# Google's PageRank

COE 322 Write-Up

Jackson Thetford (JET3522)

Mihiro Suzuki (MS84579)

Luca Labardini (LL34958)

December 2022

## 1 Abstract

Google's PageRank algorithm is an extensive algorithm that is a natural application of pointers and elegantly uses graph algorithms and linear algebra to usefully create order in the disorder of webpages on the internet. The PageRank algorithm was a foundational algorithm developed by Google. The algorithm takes into account the number of incoming links a web page has and assigns that page a certain weight, based on a probability distribution. Pages with a higher rank are considered more important, with the assumption that pages linked by many other pages are of greater importance than those with less incoming links. This ranking algorithm organizes the pages in the internet to make user interaction more efficient. There are several problems that arise with this method including SEO, the connectivity of the internet, and the varying quality of links. In this study, the methods and applications of Google's PageRank algorithm are recreated, simulated, and studied. Google's PageRank algorithm is found to be extremely powerful and useful in ordering the internet.

## 2 Introduction

Google Page rank is an algorithm developed by Google, named after the co-founder Larry Page, that is used to rank web pages based on their relevance and importance. The importance of a page is a numerical value assigned to that page, based on the number of incoming links and the quality of those links. The basic idea of this algorithm is that the importance of a page, one with a higher rank, will be linked to many other pages. For this C++ project, the internet was simulated by creating a set amount of pages, with an average number of links for each page, then creating a random web of links connecting all the pages. Taking this artificial internet, the PageRank algorithm can calculate the importance of each page from a probability distribution that is calculated by the algorithm and represents the chance of ending up at a page by randomly clicking a link from one page to another many times. A page with a higher probability to be visited by randomly clicking these links will have a higher value in the probability distribution and therefore will have a higher rank. There are several issues that can arise, however, such as if not all pages are connected to each other and the artificial boosting of a page's rank. These will be discussed further in the subsequent sections.

## 3 Code

The "Page" class is used to create objects called "pages" and assign attributes. In our case, a page represents a single website on the internet, held as an object from our page class. This is the foundational and simplest object in our algorithm and are the building blocks in our attempt to simulate the internet. In the case that a page has no links, an attempt to grab a link from that page will result in the computer output "there is none." A page is constructed by giving it a name and an identification number. Each page has attributes stored for how many links it has, the name of the page, a vector of links to other pages, and an ID. See Appendix 1.

Several basic methods exist in the page class, "global_ID", "link_amount", and "as_string", return the global ID, the number of links, and the name of the page (as a string), respectively. A method to create a link from one page to another "add_link" was also created using a pointer to the page of interest as input, and adding this pointer as a "link" to the object the method is called on. This can be viewed as creating a connection from one page to another. See Appendix 2.

In the page class, the function "click" allows the movement from one page to another given that the page links to the page of interest via a shared pointer. This "click" method takes an integer as input that is used to index another page from the page's links which are stored as a vector "link." Another method, "random_click", utilizes the "click" method, however, clicks on one of the page's links at random. This is done by getting a random integer from 0 to the length of the page's "links" vector, then calling the click method, using that random integer as input. See Appendix 3.

The "Web" class is used to create an object that stores many pages. An object from the Web class is representative of the internet, in that it holds all of the interconnected Page class objects. An object from the Web class, in our case named "Internet", is constructed by declaring how many pages are desired to be held in that object, and then automatically creates a vector of that number of pages, naming the pages as "Page " and then the ID of that page (Page 1, Page 2, Page 3, ... etc. ). In order to add pages to the web, a web constructor with an input of artSize was created. "artificialSize" was created to preallocate a larger matrix than the current web matrix to allow it to scale. See appendix 4.

From an object derived from the web class, methods have been created to return all the pages ("all_pages") in that web object, find how many pages exist ("number_of_pages"), get a specific page in the web object ("get_page"), print out the name of a page in the web object ("print"), and find a random page in the object ("random_page"). These methods are similar to the basic functions from the page class mentioned prior. Furthermore, a method to create random links between all the pages ("create_random_links") has been created to simulate many different versions of the internet every time the algorithm is run. This method

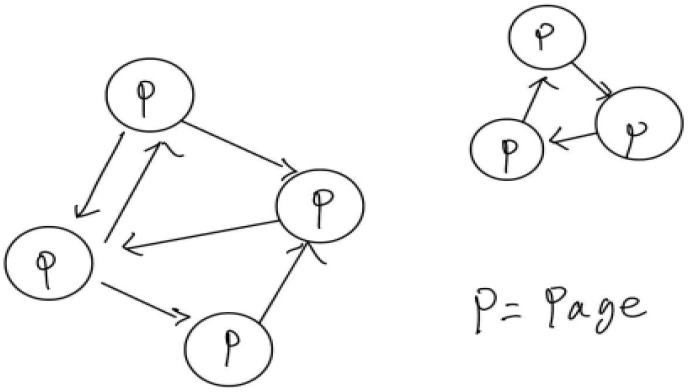
only takes the average number of links between all the pages in the web object that is desired as input. This is done by selecting two pages inside the Web object at random, X and Y, and then using the previously mentioned "add_link" method from the page class that is used to create a link from one page to another. See appendix 5.

The method "random_walk" utilizes the "random_click" method previously explained. Given a starting page and a number of "steps" or clicks, to perform, will go from one page to another page that it is linked to at random, repeatedly for the number of steps given, ending at a random page. See Appendix 6.

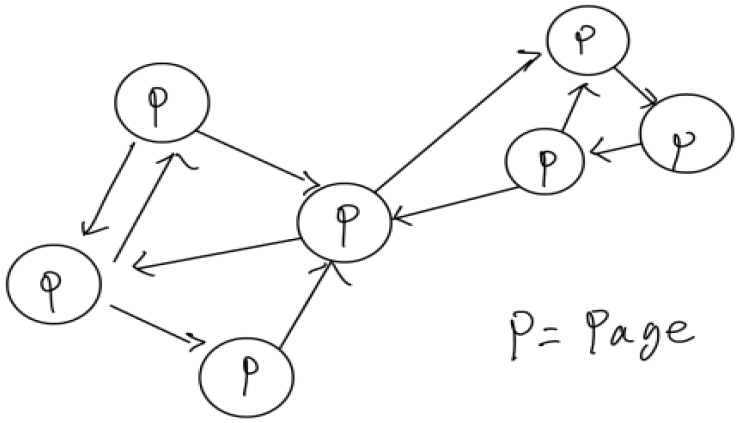
The "full_connected" method is used to determine if a web is fully connected using matrix multiplication. By raising the matrix to its own power, and if all values are positive, it can be assumed that the web is fully connected. A deeper explanation of this matrix is in the matrix class section. Lastly, the method "diameter" was created to find the diameter of a given web object. The diameter in this case is defined as the maximal shortest path between any two pages. In our case, the diameter is defined by the most clicks it would take to get from any given page to another page in the internet, given the most efficient path is taken for every page-to-page walk combination. This method essentially checks every page with every page and finds the lowest number of clicks it takes to get from one page to another. As the "random_click" method is used to define the path that is taken from one page to another, each walk from page to page is simulated ten times the number of pages in the web. This gives us assurance for all the simulations, at least one of them represents the quickest path from one page to another, or the path with the lowest amount of clicks required, which is the number recorded and assigned as the number of steps for that combination of pages. This process is repeated until every possible path from every combination of pages is taken. The lowest number of steps required to get between every combination of pages is recorded for each combination, then the greatest value is then defined as the diameter. See Appendix for 7.

The "ProbabillityDistribution" class is used for determining the probability that someone clicks a certain page. This metric is used to determine the ultimate page rank in google, with higher probability pages at the top. The "set_random" method (shown below) is used to allocate random link numbers to the pages and combined with the "normalize" method to transform them into a probability instead (shown below). See Appendix 8. This probability distribution is then used multiple times in the web class under the "globalclick" method where it converges to the ultimate probability distribution. The "as_string" method takes these probabilities and prints them out.

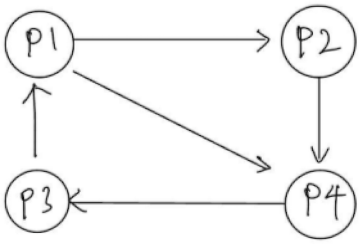
The "Matrix" class is used to manipulate matrix calculations. In order to find the shortest path algorithm we first needed to verify that our Web (internet) was fully connected. A case in which there are multiple connected components is displayed in the figure below. The nodes represent a page and the arrows represent a link that points from one page to another.



On the other hand, a case in which there is a singular web in which one can go from one node to any other is defined as a "fully connected" graph. Below is a figure representing an example of a "fully connected" graph.




To verify the condition we used matrix multiplication. The figure below shows how it was implemented.




| | | To 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| From | 1 | 1 | 1 | 0 | 1 |
| | 2 | 0 | 1 | 0 | 1 |
| | 3 | 1 | 0 | 1 | 0 |
| | 4 | 0 | 0 | 1 | 1 |

The columns represent the starting page and the rows represent the ending page. Links are represented by a 1 and 0 that are not connected. For example, page 1 links to page 2 but page 2 does not link to page 1. Therefore, (1,2) is a 1 but (2,1) is a 0. Additionally, it was assumed that links point to themselves. So the matrix was initialized by an identity matrix represented in red. These matrices are later used to verify a fully connected web. The "Matrix Multiply" is used to multiply these matrices.

# 4 Results and Discussion

While the random click method is useful for determining the probability of each page is clicked, it is not the optimal method to determine the probability distribution. By utilizing matrix multiplication and Markov Chain principles, it can be more accurately and efficiently approximated. In the random walk method, a page is randomly selected to start at, and clicks a random number of pages. Then the end pages are recorded in "landing counts" as a vector and based on the number of clicks, the probability distribution is determined. The method requires a high number of iterations. On the other hand, the "globalclick" method represents the connection between a matrix. First, a vector Pi is preallocated with one 1 and with the rest being 0s to simulate a random click. Then the preallocated Pi vector "v" is multiplied by the "distributions_matrix".The equation is represented below.

$$\pi_o A = \pi$$

Here is an example of an iteration.

$$A_0 = \begin{bmatrix} 1/3 & 1/3 & 0 & 1/3 \\ 0 & 0.5 & 0 & 0.5 \\ 0.5 & 0 & 0.5 & 0 \\ 0 & 0 & 0.5 & 0.5 \end{bmatrix} \qquad \pi_0 = \begin{bmatrix} 0 & 0 & 1 & 0 \end{bmatrix}$$

$$\text{Therefore by} \quad \pi_o A = \pi$$

$$\pi = \begin{bmatrix} 0.5 & 0 & 0.5 & 0 \end{bmatrix}$$

This equation is then repeated until it converges to a given tolerance as shown in the while loop in the main function. From running the simulations, it is apparent that the global click method takes significantly fewer iterations than the random walk method. Therefore, it is a much more computationally efficient method to determine the probability of a click. This efficiency is extremely important for scalability and when constructing something as large as the internet.

A problem that arose through the simulation of this algorithm is the artificial inflation of a page's rank. This method is called search engine optimization (SEO) and refers to the increasing of a page's rank through using keywords or creating null websites linking to the page of interest. In our case, we consider the latter. The rank of a page can be manipulated by creating many pages that have a link to the page of interest. This increases a page's chance to be randomly visited therefore giving it a greater probability in the probability distribution, therefore increasing the page's rank. This can be done fairly easily, however, is equally simple to stop. The way google solves this problem is by considering the importance of certain links. This is done by recognizing both incoming links and outgoing links, as well as taking a website's rank into account to evaluate the quality of its outgoing links. Fundamentally, this means that a low-ranking website's links to other websites will be less important when compared to a high-ranking website's links. For the SEO problem, these null websites have extremely low ranks and their links to the page attempting to be inflated are consequently useless as they have low quality.

Another metric found in this simulation is the diameter. The diameter is a testable parameter that can be used to describe our internet. As previously mentioned, the diameter of the internet is the longest path or the most clicks, one would need to take to get from any page to any other page on the internet, given the most efficient route is taken. A back-of-the-envelope estimation of the effect of the diameter will conclude that the diameter plays a significant role in the spread of the probability distribution, as well as the number of iterations needed to fairly create that distribution. The diameter is extremely important when considering the computational power needed to run this algorithm, as a larger diameter will require more clicks to get from the farthest two pages. Considering the massive internet today, the graph made up by the internet's web pages is highly connected and has a diameter of about 19 clicks. This low diameter means it is fairly easy and efficient to rank pages and get from one page to another.

# 5 Summary

In conclusion, the PageRank algorithm is a very strong algorithm that applies matrix manipulation and statistics to essentially order the internet. We found that this method worked extremely well for small-scale internet simulations, however with the actual scale of the internet being over 1 billion webpages, immense computational power would be required and a different algorithm may be more desirable. The PageRank algorithm handles the extensive graph of web pages beautifully, however, manipulation of this algorithm proves to be a great risk. One of these possible manipulations is SEO, where many null pages are created and linked to a specific page to inflate that page's rank. This was discussed earlier and is a fairly easy problem to solve as the "quality" of links can be weighted in the calculation of a page's rank. This would essentially declare these SEO-created pages of low quality and their links would have little to no effect on page ranking. Another issue with this algorithm is if the internet is not fully connected. This was also previously discussed, and in the naive page rank method, the random walk method would only access pages linked to the starting page. This means any page outside of this linkage would never be clicked on and would have an effective rank of 0 from the probability distribution. This is also heavily affected by the number of average links, whereas the internet having more links will lead to more connectivity between all the pages. Related to this idea, the diameter of the internet, the longest quickest path from any two pages, is also a useful description of the connectivity of a fully connected internet. The diameter is decreased with an increase in net size and is decreased with an increase in the average number of links each webpage has. The diameter of the actual internet is around 19 clicks. Overall, Google's PageRank algorithm is an extensive algorithm that is a natural application of pointers and elegantly uses graph algorithms and linear algebra to usefully create order in the disorder of webpages on the internet.

# 6 Bibliography


[1] Brown, David Scott. "The Role of Diameter Signaling in Today's Networks." Techopedia.com, https://www.techopedia.com/2/31603/networks/wireless/the-role-of-diameter-signaling-in-todays-networks.

[2] Eijkhout, Victor. Introduction to Scientific Programming in C++17/Fortran2008. Vol. 3, CC-BY 4.0 License, 2022.

[3] Whang, Joyce Jiyoung, et al. "Scalable Data-Driven Pagerank: Algorithms, System Issues, and Lessons Learned." Lecture Notes in Computer Science, 2015, pp. 438–450., https://doi.org/10.1007/978-3-662-48096-0_34.

# 7 Appendix: Reference Code

[1] Page class/Constructor Code

```
class Page{
    private:
        int linkamount;
        string name;
        vector<shared_ptr<Page>> links;
        int globalID;
    public:
            Page(string n, int id){
            linkamount = 0;
            links.resize(1);
            name = n;
            globalID = id;
        }
```

[2] add_link method from Page class

```
        void add_link(shared_ptr<Page> p){
            links[linkamount] = p;
            linkamount++;
            links.resize(linkamount+1);
        }
```

[3] click and random_click methods from page class

```
        shared_ptr<Page> click(int i){
            return links[i]; // "links" is a vector of all a page's links
        }

        shared_ptr<Page> random_click(){
            if(linkamount == 0)
            {
                throw("ERROR: no links. Need to program this CASE. \n");
            }
            int random = rand() % linkamount;
            //random integer in range of number of links
            return click(random);
        }
```

[4] Web class/constructor Code

```
class Web{
    private:

        int netsize;
        vector<shared_ptr<Page>> pages;
        vector<ProbabilityDistribution> distributions;
        Matrix matrix_form;
        Matrix distributions_matrix;
        int artificialSize;

    public:
        Web(int size, int artSize){
            artificialSize = artSize;
            netsize = size;
```

```
        pages.resize(artificialSize);
        for(int i = 0; i < artificialSize; i++){
            auto str = std::to_string(i);
            auto name = "Page " + str;

            pages[i] = make_shared<Page>(name, i, artificialSize);
        }
    }
```

[5] create_random_links method from Web class

```
    void create_random_links(int avgLinks){
        int n = avgLinks * netsize;
        int randomX, randomY;
        for(int i = 0; i < n; i++){
            randomX = rand() % netsize;
            randomY = rand() % netsize;
            while (randomX == randomY){randomY = rand() % netsize;}

            //Set link from page1 to page2 (page1.addlink(page2))
            auto pageX = pages[randomX]; auto pageY = pages[randomY];
            pageX->add_link(pageY);
        }
    }
```

[6] random_walk method from Web class

```
     auto random_walk(shared_ptr<Page> startingPage, int length){
        shared_ptr<Page> currentPage = startingPage;
        for(int i = 0; i < length; i++){
            if(currentPage->link_amount() == 0){return currentPage;}
            currentPage = currentPage->random_click();
        }
        return currentPage;
    }
```

[7] Diameter Method:

```
int diameter(){
    auto currentPage = getPage(1);
    auto startPage = getPage(1);
    auto endPage = getPage(2);
    int numSteps = 0;
    int totalSteps;
    int vecPosition = 0;
    vector<int> shortestPath((netsize*netsize) - netsize);
    int longestPath = 0;

    for( int i = 0; i < netsize; i++){
            startPage = getPage(i);

            for (int j = 0; j < netsize; j++){
                totalSteps = 2*netsize;
                numSteps = 0;

                if ( j == i) { j++;}
                if (j>=netsize) {break;}
```

```
                endPage = getPage(j);

                for (int w = 0; w < netsize*10; w++){
                    while (currentPage != endPage){
                    numSteps++;
                    currentPage = random_walk(currentPage, 1);
                    if (numSteps >= netsize){break;}
                }
                if (numSteps < totalSteps){
                    totalSteps = numSteps;
                }
                numSteps = 0;
                currentPage = startPage;
                }
            shortestPath[vecPosition] = totalSteps;
            vecPosition++;
        }
    }
    for (int i = 0; i < shortestPath.size(); i++){
        if (shortestPath[i] > longestPath){
            longestPath = shortestPath[i];
        }
    }
    return longestPath;
    }
```

[8] set_random and normalize methods from ProbabilityDistribution class

```
    void set_random(){
            float random;
            for(int i = 0; i < numbers.size(); i++){
                random = rand() % 10;
                numbers[i] = random;
            }
    }
    void normalize(){
            float sum = 0;
            for(int i = 0; i < numbers.size(); i++){
                sum+= numbers.at(i);
            }
            if(sum!= 0){
                for(int i = 0; i < numbers.size(); i++){
                    numbers[i] /= sum;
                }
            }
        }
```