21-241 Final Project

Lucas Zheng

Friday, December 3rd 2021

Introduction

For our 21-241 final project, I chose to explore the project on Markov chains, random walks, and PageRank. I thought it would be interesting to understand the Google PageRank algorithm since I use Google everyday, so I chose to do this project.

The PageRank Algorithm

0.1 What is it?

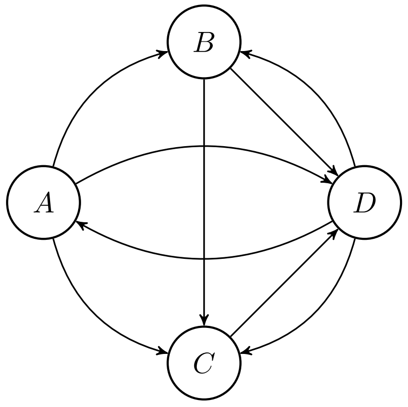
PageRank is the algorithm which Google uses to rank the web pages we see every time we search something up. It is so important because if it does not rank well, we might come across hundreds of websites that are useless to us, though we would realistically give up after browsing through 10.

0.2 Linear Algebra Background

The core linear algebraic ideas that gave birth to this algorithm are Markov Chains and Markov Matrix. Recall Markov matrices, which is a matrix that describes the transition probabilities between N states. The PageRank Markov chain describes a graph of N web pages and contains the probabilities we can go from one web page to another via a web page's links. If we randomly walk from one page to another thousands of times through links, we will end up landing on some pages more than others and eventually reach a constant probability distribution. The web pages we land on the most are ranked at the top of our search page, while the ones we land on less and ranked lower.

0.3 Example

Consider the following Markov chain, which describes a network of 4 web pages, labeled A, B, C, D. Note that the edges are directed, since two pages do not necessarily link to each other both ways.



Since A points to B, we say that there is a link of web page A that redirects you to B. To populate our Markov matrix M, we simulate walking from each node to another. For example, from A we can go to B, C, and D. Thus, we have a probability of $\frac{1}{3}$ of going to each of those. This yields the general equation of

$$M_{ij} = \frac{\text{number of arrows from web page } j \text{ to } i}{\text{total number of arrows coming out of node } j}, \quad \text{for } i, j \in \{A, B, C, D\}$$

We do this for all our nodes and obtain the following Markov matrix.

$$M = \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 1/3 \\ 1/3 & 0 & 0 & 1/3 \\ 1/3 & 1/2 & 0 & 1/3 \\ 1/3 & 1/2 & 1 & 0 \end{pmatrix} \end{array} \begin{array}{c} \\ A \\ B \\ C \\ D \end{array}$$

Notice how each column adds up to 1 since we must leave the a web page and all entries are greater than or equal to 1. This proves to us that we are indeed working with a Markov matrix. The thing about Markov Matrices is that they all have an eigenvalue with the value 1, and the absolute value of all the other eigenvalues are less than 1.

Now to actually rank the web pages, we need to perform many random walks. M alone represents 1 random walk, M^2 represents 2 random walks, and so on. Let's compute M^{50} to simulate 50 random walks.

$$M^{50} = \begin{pmatrix} 0.136364 & 0.136364 & 0.136364 & 0.136364 \\ 0.181818 & 0.181818 & 0.181818 & 0.181818 \\ 0.272727 & 0.272727 & 0.272727 & 0.272727 \\ 0.409091 & 0.409091 & 0.409091 & 0.409091 \end{pmatrix}$$

We can see that all the columns are identical, meaning that with large number of random walks, the probability of landing of one page becomes constant and independent of the page we came from. If we compute M^{51}, it is equal to M^{50} (at least to 6 decimal places), meaning we have reached a stable state.

We call the first column of our markov matrix $(0.136364, 0.181818, 0.272727, 0.409091)$ the stable vector of our Markov Matrix. Moreover, it is the eigenvector with eigenvalue 1 with component sum of 1.

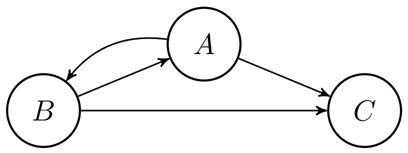
Now, we can rank our web pages according to the probabilities in this column. Suppose we start off with starting vector $\pi = (1/4, 1/4, 1/4, 1/4)$, meaning we start off at one of the 4 websites with a quarter chance each. After 50 random walks, we get $\pi = M^{50}\pi = (0.14409, 0.18732, 0.271610, 0.39697)$. We rank pages as D, C, B, A, with D having the highest rank (land of D roughly 40% of the time). Intuitively, this makes sense too since A has the least arrows pointing to it, and D is the busiest one, tied with C with most arrows arrows going into.

Model Adjustments

Our understanding is not perfect just yet. There are some extra cases that the algorithm needs to cover for.

0.4 Stochasticity Adjusment

Consider the simple graph below.



We see that C has no links coming out of it, so what happens we we randomly walk onto C? We will have nowhere to go. We call C a dangling node. We solve this problem by performing a

stochasticity adjusment. We replace zero columns in M with $1/n\ \vec{e}$, where $\vec{e}$ is a vector with all 1's. So now when we land of C, we have a $1/3$ chance of going to a node (it can visit itself!), making M stochastic because now it has random chance of going to other websites at any step. We call this new stochastic matrix S.

$$M = \begin{pmatrix} 0 & 1/2 & 0 \\ 1/2 & 0 & 0 \\ 1/2 & 1/2 & 0 \end{pmatrix} \implies S = \begin{pmatrix} 0 & 1/2 & 1/3 \\ 1/2 & 0 & 1/3 \\ 1/2 & 1/2 & 1/3 \end{pmatrix} \quad \text{by stochastic adjustment}$$

We can generalize this stochasticity fix with the equation

$$S = M + (1/n\ \vec{e})\ \vec{a}^T \qquad \text{note } \vec{e}\ \vec{a}^T \text{ is } n \times n$$

where $\vec{a}_i = 1$ if website i is a dangling node and $\vec{a}_1 = 0$ otherwise. We name $\vec{a}$ the dangling node vector.

0.5 Primitivity Adjustment

Stochastic adjustment only applies to graphs with dangling nodes; however, primitivity adjustment is crucial for all graphs. We need primitive adjustment to make sure our graphs are primitive, meaning π is unique and each entry of M after large iterations is greater than 0.

Think about a very dense graph where the middle nodes are very busy, while the nodes are the outside are extremely sparse. This might never converge since the middle nodes could monopolize the random walking, causing the page rank of the edge nodes to go to 0. We introduce the idea of teleportation to solve this issue, where instead of randomly walking from website to another, we teleport to another website in the graph. This is analogous so a user typing a new url in the search bar instead of clicking hyperlinks.

Our new stochastic and primitive matrix - the Google Matrix - G is calculated by the following equation

$$G = \alpha S + (1 - \alpha)\ 1/n\ \vec{e}e^T \qquad \text{note } \vec{e}e^T \text{ is } n \times n \text{ matrix with all 1s}$$

where, $S = M + (1/n\ \vec{e})$ from the previous subsection. The new variable α is the teleportation constant, a scalar between 0 and 1. It is also known as the dampening factor. If $\alpha = 0.8$, then 80% of the time we use random walks, and 20% of the time we teleport to a random node. $1/n\ \vec{e}e^T$ is a matrix with all $1/n$'s, so it is equally likely to teleport to all websites.

Ranking the Pages

Now that we have G, we rank the pages using the first eigenvector of G instead of M as described in section **0.3** to account for all the new adjustments. To remind ourselves, if we calculate π for

G to be (0.2, 0.5, 0.3), then 20% of the time we are on website 1, 50% of the time we are on website 2, and 30% of the time we are on website 3. Consequently, we rank them as 2,3,1 where 2 is the most important website and 1 is the least.

Algorithm Design

Now that it is clearer what the PageRank algorithm does, let us do the same with our own example. First, we make a pseudocode of our implementation. Our goal is to accept an input of a list of websites and for each website a list of links that go to the other websites. Then we need to construct a Markov matrix from this and rank the websites in the correct order.

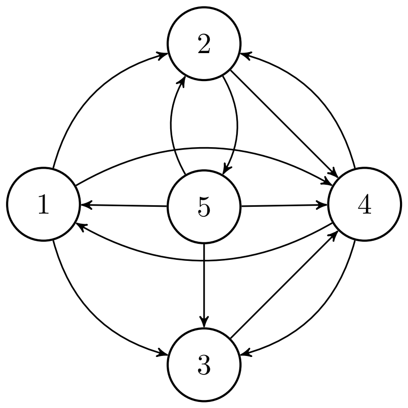
To represent our input containing a list or lists, we will take in a 2D array. Each array at each index represents a different website and those arrays contains links to other websites. Then, we compute the Markov Matrices and random walks to find the ranking.

0.6 Pseudocode

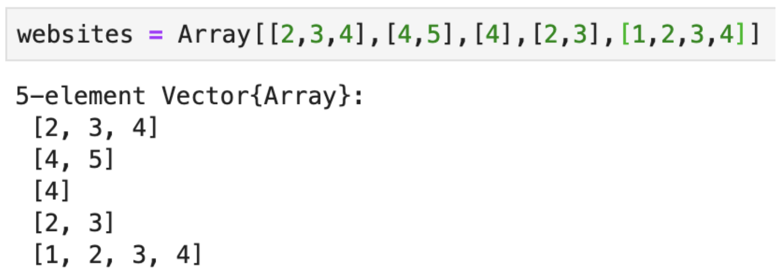
```
Initialize variables and Markov Matrix $M$ as 2D array with all zero's.
For each array $j$ in input {
    For index $i$ in array $j$ {
        input[i][j] is link from website $j$ to website $i$
        Update $M$ using equation in 0.3 Example
    }
}
Compute stochastic matrix S from M.
Compute Google matrix G from S.
Transform starting vector $\pi$ (vector of all $1/n$ since we start at random website) by G until it converges (norm of difference between current and last iterations $< 10^{-5}$)
Rank the websites by index with highest probability according to $\pi$.
```

0.7 Input

For our websites and links, instead of using A, B, C, D, E, we will represent them using numbers 1, 2, 3, 4, 5 for the sake of simplicity in our input. This is so that accessing indexes become easy (Julia arrays are 1-indexed). For example, if array contains [1, 3, 5], then that website contains links to websites 1, 3, and 5. The index we are at in our input will also always correspond to whichever website we are dealing with. Our input will be the following graph.



We represent the graph above using the following 2D array:



```
websites = Array[[2,3,4],[4,5],[4],[2,3],[1,2,3,4]]
```

```
5-element Vector{Array}:
 [2, 3, 4]
 [4, 5]
 [4]
 [2, 3]
 [1, 2, 3, 4]
```

0.8 Function Implementation

Now we implement the function to process our input.

First, we declare our function and initialize our important variables. There are no parameter types for functions in Julia.

```
function PageRank(websites)
    # important variables
    len = length(websites)  # number of websites
    e = vec(ones(len, 1))   # vectors with 1s
    a = vec(zeros(len, 1))  # vector with 1, will edit
    alpha = 0.85            # teleportation constant

    # zeros(m, n) creates m x n matrix full of 0s
    # M is square with length of number of websites
    M = zeros(len, len)
end
```

Secondly, we populate our Markov Matrix by looping through websites. We make use of *eachindex(array)*, which creates an iterable object to visit every index of array.

```
# go through each website
for j in eachindex(websites)
    # go through each link in website
    links = websites[j]; # the j'th website's links
    for i in links
        # i a link to another website!
        # length(links) = number of links website j has
        M[i,j] = 1 / length(links) # equation in Example 0.3
        # use [i,j] not [i][j] because Julia does not treat m x n
   matrix as array of arrays
        # functionality of [i,j] same as [i][j]
        # others left as 0 since no link
    end
end
```

With M constructed, we compute S and G using the equations discussed in sections **0.4** and **0.5**. We update our dangling node vector a that was initialized at the beginning then do the calculations.

```
# for i in x:y loops from int x to int y with step of 1
for i in 1:len
    # count(x->x==y, A) returns number of y's in A
    if(count(x->x==0, M[:,i])) == len
        # dangling node since col all 0s!
        a[i] = 1
    end
end

S = M + 1/len * e * transpose(a)

# alpha = 0.85, teleport 15% chance
G = alpha * S + (1 - alpha) * 1/len * e * transpose(e)
```

Now that our G is fully constructed, we need to find its stable vector. We do so using the power method.

```
epsilon = 1 # norm of difference
pi = 1/len * e # start at random website
# loop until epsilon small
# keep track of values of pi from 2 iterations
while epsilon > 10^-8 # we want epsilon <= 10^-8
    pi_next = G * pi
    epsilon = norm(pi_next - pi)  # norm difference
    pi = pi_next
end
```

Finally, we loop through π and print out the ranking for the websites with their respective pagerank values. We return G to the client. We also use the package "Printf" so we have access to the macro @printf.

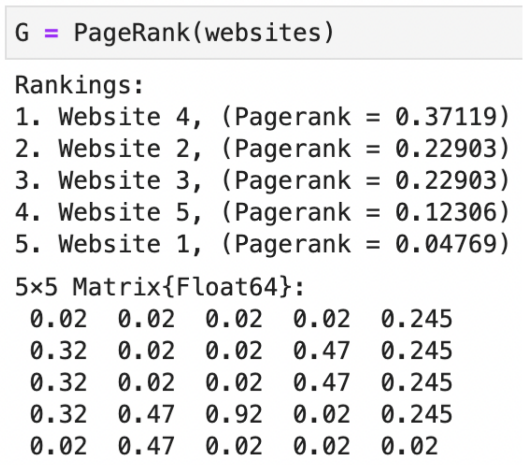
```
original = copy(pi) # creates new separate copy with same elements
println("Rankings:")
counter = 1 # for printing rankings
while !isempty(pi)
    # findmax return 2-tuple: (value, index)
    a,b = findmax(pi)
    web_num = findall(x->x==a, original)[1]
    # [1] in case of multiple indexes with same value
    original[web_num] = 0
    # set to 0 so we don't go to same index for next identical
   value
    # safe bc pagerank values are never 0 after our adjustments
    print(counter, ". Website ", web_num)
    @printf ", (Pagerank = %.5f%s" a ")"
    # @printf allows us to print certain number of decimal places
    # here, it is 5 decimal places
    println()
    # remove the max value to find next biggest in next loop
    deleteat!(pi, b)
    counter += 1
end

return G
```

0.9 Final Code (no comments)

```
using LinearAlgebra
suing Printf

function PageRank(websites)

    len = length(websites)  # number of websites
    e = vec(ones(len, 1))   # vectors with 1s
    a = vec(zeros(len, 1))  # vector with 1, will edit
    alpha = 0.85            # teleportation constant

    M = zeros(len, len)

    for j in eachindex(websites)
        links = websites[j];
        for i in links
            M[i,j] = 1 / length(links)
        end
    end
    for i in 1:len
        if(count(x->x==0, M[:,i])) == len
            a[i] = 1
        end
    end

    S = M + 1/len * e * transpose(a)

    G = alpha * S + (1 - alpha) * 1/len * e * transpose(e)

    epsilon = 1
    pi = 1/len * e
    while epsilon > 10^-8
        pi_next = G * pi
        epsilon = norm(pi_next - pi)
        pi = pi_next
    end

    original = copy(pi)
    println("Rankings:")
    counter = 1
    while !isempty(pi)
        a,b = findmax(pi)
        web_num = findall(x->x==a, original)[1]
        original[web_num] = 0
        print(counter, ". Website ", web_num)
        @printf ", (Pagerank = %.5f%s" a ")"
        println()
        deleteat!(pi, b)
        counter += 1
    end

    return G

end
```

Results

Running our PageRank algorithm on websites in **0.5 Input** yielded the following results



```
G = PageRank(websites)
```

```
Rankings:
1. Website 4, (Pagerank = 0.37119)
2. Website 2, (Pagerank = 0.22903)
3. Website 3, (Pagerank = 0.22903)
4. Website 5, (Pagerank = 0.12306)
5. Website 1, (Pagerank = 0.04769)

5×5 Matrix{Float64}:
 0.02  0.02  0.02  0.02  0.245
 0.32  0.02  0.02  0.47  0.245
 0.32  0.02  0.02  0.47  0.245
 0.32  0.47  0.92  0.02  0.245
 0.02  0.47  0.02  0.02  0.02
```

Recall our input graph, which had 2 arrows into 1; 3 into 2; 3 into 3; 4 into 4; 1 into 5. The results make sense since the ranking is correlated to how many websites link to one and is correctly based upon their pagerank values. The one with most arrows should have the highest ranking since many others link to it. Additionally, there happens to be a tie with 2 and 3 which is logical because their columns in the Markov matrix is the same.

We can also see that our teleportation is taking effect by looking at the small decimal values in the spots in the G that were 0 in M. Moreover, since all nodes had arrows going out of them, there was no need of adjusting for stochasticity.

Huge Data set (Extension)

Let's perform PageRank on an actual data set! This is going to be quite large compared to our examples but Julia is know for handling large matrix operations. We will be using the a list of websites generated by a web crawl bot from hollins.edu. The website to the data set in linked in the References section at the end of the paper.

0.10 The "hollins.edu" dataset

The data set comes in form of a ".dat" file which we will convert to a ".csv" file so that it can be read by Julia. It has 29,887 rows and 2 columns. The first 6012 rows are all the websites, indexed from 1 to 6012. The first column is the website's index, while the second column is the website's url. All the rows after row 6012 describes how all these websites are interconnected. The first column contains an index j to a webpage (website on j'th row, 2nd column) and the second column contains an index i to a webpage (website on i'th row, 2nd column) that website j links to.

0.11 Parsing the data set

After downloading the data set, we drag it into the "21-241 Julia Notebook folder" so we can use it in our Jupyter workspace. However, it is still in .dat format which Julia cannot parse. On Stack Overflow, I found a post with a function that converts .dat to .csv. It is linked in References.

```
1  function dat2csv(dat_path::AbstractString, csv_path::
       AbstractString)
2      open(csv_path, write=true) do io
3          for line in eachline(dat_path)
4              join(io, split(line), ',')
5              println(io)
6          end
7      end
8      return csv_path
9  end
10
11 function dat2csv(dat_path::AbstractString)
12     base, ext = splitext(dat_path)
13     ext == ".dat" ||
14         throw(ArgumentError("file name doesn't end with `.dat`"))
15     return dat2csv(dat_path, "base.csv")
16 end
```

* There is meant to be a "$" in front of string base.csv on line 15.

We run dat2csv("hollins.dat") which outputs a "hollins.csv" file.

Now we want to get the data set into array format. We will use "CSV" and "DataFrames" packages. Now we run he following to store the csv file in variable df which will be 29,887 by 2.

```
df = DataFrame(CSV.File("hollins.csv"))
```

We want to seperate the file into its website names and links. Julia makes this easy as we can do A = B[1:5,4], which sets A to be the array containing rows 1 to 5 and column 4 of B.

```julia
websites = df[1:6012, 2]
links = df[6013:end,1:2] # end is last index of df
```

For websites, we only need the second column since the indexes match already. To get the i'th website, we simply do websites[i]. One annoying thing is that even though the second column of links are indexes, they are actually strings since the second column of websites (urls) came first. So, we will need to parse those strings into ints which we can use to index.

Code Modification

Since we have a new form of input into our pagerank function, we need to modify it a little. First, we will take in one more parameter into our function.

```julia
function PageRank(websites, links)
```

Secondly, we change how we update our Markov matrix.

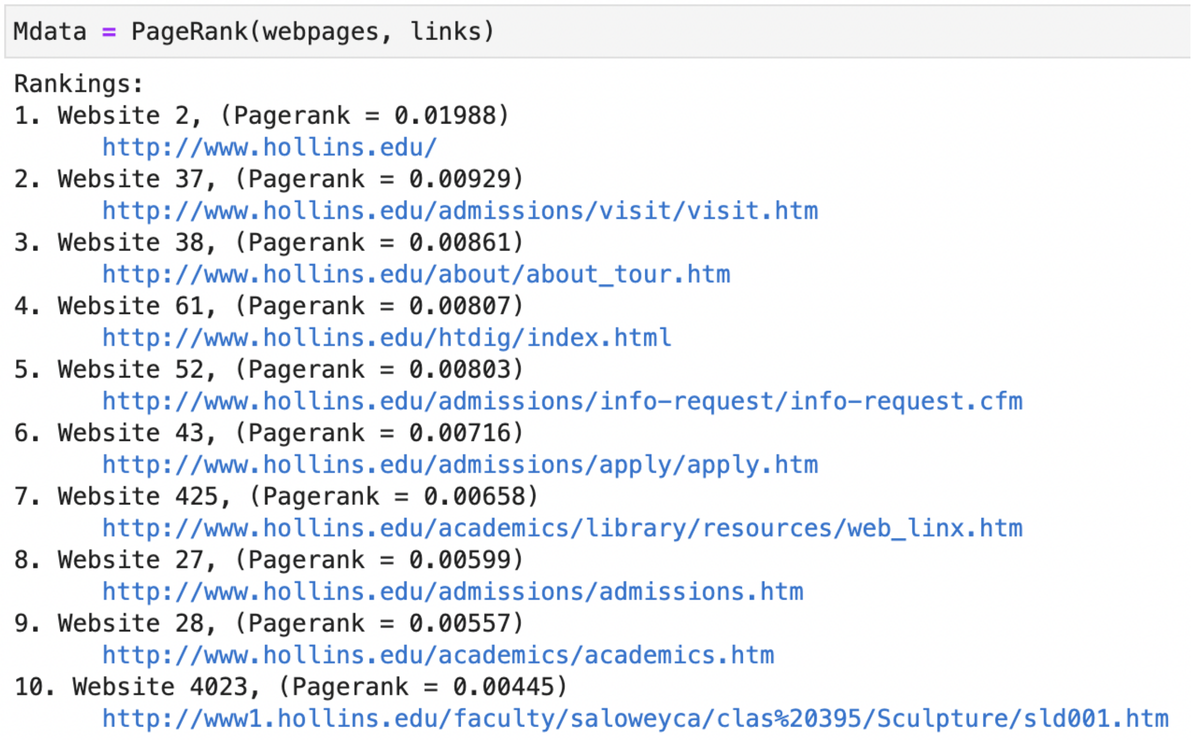
```julia
# go through each pair of links
# links has 2 columns, cols contain indexes to websites
# website in col 1 goes to website in col 2
# loop through all row of links
for j in eachindex(links[:,1])
    web_from = links[j,1]
    web_to = parse(Int32, links[j,2]) # parse bc it's string
    # link from web_from to web_to
    # divide 1 by number of websites web_from links to
    M[web_to, web_from] = 1 / count(x->x==web_from, links[:,1])
end
```

In fact, that is all we need to modify for calculations since all calculations from now depend on M and the variables we already created. The last thing to adjust is printing our rankings. Since our data set is very long, we do not want to print everything to the client. Instead, we will print out the top 10 highly ranked pages. We modify the while statement to be

```julia
while !isempty(pi)
    # findmax return 2-tuple: (value, index)
    a,b = findmax(pi)
    web_num = findall(x->x==a, original)[1]
    original[web_num] = 0
    print(counter, ". Website ", web_num)
    @printf ", (Pagerank = %.5f%s" a ")"
    println()
    println("      ", webpages[web_num])
    # remove the max value to find next biggest in next loop
    deleteat!(pi, b)
    counter += 1
    if counter == 11 # only print top 10 pagerank  values
        break
    end
end
```

Results for hollins data set

Running PageRank(websites, links), we get the following output.



```
Mdata = PageRank(webpages, links)
```

```
Rankings:
1. Website 2, (Pagerank = 0.01988)
      http://www.hollins.edu/
2. Website 37, (Pagerank = 0.00929)
      http://www.hollins.edu/admissions/visit/visit.htm
3. Website 38, (Pagerank = 0.00861)
      http://www.hollins.edu/about/about_tour.htm
4. Website 61, (Pagerank = 0.00807)
      http://www.hollins.edu/htdig/index.html
5. Website 52, (Pagerank = 0.00803)
      http://www.hollins.edu/admissions/info-request/info-request.cfm
6. Website 43, (Pagerank = 0.00716)
      http://www.hollins.edu/admissions/apply/apply.htm
7. Website 425, (Pagerank = 0.00658)
      http://www.hollins.edu/academics/library/resources/web_linx.htm
8. Website 27, (Pagerank = 0.00599)
      http://www.hollins.edu/admissions/admissions.htm
9. Website 28, (Pagerank = 0.00557)
      http://www.hollins.edu/academics/academics.htm
10. Website 4023, (Pagerank = 0.00445)
      http://www1.hollins.edu/faculty/saloweyca/clas%20395/Sculpture/sld001.htm
```

For context, Hollins is a university and these are all websites related to it. Hence, these results make a lot of sense. It is not shocking that the highest ranked website hollins' website's homepage, followed by links related to admissions, academics, and faculty. If I want to get to know a college better, I would definitely click on the homepage first, then go into academics which is what most people view as the most important aspect of college. Moreover, we see that it is at least twice as likely to land on the homepage compared to any other page judging by pagerank values because it is the central hub. It is promising to claim that our PageRank aglorithm is working.

Summary

I learned that Julia is hard, but really fast. I was quite confused by for looping in Julia because "for i in 10" doesn't actually loop through 1 to 10: it just is 10 from the beginning. It took a while for me to figure Julia out, but at the end it is amazing to see how Julia can compute high powers of matrices in no time.

I also gained more intuition about Markov matrices and stable transition matrices through working with graphs. The stable probability distribution makes a lot more sense now as I can match them directly to the graph visually. I feel that seeing what it represents is makes the concept of Markov matrices much clearer compared to describing an example without as much visualisation, such as the classic example of "$X\%$ of people leave country A to country D".

I also learned that no matter how simple the problem is, we always need to make adjustments for certain cases that inevitably arise. Now I know to be more meticulous when thinking about and solving problems in general.

Finally, I was reminded that there are so many useful resources online. Whether it was searching up Julia built-in functions or data sets, everything was open to the public. I did think finding a data set was very difficult since that took several hours. It also was my first time working with a data set I found myself, but I am happy that it works now.

References

[1] Klein Project Blog, *How Google works: Markov chains and eigenvalues*

`http://blog.kleinproject.org/?p=280`

[2] Wikipedia, *PageRank*

`https://en.wikipedia.org/wiki/PageRank`

[3] Langville, Amy N., and Carl D. Meyer. “Ranking Webpages by Popularity.” Google’s PageRank and Beyond: The Science of Search Engine Rankings, Princeton University Press, 2006, pp. 25 - 30,

`http://www.jstor.org/stable/j.ctt7t8z9.6`

[4] Hollins Dataset, *Data*

`https://www.limfinity.com/ir/`

[5] Reading a .dat file in Julia, issues with variable delimeter spacing

`https://stackoverflow.com/questions/61665998/reading-a-dat-file-in-julia-issues-with-variable-delimeter-spacing`

[6] Read CSV to DataFrame in Julia

`https://towardsdatascience.com/read-csv-to-data-frame-in-julia-programming-lang-77f3d0081c14`