21-241 Final Project (Topic 2)

Sajan Shah & Thomas Song

3 Dec. 2021

Introduction

No matter what academic background, you have probably wondered how a system as large and intricate as the internet works. It is amazing how our browsers shift through billions of websites on the internet and returns ordered lists based on relevance to our search, within fractions of a second. Although it is something that we witness multiple times on a daily basis, mountainous amounts of data are being processed before our eyes. This is actually one of the many powerful applications of Linear Algebra and CS, and we decided to explore its mechanisms through more simple models that we created.

Mechanism Overview

The sorting process is very similar to the way a Markov Chain works.

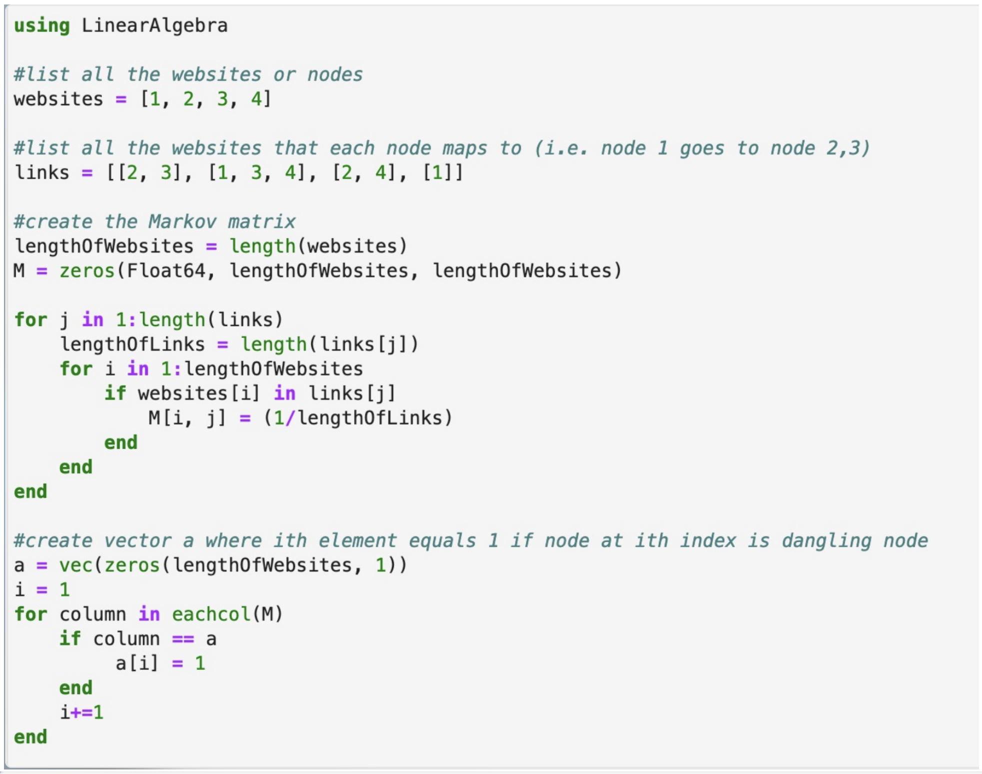
Consider a group of interconnected websites related to a search input. Each website can lead out to several other websites, and they will have several other ones leading to themselves. The group of websites leading outwards and inwards are not necessarily equivalent; in this way, we can use a directed graph to represent the websites as nodes and the contained links as the arrows leading from one node to another.

The goal here is to determine the order of relevance of the websites. One way to judge relevance is the likeliness to end up on any of the nodes on the graph. In this "directed random walk" a certain walker has several options with different probabilities to go from one node to another during a step. Using these recursions, it is possible to calculate the specific probabilities of ending up on each node after a substantial amount of iterations. This is exactly what we do with Markov Chains.

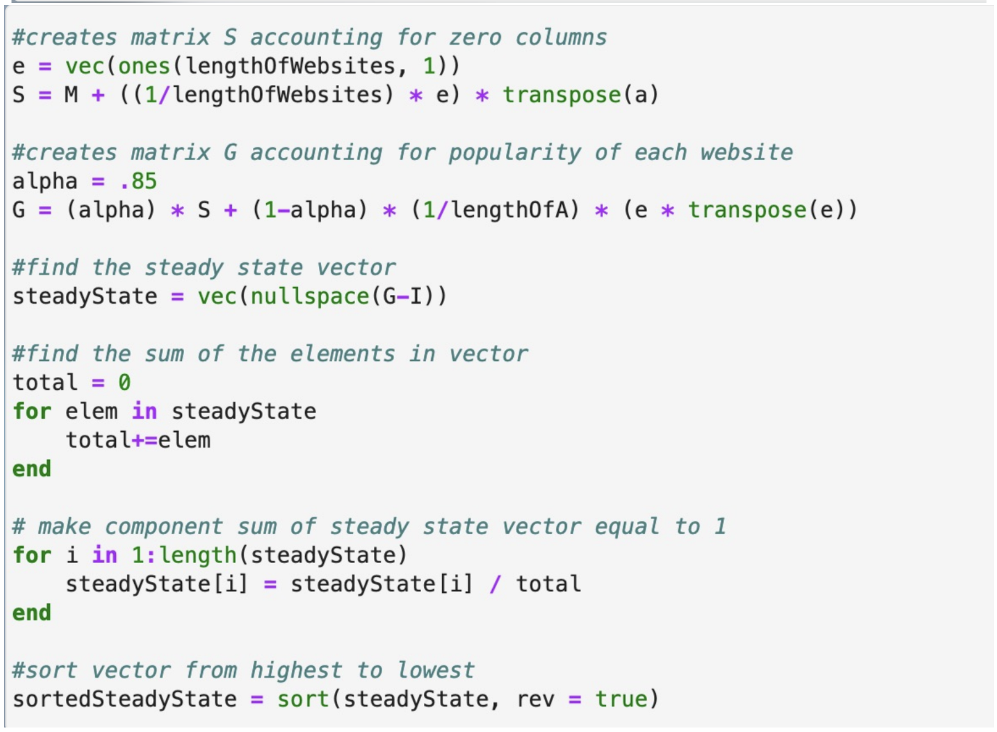
Once the probabilities are determined, they will correspond to the order of display. The more likely it is to end up on a website, the higher position it will occupy on the list.

Algorithm Process

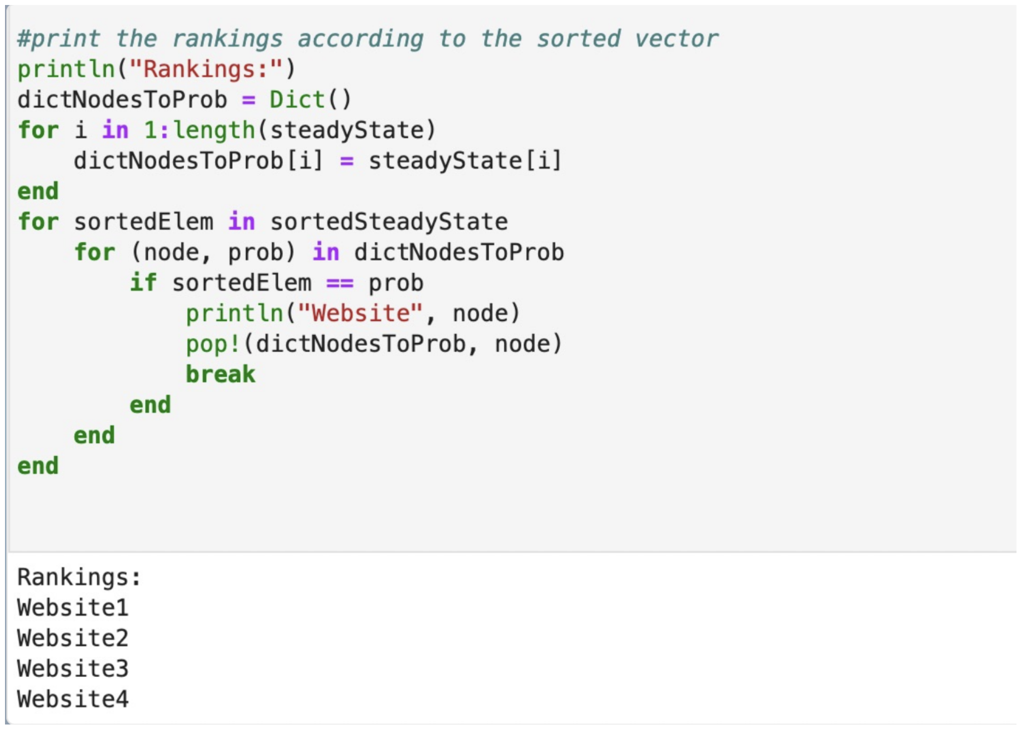
Here is the code we used.



```
using LinearAlgebra

#list all the websites or nodes
websites = [1, 2, 3, 4]

#list all the websites that each node maps to (i.e. node 1 goes to node 2,3)
links = [[2, 3], [1, 3, 4], [2, 4], [1]]

#create the Markov matrix
lengthOfWebsites = length(websites)
M = zeros(Float64, lengthOfWebsites, lengthOfWebsites)

for j in 1:length(links)
    lengthOfLinks = length(links[j])
    for i in 1:lengthOfWebsites
        if websites[i] in links[j]
            M[i, j] = (1/lengthOfLinks)
        end
    end
end

#create vector a where ith element equals 1 if node at ith index is dangling node
a = vec(zeros(lengthOfWebsites, 1))
i = 1
for column in eachcol(M)
    if column == a
       a[i] = 1
    end
    i+=1
end
```



```
#creates matrix S accounting for zero columns
e = vec(ones(lengthOfWebsites, 1))
S = M + ((1/lengthOfWebsites) * e) * transpose(a)

#creates matrix G accounting for popularity of each website
alpha = .85
G = (alpha) * S + (1-alpha) * (1/lengthOfA) * (e * transpose(e))

#find the steady state vector
steadyState = vec(nullspace(G-I))

#find the sum of the elements in vector
total = 0
for elem in steadyState
    total+=elem
end

# make component sum of steady state vector equal to 1
for i in 1:length(steadyState)
    steadyState[i] = steadyState[i] / total
end

#sort vector from highest to lowest
sortedSteadyState = sort(steadyState, rev = true)
```

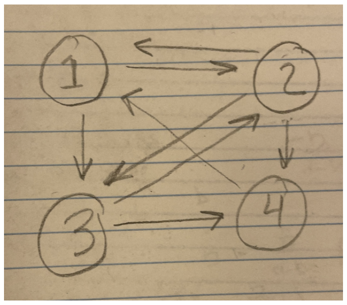


```
#print the rankings according to the sorted vector
println("Rankings:")
dictNodesToProb = Dict()
for i in 1:length(steadyState)
    dictNodesToProb[i] = steadyState[i]
end
for sortedElem in sortedSteadyState
    for (node, prob) in dictNodesToProb
        if sortedElem == prob
            println("Website", node)
            pop!(dictNodesToProb, node)
            break
        end
    end
end
```

```
Rankings:
Website1
Website2
Website3
Website4
```

The algorithm starts by taking the inputs to create a graph, which models the network of websites. It first registers the nodes as numbers, and takes inputs for the destination of edges that lead out from each node.

From this data, the program creates a square matrix to represent the recursive probabilities of going from one node to another. The k^{th} column represents the directed edges moving out of the k^{th} node; the value A_{jk} is the probability a walker will go from the k^{th} node to the j^{th} node. If there is no edge connecting the nodes, the value will be zero. Otherwise, the program takes the number of edges directed out of a node and gets its reciprocal, which becomes the probability. Below is the graph we used.



As a result, the k^{th} column represents the probabilities directed outwards, and the k^{th} row represents the probabilities directed inward to the node from other nodes. It is very similar to a Markov matrix, except for the fact that we can have zero-columns. This is because all of the edges connected to a node may be directed outwards from it - these are called sinks. To solve this problem, the algorithm locates these columns and replaces the zeroes with $\frac{1}{n}$ as seen below.

```
e = vec(ones(lengthOfA, 1))
S = M + ((1/lengthOfA) * e) * transpose(a)
```

Another difference from a naive random walk that has to be accounted for is that the average web-surfer's behavior is prone to distractions from other web pages. There is always a given probability that the surfer will not go down the given network of interconnected links, but jump to a different web page.

The PageRank algorithm accommodates for these differences through a diffusion constant, α. $0 < \alpha < 1$ is the probability that the user jumps to another unrelated web page, and $1 - \alpha$ is the probability that the user moves to a linked web page in the site they are using.

Thus, instead of the usual recursion used for Markov chains

$$r^{i+1} = Sr^i$$

we are seeking a modified equation

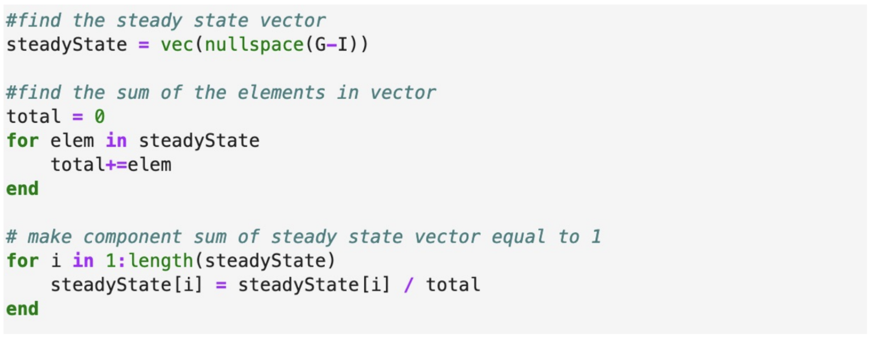
$$r^{i+1} = \alpha(Sr^i) + \frac{1-\alpha}{n}$$

where r is the vector for the ranks of nodes, S is the matrix indicating directed edges (modified to accommodate zero-columns), α is the diffusion constant, and n is the number of nodes.

In terms of what the algorithm does to the matrix, the matrix S is multiplied by the constant α and the identity matrix I multiplied by the constant $\frac{1-\alpha}{n}$ is added. This can be seen in the code segment below.

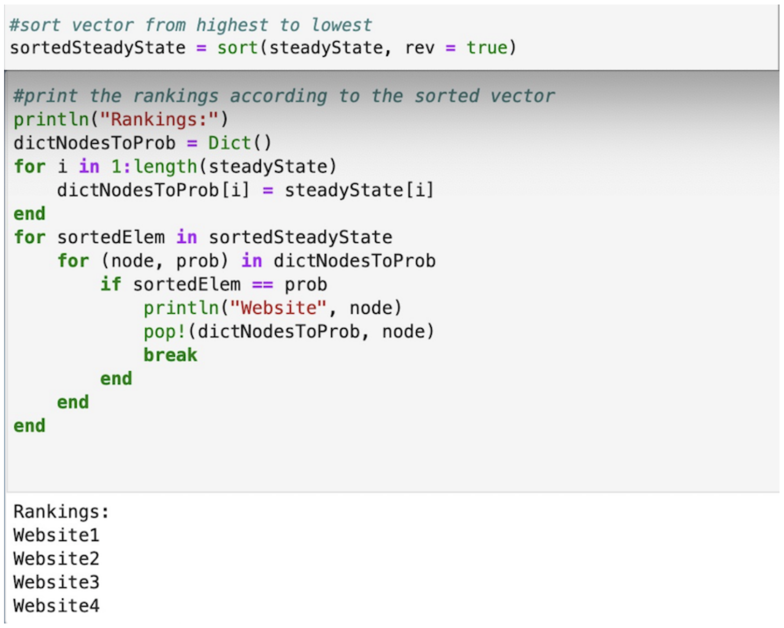
```
alpha = .85
G = (alpha) * S + (1-alpha) * (1/lengthOfA) * (e * transpose(e))
```

Now we have a viable matrix that is closely representative of the real-life situation. From here on out, the process is identical to the Markov Chain. The algorithm uses the linear algebra program to obtain the eigen-vector of G that corresponds to the eigen-value $\lambda = 1$. This is the steady state vector that indicates the relevance of each website. The eigen-vector is finally scaled so that the entries add up to 1, in order to have it display the probabilities that a walker would end up on each website.



```
#find the steady state vector
steadyState = vec(nullspace(G-I))

#find the sum of the elements in vector
total = 0
for elem in steadyState
    total+=elem
end

# make component sum of steady state vector equal to 1
for i in 1:length(steadyState)
    steadyState[i] = steadyState[i] / total
end
```

Finally, the probabilities corresponding to each node are rearranged in descending order so that the most relevant (high probability) website gets rank 1, the next gets rank 2, and so on. The rearranged results are printed.



```
#sort vector from highest to lowest
sortedSteadyState = sort(steadyState, rev = true)
```

```
#print the rankings according to the sorted vector
println("Rankings:")
dictNodesToProb = Dict()
for i in 1:length(steadyState)
    dictNodesToProb[i] = steadyState[i]
end
for sortedElem in sortedSteadyState
    for (node, prob) in dictNodesToProb
        if sortedElem == prob
            println("Website", node)
            pop!(dictNodesToProb, node)
            break
        end
    end
end
```

```
Rankings:
Website1
Website2
Website3
Website4
```

Extension - Very Large Networks

The problem with the displayed algorithm is that when matrices become to large, the run-time for computing the eigen-vector is too long. Considering that the internet will require billions of nodes, it would be impossible for our browsers to return results.

Thus, instead of the method shown above, iterations are used for very large networks to cut down the time. This is done using the recursion mentioned above.

$$r^{i+1} = \alpha(Sr^i) + \frac{1-\alpha}{n}$$

It was estimated that 45 iterations returned accurate results for a network consisting of approximately 322 million links. Using this recursive system, the time also gets cut down a lot for capacities exceeding the billions. Specifically, the time function is roughly $\log n$.

While iterations will not be accurate as the method of obtaining the steady-state vector, they are much more efficient for large networks.

Sajan Shah and Thomas Song
Julia Code:

```
using LinearAlgebra

#list all the websites or nodes
websites = [1, 2, 3, 4]

#list all the websites that each node maps to (i.e. node 1 goes to node 2,3)
links = [[2, 3], [1, 3, 4], [2, 4], [1]]

#create the Markov matrix
lengthOfWebsites = length(websites)
M = zeros(Float64, lengthOfWebsites, lengthOfWebsites)

for j in 1:length(links)
    lengthOfLinks = length(links[j])
    for i in 1:lengthOfWebsites
        if websites[i] in links[j]
            M[i, j] = (1/lengthOfLinks)
        end
    end
end

#create vector a where ith element equals 1 if node at ith index is dangling node
a = vec(zeros(lengthOfWebsites, 1))
i = 1
for column in eachcol(M)
    if column == a
         a[i] = 1
    end
    i+=1
end

#creates matrix S accounting for zero columns
e = vec(ones(lengthOfWebsites, 1))
S = M + ((1/lengthOfWebsites) * e) * transpose(a)

#creates matrix G accounting for popularity of each website
alpha = .85
G = (alpha) * S + (1-alpha) * (1/lengthOfA) * (e * transpose(e))

#find the steady state vector
steadyState = vec(nullspace(G-I))

#find the sum of the elements in vector
```

```
total = 0
for elem in steadyState
    total+=elem
end

# make component sum of steady state vector equal to 1
for i in 1:length(steadyState)
    steadyState[i] = steadyState[i] / total
end

#sort vector from highest to lowest
sortedSteadyState = sort(steadyState, rev = true)

#print the rankings according to the sorted vector
println("Rankings:")
dictNodesToProb = Dict()
for i in 1:length(steadyState)
    dictNodesToProb[i] = steadyState[i]
end
for sortedElem in sortedSteadyState
    for (node, prob) in dictNodesToProb
        if sortedElem == prob
            println("Website", node)
            pop!(dictNodesToProb, node)
            break
        end
    end
end
```