Experiments with PageRank

Student: Peter Clayton
Supervisor: Stefan Dantchev

Submitted as part of the degree of Computer Science to the Board of Examiners in the Department of Computer Science, Durham University.

Abstract –

Context: Sorting linked documents by a popularity-first approach has been adopted successfully using the PageRank algorithm first proposed in 1998 by Lawrence Page and Sergey Brin, co-founders of Google. Besides Internet search engines, the PageRank algorithm has also been used for ranking citations in academic papers. The algorithm’s results uncovered some relatively unknown journals that were cited abundantly and described as ‘scientific gems’.

Aims: To analyse and develop existing PageRank algorithms, to create a realistic model of a web surfer, compare effectiveness of the algorithms in terms of running time and space, and extend existing ranking methods using intelligent surfer algorithms.

Method: To analyse in depth the Java-based algorithms using various input in the form of directed graphs. The structure of test graphs, as input, will be compared to the structure of the Interweb. Examining how fast each algorithm will run with relevant comparisons, and effects of varying input size and other parameters are also incorporated into the investigation.

Results: The complexity of ‘summation’, ‘simple matrix’, and ‘back button’ driven algorithms are less than that of ‘eigenvalue’ driven types. However, the latter is considered the most accurate, time consuming and space consuming. The former algorithms run faster (summation being the fastest), but at a cost of accuracy. The back button algorithm can be modified and incorporated into existing algorithms to produce reasonably accurate results in less time than its eigenvalue counterpart. Where input $G = (V, E)$ and $n = |V|$, algorithms using matrices (no greater than 6200 pages) have running times of $O(n^3)$.

Conclusions: Although back button driven algorithms appear to be favourable for $|V| < 6200$, this may not be the case when cardinality is approximately 8 billion (a similar size to that of the Interweb). Test graphs of immense size are difficult to manipulate on general purpose workstations.

Keywords – PageRank algorithm, link analysis, graph structure of the Web, random surfer, Markov chains, back buttons, intelligent surfer

I. INTRODUCTION

This paper focuses on the famous PageRank algorithm first conceived in an article “The anatomy of a large-scale hypertextual Web search engine” by Lawrence Page and Sergey Brin [1]. By 1998 the Stanford University PhD candidates became the co-founders and co-owners of Google, one of the world’s most popular Internet search engines. Brin and Page successfully created an algorithm which would provide an unbiased list of linked documents based upon their popularity. The uniqueness of this particular algorithm at the time was due to the way popularity of any web page could be calculated without any influence by humans – that is to say ranking was calculated using nothing but mathematics. By analysing data from the underlying linked structure of the web, each page could be denoted a rank value which would determine its position in a list of search results.
This algorithm was not only considered useful for ordering Internet search results, but also for determining whether a page was trustworthy [15]. For example, URLs on the www.durham.ac.uk homepage (having a high Page Rank value) will have a very low chance of taking a web user to a malicious web site.

The algorithm was also later seen in applications not only concerning linked web pages, but research literature. Citations behaved as links to journals and other academic papers [5]. If enough citations were collected PageRank had the necessary link data to sort these papers, often revealing unexpected but highly valuable references [3]. As Garfield mentions “[Ranked] citation data may be useful as quantitative indicators of the impact of particular publications on the literature” [10] in his review on growing research literature.

A. Concepts of PageRank

The classic Google PageRank algorithm uses the concept of a random web surfer model traversing a given directed graph. The amount of visits the random surfer makes to any page is recorded. After a period of time (set by a damping factor, $\alpha$) the surfer stops traversing momentarily and selects a random page in the graph. Traversal then begins once again from this new position in the graph. The percentage of time spent on each page is the rank of that page (in Google, the percentages are then categorised into integer values from 0 to 10).

In the real world, Google does not entirely rely on the PageRank algorithm alone – it employs other algorithms searching for anchor text and proximity information too [1, 17]. It must be noted that PageRank is purely based on link data whilst anchor text, using keywords to describe a web page, relies on word matching and categorising. Besides the link data aspects, there are important human-related factors which make a page attain high popularity. This may include the type or content of the page, how frequently it becomes updated, the design quality of the page, and the way it was designed.

Section III of this paper will only examine the mathematical implementation of the PageRank algorithms as this is the main subject of study.

B. Structure of the Web

Ranking algorithms like PageRank are best tested on real graphs, such as the Interweb (see Section II). During their research, Brin and Page used very large portions of the World Wide Web to test their algorithm [1]. The structure of the web is not randomised, but a very large-scale, heterogeneous, highly engineered, complex system [18]. The human factor, which inevitably has some effect on the underlying structure of the web, should not be dismissed entirely when examining algorithms which output data solely based on the structure of the input.

C. Surfer Models

Two main surfer models will be discussed in this paper, namely the ‘random surfer’ and ‘back button’. The former, a simple naïve surfer, traverses a graph in the same way as described in part A of this introduction. The latter, an intelligent surfer modelling the behaviour of humans, uses a memory stack to hold states [8]. Each state represents a position, or more specifically one page on the graph. For each traversal, the state is recorded (pushed) on to the stack. A backoff probability [8] then determines when the memory stack will be accessed (popped) to ‘go back’ to the page just visited by loading the previous state into memory. This
algorithm can be made to provide rank values for any given graph as will be explained later in this paper.

D. Project Aims

The project aims to investigate the set of PageRank algorithms by implementing several instances on fabricated data sets using the Java programming language. The data sets will be in the form of directed graphs – referred to as networks in this paper where appropriate, to minimise confusion with data charts. Vertices represent web pages, and edges (or outlinks) represent URLs. Approaches to calculate the Page Rank of these networks will involve using summation algorithms, matrix manipulations, resolving the eigenvalues of these matrices using linear algebra, and applying various types of back button algorithm. Comparing these back button algorithms to existing matrix-driven and summation algorithms in terms of running time, space, and accuracy of results will also be conducted.

This paper aims to analyse the running times of algorithms against size of input to conclude on the effectiveness of each type used. By analysing the spread of results from each, it will also be possible to experiment by varying numerical parameters and combining existing algorithms to attain further results.

II. RELATED WORK

Investigating the literature surrounding PageRank is crucial to enable not only background knowledge of its application, but methods of implementation and how one can analyse the algorithms’ output analytically.

One main influential book by Langville and Meyer [12] has been of great interest and value. They describe in detail the fundamental workings of PageRank and begin with a simple summation-type algorithm. The basic functionality of PageRank is encapsulated within this, but Langville and Meyer slowly transform it further into a complex, matrix-driven algorithm. The more challenging content is found at this point, when matrix manipulation is involved including deriving eigenvectors and eigenvalues. Nevertheless, the results of their simple tests can be compared to this paper’s findings for further analysis.

The data sets used in the book are very simple – directed graphs with enumerated vertices. Networks will be designed matching these data sets which can then be used for the algorithms created in this paper. The results from these algorithms can be then compared with those in the book to assess whether they have been produced correctly. The book illustrates the functionality of different algorithms on not only sparse sets of data, but also on dense data sets (see Section III for further detail).

Additionally, Langville and Meyer investigate enhancing algorithms by considering random surfers – a first simple attempt at modelling the surfing behaviour of humans. The idea behind this strategy is to traverse the given network randomly based on a teleportation matrix [12, p49]. In effect, this imitates a human traversing an arbitrary page. A uniform distribution is defined which develops into probability distributions.

Some outcomes of altering the parameters of algorithms are also revealed, such as varying the $\alpha$ factor [12, p47]. Analysis of this paper’s findings can be compared to these results. The book will be often referred to as it contains a large proportion of the formulae that is required for this investigation.

A newsletter article by Gallardo [9] represents the underlying principle of PageRank, showing the matrix-driven model and solving the eigenvalue problems relating to them.
The article reviews the structure of the web, showing that the vastness of the Internet is not so clear-cut. Gallardo describes tendril formations and disconnected components, which together create weakly connected graphs. As PageRank works best having stronger connected components, he reviews transition probabilities to all vertices as one method of resolving these issues, similar to the teleportation matrix principle. This random surfer method would appear to eliminate the need for web-graphs to be irreducible (strongly connected).

Brin and Page’s research [1] progressed to the creation of Google, which has been vastly cited in related fields, and has been the reference material for many PageRank related articles.

Brin and Page introduce the concept of the random surfer, which incidentally shares parallels to Markov chains. These are linked states such that future states can be reached via probabilistic processes – given present states, nothing can influence the future. At their time of writing, whether they were aware of Markov chains or not, they disregarded this view and seemed confident that the random surfer strategy worked best for large networks (millions of pages) as tested on their prototype search engine having a total of 518 million hyperlinks [1].

The initial idea that a random surfer would forwardly traverse links arbitrarily over a set period of time before ‘teleporting’ to a new random page is a little unrealistic. They presume the surfer cannot go back to a page once a particular link has been selected.

Brin and Page describe the benefits of using the PageRank algorithm due to its weighting properties. Pages considered of high rank status (having a high Page Rank) which have outlinks to other pages mean these pages will in turn inherit a fraction of this status. For example, www.microsoft.com linking to John Smith’s personal web site means the latter site results in obtaining higher ranking values than if it were linked by say, www.not-a-popular-website.com.

A paper on back buttons by Fagin [8] analyses the modification of the existing Markov chain within PageRank to include a function which enables backoff processes [8]. Using a memory stack to store the position of the surfer and a backoff probability $\alpha$, it is possible to imitate revisiting a page. The likelihood of a backward step is dictated by $\alpha$, while the likelihood of a forward step is $(1 - \alpha)$. This more realistic surfing algorithm should be directly compared to a PageRank algorithm running the same data set. Experimenting with back buttons will be analysed later in this paper.

Broder’s article “unpublished” [2] describes the structure of the Interweb as a complex graph. This graph is constituted of strongly connected components, in-nodes, out-nodes and tendrils (see Figure 1). A randomly generated super-graph may not have such a distinguished structure as the Interweb. This is because the Interweb is created from many small intricate processes not determined by randomisation alone.

After analysing the web, it was concluded: “The probability that a node has in-degree $i$ is proportional to $1 / i^x$ for $x > 1$” [2]. After further study, it was agreed that $x = 2.1$ for macroscopic structures. Out-degrees were also calculated at $x = 2.72$. Comparisons can be made to the fabricated data sets discussed in the introduction. The information may only apply to macroscopic data structures, but we can find out to what extent this holds. Creating realistically sized data sets may be problematic due to processor and memory constraints. However, microscopic structures can be examined using the concepts from Broder’s article to perceive how accurately they resemble that of the Interweb.
Ding proposed a simpler solution to the PageRank algorithm which would not require the expensive use of matrices. Instead, the number of inlinks each page of a network possess represents its rank value [6, 7]. This would produce one of the fastest working algorithms available – however, it is hugely flawed. This type of algorithm would never work in the real world especially when used on the Interweb. Link farming [12, p52], the process of creating fake links to specific pages to force an increase in their popularity, would take hold. The amount of new software acting as preventative measures would seriously compromise the speed of this inlink-size algorithm. Google already have to implement modified algorithms to defend against link farming. Raghavan’s paper suggests that implementing such an inlink-size algorithm would not be feasible commenting that “there is very little correlation on the web graph between a node’s in-degree and its PageRank” [16]. For these reasons, this particular algorithm will not be further investigated.

III. Solution

It was proposed that equations from [12] would be used to create the summation and matrix-driven algorithms. A series of programs created in Java would then replicate them.

Initially, before any algorithm was studied, a simple Network and Page class was prepared. The Network class would house the main algorithm to be studied and utilise any parsers or other additional classes required (as in Figure 4 and Figure 6). The Page class was created to represent a single web page, having array lists which would retain URLs and inlinks. Methods were created to allow URLs and inlinks to be added freely. Importantly, a method to return these figures to the Network class was made. This class would then imitate the network by storing pages in an array.

A. Graph Generator and Reader

A graph generator was created that can produce a series of randomly generated networks. It can take parameter inputs of page cardinality and the total number of networks to output. The underlying structure of each generated network is completely random and no network will be identical to any other of the same size. Graph generator controllers were later devised which would automatically generate a series of networks of increasing size.

Some form of encoding syntax was needed when creating each network. A sequence of alphanumeric characters taken from the ASCII character set was chosen (see Table 1).
All encodings were written to a file which could then be accessed at any time by a file reader, the Parser. The benefit of using this encoding was it remained human readable – at least for small file sizes. An example of a network encoded file can be seen in Figure 2.

The GraphGenerator class was then later modified to better represent a macrostructure, like that of the web. Depending on sources, the average number of URLs a web page contains is said to be between 7 and 11 [15]. The generator was modified such that there would be a stronger likelihood of these URL amounts to appear than any other. Parameters included the percentage of pages between 0 and 6 URLs, 7 and 10 URLs, and 11 and \( n-1 \) URLs enabling control of the network’s structure. A network generation statistic was displayed for each generation to confirm this structure. An example can be seen in Figure 3.

The GraphGenerator class was then later modified to better represent a macrostructure, like that of the web. Depending on sources, the average number of URLs a web page contains is said to be between 7 and 11 [15]. The generator was modified such that there would be a stronger likelihood of these URL amounts to appear than any other. Parameters included the percentage of pages between 0 and 6 URLs, 7 and 10 URLs, and 11 and \( n-1 \) URLs enabling control of the network’s structure. A network generation statistic was displayed for each generation to confirm this structure. An example can be seen in Figure 3.

**Table 1. Graph Encoding Symbols.**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
<th>Valid successor(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>#</td>
<td>Create new page</td>
<td>0–9</td>
</tr>
<tr>
<td>*</td>
<td>URLs to follow</td>
<td>0–9</td>
</tr>
<tr>
<td>.</td>
<td>Page separator</td>
<td>0–9</td>
</tr>
<tr>
<td>0–9</td>
<td>Pages ( m : M \in {0, \ldots , n } )</td>
<td>0–9</td>
</tr>
</tbody>
</table>

*a* example of valid graph: \#1,2,#*0#2*1

*b* 0–9 numbers following this represent in links

---

**Figure 2.** Example of a test graph ‘graph2’: 7 hashes indicate 7 pages. Values following the hash indicate inlinks, and numbers following asterisks indicate URLs.

---

**Figure 3.** Statistics of a 6200 page network generation: Input parameters were 90.0% for 7 to 10 URLs, 9.5% for 0 to 6 URLs, and 0.5% for 11 to 6199 URLs. Due to the random creation of page amounts, exact proportions of URLs per page can never be attained, as seen.

**B. Summation Algorithm and Using Matrices**

A prototype build utilising a basic-form PageRank algorithm was first devised. The summation algorithm was implemented from Eq. (1) [12, p32].
Each page $P_i$ initially has a PageRank value $r$ of $\frac{1}{|P|}$ before entering the recursive procedure (indicated by $k$). For each page $P_i$, fractions of this initial value are then supplied to its outlinked pages based on its number of outlinks $|P_j|$. $B$ is the set of all pages $P_i$.

For the algorithm to be improved, the introduction of matrix manipulation must be considered using basic linear algebra. A Java class could have been designed for this purpose, but creating one would have been a project in itself. Therefore, it was significantly more feasible to import an existing matrix package into the project. A package called JAMA, developed by MathsWorks and the National Institute of Standards and Technology, was used to fulfil this purpose.

Replacing this summation problem using matrices should in effect produce a more efficient algorithm. Shorter code length, ease of use and faster output times were expected. Rank values for pages $P_i$ can be calculated from using one row vector, $\pi^T$, given an $n \times n$ matrix of $H_{ij} = \frac{1}{|P_j|}$. The following equation can then be made of use [12, p33]:

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

$T$ refers to the transpose of the vector and $k$ indicates the $k^{th}$-iteration. The row vector of the current iteration $\pi^{(k)T}$ is multiplied by $H$ whose elements are all fractions relating to each and every page in the network. This uses an iterative process where results should converge to a certain tolerance depending on the final value of $k$.

There is however a flaw with this iterative matrix algorithm. It is not stochastic. That is, not all of the row elements in $H$ sum to unity. This is the case when for example a page has no URLs – known as a dangling node [12, p34]. Hypothetically, a simple traversing surfer upon entering such a dangling node would have zero probability of entering any other page. As a result it becomes trapped. The PageRank algorithm must be designed such that the surfer is continuously moving to obtain accurate rank values. Clearly, modification of this iterative matrix algorithm needs to be made.

A stochasticity adjustment is fashioned from the existing matrix $H$. Any rows of zeros are replaced with $\frac{1}{n}$ which creates the stochastic matrix [12, p37]:

$$
S = H + a\left(\frac{1}{n}e^T\right)
$$

$e^T$ indicates a row vector holding $\frac{1}{n}$. Dangling node vectors (as $a$ above) are introduced here to change a row's elements to $\frac{1}{n}$ if the sum of all row elements is zero (the definition of a...
dangling node). ‘$a_i$’ becomes 1 if a dangling node exists, or 0 otherwise. This simple adjustment allows $S$ to be the transition probability matrix for a Markov chain [12, p37]. This means a simple traversing surfer upon entering a dangling node has $\frac{1}{n}$ probability of entering any other page, and so can keep on traversing. In effect, Eq. (3) provides every dangling node page with $(n - 1)$ URLs – one to every other page. In principle, this is the exact equivalent of randomly jumping to a new page.

A flaw in using Eq. (3) is that the simple surfer cannot randomly select any page after a certain amount of traversing like Google’s random surfer, as discussed in section II. A further adjustment to $S$ creates the Google matrix, which solves this problem. A real scalar $\alpha$ is introduced where $0 \leq \alpha \leq 1$ signifies the chance of the random surfer moving on to a new page such that [12, p37]:

$$G = \alpha S + (1 - \alpha) \frac{1}{n} e e^T$$

(4)

The Google matrix $G$ is formed. Alpha amount of the time, $S$ will be used to determine the next traversal move, while the rest of the time indicated by $(1 - \alpha)$, the $n \times n$ matrix of $\frac{1}{n}$ will be used. If the latter part is employed, this creates links to all other pages meaning the surfer can effectively teleport.

As $\alpha$ can be varied it can be set as a parameter. Brin and Page mention in their paper that this damping factor should be set at 0.85 [1]. Whether or not this relates to the chance of a human surfer spending 85% traversing the web and 15% visiting somewhere new, the algorithm itself strikes a balance between running time and accuracy of results with this particular value.

Although $G$ is a primitive stochastic matrix where any iteration will return a value greater than zero, it is completely dense – every element $G_{ij}$ holds some value. In terms of computer resources, this can become expensive for storing large networks. There is a solution to this problem whereby any element of $G$ can be calculated in terms of $H$ alone. This means the Google matrix is never created fully, saving space. This is possible by modifying the equation as follows [12, p38]:

$$G = \alpha S + (1 - \alpha) \frac{1}{n} e e^T$$

(5)

and the PageRank algorithm can now be changed to [12, p40]:

$$\pi^{(k+1)T} = \alpha \pi^{(k)T} H + (\alpha \pi^{(k)T} a + (1 - \alpha) \frac{1}{n} e^T)$$

(6)

where ‘$a$’ indicates the $n \times 1$ dangling node vector. There should be no difference in the results from Eq. (4) and Eq. (6) – the latter merely changes the way the computation is performed. In reference to Java, both methods may end up requiring the same amount of memory to operate due to the way Java allocates memory. This will be later investigated.
C. Using Eigenvalues

Extracting the eigenvalues of the Google matrix was the next procedure, such that the eigenvalues of \( G = \alpha S + (1 - \alpha) e v^T \) are \( \{1, \alpha \lambda_2, \alpha \lambda_3, ..., \alpha \lambda_n\} \) (\( v \) being the probability vector \( \gt 0 \)) [12, p45 to p46].

The objective is to obtain the rank value of each page stored in the rank vector \( \pi \). So far, the equations above have been iterative, whereby the more iterations that are performed the more convergence occurs – behaving like an asymptotic function as in Eq. (7):

\[
\pi^{(K+1)T} = \pi^{(K)T} G
\]

Ideally, only one iteration should render the ranks for all pages:

\[
\pi^{(K+1)T} = \pi^{(K)T}
\]

It would be advantageous if \( \pi^T = \pi^T G \) could be solved. Using eigenvector calculations, it is possible to immediately obtain the value of \( \pi \) through no use of iterations. In the case of Eq. (7) it would be as if \( K \) were equal to infinity obtaining the value with maximum convergence.

\[
\lambda \pi^T = \pi^T G
\]

As one of \( \lambda \) is already known equalling 1, it is possible to calculate \( \pi^T \) given the existing Google matrix \( G \) using the JAMA package. Firstly, eigenvectors have to be extracted. Selecting the eigenvalue decomposition method \( \text{getV()} \) produces the eigenvectors of \( G \) in a new matrix. This matrix is then inverted due to the eigenvector equation Eq. (9) being dominant left-handed [12, p39] rather than the more common right-handed.

The rank can now be calculated. The real parts of the eigenvalues are extracted, again using the JAMA package method \( \text{getRealEigenvalues()} \). This outputs an array of real numbers corresponding to the eigenvectors of \( G \). In this array, there will be one entry that is equal to 1, or close to it – for example 0.999999999994. The index of this array entry represents the index of the eigenvector matrix of \( G \) where the rank values can be found. The sum of the row’s elements of \( G \) provides the denominator of the rank fraction. Each element’s value in the same row acts as the enumerator of the rank fraction.

Figure 5 below illustrates this procedure in Java code, which prints the rank values according to this procedure on obtaining \( G \).
EigenvalueDecomposition e = new EigenvalueDecomposition(G);
eigV = e.getV(); //Answer in here
eigV = eigV.inverse(); //SOLVED!

getReal = e.getRealEigenvalues();

// Display PageRank Information...
// Find in the array of Reals which is closest value to 1.0
for (int x=0;x<numberOfPages;x++){
  if (Math.floor(getReal[x]/0.999999999999999) == 1){
    index = x;
    break;
  }
}

double[][] found = eigV.getArray();
for (int x=0;x<numberOfPages;x++){
  counter += found[index][x];
}
for (int y=0;y<numberOfPages;y++){
  System.out.println((found[index][y] / counter));
}

Figure 5. Eigenvalue decomposition of the Google matrix G into eigenvectors, obtaining eigenvalues, then extracting the reals. Once the index is found which has its element closest to value 1, the eigenvector matrix is accessed where the exact rank value can be calculated.

D. Using Back Buttons

The aim was to create a back button algorithm that could calculate Page Ranks given any network. The algorithm would traverse the network like a simple random surfer. If a dangling node was ever reached, instead of teleporting to a random page, the traversal would step back to the previously viewed page held in a memory stack. If the stack was empty, only then would it teleport. In a given percentage of the traversal time known as \( \beta \), the ‘back button’ would be initiated. Again, if the stack was ever empty a teleport would be incurred. \( \beta \) can be freely varied using a parameter so that evidence of how much the back button affects rank values can be studied further.

This traversal algorithm requires access to Page, Decoder, and LimitedStack classes (seen in Figure 6). The BackButton class decodes a given network file and simulates the network using an array of pages, Page[]. The LimitedStack is a circular buffer of a given size, set to 100 in this case, where every page traversal is recorded. The chance of the back button being used even a mere one hundred times in a row is very slim indeed, so this figure deemed an appropriate memory size to use for testing. A circular buffer was implemented because traversals can be in the millions – it would simply use up too many resources and slow the system down. For example, one 6200-page input used close to 40 million traversals during the testing phase of the project.

The actual Page Rank was deduced by an access-percentage approach. An array holding \( n \) counters was created to hold the number of times each page was accessed. On the completion of the back button traversal, the sum of all counters was used as the global denominator. Each page’s rank was then calculated by having its access counter divided by this global denominator.

To turn these fractions into the classic Page Rank values as used in the Google search engine, the highest value would indicate the most popular page – and classed as a ten. Any values within nine percent of this value would also get a Google score of ten. The next nine percent band below this would score nine, and so on until zero is reached. This same principle can be used on any ranking algorithm if required. For the purposes of studying rank values in
general, existing percentages are more than sufficient for this study and Google’s rank approach shall not be discussed in any further detail.

**E. Combining Back Buttons and Teleporting Random Surfer**

The last modification made to the back button algorithm was to incorporate teleportation. The PageRank damping factor $\alpha$ was included in conjunction with the existing factor $\beta$ (from section D). A certain percentage of the time the surfer will stop traversing forward and do one of two things: teleport to a completely random page in the network, or go back to the previously traversed page. Rules such as teleporting if the memory stack is empty still apply.

Incorporating the back button and random surfer better imitates the surfing behaviour of a human. For example, it is more likely that a human will surf the web and use the back button on a few occasions as well as deciding to visit a completely new page. In a technical sense, this combined algorithm will allow for much better coverage of the network than with back buttons alone. The better the coverage, the more chance that pages being visited will obtain a fair rank value.

There are some problems however. If $\alpha$ is too high, then too much teleportation will occur. Rank values will start to become evenly distributed, masking out the effect of the back button. This is because teleportation uses the fact that all pages have links to all other pages. Using teleportation extensively will in effect destroy the underlying structure of the network being analysed returning inaccurate results. Too little teleportation is also problematic – traversal may be confined to a small area of the network giving a high level of visits to only a few pages. The same theory applies with back buttons – too much and the ranks of pages may be overly reinforced. Not enough and other branches of the network may never be discovered. It is therefore essential to find a balance between $\alpha$ and $\beta$ values.

**F. Testing**

Part of the testing procedure involved checking any sub-programs made functioned correctly. This was achieved by running methods individually and cross-checking the results obtained by calculations worked out by hand. One important testing procedure was ensuring networks were generated correctly from the encodings (as in Figure 2). Simple networks were drawn out by hand, encoded and then fed into the program. Due to the nature of the BlueJ environment that was being used, it was possible to easily inspect the network’s field values for consistency, reducing the possibility of error in the final results.

The major part of the testing procedure was extracting results from the algorithms. Rank values were obtained from simple networks working up to large randomly generated ones. The distribution of ranks could also be analysed in graphical format. For earlier versions of the PageRank algorithm, analysing the time to convergence over the number of iterations would be part of the testing method for calculating running times.

Algorithm timing, being a large part of the testing procedure, required Java’s built-in method `System.currentTimeMillis()`. It was possible to record the time span of all algorithms to the nearest millisecond. Varying the type of algorithm, parameter value, size of network, and number of iterations all affected the running time.

Testing classes were designed so that several implementations of the algorithms could be achieved sequentially. Algorithms were fed various parameters where printouts of results for each parameter type could be thoroughly analysed. The final build after 16 revisions including testing procedure classes can be seen in Figure 6:
IV. RESULTS

A. Summation

The initial aim of the project was to obtain ranks of pages given any network using the simple equation from Eq. (1). A network having six pages (Figure 7) was input into the summation algorithm. An example of the results can be seen in Figure 8 which shows the progress of the rank values as the algorithm runs through each iteration.

These initial results prove plausible enough when comparing the two figures below. Pages 3, 5, and 4 have the highest ranks as values from the page cluster 0, 1, and 2 can only move one way (only one URL linking 2 to 4). Values from the 3, 4, and 5 cluster gain these values without having to lose any of their own. The difference eventually becomes so high that the lower cluster falls to nearly zero. This is a good example of what can happen if no back button or teleporting algorithm is in place.

The summation algorithm was heavily flawed despite positive initial results. When tested on other networks, particularly those with certain types of cycles, dangling nodes, and whose pages contain no inlinks were susceptible to have rank values tending to infinity. As no number of iterations could stabilise the rank value, better algorithms were then sought after.

![Graph generator and relation diagram](image1)

**Figure 6.** Java relation diagram: Extra testing classes and automatic graph generator as used in matrix-driven eigenvalue and back button algorithms.

![Test network](image2)

**Figure 7.** Test network: ‘graph1’.

![Summation algorithm results](image3)

**Figure 8.** Summation algorithm results from ‘graph1’.
B. Using Matrices

An algorithm was created using Eq. (2) found in section III of this paper to replicate the summation algorithm. Using the network from Figure 7, the output was identical to that of Figure 8. Like the summation algorithm, the flaw of non-stabilising values on some networks was still occurring by the presence of constant-size oscillations which would fail to resolve.

An algorithm utilising Eq. (3) was made that switched $H$ for a stochastic matrix $S$. The Google matrix was also implemented from Eq. (4). The output from these algorithms when using the network from Figure 7 shared the same basic shape as that seen in Figure 8. It was evident there was a slight change in the contours of the curves (compare Figure 8 to Figure 10). The fluctuations were less harsh than from the summation algorithm, and pages 0, 1, and 2 did not fall so sharply to zero.

When tested on other networks, oscillations were apparent but stabilised into well defined values for each page. An example of this can be seen in Figure 9.

![Figure 9](image)

**Figure 9.** Results from using Eq. (3) on a 7-page network: Some pages have identical values throughout the calculation. These are superimposed on existing lines.

The algorithm utilising the stochasticity adjustment provides well resolved results at seemingly low iterations. When testing other networks having higher page numbers (10 and over), the average number of iterations required for the values to stabilise was between 5 and 20. The Google PageRank algorithm can typically use higher values than this – around 50 to 60 [12, p47 to 48]. The exact number of iterations required for stabilisation however does depend in part on the value of $\alpha$, as will be examined later.

The next evolutionary step the algorithm undertook was utilising the Google matrix $G$ in situ as described by Eq. (5) and (6). As expected, exactly the same results were acquired as in the previous version of the algorithm using Eq. (3). Longer running times were informally noticed using this algorithm.

C. Eigenvalue Algorithm

Equations from Section III(c) were utilised whereby the exact rank value could be calculated by use of eigenvalues using no iterations. The values obtained show that it is equal to those seen in the iterative algorithms when the results stabilise. This can be shown in Figure 10, which superimposes the direct result of the eigenvalue algorithm with that of the previous method mentioned in part B.
D. Back Button Algorithm

Initial results taken from the network seen in Figure 7 using $\beta = 10\%$ show that pages 0 to 2 were accessed 0.0% of the time, page 3 44.37%, page 4 22.23% and page 5 33.40%, which incidentally corresponds very well to the summation algorithm results seen in Figure 8.

Further results taken from testing larger randomly generated networks showed similarities to that of the eigenvalue algorithm results. Values had an average error of 7.7% over the eigenvalue algorithm in network sizes over 300 pages. On a minority of pages, large anomalies were sometimes found where the difference between ranks of the back button and eigenvalue algorithms were significant – between 10 to 20%.

E. Teleportation Back Button Algorithm

Adjusting the factors $\alpha$ and $\beta$ changed the distribution curve (the range) of rank values. The effect of changing the factors was examined in detail. A rank value distribution curve taken from the eigenvalue algorithm was analysed against distribution curves from the teleportation back button algorithm. Increasing values of $\alpha$ with a constant $\beta$ created a flatter rank distribution. Any small change (< 30%) in $\beta$ with constant $\alpha$ did not change the distribution curve significantly. However, any large values (> 80%) of either factors created very flat distribution curves (smaller range). Therefore, any combination of high $\alpha$ to low $\beta$, or high $\beta$ to low $\alpha$ would cause the distribution to flatten.

The best values that obtained the closest resemblance to the eigenvalue distribution curve were 10% for both $\alpha$ and $\beta$ factors seen in Figure 11(a).

Using the damping factor $\alpha = 10\%$ and the back button factor $\beta = 10\%$, ranks had a typical error of 6.4% in network sizes over 300 pages. Using the teleportation algorithm increased accuracy by over one percent over the previous back button algorithm.

Including $\alpha$ has therefore increased the accuracy of results. Its rank values have been compared directly with the rank values of the eigenvalue algorithm in Figure 11(b). Both values continue to follow very closely to each other.
A linear running time was found in the summation algorithm which used Eq. (1). The greater the number of iterations used, the greater the gradient of its graph became by approximately 0.013 times for every extra iteration. As few as 15 iterations were required to resolve most networks, so the algorithm was generally very fast.

The running times for all the matrix-driven algorithms Eq.(2) to Eq.(6) including the eigenvalue algorithm were polynomial. Increasing the number of iterations did not affect the order of the polynomial which was of $n^4$. Changing values of alpha also did not affect the running times of any matrix-driven algorithm. It did however affect how quickly results resolved. Higher alpha values meant that more iterations were required to obtain distinct, useable results.

Analysis of the back button algorithm revealed polynomial running times of order $n^3$. Running times fluctuated heavily when the number of network traversals was low, for example $2|P|$. This was due to the calculations being completed too quickly. Iterations set at $|P|^2$ however, and stabilisation of running times was evident. Huge fluctuations appeared when the value was set to a considerably high value such as $|P|^3$. It was also noted that the higher the value of $\beta$, the more time the algorithm required to resolve the rank values (Figure 12) – however, all results were within a $3^{rd}$ order polynomial running time.

Analysis of the teleportation back button algorithm revealed that the addition of $\alpha$ appeared to increase the running time as expected (Figure 13). The extra code written which included utilising $\alpha$ meant more processor resources were required. This was trivially the case when varying the value of $\beta$ in conjunction with $\alpha$. All times were within $3^{rd}$ order polynomial – no combination of $\alpha$ or $\beta$ values affected this order.
Many of the project aims were fulfilled in this paper returning some level of success. The fabrication of networks were made effectively without introducing any bias during individual construction. Although the average number of URLs was predetermined for any one page (between 7 and 10), there was no set boundary for inlinks.

The in-degree equation for the Interweb mentioned in Section II produces a seat-curve graph following $y = x^{-2.1}$ [2]. When comparing this to the generated networks' in-degrees (no more than 6000 pages per network), a tall bell-curved distribution was seen. The peak of the distribution curve (i.e. most common in-degree for a page) shifted depending on the number of pages in the network. The rate of shift was approximately linear to that of the number of pages per network. If the number of pages were extrapolated to 8 billion (approximate page number of the Interweb) then the average in-degree for a page would work out at over 26 million. Comparing this to the findings from [2] it is quite evident that this extrapolation is invalid, which shows the limitations of using this particular type of fabricated network. The networks had too small page numbers and a linking system based on randomisation so could not possibly represent the structure of the Interweb precisely.

Using the ratio of the number of total inlinks found against the number of pages in any given generated network, the most common in-degree per page for a network the size of the Interweb would work out to be 24. Relating this to the $n^{-2.1}$ equation from [2], it would suggest 0.02% of the pages on the Interweb have in-degree of exactly 24. Further tests would have to be conducted in the future to determine the accuracy of these findings.

The random graph generator had memory limitations to approximately 12,000 pages. For future work, a more efficient encoding or storage structure could be devised (not human readable) which could allow for more pages to be stored. Reading-in the networks was also limited to 8,000 pages before stack overflow occurred using 1 Gigabyte of system RAM and upward of 1.5 Gigabytes of page file use. The program was designed to store all pages and information in RAM, so future work could alter the way networks are read so that a minimal amount of memory is used. Instead it may be possible to transfer more data into page files or transfer permanently to hard disk.
Finally, analysing the effects of changing the percentage of URL amount per page (as in Figure 3) would be also something to consider. The random graph generator was set up to include generation statistics to make this possible, but due to project time constrictions and the vast quantity of time it would require, this work was not carried out.

The creation of the summation and the following matrix-driven algorithms Eq. (1 to 6) was quite successful. The linear nature of the summation algorithm running time was expected, but unfortunately could not be compared very well to other matrix-driven algorithms. This is because the matrix-driven algorithms included some form of the JAMA package to handle basic linear algebra and matrices and so had similar running times between them. Any differences between the matrix-driven algorithms’ running times would have been subtle. Further investigation in to the details of these differences could be future work.

In terms of accuracy of rank values, the summation algorithm was flawed by not handling certain networks. This was also the case for the first matrix-driven implementation from Eq. (2). Pin-pointing exactly what causes the infinite oscillations to occur could be further investigated, although from initial research it appeared certain combinations of cycles, dangling nodes, and pages lacking a number of inlinks may have been the cause – but the idea of a small bug in the algorithm, although unlikely, will not be dismissed. For networks that could indeed stabilise, adequate results from the summation algorithm were output in linear time with very low iterations required to obtain clearly defined results.

The first fully working example of a PageRank algorithm was created using the stochastic primitive Google matrix algorithm from Eq. (3). Oscillations successfully stabilised after approximately 55 iterations. Incidentally, after less than 50 or 60 iterations this proved to be as accurate as the eigenvalue algorithm. It is a fact that this stochastic algorithm uses less memory and although considered to be in $4^{th}$ order polynomial running time, its time curve suggests it will run faster than its eigenvalue counterpart. This can be explained due to the fact less JAMA package methods are executed in the stochastic algorithm otherwise taking up space and time to run.

An investigation into the effects of resolving stable rank values using different $\alpha$ values was conducted using the stochastic algorithm. For stabilisation at approximately 55 iterations, $\alpha$ was calibrated to 0.9. At this setting, any given graph will converge reasonably quickly. However, when $\alpha$ was increased to 0.999, it carried out 5600 iterations before stabilisation occurred. Conversely, when $\alpha = 0.5$, only 10 iterations were required. As $\alpha \rightarrow 1$ the number of iterations used by the stochastic algorithm increases vigorously [4, 12, p47]. It is now understood why Brin and Page set a compromise of $\alpha = 0.85$ where the resolving power of many iterations balances the time and resources taken.

For future work, an investigation could be made on the $\alpha$ factor in more detail. A study of why certain networks do not become heavily affected by change of $\alpha$ values and other networks do would be a priority. Again, this could be down to network structure.

Some matrix-driven algorithms contained the full incarnation of the Google matrix $G$, whilst others acquired $G_{ij}$ elements in situ from an existing sparse matrix $H$ (see Eq.(5)). The idea was to imitate a method Google uses to save memory. As $G$ is a dense matrix, every element contains a value. The sparse sub stochastic matrix $H$ having many elements set at zero can be economical to store in comparison to $G$ (which, incidentally would have to have dimensions of 8 billion $\times$ 8 billion regarding its use for the Interweb) [12, p40].

In practise, when this procedure was carried out in the investigation, no resources were saved because of the way Java handles memory by allocating space for every array. Future
work could involve adopting compression algorithms, configuring new data structures, or using programming platforms that rely on managing memory, like C.

The eigenvalue algorithm worked as expected. Although it produced the most accurate results of all the PageRank algorithms created, it also used the most time and resources. 4th-order polynomial running time was verified through the use of Scilab’s linear regression function. This also compared to other algorithms which required the use of JAMA.

It is also worth mentioning the primitive nature of the method used to extract the correct index from the eigenvector matrix seen in Figure 5. Initial complications using the eigenvalue decomposition methods in JAMA set back the project investigation. If more time were available, a more efficient method of extracting the rank values from the eigenvectors would have been made. Saying this however, for every network tested, the existing method has worked every time without fail.

The back button algorithm was created very successfully obtaining positive results. These correlated very well with results from the eigenvalue algorithm. Large anomalies were sometimes found where the difference between ranks of the back button and eigenvalue algorithms were significant – between 10 to 20%. This can be explained due to the fact in some cases, the surfer would be confined within a local area of the network thus increasing the access numbers for that area. Without being able to teleport to a new area, the build up of accesses would continue showing some uneven levels of rank values.

An interesting observation is when β was increased, the time taken was reduced for pages greater than 4000 per network (Figure 12). The more time the algorithm accesses previous visited pages, the quicker the result will be. This may be due to the processing within the computer system used. Previously visited pages may already be in system cache and as a result will be processed much quicker. It must be noted, although having a high β value may seem advantageous, as discussed earlier this can flatten the range of rank values making them more difficult to resolve. This is why not only α has to have a compromise but also β – for time against accuracy.

When introducing the teleportation back button algorithm, it was noted an increase in running time occurred when α was included (still within 3rd order polynomial). One possible explanation is that the algorithm required extra resources due to an increase in code length to include the introduction of the α factor.

The distribution of rank values appeared to widen the lower the α factor value became. One explanation to this is when α is low, there is less chance of the ‘surfer’ teleporting to a new part of the network and so stays within a local area. The difference in visits from the local area to non-visited areas would justify this distribution. The value of β did not affect distributions as much, probably because it reinforced already traversed routes only once or twice on average and its effects aren’t as noticeable. For very high values however, flattening of distributions was evident as mentioned previously. Initial ideas as to why this occurred could be that the memory stack was accessed so often it became empty. As a default procedure, the algorithm would teleport and we know already that having high levels of teleportation will cause the distributions to flatten. Increasing the limited stack capacity may prevent this problem in the future.

A limitation of the back button algorithm is that once the page has been ‘popped’ off the stack and visited, there is no way of going back to the original page the surfer was on before without ‘guessing’. In other words, there does not exist a forward button. For a real world surfer, the same pair of web pages may be accessed again and again by use of the back and forward buttons. Relying on the probability of $1 / |P|$ may not represent this surfing behaviour.
accurately enough. For future work, the forward button algorithm could be investigated. It can be then re-tested to see if there is any improvement in the quality of rank value results.

VI. CONCLUSION

The solutions to the PageRank algorithm are varied and numerous. The number of possible tests that can be conducted on these algorithms are simply vast. This paper intended to experiment on a reasonably wide range of subjects rather than concentrate on any one in detail. By this approach, it not only emphasises general trends of the PageRank algorithms, but it also opens the possibility of investigating more specific experiments in detail – providing knowledge of what will and will not be feasible to study.

Due to the randomised nature of the fabricated networks mentioned in this paper, they are not indicative of the Interweb’s structure (Figure 1). The Interweb has a defined structure influenced by humans which cannot be replicated easily by any algorithm. It can however be studied using visualisation tools such as Walrus “unpublished” [13]. Having access to web crawlers to index the Interweb would be the ultimate solution to analysing its structure [1] and ensuring the best possible algorithm can be made.

Comparing between the basic summation-type and matrix-driven algorithms, it is easy to conclude that the latter is far superior in terms of accuracy. Having $\alpha \approx 0.85$, (as Brin and Page call the damping factor) appears to be a good balance between resolution of ranks and speed. Too much, and it wastes time and can fluctuate noticeably for even small changes in the Web’s structure [12, p48]. Too little and convergence is not of a great enough tolerance to provide clear results.

The back button algorithm works in 3rd order polynomial time so is distinctively quicker than any of the matrix-driven algorithms. It may be possible to replicate the algorithm by using matrix manipulations as was achieved by the summation algorithm. However, this may just increase complexity and resource requirements to a point where the benefits of having such a simple, fast, traversal-style algorithm are lost.

The teleportation back button algorithm provides a small improvement over the results from the normal back button algorithm – but at a slight extra cost. Whether the accuracy or running times outweigh one another could be a subject for further investigation.

If an extension to this study were possible, further investigation into the modifications of the PageRank algorithm would be made. For example, studying the effect of assigning pages a starting value to force higher results. Also investigating the surfing behaviour in humans to the next level, where favourite web pages are accessed more than others.

The algorithm HITS, similar to PageRank, was developed during the same time. However, it was disregarded substantially in business and academic circles. An article written by the Microsoft research department this year performed a test on several ranking algorithms including HITS, PageRank, in-degree and BM25F [14]. PageRank was not listed as the best algorithm but did outperform HITS. The extent as to why that was the case could be further investigated by implementing experiments with HITS as a next area of study.
VII. ACKNOWLEDGEMENTS

I wish to thank my supervisor, Stefan Dantchev, for all his help regarding most of the mathematical aspects of the project and Maya Clayton, my wife, for supporting me in everything I do.

REFERENCES