PageRank: an Exploration of Hyperlink Structures

September 2022 - May 2023

Robin Lyster

Student I.D. 10459970

Contents

Chapter 1

Abstract

Pagerank is an algorithm that was developed by Sergey Brin and Larry Page in the late 1990's, which uses an efficient power iteration method to calculate the centrality score of nodes within a graph. Famously, Brin and Page implemented PageRank on the World Wide Web for use in their search engine, Google search. This project sets out to explore the PageRank algorithm in depth, and implement PageRank on the graph of Wikipedia articles. The wider context of PageRank and information retrieval is discussed, followed by the history of search engines. Then, the PageRank algorithm is derived from a few base principles, and is modified to guarantee convergence. Next, PageRank is implemented in MATLAB and create a web crawler to create a graph of all Wikipedia articles. Finally, I implemented the PageRank algorithm on a graph of Wikipedia and analysed the resulting PageRank vector, including tweaking the parameter α. The results show that the category articles had the highest PageRank values owing to their natural centrality within the purposefully-organised structure of Wikipedia. Also shown is the significance of the value of α, and how changing its value affects the PageRank vector's characteristics.

Chapter 2

The Algorithm that Changed the World

2.1 Background and Introduction

In 2013, Ian Stewart published *17 Equations That Changed the World* [1]. Inside, it details what Stewart thinks to be 17 mathematical equations that have had a fundamental impact on the world as we know it, from Pythagoras' theorem to May's logistic map. The most recent equation discussed in the book was the Black-Scholes model, published in 1973; clearly, the world has changed significantly since. Arguably the greatest change the world has seen since is the invention and proliferation of the internet, and if there was an equation that most influenced the internet as it is today, it would be the *PageRank* formula.

In the late 1990s, the internet had a problem. *Search engines*, the tool used to navi-

gate the *World Wide Web*, were notorious for being difficult to navigate effectively. Due to the sheer amount of information available, search queries would often return too many websites containing too much information to reasonably cover, the vast majority of which was irrelevant to the query. Effectively refining your search was difficult and often required taking advantage of search refinement techniques that a layman would not know, such as boolean search[2]. The huge and rapidly expanding Web was at risk of becoming completely innavigable due to its sheer size [3], with searching for the relevant page being like trying to find a needle in a haystack. That is, until *Google search* was released by computer scientists Larry Page and Sergey Brin. Google search used the completely new PageRank formula, which exploited the underlying hyperlink structure of the Web. Google search was able to effectively rank websites by how well-connected they were in relation to other pages. When combined with traditional search methods, huge swathes of the web could be checked for relevance and ranked with incredible effectiveness in a fraction of a second. This completely revolutionised not only search engines, but the internet as a whole.

In this project, I will explain the history of information retrieval, and how it necessitated PageRank. Then, I will explain the mathematics behind the PageRank formula and implement it in an algorithm in *MATLAB.* Finally, I will explore the effectiveness of PageRank on Wikipedia, an encyclopedia with hyperlinks between articles.

Chapter 3

Information storage and retrieval

The content of this chapter and the next draw from the content of chapters 1 through 4 of the book *Google's PageRank and Beyond: The Science of Search Engine Rankings*[4].

The internet, in its essential form, is a huge, world-spanning store of information, accessible to anyone with an internet connection. Therefore, a search query is an attempt to retrieve information from the internet. To discuss why *PageRank* is necessary and why it is effective, it is first important to discuss the history of information storage and retrieval and how the advent of the internet has changed how we retrieve information so fundamentally. The core principle that information retrieval revolves around is that although all information is related to some topic, no topic is relevant to all information. So, when you want to perform any task, from going shopping to learning about search engine formulas, you must retrieve the relevant information required to properly execute the task.

The most convenient form of information storage is also the oldest form, which we use all

the time, often subconsciously: our brains. The process of memory is new information being stored is done when our brains refine which neurons connect to which and how strong those connections are. This new information is then retrieved when an external stimulus triggers causes neurons to fire, causing a chain reaction between these newly-formed neural connections, which in turn triggers a memory [5]. Despite being a subconscious process, our brains are one of the most complex phenomena that we know of, and we are yet to fully understand how brains work. However, the structure of our brains being comprised of a huge series of interconnected neurons is beginning to become understood well enough that is becoming ever-more pertinent in the advancing world of computers [6].

However, this is getting ahead of ourselves. How did we first start recording information physically? The earliest form of information storage that has survived to this day are cave paintings, where information has been physically carved onto rock, and accessed by walking between paintings. In this case, the meaning of most paintings has been lost to time, and the information stored would be limited in scope to symbolism.

What truly delineates prehistory and archaeology from history is the invention of writing systems [7]. The advent of writing allowed any information to be transcribed into script and recorded physically onto rock, papyrus, and eventually paper. This has close ties with the advent of society; once humanity had advanced enough agriculture to allow for specialisation of labour, recording information too complex for pictographs became more important as the sum of human knowledge expanded beyond what one person could remember. Now that written documents existed, they would need to be stored somewhere, so that they could be accessed and read. The largest of such stores included

traditional libraries, full of up to hundreds of millions of different documents and books [8](non-written stores of information still persisted, such as prose stored by method of oral tradition). But, how were libraries organised, and how was the right information obtained quickly?

Nearly all sizeable libraries are curated by a librarian, whose job is to ensure the contents of the library are organised and to assist people in finding the information they've requested, or a specific book. Another tool used is the Dewey Decimal Classification System, which assigns areas of study to sections of the rational numbers between 0 and 1000. Broad topics cover larger spaces, and smaller topics contained within are given subsections. This takes advantage of the fractal nature of information to allow for navigation down a tree, from very broad topics to very specific topics. Using both these methods is normally sufficient to navigate even the largest physical stores of information.

Although technological advancements, like the invention of the printing press, have improved the quality and longevity of information storage and subsequently improved humanity's collective knowledge, the information itself has always been of the same format: physical storage. However, this began to change rather recently.

The latter half of the 20^{th} century saw the advent of computers and digital information storage. Now, information could be encoded into a series of ones and zeroes, allowing information, such as text and images, to be stored digitally. Digital files could be stored electronically on drives, and organised in file systems and databases. Although electronically stored information was a completely novel idea and forms the basis of most stored

information today, information retrieval was still easy. This is because the size of a single hard drive was still small enough that finding a file was easy enough. However, this would soon change, and very quickly.

Multiple computers started being linked in networks, allowing access to data stored far away on another computer. Although this was initially done on a small scale for academic and military purposes, later networks were set up for commercial use. One of the earliest large-scale examples of this was France's *Minitel*, which allowed people to (among other things) make online purchases and check stock prices from a purpose-built terminal. However, the contents of Minitel were restricted, as the general public only accessed the service, instead of adding to it. Other networks didn't have this restriction and instead could be added to by anyone with a connection. As networks became larger and more connected, the possibility of one huge, world-spanning information network became more and more likely. By 1990, all the tools for a working World Wide Web (the Web) had been created, spawning the birth of the internet as we know it.

The Web is, like libraries before it, an information store, now accessible remotely by any computer able to connect to it. However, there are a few key differences that make it a different sort of information storage, which make information retrieval incredibly difficult. Yet, such differences can also be exploited in fascinating ways that make the Web far and away the most popular source of information, far surpassing all others.

There are four main differences between traditional data storage and information stored on the Web, which are:

1. The Web is huge. It contains so much information that it's difficult to obtain an accurate estimate of its size.

2. The Web is also dynamic. Unlike other information stores, where new information is added at a manageable rate, the Web is growing at an incredible rate. Furthermore, information on the Web changes all the time, with most websites changing frequently. This is in contrast to, say, books, which once published are rarely ever revised.

3. The Web is self-organised. While libraries are curated by trained librarians, anyone can submit to the Web, and it is almost entirely unorganised. These three differences alone would make retrieving information off the Web a nearly impossible task. But, there is one final key difference that is exploitable, which makes searching for information on the Web much easier than any traditional form of information storage:

4. The Web is hyperlinked! Webpages are all linked to one another. The Web isn't intentionally structured, but hyperlinks allows pages on the Web to be organised by their relation to other webpages. Hyperlinks, once exploited properly, has transformed search engines from complicated, awkward machines that worked poorly into polished, simple-to-use tools that can seemingly magically pick out relevant information from the Web quickly and effectively. This is what *PageRank* is all about.

But before jumping into hyperlinks, we must first analyse traditional search engine methods and exactly why they can't effectively search the Web.

3.1 Traditional search engines

To see how hyperlinks can be exploited to aid searching for information across the Web, it is important to know what tools search engines can use to search through more traditional information stores. Exploring these tools in the context of the Web and their subsequent shortcomings will help outline what improvements would be needed.

The most simple search engine tool that is used is the Boolean search. This search method simply crawls through documents, looking for a direct match between the search query and content within said document. An exhaustive search through the entire Web for all pages containing a certain query would take orders of magnitude too long and be wasteful, so instead search engines will consult a content index, which is a vast lookup table containing a vast array of possible search query terms, each of which return a list of all webpages containing the search query. By splitting a multiple-word search query into individual words and returning only pages containing every word, you can produce a smaller list of more relevant pages. All search engines use this technique heavily, as a verbatim match is likely to have relevance, and isn't computationally intensive. The downside here is laid bare by the method's name. A Boolean search will return either a zero or a one, with no further nuance. How do we subsequently order the pages that return a match?

Vector Space model search engines use a more complicated method to attempt to improve upon Boolean search engines. By transforming text into numeric vectors and documents into matrices, matrix analysis techniques can reveal more complicated features of text, such as semantic links between words. This allows for search queries to return relevant

documents that a Boolean search wouldn't, such as articles containing the word "rock" when a user searches for "stone formations." Vector Space models also allow for partial matching, where a webpage may be given a percentage match to a query. This allows for a ranking of results, where higher up pages are more relevant to the query than lower down ones. However, this comes at the expense of computational simplicity, as each search query would require matrix analysis of each document to produce a result, which becomes prohibitive at scale (some even require performing operations on a matrix representing an entire document collection, which increases with the square of the size of the collection). Nevertheless, semantic links are incredibly important in searching through documents, and Vector Space models can reveal links within language that previously could only be done by a human (it's very easy to know that "train" and "locomotive" are synonymous, but it's incredibly difficult to program a computer to be able to find these links on its own).

Most search engines, prior to *Google* search, would use a combination to the above methods and many others, to return an ordered list of webpages from a search query. But, producing a well-ordered list of relevant documents often required carefully-worded queries, and still required sifting far down the results list to acquire the most useful pages. Furthermore, ordering results by relevancy required techniques of computational complexity higher than $\mathcal{O}(n)$, where n is the number of documents being searched. With the Web's size increasing exponentially in the late 1990s, traditional search methods were falling short of the requirements needed to effectively search the ever-expanding Web. It follows that a system that was able to effectively combine the linear complexity of Boolean search with the useful ranking afforded by Vector Space models would revolutionise Web search engines.

Now is the time that we can finally look at why hyperlinks are the key to cracking this problem. Hyperlinks form a basis for the structure of the internet, and are one of two ways to access a webpage (the other being directly accessing a page through a *http* web address). But, why would a hyperlink exist in the first place? The answer is relevancy. A hyperlink exists to direct a user from one webpage to another that the author of the first page considers relevant. A hyperlink could be considered a recommendation for a page, with pages that are commonly linked to being considered more important. Perhaps, webpages within the Web as a whole have an underlying "pecking order" that could be revealed through hyperlinks, which could be used by a search engine to recommend more popular pages. But, how would we be able to discover this order? To progress any further, it is now essential to represent the Web in a more mathematical way, allowing us to analyse it more concretely and eventually arrive at *PageRank*.

Chapter 4

PageRank

The Web is in essence a huge collection of webpages, all linked together with hyperlinks. This can be represented mathematically by a directed graph, where a webpage is represented by a node and hyperlinks are represented by directed links between each node. A hyperlink leading out from a page is called an *outlink,* and a link leading into a page is called an *inlink*. Figure 4.1 is a small example graph of five numbered pages, where directed arrows represent hyperlinks between pages.

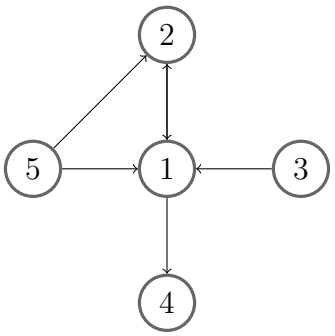

Figure 4.1: An example graph of 5 pages, where directed arrow represent hyperlinks from one to another.

Now is the time to consider what features a popular webpage would display. A popular webpage would:

1. Be linked to by many pages. This would indicate that many other webpages consider its content to be relevant, and should show up on more search queries. In figure 4.1, page 1 is linked to by 3 pages, making it the most linked-to, or "recommended" page.

2. Be linked to by popular pages. A link from a popular webpage should be weighted more than a link from a less popular one. In our example, the recommendations from page 3, which no page "recommends," would be worth less than a recommendation from page 1.

3. Be linked to by pages which link to few other pages. Having too many outlinks weakens the value of a recommendation, and would encourage collections of webpages to all link to each other to mutually and artificially increase their perceived popularity. In our example, page 5's two outlinks would each be worth half of page 3's one outlink.

From this, we arrive at the first mathematical representation of a website's popularity:

$$r(P_i) = \sum_{P_j \in B_{P_i}} \frac{r(P_j)}{|P_j|}, \tag{4.1}$$

where $r(P_i)$ represents the popularity value (from here named PageRank) of a webpage P_i, $P_j \in B_{P_i}$ represents the set of all webpages that link to P_i, and $|P_j|$ is the total number of outlinks from page P_j. The problem with this equation is that the values of $r(P_j)$ are all unknown, since to calculate any page's PageRank requires knowing the PageRank of

other pages, which in turn require knowing the PageRank of even more pages (and son on and so forth). One common way to solve this problem is to reform this equation into something iterative, and give all pages an equal initial PageRank value. If we let $r_{k+1}(P_i)$ be the PageRank of page P_i at iteration $k+1$, we get

$$r_{k+1}(P_i) = \sum_{P_j \in B_{P_i}} \frac{r_k(P_j)}{|P_j|}, \tag{4.2}$$

where $r_0 = 1/n$, where n is the total number of pages on the Web. This equation is iterated with the aim of all the PageRank values converging.

This equation represents the PageRank equation for an individual page, so would need to be calculated for every page on the Web for every iteration. Although the number of calculations needed to be made remains unchanged, this can be combined into a single calculation, when represented in matrix form. We introduce the *hyperlink matrix* $\mathbf{H}$, which is defined as $\mathbf{H}_{ij} = 1/|P_i|$ if a hyperlink from page i to page j exists, and 0 otherwise. This is a normalised version of a *binary adjacency matrix* $\mathbf{L}$, which is a matrix containing all hyperlinks between pages, stored as Boolean values (the difference being that all rows are normalised, and sum to one). Our example in figure 4.1 has the following adjacency

and hyperlink matrix:

$$\mathbf{L} = \begin{pmatrix} 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \end{pmatrix} \mathbf{H} = \begin{pmatrix} 0 & \frac{1}{2} & 0 & \frac{1}{2} & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 \end{pmatrix}$$

Note that the elements in each column j sums to

$$\sum_{P_j \in B_{P_i}} \frac{1}{|P_j|}. \tag{4.3}$$

We also introduce the PageRank vector $\boldsymbol{\pi}$, a $1 \times n$ vector containing the exact PageRank values of each page (if equation (4.2) converged), as well as $\boldsymbol{\pi}^{(k)}$, which is a $1 \times n$ vector containing the numerically calculated PageRank values of each page at the kth iteration of equation (4.2). Note that multiplying our PageRank row vector by column j (denoted $\mathbf{H}_j$) of our hyperlink matrix produces the sum

$$\boldsymbol{\pi}^{(k)T}\mathbf{H}_j = \sum_{P_j \in B_{P_i}} \frac{r_k(P_j)}{|P_j|}, \tag{4.4}$$

which is equal to $r_{k+1}(P_j)$. In fact, if we multiply the PageRank vector by the whole hyperlink matrix, we would be returned the next iteration of the PageRank vector, $\boldsymbol{\pi}^{(k+1)}$. Using matrix notation, we can rewrite the iterative PageRank equation (4.2) as

$$\boldsymbol{\pi}^{(k+1)T} = \boldsymbol{\pi}^{(k)T}\mathbf{H}. \tag{4.5}$$

Typically, vector - matrix multiplication has a complexity of order $\mathcal{O}(n^2)$, which is very computationally expensive and something we must avoid. However, the original iterative equation has an order of $\mathcal{O}(\sum_{P_i \in P} |P_i|)$, which represents having to do one multiplication for every non-zero value in $\mathbf{H}$. This is formally written as $\mathcal{O}(nnz)$, being an acronym for number of non-zeroes, which is proportional to order $\mathcal{O}(n)$. Surely when performed n times to calculate every PageRank value one by one, the total computation would be of order $\mathcal{O}(n)$? The discrepancy here is due to the hyperlink matrix $\mathbf{H}$ being a *sparse* matrix. This is a matrix with a vast majority of its values being 0. Even as the Web has more and more pages added to it, the number of pages the average website links to stays roughly the same. For $\mathbf{H}$, this means that even though the total number of values in the matrix increases by order n^2, the total number of non-zero values only goes up by order n. Effectively, the zero values can be skipped over in the calculation, reducing the order of the vector - matrix multiplication to polynomial order $\mathcal{O}(n)$. More precisely, a webpage has an average of about 10 hyperlinks to other pages, so each iteration of the PageRank equation requires $\approx 10n$ floating point operations.

Before continuing, we must ask ourselves a few questions about our iterative equation, and then analyse our equation's properties in order to see whether the equation, or its components, need any modification.

The aim of our iterative equation is to converge on a unique vector of PageRank values (one for each webpage) that contextually improves our search engine's ability to find useful and relevant information. This aim sheds some light on some of the ways our equation may fail to achieve this:

1. Our PageRank equation (4.5) may fail to ever converge.
2. $\boldsymbol{\pi}^{(k)}$ may converge to multiple values, oscillating between them.
3. $\boldsymbol{\pi}^{(k)}$ might converge to one value, but it may depend on the values in the starting vector.
4. The PageRank equation may always converge to the same vector, but only after a computationally prohibitive number of iterations.
5. There may exist a single PageRank vector that the PageRank equation will converge to in a reasonable amount of time no matter the initial values in $\boldsymbol{\pi}^{(0)}$, but it may not reveal each webpage's relative importance within the hyperlink structure of the Web.

To ensure we don't realise any of the above pitfalls, we must analyse the properties of the PageRank equation, specifically in relation to our hyperlink matrix $\mathbf{H}$. Firstly, our iterative equation is a version of the *power method,* an algorithm designed to find the dominant eigenvalue in modulus of a matrix, along with its corresponding eigenvector. Secondly and more importantly, our hyperlink matrix $\mathbf{H}$ closely resembles a transition probability matrix for a Markov chain. A Markov chain is a model for a stochastic process which describes a sequence of possible events, where the probability of each possible event only depends on the state attained in the previous event. There are two notable differences between $\mathbf{H}$ and the transition probability matrix of a Markov chain. The first is that some pages may have no hyperlinks leading out of them (our example in Figure 4.1 has one such page, page 4), which would make the rows in $\mathbf{H}$ representing them *non-stochastic*. Row stochasticity is a property of transition probability matrices, in which all rows in

the matrix must sum to exactly 1 (representing the complete probability space for transition). A row which didn't sum to one would represent an incomplete probability space, making the Markov chain no longer a fully contained system, and the sum of probabilities represented in our PageRank vector would decrease with every iteration. The rows in our hyperlink matrix which aren't stochastic are called *dangling nodes*, named for their lack of outlinks. Since division by zero is undefined, these rows cannot be normalised. The second difference is that the values in a Markov chain transition matrix are values strictly between zero and one inclusive, while $\mathbf{H}$ currently has no such restriction. This is because a number representing a probability is by definition bounded between 0 (impossible) and 1 (certain). However, a value in $\mathbf{H}$ will be either 0 or $1/|P_i|$, which (since $P_i \in \mathbb{N}$) will certainly be within $[0, 1]$.

Both of these properties make it possible to guarantee that, with a few modifications, our iterable formula will produce a unique PageRank vector that is useful. To see how and why, it is useful to rephrase the problem in the context of how people actually use the Web. We introduce the "surfer," a person clicking from page to page on the Web. This surfer starts on a random webpage, and chooses a random hyperlink on the page, clicking on it and transferring to another webpage. This is, of course, an oversimplification, but will be built upon further. If this person kept clicking hyperlinks for an infinitely long time, every webpage with a hyperlink leading to it would eventually be visited, with the surfer spending a certain proportion of their time on each page. This is essentially a discrete-time Markov chain Monte Carlo, with transition matrix $\mathbf{H}$ and probability state vector $\boldsymbol{\pi}$ at time $t \to \infty$.

It is at this point where the first problem becomes apparent. What happens when our surfer arrives at a page with no outgoing hyperlinks? On our matrix **H** this corresponds to the non-stochastic rows comprised entirely of zeroes. How would this problem be sidestepped? The simplest solution, and arguably the fairest solution, is to say that the surfer simply starts the surfing process again, by choosing a webpage at random and surfing from there. This would change our transition probabilities on our non-stochastic rows by making each value within equal to $1/n$, again with n being the total number of webpages. This is done by introducing the *dangling node vector* **a**, of length n with values of 1 at the index of each dangling node, and the one-vector **e**, of length n. This adjustment makes our matrix fully stochastic we shall name this stochastic, altered hyperlink matrix **S**. Below is the stochastic matrix associated with our example graph in Figure 4.1:

$$\mathbf{H} + \mathbf{a}(\frac{1}{n}\mathbf{e}^T) = \begin{pmatrix} 0 & \frac{1}{2} & 0 & \frac{1}{2} & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} (\frac{1}{n} \begin{pmatrix} 0 & 0 & 0 & 1 & 0 \end{pmatrix})$$

$$= \mathbf{S} = \begin{pmatrix} 0 & \frac{1}{2} & 0 & \frac{1}{2} & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ \frac{1}{5} & \frac{1}{5} & \frac{1}{5} & \frac{1}{5} & \frac{1}{5} \\ \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 \end{pmatrix}$$

Note that the transition probability on rows representing dangling nodes are all of uniform value. This is a purposeful decision, as an equal transition probability to every page ensures no page has an advantage over any other.

However, $\mathbf{S}$ being stochastic doesn't necessarily guarantee that there is a single possible PageRank vector $\boldsymbol{\pi}$, or that $\boldsymbol{\pi}^{(k)}$ will converge to $\boldsymbol{\pi}$ in a reasonable amount of time. We can solve the first problem by making $\mathbf{S}$ *positive.* A matrix is positive if every value in the matrix is strictly greater than 0. It also guarantees a few useful properties which will guarantee convergence of $\boldsymbol{\pi}^{(k)}$ to a unique probability distribution vector $\boldsymbol{\pi}$.

Perron's theorem states the following:

Definition. The *spectral radius* of a matrix $\mathbf{S}$, denoted as $\rho(\mathbf{S})$, is equal to the maximum absolute value of all eigenvalues of $\mathbf{S}$.

Theorem. (*Perron's Theorem.*) Let $\mathbf{S}$ be a positive square matrix. Then:

1. $\rho(\mathbf{S})$ is itself an eigenvalue, and has a positive eigenvector.
2. $\rho(\mathbf{S})$ is the only eigenvalue on the *disc* $|\lambda| = \rho(\mathbf{S})$.
3. $\rho(\mathbf{S})$ has *geometric multiplicity* 1. This means the dimensions of the sub-space that is spanned by every associated eigenvector of $\rho(\mathbf{S})$ is one.
4. $\rho(\mathbf{S})$ has *algebraic multiplicity* 1. This means that $\rho(\mathbf{S})$ appears only once as the root of the characteristic polynomial of $\mathbf{S}$.

A full proof of Perron's theorem can be found at [9]. Perron's theorem ensures that a positive eigenvector exists, with eigenvalue $\rho(\mathbf{S})$. Also, because $\rho(\mathbf{S})$ has a geometric multiplicity of one, all eigenvectors associated with $\rho(\mathbf{S})$ span the same linear subspace.

The next thing we need to prove is that if our stochastic matrix $\mathbf{S}$ is also positive, any starting probability distribution vector $\boldsymbol{\pi}^{(0)}$ converge to a dominant eigenvector of $\mathbf{S}$.

Let $v_2, v_2, ..., v_m$ be an eigenbasis of stochastic and positive matrix $\mathbf{S}$, where each eigenvector v_i is associated with the eigenvalue λ_i. Also let $\boldsymbol{\pi}^{(0)}$ be a probability vector of length n. As $\mathbf{S}$ is positive, the dominant eigenvalue of $\mathbf{S}$ has a positive eigenvector and $\boldsymbol{\pi}^{(0)}$ has at least one positive entry, $\boldsymbol{\pi}^{(0)}$ is in the subspace generated by the eigenvectors of $\mathbf{S}$:

$$\boldsymbol{\pi}^{(0)} = a_1 v_1 + a_2 v_2 + ... + a_n v_n, \tag{4.6}$$

where $a_1, a_2, ..., a_n$ are constants and $a_1 \neq 0$.

Theorem. *The power method, applied to* $\boldsymbol{\pi}^{(0)}$, *will converge to a dominant eigenvector.*

Proof [10]. We have

$$\begin{aligned}
\mathbf{S}^k \boldsymbol{\pi}^{(0)} =& a_1 \mathbf{S}^k v_1 + a_2 \mathbf{S}^k v_2 + ... + a_n \mathbf{S}^k v_n \\
=& a_1 \lambda_1^k v_1 + a_2 \lambda_2^k v_2 + ... + a_n \lambda_n^k v_n \\
=& a_1 \lambda_1^k \left(v_1 + \frac{a_2}{a_1} \left(\frac{\lambda_2}{\lambda_1} \right)^k v_2 + ... + \frac{a_n}{a_1} \left(\frac{\lambda_n}{\lambda_1} \right)^k v_n \right).
\end{aligned} \tag{4.7}$$

As λ_1 is the dominant eigenvalue, the expression contained within the brackets converges to v_1 as $k \to \infty$, and the whole expression (4.7) converges to a dominant eigenvector $a_1 v_1$. □

Also note that as $k \to \infty$, the expression within the brackets converges to v_1 at the

rate the other terms converge to 0. As the term associated with the second largest eigenvalue $\left(\frac{\lambda_2}{\lambda_1}\right)$ decreases exponentially slower than every other decreasing term within the expression, the convergence of the power iteration is very roughly proportional to $\left(\frac{\lambda_2}{\lambda_1}\right)$.

We now have proven that any probability vector $\boldsymbol{\pi}^{(0)}$, iterated through the power series method, will approach an eigenvector associated with the dominant eigenvalue of stochastic matrix $\mathbf{S}$, provided $\mathbf{S}$ is also positive. So, we choose $\boldsymbol{\pi}^{(0)} = \frac{1}{n}\mathbf{e}$, as any probability space will converge and the uniform probability space gives no initial bias to any page's value within $\boldsymbol{\pi}^{(k)}$. The task at hand is to now make $\mathbf{S}$ positive.

The best way to ensure this can be found by re-thinking our random surfer. When surfing the web, nobody goes on clicking hyperlinks forever. At some point, the surfer will tire of the current link path and choose another website via search. Assume our surfer has an $0 < (1 - \alpha) < 1$ probability of searching for a site instead of clicking a hyperlink, and that the surfer searches a site at random, with every page on the Web being equally likely to be chosen. To keep the sums of the rows of our matrix equal to 1, we must scale our stochastic matrix by a factor of α before adding a uniform matrix of equal size and values $(1 - \alpha)/n$, named the *positive matrix*. Our positive matrix can be written as

$$\frac{1}{n}\mathbf{e}\mathbf{e}^T = \frac{1}{n}\mathbf{E}, \tag{4.8}$$

where $\mathbf{E}$ is a square matrix of all ones. Below is the positive matrix associated with our

example network from Figure 4.1:

$$(1-\alpha)\mathbf{E} = \begin{pmatrix} \frac{1-\alpha}{5} & \frac{1-\alpha}{5} & \frac{1-\alpha}{5} & \frac{1-\alpha}{5} & \frac{1-\alpha}{5} \\ \frac{1-\alpha}{5} & \frac{1-\alpha}{5} & \frac{1-\alpha}{5} & \frac{1-\alpha}{5} & \frac{1-\alpha}{5} \\ \frac{1-\alpha}{5} & \frac{1-\alpha}{5} & \frac{1-\alpha}{5} & \frac{1-\alpha}{5} & \frac{1-\alpha}{5} \\ \frac{1-\alpha}{5} & \frac{1-\alpha}{5} & \frac{1-\alpha}{5} & \frac{1-\alpha}{5} & \frac{1-\alpha}{5} \\ \frac{1-\alpha}{5} & \frac{1-\alpha}{5} & \frac{1-\alpha}{5} & \frac{1-\alpha}{5} & \frac{1-\alpha}{5} \end{pmatrix}$$

This uniform matrix represents the $(1-\alpha)/n$ chance of the user choosing a new website from anywhere on the web at random from every possible state. The sum of each row of our scaled-down stochastic matrix is $1 \times \alpha = \alpha$, and the sum of the rows of our positive matrix are equal to $n \times (1-\alpha)/n = 1-\alpha$. So the sum of the rows of the combination of these matrices is $\alpha + (1-\alpha) = 1$, meaning our combined matrix $\alpha\mathbf{S} + (1-\alpha)\mathbf{E}$, named the *Google matrix* $\mathbf{G}$, is still stochastic. Although we will discuss changing the value of α later, we will initially fix $\alpha = 0.85$, which is the value Larry Brin and Sergey Page used in their original Paper on PageRank [11]. With $\alpha = 0.85$, the Google matrix for the graph

in Figure 4.1 would be:

$$\mathbf{G} = \alpha\mathbf{S} + (1-\alpha)\mathbf{E} = 0.85 \begin{pmatrix} 0 & \frac{1}{2} & 0 & \frac{1}{2} & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ \frac{1}{5} & \frac{1}{5} & \frac{1}{5} & \frac{1}{5} & \frac{1}{5} \\ \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 \end{pmatrix} + 0.15 \begin{pmatrix} \frac{1}{5} & \frac{1}{5} & \frac{1}{5} & \frac{1}{5} & \frac{1}{5} \\ \frac{1}{5} & \frac{1}{5} & \frac{1}{5} & \frac{1}{5} & \frac{1}{5} \\ \frac{1}{5} & \frac{1}{5} & \frac{1}{5} & \frac{1}{5} & \frac{1}{5} \\ \frac{1}{5} & \frac{1}{5} & \frac{1}{5} & \frac{1}{5} & \frac{1}{5} \\ \frac{1}{5} & \frac{1}{5} & \frac{1}{5} & \frac{1}{5} & \frac{1}{5} \end{pmatrix}$$

$$= \begin{pmatrix} \frac{3}{100} & \frac{91}{200} & \frac{3}{100} & \frac{91}{200} & \frac{3}{100} \\ \frac{22}{25} & \frac{3}{100} & \frac{3}{100} & \frac{3}{100} & \frac{3}{100} \\ \frac{22}{25} & \frac{3}{100} & \frac{3}{100} & \frac{3}{100} & \frac{3}{100} \\ \frac{1}{5} & \frac{1}{5} & \frac{1}{5} & \frac{1}{5} & \frac{1}{5} \\ \frac{91}{200} & \frac{91}{200} & \frac{3}{100} & \frac{3}{100} & \frac{3}{100} \end{pmatrix}.$$

In addition to being a positive matrix, our model becomes more realistic to how a user may actually surf the web.

We now only have one more potential problem: although we have proved our iterative method will converge to one value, we don't know how many iterations it will take. When computational efficiency is paramount, iterating the Power series too many times will render the algorithm ineffective. So, how many iterations will it take?

The *asymptotic rate of convergence* of the power method, when applied to a matrix, has been shown to roughly depend on the ratio between the two largest eigenvalues (denoted

λ_1 and λ_2). The convergence rate is the same rate that

$$\left|\frac{\lambda_1}{\lambda_2}\right|^k \to 0, \tag{4.9}$$

where k is the number of iterations done. We know that $|\lambda_1|= 1$ because the Google matrix is stochastic, and that it is the sole eigenvalue with a magnitude equal to one. Below is a proof that the second largest eigenvalue of the Google matrix $|\lambda_2|(\mathbf{G})$ is equal to $\alpha|\lambda_2|(\mathbf{S})$. Note that this proof was taken from chapter 4 of *Google's PageRank and Beyond,* for purpose of providing clarity:

Definition: The *spectrum* of a stochastic matrix $\mathbf{S} \in \mathbb{R}^{(n+1)\times(n+1)}$ is the set of all of the eigenvalues of $\mathbf{S}$, denoted $\{1, \alpha\lambda_2, \alpha\lambda_3, ..., \alpha\lambda_n\}$.

Definition: An *eigenpair* of a square matrix $\mathbf{S}$ is a mathematical pair of an eigenvector and the eigenvalue associated with it.

Theorem: *If the spectrum of the stochastic matrix* $\mathbf{S}$ *is* $\{1, \lambda_2, \lambda_3, ..., \lambda_n\}$, *then the spectrum of the Google matrix* $\mathbf{G} = \alpha\mathbf{S} + (1-\alpha)\mathbf{e}\mathbf{v}^T$ is $\{1, \alpha\lambda_2, \alpha\lambda_3, ..., \alpha\lambda_n\}$, *where* $\mathbf{v}^T$ *is a probability distribution vector.*

Proof. Because $\mathbf{S}$ is a stochastic matrix, there exists $(1, \mathbf{e})$ as an eigenpair of $\mathbf{S}$**.** Let $\mathbf{M} = (\mathbf{e} \quad \mathbf{X})$ be a non-singular matrix with eigenvector $\mathbf{e}$ as its first column. Let $\mathbf{M}^{-1} = \begin{pmatrix} \mathbf{y}^T \\ \mathbf{Y}^T \end{pmatrix}$. Then

$$\mathbf{M}^{-1}\mathbf{M} = \begin{pmatrix} \mathbf{y}^T\mathbf{e} & \mathbf{y}^T\mathbf{X} \\ \mathbf{Y}^T\mathbf{e} & \mathbf{Y}^T\mathbf{X} \end{pmatrix} = \begin{pmatrix} 1 & \mathbf{0}^T \\ \mathbf{0} & \mathbf{I} \end{pmatrix}, \tag{4.10}$$

where $\mathbf{I}$ is the identity matrix and $\mathbf{0}$ is a vertical zero vector. This gives us $\mathbf{y}^T\mathbf{e} = 1$ and $\mathbf{Y}^T\mathbf{e} = \mathbf{0}$. Then, the similarity transformation

$$\mathbf{M}^{-1}\mathbf{S}\mathbf{M} = \begin{pmatrix} \mathbf{y}^T\mathbf{e} & \mathbf{y}^T\mathbf{S}\mathbf{X} \\ \mathbf{Y}^T\mathbf{e} & \mathbf{Y}^T\mathbf{S}\mathbf{X} \end{pmatrix} = \begin{pmatrix} 1 & \mathbf{y}^T\mathbf{S}\mathbf{X} \\ \mathbf{0} & \mathbf{Y}^T\mathbf{S}\mathbf{X} \end{pmatrix} \tag{4.11}$$

shows $\mathbf{Y}^T\mathbf{S}\mathbf{X}$ contains the remaining eigenvalues of $\mathbf{S}$, $\{\lambda_2, \lambda_3, ..., \lambda_n.\}$. To see why, we begin with

$$\begin{aligned}
\det(\mathbf{M}^{-1}\mathbf{S}\mathbf{M} - \lambda\mathbf{M}^{-1}\mathbf{M}) &= \det(\mathbf{M}^{-1}(\mathbf{S} - \lambda\mathbf{I})\mathbf{M}) \\
&= \det(\mathbf{M}^{-1})\det(\mathbf{S} - \lambda\mathbf{I})\det(\mathbf{M}) \\
&= \det(\mathbf{M}^{-1})\det(\mathbf{M})\det(\mathbf{S} - \lambda\mathbf{I}) \\
&= \det(\mathbf{S} - \lambda\mathbf{I}) \\
&= \det\begin{pmatrix} 1 & \mathbf{y}^T\mathbf{S}\mathbf{X} \\ \mathbf{0} & \mathbf{Y}^T\mathbf{S}\mathbf{X} \end{pmatrix} - \lambda\mathbf{I} \\
&= \det\begin{pmatrix} 1-\lambda & \mathbf{y}^T\mathbf{S}\mathbf{X} \\ \mathbf{0} & \mathbf{Y}^T\mathbf{S}\mathbf{X} - \mathbf{I}_{n-1} \end{pmatrix} \\
&= (1-\lambda)\det(\mathbf{Y}^T\mathbf{S}\mathbf{X} - \lambda\mathbf{I}_{n-1}),
\end{aligned}$$

Where $\det(\mathbf{Y}^T\mathbf{S}\mathbf{X} - \lambda\mathbf{I}_{n-1})$ is the characteristic polynomial of $\mathbf{Y}^T\mathbf{S}\mathbf{X}$.

The solution to $1 - \lambda = 0$ Shows that the eigenvalue in $\mathbf{S}$ but not $\mathbf{Y}^T\mathbf{S}\mathbf{X}$ is 1.

If instead the similarity transformation is applied to $\mathbf{G} = \alpha\mathbf{S} + (1-\alpha)\mathbf{e}\mathbf{v}^T$, we get

$$\begin{aligned}
\mathbf{M}^{-1}(\mathbf{S} + (1-\alpha)\mathbf{e}\mathbf{v}^T)\mathbf{M} &= \alpha\mathbf{M}^{-1}\mathbf{S}\mathbf{M} + (1-\alpha)\mathbf{M}^{-1}\mathbf{e}\mathbf{v}^T\mathbf{M} \\
&= \begin{pmatrix} \alpha & \alpha\mathbf{y}^T\mathbf{S}\mathbf{X} \\ \mathbf{0} & \alpha\mathbf{Y}^T\mathbf{S}\mathbf{X} \end{pmatrix} + (1-\alpha)\begin{pmatrix} \mathbf{y}^T\mathbf{e} \\ \mathbf{Y}^T\mathbf{e} \end{pmatrix}(\mathbf{v}^T\mathbf{e} \quad \mathbf{v}^T\mathbf{X}) \\
&= \begin{pmatrix} \alpha & \alpha\mathbf{y}^T\mathbf{S}\mathbf{X} \\ \mathbf{0} & \alpha\mathbf{Y}^T\mathbf{S}\mathbf{X} \end{pmatrix} + \begin{pmatrix} (1-\alpha) & (1-\alpha)\mathbf{v}^T\mathbf{X} \\ \mathbf{0} & \mathbf{0}\mathbf{0}^\mathbf{T} \end{pmatrix} \\
&= \begin{pmatrix} 1 & \alpha\mathbf{y}^T\mathbf{S}\mathbf{X} + (1-\alpha)\mathbf{v}^T\mathbf{X} \\ \mathbf{0} & \alpha\mathbf{Y}^T\mathbf{S}\mathbf{X} \end{pmatrix}. \qquad (4.12)
\end{aligned}$$

Therefore, the spectrum of $\mathbf{G} = \alpha\mathbf{S} + (1-\alpha)\mathbf{e}\mathbf{v}^T$ is $\{1, \alpha\lambda_1, \alpha\lambda_2, ..., \alpha\lambda_n\}$, because the eigenvalues of $\mathbf{S}$ satisfy $\lambda\mathbf{x} = \mathbf{S}\mathbf{x}$, therefore the values $\alpha\lambda$ satisfy $\alpha\lambda\mathbf{x} = \alpha\mathbf{S}\mathbf{x}$. □

From this, we know that $|\lambda_2| < \alpha$, and that the power series method applied to the Google matrix will asymptotically converge at a rate at least as quickly as α^k. Using the value of $\alpha = 0.85$ suggested by Google's founders, Brin and Page [11], we can expect $k \approx 14$ iterations for every desired decimal place of accuracy within the PageRank vector. The time of convergence still heavily depends on the size of the matrix used and the accuracy required, but these numbers look promising; under 100 iterations may be enough to accurately calculate the PageRank vector.

Unfortunately, we now have a *completely dense* matrix, with every entry in $\mathbf{G}$ being non-zero. This completely undoes our savings on computation, as the linear computation time required a sparse matrix. Applying the power method to $\mathbf{G}$ to find $\boldsymbol{\pi}$ now seems to

be $\mathcal{O}(n^2)$ operation for each iteration. However, there is a work-around. We don't apply the power method (from equation (4.6)) directly to $\mathbf{G}$ like so:

$$\boldsymbol{\pi}^{(k+1)T} = \boldsymbol{\pi}^{(k)T}\mathbf{G}. \tag{4.13}$$

Instead, we can re-structure the vector-matrix multiplication by splitting $\mathbf{G}$ into its individual components:

$$\begin{aligned}
\mathbf{G} &= \alpha\mathbf{S} + (1-\alpha)\mathbf{E} && (4.14)\\
&= \alpha(\mathbf{H} + \frac{1}{n}\mathbf{a}\mathbf{e}^T) + (1-\alpha)\frac{1}{n}\mathbf{e}\mathbf{e}^T && (4.15)\\
&= \alpha\mathbf{H} + (\alpha\mathbf{a} + (1-\alpha)\mathbf{e})\frac{1}{n}\mathbf{e}^T. && (4.16)
\end{aligned}$$

Then,

$$\begin{aligned}
\boldsymbol{\pi}^{(k+1)T} &= \boldsymbol{\pi}^{(k)T}\mathbf{G} && (4.17)\\
&= \alpha\boldsymbol{\pi}^{(k)T}\mathbf{S} + \frac{1-\alpha}{n}\boldsymbol{\pi}^{(k)T}\mathbf{e}\mathbf{e}^T && (4.18)\\
&= \alpha\boldsymbol{\pi}^{(k)T}\mathbf{H} + (\alpha\boldsymbol{\pi}^{(k)T}\mathbf{a} + 1 - \alpha)\mathbf{e}^T/n. && (4.19)
\end{aligned}$$

Here, the only vector-matrix multiplication made is to $\mathbf{H}$, and the rest of equation (4.17) (called the *rank-one update*) is comprised entirely of vector operations of complexity $\mathcal{O}(n)$. Finally, we have the efficiency of equation (4.5) with assurance of convergence to an eigenvector associated with the dominant eigenvalue of $\mathbf{G}$!

This iterative equation is a power method to approximate a solution to the PageRank

problem, which is more formally stated as follows. Solve the following eigenvector problem for the PageRank vector $\boldsymbol{\pi}^T$:

$$\boldsymbol{\pi}^T = \boldsymbol{\pi}^T \mathbf{G}, \tag{4.20}$$

$$\boldsymbol{\pi}^T \mathbf{e} = 1. \tag{4.21}$$

Equation (4.19) is known as the *normalisation equation,* which ensures the PageRank vector is scaled to represent a probability vector. We can confirm our rank-one update equation works by calculating $\boldsymbol{\pi}^{(1)T}$ from $\boldsymbol{\pi}^{(0)T} = \frac{1}{n}\mathbf{e}^T$ with our manually calculated Google matrix:

$$\begin{aligned}
\boldsymbol{\pi}^{(1)T} &= \boldsymbol{\pi}^{(0)T}\mathbf{G} \\
&= \begin{pmatrix} \frac{1}{5} & \frac{1}{5} & \frac{1}{5} & \frac{1}{5} & \frac{1}{5} \end{pmatrix} \begin{pmatrix} \frac{3}{100} & \frac{91}{200} & \frac{3}{100} & \frac{91}{200} & \frac{3}{100} \\ \frac{22}{25} & \frac{3}{100} & \frac{3}{100} & \frac{3}{100} & \frac{3}{100} \\ \frac{22}{25} & \frac{3}{100} & \frac{3}{100} & \frac{3}{100} & \frac{3}{100} \\ \frac{1}{5} & \frac{1}{5} & \frac{1}{5} & \frac{1}{5} & \frac{1}{5} \\ \frac{91}{200} & \frac{91}{200} & \frac{3}{100} & \frac{3}{100} & \frac{3}{100} \end{pmatrix} \\
&= \begin{pmatrix} 0.489 & 0.234 & 0.064 & 0.149 & 0.064. \end{pmatrix}
\end{aligned}$$

$$\boldsymbol{\pi}^{(1)T} = \alpha\boldsymbol{\pi}^{(0)T}\mathbf{H} + (\alpha\boldsymbol{\pi}^{(0)T}\mathbf{a} + 1 - \alpha)\mathbf{e}^T/n$$

$$= 0.85 \begin{pmatrix} \frac{1}{5} & \frac{1}{5} & \frac{1}{5} & \frac{1}{5} & \frac{1}{5} \end{pmatrix} \begin{pmatrix} 0 & \frac{1}{2} & 0 & \frac{1}{2} & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 \end{pmatrix}$$

$$+ (0.85 \begin{pmatrix} \frac{1}{5} \\ \frac{1}{5} \\ \frac{1}{5} \\ \frac{1}{5} \\ \frac{1}{5} \end{pmatrix} \cdot \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} + 0.15) \begin{pmatrix} \frac{1}{5} & \frac{1}{5} & \frac{1}{5} & \frac{1}{5} & \frac{1}{5} \end{pmatrix}$$

$$= \begin{pmatrix} 0.425 & 0.17 & 0 & 0.085 & 0 \end{pmatrix} + (0.85 * (0.2 * 1) + 1 - 0.85) \begin{pmatrix} \frac{1}{5} & \frac{1}{5} & \frac{1}{5} & \frac{1}{5} & \frac{1}{5} \end{pmatrix}$$

$$= \begin{pmatrix} 0.489 & 0.234 & 0.064 & 0.149 & 0.064 \end{pmatrix}.$$

4.1 Implementation in MATLAB

Now that we have an intuitive understanding of the PageRank formula, it is time to implement it with computer code. The previous example shows the workings out of just one iteration of the power series, yet takes up several lines of calculation. Now, we can iterate this equation for absolutely huge graphs for many iterations automatically! Below is the important parts of the PageRank algorithm, implemented in MATLAB. The full commented code is available in the appendix.

Here, we define the `sparsepagerank` function, which takes a sparse link matrix H, as well as our value of α and residual barrier for completion, ϵ. The output of our function is our PageRank vector π, the completion time and the number of iterations for convergence.

```
function [pi, time, iter] =...
sparsepagerank(H, alpha = 0.85, epsilon = 10e-8)}
```

Next, set up the PageRank vector at iteration one. Every page begins with the same PageRank value, of length $1/n$, where n is the side length of the sparse matrix.

```
n = length(H);
pi0 = ones(1,n)/n;
```

Next, we construct a vector of all ones, and multiply it by the transposed sparse matrix to find the total number of outlinks from every page. Then, for all pages with outlinks (ever non-dangling node), we multiply each row in the link matrix by the reciprocal of their row sums to create our nearly stochastic matrix **H**. It was necessary to create a separate variable, `dividevector`, to scale the matrix because MATLAB didn't perform the operation as a sparse operation otherwise.

```
rowsumvector = ones(1, n) * H';
for i=1:n
    if rowsumvector(i)~=0
        rowsumvector(i) = 1/rowsumvector(i);
    end
end
rowsumvector=sparse(rowsumvector);
```

```
    dividevector=rowsumvector';
    H = H .* dividevector;
```

Next, find all the index of every non-zero row and from the negation of this create a vector of the indices of every zero row. Then, from this create a sparse vector **a** of ones and zeros, with ones at each zero row of **H**.

```
nonzerorows = find(rowsumvector);
zerorows = setdiff(1:n, nonzerorows);
l = length(zerorows);
a = sparse(zerorows, ones(l, 1), ones(l, 1), n, 1);
```

Finally, set up some variables necessary for iteration and timing, and perform the power series iteration. If the norm of the difference of two consecutive vectors is smaller than the threshold ϵ, terminate the program and return the results.

```
    k = 0;
    residual = 1;
    pi = pi0;
    tic;
    while (residual >= epsilon)
        prevpi = pi;
        k = k + 1;
        pi = alpha * pi * H + (alpha * (pi * a) + 1 - alpha) *...
        ((1 / n) * ones(1, n));
        residual = norm(pi - prevpi, 1);
    end
```

```
    pi=pi.';
    iter = k;
    time = toc;
    end
```

To test my MATLAB code, I ran my sparse PageRank function on the worked example for one iteration (with a residual barrier high enough as to only require 1 iteration):

```
>> H=sparse([2,1,1,3,5,5],[1,4,2,1,1,2],[1,1,1,1,1,1],5,5);
>> [pi, time, iter] = sparsepagerank(H, 0.85, 0.9);
>> pi
pi =
    0.4890    0.2340    0.0640    0.1490    0.0640
```

As this matches our manual calculations, we can now iterate our formula until $\boldsymbol{\pi}^T$ converges:

```
>> [pi, time, iter] = sparsepagerank(H, 0.85, 1e-4);
>> pi
pi =
    0.3758    0.2579    0.0689    0.2286    0.0689
```

But, does our function scale up appropriately for huge graphs? We can test this by creating a differently sized networks of pages and hyperlinks, and timing the algorithm to see if the correct relationship arises:

```
%create network of 10,000 pages with 100,000 randomly-generated hyperlinks
>> h = randi([1 10000],100000,2);
```

```
%perform PageRank and output time taken
>> [pi, time, iter] = sparsepagerank(h,10000,0.85,1e-4);
>> time
time =
    0.0036
%same with  1,000,000 links
>> h = randi([1 10000],1000000,2);
>> [pi, time, iter] = sparsepagerank(h,10000,0.85,1e-4);
>> time
time =
    0.0344
%same with 10,000,000 links
>> h = randi([1 10000],10000000,2);
>> [pi, time, iter] = sparsepagerank(h,10000,0.85,1e-4);
>> time
time =
    0.3362
```

Chapter 5

Wikipedia and Web Crawling

We now have a working MATLAB file that can calculate the PageRank of the nodes of a graph, stored in sparse matrix form. But, to generate meaningful results and analyse real-life examples, we need to be able to create a sparse matrix to analyse. How this is generated will be unique to the object being analysed. What is for certain, however, is that it will have to be largely automated. PageRank is an efficient algorithm designed to quickly calculate the primary eigenvector of huge matrices representing graphs with millions or billions of nodes and edges, so manually filling out the entries would take prohibitively long.

My chosen area of interest is Wikipedia. Wikipedia is a vast online encyclopedia, containing millions of articles, which when put together comprise the most comprehensive online encyclopedia that exists [12]. However, unlike standard encyclopedias, Wikipedia has links between related articles. Like the internet as a whole, Wikipedia can be represented as a graph, where each article is a node and links between the articles is an edge.

But, how would we be able to procure a sparse matrix representing Wikipedia articles and their links?

In just the same way that webpages on the web could be thought of as books in a library that can link between each other, Wikipedia articles are encyclopedia entries that link between each other. However, Wikipedia is online, with each article being a stored webpage. This means that Wikipedia is a subset of the web as a whole, and the same procedure that can create Google's hyperlink matrix, once slightly modified, can also do the same for Wikipedia. But what is this procedure? We introduce the web crawler.

5.1 Web Crawlers

The job of a web crawler is in its name; crawl through the web to construct our graph of pages and hyperlinks. But, how does this work? How well does it work, and are some considerations needed for optimal performance? We begin with the simplest form of the web crawler algorithm:

1. The first section of our algorithm will create a list of web pages to crawl through. To begin this process, we have two inputs: the number of pages we are going to crawl through (named n), and the page we are going to start on. To begin, create an empty sparse matrix of side length n, and a nearly empty webpage array of length n, where the first index contains a text string of our starting page.

2. Using MATLAB's `webread` function, retrieve the contents of the starter website

(typically in the form of an HTML file). Then, scan the file for hyperlinks. These will begin with `<a href=>` and end with `</a>`. Add to your array of websites each hyperlink.

3. Now, some filtering is necessary to remove some edge cases. Links to pictures and other media which isn't a website that can still be hyperlinked to need not be considered. This can be done with MATLAB's `findstr` command. Also, don't add any repeat pages already mentioned earlier in our webpage array.

4. Record all of the outlinks from our starting page on our sparse matrix. The row number will be the website currently being crawled through, and the column numbers that must be assigned a value of one will be the pages that our current page links to. By having completed step 2, we can be certain that every outward link from this site is already on our list and no link will be missed.

5. Repeat steps 2 through 4 for the second webpage on our list. This will quickly begin to fill up our webpage array with pages, since the average webpage links to more than one webpage. Once our webpage vector is full of webpages, stop adding any more but keep searching through each page in order.

6. Repeat step 4 using the `webread` function until every page has been searched for hyperlinks. Return both the sparse matrix and the webpage array and terminate.

Note this algorithm has many properties that make it, in its current form, potentially unsuitable for scraping large portions of the web:

1. The web, or even sub-sections of the web, is simply too large for one program to scour. Even if you had a hypothetical infinitely fast machine that could perform

all calculations in time order $\mathcal{O}(0)$, the algorithm must, for every page on the web, retrieve its contents from a remote server. This limits the speed of the program to the speed of your internet connection and the response time from the server each webpage is stored on. Due to the web's staggering size, pages are updated and new pages are added more quickly than one program could download. The solution to this problem is for our web crawler to be a module; a central database stores the sparse matrix and page array, and sends out individual "spiders," each of which carries out our crawling algorithm on concurrently on different parts of the web, continually updating and adding to the central database.

2. The simple web crawler algorithm is a breadth-first search algorithm. This means that the webpage array will be expected to fill exponentially faster than the crawler will travel from our starting page. If the average page has k outlinks, our webpage array will fill up after only searching for pages $\log_k(n)$ links from our starting page. This is a slight oversimplification; we're not crawling through a cycle-free tree, but the increase in the number of pages k links away from our first page is nonetheless exponential until a considerable chunk of the web has been listed. Nonetheless, an individual crawler won't stray too far from where it started. You could argue that a depth-first search would allow for a larger portion of the web to be viewed, but this would result in the crawler "skimming" through large portions of the web, skipping over the vast majority of content. The solution to ensuring broad coverage of the web is to have a crawler module working in many areas of the web, continuously updating a central model.

3. The `webread` function consumes resources on behalf of the websites being visited,

taking up bandwidth. Is it ethical to crawl the entire web with a bot? The answer to this, as with many ethical questions, can be argued from a real-life standpoint. Many bots, from the helpful to the malicious, retrieve data from the web. However, they often differ in what information they obtain. Webpage owners may instruct crawlers which parts of the website they can retrieve by including a `robots.txt` file that tells a crawler what parts of the webpage (or pages within an origin) the crawler should access. These were designed with web crawlers in mind, allowing access to content deemed relevant and non-private. This is, however, just an instruction and can be ignored [13]. I would personally argue that a web crawler is ethical if it chooses to follow the `robots.txt` file, and not following the instructions should only be done for purposes of archiving the internet.

4. Finally, the program will only be able to find articles by crawling in the same way that an internet surfer would, but without the option for starting at random once a dangling node is hit. This alone isn't going to break the crawler, which will simply move to the next page to search. However, if all pages left to search are dangling nodes and fewer than n pages have been discovered, the program will cease to execute before finding n pages. The solution to can range from relatively easily solvable to completely unsolvable, depending on the nature of the network you're dealing with. This is because unlike the other issues discussed, is inherent to the network being analysed. What really matters is how connected the graph being analysed is.

5.2 The Connectivity of Graphs

So far in our crawling algorithm, we have assumed that the graph we are crawling through is *strongly connected.* This means that any node can be accessed from any other mode, which means that every node on the graph (or page on the web) will be found by our crawler no matter where on the web we start. This, however, is rarely the case in graphs of real-life networks. The proof here is by counterexample: the existence of a dangling node means the graph is not strongly connected, because a node with no outlinks cannot reach any other node. (here is a web counterexample: `https://upload.wikimedia.org/wikipedia/commons/6/6f/Googlematrixwikipedia2009.jpg`).

The next-best case would be if the web was *weakly connected*, which is a graph in which every node can be reached if you ignore the directions specified for each edge. An example of this would be a *directed binary tree,* where all nodes can be crawled to by starting at the root node, but no other. In the case of a weakly connected graph, every node could be crawled, provided that the address of every page was already known. This makes crawling through the web nearly much more difficult, as without knowing every site on the web, you cannot know if every site has been crawled. This could result in fringe webpages with few inlinks being passed over. Fortunately for these webpages which may want to be seen and indexed, Google and most other search engines have submission pages that allow for addresses to be registered. By having multiple starting nodes and many spiders crawling concurrently within a module, large portions of the web (and importantly the most connected sites) can be indexed and re-indexed frequently.

There is, however, a case we haven't discusses: *isolated nodes.* The web isn't even weakly connected. Some webpages have an address which can be entered manually, but link to

no other webpages and have no webpages link to them. Those which don't (or specifically choose not to) submit their page to be indexed by a search engine will never be found by a crawler module, ever. In fact, about 95% of the web is thought to be lost to search engines, a small portion of which don't want to be found, being available only to those who can obtain the address by other means [14]. But, does this matter? Most of the web is well-connected enough that nearly every page that someone might want to find is connected within a singular *giant component*. Even if crawlers can't index most of the web, the nature of hyperlinks work in favour of the crawlers, allowing them to find nearly all the useful content, making search engines possible.

With these problems taken into consideration, how does web crawling scale down to Wikipedia? Also, are there any further restrictions that must be accounted for in the context of this project?

5.3 Crawling through Wikipedia

There are a number of limitations that this project has to take into account when attempting to create a web crawler. The first, and biggest, is the lack of ability to create multiple crawlers working at once. I only have access to two computers at a time, neither of which are on the same network. Furthermore, I can only run a program for a maximum length of a few hours in the background before the day's end. Nonetheless, Wikipedia may be possible to crawl through. So, I created my own web crawler on MATLAB, and modified it to only search through Wikipedia articles. The following code was altered from a template included in *Google's PageRank and Beyond* (the whole MATLAB file

contents can be found in the appendix):

Iterate the following code for every instance where the start of a hyperlink between Wikipedia articles is found within the HTML file of the article currently being crawled:

```
for f = findstr('<a href="/wiki/',page)
```

Find the title of the article (some intra-Wikipedia links aren't to articles so must be discarded, and 100 characters is a catch-all for finding all titles within hyperlinks without passing to the next hyperlink). If no title, pass to next hyperlink.

```
e = min(findstr('" title="',page(f:f+100)));
if isempty(e), continue, end
```

From title, create a string that represents the HTTP address of the page. This involves concatenating the start of the address before the title, ridding any non-printable characters (such as non-ASCII characters or control characters), and deleting any unnecessary training forward slashes.

```
url = strcat('https://en.wikipedia.org',...
deblank(page(f+9:f+e-2)));
url(url<' ') = '!';
if url(end) == '/',url(end) = []; end
```

Next, consult a list of strings that, if occurring in the web address, should be overlooked. This includes images and other non-article items, discussion pages, article categories, and other miscellaneous Wikipedia artefacts. If an address contains one of these strings, it isn't an article and shouldn't be included in our list of Wikipedia articles or checked for hyperlinks.

```
skips = {'.gif','.jpg','.pdf','.css','lmscadsi',...
'cybernet', 'Special', 'Wikipedia:', 'Portal:',...
'Help:','Talk:','Main_Page','File:', 'Template:',...
'Category:','talk:'};
skip = any(url=='!') | any(url=='?') | any(url=='#')...
| any(url=='"');
k = 0;
while ~skip && (k < length(skips))
    k = k+1;
    skip = ~isempty(findstr(url,skips{k}));
end
if skip
    if isempty(findstr(url,'.gif')) &&...
    isempty(findstr(url,'.jpg'))
        %disp(['      skip ' url])
    end
    continue
end
```

5.4 Complexity of Crawling

Of the two inputs into our crawler algorithm, only the matrix size, n, will affect the execution time of the program in a way we can measure.

The starting page does affect the completion time. Some articles are longer than others and may connect to other long articles, increasing the amount of data to be accessed and crawled through. Some articles have no outlinks, or may link only to dead-ends (dangling nodes), which would cause only a tiny sub-graph of Wikipedia to be documented before the crawler terminates. However, neither of these properties affect the execution time in a measurable manner, as in the context of the input article the only possible way you could even measure complexity would be a function of the average length of the articles crawled or the number of outlinks (both of which would theoretically a linear combination of execution times of different sections of the algorithm).

The size of the sparse matrix n determines how many pages are crawled through. Whether or not the array of pages is full or not, the crawler must retrieve and read the whole article and produce a list of all intra-network links in order to fill the sparse matrix; this part of the algorithm operates in linear time. Even though the length of the list increases with time, which would imply a linear increase in time taken to check if a page had been crawled yet (and therefore a quadratic increase in total time spent doing so), we employ a hash function that decreases the average case search time to order $\mathcal{O}(1)$ and total search time to order $\mathcal{O}(n)$. Theoretically, our crawler has execution time order $\mathcal{O}(n)$. But, does this scale up practically? I ran my crawler module on Wikipedia ten times for ten different values of n from 10 to 100, and plotted the average execution time (see figure 5.1). In this case, I chose the same starting article for each run, so that the crawler crawls through the same articles in the same order, to reduce the influence of the starting article. I chose the starting article to be the United States of America, which is a large article with plenty of links to other large articles (reducing the variance in article length as a proportion of the

length of each article).

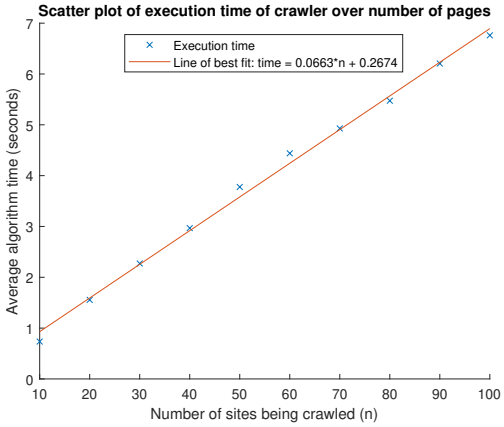

Figure 5.1: A scatter plot of the mean execution time of the crawler module for different values of n and the same starting page.

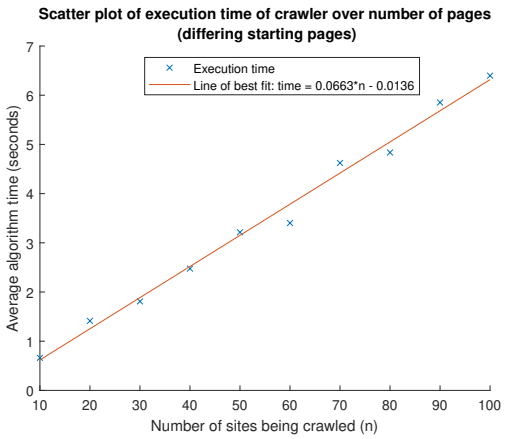

Figure 5.2: A scatter plot of the mean execution time of the crawler module for different values of n and with different starting pages.

Also plotted is a scatter plot for the same simulation, but instead choosing a random article from a list of The first 100 articles found starting from the article for Skirwith Abbey, a small article about a country house in Cumbria that links to a mixture of small and large articles (see figure 5.2). Here, the variance in execution time is slightly larger due to the difference in starting articles, but the line of best fit has the same gradient, which suggests the starting page won't have an effect on the running time for large values of n.

Figures (5.3) and (5.4) show the crawler's performance on larger values of n, without taking a mean of 10.

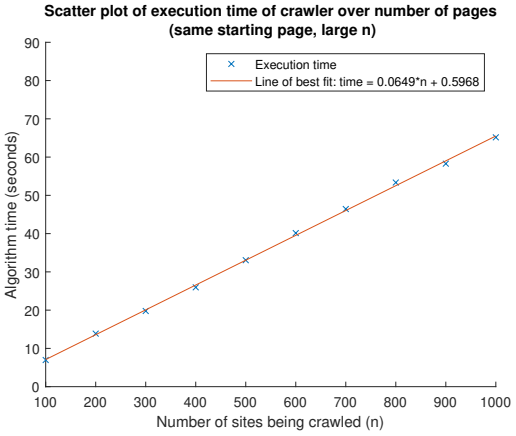

Figure 5.3: A scatter plot of the execution time of the crawler module for different large values of n and with the same starting page.

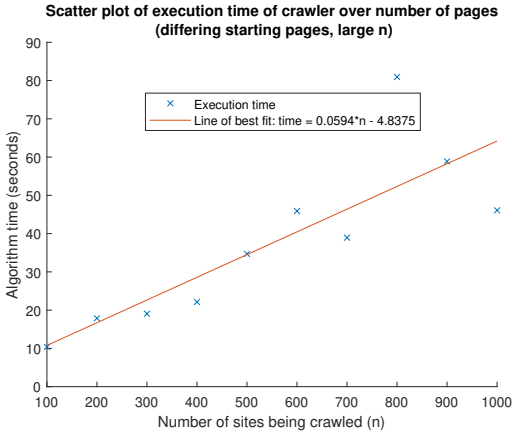

Figure 5.4: A scatter plot of the execution time of the crawler module for different large values of n and with different starting pages.

As expected, the variance is much larger for the different page starts due to not taking a mean of 10, but again the average time to crawl a page has remained steady.

From inspection, the execution time is linear (for small values of n), taking about 0.06 to 0.07 seconds to index a page. According to Wikipedia itself, there are 6,656,201 articles on the English Wikipedia as of the 14th May, 2023 [15]. Assuming linear execution time holds and indexing all of English Wikipedia (and that nearly every article in Wikipedia is connected in one giant component), my crawler module would take approximately $0.065 * 6,656,201 = 432653.065 \approx 433000$ seconds, or 5 days. Unfortunately, this is too long for my program to run with the resources I have. This can't be changed with having a faster computer, since the time component of the crawler spent retrieving data from Wikipedia is roughly 60% of the total time, meaning that even with a theoretically infinitely fast machine approximately 3 days' time would be spent waiting for the page contents to be retrieved.

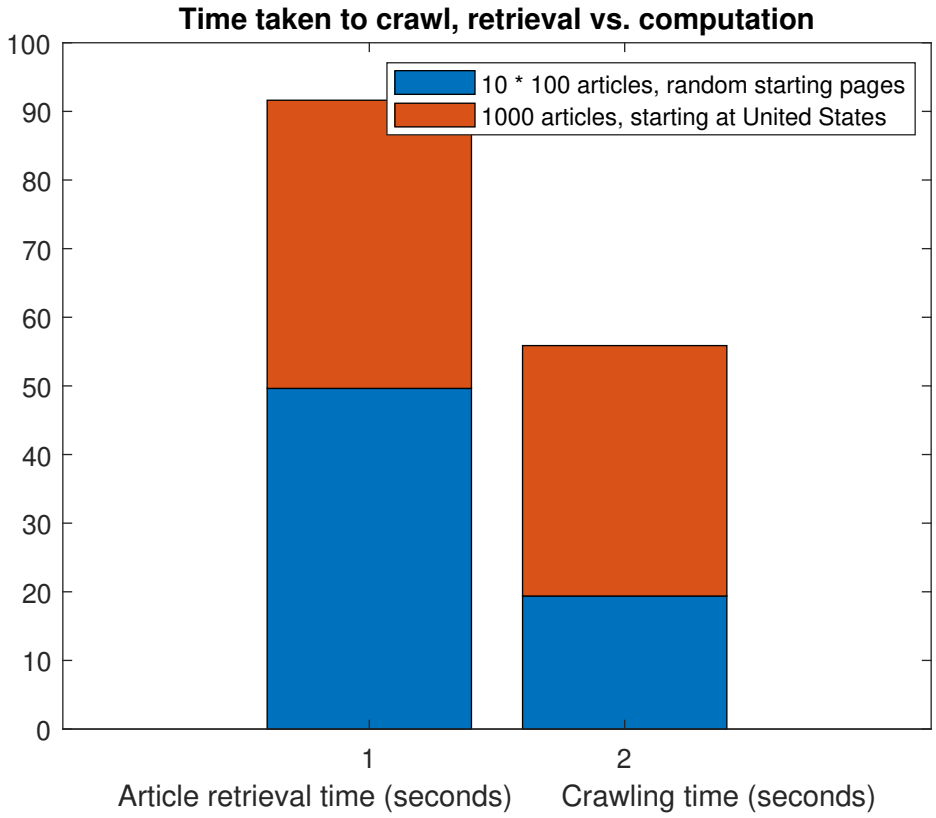

Figure 5.5: A bar graph of the time taken for different parts of the crawling algorithm. The article retrieval time depends on the speed of my internet connection and Wikipedia's servers, can't be shortened.

Wikipedia regularly dumps data on its contents, including page-to-page links, but unfortunately these are stored in `.sql` format, which I couldn't get to open, even after two weeks' of attempting to do so. Fortunately, I'm not the first person to try crawling through Wikipedia's links on MATLAB. Professor David Gleich of Purdue University had a copy of the hyperlink matrix of all of Wikipedia's articles in sparse matrix form on hand from 2007, along with a vector of the articles' names. I would like to express my gratitude to David for kindly responding, even 16 years after creating these files [19,20].

Chapter 6

Wikipedia's PageRank Vector

6.1 Article PageRank Values and Qualitative Analysis

Now that we have a sparse matrix of Wikipedia's inter-article hyperlinks, we can finally perform our PageRank algorithm in MATLAB to produce a PageRank vector! For this very first run, I will be using $\alpha = 0.85$ to simulate a user surfing through articles like a web user:

```
>> H = load("wikipedia_sparse_matrix.mat");
>> H = H.Problem.A; %load sparse matrix from file
>> article_name = readlines("wikipedia_articles.txt");
>> [pi, time, iter] = sparsepagerank(H, 0.85, 1e-8);
>> pagerank = table(article_name, pi);...
% join article names with pagerank value
>> pagerank = sortrows(pagerank, 2, 'descend');...
```

```
% sort in descending order of pagerank value
>> pagerank(1:10,:)
>> time
>> iter
                    article_name           pi
-----------------------------------    ----------

"United States"                          0.0027517
"Category:Categories"                     0.001345
"Category:Wikipedia administration"      0.0013297
"United Kingdom"                         0.0011497
"Category:Redirects"                     0.0011401
"Race (United States Census)"            0.0010323
"Category:Society"                      0.00098213
"Category:Tracking categories"          0.00097056
"Category:Living people"                0.00092971
"Category:Categories by country"        0.00092306
```

From inspection, PageRank has worked! In what metrics, though?

In the most literal way, the calculation completed and gave us a vector that sums to a value of one.

Ina less easy to quantify way, PageRank has worked because articles with the highest PageRank values are clearly well-connected articles. Nine of the top 10 can be cate-

gorised into one of two things: important topics and lists.

The two articles that could be classified as "important topics" are the articles about the United Kingdom and the United states. Both are large, English-speaking nations with long and well-established presences on the world stage, both of which generally haven't suppressed freedom of information compared to the rest of the world. What ranks them above every other topic isn't necessarily quantifiable but is tangible. Humanity, as well as its rate of information creation, has grown exponentially in the last few hundred years. Many of the advancements made that shrank and connected the world, such as industrialisation and colonialism (for better or for worse) are inextricably tied to the United Kingdom, and the United States was the dominant world power for the latter half of the 20^{th} century, when the majority of information that exists today was created. This is before mentioning the context of English Wikipedia, an American-created site written in English, mostly by English-speaking people mostly residing in the UK or the US.

Most of the rest of the articles fall into the "lists" category. To understand why this makes them well connected, I propose a model that orients the graph of Wikipedia (and all information): the connected tree. My proposed model borrows off of the same principle of the Dewey decimal system, but also draws from how information is linked. Suppose the aggregate sum of all human knowledge is placed on a scale from 0 to 1000, and iteratively partitioned into categories and sub-categorised down to the granular level. Each level of categorisation is linked up to one or more parent super-categories and down to their child sub-categories, as well as to related topics within their layer (as information cannot be strictly categorised and seemingly separate topics can be intrinsically linked).

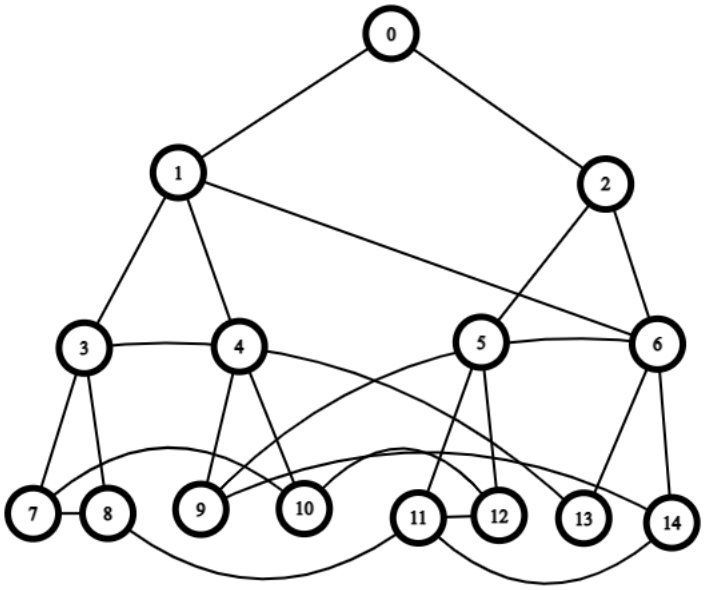

Figure 6.1: An example graph illustrating a simplified version of what my model of information would look like. Although the link between information isn't necessarily directed, if the nodes represented Wikipedia articles, the edges would be directed up the tree.

It would follow, from this model, that the upper layers of this tree would link directly to more topics below them than layers further down, increasing their overall connectivity. Bringing this to its natural logical conclusion, the node at the very top layer would be the most well-connected one. Looking back at our top 10, the article Category:Categories, or

our list of lists, linked up to from all the individual articles in their categories section, rests above all except the United States. Following just behind are categories encompassing large swathes of information (such as society), and a few which are integral to Wikipedia's structure (such as tracking categories).

In fact, the only outlier appears to be the article about "Race (United States Census)". This article doesn't exist anymore and was never archived, so not much about this page can be said. Although clearly close in topic to the United States, this alone doesn't explain such a high ranking. Perhaps there was an error in the data collection, allotting this page a much higher ranking than its true value (or a different article's PageRank value was attributed to this one).

However, this is the list produced from using Google's value of α. Should Wikipedia have the same value of α, and does altering the value of α work in practice the same as in theory?

6.2 Altering the Value of α

The inclusion of the uniform matrix $(1-\alpha)\frac{1}{n}\mathbf{e}\mathbf{e}^T$ is necessary to guarantee convergence of our Google matrix in a reasonable amount of time, and also models real web surfing more realistically. A high value of α increases the computation time, which for incredibly large matrices becomes computationally prohibitive. However, the lower the value of α, the less closely the resulting PageRank vector actually reflects the internal structure of the web, defeating the whole point of this exercise. But, how does changing the value of

α change our resulting PageRank vector?

Let's start with the obvious: a value of $\alpha = 0$ will snap to the uniform vector of $\frac{1}{n}\mathbf{e}^T$. If we start with $\boldsymbol{\pi}^{(0)} = \frac{1}{n}\mathbf{e}^T$, then $\|\boldsymbol{\pi}^{(1)} - \boldsymbol{\pi}^{(0)}\| = 0$, and the program will terminate after one iteration. If $\boldsymbol{\pi}^{(0)}$ is a sufficiently different probability vector, the program will iterate a second time, then $\|\boldsymbol{\pi}^{(2)} - \boldsymbol{\pi}^{(1)}\| = 0$ and the program terminates after the second iteration. Regardless, setting $\alpha = 0$ doesn't reveal any ordering of pages. But, once we begin to increase the value of α, the internal structure of Wikipedia begins to alter the PageRank values of each article, and some of the properties of the PageRank vector:

Value of α	**Maximum PageRank value**	**Standard Deviation of PageRank vector**	**Number of iterations**	**Calculation time (s)**
0	7.4×10^{-8}	0	1	0.2472
0.1	2.8×10^{-4}	3.46×10^{-7}	7	1.789
0.5	1.6×10^{-3}	1.94×10^{-6}	19	4.896
0.85	2.8×10^{-3}	3.91×10^{-6}	67	17.18
0.99	8.7×10^{-3}	1.02×10^{-5}	791	208.3
0.999	1.1×10^{-2}	1.27×10^{-5}	7884	2,127

Table 6.1: A table showing some stats about the PageRank vectors of different Wikipedia articles.

Table 6.1 shows some general trends, which match what they're expected to do in context. The ratio between completion time and iteration count is roughly 0.26 for every value of α, which reflects that the same number of calculations are being performed each iteration. Also, as α, increases, the maximum PageRank value increases as well as the standard deviation. This makes sense, as the PageRank values tend to be damped by the introduction of the uniform matrix, which increases the PageRank value of nodes with no inlinks. So, when α increases, the PageRank value of the nodes with no inlinks tend towards zero, and the PageRank of the other pages increases as a result. The number of

iterations also increases exponentially as $1-\alpha$ decreases logarithmically towards 1 (when α is ten times closer to one, you can expect the number of iterations to increase roughly tenfold). But, just how different does Wikipedia's PageRank vector change between different values of α? How much do the PageRank values change, and do the ranks change as well?

Average difference in PageRank value ($\times 10^{-7}$) between different values of α	**0**	**0.1**	**0.5**	**0.85**	**0.99**	**0.999**
0	0	0.364	1.86	3.43	4.31	4.39
0.1		0	1.54	3.19	4.15	4.13
0.5			0	1.79	2.91	2.95
0.85				0	1.25	1.36
0.99					0	0.161
0.999						0

Table 6.2: A table showing the average difference in PageRank value between vectors calculated from different values of α.

Table 6.2 shows that when compared to the uniform matrix, a bigger value of α results in a more pronounced difference in PageRank value. Also, the closer the value of α gets to one, the more quickly the values in the PageRank vector change.

For Google, larger differences in PageRank value result in PageRank contributing a bigger weighting in their search engine. This is because the results Google shows is a product of a page's PageRank value and another score, calculated with traditional search engine methods (e.g. whether the words in the search term match the contents of the page). As a result, a bigger PageRank difference will cause a more dramatic change in the combined

score. However, a large change means nothing more than a tiny change in practice. When seeing search results, you simply get returned the results in order regardless of the actual size of the difference is, meaning the rank order of the pages is what really matters. So, how similar are the ranks of the different PageRank vectors?

Average difference in PageRank rank	**0**	**0.1**	**0.5**	**0.85**	**0.99**	**0.999**
0	0	952,589	938,624	925244	918,600	918,458
0.1		0	92,335	198,438	305,303	321,729
0.5			0	114491	233,334	251,888
0.85				0	127,030	147,472
0.99					0	21,665
0.999						0

Table 6.3: A table showing the average difference in PageRank rank between vectors calculated from different values of α.

To make sense of the rank difference table, we start by looking at the $\alpha = 0$ row. When $\alpha = 0$, the PageRank value of each page is equal, the every page has the same rank of of $0.5 * (n+1) = 1783454$. So, any significant kind of change in rankings should see each rank change by roughly $\frac{n}{4}$. This is shown to be the case, where the average rank change stays relatively near that value. The rank indexing system I use doesn't allow for joint rankings to hold the same value, which is why the average rank change between $\alpha = 0$ and the other vectors isn't exactly $\frac{n}{4}$.

What is also clear is that yes, as $\alpha = 0$ increases, the ranks do change rather significantly! Right up until $\alpha = 0.99$, the average page jumps up or down by about 3%. After $\alpha = 0.99$, the ranks begin to settle more, suggesting a value very close to one won't change much more. As setting α very close to one is very computationally expensive ($\alpha = 0.999$ took 35 minutes on a fast computer, and Wikipedia is comparatively much smaller than

the web), I would argue that values above $\alpha = 0.99$ don't give enough return on their computational effort to bother with.

Now we have some different PageRank vectors to look at, we can also analyse the top articles of each to see if any trends emerge with changing α.

Value of α → Rank number↓	**0.1**	**0.999**
1	United States	Category:Wikipedia administration
2	Category:Living people	Category:Redirects
3	2006	Category:Tracking categories
4	England	Category:Categories
5	2005	Category:Unprintworthy redirects
6	United Kingdom	Category:Redirects from shortcut
7	Canada	CAT:WIKI
8	2004	CAT:WP
9	France	Category:Fictional characters by superhuman power
10	Australia	Category:Superhuman powers

Table 6.4: A table showing the top ten articles of the PageRank vectors for $\alpha = 0.1$ and $\alpha = 0.999$.

Table 6.4 shows some rather interesting results! With a low value of α, the structure of the web isn't used much. Few iterations are required for convergence, and the damping effect means that the nodes with no inlinks have a value 87 times greater for $\alpha = 0.1$. Consequently, the top few pages are very similar to list of the pages with the most inlinks (see table 6.5 below). Another type of page arises from an artefact of the Wikipedia matrix: the link matrix was generated in 2007. As a result, the recent years before 2007 populate the top-ranked articles.

	Top 10 articles by sum of direct inlinks	**Total weighted inlink sum**
1	United States	9304.6
2	Category:Living people	8825.8
3	England	3974.2
4	2006	3873.9
5	2005	3499.4
6	Canada	3460.5
7	United Kingdom	3405.0
8	2004	3080.1
9	France	2712.1
10	Australia	2689.3

Table 6.5: A table showing the top 10 pages by the sum of inward links (the inward link values are weighted by the total outlinks each page linking to it).

In fact, the top article for $\alpha = 0.999$ ranks 5796th for its inlink sum, and 2049th on the $\alpha = 0.1$ PageRank vector. This is due to the fact that the top 10 results for the value of $\alpha = 0.999$ is very different indeed! Every single article is a category, with even the mighty United States falling to 13th place. This is because PageRank is a measure of centrality, and category articles, although not always directly linked to as much as popular articles, are central by definition (the top rows of figure 6.1 would be categories).

So, what should Wikipedia's value of α be? Since Wikipedia doesn't need a search function like Google (since each topic has exactly one page devoted to it, you only need to use regular search techniques), I would argue that a value as close to one as possible is better. I believe the true centrality of articles is a better use of PageRank on Wikipedia than trying to closely model a surfer. With $\alpha = 0.999$ being close to the reasonable time limit, and getting rid of the categories (which are slightly disingenuous answers to the question, "What's the most important topic?"), we finally arrive at an improved list of the Wikipedia articles with the highest PageRank values:

	Top 10 articles ($\alpha = 0.999$, categories excluded)
1	Wikimedia Commons
2	United States
3	United Kingdom
4	France
5	England
6	2005
7	2006
8	Human
9	Germany
10	Canada

Table 6.6: A list of the non-articles articles with the highest PageRank values, where $\alpha = 0.999$ is very close to one.

Here, we find the final two novel articles. Wikimedia Commons is a repository of free-use, public domain media used in articles, and is linked to by lots of articles containing media. Finally, there's one more article that is helped by centrality instead of direct inlinks, one that encompasses and links all information ever collected: humans.

Chapter 7

Conclusion

My objective for this project was to implement the PageRank algorithm on the graph of Wikipedia articles, as well as provide an in-depth look at PageRank itself. This included how and why the PageRank algorithm came to be, how PageRank works and what modifications are needed to ensure the same results are generated, and within reasonable time.

To this end, I have discussed the history of information retrieval, from the first written records to the invention of the internet, and how the advancement of information storage made traditional search methods necessitated the creation of PageRank. Then, I derived the PageRank algorithm from some basic centrality principles, discussed some potential implementation problems, and made alterations to the original algorithm that are proven to result in convergence to the same, normalised PageRank vector in a feasible amount of time, given any probability vector $\boldsymbol{\pi}^{(0)}$ and a hyperlink matrix $\mathbf{H}$. Next, I implemented PageRank in MATLAB, as well as a web crawling module in attempt to create a hyperlink matrix of Wikipedia. Although both algorithms worked as expected, the web

crawling module was restricted in speed by the time taken to retrieve information from the internet, and I instead obtained my Wikipedia sparse matrix from somebody who had already created one. Then, I performed the PageRank algorithm on the Wikipedia sparse matrix, and performed a qualitative analysis on the results. The articles with the highest PageRank values fell into two categories: category articles that are central in Wikipedia's deliberately organised internal structure, and articles of broad or important topics, linked to directly by masses of related articles. Finally, I tested the effect of changing the value of α on the resulting PageRank vector, and suggested what value of α was most suitable for analysing Wikipedia. The closer the value of α to one, the more iterations were required for convergence, and the larger the derivative $\frac{d\boldsymbol{\pi}(\alpha)}{d\alpha}$ became. Low values of α resulted in the broad topics with plenty of direct links dominating the top spots, while values of α approaching one revealed that the category articles were truly the most central articles, once the damping effect of a low value of α was removed.

In addition to completing this project, I also gave a 20-minute presentation about Google's PageRank algorithm, inspired by this project, at the 2023 Manchester Interdisciplinary Mathematics Undergraduate Conference.

However, there were some results I generated that I was unable to fully analyse in this project. Firstly was analysing the distribution of the values within $\boldsymbol{\pi}$. For higher values of α, the distribution of the values in $\boldsymbol{\pi}$ appeared to follow a *Zipfian distribution*. The Zipfian distribution is a discrete probability distribution that works as follows: assign a value of 1 to the most likely event. Then, for the nth most likely event, assign a value of $\frac{1}{n}$. Finally, normalise the values so that they sum to one. This inverse proportionality

to rank is observed in many other contexts, including frequency of words in the English language, personal incomes, and even user-chosen PINs. One possible explanation is *preferential attachment,* or cumulative advantage, where articles with a large number of links are more likely to be linked to purely on account of that more-linked to articles are more well-known of. Popular articles are often the longest and most well-cited (which makes them more likely considered worth linking to), and could be linked to more purely on account of their already existing popularity. Perhaps this isn't the case, and instead the topics behind each article just happen to naturally display this characteristic. However, this distribution of PageRank values isn't intrinsically linked to the PageRank algorithm itself (as all distributions are normalised anyway, like $\boldsymbol{\pi}$) and I couldn't find anything to prove that this is necessarily the right distribution apart from conjecture and visual similarity (see figure 7.1).

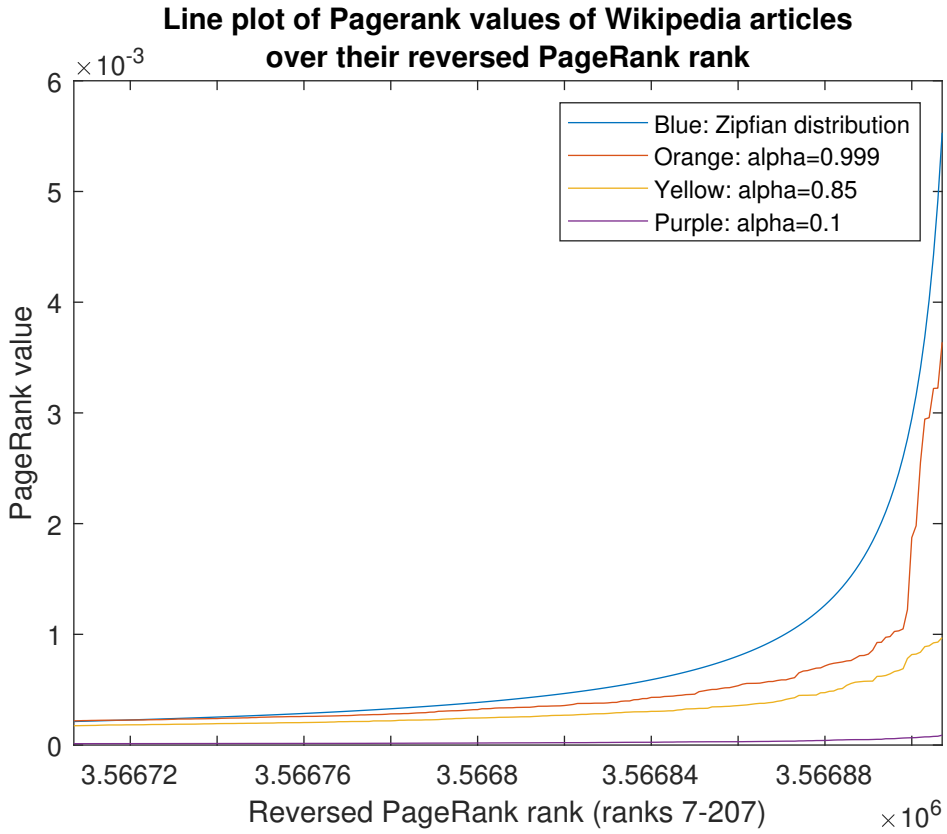

Figure 7.1: A line plot of the PageRank values over the reversed PageRank ranks, for different values of α. Also shown is the Zipfian distribution, which the sorted PageRank vector appears to approach as $\alpha \to 1$. The top 6 and bottom 3.5667 million ranks are omitted to better show the shape of the distributions.

Another thing I hoped to do was visualise the huge graph that is Wikipedia. With over 3,500,000 articles and over 45,000,000 total links, MATLAB's digraph plotter was completely unable to handle anything anywhere near that size graph (by several orders of magnitude), instead producing indecipherable, useless figures for any graph with more than a few dozen nodes (see figure 7.2).

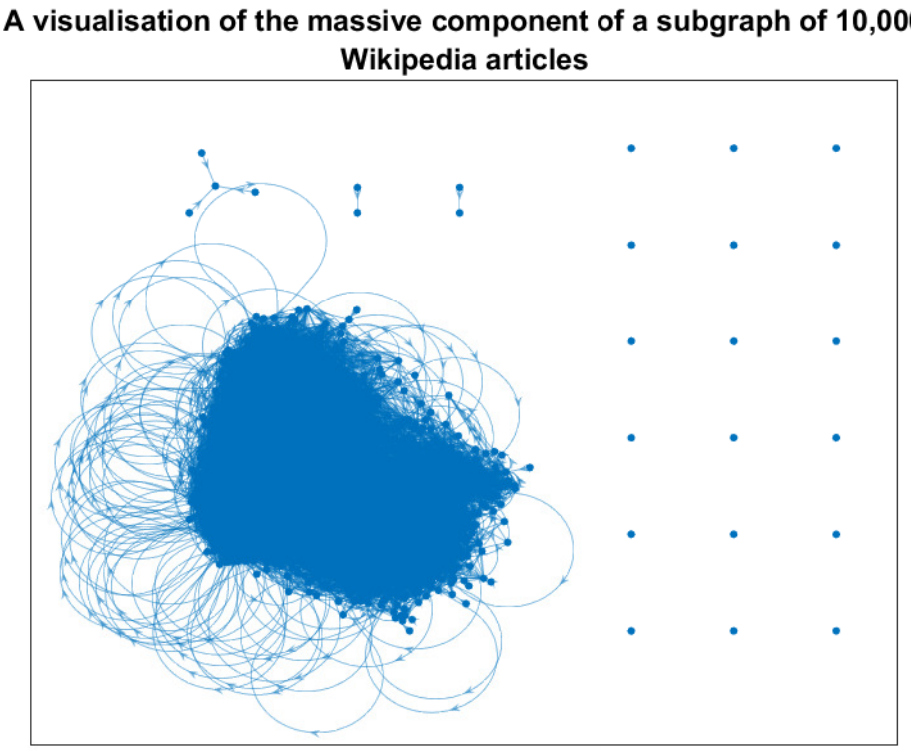

Figure 7.2: A zoomed-in graph plot of a small subgraph of Wikipedia (10,000 articles and 61,398 directed edges). Problems with this visualisation are the thousands of disconnected components that only link outside the subgraph (which trail about a metre off the top and to the right of this figure), and the single giant component which only displays a jumbled mass of nodes that form a solid blue mass. The content of this graph is both untrue to Wikipedia's actual form and completely useless as a form of data visualisation.

However, there are some other pieces of software that are specially created for large data visualisation [16], which can display vast amounts of nodes in a layout that can still present useful information about the graph. Due to Wikipedia being, on average, more well connected than most of the example graphs, the program could only display a subset of 100,000 nodes (see figure 7.3).

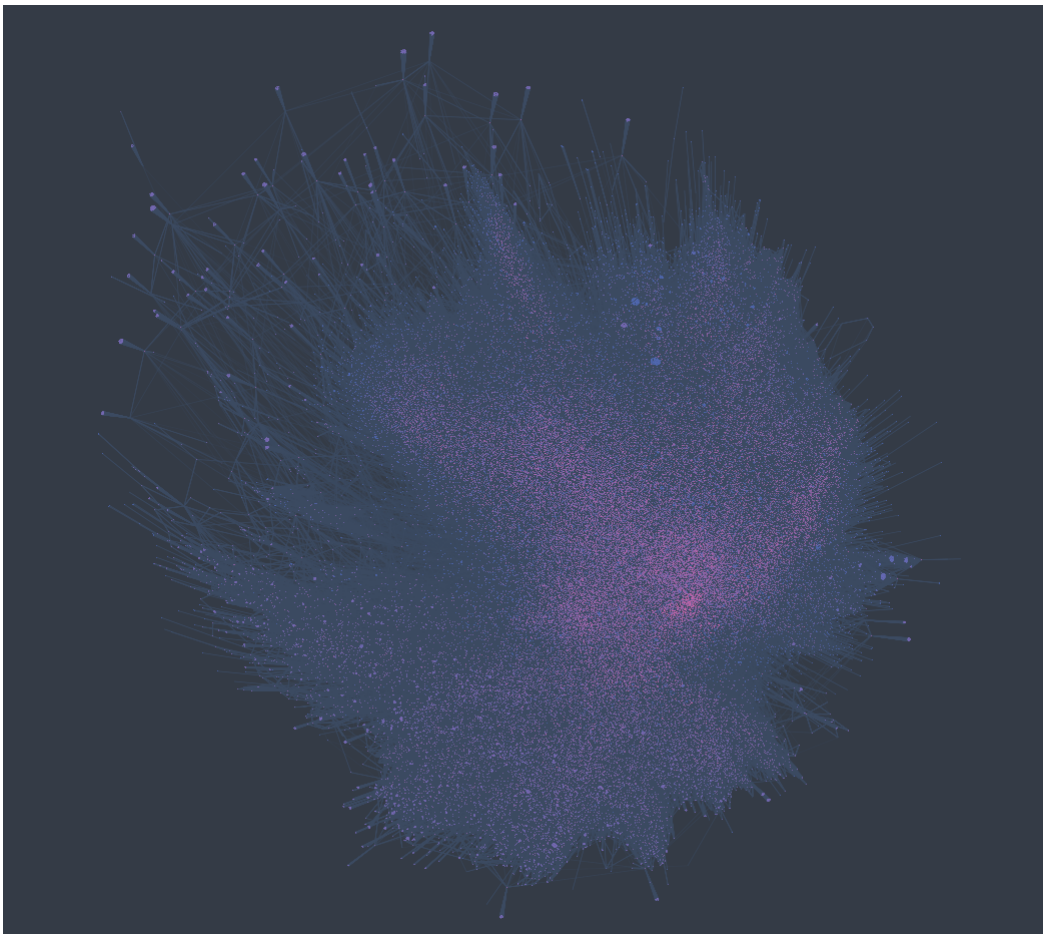

Figure 7.3: A graph of a subset of Wikipedia containing 100,000 nodes and 2,018,969 edges. This graph visualisation is much better!

The program I used, *Cosmograph*, visualises the graph by making nodes repel each other, and edges attract connected nodes through tension. As a result, the visualisation begins to take shape, with giant component forming a bulging mass of tension and repulsion (see figure 7.3), with more connected nodes being pulled to the centre and peripheral nodes being forced out towards the edge (see figure 7.4).

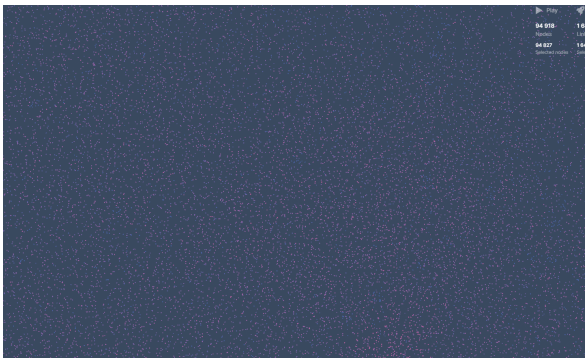

Figure 7.4: A zoom-in of the centre of the graph from figure (7.2). As with the MATLAB plot, there is a huge mass of deeply-connected articles where links between are so frequent that they are indistinguishable from each other.

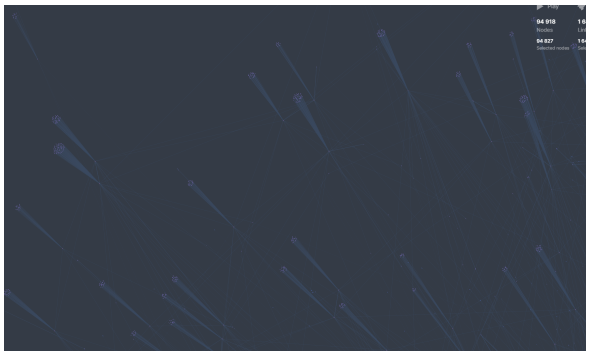

Figure 7.5: A zoom-in of the edge of the graph from figure (7.3). Here, clusters of isolated articles collectively link to only one other article, and "trail off" from the center of the graph.

Despite being a much improved visualisation compared to the MATLAB standard graph visualiser, Cosmograph could only render about 3% of Wikipedia, so the true structure is still obscured.

There were also two other features that I would, given more time, would like to have analysed and implemented. Firstly, I would have liked to test the convergence rate for different starting probability vectors $\boldsymbol{\pi}^{(0)}$. Although any probability vector will certainly converge to $\boldsymbol{\pi}$, will $\boldsymbol{\pi}^{(k)}$ converge more quickly if we create heuristic rough estimate of $\boldsymbol{\pi}$ and set $\boldsymbol{\pi}^{(0)}$ to that value? If we make $\boldsymbol{\pi}^{(0)} = [1, 0, 0, ..., 0]$, will $\boldsymbol{\pi}^{(k)}$ converge much more slowly, due to the entire probability space starting in a one article, and how much would the starting article change the convergence time?

Given more time, I also would have analysed the *HITS algorithm*, which separates outlinks and inlinks to generate two different scores: the *authority score* $\mathbf{x}^{(i)}$, which measures

the centrality of a page only from its inlinks, and the *hub score* $\mathbf{y}^{(i)}$, which measured the centrality of a page only from its outlinks. In eigenvector problem form, the HITS algorithm finds a numerical solution to the following two systems of equations:

$$\mathbf{x} = \mathbf{H}^T\mathbf{H}\mathbf{x} \tag{7.1}$$

$$\mathbf{x}^T\mathbf{e} = 1 \tag{7.2}$$

$$\mathbf{y} = \mathbf{H}\mathbf{H}^T\mathbf{y} \tag{7.3}$$

$$\mathbf{y}^T\mathbf{e} = 1 \tag{7.4}$$

Had I implemented the HITS algorithm, I would predict that the articles with the highest authority scores would have been the category articles, which don't link out to many articles but are linked in to (directly and indirectly) by every article belonging to that category. On the other hand, the highest hub scores would have been the important topic articles (e.g. *The United States*), on account of those articles being among the longest articles with the most outlinks. Another category of article not yet seen in the top PageRank lists, is the *list articles.* These articles, unlike category articles, do have large numbers of outlinks to sub-topics, making them a practical hub for human use, and potentially good candidates for articles with high hub scores.

After all this analysis, we arrive at the final question: is PageRank effective? Are the generated results useful for ordering webpages in a way that drastically cuts down the required expertise and time needed to find the information you're looking for? Well, Google became the most-used search engine by market share in 2004 currently has a 93% market

share of the search engine market [17]. Although it is unclear how large a role PageRank plays in Google's search engine algorithm today, it certainly was effective enough to propel Google above other much more popular search engines at the time, to the point of having a virtual monopoly on the market today. But what about Wikipedia?

In my opinion, implementing PageRank on Wikipedia isn't about searching efficiency. To find information on a topic, you simply look up said topic, and choose the top result. This is because an article is a collated collection of all relevant information on a topic in ordered, well-referenced prose, eliminating any need for choosing between different sources. Instead, implementing PageRank on Wikipedia can instead reveal insights into the very nature of information. Knowledge, in isolation, is only as useful as the direct practical application of said knowledge. Much more important, in my opinion, are the links between pieces of information. A good working knowledge of the structure of information allows for a greater understanding of the world at large, and learning one new thing often allows for endless new comparisons and connections to be made. In fact, information in our world is so interconnected that the *diameter* of Wikipedia (the longest distance between any two nodes on the Wikipedia graph) is believed to be just 7 [18]! Despite the amount of available information out there, and the staggering amount of effort required to create an effective tool to sort through the information relevant to what you're looking for, we can still link the two most separate topics through just 6 other topics, all of which connect in a chain of close relevance. What PageRank reveals, through its application to Wikipedia, is an unparalleled insight to the very nature of information itself.

Chapter 8

Appendix

8.1 Link to the sparse Wikipedia matrix

A downloadable copy of the sparse Wikipedia matrix can be found here: `https://www.cise.ufl.edu/research/sparse/matrices/Gleich/wikipedia-20070206.html`

The accompanying list of article names can be found as a MATLAB mat-file here: `http://www.cs.purdue.edu/homes/dgleich/scratch/enwiki-20070206-pages-articles.pages`

If the code for the `sparsepagerank` function crashes at the line:
`H = H'.* sparse(rowsumvector)';` please email me and I will be more thank happy to send you the normalised Wikipedia matrix that will work for the `pagerank_stochastic` function.

8.2 MATLAB code for PageRank

PageRank function for sparse hyperlink matrix H:

```
function [pi, time, iter] = sparsepagerank(H, alpha, epsilon)
%INPUTS
%pi0 = starting PageRank vector
%H = sparse matrix
%n = size of H
%alpha = scaling parametervect(:,1:5
%epsilon = convergence tolerance
%OUTPUTS
%pi = PageRank vector
%time = time taken to calculate PageRank
%iter = number of iterations taken to calculate PageRank
n = length(H);
pi0 = ones(1,n)/n;
rowsumvector = ones(1, n) * H';
for i=1:n
    if rowsumvector(i)~=0
        rowsumvector(i) = 1/rowsumvector(i);
    end
end
rowsumvector=rowsumvector';
%Line below sometimes performs non-sparse operation; use...
```

```
%other file if so
H = H'.* sparse(rowsumvector)';
nonzerorows = find(rowsumvector);
zerorows = setdiff(1:n, nonzerorows);
l = length(zerorows);
a = sparse(zerorows, ones(l, 1), ones(l, 1), n, 1);
%a = zeros(n,1);
k = 0;
residual = 1;
pi = pi0;
tic;
while (residual >= epsilon)
    prevpi = pi;
    k = k + 1;
    %if k == 1, disp(pi); disp(full(H)); disp(pi * H); end
    pi = alpha * pi * H + (alpha * (pi * a) + 1 - alpha)...
    * ((1 / n) * ones(1, n));
    residual = norm(pi - prevpi, 1);
end
pi=pi.';
iter = k;
time = toc;
end
```

PageRank for H, already normalised by row (occasionally the program above refuses to

normalise each row and runs forever):

```
function [pi, time, iter] = pagerank_stochastic(H, alpha, epsilon)
%INPUTS
%pi0 = starting PageRank vector
%H = sparse matrix
%n = size of H
%alpha = scaling parameter
%epsilon = convergence tolerance
%OUTPUTS
%pi = PageRank vector
%time = time taken to calculate PageRank
%iter = number of iterations taken to calculate PageRank
n = length(H);
pi0 = ones(1,n)/n;
rowsumvector = ones(1, n) * H';
for i=1:n
    if rowsumvector(i)~=0
        rowsumvector(i) = 1/rowsumvector(i);
    end
end
%H = bsxfun(@times,sparse(H),sparse(dividevector));
nonzerorows = find(rowsumvector);
zerorows = setdiff(1:n, nonzerorows);
l = length(zerorows);
```

```
a = sparse(zerorows, ones(l, 1), ones(l, 1), n, 1);
%a = zeros(n,1);
k = 0;
residual = 1;
pi = pi0;
tic;
while (residual >= epsilon)
    prevpi = pi;
    k = k + 1;
    %if k == 1, disp(pi); disp(full(H)); disp(pi * H); end
    pi = alpha * pi * H + (alpha * (pi * a) + 1 - alpha) *...
    ((1 / n) * ones(1, n));
    residual = norm(pi - prevpi, 1);
end
pi=pi.';
iter = k;
time = toc;
end
```

8.3 MATLAB code for Wikipedia article web crawler

```
function [time,U,L,hash,page] = surfer(root,n)
    %SURFER function creates the adjacency matrix of a portion of
    %the Web. Creates an adjacency matrix of dimensions n by n
```

```
%starting from the root link. The output U is a cell array of
%the URLs visited and L is a sparse matrix of links between
%pages, where  L(i,j) = 1 if url(i) links to url(j).
U = cell(n,1);
hash = zeros(n,1);
L = logical(sparse(n,n));
m = 1;
U{m} = root;
hash(m) = hashfun(root);
tic;
for j = 1:n
    %Try to open a page.
    try
        %disp(['open ' num2str(j) ' ' U{j}])
        page = webread(U{j});
    catch
        %disp(['fail ' num2str(j) ' ' U{j}])
        continue
    end
    %Follow the wikipedia links from the open page.

    for f = findstr('<a href="/wiki/',page)
        e = min(findstr('" title="',page(f:f+100)));
        if isempty(e), continue, end
```

```
url = strcat('https://en.wikipedia.org',...
deblank(page(f+9:f+e-2)));
url(url<' ') = '!'; % Nonprintable characters
if url(end) == '/',url(end) = []; end
%Look for links that should be skipped.

skips = {'.gif','.jpg','.pdf','.css','lmscadsi',...
'cybernet', 'Special', 'Wikipedia:', 'Portal:',...
'Help:','Talk:','Main_Page','File:', 'Template:',...
'Category:','talk:'};
skip = any(url=='!') | any(url=='?') | any(url=='#')...
| any(url=='"');
k = 0;
while ~skip && (k < length(skips))
    k = k+1;
    skip = ~isempty(findstr(url,skips{k}));
end
if skip
    if isempty(findstr(url,'.gif')) &&...
    isempty(findstr(url,'.jpg'))
        %disp(['      skip ' url])
    end
    continue
end
```

```
% Check if page is already in url list.

i = 0;
for k = find(hash(1:m) == hashfun(url))'
    if isequal(U{k},url)
        i = k;
        break
    end
end

%Add a new url to the graph there if there are fewer than n.

if (i == 0) && (m < n)
    m = m+1;
    U{m} = url;
    hash(m) = hashfun(url);
    i = m;
end

% Add a new link.

if i > 0
    %disp(['      link ' int2str(i) ' ' url])
```

```
                L(i,j) = 1;
            end
        end
    end
    save('linklist - Copy.mat', 'U');
    save('sparsematrix - Copy.mat', 'L');
    save('hash - Copy.mat','hash');
    page = U{j};
    fid = fopen('page - Copy.txt','wt');
    fprintf(fid, page);
    fclose(fid);
    time = toc;
end
%-----------------------
function h = hashfun(url)
    %Almost unique numeric hash code for pages already visited.
    h = length(url) + 1024 * sum(char(url));
end
```

Bibliography

[1] Stewart, I. (2013). *In Pursuit of the Unknown: 17 Equations That Changed the World*, ISBN-13: 9780465085989. Basic Books.

[2] Vaughan-Nichols, S. (2017). *Before Google: A history of search engines.* https://www.hpe.com/us/en/insights/articles/how-search-worked-before-google-1703.html

[3] Starry Internet (2019). *How big is the internet?* https://starry.com/blog/inside-the-internet/how-big-is-the-internet

[4] Langville, A., Meyer, C. (2006). *Google's PageRank and Beyond: The Science of Search Engine Rankings*, ISBN-13: 9780691122021. Princeton University Press.

[5] McDermott, K. B. & Roediger, H. L. (2023) *Memory (Encoding, Storage, Retrieval),* https://nobaproject.com/modules/memory-encoding-storage-retrieval

[6] Mijwil, Maad & Esen, Adam & Alsaadi, Aysar. (2019). *Overview of Neural Networks,* https://www.researchgate.net/publication/332655457_Overview_of_Neural_Networks

[7] Khan Academy. Accessed 14th May, 2023. *Prehistory before written records,* https://www.khanacademy.org/humanities/world-history/world-history-beginnings/origin-humans-early-societies/a/learning-about-prehistory-article

[8] Library of Congress. Accessed 14th May, 2023. *Fascinating Facts,* https://www.loc.gov/about/fascinating-facts/

[9] H. Cairns (2014). *A short proof of Perron's theorem,* https://pi.math.cornell.edu/~web6720/Perron-Frobenius_Hannah%20Cairns.pdf

[10] The Hebrew University of Jerusalem. Accessed 14th May, 2023. *The Power Method,* https://www.cs.huji.ac.il/w~csip/tirgul2.pdf

[11] S. Brin and L. Page (1998). *The Anatomy of a Large-Scale Hypertextual Web Search Engine,* http://infolab.stanford.edu/~backrub/google.html

[12] Guinness World Records (2020). *Largest Encyclopedia Online,* https://www.guinnessworldrecords.com/world-records/85651-largest-encyclopedia-online

[13] Google (2023). *Introduction to robots.txt,* https://developers.google.com/search/docs/crawling-indexing/robots/intro

[14] C. Stobing (2018). *Using deep web search engines for academic and scholarly research,* https://www.comparitech.com/blog/vpn-privacy/using-deep-web-search-engines-for-academic-research/

[15] Wikipedia. Accessed 14th May, 2023. *Size of Wikipedia,* https://en.wikipedia.org/wiki/Wikipedia:Size_of_Wikipedia

[16] Cosmograph (2023). *Cosmogrpah: visualise big networks within seconds,* https://cosmograph.app/

[17] Statcounter (2023). *Search Engine Market Share Worldwide,* https://gs.statcounter.com/search-engine-market-share

[18] Wikipedia. Accessed 14th May, 2023. *Six degrees of Wikipedia,* https://en.wikipedia.org/wiki/Wikipedia:Six_degrees_of_Wikipedia#Seven-degree_chains

[19] D. Gleich & P. Constantine (2007), *Using Polynomial Chaos to Compute the Influence of Multiple Random Surfers in the PageRank Model,* https://www.cs.purdue.edu/homes/dgleich/publications/constantine2007%20-%20pagerank%20pce.pdf

[20] D. Gleich (6th February, 2007), *wikipedia-20070206,* https://sparse.tamu.edu/