1 Introduction

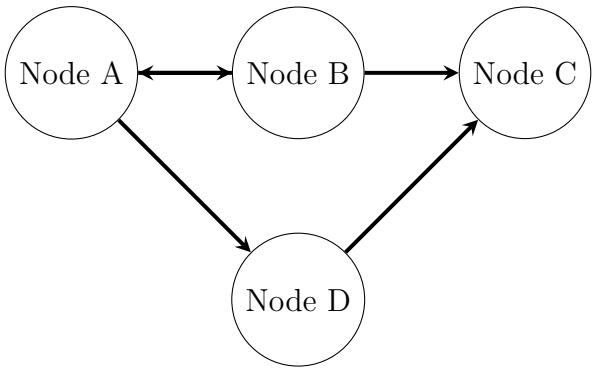
1.1 What is PageRank

Created in 1997 by Larry Page and Sergey Brin, PageRank is an algoirthm used by Google to organize webpage results by the order of their importance. The formal definition of how PageRank works is "by counting the number and quality of links to a page to determine a rough estimate of how important the website is" [2]. Describing it in simpler terms, imagine there are 100 people, let's call them "surfers," distributed evenly among 4 websites (place 25 people in each website). Further, each website has different probabilities to which website each "surfer" can surf to next. If you allow each surfer to surf this web for an infinite amount of times, you will eventually reach a steady state, or a state where a set number of surfers will stay constant in each website [2]. The PageRank algorithm calculates this number of surfers to rank which website is most important, by calculating which websites have the highest number of surfers. And with this idea lies the basis of the PageRank algorithm.

1.2 How PageRank works

1.2.1 Adjacency Matrix

Applying PageRank to websites, you can assume each webpage is a node and each node connects to other nodes. A node is connected to another node if the website has a hyperlink, or references, the other website. From this graph of nodes, you can create an nxn adjacency matrix A. For matrix A, each A_{ij} is equal to one if there is a connection from node i to node j, and every A_{ij} is equal to zero if there is no connection from node i to node j. Take the below 4 website node graph, for example:



Following the rules for each value in A_{ij}, we can generate a matrix A:

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

And we find the adjacency matrix to be A.

1.2.2 Markov Matrix

Once the adjacency matrix has been found, convert matrix into a Markov matrix that calculates the probability of moving from node to another node. To find this Markov matrix, you take the adjacency matrix and divide it column by column. In each column, you find the sum of values in each column and divide the column by the sum. The idea formalized, assumes a Markov matrix M: $M_{ij} = \frac{A_{ji}}{\sum_{i=1}^{n} A_{ji}}$. Given this Markov matrix, this will tell you the "initial distribution and the transitions" for each node [1].
Following our example from 1.2.1, we can apply the above formula to get the Markov matrix:

$$M = \begin{pmatrix} 0/1 & 1/1 & 0/1 & 0/1 \\ 1/1 & 0/1 & 0/1 & 0/1 \\ 0/2 & 1/2 & 0/2 & 1/2 \\ 0/1 & 0/1 & 0/1 & 1/1 \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1/2 & 0 & 1/2 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

1.2.3 Finding Long Term Distribution

To find long term distribution of number of "surfers" on each webpage, we can summarize the most efficient approach as solving the equation $\vec{x} = P\vec{x}$.
Assume an initial distribution of "surfers" on each webpage calculated by the inverse of the number of webpages. This will be our initial distribution vector of $\vec{x}_x$. Next, we keep applying the formula $\vec{x}_{t+1} = M\vec{x}_t$, until $\vec{x}_{t+1} = \vec{x}_t$. This vector will give us the long term distribution of surfers among the webpages. Thus, from this, pages with a higher probability distribution can be labelled as "stronger" and those with lower probability distribution can be labelled as "weaker".

1.2.4 Dampening Effect

Now, the normal page rank algorithm has a flaw. What would happen if one node had no exit points. In other words, what would happen if one website did not have any links to other websites? In a small case, like the one in example 1.2.1, if a website had no exiting links, then that website would get all the surfers. Thus, we need to include a "dampening effect" that assumes that each surfer has an equal probability to go to any other website once

they reach that "no-exit" webpage. To account for this fact, we need to add a "teleportation" factor to our Markov matrix:

$$\hat{M} = (1 - \alpha)M + \frac{\alpha}{N}(N \times N \text{ matrix of 1's})$$

In this case, you add a dampening factor α 0f 0.85. Here we apply a new Markov matrix, with each surfer have a probability of α/N to transition to a random state [4]. This will help account for webpages with dead-ends, preventing the original thought of one webpage getting all the surfers.

1.2.5 HITS

Similar to PageRank, HITS is used to organize webpages by how "strong" they are. The main difference is that HITS gives you the authority scores of webpages, which describes which websites are the "strongest", but it also gives you a hub scores for the websites which deliver the most "surfers." This algorithm is defined in a similar fashion as PageRank, but is split into two parts: one consisting of the hub scores and the other as the authority scores. First, we assume a hub scores score of 1 for each webpage, represented by a $\vec{h}$ of size n with each $n_{j1} = 1$. Next we find the authority score matrix by finding Adjacency matrix transpose times the hub scores: $\vec{a} = A^T\vec{h}$. Next, update your hub scores by finding the transpose of the adjacency matrix times the authority vector: $\vec{h'} = A\vec{a}$. Repeat this process, until your hub score before running your nth process is equal to the hub score after the nth process, and your authority score before running the nth process is equal to the authority score after running the nth process. Once this process is achieved, normalize the hub score and authority score vector. And from the authority score, you can find the "strongest" webpages based on which $\vec{a}_{i1}$ has the greatest value.

2 Code

2.1 PageRank without Dampening

```
function findStateProbabilities(markovMatrix::Matrix{Float64}; tol::Float64=1e-6)
    n = size(markovMatrix, 2)
    surfDistribution = fill(1/n, n)
    newSurfers = zeros(n)
    change = Inf # Theoretically infinite at the start

    while surfDistribution != newSurfers
        # newSurfers = markovMatrix .* surfDistribution
        mul!(newSurfers, markovMatrix, surfDistribution)
        change = norm(newSurfers - surfDistribution)  # Convergence check
        surfDistribution .= newSurfers
    end
```

```
    return surfDistribution
end
```

Meaning of Variables:

`markovMatrix`: Markov Matrix of transition probabilities between nodes (webpages)

`n`: size of the markov matrix

`surfDistribution`: Creates a vector of size n, representing an equal proportion of "surfers" distributed among each webpage.

`newSurfers`: Represents vector that will be updated to represent proportion of "surfers" among each webpage after each iteration.

`change`: Difference between newSurfers and surfDistribution

Next, you enter while loop that only breaks when surfDistribution is equal to newSurfers, so only breaks when proportion is constant.

`mul!(newSurfers, markovMatrix, surfDistribution)`: Calculates markovMatrix * surfDistribution and sets that value equal to newSurfers.

`change`: Convergence check

`surfDistribution .= newSurfers`: sets surfDistribution to newSurfers to update for new while loop

2.2 PageRank with Dampening

```
    # This function does not modify the input markov matrix,
    # so we don't need to make a copy within the function
function dampedPageRank(markov_matrix::Matrix{Float64}, dampingFactor,
tol::Float64 = 1e-6)
    n = size(markov_matrix, 1)
    r = fill(1.0 / n, n)

    teleport = (1.0 - dampingFactor) / n
    iteration = 0
    change = tol

    while change >= tol
        r_new = dampingFactor * (markov_matrix * r) .+ teleport
        # Calculate Frobenius norm
        change = norm(r_new - r, 1)
```

```
        r = r_new
    end

    return r
end
```

`markov_matrix`: Markov Matrix of transition probabilities between nodes (webpages).

`dampingFactor`: Set probability of user exiting "dead end" website.

`tol`: Used to calculate how much difference we need to specify convergence.

`n`: size of the Markov matrix.

`r`: Creates a vector of size n, representing an equal proportion of "surfers" distributed among each webpage.

`teleport`: Accounts for teleportation factor in "dead end" matrix.

`change`: Original change.

Enter while loop, that only breaks once difference is less than set tolerance.
`r_new`: Creates new distribution vector, accounting for teleportation.

`change`: Calculates new differences between `r_new` and `r`.

`r`: Sets r to equal new distribution vector.

2.3 HITS

```
# In `pagerank.ipynb`
# Make a copy of the `hitsMatrix` because we modify it
function hits_algorithm(hitsMatrix::Matrix{Float64})
    adjacencyMatrixTran = copy(hitsMatrix)
    adjacencyMatrix = transpose(copy(hitsMatrix))
    numStates = size(adjacencyMatrixTran, 1)
    hubScores = ones(numStates)
    authorityScores = ones(numStates)
    prevHubScores = zeros(numStates)
    prevAuthorityScores = zeros(numStates)
    k = 0
    while hubScores != prevHubScores||
          authorityScores != prevAuthorityScores
        k += 1
        authorityScores .= adjacencyMatrixTran * hubScores
```

```
        hubScores .= adjacencyMatrix * authorityScores
        prevAuthorityScores .= authorityScores
        prevHubScores .= hubScores
    end
    authorityScores ./= norm(authorityScores, 2)
    hubScores ./= norm(hubScores, 2)
    println(k)
    return hubScores, authorityScores
end
```

`hitsMatrix`: calls in an `adjacencyMatrix`.

`adjacencyMatrixTran`: Sets the transpose of adjacency matrix.

`adjacencyMatrix`: Sets adjacency Matrix.

`numStates`: Sets the number of webpages being tested.

`hubScores`: Creates a vector that assumes an original hub score of 1 for each webpage, vector of size number of webpages.

`authorityScores`: Creates a vector that assumes an original authority score of 1 for each webpage, vector of size number of webpages.

`prevHubScores`: Creates a vector of 0s of size number of webpages, since their is no previous hub score vector.

`prevAuthorityScores`: Creates a vector of 0s of size number of webpages, since their is no previous authority score vector.

`k`: Counts iterations

Enter while loop, only breaks if `hubScores` is not equal to `prevHubScores` and `authorityScores` is not equal to `prevAuthorityScore`.
`authorityScores`: Calculates new authority score, based on HITS algorithm.

`hubScores`: Calculatoes new hubScores based on HITS algoirthm.

`prevAuthorityScores, prevHubScores`: Reassigns values of previous authority and hub scores to compare when entering new while loop.

`norm`: Once out of while loop, normalize authority scores and hub scores to calculate final authority vector and hub vector.

3 Results

In order to thoroughly test our algorithms, we wanted to create different types of graphs. Instead of hard-coding different Markov matricies into our project, which would be impractical to do, we create adjacency lists in a CSV (Comma-separated values) text file. We have a Julia script `generator.ipynb` that generates dense graphs of a given size. We generated graphs up to 10,000 nodes, which would hopefully capture the essence of a large web database.

```
3
1, 3
1, 2, 3
```

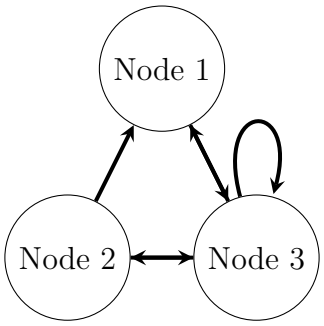


Figure 1: Above, the adjacency list (CSV file) where the first row (node 1) is connected to 3, the second row (node 2) is connected to both the 1st and 3rd nodes, and the third row (node 3) is connected to 1st, 2nd, and 3rd nodes.

However, in reality, the graphs created by the internet are not dense. Small and irrelevant websites oftentimes have no references to it, creating a sparse graph. We also wanted to test our algorithms on real representation of what the internet looks like at a smaller scale. So we made a Julia script to web-scrape all the hyperlinks from a list of links from a given search query, `links.txt`, and create an adjacency list from there.

Since the `generator.ipynb` and `webscraping.ipynb` code is not directly relevant to the math behind PageRank and HITS, this paper won't be going in depth into what these scripts do. However, for completeness and to facilitate reproducibility, we have included these scripts and documentation in the Appendix.

In short, we do testing in two different types of datasets: artificially generated graphs that are dense (and large in size), vs a real life representation of internet graphs that are sparser and more reflective of the actual connectivity found in typical web databases. This dual approach allows us to comprehensively evaluate our algorithms in both idealized and realistic scenarios, ensuring robustness in a variety of network topologies.

3.1 Randomly Generated Dense Graphs

The maximum number of edges (connections) for a given graph of size v is $v(v-1)$. Since the following graphs are dense, the number of connections is close to that $v(v-1)$ value.

Table 1: Non-Damped PageRank Algorithm Run time (in milliseconds)

Graph Size	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Average ± SE
3 nodes	0.007	0.006	0.008	0.007	0.006	0.0068 ± 0.001
5 nodes	0.010	0.007	0.006	0.006	0.008	0.0074 ± 0.001
50 nodes	0.035	0.011	0.017	0.011	0.012	0.0172 ± 0.005
500 nodes	0.124	0.152	0.170	0.159	0.155	0.152 ± 0.008
1000 nodes	0.410	0.475	0.435	0.644	0.460	0.4848 ± 0.041
5000 nodes	8.537	8.157	10.095	7.569	9.619	8.795 ± 0.466
10000 nodes	33.453	34.318	38.827	35.819	38.399	36.163 ± 1.072

Table 2: Damped PageRank Algorithm Runtime (in milliseconds)

Graph Size	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Average ± SE
3 nodes	0.008	0.007	0.008	0.011	0.007	0.008 ± 0.001
5 nodes	0.010	0.009	0.009	0.012	0.009	0.010 ± 0.001
50 nodes	0.041	0.040	0.040	0.040	0.043	0.041 ± 0.001
500 nodes	0.361	0.326	0.343	0.351	0.399	0.356 ± 0.012
1000 nodes	1.338	1.191	1.948	1.558	1.389	1.485 ± 0.130
5000 nodes	36.098	37.553	35.380	35.380	37.994	36.481 ± 0.548
10000 nodes	152.422	149.020	158.221	166.270	139.245	153.036 ± 4.521

Table 3: HITS Algorithm Runtime (in milliseconds)

Graph Size	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Average ± SE
3 nodes	0.054	0.052	0.070	0.064	0.062	0.060 ± 0.001
5 nodes	0.056	0.058	0.056	0.057	0.057	0.057 ± 0.001
50 nodes	0.077	0.069	0.065	0.066	0.071	0.070 ± 0.002
500 nodes	1.958	1.142	0.870	2.270	1.725	1.593 ± 0.258
1000 nodes	4.844	7.397	6.498	7.101	6.102	6.388 ± 0.447
5000 nodes	575.708	215.650	222.731	215.593	214.166	288.770 ± 71.750
10000 nodes	897.972	1132.708	888.254	1078.009	918.016	982.992 ± 50.926

3.2 Real Life Sparse Graphs

With the real life datasets we created, we can compare them to the "correct" rankings – the ones done by Google. Initially we tried comparing the vectors our algorithm outputs with the Google ranks with MSE (Main Squared Error). However, MSE does not capture a sense of "order" effectively, because emphasis is place on difference in magnitude between the two vectors rather than order. Instead, a better metric for order is to use the Spearman's Rank Correlation coefficient which is conveniently part of Julia's `Statistics` package. The coefficient is 1 if the vector is perfectly ordered, 0 if it is perfectly disordered (no correlation), and -1 if it is in perfect reverse order.

We analyze five simple prompts that range from "how to" questions to famous historical events to famous people – representative of what people would typically Google. Note that the number of nodes per prompt is not consistent, this is due to a limitation with Julia's webscraping packages where some pages would limit our script's (`webscraping.ipynb`) searching for hyperlinks.

Table 4: Undamped PageRank Spearman's coefficient compared to Google rankings

Graph Size	Search Prompt	Spearman's ρ
200 nodes	"How to fix a flat tire"	0.85594
195 nodes	"Cristiano Ronaldo"	0.86041
200 nodes	"New York travel guide"	0.77507
125 nodes	"The Beatles"	0.70290
150 nodes	"World War 2"	0.73193

Table 5: Damped PageRank Spearman's coefficient compared to Google rankings

Graph Size	Search Prompt	Spearman's ρ
200 nodes	"How to fix a flat tire"	0.85614
195 nodes	"Cristiano Ronaldo"	0.86055
200 nodes	"New York travel guide"	0.77524
125 nodes	"The Beatles"	0.70239
150 nodes	"World War 2"	0.73108

Table 6: HITS Spearman's coefficient compared to Google rankings

Graph Size	Search Prompt	Spearman's ρ
200 nodes	"How to fix a flat tire"	0.85773
195 nodes	"Cristiano Ronaldo"	0.86820
200 nodes	"New York travel guide"	0.77946
125 nodes	"The Beatles"	0.73455
150 nodes	"World War 2"	0.74665

3.3 Observations

With the randomly generated dense graphs, the non-damped PageRank had the fastest times, followed closely be the damped PageRank. Finally, our implementation of the HITS algorithm ran 6-7 times slower than the damped PageRank. It is expected that our HITS algorithm be the slowest, because as previously described, it calculates two different vectors simultaneously (hub and authority scores).

The time it takes for the random walk to converge and find the steady state vector is also a factor in the runtime. In our implementations, we did not keep track of the number of iterations, nor did we put a cap. If we had placed a limit on the maximum iterations it would take to converge, say 100, then comparing times across algorithms would make more sense, as we could isolate a running of the loop individually. This is a limitation of timing the runtime in Julia in seconds, as opposed to finding the time complexity. Furthermore, these times can change depending on the machine it was run on.

Nonetheless, our tables convey the picture that the HITS algorithm is computationally expensive. A one second runtime for the 10,000 node graph may not seem like a lot, however if we imagine a person searching something up and waiting one second for results to appear, the difference becomes very noticeable when compared to PageRank.

As for the real life dataset, we would like to point out the top five results remained almost constant for the un-damped, damped, and HITS algorithm. However, the HITS algorithm consistently had higher Spearman's coefficients than the damped and undamped PageRanks. Here were the results (ellipsis denote URL was too long to fit in page):

"How to fix a flat tire"

`https://www.tiktok.com/.../how-to-repair-a-flat-tire-in-call-of-duty-zombies`
`https://www.amazon.com/Fix-Flat-S60430-Inflator-Eco-Friendly/dp/B01FX5TKGQ`
`https://www.yelp.com/...find_desc=Flat+Tire+Repair&find_loc=Los+Angeles%2C+CA`
`https://www.fixaflat.com/pages/how-to-install-fix-a-flat`
`https://www.reddit.com/.../tire_center_will_not_repair_a_tire_that_contains/`

"Cristiano Ronaldo"

`https://www.facebook.com/Cristiano/`
`https://twitter.com/cristiano?lang=en`
`https://www.instagram.com/cristiano/?hl=en`
`https://www.bbc.com/news/technology-67566602`
`https://www.marca.com/en/football/2023/12/01/656a5747e2704ecd678b45e8.html`

"New York travel guide"

`https://www.facebook.com/groups/nyctraveltipsandhacks/`
`https://www.instagram.com/newyorkcity.explore/?hl=en`
`https://www.youtube.com/watch?v=I9gotv1kmzM`
`https://www.getyourguide.com/new-york-city-l59/`
`https://www.iloveny.com/`

"The Beatles"

https://www.facebook.com/thebeatles/
https://twitter.com/thebeatles?lang=en
https://www.youtube.com/channel/UCc4K7bAqpdBP8jh1j9XZAww
https://www.instagram.com/thebeatles/?hl=en
https://www.tiktok.com/@thebeatles?lang=en

"World War 2"

https://www.highpointnc.gov/2111/World-War-II
https://ge.usembassy.gov/remembering-ve-day-honoring-the-end-of-world-war-ii/
https://www.pbs.org/wgbh/masterpiece/.../world-war-ii-major-events-timeline/
https://www.nationalww2museum.org/.../america-goes-war-take-closer-look
https://en.wikipedia.org/wiki/World_War_II

With the exception of the World War 2 example, all of our search results have at least two forms of social media in them. This is expected, as websites nowadays link their socials – so websites like TikTok, Twitter, Youtube, Instagram, and Facebook get a lot of references that boosts their popularity to the top. However, can also be a limitation: a person searching for "Cristiano Ronaldo" might want to know more about his life, say in a Wikipedia article, rather than look for him on social media. Furthermore, as we can see in the flat tire example, the first search result is not directly relevant as it is about changing flat tires in a video game. Just because the video is on TikTok, a popular platform for short tutorial demos, it shouldn't mean this particular upload should be recommended.

This is an example of a "Rank leakage," as there are too many outbound hyperlinks to social media platforms. When in reality, their connections should be devalued because they are so commonplace. This is a limitation of our three implementations, however, a solution already exists. Google increased their PageRank performance by increasing the weight on links that seemed more important, and devalued bad connections (like to social media websites in our case) by lowering their weighted average [3].

Moreover, the results indicate a limitation of using the Spearman's rank coefficient as a metric to examine our results. The above search results are reasonable for everything except for "How to fix a flat tire". Our search results for that prompt seem very random: it sends us to an amazon link, a TikTok, and a review for a mechanic in Los Angeles before the first useful URL (fixaflat.com). Yet looking at our table, it has the highest ρ. We believe this is due to the nature of Spearman's coefficient, which does not adequately account for the relative importance of the positions in the vectors. For instance, the order of the first 20 pages should be much more important than the order of the last 180 pages, as a Googler most of the time only cares about the first page of results, and rarely examines anything after the first 20.

To conclude, the HITS algorithm seems to be the best at ranking the pages, but it comes at a price: runtime performance.

4 Summary

This paper provides a basic exploration into PageRank and the HITS algorithm, by detailing its use in a search engine. We began by first learning about these algorithms and detailing how the PageRank, damped and undamped, algorithm and HITS algorithm function. Then, we provided the implementation of each of these algorithms in Julia code. Upon implementation and analysis, two various points were brought up

For the first point, we realized that the most efficient out of the three algorithm was the undamped PageRank algorithm, followed by the PageRank with dampening, and lastly the HITS algorithm. We learned that the HITS algorithm would be the least efficient, as it calculates two vectors simultaneously. This contrast to the undamped PageRank and damped PageRank, as these algorithms only computer one vector at a time. Comparing damped and undamped, undamped requires fewer calculations, thus making it more efficient than the damped algorithm.

For the second point, we realized that the HITS algorithm typically gives more accurate results. This is true as it focuses on a two-dimensional analysis. Not only does it focus on which websites have the most "authority", but it also focuses on which websites send out a greater percentage of "surfers." Thus, by analyzing the two sides of "surfing," the algorithm is able to generate a more efficient result. This contrasts to the undamped and damped PageRank, which do a one-dimensional analysis that focuses solely on distribution among the webpages. Thus, throughout this paper we learned about the basis of the how a Google search engine functions and the various implementations of the search engine.

References

[1] Amine Amrani. Pagerank algorithm, fully explained. *Medium*, 2020. https://towardsdatascience.com/pagerank-algorithm-fully-explained-dc794184b4af.

[2] Jayant Bisht. Page rank algorithm and implementation. *GeeksforGeeks*, 2017. https://www.geeksforgeeks.org/page-rank-algorithm-implementation/.

[3] Matt Cutts. Pagerank sculpting. *Matt Cutts: Gadgets, Google, and SEO*, 2009. https://www.mattcutts.com/blog/pagerank-sculpting/.

[4] Reducible. Pagerank: A trillion dollar algorithm. YouTube, 2022. https://www.youtube.com/watch?v=JGQe4kiPnrU.

Appendix

In [102]:
```
import Pkg; Pkg.add("DataFrames")
Pkg.add("CSV")
```

```
  Resolving package versions...
 No Changes to `~/.julia/environments/v1.9/Project.toml`
 No Changes to `~/.julia/environments/v1.9/Manifest.toml`
  Resolving package versions...
 No Changes to `~/.julia/environments/v1.9/Project.toml`
 No Changes to `~/.julia/environments/v1.9/Manifest.toml`
```

Import libraries that will be useful to us:

- CSV: Used to read the adjacency list CSV file
- LinearAlgebra: Used to calculate transposes, Frobenius norms, etc.
- Statistics: Used to calculate MSE for testing

In [1]:
```
using CSV
using LinearAlgebra
using Statistics
```

Helper function that converts CSV file into an adjacency matrix

These helper functions are used to convert graphs stored in a CSV file into matrices. This is useful for reading larger graphs stored in separate files that otherwise would be difficult to hard-code into the Julia program.

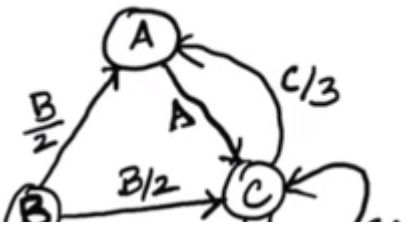
As an example, a CSV file has the following structure. It is an adjacency list but in CSV format

```
3
1, 3
1, 2, 3
```

So row 1 (the first node) is connected to the 3rd node only.

Row 2 (the second node) is connected to both the 1st and 3rd nodes.

Row 3 (the third node) is connected to the 1st, 2nd, and 3rd nodes.



So the resulting adjacency matrix would be:

```
0 1 1
0 0 1
1 1 1
```

In [2]:
```
function csv_to_adjacency_matrix(file_path::String)
    # Read the CSV file
    # The empty rows are nodes with no connections. We still want to in
    # little to no impact in ranking importance.
    csv_file = CSV.File(file_path, header=false, silencewarnings=true,

    adjacency_matrix = zeros(Float64, length(csv_file), length(csv_file

    # Iterate over each row
    for (node, row) in enumerate(csv_file)
        # Iterate over each field in the row
        for field in row
            if !ismissing(field)
                adjacency_matrix[field, node] = 1
            end
        end
    end

    return adjacency_matrix
end
```

Out[2]:
```
csv_to_adjacency_matrix (generic function with 1 method)
```

Helper function that converts adjacency matrix into a Markov matrix

Normalizes all the columns so the sum of their probabilities equals 1.

For example, the previous adjacency matrix becomes the following Markov matrix:

```
0.0  0.5  0.333333
0.0  0.0  0.333333
1.0  0.5  0.333333
```

In [3]:
```
function normalize_columns(file_path::String)
    # Create a copy of the matrix to avoid modifying the original
    normalized_matrix = csv_to_adjacency_matrix(file_path)
    matrix = copy(normalized_matrix)

    # Get the number of columns
    num_columns = size(matrix, 2)

    for col in 1:num_columns
        # Calculate the sum of the column
        col_sum = sum(matrix[:, col])

        # Avoid division by zero
        if col_sum != 0
            # Normalize the column
            normalized_matrix[:, col] .= matrix[:, col] ./ col_sum
        end
    end

    return normalized_matrix
end
```

Out[3]:
```
normalize_columns (generic function with 1 method)
```

In [5]:
```
base_path = "/home/jose/Documents/CMU/21241/final-project/dummy-graphs/

# Put the path to your CSV file
# Concat strings in Julia with *
markov_matrix = normalize_columns(base_path * "small-graph.csv")
adjacency_matrix = csv_to_adjacency_matrix(base_path * "small-graph.csv

markov_matrix
```

Out[5]:
```
3×3 Matrix{Float64}:
 0.0  0.5  0.333333
 0.0  0.0  0.333333
 1.0  0.5  0.333333
```

Below, we create the PageRank algoritm with a dampening factor. These are their arguments:

- markov_matrix: The Markov matrix of the random walker.
- dampingFactor: A Float value between 0 and 1
- tol: The tolerance -- is correlated to how many iterations we want the algorithm to run

In [6]:
```
# This function does not modify the input markov matrix, so we don't ne
function dampedPageRank(markov_matrix::Matrix{Float64}, dampingFactor,
    n = size(markov_matrix, 1)
    r = fill(1.0 / n, n)

    teleport = (1.0 - dampingFactor) / n
    iteration = 0
    delta = tol

    # Iteratively compute the PageRank
    while delta >= tol
        r_new = dampingFactor * (markov_matrix * r) .+ teleport
        # Calculate Frobenius norm
        delta = norm(r_new - r, 1)
        r = r_new
    end

    return r
end
```

Out[6]:
```
dampedPageRank (generic function with 2 methods)
```

Below, assume an initial vector where surfers are distributed evenly among the nodes. Then find the steady state vector to find distribution of all surfers among the states

In [7]:
```
function findStateProbabilities(markovMatrix::Matrix{Float64}; tol::Flo
    n = size(markovMatrix, 2)
    surfDistribution = fill(1/n, n)
    newSurfers = zeros(n)
    change = Inf # Theoretically infinite at the start

    while surfDistribution != newSurfers
        # newSurgers = markovMatrix .* surfDistribution
        mul!(newSurfers, markovMatrix, surfDistribution)
        change = norm(newSurfers - surfDistribution)  # Convergence che
        surfDistribution .= newSurfers
    end

    return surfDistribution
end
```

Out[7]:
```
findStateProbabilities (generic function with 1 method)
```

Below is the HITS algoirthm. It calculates the authority scores and hub scores.

In [8]:
```
function hits_algorithm(hitsMatrix::Matrix{Float64})
    adjacencyMatrixTran = copy(hitsMatrix)
    adjacencyMatrix = transpose(copy(hitsMatrix))
    numStates = size(adjacencyMatrixTran, 1)
    hubScores = ones(numStates)
    authorityScores = ones(numStates)
    prevHubScores = zeros(numStates)
    prevAuthorityScores = zeros(numStates)
    k = 0
    while hubScores != prevHubScores||
          authorityScores != prevAuthorityScores
        k += 1
        authorityScores .= adjacencyMatrixTran * hubScores
        hubScores .= adjacencyMatrix * authorityScores
        prevAuthorityScores .= authorityScores
        prevHubScores .= hubScores
    end
    authorityScores ./= norm(authorityScores, 2)
    hubScores ./= norm(hubScores, 2)
    println(k)
    return hubScores, authorityScores
end
```

Out[8]:
```
hits_algorithm (generic function with 1 method)
```

TESTING our three different algorithms

In [9]:
```
# In this function we modify the authority vector, so we crete a copy w
function indicesSortedVector(authority)
    testAuthority = copy(authority)
    sortedIndexAuthority = sortperm(testAuthority, rev=true) # Greatest

    sortedVectorAuthority = testAuthority[sortedIndexAuthority]

    return sortedIndexAuthority, sortedVectorAuthority
end
```

Out[9]:
```
indicesSortedVector (generic function with 1 method)
```

In [10]:
```
@time hub, authority = hits_algorithm(adjacency_matrix)

organizedPagesIndex, organizedPages = indicesSortedVector(authority)
println("HIT")
println(organizedPagesIndex)
println(organizedPages)
```

```
1
  0.031122 seconds (14.76 k allocations: 1.239 MiB, 96.85% compilation
time)
HIT
[3, 1, 2]
[0.8017837257372732, 0.5345224838248488, 0.2672612419124244]
```

In [11]:
```
@time orderedStates = dampedPageRank(markov_matrix, 0.85)


organizedPagesIndexDamp, organizedPagesDamp = indicesSortedVector(order
println("Dampening")
println(organizedPagesIndexDamp)
println(organizedPagesDamp)
```

```
  0.000013 seconds (54 allocations: 4.203 KiB)
Dampening
[3, 1, 2]
[0.5208692975273159, 0.28155110874039785, 0.1975795937322862]
```

In [12]:
```
@time noDampening = findStateProbabilities(markov_matrix)

organizedPagesIndexNoDamp, organizedPagesNoDamp = indicesSortedVector(n

println("No Dampening")
println(organizedPagesIndexNoDamp)
println(organizedPagesNoDamp)
```

```
  0.000012 seconds (4 allocations: 304 bytes)
No Dampening
[3, 1, 2]
[0.6111111111111112, 0.2777777777777778, 0.1111111111111111]
```

Code block that helps visualize how the web pages would be ranked according to our three different algorithms.

In [114]:
```
function displayPages(lineIndices)
    # Read all lines from the file
    lines = readlines(base_path * "links.txt")

    # Iterate over the indices and print the corresponding line
    for index in lineIndices
        println(lines[index])
    end
end

println("HITS ranking")
displayPages(organizedPagesIndex)

# println("Damped ranking")
# displayPages(organizedPagesIndexDamp)

# println("Undamped ranking")
# displayPages(organizedPagesIndexNoDamp)
```

```
HITS ranking
https://www.facebook.com/thebeatles/ (https://www.facebook.com/thebeat
les/)
https://twitter.com/thebeatles?lang=en (https://twitter.com/thebeatle
s?lang=en)
https://www.youtube.com/channel/UCc4K7bAqpdBP8jh1j9XZAww (https://www.
youtube.com/channel/UCc4K7bAqpdBP8jh1j9XZAww)
https://www.instagram.com/thebeatles/?hl=en (https://www.instagram.com
/thebeatles/?hl=en)
https://www.tiktok.com/@thebeatles?lang=en (https://www.tiktok.com/@th
ebeatles?lang=en)
https://www.thebeatles.com/ (https://www.thebeatles.com/)
https://en.wikipedia.org/wiki/The_Beatles (https://en.wikipedia.org/wi
ki/The_Beatles)
https://www.rollingstone.com/music/music-lists/100-greatest-beatles-so
ngs-154008/ (https://www.rollingstone.com/music/music-lists/100-greate
st-beatles-songs-154008/)
https://www.amazon.com/music/player/artists/B00GB0QTOY/the-beatles (ht
tps://www.amazon.com/music/player/artists/B00GB0QTOY/the-beatles)
```

How close are our rankings to the actual Google rankings

The order in which the links are in `links.txt` are how Google ordered them when searching for the query. We can use MSE (Mean Squared Error) to calculate how off we are.

Better way to test

MSE does not capture a sense of "order" effectively, because enphasis is place on difference in magnitude between the two vectors rather than order.

A better metric is to use the Spearmans Rank Correlation, where:

- 1 is a perfectly ordered vector

- 0 is perfect disorder: No correlation between our rankings and Google's rankings
- -1 is a perfect inverse order

So the closer we are to 1, the better our algorithm is (using Google rankings as our goal)

In [60]:
```
function spearmansRankCorrelation(x)
    n = length(x)
    y = 1:n  # Perfectly ordered vector
    return cor(x, y)
end

println("HITS rank compared to Google")
println(spearmansRankCorrelation(organizedPagesIndex))
println("Damped rank compared to Google")
println(spearmansRankCorrelation(organizedPagesIndexDamp))
println("Undamped rank compared to Google")
println(spearmansRankCorrelation(organizedPagesIndexNoDamp))
```

```
HITS rank compared to Google
0.7466569925630329
Damped rank compared to Google
0.7310828949755124
Undamped rank compared to Google
0.7319317975693814
```

Generator

Julia script that generates artificial graphs of a given size. Stores these in a CSV file to be later parsed in `pagerank.ipynb` .

Recall that the CSV file has the following structure (as an example):

```
3
1, 3
1, 2, 3
```

So row 1 (the first node) is connected to the 3rd node only.

Row 2 (the second node) is connected to both the 1st and 3rd nodes.

Row 3 (the third node) is connected to the 1st, 2nd, and 3rd nodes.

Since these graphs are connected at random, it may not be representative of what an actual website dataset may look like. Nevertheless we can create huge graphs with this script.

In [2]:
```julia
using Random
using StatsBase

function generate_graph_text_file(num_nodes::Int, file_path::String)
    open(file_path, "w") do file
        # Generate connections for each node
        for node in 1:num_nodes
            # Randomly decide the number of connections for this node
            num_connections = rand(1:num_nodes - 1)

            # Choose random nodes to connect with
            connections = sample(setdiff(1:num_nodes, [node]), num_conn

            # Write connections to the file
            connections_str = join(connections, ", ")
            write(file, "$connections_str\n")
        end
    end
end

generate_graph_text_file(5000, "large-graph-5000.csv")
```

Quick script that creates a realistic website dataset

The goal for this script is to create a realistic graph with real websites. We scrape a set of websites for a specific search query, and paste them in `links.txt`, and for each website, we find all the hyperlinks. These are the node connections. Then we build the CSV dataset from there. This graph dataset will then be fed into our algorithm in `pagerank.ipynb`.

The resulting graph is very sparse, unlike the densely connected graph that `generator.ipynb` creates artificially.

By creating a realistic graph, albeit smaller than one used by Google, we hope to capture many of the real life obstacles PageRank faces. These include sinks, important sources, popular social media websites that may be irrelevant, links to ads, and so much more. This wouldn't be possible otherwise with a randomly created graph.

In [1]:
```
# What you want to search
query = "the beatles"
# How many results do you want
num_nodes = 200
```

Out[1]:
```
200
```

In [ ]:
```
using HTTP
using Gumbo
using Cascadia
```

Scrape all the websites that will be used for our nodes. We can do a quick Google search.

Keep in mind that sometimes Google blocks this script because we are querying too rapidly, and it detects that we are a bot. So there is a cooldown time period.

An interesting benchmark for our PageRank algorithm would be how close our results are to a real Google search.

In [2]:
```
results = String[]

# Open the file and read line by line
open("the-beatles-125/links.txt", "r") do file
    for line in eachline(file)
        # Strip the line to remove any leading/trailing whitespace
        clean_line = strip(line)

        # Append the cleaned line to the links array
        push!(results, clean_line)
    end
end

print(results)
keys = [extract_domain(f) for f in results]
print(keys)
```

Helper function that gets all the hyperlinks from a web page

The popularity of a website is determined by how much other websites reference it (how many connections that node has). This function scrapes the website for any URLs it makes reference to.

One cool thing we do is we ignore all the hyperlinks that reference its own website. For example, the webpage

`https://www.reliancedigital.in/solutionbox/how-to-diagnose-laptop-problems-and-fix-them/`

Has the following hyperlinks inside its own text:

```
https://www.reliancedigital.in/solutionbox/category/product-reviews/
https://www.reliancedigital.in/solutionbox/category/product-reviews/mobiles-tablets-reviews/
https://www.reliancedigital.in/solutionbox/category/product-reviews/computers-laptops-product-review/
https://www.reliancedigital.in/solutionbox/category/product-reviews/tv-audio-product-reviews/

...

https://www.reliancedigital.in/solutionbox/category/buying-guides/home-appliances-buying-guides/
https://www.reliancedigital.in/solutionbox/category/buying-guides/health-personalcare/
https://www.reliancedigital.in/solutionbox/category/buying-guides/batteries-juice-packs/
https://www.reliancedigital.in/solutionbox/category/buying-guides/gaming-buying-guides/
```

and many, many more that come from the same domain, `reliancedigital.in` . We only want to count this domain name once as a result, because it we counting it multiple times, it will blow up its own popularity in the graph because it keeps referencing itself. This will skew our PageRank algorithm findings as it'll think this website is really popular because it keeps getting referenced, but in reality its just referencing itself (almost like cheating).

So we filter all the hyperlinks that come from the same domain. That way we keep the hyperlinks that really come from other sources, and that adds variety to the graph and is a more representative showing of popularity.

In [5]:
```
function get_all_hyperlinks(url::String)
    # Perform the HTTP request and parse the HTML
    response = HTTP.get(url)
    soup = parsehtml(String(response.body))

    # Extract all hyperlinks
    urls = String[]
    for link in eachmatch(Selector("a"), soup.root)
        hyperlink = get_attribute(link, "href")

        # Continue to the next link if hyperlink is nothing
        if isnothing(hyperlink)
            continue
        end

        # Extract the domain name from the hyperlink
        domain_name = HTTP.URIs.URI(hyperlink).host

        # Filter out self-referencing hyperlinks
        if domain_name !== nothing && domain_name ∉ url
            push!(urls, domain_name)
        end
    end

    return urls
end
```

Helper function that makes a temp CSV file to store data so far

We store the data in a file as a intermediary instead of in a variable.

In [26]:
```
function make_nodes(path)
    # Read lines from the links.txt file
    lines = readlines("links.txt")

    # Open the output file for writing
    open(path, "w") do fp
        for i in 1:length(lines)
            try
                println(i)
                # Write the key to the file, assuming 'keys' array is a
                write(fp, chomp(keys[i]))

                # Get all hyperlinks from the line
                hlinks = get_all_hyperlinks(strip(lines[i]))

                # Write each hyperlink to the file
                for l in hlinks
                    write(fp, ", " * l)
                end

                # Write a newline to separate entries
                write(fp, "\n")

                # Sleep for a short time
                sleep(0.1)

            ### This is usually bad practice but if we encounter an con
            # we want to continue as if nothing happened in order to bu
            catch e
                continue
            end
        end
    end
end

make_nodes("temp.csv")
```

Create a dictionary from the temp CSV file.

Each key is a node, and the entries are an array to all the websites that node has hyperlinks to. We then remove duplicate hyperlinks because only one is needed.

In [28]:
```
function csv_to_dict(path)
    node_dict = Dict{String, Vector{String}}()

    # Open the file and read line by line
    open(path, "r") do file
        for line in eachline(file)
            # Split the line by comma and strip whitespace
            entries = split(strip(line), ',')

            # Check if the line is not empty and then process
            if !isempty(entries)
                key = entries[1]
                values = entries[2:end]

                # Add to the dictionary
                node_dict[key] = get(node_dict, key, String[]) |> x ->
            end
        end
    end

    return node_dict
end

node_dict = csv_to_dict("temp.csv")
println(node_dict)
```

```
{'www.wikihow.com': ['www.facebook.com', 'fr.wikihow.com', 'www.pinter
est.com', 'www.carsdirect.com', 'www.wikihow.it', 'www.youtube.com', '
knowhow.napaonline.com', 'www.gonift.com', 'twitter.com', 'ar.wikihow.
com', 'autorepair.about.com', 'ru.wikihow.com', 'www.instagram.com', '
www.tiktok.com'], 'www.quora.com': [], 'techtirerepairs.com': [], 'ww
w.mach1services.com': [], 'germaniainsurance.com': ['classic.germaniac
onnect.com', 'www.facebook.com', 'www.autoguide.com', 'germania-ciam.o
kta.com', 'www.instagram.com', 'policyholders.germaniaconnect.com', 't
witter.com', 'www.thedrive.com', 'www.brandtackle.com', 'roadsumo.com
', 'germaniacreditunion.com', 'www.linkedin.com'], 'www.amfam.com': ['
www.facebook.com', 'play.google.com', 'www.ghsa.org', 'instagram.com',
'www.pinterest.com', 'newsroom.amfam.com', 'injuryfacts.nsc.org', 'ww
w.twitter.com', 'www.youtube.com', 'apps.apple.com', 'www.digicert.com
', 'b2b.amfam.com', 'www.iii.org', 'www.ncsl.org', 'www.linkedin.com',
'chat-ui.amfam.com'], 'www.progressive.comwww.sullivantire.com': ['ww
w.linkedin.com', 'www.facebook.com', 'www.tireindustry.org', 'www.usti
res.org', 'www.sullivantirewholesale.com', 'www.youtube.com', 'www.twi
tter.com', 'www.pintrest.com', 'twitter.com', 'portal.sullivantire.com
', 'www.instagram.com'], 'www.bicycling.com': ['www.dynaplug.com', 'ww
```

Main function that creates the CSV graph

Parses the dictionary and gets the indices of all the connections, writing them into a file.

In [29]:

```
function create_csv_dataset(path)
    # Open the file for writing
    open(path, "w") do file
        for (n, connections) in node_dict
            indexed_connections = String[]

            for c in connections
                if c in keys
                    push!(indexed_connections, string(findfirst(==(c),
                end
            end

            # Write to file
            write(file, join(indexed_connections, ", "))
            write(file, "\n")
        end
    end
end

create_csv_dataset("test.csv")
```