American University of Armenia

College of Science and Engineering

Numerical Analysis Project

Google PageRank Algorithm

Team: Arusyak Hakobyan , Vahan Melkonyan,
Vahagn Khachatryan

Fall, 2017

Chapter 1
Introduction

In this paper we discuss the three most popular approaches for solving the Google PageRank problem. We start by giving the initial probabilistic algorithm by Page and Brin. Then we introduce the simple dynamical systems representation of PageRank, which has some drawbacks. In the final part we turn to Linear Analysis to adjust and improve the previously mentioned algorithm and obtain the final version of the algorithm.

1.1 Origins of Google

When we think of Google now, we immediately picture a multibillion company that offers highest quality technology in many industries and is one of the global leaders in terms of development of new technology. Although, like most companies Google was created by two guys in their garage. Of course, in this case said *garage* is in the scope of the Stanford University, where Sergey Brin and Larry Page had a thesis on a new revolutionary web search algorithm that could beat anything similar in the market at the time in terms of scalability, efficiency and relevance of the search results.
In their research paper “The Anatomy of a Large-Scale Hypertextual Web Search Engine”(1) Brin and Page first introduce the world to Google and their PageRank algorithm. A year later Rajeev Motwani and Terry Winograd joined Brin and Page to publish “The PageRank Citation Ranking: Bringing Order to the Web”(2). These were the first steps made towards the technology giant Google is today.

1.2 PageRank

Firstly, a little bit of backstory. Page and Brin, when writing their Stanford paper about Google, back in 1998 made sure to stress the fact that for the web of the late 90's web searching was terribly inefficient with "only one out of four web search giants being able to find themselves in the web." That said, all four were still major public companies and we can imagine Brin and Page knew what Google could become if they managed to beat the existing web search methods.

And they did just that. At the time, there were 2 major resources that went almost unused in the search engines of the time: the web's link-structure and the anchor-texts. Google was designed to use those two features to its advantage, as they provided a very intuitive method of ranking the importance of a certain page.

Fun fact: The PageRank algorithm is named after Larry Page himself.

1.2.1 The Web's Link Structure

The first and major ace in Google's pocket is that it utilized **the web's link-structure** to intuitively make it so its search results were more relevant and the *junk* pages would go unnoticed.

What is a **link-structure**? Generally, pages in the web have *links* to pages and are *linked* by other pages in return, thus creating a massive link-based vastly interconnected infrastructure. Page and Brin's idea was to utilize this infrastructure as an intuitive filtering tool to determine the relevance of a certain page.

1.2.2 Anchor Text

The second web feature utilized by Google was **anchor text**, which is the text usually accompanying a link to another page. **Anchor texts** are useful, firstly because they can be associated and give information about pages they link to, even before visiting those pages. And, according to Brin and Page the information provided by **anchor texts** even better describes the page it links to than the page itself. Secondly, anchor pages exist

for not only text-based pages, but also other types of documents like images and databases.

Chapter 2
Probabilistic Method

2.1 Original Algorithm

The Google Pagerank algorithm is built upon the web's link-structure utilizing information such as the amount of *link* to the page and the amount of *link* the page links to. It is important to grasp that PageRank is not the mechanism retrieving results for a search, it is the algorithm figuring out the relevance of the retrieved pages and the one *ranking* them by importance.

To understand **PageRank** probabilistically, Page and Brin bring the example of a random internet surfer. "We assume there is a "random surfer" who is given a web page at random and keeps clicking on links, never hitting "back" but eventually gets bored and starts on another random page. The probability that the random surfer visits a page is its **PageRank**.**"** (Brin & Page, 1998)

A page's **PageRank** is directly correlated to the pages' ranks that link to that page divided by the amount of pages each of those *linker* pages links to. We can think of it this way, pages that are linked by a lot of other pages will most probably be important and if a search matches with that page, it would most probably be of relevance. Also, any page linked by that relevant page should also be relevant for any search that finds that page. For example, a lot of pages link to the CNN website, and the CNN website doesn't link to many pages outside of its domain. This will result in whatever page that gets linked from the CNN website's home page would get a large margin of its *rank*.

This is exactly why, in SEO (Search Engine Optimization), it is crucial for your website to get linked from different high rank holding websites like giant journals, blogs and magazines, as they will transfer a good chunk of their already massive rank score to your website, thus making it more *relevant* for Google and other search engines, that are utilizing some sort of **PageRank** in their algorithms.

Knowing this, the probabilistic calculation of a page's *rank* will look like this:

$$PR(A) = (1-d) + d\ (PR(P1)/C(P1) + ... + PR(Pn)/C(Pn))$$

Where PR(Pn) are the *ranks* of all the pages that link to our page, C(Pn) are the numbers of links the page has overall and the d is the **damping factor**.

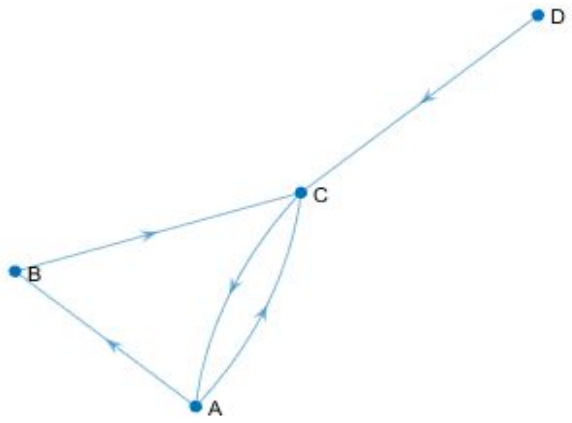
2.1.1 The Damping Factor

In the *random surfer* example, the **damping factor** d is a number between 0 and 1 (usually 0.85). It illustrates the probability at each page, that the surfer will get bored and switch to a whole another page (start all over again). Thus, the pageranks of all linking pages will only affect the pagerank of our page for a 0.85 and the (d-1) here illustrates the probability of the **random surfer** getting bored and switching to another random page.

As **PageRank** is an iterative algorithm, just one iteration of the equation for every page, won't give us a good result, in fact, according to Brin for the vast web, at least hundred **PageRank** iterations are needed to get a decent result. However, the amount of iterations of the algorithm won't affect the overall sum (more like the average) of our pages' ranks as that number will generally converge to one (all ranks of all pages divided by the number of pages will converge to one).(3)

2.2 Examples

2.2.1 Example 1: 4 Nodes 1 Sink



Example 1

```
d = 0.85;                    Setting the damping factor

rankA = 1;                   Setting the base values
rankB = 1;
rankC = 1;
rankD = 1;

for k = 1:10                 Choose the number of iterations

[ rankAnew, rankBnew, rankCnew, rankDnew ] = PageRank (G, d, rankA, rankB, rankC,
rankD);

rankA = rankAnew;
rankB = rankBnew;
rankC = rankCnew;
rankD = rankDnew;

end
```

```
function [ rankAnew, rankBnew, rankCnew, rankDnew] = PageRank (G, d, rankA,
rankB, rankC, rankD)

rankAnew = (1-d) + d*(rankC/1);
rankBnew = (1-d) + d*(rankA/2);
rankCnew = (1-d) + d*(rankB/1 + rankA/2 + rankD/1);
rankDnew = (1-d) + d*(0);

end
```

Here's an example with four pages that have quite a few connections between them. Before doing any calculations, we will do some observations on this graph and check if we were right later. Firstly, we see our page D has no incoming links but links to page C itself, thus it should end up with the minimum 1-d value after every iteration. Next, we can see a clear favorite to get the highest rank – page C, as it gets backlinks from 3 different sources. Page A is the only page that gets linked by page C, so we can deduce that it's going to end up with a high rank as well.

After 2 iterations, the results are:

- Rank of A = 2.0837
- Rank of B = 0.5750
- Rank of C = 1.1913
- Rank of D = 0.15

After 10 iterations:

- Rank of A = 1.5002
- Rank of B = 0.7797
- Rank of C = 1.5700
- Rank of D = 0.15

As we can see, our observations were correct, page D is constant 0.15, which is 1-d and we have page C leading the board with page A not far behind. However, it is important to notice that two iterations weren't quite enough for us to get the correct result, thus it is generally important to let a couple of iterations run to formulate the correct result.

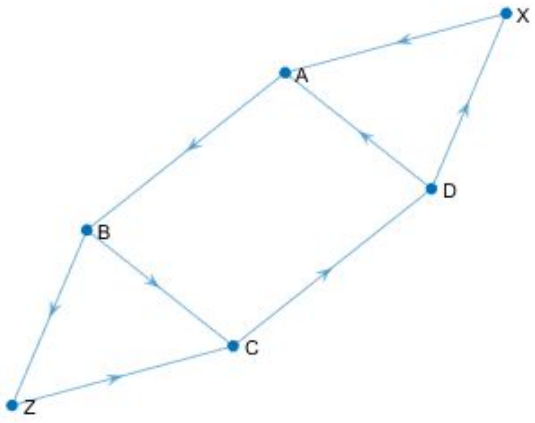
Also in the first case the sum of our ranks is equal to four, thus making each page have a rank of one on average. In the second case of 10 iterations our sum is 3.9999 (converging to 4, so it's alright).

2.2.2 Example 2-3: Loops

If we have a setup of our pages as a loop, the ranks will distribute evenly amongst all of them, resulting in each page in the loop having a rank of 1.

Loop with a twist

Now let's add pages X and Z so that they move some values around as well as add a total of 2 rank value into the equation, we will get:



Example 3

If we modify our function to fit the new graph we will get:

```
function [ rankAnew, rankBnew, rankCnew, rankDnew, rankZnew, rankXnew] =
PageRank (G, d, rankA, rankB, rankC, rankD, rankZ, rankX)

rankZnew = (1-d) + d*(rankB/2);
rankAnew = (1-d) + d*(rankD/2 + rankX/1);
rankBnew = (1-d) + d*(rankA/1);
rankCnew = (1-d) + d*(rankB/2 + rankZ/1);
rankDnew = (1-d) + d*(rankC/1);
rankXnew = (1-d) + d*(rankD/2);
```

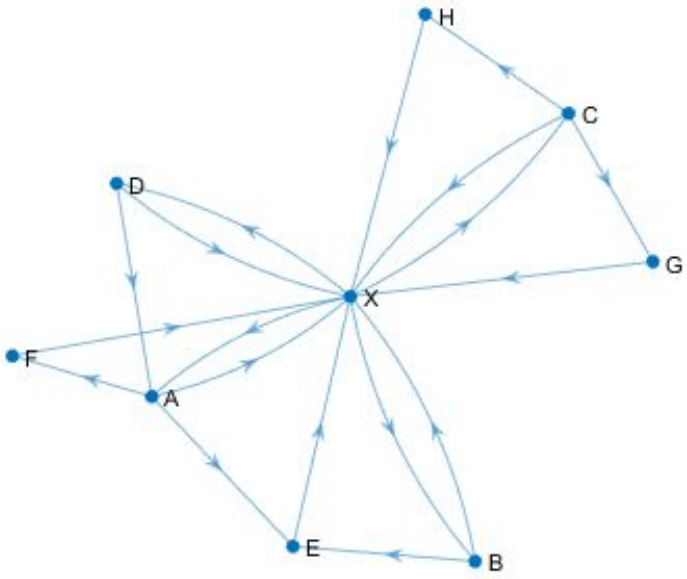
end

And our values will be (after 100 iterations) equal to:

- Rank of A = 1.1922
- Rank of B = 1.1634
- Rank of C = 1.1922
- Rank of D = 1.1634
- Rank of X = 0.6444
- Rank of Z = 0.6444

And indeed, we can see that our pages break our equal loop and transfer some rank from D to A and B to C pages respectfully, thus making pages A and C slightly unbalanced.

2.2.3 Example 4: Graph Tyrant



Let's create a link-structure, which is centered around a single page X, let's call this page the *home* page. Our home page links to four other pages (A, B, C, D), which on their end link to four more pages (E, F, G, H). One would think that our home page's

rank would get depleted very fast, but there's a trick. Every single page on this structure links back to the home page. It is the **Home** page, after all.

Changing up the function a bit to suite the graph we will get:

rankXnew = (1-d) + d*(rankE/1 + rankF/1 + rankG/1 + rankH/1 + rankA/3 + rankB/2 + rankC/3 + rankD/2);

rankAnew = (1-d) + d*(rankX/4 + rankD/2);
rankBnew = (1-d) + d*(rankX/4);
rankCnew = (1-d) + d*(rankX/4);
rankDnew = (1-d) + d*(rankX/4);

rankEnew = (1-d) + d*(rankB/2 + rankA/3);
rankFnew = (1-d) + d*(rankA/3);
rankGnew = (1-d) + d*(rankC/3);
rankHnew = (1-d) + d*(rankC/3);

After 100 iterations we get the following values:

- Rank of X = 3.2146
- Rank of A = 1.1872
- Rank of B = 0.8331
- Rank of C = 0.8331
- Rank of D = 0.8331
- Rank of E = 0.8404
- Rank of F = 0.4864
- Rank of G = 0.3860
- Rank of H = 0.3860

As we can see, at the expense of its child pages, the *homepage* accumulates very large rank number, which results in links incoming from it being very valuable. This style can be attributed to websites like Wikipedia or major media outlets. This is why a link from CNN or Forbes can move your webpage significantly in the Google searches.

However, in this example we only have ranks from our own system being allocated into different pages. When all of the pages of our system (the small ones and the home page) start getting linked from outside sources, that's where our Home page will get a significantly higher rank. Of course, we would be lucky if each page gave our page even a 0.1 rank, but even those small numbers get effective when in millions.

Chapter 3
Dynamical Systems Method

We have seen the probabilistic model of PageRank when we consider a random surfer who jumps from page to page. This was the initial solution and gives us a general idea of how the algorithm works. Now let's introduce a specific algorithm that calculates PageRank by using dynamical systems. Let us assume that we already know the structure of the web as a forest, which means that we have done crawling and gathered the information we need. We take a connected directed graph, also known as a tree and we calculate the PageRank for each of the nodes. This means that we represent each website by a vertex and then show the links between two vertices as a directed edge. Now let us go through the algorithm step-by-step and look at the Matlab implementation.

3.1 Naïve algorithm

Step 0: Denote the number of nodes by n. Let us take n = 5 for this example.

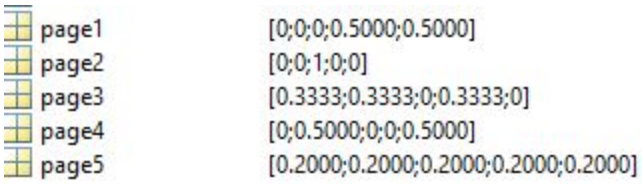
Step 1: We already have our 5 nodes, next we need to declare the links. For each node we create an array of size n and name it page(i), where i goes from 1 to 5. If the current vertex links to another vertex k then the kth value of the current vertex is 1, otherwise it is 0.

```
n = 5;
page1 = [0 0 0 1 1];
page2 = [0 0 1 0 0];
page3 = [1 1 0 1 0];
page4 = [0 1 0 0 1];
page5 = [1 1 1 1 1];
```

Here page1 links to page4 and page5, as it has ones at the last two slots.

Step 2: As mentioned in the probabilistic case, each node is giving equal amount of importance to the vertexes it links to. In our case we consider all the vertexes to have an equal amount of value of 1. If the page(i) links to k pages, then each of them receives 1/k value. In order to calculate this, we create a function called value() which takes the given page and calculates the number of pages it links to by getting the sum of the array. Then the function multiples the array with the reciprocal of the value of the sum.

```
function page = value(i)
i = ((1/sum(i)).*i).';
page = i;
end
```



We see that the arrays changed to take in the 1/k values and the value() function transposed the arrays so that we can concatenate them vertically.

Step 3: We need to create a matrix A representing the graph by concatenating the arrays. So we join them like so:

```
A = [page1 page2 page3 page4 page5];
```

We already have the matrix that holds the values of each link.

Step 4: We create the vector *v*, which will hold our initial, current and final values of the PageRanks of the pages.

```
v = 1/n.*ones(n,1);
```

v is a vertical vector (nX1) filled with 1/n values, which in our case is ⅕. This means that we assign equal PageRank values to each node from the start that add up to 1.

Step 5: Calculating the PageRank. This is where the PageRank is calculated and the dynamical systems are used. We call the function PageRankErr() to assign the final PageRank values to the vector *v*, which is our answer.

```
v = PageRankErr(A,v,0.1);
```

Here is how PageRankErr() works:

```
function rank = PageRankErr(A,v,e)
v1 = v;
err = sum(v);
while err > e;
   v = A*v;
   err = abs(sum(v1) - sum(v));
   v1 = v;
end
rank = v;
end
```

We input the matrix A, matrix v and a number e, which will be our error estimate. PageRank is being calculated when we do the A*v matrix multiplication. As we repeat to multiply with A, the vector v gets closer and closer to the actual solution v*. So the error of A^3v is less than that of A^2v, which is less than that of Av and so on. The error of A^nv tends to 0 as n tends to infinity and our vector v tends to the actual result v*. However we cannot afford to do infinite calculations for reasons of time. So we define a stopping condition e, which will help us get out of the while loop, but still have a pretty accurate result. We chose e to be the absolute value of the difference of v^k and v^{k-1}, where v^k is the vector we get after k iterations. Which means that after some k iterations, the next values of v are very very close to v^k. In this way we can control the accuracy of the result by changing e.

This method of finding PageRank has a few drawbacks. One of the major ones is that it will not work when we have a separate vertex or a few separate trees. These and other issued will be discussed later in the paper.

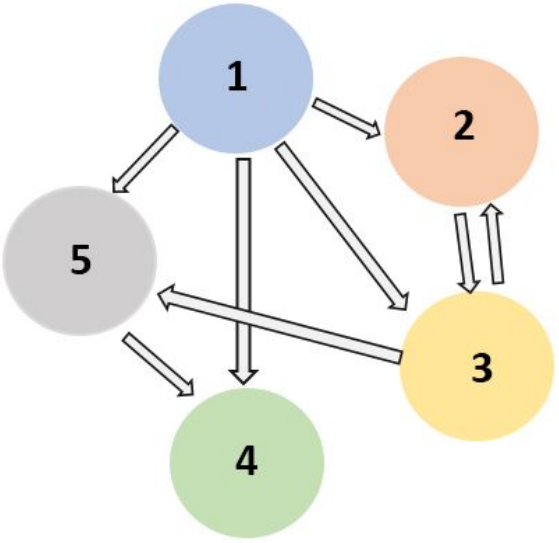
Chapter 4

Linear Algebra Method

Calculation of Google PageRank can be done in various ways. However, the most popular way is using linear algebra tools. We will go deep into into this method by gradually developing the initial, simplest form. And in the end come up with the real Google pagerank algorithm.(4)

4.0.1 Mathematical Model of Internet

First step is representing internet in a graph form.



(Figure 1.1)

Represented web graph shows web pages as nodes and edges as links.

4.1 Algorithm Optimization

4.1.1 Initial Pagerank Summation Formula

The pagerank of page P_i is equal to the sum of pageranks of all pages pointing to P_i, that is

$$r(P_i) = \sum_{P_j \epsilon B_{P_i}} \frac{r(P_j)}{|P_j|}$$

where r (P_i) is the pagerank of page P_i, B_{P_i} is the set of pages pointing to P_i and $|P_j|$ is the number of outlinks from page P_j.

The problem in this representation is that r (P_j), which is the pageranks of pages in-linking to page P_i, are not known. So, next step of developing the method is to get iterative method by assuming that all pages initially have equal pageranks, say if there are n pages , each page has pagerank 1/n.
Now, the iterative method at k+1 iteration is

$$r_{k+1}(P_i) = \sum_{P_j \epsilon B_{P_i}} \frac{r_k(P_j)}{|P_j|}$$

with initial value $r_0(P_i) = 1/n$.

4.1.2 Matrix Representation

For getting rid of that “boring” summation symbol our next task is representing the graph in a matrix form.
Say, every page i has n $\geq$0 outlinks. Now if page i has link to page j, then $S_{ij} = 1/n$, otherwise $S_{ij} = 0$. S_{ij} is actually the probability that surfer moves from page i to page j.
For example the matrix representation of Figure 1.1 will be

$$S = \begin{pmatrix} 0 & 1/4 & 1/4 & 1/4 & 1/4 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1/2 & 0 & 0 & 1/2 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

The nonzero elements of row and column i correspond to out-linking and in-linking pages of page i, respectively.

For iterative matrix representation, there is a need to introduce a row vector $\pi^{(k)T}$, which is the pagerank vector at the k-th iteration.

Now, the matrix representation at k+1 iteration is

$$\pi^{(k+1)T} = \pi^{(k)T} \cdot S$$

4.2 Iterative Method Concerns

The questions arising from iterative method are

- Will the iterative method converge to final value or not?
- In which case are we guaranteed to have convergence?
- Do we have convergence to unique vector or not?
- Does the initial choice of the vector $\pi^{(0)T}$ influence on convergence?

So next we will give answers to this questions and come up with the iterative method which satisfies all questioned properties.

4.2.1 Choice of the initial value

At first, initial value was taken to be $\pi^{(0)T} = 1/n\, e^T$, where e^T is a row vector of all 1's. This choice brings lots of problems: cycles, which cause iterates converge indefinitely(e.g. the nodes 2 and 3 in Figure 1.1), rank sinks such as dangling nodes: nodes having no outlinks(e.g. 4 node in Figure 1.1) .

The first adjustment is replacing 0^T rows with $1/n \cdot e^T$.

$$S' = S + \alpha(1/n \cdot e^T)$$

where α = 1 if page i is a dangling node and otherwise α = 0.
So now the random surfer can hyperlink to any page at random after entering a dangling node.

Let's return to our example (Figure 1.1). After the adjustment our matrix S will be

$$S' = \begin{pmatrix} 0 & 1/4 & 1/4 & 1/4 & 1/4 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1/2 & 0 & 0 & 1/2 \\ 0 & 0 & 0 & 0 & 0 \\ 1/5 & 1/5 & 1/5 & 1/5 & 1/5 \end{pmatrix}$$

The matrix we get after the adjustment is stochastic, which means it is nonnegative nxn matrix and each row sum is one. But this change doesn't guarantee the desired properties: unique pagerank, convergence. So new adjustment is needed.

The offered new matrix is the following

$$G = \beta S' + (1-\beta)\cdot E = \beta S' + (1-\beta)\, 1/n \cdot e \cdot e^T = \beta(S + 1/n \cdot \alpha \cdot e^T) + \\ + (1-\beta)1/n \cdot e \cdot e^T = \beta S + (\beta\alpha + (1-\beta)e)\cdot 1/n \cdot e^T$$

where β is the scalar from interval [0,1] (parameter controlling the proportion of time the random surfer follows the hyperlinks as opposed to teleporting), E = $1/n \cdot e \cdot e^T$ is called teleportation matrix (E is uniform, so teleporting is random i.e when teleporting, it is equiprobable to jump to any page).

4.2.2 Google PageRank method

So after the adjustments the iterative pagerank method is

$$\pi^{(k+1)T} = \pi^{(k)T} G$$

Returning to our example(Figure 1.1) for $\alpha = 0.8$, the matrix G will be

$$G = 0.8\,S + (0.8 \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} + 0.2 \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}) \, 1/5 \, (1\ 1\ 1\ 1\ 1)$$

$$= \begin{pmatrix} 0 & 1/5 & 1/5 & 1/5 & 1/5 \\ 0 & 0 & 4/5 & 0 & 0 \\ 0 & 2/5 & 0 & 0 & 2/5 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 4/5 & 0 \end{pmatrix} + \begin{pmatrix} 1/25 & 1/25 & 1/25 & 1/25 & 1/25 \\ 1/25 & 1/25 & 1/25 & 1/25 & 1/25 \\ 1/25 & 1/25 & 1/25 & 1/25 & 1/25 \\ 1/5 & 1/5 & 1/5 & 1/5 & 1/5 \\ 1/25 & 1/25 & 1/25 & 1/25 & 1/25 \end{pmatrix} =$$

$$= \begin{pmatrix} 1/25 & 6/25 & 6/25 & 6/25 & 6/25 \\ 1/25 & 1/25 & 21/25 & 1/25 & 1/25 \\ 1/25 & 11/25 & 1/25 & 1/25 & 11/25 \\ 1/5 & 1/5 & 1/5 & 1/5 & 1/5 \\ 1/25 & 1/25 & 1/25 & 21/25 & 1/25 \end{pmatrix}$$

Due to the adjustments we have made, the matrix G is

- stochastic as it's complex combination of two stochastic matrices S' and E
- irreducible as every page is directly connected to every other page
- aperiodic because of self-loops
- primitive as there is some k s.t $G^k > 0$. Therefore there exists a unique positive π^T and the power method applied to G is guaranteed to converge to this vector

Modifications brought to the idea of Markov chain. First and second modifications guarantee that the matrix S is stochastic and primitive. Stochasticity implies our matrix is transition probability matrix for Markov chain and primitivity implies both the irreducibility and aperiodicity of our matrix. Therefore, the stationary vector of the chain (i.e the pagerank vector) exists, is unique and can be found using power method.

4.3 Computation of PageRank

We need to use power method for computing the eigenvector of G, which is the PageRank we are seeking.

Our aim is to find π such that

$$\pi^T G = \pi^T$$

Power method says pick an initial guess $x^{(0)}$, repeat $[x^{(k+1)}]^T = [x^{(k)}]^T$

Here each iteration is a multiplication of a matrix and a vector.

The cost of all iterations is the number of nonzero elements in matrix S.

Another method for calculation is reformulating the problem to be

$$\pi^T (I - G) = 0^T \qquad \pi^T e = 1$$

Here we solve linear homogeneous system for π^T.

Chapter 5

The Results and Conclusion

We looked at three methods of the PageRank algorithm implementation. Each of them had their advantages and problems. We tried to optimise our algorithms to include various cases that might exist in the web, such as self referencing or completely independent sites. These algorithms still have a great deal of improvement space available in terms of correctness, applicability to the real example of the web, time and space complexity and so on.

Knowledge of the Google Pagerank algorithm is crucial nowadays if one is working on a website and wants to improve their findability / relevance on search engines. Key practices are to have a home page to accumulate the ranks inside the system and generally, provide good content on as many pages as possible. Good, shareable, linkable content is the key to having a good SEO and being #1 on Google search.

Chapter 6

Appendices

6.1 Algorithm Implementation in MatLab

The following are the MatLab codes appropriate to each method

6.1.1 Probabilistic

Example 1:

```
EdgeTable = table({'A' 'B'; 'A' 'C'; 'B' 'C';'C' 'A';'D' 'C'},[1/2 1/2 1 1 1]', ...
   'VariableNames',{'EndNodes','Weight'});
G = digraph(EdgeTable);
plot(G);

d = 0.85;

rankA = 1;
rankB = 1;
rankC = 1;
rankD = 1;

for k = 1:10
[ rankAnew, rankBnew, rankCnew, rankDnew ] = PageRank (G, d, rankA, rankB, rankC,
rankD);
```

```
rankA = rankAnew;
rankB = rankBnew;
rankC = rankCnew;
rankD = rankDnew;

end

function [ rankAnew, rankBnew, rankCnew, rankDnew ] = PageRank( G, d, rankA,
rankB, rankC, rankD )

rankAnew = (1-d) + d*(rankC/1);
rankBnew = (1-d) + d*(rankA/2);
rankCnew = (1-d) + d*(rankB/1 + rankA/2 + rankD/1);
rankDnew = (1-d) + d*(0);

end
```

Example 2 & 3:

```
EdgeTable = table({'A' 'B'; 'B' 'C'; 'C' 'D'; 'D' 'A'; 'B' 'Z'; 'X' 'A'; 'D' 'X'; 'Z' 'C';},[1 1 1 1 1 1 1
1]', ...
    'VariableNames',{'EndNodes','Weight'});
G = digraph(EdgeTable);
plot(G);

d = 0.85;

rankA = 1;
```

```
rankB = 1;
rankC = 1;
rankD = 1;
rankZ = 1;
rankX = 1;

for k = 1:100
[ rankAnew, rankBnew, rankCnew, rankDnew, rankZnew, rankXnew ] = PageRank (G,
d, rankA, rankB, rankC, rankD, rankZ, rankX);

rankZ = rankZnew;
rankA = rankAnew;
rankB = rankBnew;
rankC = rankCnew;
rankD = rankDnew;
rankX = rankXnew;

end

function [ rankAnew, rankBnew, rankCnew, rankDnew, rankZnew, rankXnew ] =
PageRank( G, d, rankA, rankB, rankC, rankD, rankZ, rankX )

rankZnew = (1-d) + d*(rankB/2);
rankAnew = (1-d) + d*(rankD/2 + rankX/1);
rankBnew = (1-d) + d*(rankA/1);
rankCnew = (1-d) + d*(rankB/2 + rankZ/1);
rankDnew = (1-d) + d*(rankC/1);
rankXnew = (1-d) + d*(rankD/2);
```

```
end
```

Example 4:

```
EdgeTable = table({'X' 'A'; 'X' 'B'; 'X' 'C'; 'X' 'D'; 'A' 'X'; 'B' 'X'; 'C' 'X'; 'D' 'X'; 'D' 'A' ; 'A' 'E';
'A' 'F'; 'B' 'E'; 'C' 'G'; 'C' 'H'; 'E' 'X'; 'F' 'X'; 'G' 'X'; 'H' 'X';},[1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1]', ...
    'VariableNames',{'EndNodes','Weight'});
G = digraph(EdgeTable);
plot(G);

d = 0.85;

rankX = 1;

rankA = 1;
rankB = 1;
rankC = 1;
rankD = 1;

rankE = 1;
rankF = 1;
rankG = 1;
rankH = 1;

for k = 1:100
[ rankXnew, rankAnew, rankBnew, rankCnew, rankDnew, rankEnew, rankFnew,
rankGnew, rankHnew ] = PageRank (G, d, rankX, rankA, rankB, rankC, rankD, rankE,
rankF, rankG, rankH);
```

```
rankX = rankXnew;

rankA = rankAnew;
rankB = rankBnew;
rankC = rankCnew;
rankD = rankDnew;

rankE = rankEnew;
rankF = rankFnew;
rankG = rankGnew;
rankH = rankHnew;

end

function [ rankXnew, rankAnew, rankBnew, rankCnew, rankDnew, rankEnew,
rankFnew, rankGnew, rankHnew ] = PageRank (G, d, rankX, rankA, rankB, rankC,
rankD, rankE, rankF, rankG, rankH)

rankXnew = (1-d) + d*(rankE/1 + rankF/1 + rankG/1 + rankH/1 + rankA/3 + rankB/2 +
rankC/3 + rankD/2);

rankAnew = (1-d) + d*(rankX/4 + rankD/2);
rankBnew = (1-d) + d*(rankX/4);
rankCnew = (1-d) + d*(rankX/4);
rankDnew = (1-d) + d*(rankX/4);

rankEnew = (1-d) + d*(rankB/2 + rankA/3);
rankFnew = (1-d) + d*(rankA/3);
```

```
rankGnew = (1-d) + d*(rankC/3);
rankHnew = (1-d) + d*(rankC/3);

end
```

6.1.2 Dynamical Systems

The dynamical systems algorithm is the following:

```
n = 5;
page1 = [0 0 0 1 1];
page2 = [0 0 1 0 0];
page3 = [1 1 0 1 0];
page4 = [0 1 0 0 1];
page5 = [1 1 1 1 1];

page1=value(page1);
page2=value(page2);
page3=value(page3);
page4=value(page4);
page5=value(page5);

A = [page1 page2 page3 page4 page5];
v = 1/n.*ones(n,1);
v = PageRankErr(A,v,0.1);
```

Where the ***value()*** function is:

```
function page = value(i)
```

```
i = ((1/sum(i)).*i).';
page = i;
end
```

And the ***PageRankErr()*** function is:

```
function rank = PageRankErr(A,v,e)
v1 = v;
err = sum(v);
while err > e;
    v = A*v;
    err = abs(sum(v1) - sum(v));
    v1 = v;
end
rank = v;
end
```

6.1.3 Linear Algebra

The MatLab code given below is calculating the eigenvector for Figure 1.1 example using power method.

```
%% LA example
% pagerank vector calculation using power method
n = 5;
i = [1 1 1 1 2 3 3 5];
j = [2 3 4 5 3 2 5 4];
G = sparse(i,j,1,n,n);          % creating sparse matrix
spy(G)                          % visually seeing the distribution
```

```
%% PageRank Vector (Linear Homogenous System Solution)
p = 0.8;                %scalar parameter
c = sum(G,1);           % column sums
k = find(c~=0);
D = sparse(k,k,1./c(k),n,n);
e = ones(n,1);
I = speye(n,n);
pi = (I - p*G*D)\e;
pi = pi/sum(pi);                %pagerank vector

%% Power Method
z = ((1-p)*(c~=0) + (c==0))/n;
A = p*G*D + e*z;
pi = e/n;
oldpi = zeros(n,1);
while norm(pi - oldpi) > .01
oldpi = pi;
pi = A*pi;
end
pi = pi/sum(pi);                %pagerank vector
```

Bibliography

1. Brin, S. and Page, L. (1998), *The Anatomy of a Large-Scale Hypertextual Web Search Engine*: http://ilpubs.stanford.edu:8090/361/ http://ilpubs.stanford.edu:8090/361/1/1998-8.pdf
2. Page, Lawrence and Brin, Sergey and Motwani, Rajeev and Winograd, Terry (1999), *PageRank Citation Ranking*: http://ilpubs.stanford.edu:8090/422/ http://ilpubs.stanford.edu:8090/422/1/1999-66.pdf
3. Ian Rogers, *The Google Pagerank Algorithm and How It Works*: http://www.cs.princeton.edu/~chazelle/courses/BIB/pagerank.htm
4. Amy N. Langville, Carl D. Meyer, *Google's PageRank and Beyond: The Science of Search Engine Rankings:* http://geza.kzoo.edu/~erdi/patent/langvillebook.pdf