PageRank Computation and Comparison

December 24, 2015

Abstract

The PageRank algorithm was invented by Larry Page in 1999, which was used as the algorithm to determine the importance of websites. Larry built a mathematical model and computed real values of each web page. It was implemented by iteration of matrix multiplication. He proved that the values of matrix would converge after a series of iteration, and as the number of web pages grows, the times of iteration would also grow. However, we usually focus on the rank order of pages, while regardless to the real value of each page. Then in most cases we could get a relatively stable order of web pages, though maybe the matrix does not converge. In this article, I try to use experiments to detect my conjecture.

1 Brief Introduction to the PageRank

The algorithm abstracts web to a vast directed graph. We know most pages has some number of forward links (outedges) and backlinks (inedges) (shows in Figure 1). In this algorithm, Larry regards that a web is more important when more other webs link to it.

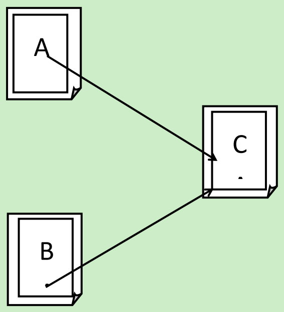


Figure 1: Links of web pages

Let μ be a web page. Then let F_μ be the set of pages μ points to and B_μ be the set of pages that point to μ. Let $N_\mu = |F_\mu|$ be the number of links from u and let c be a factor used for normalization, and R is a simplified version of PageRank:

$$R(\mu) = c \sum_{v \in B_\mu} \frac{R(v)}{N_v}$$

Note that the rank of a page is divided among its forward links evenly to contribute to the ranks of the pages they point to. So that $c \le 1$ because there are a number of pages with

no forward links and their weight is lost from the system.
To make things more clearly, we can image there are only four web pages in our space, which constructed a directed connected graph. And we could use a square matrix with the rows and column to corresponding the graph. Let $A_{\mu,\nu} = {}^{1}\!/\!{}_{N_\mu}$ if there is an edge from μ to ν and $A_{\mu,\nu} = 0$ if not. If we treat R as a vector over web pages, then we have $R = cAR$. So R is an eigenvector of A with eigenvector c. And we could use a matrix (See Figure 3) to represent this graph. Then we could compute the probability matrix by using $R' = cMR$.

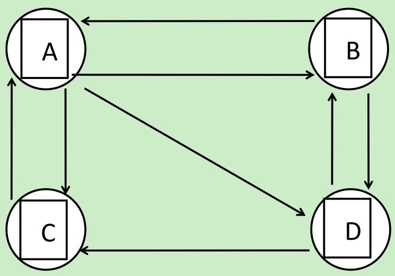


Figure 2: Directed connected graph

$$M = \begin{bmatrix} 0 & 1/2 & 1 & 0 \\ 1/3 & 0 & 0 & 1/2 \\ 1/3 & 0 & 0 & 1/2 \\ 1/3 & 1/2 & 0 & 0 \end{bmatrix}$$

Figure 3: Directed connected graph Matrix

If we set $c = 1$, we can get the new page vector:

$$R' = MR = \begin{bmatrix} 0 & 1/2 & 1 & 0 \\ 1/3 & 0 & 0 & 1/2 \\ 1/3 & 0 & 0 & 1/2 \\ 1/3 & 1/2 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1/4 \\ 1/4 \\ 1/4 \\ 1/4 \end{bmatrix} = \begin{bmatrix} 9/24 \\ 5/24 \\ 5/24 \\ 5/24 \end{bmatrix}$$

Iterate this procedure, and output the following vector sequence:

$$\begin{bmatrix} 1/4 \\ 1/4 \\ 1/4 \\ 1/4 \end{bmatrix} \begin{bmatrix} 9/24 \\ 5/24 \\ 5/24 \\ 5/24 \end{bmatrix} \begin{bmatrix} 15/48 \\ 11/48 \\ 11/48 \\ 11/48 \end{bmatrix} \cdots \begin{bmatrix} 3/9 \\ 2/9 \\ 2/9 \\ 2/9 \end{bmatrix}$$

We can see this is actually a ***Markov Process***, the iteration will converge after several times, and the final state of the vector is independent with the initiate vector. However, this simple situation relies on a condition, which is that the graph must be directed connected. It means a person could get to each page no matter which page he is looking right now. Consider two web pages that point to each other but to no other pages. And suppose there are some other pages pointing to them. Then, during iteration, the loop forms a sort of trap which we can call a rank sink (See figure 4). No matter which page a person is looking, he finally will drop into the sink if he continues his surfing for enough long time.

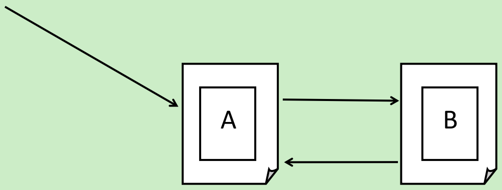


Figure 4: Rank sink

Intuitively, this can be solved by thinking of the real behavior of a random surfer. The random surfer simply keeps clicking on successive links at random. However, if a real Web surfer ever gets into a small loop of web pages, it is unlikely that the surfer will continue in the loop forever. Instead, the surfer will jump to some other page. Thus we define the parameter c as the probability that a surfer will continue looking forward the links on the current web page, otherwise he will open another page randomly, the probability of which is thus $1-c$. We rewrite our formula in the following:

$$R' = cMR + (1-c)E$$

Where E is some vector over the web pages that corresponds to a source of rank. In most cases let E be uniformed over all web pages with value a, because we regard that a random surfer could possibly open any page in the web with the same probability. Consider another web page graph in Figure 5. If a surfer goes into page C, he will never go out anywhere. This is not a Markov Process then, but we could use our new formula to ensure the matrix still converge after several times of iteration.

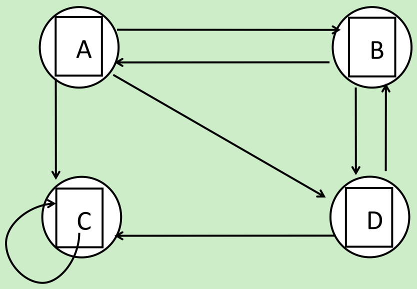


Figure 5: Directed non-connected graph

$$R' = cMR + (1-c)E = 0.8 \times \begin{bmatrix} 0 & 1/2 & 0 & 0 \\ 1/3 & 0 & 0 & 1/2 \\ 1/3 & 0 & 1 & 1/2 \\ 1/3 & 1/2 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1/4 \\ 1/4 \\ 1/4 \\ 1/4 \end{bmatrix} + 0.2 \times \begin{bmatrix} 1/4 \\ 1/4 \\ 1/4 \\ 1/4 \end{bmatrix} = \begin{bmatrix} 9/60 \\ 13/60 \\ 25/60 \\ 13/60 \end{bmatrix}$$

Where c is set to be 0.8, and we continue this calculation with the following sequence:

$$\begin{bmatrix}1/4\\1/4\\1/4\\1/4\end{bmatrix}\begin{bmatrix}9/60\\13/60\\25/60\\13/60\end{bmatrix}\begin{bmatrix}41/300\\53/300\\153/300\\53/300\end{bmatrix}\cdots\begin{bmatrix}15/148\\19/148\\95/148\\19/148\end{bmatrix}$$

2 Conjecture and Procedure

With PageRank algorithm, we could sort the web page to provide surfers the searching results in some order. Typically, the larger rank a page has, the more prior it should be displayed. If we get a converged vector we would know the rank of every page. However, I suppose that as the iteration goes on, the order of the vector will be more stable. Thus, it's no need for us to wait for the convergence of the vector, instead, we can use the intermediate result to sort. To prove my suggestion, I use the following procedure to perform the experiment:

I. Randomly generate a directed graph matrix G, and set $G[i,j]=1$, if there existed a link from i to j, otherwise $G[i,j]=0$;

II. Compute the transform matrix according to the directed graph. For instance, if we get a directed graph matrix (Suppose we have only four web pages):

$$G=\begin{bmatrix}0&1&1&1\\1&0&0&1\\0&0&1&0\\0&1&1&0\end{bmatrix}$$

The transform matrix T will be the transpose matrix of graph matrix, and set $T[i,j]=1/N$ if $G[j,i]=1$, where N is the total links of a page point to others. Using the graph matrix above, we get the following transform matrix:

$$T=\begin{bmatrix}0&1/2&0&0\\1/3&0&0&1/2\\1/3&0&1&1/2\\1/3&1/2&0&0\end{bmatrix};$$

III. Iterate the computation, and after each calculation, we sort the array of pages according to its rank value in a descend order. Here we define a new parameter of stability*:*

$$S=\sum_i D_i \Big/ T_{\max}$$

where S is short for *stability*, D_i means the distance of node i moved in a process of sort, and $T_{\max}$ presents the max moving distance in total if the order of array is in the worst order. The values of these two parameters vary from different sorting algorithm. Here we choose the ***Bubble Sort*** (a stable sorting algorithm), and if we get an array of integers which is in an ascend order like the following:

$$[1,2,3,4,5.......N]$$

To sort this array with a descend order using Bubble Sort, we have to reverse the array. If we change the place of $node1$ and $node2$, we define this moving distance of $node1$ is 1, and after the first time of Bubble Sort, the sequence of this array would turn to:

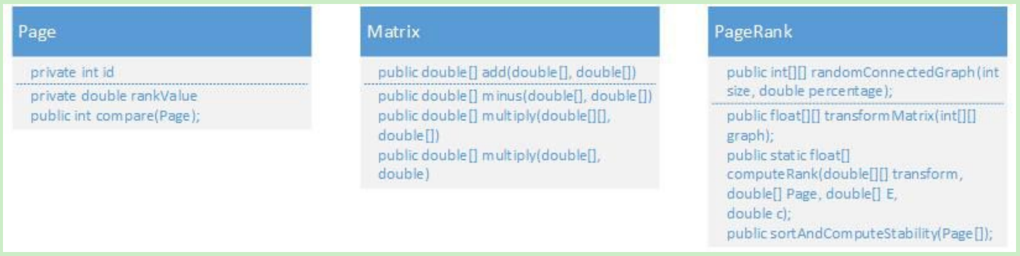
$$[2,3,4,5.....N,1]$$

Then the moving distance of $node1$ is $N-1$. Similarly, we could get the total moving distances of all nodes is $\sum_i D_i$, and in the above example $\sum_i D_i = N(N-1)/2$, which is the worst situation in an array of length N. Then the $T_{\max} = N(N-1)/2$.

Using the parameter ***stability***, we could judge whether an array is stable. The less value the more stable. Obviously, S is zero if the array itself is already well-organized in one process of sorting, and S is one only when the array is in an ascend order;
IV. Record the stability in each time of iteration, and we set a threshold as the mark of convergence. If the difference in two results from two loops of iteration is less than threshold, we believe the matrix has converged.

3 Experiment and Results

I wrote Java Program to perform the experiment:



```
Page
private int id
private double rankValue
public int compare(Page);
```

```
Matrix
public double[] add(double[], double[])
public double[] minus(double[], double[])
public double[] multiply(double[][], double[])
public double[] multiply(double[], double)
```

```
PageRank
public int[][] randomConnectedGraph(int size, double percentage);
public float[][] transformMatrix(int[][] graph);
public static float[] computeRank(double[][] transform, double[] Page, double[] E, double c);
public sortAndComputeStability(Page[]);
```

Figure 6: UML

The class of *Page* is abstracted from a real web page entity. It has two properties, *id* is the mark of *Page* class. Firstly I generate a specific number of *Page*s, and the *id* of each page is given the value of its index in the array; *rankValue* is rank value of each page, which is computed in the matrix iteration.
The class of *Matrix* provides us some basic calculation in matrix or vector, such as *add, minus* and *multiply.*
The class of *PageRank* provides the computation of page rank. The method of *randomConnectedGraphic* is used to randomly generate a directed graph, which represents the relationship between web pages. It has two parameters, the first *size* is the size of the matrix, which is the number of *Page*s we create before. The second *percentage* is the percentage of links in the graph. For example, if we has a 4×4 matrix, and the *percentage* is

0.5. Then we will get a matrix with half zeros and half ones; The *transformMatrix* method generates the transform matrix according to the graph matrix; The *computeRank* method will compute the new value of vector using the formulation $R' = cMR + (1-c)E$ with the input parameters; And the method of *sortAndComputeStability* will sort the array of *Page*s according to the rank value, and return the stability defined by us above.

Experiment I

I pick up different parameters to perform the experiment:

I. $size = 10, threshold = 0.001; percent = 0.5; c = 0.8$

First time of running:

Loop	Stability
1	0.44444445
2	0.11111111
3	0.044444446
4	0.022222223
5	0.0
6	0.0

Second time of running:

Loop	Stability
1	0.33333334
2	0.0
3	0.022222223
4	0.0
5	0.0

Third time of running:

Loop	Stability
1	0.4888889
2	0.13333334
3	0.022222223
4	0.022222223
5	0.0
6	0.0

II. $size = 20, threshold = 1/size^3; percent = 0.5; c = 0.8$

First time of running:

Loop	Stability
1	0.5578947368421052
2	0.07368421052631578

3	0.02631578947368421
4	0.0
5	0.0
6	0.0

Second time of running:

Loop	Stability
1	0.49473684210526314
2	0.05789473684210526
3	0.005263157894736842
4	0.005263157894736842
5	0.0
6	0.0

III. $size = 100, threshold = 1/size^3; percent = 0.5; c = 0.8$

First time of running:

Loop	Stability
1	0.4593939393939394
2	0.022424242424242423
3	0.0022222222222222222
4	0.0
5	0.0
6	0.0
7	0.0

Second time of running:

Loop	Stability
1	0.5038383838383839
2	0.024646464646464646
3	0.0022222222222222222
4	2.0202020202020202E-4
5	0.0
6	0.0

IV. $size = 500, threshold = 1/size^3; percent = 0.5; c = 0.8$

First time of running:

Loop	Stability
1	0.5010100200400801
2	0.011759519038076153
3	4.3286573146292587E-4
4	8.016032064128256E-6

5	0.0
6	0.0
7	0.0

V. $size = 1000, threshold = 1/size^3; percent = 0.5; c = 0.8$

First time of running:

Loop	Stability
1	0.5088468468468469
2	0.008106106106106107
3	2.042042042042042E-4
4	6.006006006006006E-6
5	0.0
6	0.0

VI. $size = 1000, threshold = 0.01/size^3; percent = 0.5; c = 0.8$

First time of running:

Loop	Stability
1	0.4914154154154154
2	0.00785985985985986
3	1.8618618618618617E-4
4	4.004004004004004E-6
5	0.0
6	0.0
7	0.0
8	0.0

VII. $size = 5000, threshold = 0.01/size^3; percent = 0.5; c = 0.8$

First time of running:

Loop	Stability
1	0.505499099819964
2	0.0036177635527105422
3	3.856771354270854E-5
4	7.201440288057612E-7
5	0.0
6	0.0
7	0.0
8	0.0

VIII. $size = 5000, threshold = 1E-5/size^3; percent = 0.5; c = 0.8$

First time of running:

Loop	Stability
1	0.49311934386877376
2	0.003574714942988598
3	4.456891378275655E-5
4	3.200640128025605E-7
5	0.0
6	0.0
7	0.0
....	....
23	0.0

From the above output, we can see that different parameters would take influence on the running results. While the convergence of matrix is generally rapid. What's the most important, the order of pages has been stable after several iteration (It takes 4 times of iteration in my experiment). Though the rank value of them might keep changing after another iteration, their order stay the same. Thus we could use the intermediate results of iteration to sort the page array, without waiting for the convergence. Or we could set the threshold to a bigger value, which could also reduce the times of iteration. **It is noteworthy that** because I test my program only on laptop which has 4-core cpu and 4G RAM, so I did not set the number of pages too large. In the real world, the number of web pages has exceeded 10 billion, thus the transform matrix would be $10billion \times 10billion$. It's not wisdom to directly compute such big matrix, we should take other measures like Map-Reduce. However, I think the conclusion should be the similar regardless of the quantity of web pages.

Experiment II

To make results more convincible, I use real PageRank inputs instead of synthesizing them. Here I use SNAP (Stanford Network Analysis Platform: http://snap.stanford.edu/) to help me simulate my experiment. And I pick up two datasets, one is smaller and the other is bigger. One of them is wiki-Vote (Wikipedia who-votes-on-whom network), it has 7,115 nodes and 103,689 edges. Another is web-Stanford (Web graph of Stanford.edu), which has 281,903 nodes and 2,312,497 edges. Because of their vast quantity, it's not feasible to handle them directly with matrix multiplication. The best is to using Map Reduce. While because of the restriction of devices, I use ***Sparse Matrix*** to store the graph of dataset, which could save a lot more RAM than ordinary matrix.

Data I: wiki-Vote: 7,115 nodes and 103,689 edges

I. $threshold = 0.01, DampedFactor = 0.8$

First time of running:

LOOP	STABILITY	COMPUTING TIME (ms)	SORTING TIME (ms)
1	0.240233791	607	400

2	0.013234759	349	344
3	0.011905182	73	299
4	0.006232798	214	187
5	0.004813724	46	205
6	0.004077753	34	166
7	0.003342098	46	131
8	0.002766669	38	122
9	0.002311912	24	113

Second time of running:

LOOP	STABILITY	COMPUTING TIME (ms)	SORTING TIME (ms)
1	0.240233791	229	447
2	0.013234759	90	226
3	0.011905182	77	217
4	0.006232798	84	226
5	0.004813724	53	170
6	0.004077753	39	147
7	0.003342098	48	157
8	0.002766669	30	150
9	0.002311912	28	132

II. $threshold = 0.0001, DampedFactor = 0.8$

First time of running:

LOOP	STABILITY	COMPUTING TIME (ms)	SORTING TIME (ms)
1	0.240233791	374	491
2	0.013234759	147	206
3	0.011905182	90	252
4	0.006232798	201	183
5	0.004813724	52	148
6	0.004077753	38	141
7	0.003342098	55	156
8	0.002766669	28	147
9	0.002311912	44	135
10	0.002024533	27	126
11	0.001716015	25	129
12	0.001458231	26	115
13	0.001272164	41	107

Second time of running:

LOOP	STABILITY	COMPUTING TIME	SORTING TIME

		(ms)	(ms)
1	0.240233791	312	446
2	0.013234759	566	211
3	0.011905182	57	252
4	0.006232798	99	209
5	0.004813724	32	154
6	0.004077753	31	154
7	0.003342098	37	148
8	0.002766669	48	133
9	0.002311912	33	142
10	0.002024533	38	129
11	0.001716015	33	114
12	0.001458231	30	111
13	0.001272164	40	106

The *COMPUTING TIME* is the running time of computing the rank value in each loop. The *SORTING TIME* is the running time of sorting the array of pages.

In the following, we use the much larger dataset *web-Stanford* to check my experiment. However, because of the dramatically huge quantity of data, the sort method would take very long time to sort the array and compute the stability, which is intolerable. To improve such situation, I use the embedded sorting method in Java JDK: *Arrays.sort*. It's more rapid, though we can not intuitively see the stability of an array, we could still know it according to the elapsed time the method spends. The more ordered an array is, the less time it would cost when sorting.

Data I: web-Stanford: 281,903 nodes and 2,312,497 edges

I. $threshold = 0.01, DampedFactor = 0.8$

LOOP	COMPUTING TIME (ms)	SORTING TIME (ms)
1	1187	619
2	1952	432
3	2748	210
4	1671	124
5	1390	98
6	1391	47
7	1265	21
8	1202	14
9	1473	18
10	1640	12

II. $threshold = 0.0001, DampedFactor = 0.8$

LOOP	COMPUTING TIME (ms)	SORTING TIME (ms)
1	1534	625
2	1813	381
3	2876	237
4	1564	202
5	1629	127
6	1610	51
7	1548	47
8	1638	28
9	1367	18
10	1759	18
11	1502	13
12	1348	10
13	1563	6

Then we could see that as the loop continues, the running time of sorting is becoming less, which means the array of page tends towards stability.

4 Reference

I. Wikipedia PageRank: https://en.wikipedia.org/wiki/PageRank
II. The PageRank Citation Ranking: Bringing Order to the Web, Larry Page, Sergey Brin, 1999
III. PageRank 算法简介及 Map-Reduce 实现:
http://jingyan.baidu.com/article/4ae03de31bbf883eff9e6b1b.html