REPORT

Chumki Acharya

Summer Research Fellow
National Institute of Science & Technology
May,2014

Topic: Page Ranking algorithm

CONTENTS

LIST OF FIGURES

ABSTRACT

With the increasing use of Internet, the web structure is getting more complex day by day. The web users get easily lost in the complex web's hyper structure. The main aim is to give the precise and relevant information according to the needs to the users through an efficient search engine without consuming the time. This report includes different aspects of Page Ranking algorithms used for Information Retrieval to deliver the most relevant pages to the users. Different Page Rank based algorithms like Page Rank (PR), WPR (Weighted Page Rank), HITS (Hyperlink Induced Topic Selection) are discussed with a detail analysis of the search engine architecture.

1. INTRODUCTION

We live in a computer era. Internet is part of our everyday lives and information is only a click away. Just to open our favourite search engine, like Google, Yahoo, type in the key words, and the search engine will display the pages relevant for our search. Information is the most valuable quantity in the present age. Automation and Computerization has led to extremely large data repositories. The World Wide Web is one such example. World Wide Web (WWW), by name The Web, the leading information retrieval service of the Internet (the worldwide computer network). The Web gives users access to a vast array of documents that are connected to each other by means of hypertext or hypermedia links—*i.e.,* hyperlinks, electronic connections that link related pieces of information in order to allow a user easy access to them. Hypertext allows the user to select a word from text and thereby access other documents that contain additional information pertaining to that word; hypermedia documents feature links to images, sounds, animations, and movies. The Web operates within the Internet's basic client-server format; servers are computer programs that store and transmit documents to other computers on the network when asked to, while clients are programs that request documents from a server as the user asks for them. Browser software allows users to view the retrieved documents. A hypertext document with its corresponding text and hyperlinks is written in Hyper Text Markup Language (HTML) and is assigned an online address called a Uniform Resource Locator (URL).

The World Wide Web (WWW) is rapidly growing on all aspects and is a massive, dynamic and mostly unstructured data repository. As on today WWW is the huge information repository

for knowledge reference. There are a lot of challenges in the Web: Web is large, Web pages are semi structured, and Web information tends to be diversity in meaning, degree of quality of the information extracted and the conclusion of the knowledge from the extracted information. So it is important to understand and analyze the underlying data structure of the Web for efficient Information Retrieval. Web mining techniques along with other areas like Database (DB), Natural Language Processing (NLP), Information Retrieval (IR), Machine Learning etc. can be used to solve the challenges. Search engines like Google, Yahoo, Bing etc., are used to find information from the World Wide Web (WWW) by the users.

2. SEARCH ENGINE ARCHITECTURE :

The simple architecture of a search engine is shown in Figure 1. There are 3 important components in a search engine. They are Crawler, Indexer and Ranking mechanism. The crawler is also called as a robot or spider that traverses the web and downloads the web pages. The downloaded pages are sent to an indexing module that parses the web pages and builds the index based on the keywords in those pages. An index is generally maintained using the keywords. When a user types a query using keywords on the interface of a search engine, the query processor component match the query keywords with the index and returns the URLs of the pages to the user. But before showing the pages to the user, a ranking mechanism is done by the search engines to show the most relevant pages at the top and less relevant ones at the bottom.

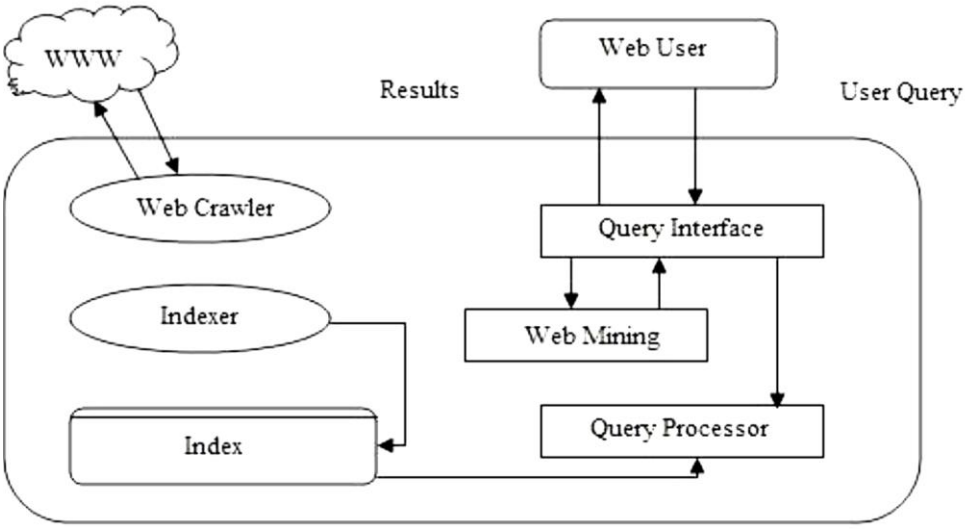


Figure 1: Simple Architecture of Search Engine.

In this section, we will give a high level overview of how the whole system works as pictured in Figure 2. Most of Google is implemented in C or C++ for efficiency and can run in either Solaris or Linux. In Google, the web crawling (downloading of web pages) is done by several distributed crawlers. There is a URL server that sends lists of URLs to be fetched to the crawlers. The web pages that are fetched are then sent to the store server. The store server then compresses and stores the web pages into a repository. Every web page has an associated ID number called a docID which is assigned whenever a new URL is parsed out of a web page. The indexing function is performed by the indexer and the sorter. The indexer performs a number of functions. It reads the repository, uncompresses the documents, and parses them. Each document is converted into a set of word occurrences called hits. The hits record the word, position in document, an approximation of font size, and capitalization. The indexer distributes these hits into a set of "barrels", creating a partially sorted forward index. The indexer performs another important function. It parse out all the links in every web page and stores important information about them in an anchors file. This file

contains enough information to determine where each link points from and to, and the text of the link.

The URLresolver reads the anchors file and converts relative URLs into absolute URLs and in turn into docIDs. It puts the anchor text into the forward index, associated with the docID that the anchor points to. It also generates a database of links which are pairs of docIDs. The links database is used to compute PageRanks for all the documents. The sorter takes the barrels, which are sorted by docID and resorts them by wordID to generate the inverted index. This is done in place so that little temporary space is needed for this operation. The sorter also produces a list of wordIDs and offsets into the inverted index. A program called DumpLexicon takes this list together with the lexicon produced by the indexer and generates a new lexicon to be used by the searcher. The searcher is run by a web server and uses the lexicon built by DumpLexicon together with the inverted index and the PageRanks to answer queries.

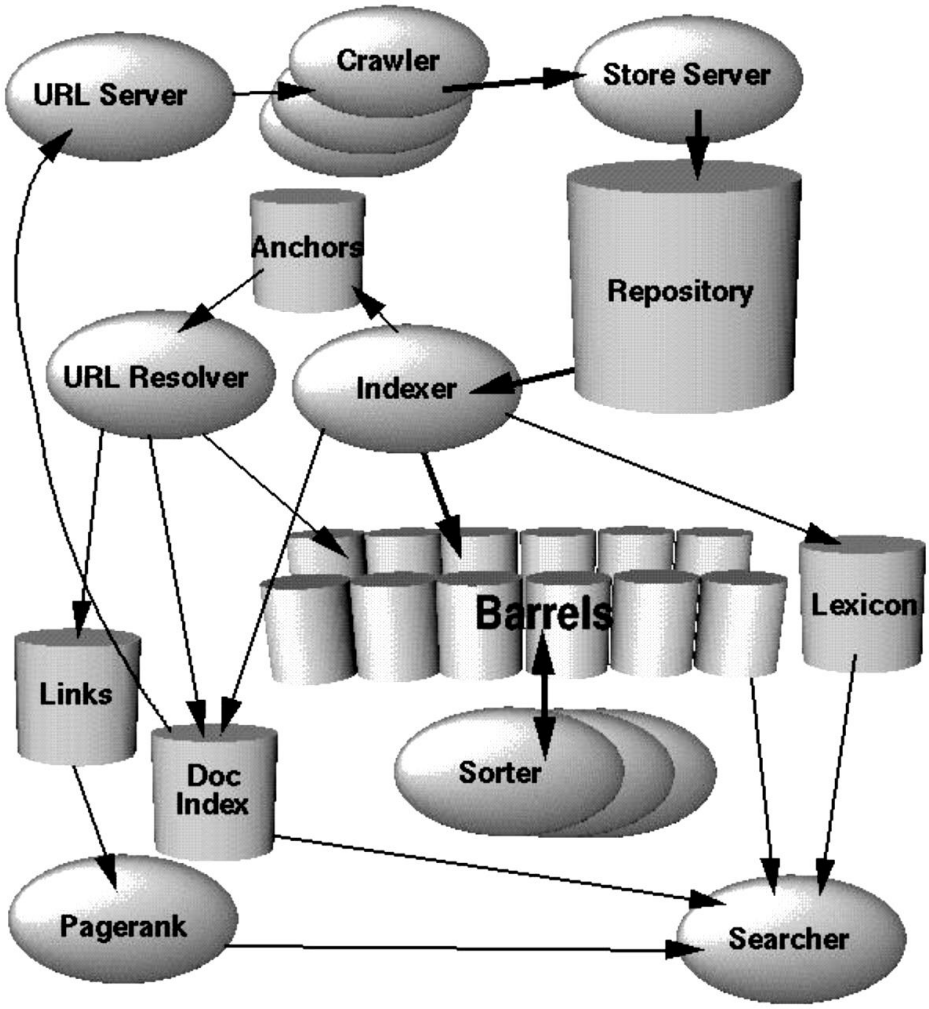


Figure 2: High Level Google Architecture.

2.1 Major Data Structures

Google's data structures are optimized so that a large document collection can be crawled, indexed, and searched with little cost. Although, CPUs and bulk input output rates have improved dramatically over the years, a disk seek still requires about 10 ms to complete. Google is designed to avoid disk seeks whenever possible, and this has had a considerable influence on the design of the data structures.

2.1.1 BigFiles

BigFiles are virtual files spanning multiple file systems and are addressable by 64 bit integers. The allocation among multiple file systems is handled automatically. The BigFiles package also handles allocation and deallocation of file descriptors, since the operating systems do not provide enough for our needs. BigFiles also support rudimentary compression options.

2.1.2 Repository

The repository contains the full HTML of every web page. Each page is compressed. In the repository, the documents are stored one after the other and are prefixed by docID, length, and URL. The repository requires no other data structures to be used in order to access it. This helps with data consistency and makes development much easier; we can rebuild all the other data structures from only the repository and a file which lists crawler errors.

2.1.3 Document Index

The document index keeps information about each document. It is a fixed width index, ordered by docID. The information stored in each entry includes the current document status, a pointer into the repository, a document checksum, and various statistics. If the document has been crawled, it also contains a pointer into a variable width file called docinfo which contains its URL and title. Otherwise the pointer points into the URLlist which contains just the URL. Additionally, there is a file which is used to convert URLs into docIDs. It is a list of URL checksums with their corresponding docIDs and is sorted by checksum. In order to find the docID of a particular URL, the URL's checksum is computed and a binary search is performed on the checksums file to find its docID. URLs

may be converted into docIDs by doing a merge with this file. This is the technique the URLresolver uses to turn URLs into docIDs.

2.1.4 Lexicon

The lexicon has several different forms. One important change from earlier systems is that the lexicon can fit in memory for a reasonable price. In the current implementation we can keep the lexicon in memory on a machine with 256 MB of main memory. The current lexicon contains 14 million words (though some rare words were not added to the lexicon). It is implemented in two parts -- a list of the words and a hash table of pointers.

2.1.5 Hit Lists

A hit list corresponds to a list of occurrences of a particular word in a particular document including position, font, and capitalization information. Hit lists account for most of the space used in both the forward and the inverted indices. Because of this, it is important to represent them as efficiently as possible. We considered several alternatives for encoding position, font, and capitalization -- simple encoding (a triple of integers), a compact encoding (a hand optimized allocation of bits), and Huffman coding. In the end we chose a hand optimized compact encoding since it required far less space than the simple encoding and far less bit manipulation than Huffman coding. Our compact encoding uses two bytes for every hit. There are two types of hits: fancy hits and plain hits. Fancy hits include hits occurring in a URL, title, anchor text, or meta tag. Plain hits include everything else. A plain hit consists of a capitalization bit, font size, and 12 bits of word position in a document . We use font size relative to the rest of the document because when searching, you do not want to rank otherwise identical documents differently just because one of the

documents is in a larger font. The length of a hit list is stored before the hits themselves. To save space, the length of the hit list is combined with the wordID in the forward index and the docID in the inverted index.

2.1.6 Forward Index

The forward index is actually already partially sorted. It is stored in a number of barrels (we used 64). Each barrel holds a range of wordID's. If a document contains words that fall into a particular barrel, the docID is recorded into the barrel, followed by a list of wordID's with hitlists which correspond to those words. This scheme requires slightly more storage because of duplicated docIDs but the difference is very small for a reasonable number of buckets and saves considerable time and coding complexity in the final indexing phase done by the sorter.

2.1.7 Inverted Index

The inverted index consists of the same barrels as the forward index, except that they have been processed by the sorter. For every valid wordID, the lexicon contains a pointer into the barrel that wordID falls into. It points to a doclist of docID's together with their corresponding hit lists. This doclist represents all the occurrences of that word in all documents. An important issue is in what order the docID's should appear in the doclist. One simple solution is to store them sorted by docID. Another option is to store them sorted by a ranking of the occurrence of the word in each document. Since both the options have some flaws, we chose a compromise between these options, keeping two sets of inverted barrels -- one set for hit lists which include title or anchor hits and another set for all hit lists. This way, we check the

first set of barrels first and if there are not enough matches within those barrels we check the larger ones.

2.2 Crawling the Web

Running a web crawler is a challenging task. There are tricky performance and reliability issues and even more importantly, there are social issues. Crawling is the most fragile application since it involves interacting with hundreds of thousands of web servers and various name servers which are all beyond the control of the system.

In order to scale to hundreds of millions of web pages, Google has a fast distributed crawling system. A single URLserver serves lists of URLs to a number of crawlers. Both the URLserver and the crawlers are implemented in Python. Each crawler keeps roughly 300 connections open at once. This is necessary to retrieve web pages at a fast enough pace. At peak speeds, the system can crawl over 100 web pages per second using four crawlers. This amounts to roughly 600K per second of data. A major performance stress is DNS lookup. Each crawler maintains its own DNS cache so it does not need to do a DNS lookup before crawling each document. Each of the hundreds of connections can be in a number of different states: looking up DNS, connecting to host, sending request, and receiving response. These factors make the crawler a complex component of the system. It uses asynchronous IO to manage events, and a number of queues to move page fetches from state to state. Also, because of the huge amount of data involved, unexpected things will happen. Because of the immense variation in web pages and servers, it is virtually impossible to test a crawler without running it on large part of the Internet.

Invariably, there are hundreds of obscure problems which may only occur on one page out of the whole web and cause the crawler to crash, or worse, cause unpredictable or incorrect behaviour. Systems which access large parts of the Internet need to be designed to be very robust and carefully tested. Since large complex systems such as crawlers will invariably cause problems, there needs to be significant resources devoted for this and solving these problems as they come up.

2.3 Indexing the Web

2.3.1 Parsing -- Any parser which is designed to run on the entire Web must handle a huge array of possible errors. These range from typos in HTML tags to kilobytes of zeros in the middle of a tag, non-ASCII characters, HTML tags nested hundreds deep, and a great variety of other errors that challenge anyone's imagination to come up with equally creative ones. For maximum speed, instead of using YACC to generate a CFG parser, we use flex to generate a lexical analyzer which we outfit with its own stack. Developing this parser which runs at a reasonable speed and is very robust involved a fair amount of work.

2.3.2 Indexing Documents into Barrels -- After each document is parsed, it is encoded into a number of barrels. Every word is converted into a wordID by using an in-memory hash table -- the lexicon. New additions to the lexicon hash table are logged to a file. Once the words are converted into wordID's, their occurrences in the current document are translated into hit lists and are written into the forward barrels.

2.3.3 Sorting -- In order to generate the inverted index, the sorter takes each of the forward barrels and sorts it by wordID to produce an inverted barrel for title and anchor hits and a full text inverted barrel.
This process happens one barrel at a time, thus requiring little temporary storage. Also, we parallelize the sorting phase to use as many machines as we have simply by running multiple sorters, which can process different buckets at the same time. Since the barrels don't fit into main memory, the sorter further subdivides them into baskets which do fit into memory based on wordID and docID. Then the sorter, loads each basket into memory, sorts it and writes its contents into the inverted barrel.

2.4. Searching

The goal of searching is to provide quality search results efficiently. Many of the large commercial search engines seemed to have made great progress in terms of efficiency. Therefore, we have focused more on quality of search in our research, although we believe our solutions are scalable to commercial volumes with a bit more effort.
To put a limit on response time, once a certain number (currently 40,000) of matching documents are found, the searcher automatically goes to a state where the sub-optimal results would be returned. Sorting the hits according to PageRank, seemed to improve the situation.

3. PAGE RANKING ALGORITHMS :

With the growing number of Web pages and users on the Web, the number of queries submitted to the search engines are also growing rapidly day by day. Therefore, the search engines needs to be more efficient in its processing way and output. Web mining

techniques are employed by the search engines to extract relevant documents from the web database documents and provide the necessary and required information to the users. The search engines become very successful and popular if they use efficient ranking mechanisms. Now these days it is very successful because
of its PageRank algorithm. Page ranking algorithms are used by the search engines to present the search results by considering the relevance, importance and content score and web mining techniques to order them according to the user interest.

3.1 PageRank Algorithm

PageRank algorithm is developed by Brin and Page during their Ph. D at Stanford University. PageRank algorithm is used by the famous search engine that is Google. This algorithm is the most commonly used algorithm for ranking the various pages. Working of the PageRank algorithm depends upon link structure of the web pages. The PageRank algorithm is based on the concepts that if a page contains important links towards it then the links of this page towards the other page are also to be considered as important pages. The PageRank considers the back link in deciding the rank score. If the addition of the all the ranks of the back links is large then the page then it is provided a large rank. Therefore, PageRank provides a more advanced way to compute the importance or relevance of a web page
than simply counting the number of pages that are linking to it. If a backlink comes from an important page, then that backlink is given a higher weighting than those backlinks comes from non-important pages. In a simple way, link from one page to another page may be considered as a vote. However, not only the number of votes a page receives is considered important, but the

importance or the relevance of the ones that cast these votes as well.

If we create a web page *i* and include a hyperlink to the web page *j*, this means that we consider *j* important and relevant for our topic. If there are a lot of pages that link to *j*, this means that the common belief is that page *j* is important. If on the other hand, *j* has only one backlink, but that comes from an authoritative site *k*, (like www.google.com, www.cnn.com, www.cornell.edu) we say that *k* transfers its authority to *j*; in other words, *k* asserts that *j* is important. Whether we talk about popularity or authority, we can iteratively assign a rank to each web page, based on the ranks of the pages that point to it.

To this aim, we begin by picturing the Web net as a directed graph, with nodes represented by web pages and edges represented by the links between them.

Suppose for instance, that we have a small Internet consisting of just 4 web sites www.page1.com, www.page2.com, www.page3.com, www.page4.com, referencing each other in the manner suggested by the picture:

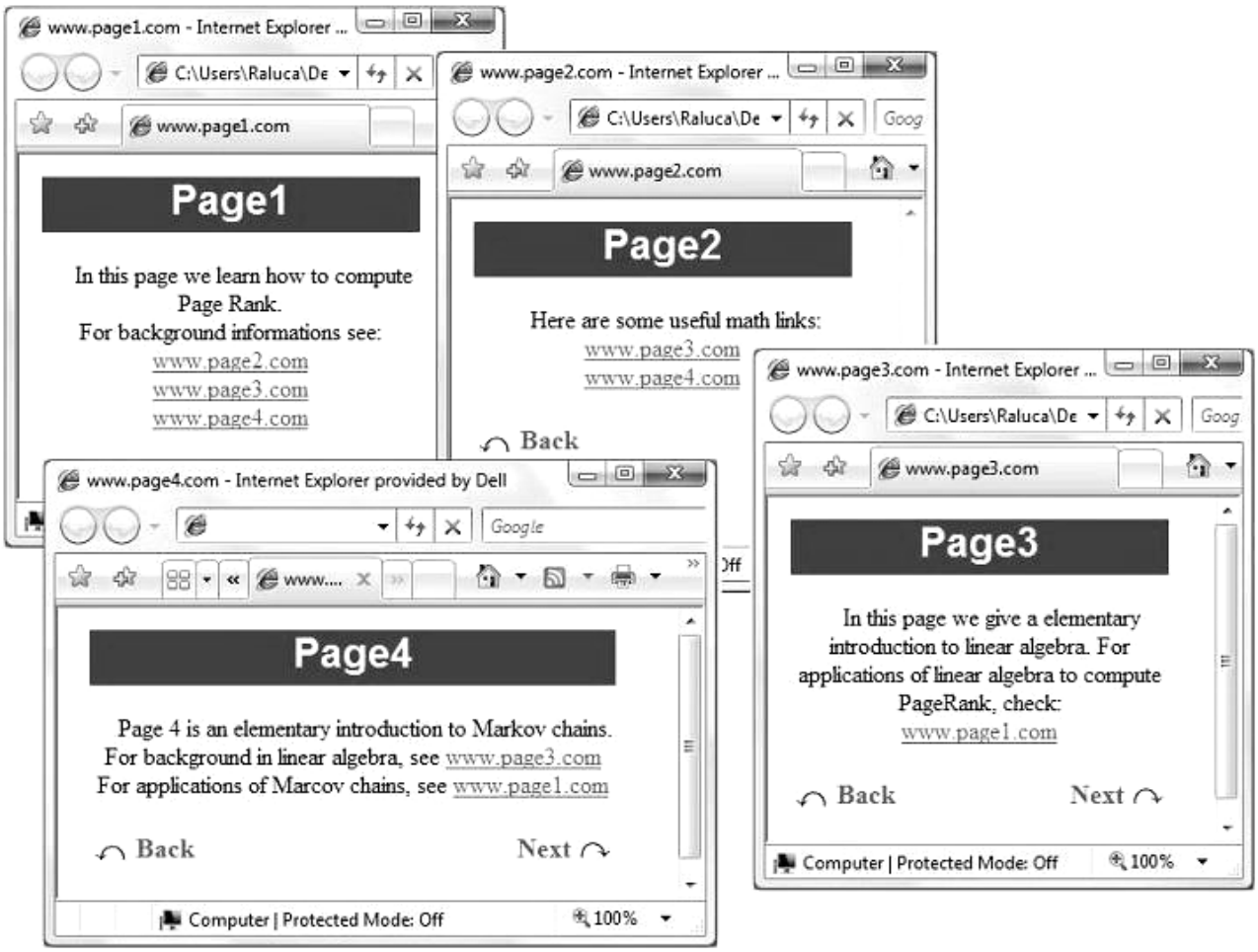


Figure 3: Showing links between the web pages

We "translate" the picture into a directed graph with 4 nodes, one for each web site. When web site *i* references *j*, we add a directed edge between node *i* and node *j* in the graph. For the purpose of computing their page rank, we ignore any navigational links such as back, next buttons, as we only care about the connections between different web sites. For instance, Page1 links to all of the other pages, so node 1 in the graph will have outgoing edges to all of the other nodes. Page3 has only one link, to Page 1, therefore node 3 will have one outgoing edge to node 1. After analyzing each web page, we get the following graph:

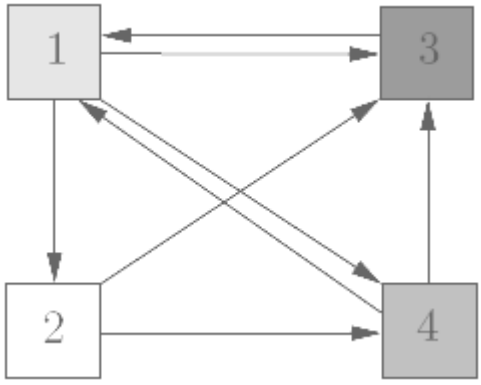


Figure 4 : Graph representation of the pages

In our model, each page should transfer evenly its importance to the pages that it links to. Node 1 has 3 outgoing edges, so it will pass on 1/3 of its importance to each of the other 3 nodes. Node 3 has only one outgoing edge, so it will pass on all of its importance to node 1. In general, if a node has *k* outgoing edges, it will pass on 1/k of its importance to each of the nodes that it links to. Let us better visualize the process by assigning weights to each edge.

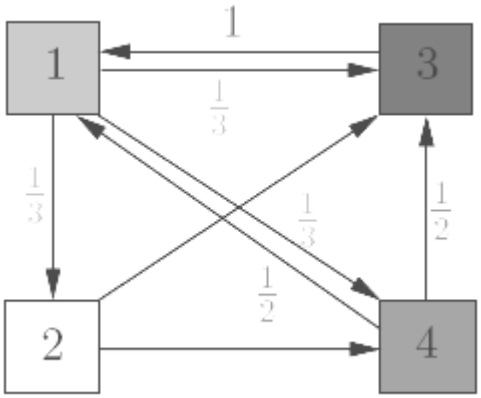


Figure 5 : Graph representation of the pages showing the importance

Let us denote by *A* the transition matrix of the graph, $A = \begin{bmatrix} 0 & 0 & 1 & \frac{1}{2} \\ \frac{1}{3} & 0 & 0 & 0 \\ \frac{1}{3} & \frac{1}{2} & 0 & \frac{1}{2} \\ \frac{1}{3} & \frac{1}{2} & 0 & 0 \end{bmatrix}$

Suppose that initially the importance is uniformly distributed among the 4 nodes, each getting ¼. Denote by *v* the initial rank vector, having all entries

equal to ¼. Each incoming link increases the importance of a web page, so at step 1, we update the rank of each page by adding to the current value the importance of the incoming links. This is the same as multiplying the matrix A with v . At step 1, the new importance vector is $v1 = Av$. We can iterate the process, thus at step 2, the updated importance vector is $v2 = A(Av) = A^2v$. Numeric computations give:

$$v = \begin{pmatrix} 0.25 \\ 0.25 \\ 0.25 \\ 0.25 \end{pmatrix}, \quad Av = \begin{pmatrix} 0.37 \\ 0.08 \\ 0.33 \\ 0.20 \end{pmatrix}, \quad A^2 v = A(Av) = A \begin{pmatrix} 0.37 \\ 0.08 \\ 0.33 \\ 0.20 \end{pmatrix} = \begin{pmatrix} 0.43 \\ 0.12 \\ 0.27 \\ 0.16 \end{pmatrix}$$

$$A^3 v = \begin{pmatrix} 0.35 \\ 0.14 \\ 0.29 \\ 0.20 \end{pmatrix}, \quad A^4 v = \begin{pmatrix} 0.39 \\ 0.11 \\ 0.29 \\ 0.19 \end{pmatrix}, \quad A^5 v = \begin{pmatrix} 0.39 \\ 0.13 \\ 0.28 \\ 0.19 \end{pmatrix}$$

$$A^6 v = \begin{pmatrix} 0.38 \\ 0.13 \\ 0.29 \\ 0.19 \end{pmatrix}, \quad A^7 v = \begin{pmatrix} 0.38 \\ 0.12 \\ 0.29 \\ 0.19 \end{pmatrix}, \quad A^8 v = \begin{pmatrix} 0.38 \\ 0.12 \\ 0.29 \\ 0.19 \end{pmatrix}$$

Figure 6: Showing the page rank vector

We notice that the sequences of iterates $v, Av, ..., A^kv$ tends to the equilibrium value $v^* = \begin{pmatrix} 0.38 \\ 0.12 \\ 0.29 \\ 0.19 \end{pmatrix}$

We call this the PageRank vector of our web graph.

3.1.1.Problems in changing the web graph

1. Nodes with no outgoing edges (dangling nodes)

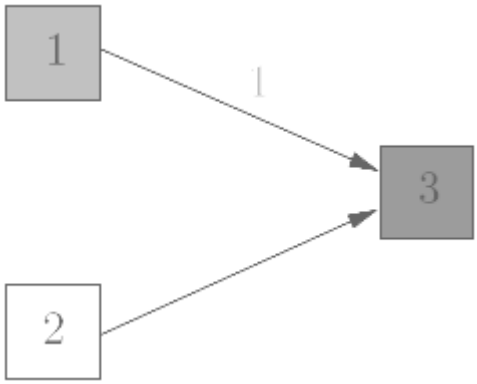


Figure 7: Graph with dangling nodes

We iteratively compute the rank of the 3 pages:

$$v_0 = \begin{bmatrix} \frac{1}{3} \\ \frac{1}{3} \\ \frac{1}{3} \end{bmatrix}, \quad v_1 = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} \frac{1}{3} \\ \frac{1}{3} \\ \frac{1}{3} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \frac{2}{3} \end{bmatrix}, \quad v_2 = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 0 \\ 0 \\ \frac{2}{3} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

.

So in this case the rank of every page is 0. This is counterintuitive, as page 3 has 2 incoming links, so it must have some importance!

2. <u>Disconnected components :</u>

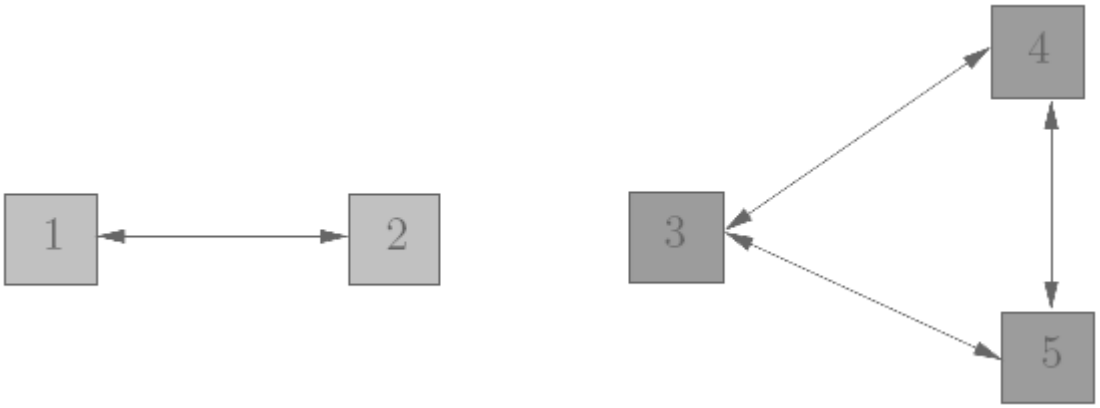


Figure 8: Graph with disconnected components

random surfer that starts in the first connected component has no way of getting to web page 5 since the nodes 1 and 2

have no links to node 5 that he can follow. Linear algebra fails to help as well.

3.1.2 The solution of Page and Brin:

In order to overcome these problems, fix a positive constant *d* between 0 and 1, which we call the damping factor.
It is given by
PR(A) = (1-d) + d (PR(T1)/C(T1) + ... + PR(Tn)/C(Tn))
Where
PR(A) is the PageRank of page A,
PR(Ti) is the PageRank of pages Ti which link to page A,
C(Ti) is the number of outbound links on page Ti and d is a damping factor which can be set between 0 and 1.
So, first of all, we see that PageRank does not rank web sites as a whole, but is determined for each page individually. Further, the PageRank of page A is recursively defined by the PageRanks of those pages which link to page A.

Let us take an example of hyperlink structure of four pages *A*, *B*, *C* and *D* as shown in Figure 4.
The PageRank for pages *A*, *B*, *C* and *D* can be calculated by using the above equation.

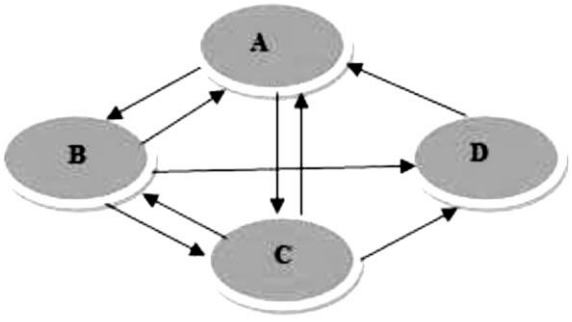


Figure 9: Hyperlink structure of four pages

Let us assume the initial PageRank as 1 and do the calculation. The value of damping factor d is
put to 0.85.

$PR(A) = (1\text{-}d) + d\,(PR(B)/C(B)+PR(C)/C(C)+PR(D)/C(D))$
$= (1\text{-}0.85) + 0.85(1/3+1/3+1/1)$
$= 1.5666667$ (3)
$PR(B) = (1\text{-}d) + d((PR(A)/C(A) + (PR(C)/C(C))$
$= (1\text{-}0.85) + 0.85(1.5666667/2+1/3)$
$= 1.0991667$ (4)
$PR(C) = (1\text{-}d) + d((PR(A)/C(A) + (PR(B)/C(B))$
$= (1\text{-}0.85) + 0.85(1.5666667/2+1.0991667/3)$
$= 1.127264$ (5)
$PR(D) = (1\text{-}d) + d((PR(B)/C(B) + (PR(C)/C(C))$
$= (1\text{-}.085) + 0.85(1.0991666/3+1.127264/3)$
$= 0.7808221$ (6)

For the second iteration by taking the above *PageRank* values from (3), (4), (5) and (6). The second iteration PageRank values are as following:

$PR(A) = 0.15 + 0.85((1.0991667/3) + (1.127264/3)+(0.7808221/1)$
$= 1.4445208$ (7)
$PR(B) = 0.15 + 0.85((1.4445208/2) + (1.127264/3))$
$= 1.0833128$ (8)
$PR(C) = 0.15 + 0.85((1.4445208/2) + (1.0833128/3))$
$= 1.07086$ (9)
$PR(D) = 0.15 + 0.85((1.0833128/3)+(1.07086/3))$
$= 0.760349$ (10)

During the computation of 34th iteration, the average of the all web pages is 1. Some of the PageRank values are shown in Table:

Iteration	A	B	C	D
1	1	1	1	1
2	1.5666667	1.0991667	1.127264	0.7808221
3	1.4445208	1.0833128	1.07086	0.760349
.....	.....	.....	.....	.....
17	1.3141432	0.9886763	0.9886358	0.7102384
18	1.313941	0.9885384	0.98851085	0.71016395
19	1.3138034	0.98844457	0.98842573	0.7101132

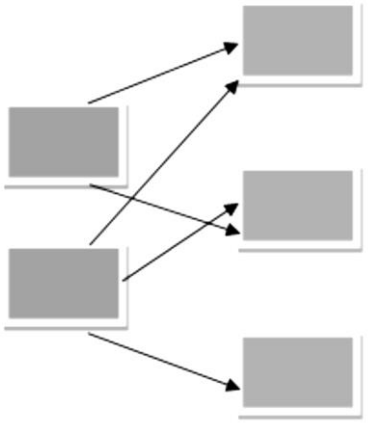
Table : Page Rank values.

3.2 Weighted Page Rank

Weighted PageRank Algorithm is proposed by Wenpu Xing and Ali Ghorbani. Weighted PageRank algorithm (WPR) is the modification of the original PageRank algorithm. WPR decides the rank score based on the popularity of the pages by taking into consideration the importance of both the inlinks and outlinks of the pages. This algorithm provides high value of rank to the more popular pages and does not equally divide the rank of a page among it's outlink pages. Every out-link page is given a rank value based on its popularity. Popularity of a page is decided by observing its number of in links and out links. As suggested, the performance of WPR is to be tested by using different websites and future work include to calculate the rank score by utilizing more than one level of reference page list and increasing the number of human user to classify the web pages.

3.3 HITS Algorithm

The HITS algorithm is proposed by Kleinberg in 1988. HITS algorithm identifies two different forms of Web pages called hubs and authorities. Authorities are pages having important contents. Hubs are pages that act as resource lists, guiding users to authorities. Thus, a good hub page for a subject points to many authoritative pages on that content, and a good authority page is pointed by many good hub pages on the same subject. Hubs and Authorities are shown in figure 5. In this a page may be a good hub and a good authority at the same time. This circular relationship leads to the definition of an iterative algorithm called HITS (Hyperlink Induced Topic Selection). HITS algorithm is ranking the web page by using inlinks and outlinks of the web pages. In this a web page is named as authority if the web page is pointed by many hyper links and a web page is named as hub if the page point to various hyperlinks. An Illustration of hub and authority are shown in figure 5. HITS is, technically, a link based algorithm. In HITS algorithm, ranking of the web page is decided by analyzing their textual contents against a given query. After collection of the web pages, the HITS algorithm concentrates on the structure of the web only, neglecting their textual contents.



Hub Authorities

Figure 10. Illustration of hubs and authorities

4. **STORAGE ISSUES :**

The size of the Transition matrix makes storage issues nontrivial. However this is the fact that the web graph is *sparse*. If most of the entries of a matrix are zero then the matrix is said to be sparse. In such a case it may be very expensive to store the zero values.
This means that a node *i* has a small number of outgoing links (a couple of hundred at best, which is extremely small corresponding to the 30 billion nodes it could theoretically link to). Hence the transition matrix *A* has a lot of entries equal to 0.
When storing and manipulating sparse matrices on a computer, it is beneficial and often necessary to use specialized algorithms and data structures that take advantage of the sparse structure of the matrix. Operations using standard dense-matrix structures and algorithms are relatively slow and consume large amounts of memory when applied to large sparse matrices. Sparse data is by nature easily compressed, and this compression almost always results in significantly less computer data storage usage.

Substantial memory requirement reductions can be realized by storing only the non-zero entries. Depending on the number and distribution of the non-zero entries, different data structures can be used and yield huge savings in memory when compared to the basic approach.

Formats can be divided into two groups: (1) those that support efficient modification, and (2) those that support efficient matrix operations. The efficient modification group includes DOK (Dictionary of keys), LIL (List of lists), and COO (Coordinate list) and is typically used to construct the matrix. Once the matrix is constructed, it is typically converted to a format, such as CSR

(Compressed Sparse Row) or CSC (Compressed Sparse Column), which is more efficient for matrix operations.

4.1. Dictionary of keys (DOK)

DOK represents non-zero values as a dictionary (e.g., a hash table or binary search tree) mapping (row, column) pairs to values.

4.2. List of lists (LIL)

LIL stores one list per row, where each entry stores a column index and value.

For example-

$$A = \begin{pmatrix} 1 & 0 & 0 & 2 & 0 & 3 \\ 4 & 5 & 0 & 0 & 0 & 0 \\ 0 & 6 & 7 & 0 & 0 & 8 \\ 9 & 0 & 0 & 10 & 11 & 12 \\ 0 & 13 & 0 & 0 & 14 & 0 \\ 0 & 0 & 0 & 0 & 0 & 15 \end{pmatrix}$$

ROW	(col,val)	(col,val)	(col,val)	(col,val)
0	(0,1)	(3,2)	(5,3)	
1	(0,4)	(1,5)		
2	(1,6)	(2,7)	(5,8)	
3	(0,9)	(3,10)	(4,11)	(5,12)
4	(1,13)	(4,14)		
5	(5,15)			

Figure 11.1- LIL format of sparse matrix

4.3. Coordinate list (COO)

COO stores a list of (row, column, value) tuples.

For example-

$$A = \begin{pmatrix} 1.5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 2.3 & 0 & 1.4 & 0 & 0 & 0 & 0 \\ 0 & 0 & 3.7 & 0 & 0 & -2.7 & 0 & 0 \\ 0 & -1.6 & 0 & 2.3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 5.8 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 7.4 & 0 & 0 \\ 0 & 0 & 1.9 & 0 & 0 & 0 & 4.9 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 3.6 \end{pmatrix}$$

row	0	1	1	2	2	3	3	4	5	6	6	7
column	0	1	3	2	5	1	3	4	5	2	6	7
value	1.5	2.3	1.4	3.7	–2.7	–1.6	2.3	5.8	7.4	1.9	4.9	3.6

Figure 11.2- COO format of sparse matrix

4.4. Compressed sparse row (CSR or CRS)

CSR is effectively identical to the Yale Sparse Matrix format, except that the column array is normally stored ahead of the row index array. I.e. CSR is (val, col_ind, row_ptr), where val is an array of the (left-to-right, then top-to-bottom) non-zero values of the matrix; col_ind is the column indices corresponding to the values; and, row_ptr is the list of value indexes where each row starts. The name is based on the fact that row index information is compressed relative to the COO format.

For example -

$$\mathbf{A} = \begin{pmatrix} 1 & 0 & 0 & 2 & 0 & 3 \\ 4 & 5 & 0 & 0 & 0 & 0 \\ 0 & 6 & 7 & 0 & 0 & 8 \\ 9 & 0 & 0 & 10 & 11 & 12 \\ 0 & 13 & 0 & 0 & 14 & 0 \\ 0 & 0 & 0 & 0 & 0 & 15 \end{pmatrix}$$

```
vals: 1 2 3 4  5  6  7 8 9 10 11 12 13 14 15
cols: 0 3 5 0  1  1  2 5 0  3  4  5  1  4  5
rows: 0 3 5 8 12 14 15
```

Figure 11.3- CSR format of sparse matrix

4.5. Compressed sparse column (CSC or CCS)

CSC is similar to CSR except that values are read first by column, a row index is stored for each value, and column pointers are stored. I.e. CSC is (val, row_ind, col_ptr), where val is an array of the (top-to-bottom, then left-to-right) non-zero values of the

matrix; row_ind is the row indices corresponding to the values; and, col_ptr is the list of val indexes where each column starts. The name is based on the fact that column index information is compressed relative to the COO format.

For example-

$$A = \begin{pmatrix} 10 & 0 & 0 & 0 & -2 & 0 \\ 3 & 9 & 0 & 0 & 0 & 3 \\ 0 & 7 & 8 & 7 & 0 & 0 \\ 3 & 0 & 8 & 7 & 5 & 0 \\ 0 & 8 & 0 & 9 & 9 & 13 \\ 0 & 4 & 0 & 0 & 2 & -1 \end{pmatrix}.$$

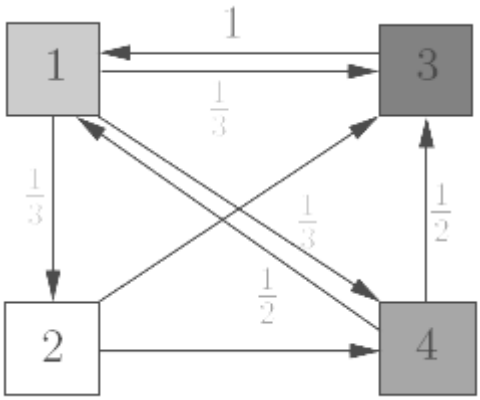
val	10	-2	3	9	3	7	8	7	3 ··· 9	13	4	2	-1
col_ind	1	5	1	2	6	2	3	4	1 ··· 5	6	2	5	6
row_ptr	1	3	6	9	13	17	20						

Figure 11.4- CSC format of sparse matrix

5. IMPLEMENTATION AND RESULTS :

We have taken the previous example for the implementation. The above graph of four nodes in Figure 5 has been considered for finding out the page rank vector. The complete source code has been included in the Appendix.

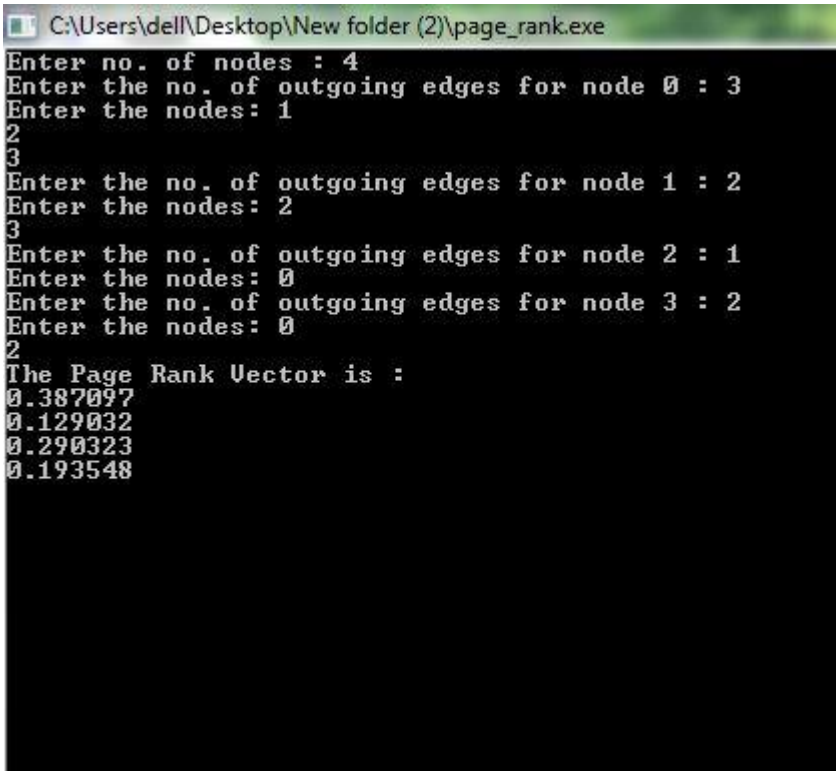
The graph is demonstrated as :



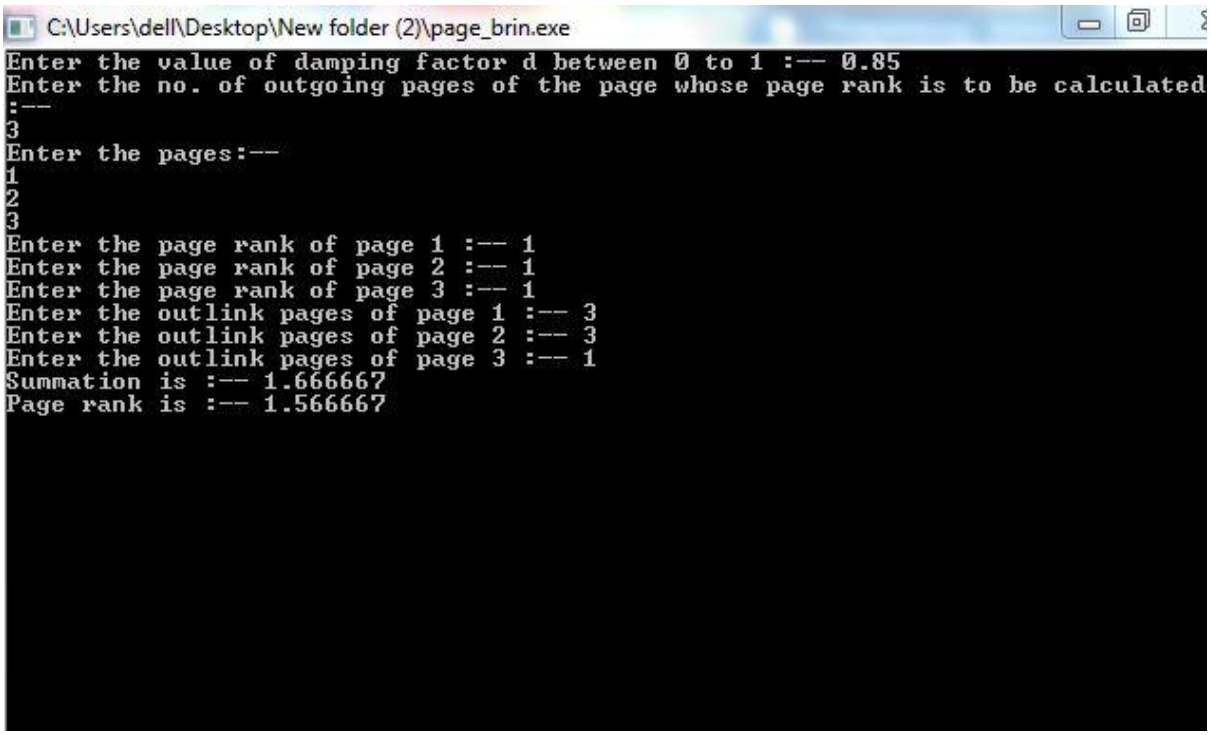
Let us denote by A the transition matrix of the graph, A =

$$\begin{bmatrix} 0 & 0 & 1 & \frac{1}{2} \\ \frac{1}{3} & 0 & 0 & 0 \\ \frac{1}{3} & \frac{1}{2} & 0 & \frac{1}{2} \\ \frac{1}{3} & \frac{1}{2} & 0 & 0 \end{bmatrix}$$

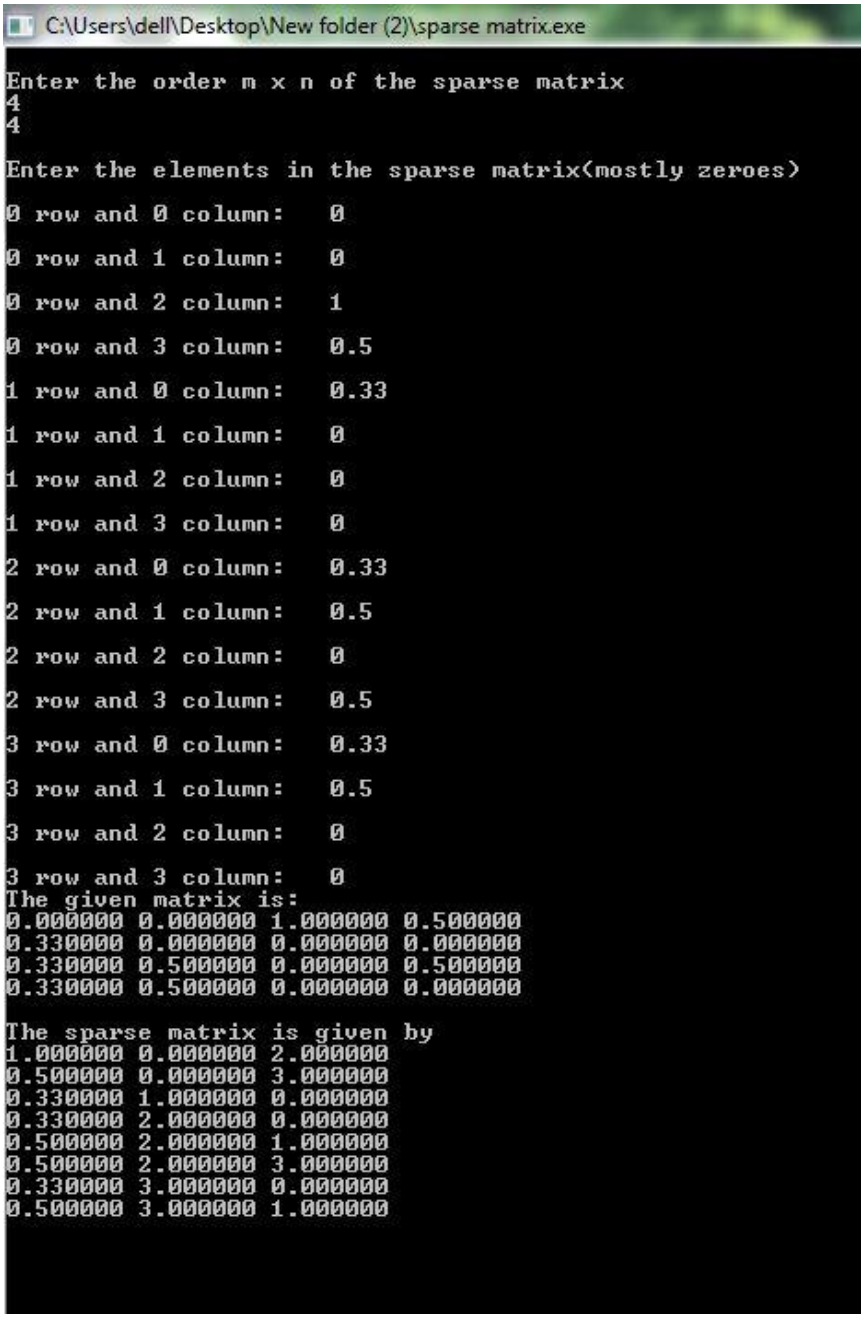
The page rank vector of this graph has been calculated and the output has been observed to be



The graph of Figure9 has been taken for calculating the page rank of each page by page and brin method taking the damping factor into consideration.



Since the transition matrix is a sparse matrix and having high magnitudes, the storage of the matrix requires different techniques in order to prevent memory consumption.
The sparse matrix representation of the above graph is observed to be



Calculating the page rank from this sparse matrix also has been done.

6.CONCLUSION AND FUTURE WORK :

In this project work we gave an overview of search engines detail architecture, how the search engine works and how the user can get the most relevant page as output from the search engine through different page ranking algorithms. Computation of Page Rank vector, Page rank of a particular web page and storage issues for storing the web graph was discussed and implemented for a simple graph. Most of the work has been concentrated on page ranking with an overview of weighted page rank algorithm and HITS algorithm.

Due to time constraints an efficient search engine with a new page rank algorithm to provide the users the most relevant search was not developed. However the basic aspects have been understood and implemented. The future work consists of the following:

- Design and analysis of a new page rank algorithm.
- Design of an efficient search engine.

  Details study and analysis of other page ranking algorithms are to be done to develop a new algorithm and more works are to be done on search engine crawler and indexer.

7. APPENDIX:

1. Code for page rank vector calculation in C++

```
#include <iostream>
#include <vector>
using namespace std;
#include<conio.h>
vector<vector<double> > createTransitionMatrix(int n)
{
    vector<vector<double> > matrix;
    vector<double> row(n,0);
    for(int i=0; i<n; i++)
    matrix.push_back(row);
    return matrix;
}

vector<vector<double> > createVector(int n)
{
   vector<vector<double> > vect;
    double temp=(double)1/n;
   vector<double> row(1,temp);
    for(int i=0; i<n; i++)
       vect.push_back(row);
    return vect;
}

vector<vector<double> > initRank(int n)
{
   vector<vector<double> > rank;
    vector<double> row(1,0.0);
    for(int i=0; i<n; i++)
       rank.push_back(row);
```

```
    return rank;
}


vector<vector<double> >
initTransitionMatrix(vector<vector<double> > matrix)
{
   int n , node;
    vector<int> row;
    for(int i=0; i< matrix.size(); i++)
   {
      cout << "Enter the no. of outgoing edges for node
"<<i<<" : ";
      cin >> n;
      if(n > 0)
      {
        cout << "Enter the nodes: ";
            for(int j=0; j<n; j++)
            {
               cin >> node;
                    row.push_back(node);
       }
            for(int j=0; j < row.size(); j++)
              matrix[row[j]][i]=(double)1/n;
            row.clear();
      }
    }
    return matrix;
}

int equalsMatrix(vector<vector<double> > mat ,
vector<vector<double> > matrix)
```

```
{
   if(mat.size()!=matrix.size())
            return -1;
    for(int i=0; i<mat.size(); i++)
    {
       if(mat[i].size()!=matrix[i].size())
              return -1;
            for(int j=0; j<mat[i].size(); j++)
              if(mat[i][j] != matrix[i][j])
                 return -1;
    }
    return 0;
}

vector<vector<double>                                    >
multiply(vector<vector<double>        >        matrix        ,
vector<vector<double> > mat)
{
    vector<vector<double> > result;
    result=initRank(mat.size());
    for(int i=0; i<matrix.size(); i++)
   {
            for(int j=0; j<mat[i].size(); j++)
            {
               result[i][j]=0;
               for(int k=0; k<matrix[i].size(); k++)
               result[i][j]=result[i][j]+(matrix[i][k]                *
mat[k][j]);
            }
    }
    return result;
}
```

```
vector<vector<double>                                    >
createRankMatrix(vector<vector<double>                   >
transitionMatrix , vector<vector<double> > vect)
{
    int count=0;
    vector<vector<double> > rank;
    vector<vector<double> > res;
    rank=initRank(vect.size());
    res=multiply(transitionMatrix , vect);
    rank=multiply(transitionMatrix , res);

    while(equalsMatrix(rank , res)!=0)
    {
            res=multiply(transitionMatrix , rank);
            rank=multiply(transitionMatrix , res);
    }
    return rank;
}

int main()
{
   int n=0;
    vector<vector<double> > transitionMatrix;
    vector<vector<double> > vect;
    vector<vector<double> > rank;
    cout<< "Enter no. of nodes : ";
    cin >> n;
    /*------- CREATE VECTOR AND TRANSITION MATRIX
FOR THE GIVEN GRAPH---------*/
    vect = createVector(n);
    rank = initRank(n);
```

```
   transitionMatrix = createTransitionMatrix(n);
    /*----INITIALISE TRANSITION MATRIX BASED ON THE
OUT-DEGREE OF THE NODES---*/
    transitionMatrix=initTransitionMatrix(transitionMatri
x);
    /*----INITIALISE TRANSITION MATRIX BASED ON THE
OUT-DEGREE OF THE NODES---*/
    rank=createRankMatrix(transitionMatrix , vect);
   /*--------------------PRINT THE PAGE RANK MATRIX------
-------------------*/
    cout<< "The Page Rank Vector is : " << endl;
    for(int i=0; i < rank.size(); i++)
    {
       for(int j=0; j<rank[i].size(); j++)
          cout << rank[i][j] << "  ";
       cout << endl;
    }
    getch();
   return 0;
}
```

2. Code for finding the page rank considering the damping factor in C

```
#include<stdio.h>
#include<conio.h>
int main()
{   int m[10],n,a[10],i;
   float d,d2,b,c,pr[10],x[10],s;
   printf("Enter the value of damping factor d between 0
to 1 :-- ");
   scanf("%f",&d);
```

```
d2= 1-d;
printf("Enter the no. of outgoing pages of the page whose page rank is to be calculated :--\n");
scanf("%d",&n);
printf("Enter the pages:--\n");
for (i=0;i<n;i++)
{scanf("%d", &a[i]);}
for (i=0;i<n;i++)
{printf("Enter the page rank of page %d :-- ", a[i]);
scanf("%f",&pr[i]);}
for (i=0;i<n;i++)
{
printf("Enter the outlink pages of page %d :-- ", a[i]);
scanf("%d",&m[i]);}
for (i=0;i<n;i++)
{x[i]=pr[i]/m[i];
s=s+x[i];}
printf("Summation is :-- %f\n",s);
b= s*d;
c=d2+b;
printf("Page rank is :-- %f",c);
getch();
return 0;
}
```

3. Code for storing the transition matrix into sparse matrix format in C

```
#include<stdio.h>
#include<conio.h>
int main()
{
```

```
int m,n,s=0,i,j;
float A[10][10],B[10][3];
printf("\nEnter the order m x n of the sparse matrix\n");
scanf("%d%d",&m,&n);
printf("\nEnter the elements in the sparse matrix(mostly
zeroes)\n");
for(i=0;i<m;i++)
{
 for(j=0;j<n;j++)
 {
  printf("\n%d row and %d column:   ",i,j);
  scanf("%f",&A[i][j]);
 }
}
printf("The given matrix is:\n");
for(i=0;i<m;i++)
{
 for(j=0;j<n;j++)
 {
  printf("%f ",A[i][j]);
 }
 printf("\n");
}
for(i=0;i<m;i++)
{
 for(j=0;j<n;j++)
 {
  if(A[i][j]!=0)
  {
   B[s][0]=A[i][j];
   B[s][1]=i;
   B[s][2]=j;
```

```
   s++;
  }
 }
}
 printf("\nThe sparse matrix is given by");
 printf("\n");
 for(i=0;i<s;i++)
 {
  for(j=0;j<3;j++)
  {
   printf("%f ",B[i][j]);
  }
  printf("\n");
 }
 getch();
return 0;}
```

4. Code for finding page rank from sparse matrix in C

```
#include<stdio.h>
#include<stdlib.h>
 #include<conio.h>
 float a[17][17]={{0,0,1,0.5},
 {0.33,0,0,0},{0.33,0.5,0,0.5},{0.33,0.5,0,0} };

 float rank[17]={1,1,1,1};
 float page_rank[17];
 float d=0.85;
 float d_minus=0.15;
 int i,j;
 float s[290][3];
void copy() {
```

```
        for(i=0;i<17;i++)
        rank[i]=page_rank[i]; }
void create_sparse()
{
int k,x;
x=0;k=1;
 for(i=0;i<17;i++)
 for(j=0;j<17;j++)
 if(a[i][j]!=0)
  x++;
s[0][0]=i;
 s[0][1]=i;
 s[0][2]=x;
for(i;i<17;i++)
 for(j=0;j<17;j++)
  if(a[i][j]!=0)
{s[k][0]=i;
s[k][1]=j;
s[k][2]=a[i][j];
k++;}
for(i=0;i<k;i++) {
for(j=0;j<3;j++)
{  printf (" %f ",s[i][i]);
} printf(" \n");
}
}

void multiply_sparse(){
     int k=1,cnt=0;
      int t;
      do{
         for(i=0;i<17;i++)
```

```
{
page_rank[i]=0;
 for(k=1;k<=s[0][2];k++) {


if(s[k][0]=i)
{ t=s[k][1];
 page_rank[i]+=s[k][2]*rank[t];}
  else if(s[k][0]>i)
   break; }
}
cnt++;
 for(i=0;i<17;i++)
  page_rank[i]=(d*page_rank[i])+d_minus;
  copy(); }while(cnt<40);

for(i=0;i<17;i++)
 printf(" \n%f",page_rank[i]);
}
 main() {
create_sparse();
multiply_sparse();
system("PAUSE"); }
```

8. REFERENCES :

1. http://en.wikipedia.org/wiki/PageRank

2. http://pr.efactory.de/e-pagerank-algorithm.shtml

3.http://www.math.cornell.edu/~mec/Winter2009/RalucaRemus/Lecture3/lecture3.html

4. Role of Ranking Algorithms for Information Retrieval by Laxmi Choudhary and Bhawani Shankar Burdak

5. Deeper Inside PageRank by Amy N. Langville and Carl D. Meyer

6. A Relation-Based Page Rank Algorithm for
Semantic Web Search Engines by Fabrizio Lamberti, Member, IEEE, Andrea Sanna, and Claudio Demartini, Member, IEEE

7. The Anatomy of a Large-Scale Hypertextual
Web Search Engine by Sergey Brin and Lawrence Page

8. The Page Rank citation Ranking: Bringing order to the web

9. http://infolab.stanford.edu/~backrub/google.html

10. http://en.wikipedia.org/wiki/Web_search_engine

11.https://software.intel.com/sites/products/documentation/hpc/mkl/mklman/GUID-9FCEB1C4-670D-4738-81D2-F378013412B0.htm

12. http://web.lib.aalto.fi/en/ecourse/?ecid=2&pid=146