The PageRank Search Algorithm

Mishkin Faustini

Abstract

This paper serves to show various implementations of the PageRank algorithm in order to show the pros and cons of each approach in terms of scalability and space time cost of evaluation. There are several ways to calculate the PageRank; iteratively, algebraically, using inverse iteration or the power method. The motivation for each method will be discussed, each algorithm will be shown along with any implementation issues, and experimental results discussed.

Introduction/Motivation

One of the most important algorithms in modern day computing is PageRank, the algorithm developed by Larry Page and Sergey Brin as part of a research project at Stanford University. Now PageRank is used as the backbone to Google's search engine operations which is arguably the most useful service available on the internet.

Basic concepts and theory

Basic Formulation

PageRank is a probability distribution used to represent the likelihood that a person would randomly visit a particular webpage. The idea is to imagine a random web surfer visiting a page and randomly clicking links to visit other pages then randomly going to a new page and repeating the process. The probably that the surfer visits a given page is that page's PageRank. In this regard we can consider this process as a Markov chain where the states are pages and the transitions are the links between pages. So what does the probability distribution look like? A probability can be expressed as a numerical value ranging from 0 to 1.0. Thus a page with the value of 0.7 is equal to a 70% chance that a user would randomly visit it

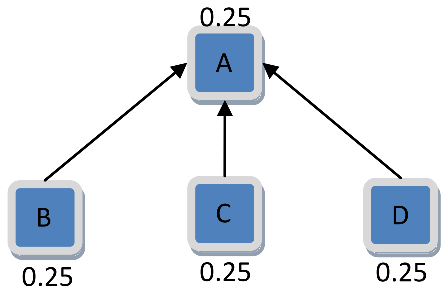


Figure 1 – Set W: Our four pages and their links

Suppose that we have a set of pages W that we wish to run through the PageRank algorithm. To begin we assume that each page has equal probability of being chosen on a random surfer walk. Thus, each page begins with a PageRank of 0.25. If we name the pages A, B, C and D and pages B, C and D link to page A then,

$$PR(A) = \frac{PR(B)}{L(B)} + \frac{PR(C)}{L(C)} + \frac{PR(D)}{L(D)}$$

Where PR(X) is the PageRank(x) and L(X) is the number of outbound links on page X. Thus in the simple case mentioned above we have:

$$PR(A) = \frac{0.25}{1} + \frac{0.25}{1} + \frac{0.25}{1} = 0.75$$

More generally this can be expressed as:

$$PR(a) = \sum_{a \in W_b} \frac{PR(a)}{L(a)}$$

Dampening Factor

PageRank has included a factor to simulate the odds that a user (or our random web surfer) stops clicking links. Thus there is a chance that at any given page the user will stop clicking. The way we can simulate this factor is by using a *dampening factor.* The dampening factor that has been tried and tested in numerous studies happens to be about 0.85. This slightly changes our formula:

$$PR(A) = \frac{1-d}{n} + d\left(\frac{PR(B)}{L(B)} + \frac{PR(C)}{L(C)} + \frac{PR(D)}{L(D)}\right) = \frac{1-d}{N} + d \sum_{a \in W_b} \frac{PR(a)}{L(a)}$$

Where *n* is the total number of pages in the system and *d* is the dampening factor.

Matrix Formation and Calculation

If we have a matrix G that is the adjacency matrix showing the connectivity of all our pages then we can determine the number of inbound and outbound links for a given page $i \text{ or } j$ in our matrix by formulating the two equations, respectively: $r_i = \sum_j g_{ij}$, $c_j = \sum_i g_{ij}$

To solve we must first convert our equation into a modified adjacency matrix, A, and PageRank values will then be the dominant eigenvectors of our matrix. A is an n-by-n matrix that:

$$a_{ij} = \begin{cases} \frac{pg_{ij}}{c_j} + \delta : c_j \neq 0 \\ \frac{1}{n} : \ c_j = 0 \end{cases}, \delta = \frac{1-p}{n}, \ p = 0.85, \ n = \#\, Pages$$

Because we have a matrix which holds the transition probability between pages and the sum of its columns equal one we can conclude the following by *Perron-Frobenius theorem*,

$$x = Ax$$

Where X is a unique matrix if we have a scaling factor such that, $\sum_i x_i = 1$. The resulting solution to *x* is the PageRank calculation.

Algorithms

It should be noted that there are several algorithms to solve the PageRank problem. Here we will look at the Power method and Inverse Iteration. The following are MATLAB implementations of the PageRank algorithm calculation:

Pseudo-code

Power Method

```
% Eliminate any self-referential links
% c = out-degree, r = in-degree
% Scale column sums to be 1 (or 0 where there are no out links).
% Calculate the following..
% G = p*G*D;
% x = initial equal link value (1/n)
% xprev = 1;
% while sum(abs(xprev-x)) > 0.001
%    xprev = x;
%    x = G*x + e*(z*x);
% end
% Normalize so that sum(x) == 1.
```

Inverse Iteration

```
% Eliminate any self-referential links
% c = out-degree, r = in-degree
% Scale column sums to be 1 (or 0 where there are no out links).
% Calculate delta = (1-p)/n
% Calculate A = p*g*D + delta
% solve e =(I-A)*x
% Normalize so that sum(x) == 1.
```

Note: See code files at end of paper for implementation: pagerank_powermethod.m, pagerank_inverseIteration.m, PageRank.cs

Implementation issues

One of the largest implementation issues is the sheer size of the datasets on which PageRank is being calculated. Companies like Google cannot directly use matrix solvers to compute PageRank because their datasets are much too large. Instead a method such as the Power Method is used where a broad sweep over the database can be calculated in several passes.

A second issue with the PageRank algorithm is that it favors older more well established pages over newer pages. If a new page enters the system it will have relatively few outbound and inbound links as compared to a site that has a number of links in and out of it.

Experiment results

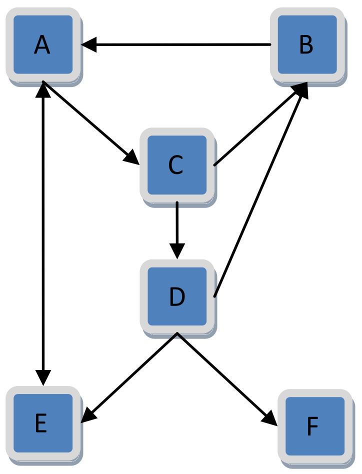
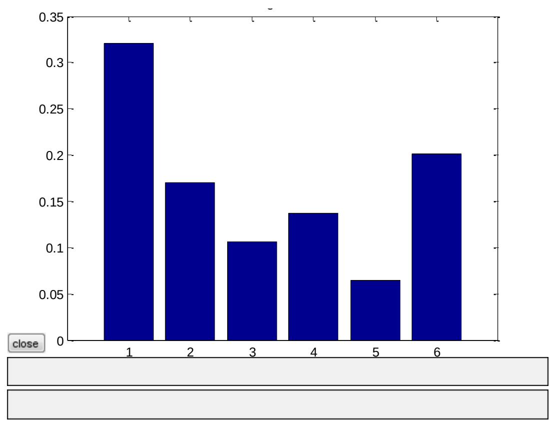


Figure 2 - PageRank Example Scenario

	A	B	C	D	E	F
A			X		X	
B	X					
C		X		X		
D		X			X	X
E						
F						

Corresponding link adjacency matrix

For this example we will look at a basic link structure and attempt to calculate the PageRank using MATLAB.



Page	PageRank	In	Out
A	0.3210	2	2
C	0.1705	1	2
D	0.1066	1	3
B	0.1368	2	1
F	0.0643	1	0
E	0.2007	2	1

After running pagerank_powermethod(U,G) using our adjacency matrix we have the above PageRank results. The results show that page F which has little connection and little chance of being clicked on has the lowest PageRank while A has the highest PageRank.

Thus we can see that the PageRank is being calculated correctly.

The following MATLAB code can reconstruct this scenario:

```
>> i = [2 6 3 4 4 5 6 1 1];

>> U = {'A','C','D','B','F','E'};

>> j = [1 1 2 2 3 3 3 4 6];

>> n=6;

>> G = sparse(i,j,1,n,n);

>> pagerank_powermethod(U,G)
```

Concluding Remarks

There are many tweaks that can be done to further enhance PageRank's performance in terms of calculation efficiency and results. The basic PageRank algorithm is an interesting algorithm to learn. The algorithm itself is far simpler than one may think but the difficulty seems to be in calculating the PageRank efficiently for massive datasets.

A deeper understanding of the inner workings of how to efficiently compute PageRank for massive datasets would be a natural progression from this point.

Acknowledgements

Special thanks to professor Professor Zhaojun Bai and the University of California at Davis to whom has introduced me to Scientific Computation and Cleve Moler for his very informative book Numerical Computing with MATLAB for which I have learned from all quarter.

References

Numerical Computing with MATLAB by Cleve Moler

- http://www.mathworks.com/moler/chapters.html

L. Page, S. Brin, R. Motwani and T. Winograd "The PageRank Citation Ranking: Bringing Order to the Web", Stanford Digital Library working paper SIDL-WP-1999-0120 (version of 11/11/1999). See: http://www-diglib.stanford.edu/cgibin/get/SIDL-WP-1999-0120

A. Arasu, J. Novak, A. Tomkins and J. Tomlin, "PageRank Computation and the Structure of the Web: Experiments and Algorithms", Technical Report, IBM Almaden Research Center, Nov. 2001.

Files

Pagerank_inverseiteration.m

```
function x = pagerank_inverseiteration(U,G,p)
% PAGERANK  Google's PageRank
% pagerank(U,G,p) uses the URLs and adjacency matrix produced by SURFER,
% together with a damping factory p, (default is .85), to compute and plot
% a bar graph of page rank, and print the dominant URLs in page rank order.
% x = pagerank(U,G,p) returns the page ranks instead of printing.
% See also SURFER, SPY.

if nargin < 3, p = .85; end

% Eliminate any self-referential links

G = G - diag(diag(G));

% c = out-degree, r = in-degree

[n,n] = size(G);
c = sum(G,1);
r = sum(G,2);

% Scale column sums to be 1 (or 0 where there are no out links).

k = find(c~=0);
D = sparse(k,k,1./c(k),n,n);

% Calculate delta = (1-p)/n
% Calculate A = p*g*D + delta
% solve e =(I-A)*x
delta = (1-p)/n;

 e = ones(n,1);
 I = speye(n,n);

A = p*G*D + delta;
x = (I - A)\e;
x = x/sum(x);

% Normalize so that sum(x) == 1.

x = x/sum(x);
```

Pagerank_powermethod.m

```
function x = pagerank_powermethod(U,G,p)

if nargin < 3, p = .85; end

% Eliminate any self-referential links

G = G - diag(diag(G));

% c = out-degree, r = in-degree

[n,n] = size(G);
c = sum(G,1);
r = sum(G,2);

% Scale column sums to be 1 (or 0 where there are no out links).

k = find(c~=0);
D = sparse(k,k,1./c(k),n,n);

% Solve (I - p*G*D)*x = e

e = ones(n,1);


G = p*G*D;
z = ((1-p)*(c~=0) + (c==0))/n

x = ones(n,1)/n;
xprev = 1;

while sum(abs(xprev-x)) > 0.001
    xprev = x;
    x = G*x + e*(z*x);
end


% Normalize so that sum(x) == 1.

x = x/sum(x);
```