrix = zeros((N, N)); # initialize the adjacency matrix to zeros
    #sets specific values in hits_matrix to 1 based on values in webs
    for n in eachindex(webs)
        #creates a vector of size N an initializes each value to 0
        vector = zeros(N)
        #sets value of vector[m] to 1 for each m in webs[n]
        for m in webs[n]
            vector[m] = 1
        end
        #set values in hits_matrix to vector
        hits_matrix[n,:] = vector
    end
    return hits_matrix #returns the constructed matrix
end
```

This function constructs the transition matrix. In the hits algorithm, the transition matrices are the adjacency matrix as well as the transpose.

6.2.2 HITS Iterator

```
#Calculates the new authority and hub values using the previous authority
#values, previous hub values, adjacency matrix, and its transpose
function update_hits(websites)
    #Calls construct_hits_matrix to generate adjacency matrix
    matrix = construct_hits_matrix(websites)


    #Sets initial weight of hub vector nodes to 1
    #This starts each node with having a hub score of 1
    hub_values = ones(size(matrix)[1])
    #Sets initial weight of authority vector nodes to 1
    #This starts each node with having an authority score of 1
    auth_values = ones(size(matrix)[1])

    for i in 1 : 100 #iterates through loop numIterations times

        #Calculates the new authority weight vector
        #Updates each node's authority score to be equal to the sum of the
        #hub scores of each node that points to it.
        auth_next = transpose(matrix) * hub_values

        #Calculates the new hub weight vector
        #Update each node's hub score to be equal to the sum of the authority
        #scores of each node that it points to.
        hub_next = matrix*auth_next
```

```
        #Normalizes authority and hub values so sum of entries is 1
        #Normalizes hub values by dividing each hub score by square root of the
        #sum of the squares of all hub scores
        hub_next = normalize(hub_next)
        #Normalizes authority values by dividing each authority score by square
        #root of the sum of the squares of all authority scores
        auth_next = normalize(auth_next)

        #computes dot product between auth_next - auth_values and auth_next - auth_values
        #checks if dot product is below epsilon
        if dot(auth_next - auth_values, auth_next - auth_values) < 0.000000001

        #computes dot product between hub_next - hub_values and hub_next - hub_values
        #checks if dot product is below epsilon
            if dot(hub_next - hub_values, hub_next - hub_values) < 0.00001
                break #runs process until dot product is smaller than threshold
            end
        end

        #sets new hub_values and auth_values
        hub_values = hub_next
        auth_values = auth_next
    end

    #returns hub_values and auth_values after iterating through loop
    return hub_values, auth_values
end
```

This function calculates the normalized authority and hub values using the previous authority and hub values and the two transition matrices. This process is iterated for a number of iterations. This process is run until the dot product of the difference between the new and old authority values and the dot product of the difference between the new and old hub values is smaller than the given threshold. The code outputs the authority and hub scores

6.3 Experimental Damping Implementation

6.3.1 The Service Worker File

```
// Installation / Service Worker life cycle file


// A small helper function to read local storage originally from:
// https://stackoverflow.com/questions/59440008/how-to-wait-for-asynchronous-chrome-storage-local-get-t
const readLocalStorage = async (key) => {
    return new Promise((resolve, reject) => {
        chrome.storage.local.get([key], function (result) {
            if (result[key] === undefined) {
                reject();
            }
            else {
                resolve(result[key]);
            }
        });
    });
};

// A function to initiate our local storage values
// it runs asynchronously, and retrieves a single
// data object as opposed to two individual integers
async function initVals() {
    // Passing the data from the promise to the then function
    readLocalStorage('damping_data').then((data) => {
        // Need to do null checks in JS, in case the storage has not been initialized
        if (data == undefined) {
            // if this state is reached, we simply set values such that the rest of the
            // function will run normally
            data = {'page_changes': 0, 'link_clicks': 0};
        }
        // helper variables
        page_changes = data['page_changes'];
        // More null checks
        links_clicked = data['link_clicks'];
        if (page_changes == null || page_changes == undefined || page_changes == NaN) {
            page_changes = 0;
            data['page_changes'] = page_changes;
        }
        if (links_clicked == null || links_clicked == undefined || links_clicked == NaN) {
            links_clicked = 0;
            data['links_clicked'] = links_clicked;
        }
        // sets the data properly
        chrome.storage.local.set({'damping_data': data}, () => {});
        // More null checks to ensure execution doesn't stop
        // as we want to be gather data as correctly as possible
        // and avoid unexpected behavior
    }).catch(() => {
        data = {'page_changes': 0, 'link_clicks': 0};
        chrome.storage.local.set({'damping_data': data}, () => {
```

```
            console.log("catching");
        });
    });
}

// We set a listener on installation
chrome.runtime.onInstalled.addListener(() => {
    console.log('inited');
    // On installation, we run our helper function
    // with an extra null check to be safe
    initVals().catch(() => {
        data = {'page_changes': 0, 'link_clicks': 0};
        chrome.storage.local.set({'damping_data': data}, () => {
            console.log("catching");
        });
    });
});


// The majority of the extension.
// We both inject code into our pages to check for link clicks
// and increase the page changes variable whenever the page/tab
// changes for any reason.

// A discussion of the limitations of the "onUpdated" event
// can be found in the Limitations subsection
// https://developer.chrome.com/docs/extensions/reference/tabs/#event-onUpdated
// gives the meager documentation provided by Google on the event
chrome.tabs.onUpdated.addListener((tabId, changeInfo) => {
    if (changeInfo.status === 'complete') {
        readLocalStorage('damping_data').then((data) => {
            // Very similar to the initiation but with fewer null checks
            page_changes = data['page_changes'];
            links_clicked = data['link_clicks'];
            data['page_changes'] = data['page_changes'] + 1;
            chrome.storage.local.set({'damping_data': data}, () => {
                console.log("Set values correctly.");
            });
        });
        // Injecting the link click logging script
        chrome.scripting.executeScript({
            target: {tabId: tabId},
            files: ["/content.js"]
        })
        .then(() => {
            console.log("Script injected successfully.");
        });
    }
})
```

This file holds the code that "piggybacks" on the Chrome browser at a high level called a "service worker". It has essentially two functions, initializing storage values on installation of the extension and adding a listener for page changes. This uses the "onUpdated" Chrome event discussed above. When the listener is fired, it increments the page changes gathered and injects a script into

the website to listen for link clicks.

6.3.2 The Injection File

```
// A small helper function to read local storage originally from:
// https://stackoverflow.com/questions/59440008/how-to-wait-for-asynchronous-chrome-storage-local-get-t
// A small discussion of the issue of naming conflicts can be found in the limitations subsection.
const readLocalStorage = async (key) => {
    return new Promise((resolve, reject) => {
        chrome.storage.local.get([key], function (result) {
            if (result[key] === undefined) {
                reject();
            }
            else {
                resolve(result[key]);
            }
        });
    });
};

// A function that increments the stored number of links clicked by one
async function grabNewLinks() {
    // Reads the current data
    readLocalStorage('damping_data').then((data) => {
        // adds 1
        data['link_clicks'] = data['link_clicks'] + 1;
        // sets new value
        chrome.storage.local.set({'damping_data': data}, () => {
        });
    })
}

async function setUp() {
    // Logs the ability to affect storage on the page
    // A discussion of the issue of storage bugs can be found in the Limitations subsection.
    console.log(chrome.storage);

    // Whenever a click anywhere on the page occurs, we call the following function
    document.addEventListener("click", (event) => {
        // We simply check that the target of the click has a href value
        // which means the user will follow the value to another website
        // A discussion of the issue of this as a criterion for a link click can be found in
        // the Limitations subsection.
        if (event.target.href != '' && event.target.href != null && event.target.href != undefined) {
            // increment the number of links clicked
            grabNewLinks().catch(e => console.log(e));
        }
    })
}

// Calling the set up function on the page we inject it to
setUp().catch(e => console.log(e));
```

This file holds the code that is being injected into web pages in order to listen for link clicks

then increment the value in storage. We discuss the limits of the listening for link clicks and the limitations of the architecture at length above.

6.3.3 Data Retrieval JavaScript

```
// A small helper function to read local storage originally from:
// https://stackoverflow.com/questions/59440008/how-to-wait-for-asynchronous-chrome-storage-local-get-t
const readLocalStorage = async (key) => {
    return new Promise((resolve, reject) => {
        chrome.storage.local.get([key], function (result) {
            if (result[key] === undefined) {
                reject();
            }
            else {
                resolve(result[key]);
            }
        });
    });
};

// A small function to initiate the values for our options page, which displays the data gathered.
async function init() {
    readLocalStorage('damping_data').then((data) => {
        page_changes = data['page_changes'];
        links_clicked = data['link_clicks'];
        // Simply setting the HTML of the page
        $('#total_link_clicks')[0].innerHTML = "Total Link Clicks: " + links_clicked;
        $('#total_page_changes')[0].innerHTML = "Total Page Changes: " + page_changes;
        $('#damping_coeff')[0].innerHTML = "Damping Factor: " + links_clicked/page_changes;
        console.log("Successful Initiation!");
    }).catch(e => console.log(e));
}

init().catch(e => console.log(e));
```

A small JavaScript file to read and display the logged values for the damping factor in the options page of the extension.

6.3.4 Data Retrieval HTML

```
<!-- A simple html page to retrieve the data logged by the extension -->
<!DOCTYPE html>
<html>
  <head>
    <script
    src="jquery-3.6.0.min.js"
    integrity="sha256-/xUj+3OJU5yExlq6GSYGSHk7tPXikynS7ogEvDej/m4="
    crossorigin="anonymous"></script>
      <meta charset="utf-8" />
  </head>
  <body>
      <div id ="nodeappend">
        <h1 id="total_link_clicks"></h1>
```

```
        <h1 id="total_page_changes"></h1>
        <h1 id="damping_coeff"></h1>
      </div>
  </body>
  <script src="options.js" type="text/javascript"></script>
</html>
```

The raw HTML of the data retrieval page that displays the gathered values for the damping factor.

7 Bibliography

1. Brin, S., Page, L. (1998). The anatomy of a large-scale hypertextual web search engine. Computer Networks and ISDN Systems, 30(1-7), 107–117. https://doi.org/10.1016/s0169-7552(98)00110-x

2. Google Chrome. (n.d.). Chrome.tabs. Chrome Developers. Retrieved December 9, 2022, from https://developer.chrome.com/docs/extensions/reference/tabs/event-onUpdated

3. Hubs and Authorities. Hubs and authorities. (n.d.). Retrieved December 9, 2022, from https://nlp.stanford.edu/IR-book/html/htmledition/hubs-and-authorities-1.html

4. Raluca Tanase, R. R. (n.d.). Lecture 4. HITS Algorithm - Hubs and Authorities on the Internet. Retrieved December 9, 2022, from http://pi.math.cornell.edu/ mec/Winter2009/RalucaRemus/Lecture4/lecture4.html