Page Rank Paper

Josh Zhou, joshuazh@andrew.cmu.edu

December 11, 2020

1 Introduction: What is Page Rank?

Ever since its inception in 1998, Google has been the premier search engine of the 21st century. You can find hundreds of links to information on almost any topic or keyword within milliseconds. Although this may seem like black magic, the web page ranking algorithm that Google uses, dubbed Page Rank, actually has its basis in fundamental linear algebra topics.

2 Linear Algebra Background

Before diving into the specifics of the Page Rank algorithm, it is important to cover some linear algebra concepts that are fundamental to the algorithm. For starters, let's start by discussing eigenvectors, eigenvalues, and markov matrices and their properties.

An eigenvector is defined as a nonzero vector x of an $n \times n$ matrix A such that $Ax = \lambda x$ where λ is the eigenvalue. Additionally, for any $n \times n$ matrix A, A and A^T will share the same eigenvalues.

A markov matrix is defined as an $n \times n$ matrix M such that

1) $M_{ij} \geq 0$ for all i, j (i.e. non-negative entries)

2) $\sum_{i=1}^{n} M_{ij} = 1$ for all j (i.e. each column sums to 1)

This also means that every markov matrix has an eigenvalue 1 because $M^T * \mathbb{1} = \mathbb{1}$. Since a matrix and its transpose share the same eigenvalues, then M also has eigenvalue 1.

A probability vector has the same definition as a markov matrix, except it is a vector instead of a matrix.

Using this definition of a markov matrix, we can represent markov chains, which are systems that experience transitions from one state to another, as a markov matrix where M_{ij} is the probability to transition from state i to state j. Through markov chains, we can express a variety of different models and systems, one of which will be for the probability of landing on a certain website. Additionally, there exists a special subset of markov matrices called positive markov matrices which have the following properties:

1) $M_{ij} > 0$ for all i, j (i.e. strictly positive entries)

2) $\sum_{i=1}^{n} M_{ij} = 1$ for all j (i.e. each column sums to 1)

Notice how the only difference in definition between a markov matrix and a positive markov matrix is that a positive markov matrix must have strictly positive entries.

Lastly, consider the Perron-Frobenius theorem, which states that a positive markov matrix has the following properties:

1) 1 is an eigenvalue of multiplicity 1 (i.e. it's the only eigenvalue with value 1)

2) 1 is the largest eigenvalue where all other eigenvalues have absolute values less than 1

There exists an eigenvector π for eigenvalue 1, where each component is greater than or equal to 0 and WLOG, we can assume the sum of the components is 1.

Another key property of positive markov matrices is that for a positive markov matrix M, M^k for large k approaches $(x_1|\dots|x_1)$ where x_1 is the eigenvector with eigenvalue 1 (commonly referred to as the steady state vector)(matrix power convergence property).

3 The Page Rank Algorithm

With all of the background linear algebra information out of the way, we can get into the specifics on how exactly the Page Rank algorithm works. Although the web may just seem like an amalgamation of random websites, we can actually think of it as a directed graph, where each node is a website and an edge is a link between two websites. Consider the following sample graph:

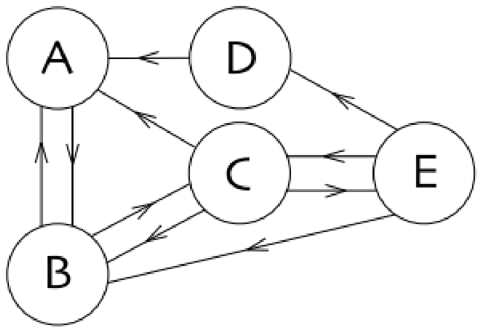


Figure 1: A simple web.

[Rou]

Let's examine node B. B has 2 outgoing edges, one to A and one to C. Thus, if we were to consider someone currently browsing site B, they could either go to site A or site C next. We give each site an equal probability (in this case it would be 1/2) of being clicked on next. Take node E as another example. If someone is currently on site E, we say they have a chance of going to sites C, D, or B next since we assume each site has equal probability of getting traversed to next. This methodology of starting at a page and choosing at random which page to go to next is called a random walk. Thus, at any stage, we can express the probability of going from one page to another in the form of a markov matrix, where each column represents the current page you're on and each row represents the page you'll arrive on. Thus, for the sample graph above, its markov matrix looks as follows:

$$P = \begin{array}{c} \\ \left(\begin{array}{ccccc} A & B & C & D & E \\ 0 & \frac{1}{2} & \frac{1}{3} & 1 & 0 \\ 1 & 0 & \frac{1}{3} & 0 & \frac{1}{3} \\ 0 & \frac{1}{2} & 0 & 0 & \frac{1}{3} \\ 0 & 0 & 0 & 0 & \frac{1}{3} \\ 0 & 0 & \frac{1}{3} & 0 & 0 \end{array} \right) \end{array} \begin{array}{c} \\ A \\ B \\ C \\ D \\ E \end{array}$$

[Rou]

With this matrix P, we can represent a step in the random walk with P, and we can represent n-steps as P^n. Thus, given some starting condition v (where v is a probability vector where each component represents the probability that you start on each website), we can represent the new probability vector after 1-step in the random walk as Pv.

Let's see what happens after n steps where n is relatively large. For this example, let's compute P^n and see what we get. Following our example, P^{32} looks as follows:

$$P^{32} = \begin{array}{c} \\ \\ \\ \\ \\ \end{array}\begin{array}{ccccc|c} A & B & C & D & E & \\ 0.293 & 0.293 & 0.293 & 0.293 & 0.293 & A \\ 0.390 & 0.390 & 0.390 & 0.390 & 0.390 & B \\ 0.220 & 0.220 & 0.220 & 0.220 & 0.220 & C \\ 0.024 & 0.024 & 0.024 & 0.024 & 0.024 & D \\ 0.073 & 0.073 & 0.073 & 0.073 & 0.073 & E \end{array}$$ [Rou]

Thus, for large n, the probability of being on any page is independent of the starting conditions as P has the form $(x_1|\ldots|x_1)$. In this example, notice how each column is the same. Thus, x_1 would be the vector (0.293, 0.390, 0.220, 0.024, 0.073). Using this, we can take x_1 and say that the ith component is the probability that we will be on page i at step n.

Thus, comparing each probability in x_1, we can rank pages by importance based on their probabilities. The vector x_1 is commonly referred to as the stationary distribution or the Page Rank vector. [Rou]

Although in this example, all the columns in P eventually converged to x_1, this is not always the case. Additionally, when browsing the web, instead of only relying on links to navigate through pages, sometimes we'll enter a webpage's address straight into the search bar and "spontaneously" appear on the page, whether it be because our attention has shifted elsewhere or the current page has no outgoing links. This current "spontaneous" action is not accounted for in our implementation of Page Rank. Another problem we are currently facing is the fact that we cannot guarantee that the eigenvalue 1 will have multiplicity 1, which means we can have multiple eigenvectors that correspond to eigenvalue 1. This is because the markov matrices we were dealing with were not positive.

Page and Brin came up with a solution for both of these problems, which they called damping. This solution transforms the markov matrix of probabilities with the following equation: $P_\beta = (1-p) * X + p * B$, where X is the $n \times n$ markov matrix, B is a $n \times n$ matrix where every value is $\dfrac{1}{n}$, and p is the damping factor [Tan]. The damping factor is typically around 0.15. This equation can be interpreted as follows:

Since most of the time when we navigate between pages, we are doing so through link traversal, we can take our original markov matrix, which represents the probabilities from going one page to another through links, and multiply it by (1-p), which is around 0.85 when p = 0.15. Thus, in total, the $(1-p) * X$ term represents the fact that most of the time (85%), we'll travel to another page through a website's links.

However, as previously mentioned, there are also times where we "spontaneously" travel to another page. We use the $p * B$ term to account for this. The damping factor p represents the probability that we'll spontaneously move to another page, and when this happens, we assume that each page has an equally likely chance of being landed on. Thus, every probability in markov matrix B will have equal probability (all the values are $\frac{1}{n}$). Thus, in total, the $p * B$ term represents the fact that some of the time (15%), we'll spontaneously travel to another page with each page having equal probability.

Additionally, since both X and B are markov matrices and $(1 - p) + p = 1$, then P_β will not only still be a markov matrix, it will also have all strictly positive probabilities, since we are adding on the $p * B$ term. Because of this, we can use the Perron-Frobenius theorem to show that there will only exist 1 eigenvector for the eigenvalue 1, and we can use the Matrix Power Convergence property to show that P_β^k for large k will be in the form $(x_1, \ldots, x_1)$ where x_1 is the sole eigenvector with eigenvalue 1. This vector x_1 will be our stationary distribution vector.

4 Implementation

Now that we have covered the basic idea of the Page Rank algorithm, we can discuss the code implementation of Page Rank.

To start off, let's examine the code for initialization and creating the Markov matrix. For my fixed examples, I create an array of websites and a dictionary to hold the links.

```julia
websites = ["google.com", "diderot.com", "discord.com", "youtube.com", "gmail.com", "zoom.com"]

websiteLinks= Dict(1=>["diderot.com", "youtube.com", "gmail.com", "zoom.com"],  #google.com
                   2=>["google.com", "gmail.com", "zoom.com"],  #diderot.com
                   3=>["google.com", "youtube.com", "diderot.com"],  #discord.com
                   4=>["gmail.com", "google.com"], #youtube.com
                   5=>["google.com", "diderot.com", "zoom.com", "discord.com"], #gmail.com
                   6=>["youtube.com", "google.com", "gmail.com"]) #zoom.com
```

Notice that each key in the dictionary corresponds to the index of the website in the websites array. Take google.com for example. Its website links are stored in the dictionary with key 1, which corresponds to its index in the websites array (since Julia uses 1 indexing).

I then create a function called generateMarkovMatFixed which takes in the array of websites and the dictionary of website links and generates a markov matrix from it. In pseudo code, the function looks as follows:

```
function generateMarkovMatFixed(websites, websiteLinks
    )

#websites is an array of websites
#websiteLinks is a dictionary of website links

    Declare variable, dim

    Set dim equal to length of websites

    Create a dim x dim 2d-array of zeros

    Loop through the websites in the website array by
    index

        Get the list of links for the current website

        Loop through the list of links by index

            Assign the value 1/(number of links for
    current website) to the 2d-array at row k and
    column n where k is the index of the current link
    in the websites array and n is index of the current
     website we are on in the outer loop

return the 2d-array

end
```

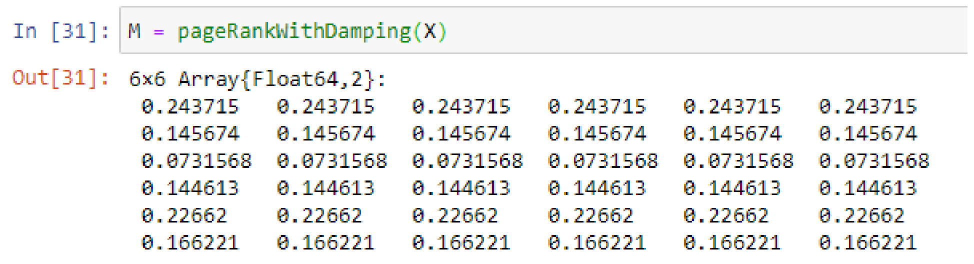
In actual code, this looks like:

```
function findInd(websites, websiteName)
    for i in 1:size(websites, 1)
        if websites[i] == websiteName
            return i
        end
    end
end

function generateMarkovMatFixed(websites, websiteLinks
    )
    numWebsites = size(websites, 1)
    m = zeros((numWebsites, numWebsites))

    for i in 1:numWebsites
        linkedSites = websiteLinks[i]
        for j in 1:size(linkedSites, 1)
            m[CartesianIndex.(findInd(websites, linkedSites[j]), i)] = 1/size(linkedSites, 1)
        end
    end
    return m
end
```

The function, findInd, simply finds the index of a website in the websites array. Once we have our markov matrix, we can apply the page rank equation with damping to see if it approaches the form $(x_1, \ldots, x_1)$. In our ongoing example, we store the output of generateMarkovMatFixed to a variable X and run the function, pageRankWithDamping, which takes in our markov matrix X and applies the page rank damping equation to it as well as taking it to a large poewr. The function looks as follows:

```
function pageRankWithDamping(matrix)
    p = 0.15
    B = ones(size(matrix)) * 1/size(matrix)[1]
    #P_{\beta} = (1 - p) * X + p * B
    #Take P_{\beta} to a large power
    return ((1 - p) * matrix + p * B)^100
end
```

After we get our positive markov matrix back, we can select a column and see if it is indeed an eigenvector with eigenvalue 1. One thing to note is I could have simply used the function eigvec (which comes in the Linear Algebra package) to get the eigenvector with eigenvalue 1, but for large graphs (like the one I analyze in a later section), there are more efficient ways to find x_1 instead of using the eigvec function and Gaussian elimination.

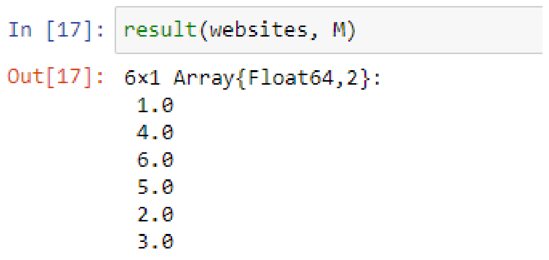


```
In [31]: M = pageRankWithDamping(X)

Out[31]: 6x6 Array{Float64,2}:
          0.243715   0.243715   0.243715   0.243715   0.243715   0.243715
          0.145674   0.145674   0.145674   0.145674   0.145674   0.145674
          0.0731568  0.0731568  0.0731568  0.0731568  0.0731568  0.0731568
          0.144613   0.144613   0.144613   0.144613   0.144613   0.144613
          0.22662    0.22662    0.22662    0.22662    0.22662    0.22662
          0.166221   0.166221   0.166221   0.166221   0.166221   0.166221
```

As shown in the image above, M takes the form $(x_1, \ldots, x_1)$ as expected since M is a positive markov matrix taken to a large power. From observation, we see that the probability at index 1 is the greatest, which corresponds to “google.com”. This makes sense because every other website has a link going to “google.com”, so it figures that at any given time, a user will most likely be on"google.com". To see the full rankings, we can run the results function, which simply takes the probabilities from the eigenvector x_1 and the list of websites and returns a list of ranks for each site. The function looks as follows:

```
function result(websites, pi)
    results = zeros((size(websites, 1), 1))
    sort = sortperm(pi)
    for i in 1:size(websites, 1)
        results[sort[i]] = size(websites, 1) - i + 1
    end
    return results
end
```

When we run result(websites, M) where M is the positive markov matrix taken to a large power, we get the following result:



```
In [17]: result(websites, M)

Out[17]: 6x1 Array{Float64,2}:
          1.0
          4.0
          6.0
          5.0
          2.0
          3.0
```

As expected, the first index, which corresponds to google.com, has rank 1, the highest priority. If we continue following the results, we find that gmail.com has rank 2, zoom.com has rank 3, etc.

5 Efficiency With Page Rank

When calculating the page rank vector with damping, we often provide it with a probability vector that represents the "starting conditions", which essentially is the probability that you'll start on any given page (call this vector S_0). For the sake of simplicity, we'll consider the starting conditions for each page to all have equal probability. Thus, with the page rank algorithm we just defined in the previous section, all we'd need to do is take $P_\beta^n * S_0$ for large n, and we would get our stationary distribution vector. However, what if we have billions of pages to consider? Then, this operation becomes quite costly, especially considering that P_β has all non-zero values. Additionally, for large matrices, calculating the eigenvector with eigenvalue 1 through ordinary means such as gaussian elimination go out the window as well.

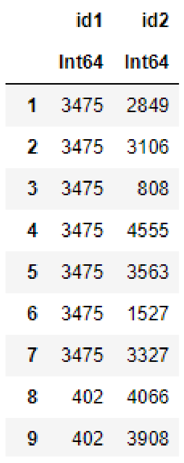
To simplify the cost of everything, let's do some algebraic manipulation. For starters, let S_{n+1} be defined recursively as $S_{n+1} = P_\beta * S_n$ where S_i is the stationary distribution at step i. However, since we know the equation for

$P_\beta = (1-p) * X + p * B$, then we can plug that into $S_{n+1} = P_\beta * S_n$ to get $S_{n+1} = (1-p)*X*S_n+p*B*S_n$. However, since B is a symmetric markov matrix with every value the same, then $B * S_n$, where the sum of the components of S_n is 1, is equal to S_0. Thus, the final equation is $S_{n+1} = (1-p) * X * S_n + p * S_0$, which will be significantly faster than the previous version since our markov matrix X, especially for large data sets, will have many 0 values. Thus, we won't have to deal with nearly as many taxing computations as P_β and can save a lot of time.

6 Extra Implementation: Real Datasets

Now that we have covered efficiency, let's apply Page Rank to a real dataset. For this example, we'll be using a Wikipedia Article Networks webgraph dataset. This dataset represents the page-page networks on specific topics (in this example, we'll be using the dataset on squirrels). In the squirrel dataset, there are 5,201 nodes (webpages) and 198,493 edges (links between pages). There are some things to note about this dataset. First is that all of the edges are undirected, which means that if an edge exists between node 1 and node 2, there is a link going from node 1 to node 2 and a link from node 2 to node 1. Additionally, the pages are indexed from 0, so the maximum page index is 5,200. Note, all of this information was provided in the dataset's README.txt.

Since the data is provided to us in a .csv file, we'll need to import two new packages, CSVFiles and DataFrames. To get a general sense of what we're dealing with, here is what the first couple of rows look like in our dataset:



	id1	id2
	Int64	Int64
1	3475	2849
2	3475	3106
3	3475	808
4	3475	4555
5	3475	3563
6	3475	1527
7	3475	3327
8	402	4066
9	402	3908

To generate the markov matrix from the csv file, let's define a function CSV-GenerateMarkov that takes in the dataset location and the number of nodes as input. We use the number of nodes to determine the size of our markov matrix. In pseudocode, it is as follows:

```
Function CSVGenerateMarkov(text, node)

    Load CSV file into a DataFrame

    Create a node x node 2d-array with all zeros

    For each row in the DataFrame

        Get the values in columns id1 and id2 (refer
    to as i, j respectively)

        In row i+1, col j+1 of the 2-d array, store
    value 1
        In row j+1, col i+1 of the 2-d array, store
    value 1

    For each column in the 2d-array

        Count the number of non-zero entries in the
    column

        If that value is not 0
            Divide the entire column by that value

    Return the 2d-array


```

Notice that we store the values at row i+1, col j+1 and row j+1, col i+1. This is because the nodes are indexed starting from 0 whereas Julia indexes starting from 1. Additionally, since the edges are undirected, we need to mark both directions of the edge. In real code, this function looks like this:

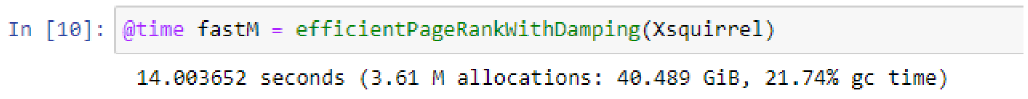
```julia
function countNonZero(arr)
    count = 0
    for elem in arr
        if(elem == 1)
            count = count + 1
        end
    end
    return count
end

function CSVGenerateMarkov(text, dim)
    df = DataFrame(load(text))
    m = zeros((dim, dim))
    for row in eachrow(df)
        m[CartesianIndex.(row.id2 + 1, row.id1 + 1)] = 1
        m[CartesianIndex.(row.id1 + 1, row.id2 + 1)] = 1
    end
    for i in 1:(dim)
        nums = countNonZero(m[:, i])
        if(nums != 0)
            m[:, i] = m[:, i] / nums
        end
    end
    return m
end
```

The function countNonZero is a helper function that counts the number of non-zero elements in an array. Now that we have our markov matrix, we can apply the page rank algorithm with damping. However, unlike the previous fixed examples, we have significantly more nodes and edges. To test the difference in efficiency, we can implement two new functions, efficientPageRankWithDamping (ePRWD) and newPageRankWithDamping (nPRWD). For both of these functions, instead of solely calculating the positive markov matrix to high powers, we also apply a starting vector X, whose values are all equal and sum to 1. The starting vector X has dimensions $n \times 1$ where n is both the number of rows and columns of the markov matrix. The purpose of this starting vector is to simulate a "starting condition" for the web pages. The ePRWD function takes in a markov matrix and a default argument, power = 50, and applies the equation derived in the efficiency section of the paper to find the stationary distribution. The nPRWD function performs the exact same calculations as the original pageRankWithDamping function except it applies the starting vector X to it. Both functions are shown below:

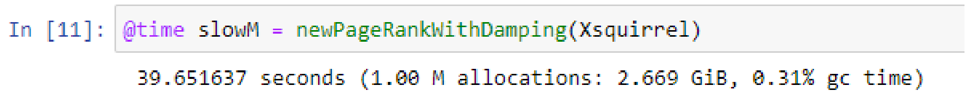
```
function efficientPageRankWithDamping(matrix, power = 50)
    if(power == 0)
        return ones((size(matrix)[1], 1)) * 1/size(matrix)[1]
    end
    p = 0.15
    B = ones(size(matrix)) * 1/size(matrix)[1]
    starting = ones((size(matrix)[1], 1)) * 1/size(matrix)[1]

    #S_{n+1} = (1 - p) * X * S_n + p * S_0

    return (1 - p) * matrix *
    efficientPageRankWithDamping(matrix, power - 1) + p
     * starting
end

function newPageRankWithDamping(matrix)
    p = 0.15
    B = ones(size(matrix)) * 1/size(matrix)[1]
    starting = ones((size(matrix)[1], 1)) * 1/size(
    matrix)[1]
    return ((1 - p) * matrix + p * B)^50 * starting
end
```

Notice how the efficientPageRankWithDamping function is a recursive function. We need to use recursion since the stationary distribution at an arbitrary step is defined recursively ($S_{n+1} = P_\beta * S_n$; $S_n = P_\beta * S_{n-1}$; ... ; $S_1 = P_\beta * S_0$). Thus, in the return statement, S_n is a recursive call with power - 1 as an argument. This runs until we hit the base case when power $= 0$, in which we return S_0.

Running both of these functions on the markov matrix generated from the squirrel dataset, we can see a substantial difference in the time spent. In the images below, Xsquirrel is the markov matrix generated from CSVGenerateMarkov.

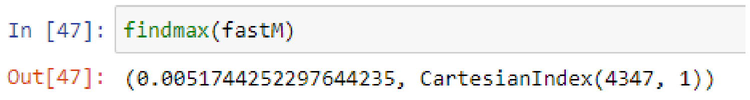


```
In [10]: @time fastM = efficientPageRankWithDamping(Xsquirrel)

          14.003652 seconds (3.61 M allocations: 40.489 GiB, 21.74% gc time)
```



```
In [11]: @time slowM = newPageRankWithDamping(Xsquirrel)

          39.651637 seconds (1.00 M allocations: 2.669 GiB, 0.31% gc time)
```

To verify our results, let's first determine the node with the highest priority. By running the function findmax, we can find the node with the highest probability and its index. Running it on the result returned from efficientPageRankWithDamping, we see that index 4347 has the highest probability.



```
In [47]: findmax(fastM)

Out[47]: (0.0051744252297644235, CartesianIndex(4347, 1))
```

To verify this, let's create a function that takes in the csv file and the number of nodes and return an array that contains the number of incoming links that each site has. The function looks like this:

```
function findMostIncoming(text, dim)
```

```
    indCount = zeros((dim + 1, 1))
    df = DataFrame(load(text))
    for row in eachrow(df)
        indCount[row.id1 + 1] += 1
        indCount[row.id2 + 1] += 1
    end
    return indCount
end
```

This function acts almost identically to CSVGenerateMarkov except we use the indCount array to keep track of the incoming links for each site. After running findMostIncoming on the squirrels dataset, we can use the findmax function on the returned array to see which node has the most incoming links. Sure enough, we see that node 4347 has the most incoming links, which aligns with our previous calculation. Thus, at step 50 (since we took the markov matrix to the 50th power), a user has the highest probability of being on page 4346 (we subtract 1 from index since Julia starts indexing at 1 but the nodes are indexed from 0).

7 Conclusion

Throughout this entire project, it was fascinating to see how all of the properties we discussed in class on markov matrices finally came into play in something we use so frequently and often take for granted. I learned about the theory behind the Page Rank algorithm, which is rooted in the properties of positive markov matrices and its eigenvectors. I also learned how the stationary distribution is ”truly” calculated in real-world examples with billions of pages and got to apply that to a real dataset and see explicit results.

Although I highlighted the most important sections of my code in the paper, I also made sure to include some extra examples in the full code base section below. There, I've added an additional example with a disjoint graph and an example with 11,000 nodes.

References

[Rou] Christiane Rousseau. *How Google Works.* URL: `http://dmuw.zum.de/images/f/f8/Google_klein_2.pdf`. (accessed: 12.09.2020).

[Tan] Raluca Tanase. *The Mathematics of Google Search.* URL: `http://pi.math.cornell.edu/~mec/Winter2009/RalucaRemus/Lecture3/lecture3.htmldf`. (accessed: 12.09.2020).

8 Full Code Base

pagerankfinal

December 10, 2020

[1]:
```
using LinearAlgebra
using JSON
using CSVFiles, DataFrames
```

[7]:
```
#Helper Function with Finding Index
function findInd(websites, websiteName)
    for i in 1:size(websites, 1)
        if websites[i] == websiteName
            return i
        end
    end
end

#Generate Markov Matrix Function for fixed example
function generateMarkovMatFixed(websites, websiteLinks)
    numWebsites = size(websites, 1)
    m = zeros((numWebsites, numWebsites))
    for i in 1:numWebsites
        linkedSites = websiteLinks[i]
        for j in 1:size(linkedSites, 1)
            m[CartesianIndex.(findInd(websites, linkedSites[j]), i)] = 1/
 ↪size(linkedSites, 1)
        end
    end
    return m
end
```

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

[23]:
```
#returns array of results where priority is in increasing order (1 is highest␣
 ↪priority)
function result(websites, markov)
    pi = markov[:, 1]
    results = zeros((size(websites, 1), 1))
    sort = sortperm(pi)
    for i in 1:size(websites, 1)
        results[sort[i]] = size(websites, 1) - i + 1
    end
    return results
end
```

[23]:
```
result (generic function with 1 method)
```

[33]:
```
#page rank w/damping
function pageRankWithDamping(matrix)
    p = 0.15
    B = ones(size(matrix)) * 1/size(matrix)[1]
    return ((1 - p) * matrix + p * B)^100
end
```

[33]:
```
pageRankWithDamping (generic function with 1 method)
```

0.1 Connected Graph

[5]:
```
#Instantiate input data

websites = ["google.com", "diderot.com", "discord.com", "youtube.com", "gmail.
 ↪com", "zoom.com"]

websiteLinks= Dict(1=>["diderot.com", "youtube.com", "gmail.com", "zoom.com"], ␣
 ↪#google.com
                   2=>["google.com", "gmail.com", "zoom.com"],  #diderot.com
                   3=>["google.com", "youtube.com", "diderot.com"],  #discord.
 ↪com
                   4=>["gmail.com", "google.com"], #youtube.com
                   5=>["google.com", "diderot.com", "zoom.com", "discord.com"],␣
 ↪#gmail.com
                   6=>["youtube.com", "google.com", "gmail.com"]) #zoom.com

# dicts may be looped through using the keys function:
for k in sort(collect(keys(websiteLinks)))
    print(websites[k], "'s links: ", websiteLinks[k], "\n")
end
println()
```

```
google.com's links: ["diderot.com", "youtube.com", "gmail.com", "zoom.com"]
diderot.com's links: ["google.com", "gmail.com", "zoom.com"]
discord.com's links: ["google.com", "youtube.com", "diderot.com"]
youtube.com's links: ["gmail.com", "google.com"]
gmail.com's links: ["google.com", "diderot.com", "zoom.com", "discord.com"]
zoom.com's links: ["youtube.com", "google.com", "gmail.com"]
```

[8]:
```
X = generateMarkovMatFixed(websites, websiteLinks)
```

[8]:
```
6×6 Array{Float64,2}:
 0.0   0.333333  0.333333  0.5  0.25  0.333333
 0.25  0.0       0.333333  0.0  0.25  0.0
 0.0   0.0       0.0       0.0  0.25  0.0
 0.25  0.0       0.333333  0.0  0.0   0.333333
 0.25  0.333333  0.0       0.5  0.0   0.333333
 0.25  0.333333  0.0       0.0  0.25  0.0
```

[35]:
```
M = pageRankWithDamping(X)
```

[35]:
```
6×6 Array{Float64,2}:
 0.243715   0.243715   0.243715   0.243715   0.243715   0.243715
 0.145674   0.145674   0.145674   0.145674   0.145674   0.145674
 0.0731568  0.0731568  0.0731568  0.0731568  0.0731568  0.0731568
 0.144613   0.144613   0.144613   0.144613   0.144613   0.144613
 0.22662    0.22662    0.22662    0.22662    0.22662    0.22662
 0.166221   0.166221   0.166221   0.166221   0.166221   0.166221
```

[39]:
```
#Confirm the columns of M are eigenvectors with eigenvalue 1 of S
#S is the P_{\beta} pos. markov matrix without taking it to a high power

B = ones(size(X)) * 1/size(X)[1]
S = ((1 - 0.15) * X + 0.15 * B)

isapprox(S * M[:, 1], M[:, 1])
```

[39]:
```
true
```

[25]:
```
result(websites, M)
```

[25]:
```
6×1 Array{Float64,2}:
 1.0
 4.0
 6.0
 5.0
 2.0
 3.0
```

0.2 Disconnected Graph

[40]:
```
websitesDisjoint = ["google.com", "diderot.com", "discord.com", "youtube.com",␣
 ↪"gmail.com", "zoom.com"]

websiteLinksDisjoint= Dict(1=>["youtube.com", "gmail.com", "zoom.com"], ␣
 ↪#google.com
                      2=>["discord.com"],  #diderot.com
                      3=>["diderot.com"],  #discord.com
```

```
                        4=>["gmail.com", "google.com"], #youtube.com
                        5=>["google.com", "zoom.com"], #gmail.com
                        6=>["youtube.com", "gmail.com"]) #zoom.com

# dicts may be looped through using the keys function:
for k in sort(collect(keys(websiteLinksDisjoint)))
    print(websitesDisjoint[k], "'s links: ", websiteLinksDisjoint[k], "\n")
end
println()
```

```
google.com's links: ["youtube.com", "gmail.com", "zoom.com"]
diderot.com's links: ["discord.com"]
discord.com's links: ["diderot.com"]
youtube.com's links: ["gmail.com", "google.com"]
gmail.com's links: ["google.com", "zoom.com"]
zoom.com's links: ["youtube.com", "gmail.com"]
```

[41]:
```
Xprime = generateMarkovMatFixed(websitesDisjoint, websiteLinksDisjoint)
```

[41]:
```
6×6 Array{Float64,2}:
 0.0       0.0  0.0  0.5  0.5  0.0
 0.0       0.0  1.0  0.0  0.0  0.0
 0.0       1.0  0.0  0.0  0.0  0.0
 0.333333  0.0  0.0  0.0  0.0  0.5
 0.333333  0.0  0.0  0.5  0.0  0.5
 0.333333  0.0  0.0  0.0  0.5  0.0
```

[42]:
```
Mprime = pageRankWithDamping(Xprime)
```

[42]:
```
6×6 Array{Float64,2}:
 0.169318  0.169318  0.169318  0.169318  0.169318  0.169318
 0.166667  0.166667  0.166667  0.166667  0.166667  0.166667
 0.166667  0.166667  0.166667  0.166667  0.166667  0.166667
 0.140029  0.140029  0.140029  0.140029  0.140029  0.140029
 0.199542  0.199542  0.199542  0.199542  0.199542  0.199542
 0.157778  0.157778  0.157778  0.157778  0.157778  0.157778
```

[46]:
```
#Confirm the columns of Mprime are eigenvectors with eigenvalue 1 of Sprime
#Sprime is the P_{\beta} pos. markov matrix without taking it to a high power

Bprime = ones(size(Xprime)) * 1/size(Xprime)[1]
Sprime = ((1 - 0.15) * Xprime + 0.15 * Bprime)

isapprox(Sprime * Mprime[:, 1], Mprime[:, 1], atol=1e-5)
```

[46]:
```
true
```

[46]:
```
result(websitesDisjoint, Mprime)
```

[46]:
```
6×1 Array{Float64,2}:
 2.0
 4.0
 3.0
 6.0
 1.0
 5.0
```

0.3 Real Dataset Example

[24]:
```
#musae_squirrel_edges.csv
#dataset of page-page networks on specific topics (this dataset is on squirrels)
#Has 5201 nodes (or webpages) and 198,493 edges (undirected links between pages)
#pages indexed from 0, so max page index is 5200

DataFrame(load("musae_squirrel_edges.csv"))
```

[24]:

```
    │ id1    id2
    │ Int64  Int64
────┼─────────────
  1 │ 3475   2849
  2 │ 3475   3106
  3 │ 3475    808
  4 │ 3475   4555
  5 │ 3475   3563
  6 │ 3475   1527
  7 │ 3475   3327
  8 │  402   4066
  9 │  402   3908
 10 │  402   2820
 11 │  402   4903
 12 │  402    715
 13 │  402   5112
 14 │  817   3563
 15 │  817   4855
 16 │ 5057   4481
 17 │ 5057   3975
 18 │ 5057   2408
 19 │ 5057   2539
 20 │ 5057   4236
 21 │ 5057   3372
 22 │ 5057   4303
 23 │ 5057   4658
 24 │ 5057   3829
 25 │ 5057   5078
 26 │ 5057    313
 27 │ 5057   4410
 28 │  777   3609
 29 │  777   5122
 30 │  777   4650
... │  ...    ...
```

[7]:
```
function countNonZero(arr)
    count = 0
    for elem in arr
        if(elem == 1)
            count = count + 1
        end
    end
    return count
end

function CSVGenerateMarkov(text, dim)
    df = DataFrame(load(text))
    m = zeros((dim, dim))
    for row in eachrow(df)
        m[CartesianIndex.(row.id2 + 1, row.id1 + 1)] = 1
        m[CartesianIndex.(row.id1 + 1, row.id2 + 1)] = 1
    end
    for i in 1:(dim)
        nums = countNonZero(m[:, i])
        if(nums != 0)
            m[:, i] = m[:, i] / nums
        end
    end
    return m
end
```

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

[8]:
```
Xsquirrel = CSVGenerateMarkov("musae_squirrel_edges.csv", 5201)
```

[8]:
```
5201×5201 Array{Float64,2}:
 0.0         0.0  0.0        0.0         0.0  …  0.0       0.0        0.0
 0.0         0.0  0.0        0.0         0.0     0.0       0.0        0.0
 0.0         0.0  0.0        0.0         0.0     0.0       0.0        0.0
 0.0         0.0  0.0        0.0         0.0     0.0       0.0        0.0
 0.0         0.0  0.0        0.0         0.0     0.0       0.0        0.0
 0.0         0.0  0.0        0.0         0.0  …  0.0       0.0        0.0
 0.0         0.0  0.0        0.0         0.0     0.0       0.0        0.0
 0.0         0.0  0.0        0.0         0.0     0.0       0.0        0.0
 0.0         0.0  0.0        0.0         0.0     0.0       0.0        0.0
 0.0         0.0  0.0        0.0         0.0     0.0       0.0        0.0
 0.0         0.0  0.0        0.0         0.0  …  0.0       0.0        0.0
 0.0         0.0  0.0        0.0         0.0     0.0       0.0        0.0
 0.0         0.0  0.0        0.0         0.0     0.0       0.0        0.0
 ⋮                                            ⋱
 0.00649351  0.0  0.0        0.00666667  0.0     0.037037  0.0        0.0
 0.0         0.0  0.0        0.0         0.0  …  0.0       0.0        0.0
 0.0         0.0  0.0        0.0         0.0     0.0       0.0        0.0
 0.0         0.0  0.0        0.0         0.0     0.0       0.0        0.0
 0.00649351  0.0  0.0        0.00666667  0.0     0.0       0.0243902  0.0
 0.0         0.0  0.0        0.0         0.0     0.0       0.0        0.0
 0.0         0.0  0.0        0.0         0.0  …  0.0       0.0        0.0
 0.00649351  0.0  0.0769231  0.00666667  0.0     0.037037  0.0        0.0
 0.0         0.0  0.0        0.0         0.0     0.0       0.0        0.0
 0.0         0.0  0.0        0.0         0.0     0.0       0.0        0.0
 0.0         0.0  0.0        0.0         0.0     0.0       0.0        0.0
 0.0         0.0  0.0        0.0         0.0  …  0.0       0.0        0.0
```

[9]:
```
function efficientPageRankWithDamping(matrix, power = 50)
    if(power == 0)
```

```
        return ones((size(matrix)[1], 1)) * 1/size(matrix)[1]
    end
    p = 0.15
    B = ones(size(matrix)) * 1/size(matrix)[1]
    starting = ones((size(matrix)[1], 1)) * 1/size(matrix)[1]
    return (1 - p) * matrix * efficientPageRankWithDamping(matrix, power - 1) +␣
 ↪p * starting
end
function newPageRankWithDamping(matrix)
    p = 0.15
    B = ones(size(matrix)) * 1/size(matrix)[1]
    starting = ones((size(matrix)[1], 1)) * 1/size(matrix)[1]
    return ((1 - p) * matrix + p * B)^50 * starting
end
```

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

```
[10]: @time fastM = efficientPageRankWithDamping(Xsquirrel)
```

```
 14.003652 seconds (3.61 M allocations: 40.489 GiB, 21.74% gc time)
```

```
[10]: 5201×1 Array{Float64,2}:
 0.00024764341635520143
 5.883654437119404e-5
 4.7634285996364285e-5
 0.0002175872621968898
 6.446739511168494e-5
 5.502522689030297e-5
 5.29844303208086e-5
 0.00018087817983672888
 4.438339815942886e-5
 6.903322853840814e-5
 0.000178148819967327
 6.476298196547737e-5
 9.122145487608214e-5

 0.0016080909904889744
 0.0002211667874591535
 0.00013855744053936005
 5.70002440104511e-5
 0.0015538421934595196
 0.002328658664354606
 6.231179981845357e-5
 0.0016028765469088397
 0.00031035954421267304
 8.840909946418807e-5
 0.00013517026116400738
 0.00014034259080621187
```

[11]:
```
@time slowM = newPageRankWithDamping(Xsquirrel)
```

```
 39.651637 seconds (1.00 M allocations: 2.669 GiB, 0.31% gc time)
```

[11]:
```
5201×1 Array{Float64,2}:
 0.00024764341635520143
 5.8836544371193824e-5
 4.763428599636419e-5
 0.00021758726219688987
 6.446739511168476e-5
 5.502522689030285e-5
 5.298443032080849e-5
 0.00018087817983672896
 4.438339815942876e-5
 6.903322853840803e-5
 0.00017814881996732707
 6.476298196547715e-5
 9.122145487608195e-5
 ⋮
 0.0016080909904889742
 0.0002211667874591536
 0.0001385574405393595
 5.7000244010450954e-5
 0.0015538421934595192
 0.002328658664354599
 6.231179981845338e-5
 0.0016028765469088399
 0.00031035954421267315
 8.840909946418789e-5
 0.00013517026116400703
 0.00014034259080621133
```

[12]:
```
#both stationary distribution vectors are the same
isapprox(slowM, fastM)
```

[12]:
```
true
```

[47]:
```
findmax(fastM)
```

[47]:
```
(0.005174425229764423 5, CartesianIndex(4347, 1))
```

[15]:
```
function findMostIncoming(text, dim)
    indCount = zeros((dim + 1, 1))
    df = DataFrame(load(text))
    for row in eachrow(df)
        indCount[row.id1 + 1] += 1
        indCount[row.id2 + 1] += 1
    end
    return indCount
end
```

```
[15]: findMostIncoming (generic function with 1 method)
```

```
[16]: incomeNum = findMostIncoming("musae_squirrel_edges.csv", 5201)
```

```
[16]: 5202×1 Array{Float64,2}:
        154.0
          6.0
         14.0
        150.0
          5.0
         15.0
         12.0
        117.0
          1.0
         21.0
        115.0
          3.0
         31.0

        154.0
         21.0
         17.0
       1120.0
        198.0
         10.0
       1171.0
        240.0
         29.0
         45.0
          7.0
          0.0
```

```
[18]: findmax(incomeNum)
```

```
[18]: (2087.0, CartesianIndex(4347, 1))
```

0.4 Additional Dataset to show difference in efficiency

```
[42]: #same as squirrel dataset, except has 11,631 nodes and 170,918 edges
      Xcrocodile = CSVGenerateMarkov("musae_crocodile_edges.csv", 11631)
```

```
[42]: 11631×11631 Array{Float64,2}:
       0.0  0.0  0.0  0.0        0.0  0.0  …  0.0  0.0          0.0         0.0
       0.0  0.0  0.0  0.0        0.0  0.0     0.0  0.0          0.0         0.0
```

```
 0.0  0.0  0.0  0.0        0.0  0.0     0.0  0.0          0.0        0.0
 0.0  0.0  0.0  0.0        0.0  0.0     0.0  0.000636943  0.0        0.0
 0.0  0.0  0.0  0.0        0.0  0.0     0.0  0.0          0.0        0.0
 0.0  0.0  0.0  0.0        0.0  0.0  …  0.0  0.0          0.0        0.0
 0.0  0.0  0.0  0.0        0.0  0.0     0.0  0.000636943  0.0        0.0
 0.0  0.0  0.0  0.0        0.0  0.0     0.0  0.000636943  0.0        0.0
 0.0  0.0  0.0  0.0        0.0  0.0     0.0  0.0          0.0        0.0
 0.0  0.0  0.0  0.0        0.0  0.0     0.0  0.0          0.0        0.0
 0.0  0.0  0.0  0.0        0.0  0.0  …  0.0  0.0          0.0        0.0
 0.0  0.0  0.0  0.0        0.0  0.0     0.0  0.0          0.0        0.0
 0.0  0.0  0.0  0.0        0.0  0.0     0.0  0.0          0.0        0.0

 0.0  0.0  0.0  0.0        0.0  0.0     0.0  0.0          0.0        0.0
 0.0  0.0  0.0  0.0        0.0  0.0  …  0.0  0.0          0.0        0.0
 0.0  0.0  0.0  0.0        0.0  0.0     0.0  0.0          0.0        0.0
 0.0  0.0  0.0  0.0        0.0  0.0     0.0  0.0          0.0        0.0
 0.0  0.0  0.0  0.0        0.0  0.0     0.0  0.0          0.0        0.0
 0.0  0.0  0.0  0.0        0.0  0.0     0.0  0.0          0.0        0.0
 0.0  0.0  0.0  0.0        0.0  0.0  …  0.0  0.0          0.0        0.0
 0.0  0.0  0.0  0.0        0.0  0.0     0.0  0.0          0.0        0.0
 0.0  0.0  0.0  0.0        0.0  0.0     0.0  0.0          0.0        0.0
 0.0  0.0  0.0  0.0333333  0.0  0.0     0.0  0.0          0.0        0.0
 0.0  0.0  0.0  0.0        0.0  0.0     0.0  0.0          0.0217391  0.0
 0.0  0.0  0.0  0.0        0.0  0.0  …  0.0  0.0          0.0        0.0
```

```
[63]: @time fastMCroc = efficientPageRankWithDamping(Xcrocodile)
```

```
62.076384 seconds (1.01 k allocations: 201.609 GiB, 24.20% gc time)
```

```
[63]: 11631×1 Array{Float64,2}:
 8.458833903287631e-5
 1.6652475832516783e-5
 0.00019145935419371507
 6.309512935357827e-5
 2.6358753809455273e-5
 1.7869280451223095e-5
 6.309512935357827e-5
 6.309512935357827e-5
 4.3585351474861785e-5
 1.868684885796903e-5
 0.00010031813437503532
 6.621021877187009e-5
 0.0001206039472007654

 0.00011088996563726 85
 8.733521063072639e-5
 7.951975571943295e-5
 5.190180604517787e-5
 0.0008983087876972979
 0.0006439430196361107
 0.00013716483086460707
 4.8871367833648306e-5
 0.0005622327402725814
 0.0029585873984982487
 0.0005973759992581139
 0.00016542783140467032
```

[50]:
```
@time slowMCroc = newPageRankWithDamping(Xcrocodile)
```

```
443.405510 seconds (34 allocations: 13.103 GiB, 0.15% gc time)
```

[50]:
```
11631×1 Array{Float64,2}:
 8.458833903287514e-5
 1.665247583251655e-5
 0.0001914593541937125
 6.309512935357727e-5
 2.63587538094549e-5
 1.7869280451222857e-5
 6.309512935357727e-5
 6.309512935357727e-5
 4.358535147486121e-5
 1.868684885796878e-5
 0.00010031813437503455
 6.621021877186913e-5
 0.00012060394720076365
 ⋮
 0.00011088996563726703
 8.733521063072539e-5
 7.951975571943181e-5
 5.190180604517715e-5
 0.0008983087876972857
 0.0006439430196361013
 0.0001371648308646053
 4.8871367833647635e-5
 0.0005622327402725732
 0.002958587398498203
 0.0005973759992581066
 0.00016542783140466845
```