Tackling PageRank with R
by
Erica Christenson & Sadhana Nathan

I. Introduction to PageRank

PageRank is a method developed by Larry Page and Sergey Brin at Stanford University that uses the link structure of the web to rank the importance of web pages, and assigns numeric values to represent their importance. By using the link structure, if page A has more back links (other pages that contain a link to page A) than page B, page A is considered more important than page B (see Figure 1). In an article called "Google's PageRank Explained", Phil Craven describes that, "Google figures that when one page links to another page, it is effectively casting a vote for the other page. The more votes that are cast for a page, the more important the page must be" [Cra03].

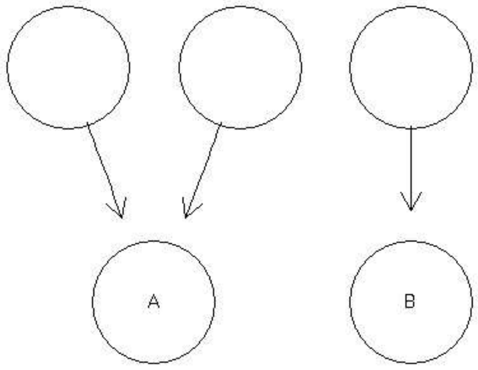

Figure 1.

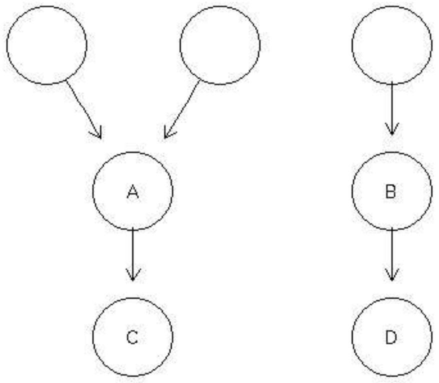

Figure 2.

While PageRank is concerned with the number of back links a page has, PageRank also takes into consideration the rankings of the pages that contain back links. So a higher ranked page bumps up the ranking of sites that it links to more so than a page with a lower ranking. For example, if page C and page D each have one back link from page A and page B, respectively, and page A has a higher rank than page B, then page C will rank higher than page D (see Figure 2). With his casting votes metaphor, Phil Craven points out that, "The importance of the page that is casting the vote determines how important the vote itself is" [Cra03].

II. Our approach to PageRank

a. Data Collection

Data collection begins with a set of initial URIs to parse. The slashes at the end of the set of initial URIs are removed (if there are any). This is done to ensure that the end results will consist of completely unique URIs instead of similar versions of the same URI. Before parsing of a URI begins, the set of initial URIs is checked to make sure it is nonempty. If it is nonempty, the function begins parsing the initial URIs in the order they are given in the vector, and the set is also put into a list called toBeProcessed. Later, the forward links collected from all the different URIs will also be added onto this toBeProcessed list.

The inital URIs are the first to be parsed because the function is not traditionally recursive, and instead links are parsed on a "first come first serve basis." So the forward links retrieved from parsing the first initial URI (call it A) is attached to the end of the list called toBeProcessed. If there is more than one inital URI this is the next link to be processed (call it B), and B’s forward links will be added to the end of toBeProcessed so that B's forward links succeed A's forward links. This process continues until all of the initial URIs have been exhausted, and then A's forward links will be processed in the order in which they were retrieved, followed by B's forward links, and so on. The function halts when the list toBeProcessed has been exhausted or when the specified maximum limit has been reached.

During data collection, only one URI is parsed at a time. Every time a new URI begins the parsing process, it is checked to make sure it is valid. If the URI is not valid, the URI is ignored and the function begins parsing the next URI. If the URI is valid, it is parsed and the forward links on the page are obtained. From this collection of forward links, links to email addresses are identified and removed from the collection if the email argument is FALSE. The collection of forward links is then stored in a named list where the name of each element in the list is the URI from which the forward links were obtained and the contents of each element is a character vector containing the forward links. This list is used as the output of the function if the extend argument is FALSE. Otherwise, the list is modified in a way that will be explained later.

The set of forward links obtained from parsing a page cannot be directly stored into the toBeProcessed list as potential URIs to process because the set contains internal links (identified by hash marks), non-html pages (such as pages with extensions .pdf, .zip, .exe), URIs outside the specified domain(s), and relative links (which are not fully qualified). For this reason, internal links and all non-html pages are removed from the set of potential URIs to process. Then, the host information from the URI that the forward links were acquired from is used to make completely qualified URIs out of all relative links in the set of potential URIs to process. If the domain argument is TRUE, the host information from the initial URIs is used as a way to restrict the collection of potential URIs to certain domains. Following this, the URI that has just been parsed to obtain its forward links is removed from the list toBeProcessed to ensure that it is not parsed again. Next, the set of potential URIs is checked against the URIs contained in toBeProcessed and also against all the links in the list that will be used as the output. This guarantees that the potential links have not already been parsed and are not already in the toBeProcessed list. Finally, the set of potential links is added to the list toBeProcessed. In the case that the extend argument is TRUE, further processing is done to make completely qualified URIs out of the forward links in the named list that will be used as the output to the function. This is done by going through the forward links of each named element in the list, checking to see if it is fully qualified, and if it is not, using the host information from the name of the list it comes from to make a fully qualified URI.

As mentioned above, the data collection function *catalogLinks* produces a named list where each named element is a URI that has been processed and its contents are the forward links retrieved from the particular site. While this list structure served well to collect the data, the data must be manipulated in order to analyze it further. Thus, the function *connectivityMatrix* is used to manipulate the data into a different structure that will make the data easier to analyze.

b. Manipulating the Data

The *connectivityMatrix* function creates a square matrix with the number of rows and columns equal to the amount of unique links collected from traversing the URI(s) given to the function catalogLinks. As mentioned above, a maximum number of links to be processed can be set as an argument in the function *catalogLinks*. However, this does not determine the size of the connectivity matrix because this limit only represents the number of sites visited, but does not take into account the total number of forward links gathered at each of the visited sites. Thus, the dimension of the connectivity matrix will often be bigger than the limit set by the function *catalogLinks*.

The structure of the output of *catalogLinks* is used to determine which URIs link to each other. The matrix created by using the output is filled with zeros and ones where the ith row has a unique name (usually a fully qualified URI) corresponding to the jth column with the same name. A one in the ith row, jth column represents a forward link from the jth URI to the ith URI and also a back link from the ith URI to the jth URI. A zero means that no direct forward link exists from the jth URI to the ith URI. The matrix that is created is quite sparse, containing a majority of zeros.

The function *connectivityMatrix* can either use the output generated by *catalogLinks* to make a matrix or it can be given a set of URI(s). If it is given a set of URI(s) then *connectivityMatrix* passes those URIs to *catalogLinks* called within the function. In either case, the same structure is generated (a list where each named element is a URI that has been processed by *catalogLinks* and its contents are the forward links retrieved from that particular site). The function goes on to deal with the internal links. We remove these URIs because they only link within webpages, and they are not outgoing links. Next, the function creates fully qualified URIs out of the relative links to establish a unique identification for each URI. This is done by first checking each forward link to see if it qualifies as a relative URI. Then, for each forward link that is a relative link, the name of the list that the relative link originated from is used to determine a host that can be appended to the relative link. Once each URI has a unique identification, a unique set of URIs is taken from the entire collection of links and will be used to index the rows and columns of the matrix.

At this point, a square zero matrix is constructed with dimensions equivalent to the number of URIs in the unique set. The set of unique URIs is used as names for the rows and columns of the matrix. We use the structure from the output of *catalogLinks* as indicators for where ones should be placed in the matrix, signifying that a link exits between two websites. This is done by using the name of an element in the list as an index for the column and the contents of the character vector corresponding to the element in the list as an index for the rows of the matrix, and ones are placed accordingly.

The connectivity matrix created with the function *connectivityMatrix* organizes the data in a matrix structure which is necessary to determine the page rank of each collected URI. This structure of the connectivity matrix will next be used to figure out the transition probabilities.

c. Finding the Transition Probabilities

The function *transitionMatrix* uses the output from *connectivityMatrix* and replaces the zeros and ones with the transition probabilities for each element in the matrix.

Before continuing, some terms that will be used later must be explained. The sum of the kth column in the connectivity matrix is the outdegree of the URI corresponding to the kth column. The outdegree can be thought of as the number of sites the kth site links to. In contrast to this, the indegree is the sum of the kth row and represents the number of sites linking to the kth site, or the backlinks to the kth site. The value p is the fraction of time a link from a currently visited page will be followed, and 1-p is the fraction of time that an arbitrary URI will be followed [Cle02]. The function *transitionMatrix* uses .85 as the value for p.

TransitionMatrix begins by finding the outdegree for every URI in the connectivity matrix. If the outdegree of the jth URI (the URI corresponding to the jth column) is greater than zero, then every element in the jth column is multiplied by p, divided by the column sum and a factor delta is added to each element. Delta is defined as (1-p)/(total number of pages collected). If the outdegree is zero, then there are no outgoing links from that page. This means that there is an equal probability of staying at the jth URI and going to any other page in the collection of unique URIs. Thus, each element in a column that sums to zero is given a transition probability equal to 1/(total number of pages collected).

The function *transitionMatrix* creates a matrix filled with the transition probabilities of moving from one state to another or from a given page to any of the pages that it links to. These transition probabilities are calculated and the transition matrix is constructed through the function *pageRanker* which calls the function *transitionMatrix*. The final steps in figuring out the page ranks are described in the next section of the paper.

d. Obtaining page rank's

The transition matrix is passed to the *eigen* function to compute the page ranks. The eigenvector corresponding to the eigenvalue equal to one contains the page ranks for each website possessing a row and column in the connectivity matrix. The function *pageRanker* finds the transition probabilities with the function *transitionMatrix* that is called within it, and then calls the *eigen* function to finally compute the page ranks for all the links that are collected by the function *catalogLinks*. The output of *pageRanker* is a named vector that is the length of the dimension size of the inputted connectivity matrix, that is the number of unique URIs collected with *catalogLinks*. Each element corresponds to the page rank of one of the websites collected and stored in the connectivity matrix. The vector is ordered from the pages with the highest page ranking to the pages with the lowest page ranking. Thus, the pages are ordered from most important to least important. The name of each element in the vector corresponds to the name of the website that the page rank belongs to.

III. Examples and Results

a. PageRank by Example

As mentioned earlier, PageRank takes into account not only the number of back links into a page, but also the rank of the pages that link into a page. The following examples will expand on those presented earlier and help clarify how PageRank works.

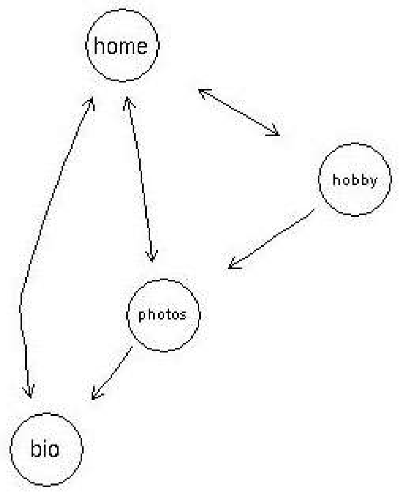

Figure 4.

In the above example of a website [HT03], *hobby* has the lowest page rank because it has only one back link from *home*, and *photos* and *bio* each are referred to by *home* as well (see Figure 4). Next in ranking is *photos*, because in addition to the back link from *home*, it also has a back link from *hobby*, which adds to its page rank. Even though *bio* has two back links just like *photos* does, it does not have the same page rank as *photos*. *Bio* has a higher page rank because *photos* holds more rank than *hobby,* and as a result has more rank to give to *bio* than *hobby* has to give to *photos*. Thus, *bio* accumulates more rank from its back links than *photos* does, and *bio* is bumped up above *photos*. Lastly, *home* holds the highest page rank because it gains ranking from all the other pages that refer to it. In this example, *home* equally distributes the degree of its page rank among the three other pages, making it so that no one page benefits more than another page from its forward link.

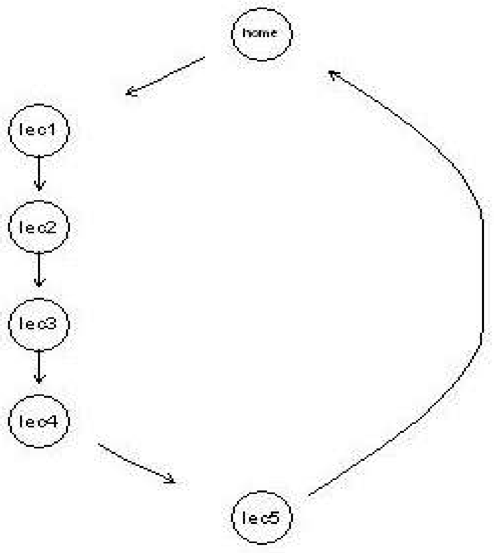

Figure 5.

Figure 5 is an example of a symmetric network of links [HT03]. *Home* links to *lecture 1*, and then each lecture links to the next successive lecture. The loop ends with *lecture 5* linking back to *home*. With this structure each page has an equal page rank. This is because every page is referenced by only one other page, and each page has the same degree of page rank to pass onto the page it links to.

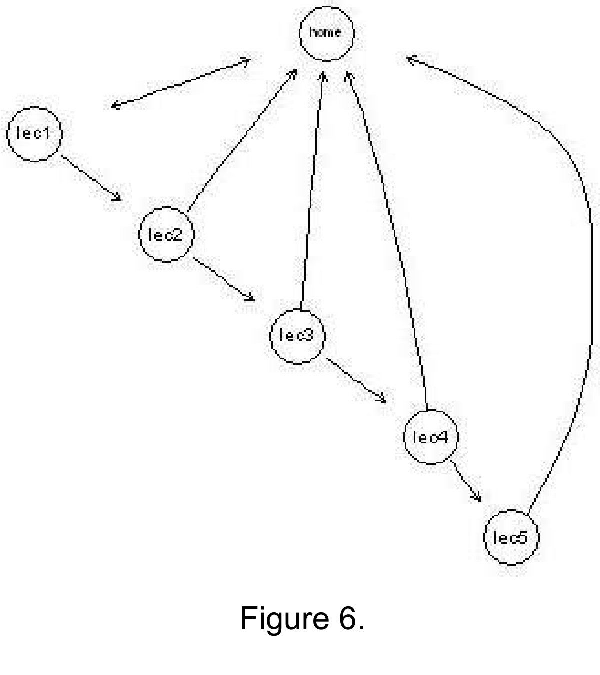

Figure 6.

This example (Figure 6) has a slight variation from the previous example [HT03]. Now, all the pages refer back to *home*. *Home* has the highest page rank because it has the greatest number of back links and thus all the pages give some of their page rank to *home*. Since *home* directly points to *lecture 1*, it has the next highest page rank. *Lecture 2* has a lower page rank than *lecture 1* because *lecture 1* has less page rank to pass onto *lecture 2* than *home* had to pass onto *lecture 1*. This continues to happen, and *lecture 5* ends up having the lowest page rank.

b. PageRank results

Since the web is so complex and some websites are so big, we decided to limit ourselves to the domain of the website we chose to study. Two-hundred-nine unique URIs were obtained from crawling ggobi.org. Table 1 lists the indegree, PageRank (obtained using the *pageRanker* function), and node color/number (corresponding to Figure 7) for the seven highest ranked sites of ggobi.org. Not surprising, the top sites all had very high indegrees which means they all had many other sites (at least 80) linking to them. In fact, the top seven sites had higher indegrees, and thus more backlinks, than any other sites within ggobi.org.

Table 1.

URI	Indegree	PageRank	**Node Color/Number**
http://www.ggobi.org/html/index.html	87	0.3433636	Red / 18
http://www.stack.nl/~dimitri/doxygen/index.html	81	0.3360240	Pink / 66
http://www.ggobi.org/html/modules.html	80	0.3258151	Blue / 61
http://www.ggobi.org/html/globals.html	80	0.3258151	Purple / 65
http://www.ggobi.org/html/annotated.html	80	0.3258151	Green / 62
http://www.ggobi.org/html/files.html	80	0.3258151	Orange / 63
http://www.ggobi.org/html/functions.html	80	0.3258151	Yellow / 64

Figure 7 is a graph of the links between the unique URIs in ggobi.org. Numbers were used in the graph because the URI corresponding to the node would not fit within the small space. The colored edges in the graph represent a link into one of the top seven sites. For example, a yellow edge from node 99 shows a link from the URI corresponding to node 99 to the site http://www.ggobi.org/html/functions.html which is represented in the graph by a yellow node numbered 64.

Figures 8-14 show the seven highest ranked sites individually and the sites that link into them. The colors and numbers representing the different sites are the same as those in Figure 7 and described in Table 1.

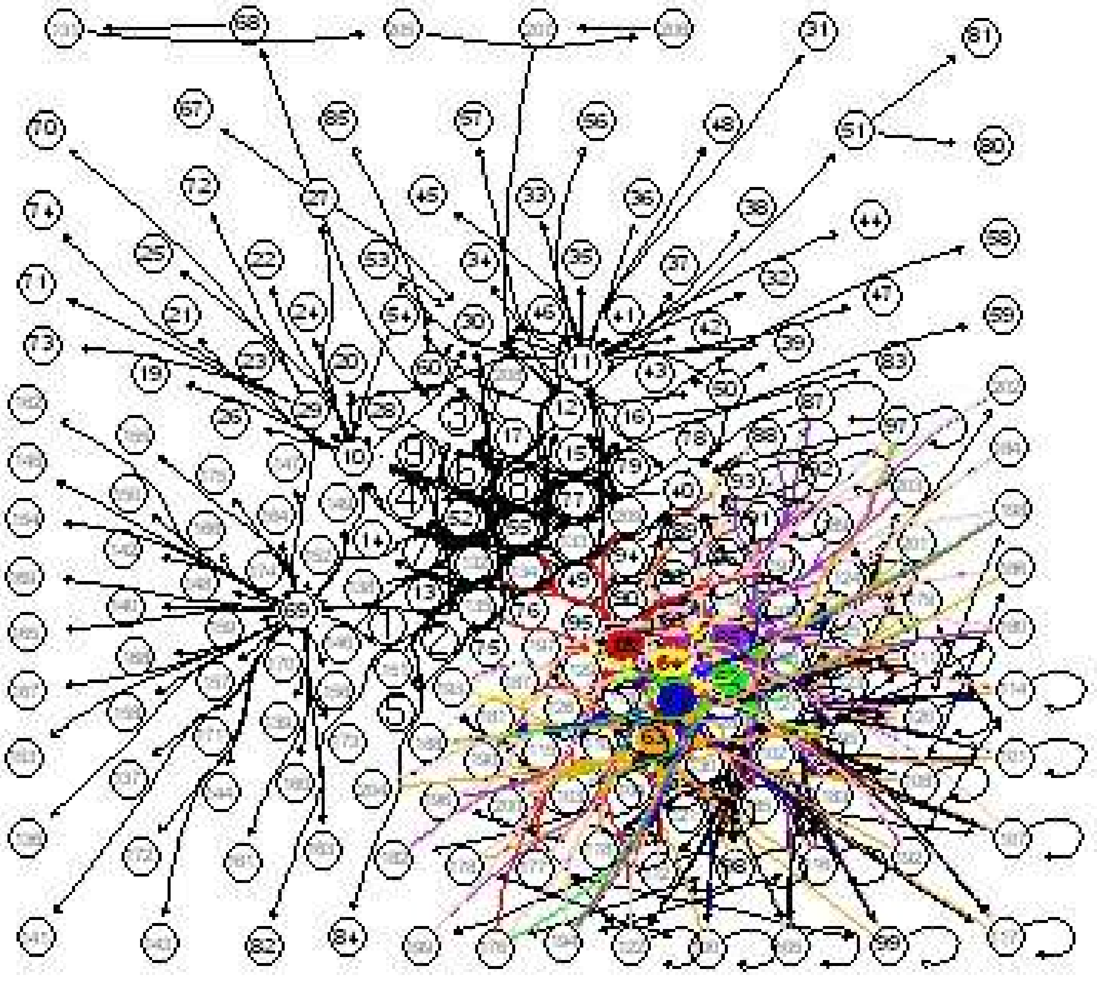

Figure 7.

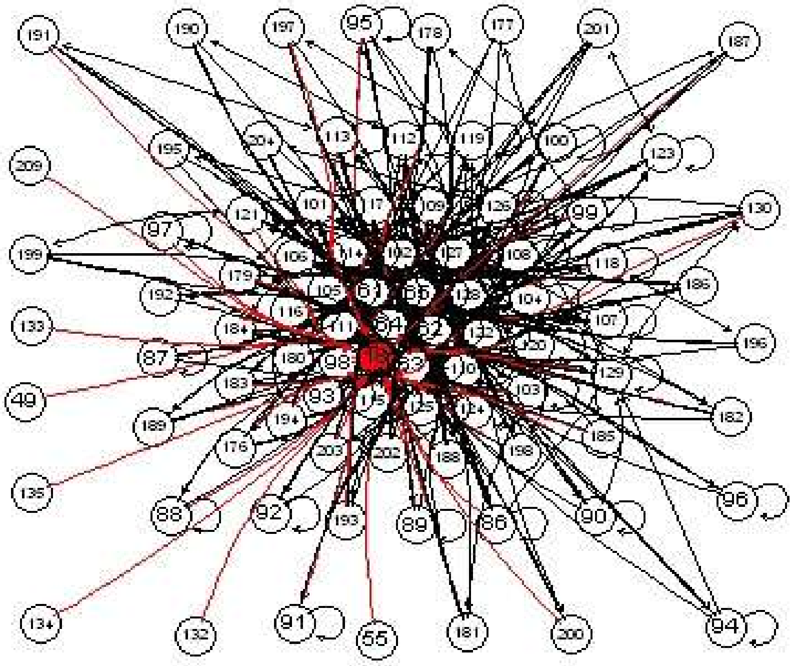
Figure 8.

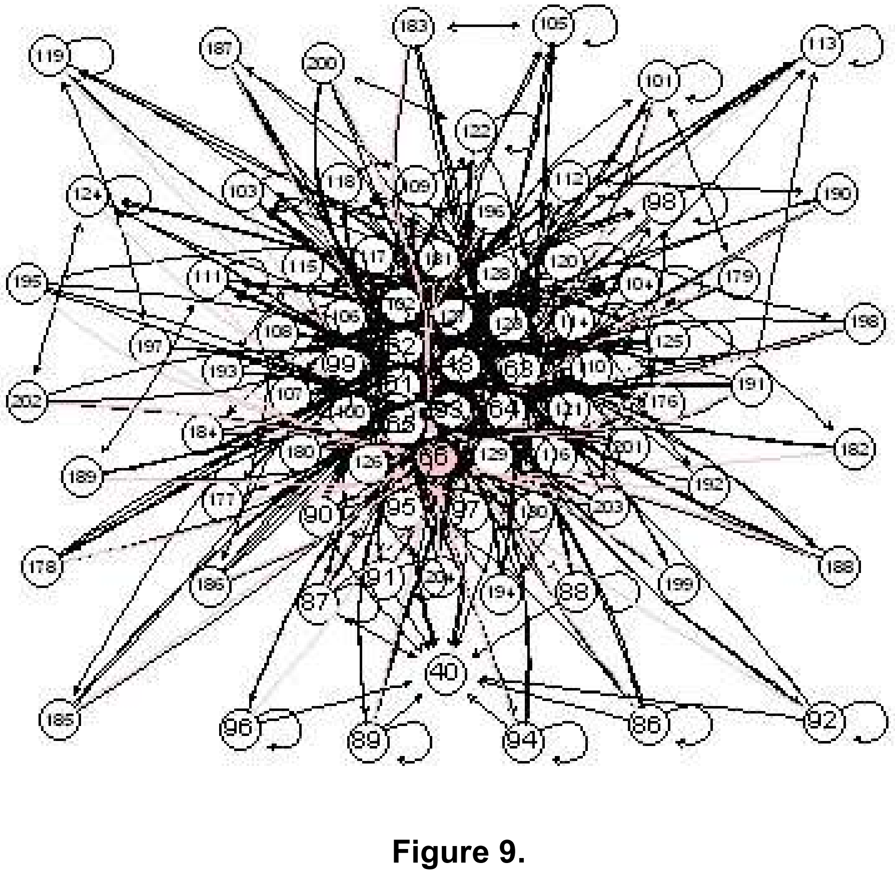
Figure 9.

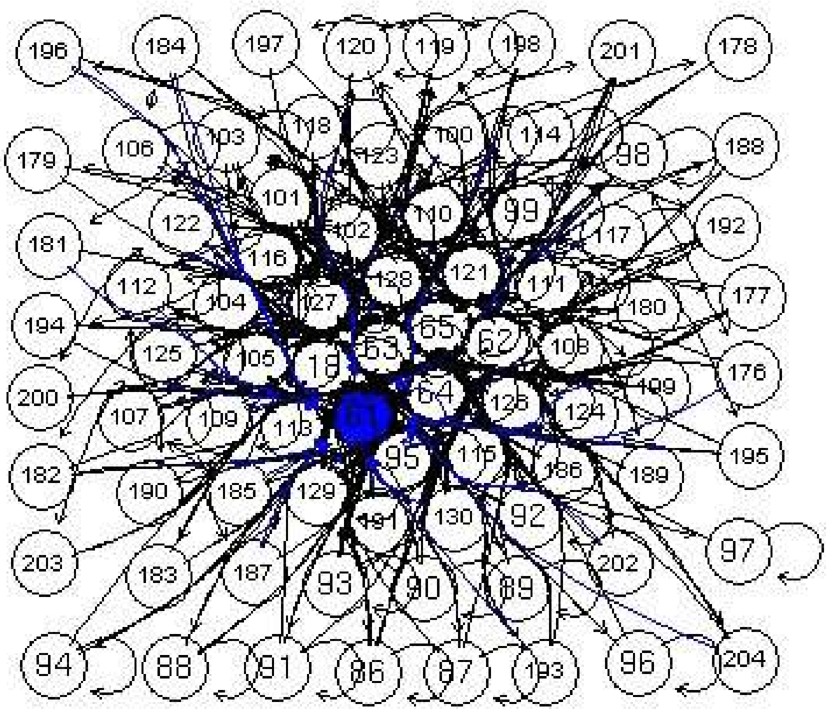

Figure 10.

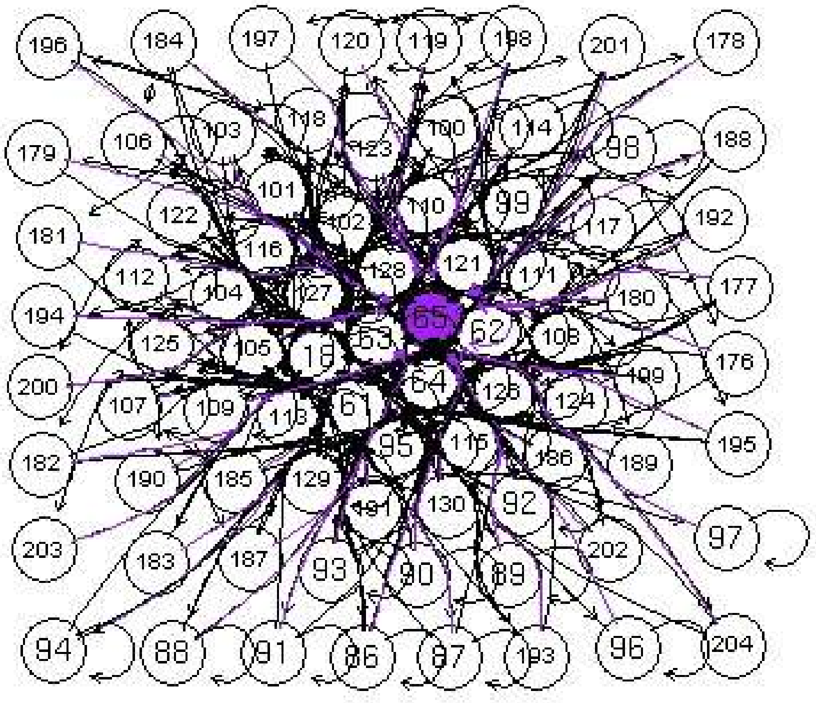

Figure 11.

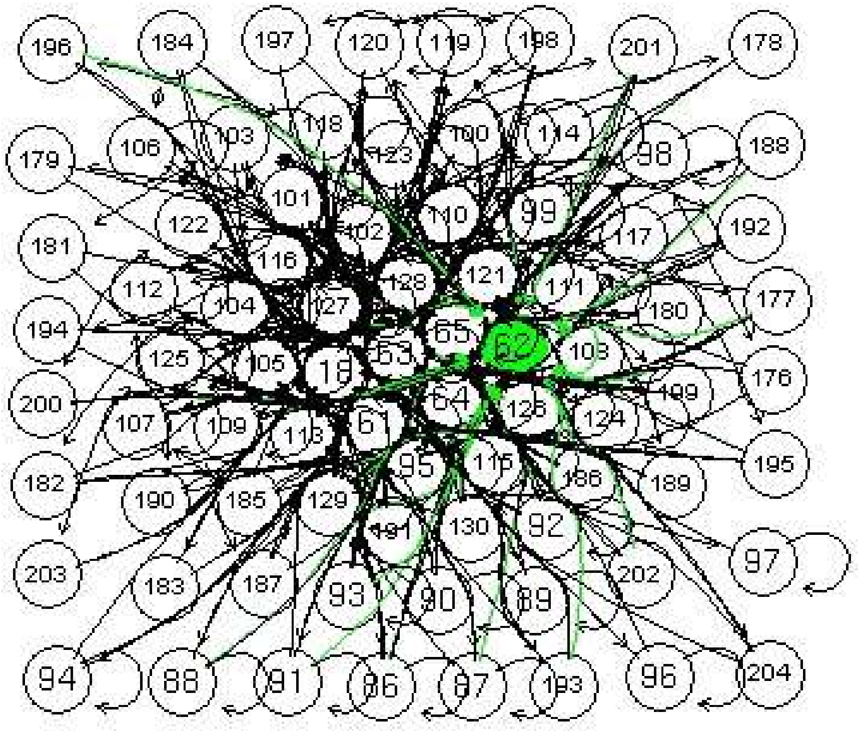

Figure 12.

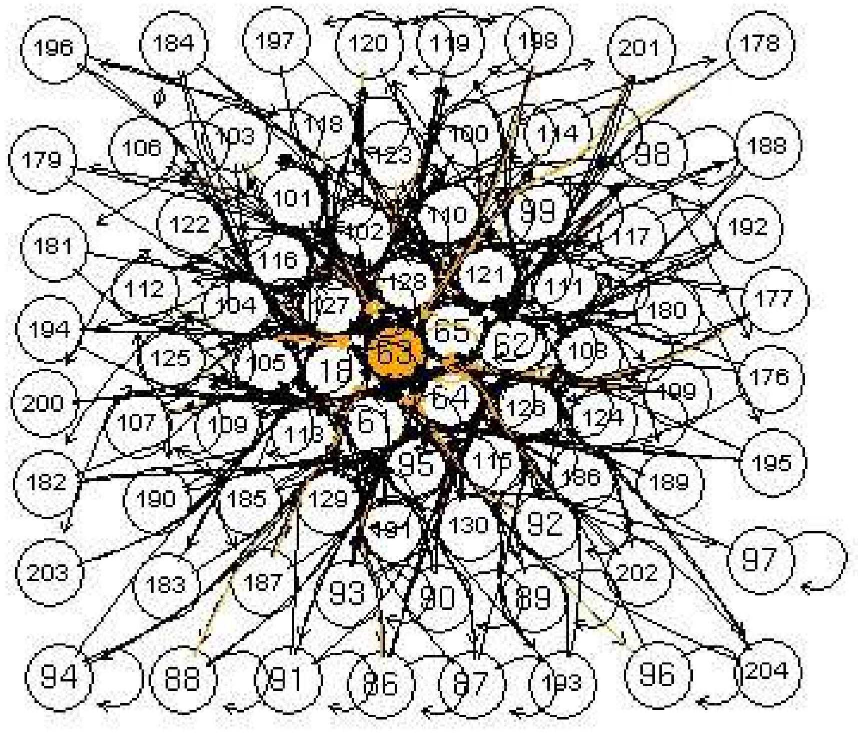

Figure 13.

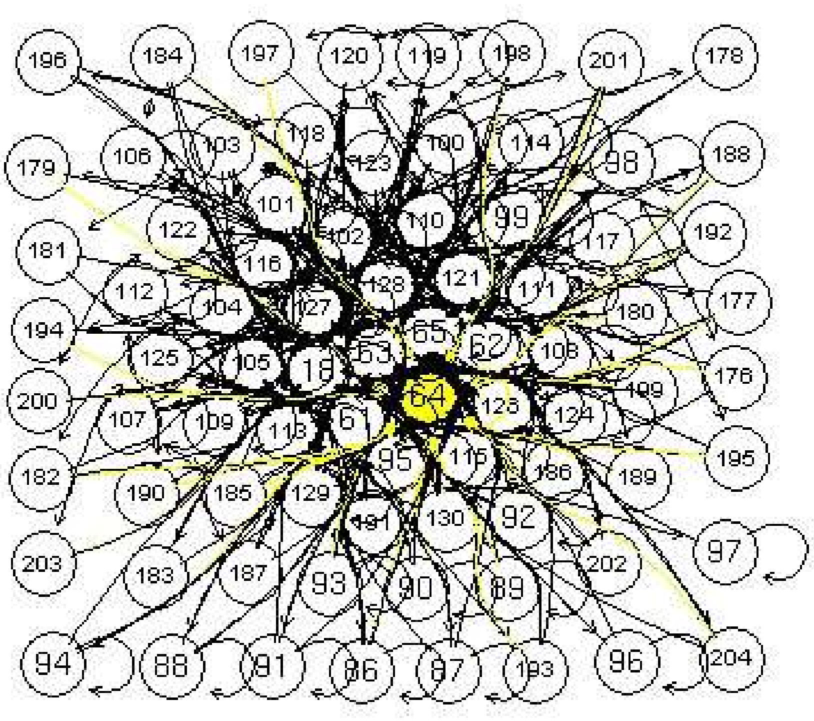

Figure 14.

IV. Problems We are Aware of

There is a chance that URIs which point to the same pages could all be processed. For example, http://www.stat.berkeley.edu and http://stat.berkeley.edu point to the same page in a browser, but because they are syntactically different, the function *catalogLinks* can not detect it and eliminate one from being processed.

Another problem is that the number of unique URIs crawled was limited by the amount of memory needed to compute the eigenvectors and eigenvalues with the *pageRanker* function. This posed a problem because we were not able to crawl the entire extent of the World Wide Web. We had to limit the process set by a maximum in the function *catalogLinks*. By doing this, we were able to get a good idea of the actual page ranks that a site would receive, but they are not entirely accurate.

V. Things we learned and challenges that we faced

The greatest challenge we faced was determining how to go about traversing the web. Using a traditionally recursive method results in traversing down a path until an endpoint is reached. Since we had to limit the number of total links that would be processed we were unsure whether the function would ever return to complete parsing the forward links it encountered early in the process. A method we devised traversed by levels. This ensured that forward links were processed in the order in which they were retrieved. Thus, our method allowed us to get a better view of the connections between websites. In the following examples we demonstrate the benefits of the method that we chose to use.

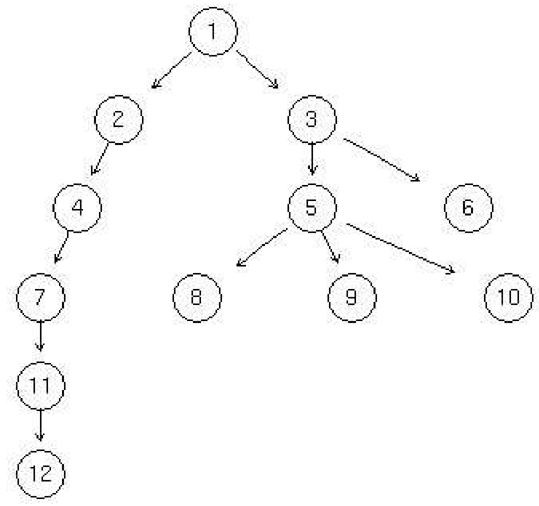

Figure 15.

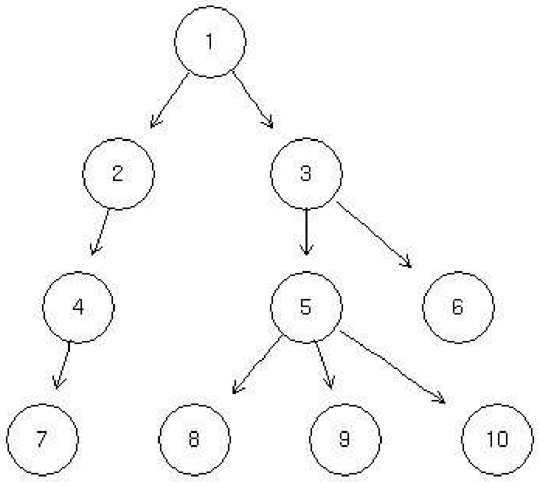

Figure 16.

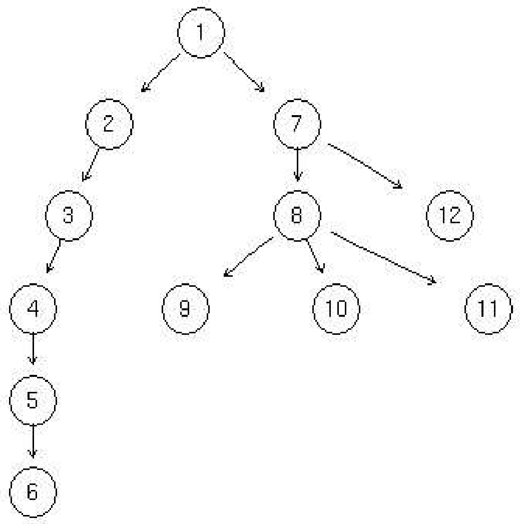

Figure 17.

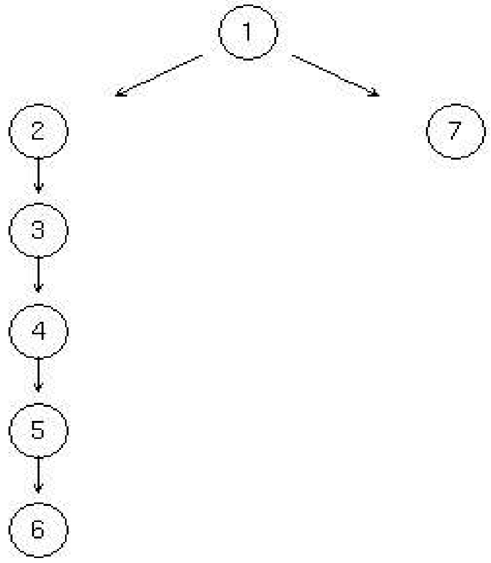

Figure 18.

Figure 15 illustrates the order, according to the numbers in each node, that a website would be traversed using our non-traditionally recursive method, and Figure 17 shows the traversing order using the recursive method. Figure 16 shows how much of the website is traversed if the maximum number of URIs to visit is set to 5 using our non-traditionally recursive method. As shown by the graphs, the method that we used enables us to see a cohesive portion of the connections within the website. This is an important aspect because we were not able to traverse the entire extent of the web, so if a traditionally recursive method were used, as seen in Figure18, the left most portion would be traveled down leaving the web of connections stemming from node 7 unknown about. Figure 18 reveals the loss of information which occurs from traversing down in a traditional manner, and how the complete picture is lost resulting in an interrupted representation of how the pages are connected.

Processing order presented another challenge as we were constructing the function *catalogLinks*. For example, when the email argument is FALSE, email addresses need to be removed before storing the forward links of a given page otherwise unnecessary processing would need to be done to the final output. Also, internal links had to be removed from the list of potential links to process before completely qualified URIs were formed because the helper function *hashURI* was not able to handle completely qualified URIs. Through trial and error, we ended up discovering a proficient way to process the URIs.

It was a painstaking process to maximize efficiency and make sure that the process was accurately done. There were many occasions when we found that it would be easier to work with multiple lists of the links collected, but we had to think about using one list for several purposes because of the memory taken up by storing other versions of the list. In the beginning of this project, we stored everything in different variables instead of just modifying the variables we already had. For example, instead of subtracting the link that was recently parsed from the list toBeProcessed (as explained above), we were storing a subset of the toBeProcessed list in an entirely new variable. This was unnecessary because instead of making a dynamic list to keep track of the links that were going to be processed, we created two lists that were almost identical to each other. We did this because we felt it was easier to keep track of the complex process and we had also never encountered memory storage problems in our past experiences with R. Eventually, we discovered the practical aspects of using a dynamic list, and began modifying previously stored objects instead of creating new ones.

When we began this project we did not realize the immense variation between web pages. Making a generalized function that could handle all the different cases that we might come upon was difficult because we could not foresee all the different possibilities that might arise. For instance, we never thought that every forward link collected from a URI could potentially be thrown out by the various ways we filter the retrieved links. This resulted in an empty vector being passed to functions that needed a vector to have content. As a result, the structure of the vector was changed into an empty list which completely paralyzed our function, and was yet another improvement we had to make to the function.

VI. Conclusion

Our research was an attempt to replicate the way that Google takes the structure of the web and ranks pages according to their positions within the web. We did it on a very small scale compared to Google in order to learn how the process and algorithm worked to compute the page ranks. Even though it was on a very small scale, we learned a lot about all the cases Google deals with in order to obtain these page ranks such as badly formed html, and the filtering process necessary to obtain the URIs of interest. We tried to remedy all the problems we ran into, but there were just too many and not enough time.

Needless to say, now that we know everything that Google deals with, we have come to respect Google and all the foresight its creators had in dealing with the complexity of the growing web.

References

[Cle02] Cleve Moler. The World's Largest Matrix Computation: Google's PageRank is an eigenvector of a matrix of order 2.7 billion. http://www.mathworks.nl/company/newsletters/news_notes/clevescorner/oct02_cleve.html.

[Cra03] Phil Craven. Google's PageRank explained and how to make the most of it. http://www.webworkshop.net/pagerank.html.

[HT03] Desmond J. Higham and Alan Taylor. The Sleekest Link Algorithm. 28 August 2003.