PageRank

Caroline, Victoria and Magnus

January 2026

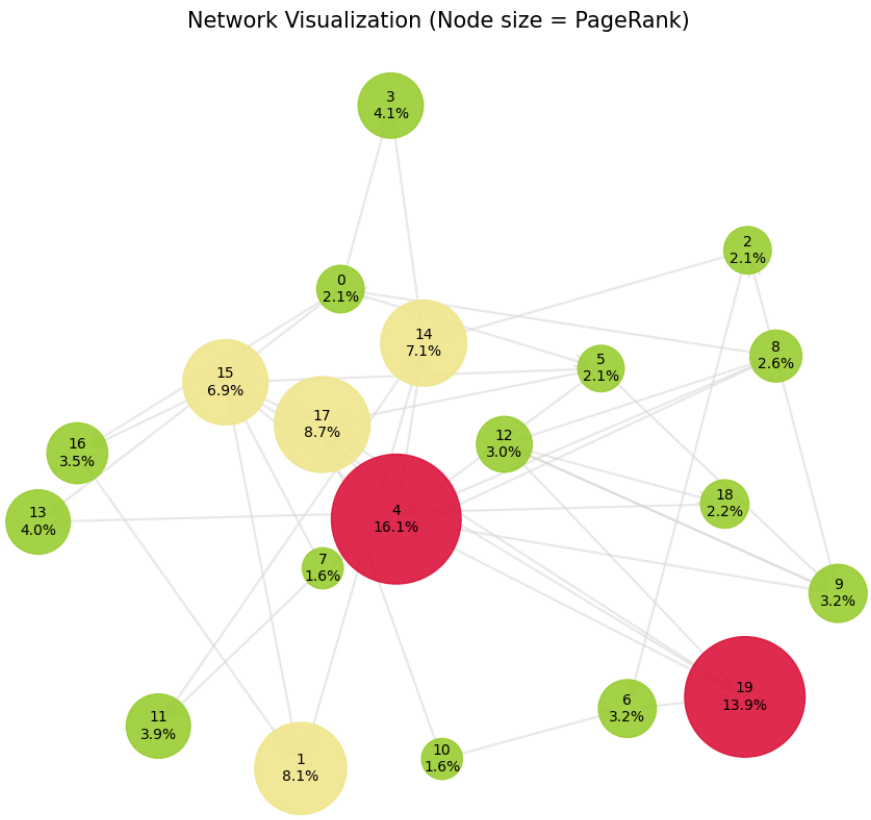

Figure 1: Plot of matrix_pagerank model

1 Introduction

In this project we explore the PageRank algorithm. This algorithm considers a finite amount of webpages connected by directed links. It ranks the pages based on the probability of ending at a certain page k at a given time t, if you start at a randomly selected page, and move through the links after the rule, that every time you end up at a page you move to the next page by randomly selecting a link on that page. The algorithm also has a damping factor, which allows the simulation to jump to a random page, even if it lands on a page with no outgoing links.

There are different approaches to the algorithm.

The first we looked at is the Random surfer model. In this model we start by choosing a random page

to start at, then after that choose a link between the ones on the page. Leading to the next page, and keeps repeating that t times. If you end up at a page, that don't link to any other page, you choose a random page among all the pages. To get the probability you count how many times you visit the page N_p over the number of pages visited n. They should all add up to 1.

Another approach is the Recursive model. This model looks at the probability of being at a page on a certain time by first setting all the probabilities to 1/N, representing starting on a page in the Random Surf. Thereafter the formula, $PR(p) = (1-d) * \frac{1}{N} + d * \sum_{q \in Inbound(p)} \frac{PR(q)}{Outbound}$, is applied to all the pages to calculate the probability of reaching that page on the next step. This step is done a sufficient number of times to get the probability distribution, which the function will output as the PageRank.

The last models are matrix formulations. There are two different approaches in calculating the PageRank from the matrix, but both of them depend on formulating a modified adjacency matrix, which describes how the pages are linked, where if a page isn't linked to any other, it is set as linking to all pages.

The first matrix approach finds an eigenvector corresponding to the eigenvalue 1 and multiply it to ensure, that the entries are positive and sum to 1. The other matrix approach calculates the PageRank by raising the modified adjacency matrix to a high enough power, and choosing the first column, which will correspond to the eigenvector described above.

All the different approaches also include a damping factor, which implements a way to get out of parts of the web, even if it is not connected to the rest of the web.

The code implementation is in the Appendix.

2 Theoretical problems

Problem 1

Assuming that $\mathbf{x}$ is a probability vector, i.e. its elements sum to 1, show that Equation (2.3): $\mathbf{x} = \frac{(1-d)}{N} \cdot \mathbf{e} + dL\mathbf{x}$ can be written as $\mathbf{x} = M_d\mathbf{x}$ (Equation 2.4)

Proof

From Equation 2.3 the components of $\mathbf{x}$ will be:

$$\mathbf{x}_i = \frac{(1-d)}{N} \cdot 1 + d \cdot \sum_{k=1}^{N} (L_{ik}\mathbf{x}_k)$$

According to Equation 2.4 the components of $\mathbf{x}$ will be:

$$\mathbf{x}_i = \sum_{k=1}^{N} \left(\frac{(1-d)}{N} + d \cdot L_{ik} \right) \mathbf{x}_k = \sum_{k=1}^{N} \left(\frac{(1-d)}{N} \cdot \mathbf{x}_k + d \cdot L_{ik} \cdot \mathbf{x}_k \right)$$

$$= \sum_{k=1}^{N} \left(\frac{(1-d)}{N} \cdot \mathbf{x}_k \right) + \sum_{k=1}^{N} (d \cdot L_{ik} \cdot \mathbf{x}_k) = \frac{(1-d)}{N} \cdot \sum_{k=1}^{N} (\mathbf{x}_k) + d \cdot \sum_{k=1}^{N} (L_{ik} \cdot \mathbf{x}_k)$$

Since $\mathbf{x}$ is a probability vector: $\sum_{k=1}^{N} x_k = 1$:

$$\frac{(1-d)}{N} \cdot \sum_{k=1}^{N} (\mathbf{x}_k) + d \cdot \sum_{k=1}^{N} (L_{ik} \cdot \mathbf{x}_k) = \frac{(1-d)}{N} \cdot 1 + d \cdot \sum_{k=1}^{N} (L_{ik} \cdot \mathbf{x}_k)$$

And thus $\mathbf{x}_i$ is the same for both Equation 2.3 and 2.4, which shows that Equation 2.3 can be written as Equation 2.4. $\square$

Problem 2

Lemma 3.1: If A is a Markov matrix then A^m is also a Markov matrix for $m = 0, 1, 2, \cdots$

Proof

A and B are both Markov matrices, therefore we know that the sum of each of their columns is 1:

$$\sum_i A_{ij} = 1, \qquad \sum_i B_{ij} = 1$$

We can now write the sum of each column in the product AB by using the definition of a matrix product:

$$\begin{aligned}
\sum_i (AB)_{ij} &= \sum_i (\sum_k A_{ik} B_{kj}) \\
&= \sum_i A_{i1} B_{1j} + \sum_i A_{i2} B_{2j} + \cdots + \sum_i A_{in} B_{nj} \\
&= B_{1j} \sum_i A_{i1} + B_{2j} \sum_i A_{i2} + \cdots + B_{nj} \sum_i A_{in} \\
&= B_{1j} + B_{2j} + \cdots + B_{nj} \\
&= \sum_i B_{ij} = 1 = 1
\end{aligned}$$

The product of two Markov matrices is therefore also a Markov matrix. A^2 is a product of two Markov matrices A and A, and is therefore also a Markov matrix. A^m for $m = 0, 1, 2, \cdots$, can be seen as a product of the Markov matrices A and A^{m-1}, and is therefore also a Markov matrix. □

Problem 3

Lemma 3.2: If $A \in R^{nxn}$ is a Markov matrix then the spectral radius $rad_A = 1$

Proof

We split the proof into three main steps: showing A has $\lambda = 1$, proving norm inequality, and finally proving the spectral radius is $rad_A = 1$.

Step 1: A has $\lambda = 1$
We start by proving that A, has an eigenvalue equal to 1. To do that, we find the eigenvalues for the transposed of A. We know that the transposed of A_k has an eigenvalue, that is equal to 1 as shown in Equation (3.2). This means that A also has an eigenvalue that is equal to 1, since:

$$det(A - \lambda \cdot I_n) = det(A^T - \lambda \cdot I_n)$$

As you can expand a determinant by a row or a column.

Step 2: Norm inequality
Now we show the inequality $\|Xv\|_\infty \leq \|X\|_\infty \cdot \|v\|_\infty$· Let the rows in X be denoted by w_i for $i = 1, 2, ..., m$, then we have

$$\begin{aligned}
\|Xv\|_\infty &= \max_i (|w_i \cdot v|) \\
&\leq \max_i (|(w_i)_1| \cdot |v_1| + |(w_i)_2| \cdot |v_2| + \cdots + |(w_i)_n| \cdot |v_n|) \\
&\leq \max_i (\sum_j |(w_i)_j| \cdot \|v\|_\infty) \\
&= \|v\|_\infty \cdot \max_i \sum_j |(w_i)_j| = \|X\|_\infty \cdot \|v\|_\infty
\end{aligned}$$

Which shows the inequality holds.

Step 3: Spectral radius is $rad_A = 1$
Now we want to prove the spectral radius is $rad_A = 1$. We will prove by contradiction. Suppose that λ is an eigenvalue with $|\lambda| > 1$. Let v be a corresponding eigenvector and consider $A^k v$ as $k \to \infty$. We have that $A^k v = \lambda^k v$, then take the norm

$$\|A^k v\|_\infty = \|\lambda^k v\|_\infty = |\lambda^k| \cdot \|v\|_\infty$$

By Step 2, we have $\|\lambda^k v\|_\infty \leq \|A^k\|_\infty \cdot \|v\|_\infty$. Furthermore, we know that $\|A^k\|_\infty \leq n$, so

$$\begin{aligned} \|\lambda^k v\|_\infty \leq \|A^k\|_\infty \cdot \|v\|_\infty &\leq n \cdot \|v\|_\infty \\ \Rightarrow |\lambda^k| \cdot \|v\|_\infty &\leq n \cdot \|v\|_\infty \\ \Rightarrow |\lambda^k| &\leq n \end{aligned}$$

Since, for any k, our eigenvalue λ has to be $-1 \leq \lambda \leq 1$, because any value greater would grow to infinity as $k \to \infty$, which will eventually be greater than n, since our n is finite. We have reached a contradiction and thus the maximum absolute value among the eigenvalues of A is $rad_A = 1$. $\square$

Problem 4

If A is Markov matrix and $0 < \tau < 1$ prove that A_τ is also a Markov matrix and has strictly positive entries, $(A_\tau)_{ij} > 0$.

Proof

First we show that A_τ is also a Markov matrix. Let $0 < \tau < 1$. For $A \in \mathbb{R}^{n \times n}$, set:

$$A_\tau = \tau \cdot A + \frac{1-\tau}{n} \cdot E_n$$

where E_n is a $n \times n$ matrix consisting entirely of 1's.

If A is a Markov matrix, then each of the columns in A sum up to 1:

$$\sum_i A_{ij} = 1$$

To show that A_τ is also a Markov matrix, we need to show that each of the columns in A_τ sum up to 1:

$$\begin{aligned} \sum_i (A_\tau)_{ij} &= \sum_i (\tau \cdot A_{ij} + \frac{1-\tau}{n} \cdot (E_n)_{ij}) \\ &= \sum_i (\tau \cdot A_{ij}) + \sum_i (\frac{1-\tau}{n} \cdot (E_n)_{ij}) \\ = \tau \cdot \sum_i (A_{ij}) + \sum_i (\frac{1-\tau}{n} \cdot 1) &\text{ (since all components in } E_n \text{ is 1)} \\ &= \tau \cdot 1 + \frac{1-\tau}{n} \cdot n \end{aligned}$$

(since the columns in A each add up to 1, and the second term is added n times as E_n is an n $\times$ n matrix)

$$= \tau + 1 - \tau = 1$$

Thus A_τ is a Markov matrix.

Now we need to prove that A_τ has strictly positive entries, $(A_\tau)_{ij} > 0$. We will prove by contradiction. Since we know A_τ is a Markov matrix, let there be an element $(A_\tau)_{ij} = 0$.

For $(A_\tau)_{ij} = 0$, it must be the case that

$$(A_\tau)_{ij} = \tau A_{ij} + \left(\frac{1-\tau}{n} \cdot E_n\right)_{ij} = 0$$

We can split it up into two equations:

$$\tau A_{ij} = 0, \qquad \left(\frac{1-\tau}{n} \cdot E_n\right)_{ij} = 0$$

Since every element in E_n consists of 1s, to get a 0-element, we must have $\tau = 1$ such that $(1-1)/n = 0$. This is a contradiction, since $0 < \tau < 1$. Therefore A_τ is a Markov matrix, and $(A_\tau)_{ij} > 0$ for all i, j. $\square$

Problem 5

For a Markov matrix A with strictly positive entries, we need to prove that the transposed matrix A^T has exactly one eigenvalue λ with $|\lambda| = 1$, namely $\lambda = 1$, and that the corresponding eigenspace is $\mathscr{E}_1 = span\{\mathbf{e}\}$, where $\mathbf{e} = [1, 1, ..., 1]^T$. Furthermore that any other eigenvalue has modulus less than 1.

Proof

From Problem 3 we know that A and A^T have the same eigenvalues and that $rad_A = 1$. Therefore $rad_{A^T} = 1$, which means that for λ an eigenvalue for A^T then $|\lambda| \leq 1$. We have also shown that A and A^T does in fact have an eigenvalue $\lambda = 1$ as shown in Equation (3.2)

Suppose λ is an eigenvalue of A^T with $|\lambda| = 1$ and $\mathbf{v}$ is a corresponding eigenvector. Let k denote an index s.t. $\|\mathbf{v}\|_\infty = |v_k|$ and noting $(A^T)_{kj} = (A)_{jk} = a_{jk}$. Now we can write:

$$\|v\|_\infty = |v_k| = |\lambda| \cdot |v_k| = |\lambda \cdot v_k| = |\sum_{j=1}^{n} a_{jk} \cdot v_j| \tag{3.3}$$

(Since v is an eigenvector for A^T: $A^T v = \lambda v \to |A^T v| = |\lambda v|$. Then by only looking at the k´th row in A^T we get: $|\sum_{j=1}^{n} (A^T)_{kj} \cdot v_j| = |\lambda v_k| \to |\sum_{j=1}^{n} a_{jk} \cdot v_j| = |\lambda v_k|$.)

$$\leq \sum_{j=1}^{n} |a_{jk} \cdot v_j| \text{ (By the triangle inequality)} \tag{3.4}$$

$$= \sum_{j=1}^{n} a_{jk} \cdot |v_j| \text{ (All components in A has positive real values)}$$

$$\leq \sum_{j=1}^{n} a_{jk} \cdot \|v\|_\infty \text{ (Since } |v_j| \leq \|v\|_\infty \text{ for all } j = 1, 2, ...) \tag{3.5}$$

$$= \|v\|_\infty$$

$\|v\|_\infty$ does not depend on j now, and can be put outside the summation: $= \|v\|_\infty \cdot \sum_{j=1}^{n} a_{jk}$. This summation is a column in the Markov matrix A and add to 1.

The beginning and end are the same ($\|v\|_\infty$), therefore the inequalities will have to be equalities.

By using the first inequality: $|\sum_{j=1}^{n} a_{jk} \cdot v_j| = \sum_{j=1}^{n} |a_{jk} \cdot v_j|$ and part 2 of the triangle inequality, we now know that $a_{jk}v_j$ for $j = 1, 2, ...$ are non-negative real multiples of each other. By also using the second inequality: $\sum_{j=1}^{n} a_{jk} \cdot |v_j| = \sum_{j=1}^{n} a_{jk} \cdot \|v\|_\infty$ and A has strictly positive entries, we can conclude that $|v_j| = \|v\|_\infty = |v_k|$ for all $j = 1, 2, ...$ and we must have $v = Ce$, where $|C| = \|v\|_\infty = |v_k|$.

Thus the corresponding eigenvector must be a multiple of $\mathbf{e}$, and there is only one eigenvalue for which $|\lambda| = 1$ as the corresponding eigenvectors for distinct eigenvalues of the same matrix will be linearly independent.

The corresponding eigenspace for the eigenvalue $\lambda = 1$ is thus $\mathscr{E}_1 = span\{\mathbf{e}\}$, where $\mathbf{e} = [1, 1, ..., 1]^T$. □

Problem 6

Theorem 3.4 Let $n \geq 2$ and $A \in \mathbb{R}^{n \times n}$ be a Markov matrix with strictly positive entries. Then there is a unique vector $x \in \mathbb{R}^n$ such that $Ax = x$, all entries of x are strictly positive and sum to 1, and:

$$\lim_{k \to \infty} (A)^k = xe^t = [x, x, ..., x]$$

If A is Markov matrix with strictly positive entries, the unique probability vector x satisfying $Ax = x$ is called the stationary distribution for A.

Explain the relevance of Theorem 3.4 to the PageRank algorithm.

In programming task 3, we managed to compute the PageRank by taking the power of the matrix M_d (which we showed to be equivalent to the recursive model in Problem 1), and then taking the first column of the matrix, to get the PageRank vector x.

Theorem 3.4 is relevant for computing the PageRank algorithm, because it shows that we can find the ranking of a site by taking the power $(M_d)^k$ as $k \to \infty$. It will cause it to converge to a matrix with the PageRank vector x in its columns. It also shows that there is a unique ranking for the web. This method causes our simulation to converge faster than computing all the eigenvectors of the matrix and simulating it with the random surfer model. This is crucial, since, for large networks of pages, we want to be able to compute the PageRank fast and efficient.

We can test the runtime by implementing a time function in our three different scenarios. We will generate a random web by the function make_web, and count the seconds that elapses for the functions: `random_surf`, `recursive_pagerank`, `eigenvector_pagerank`, and `matrix_pagerank` (Thm. 3.4.):

```
>>> web = make_web(100, 20)
>>> ranking1 = random_surf(web, 100000)
>>> ranking2, iterations = recursive_pagerank(web, 0.0000001)
>>> ranking3 = eigenvector_pagerank(web)
>>> ranking4 = matrix_pagerank(web, 20)

Total runtime of random_surf is 3.743 seconds
Total runtime of recursive_pagerank is 0.0216 seconds
Total runtime of eigenvector_pagerank is 0.009 seconds
Total runtime of matrix_pagerank is 0.001 seconds
```

This is merely an example of 100 pages where each page links to between 0-20 (If you want to see comparisons between our implementation on larger networks, see Comparing our implementations) This shows the relevance of the theorem in the algorithm.

Problem 7

Prove Theorem 3.4 for the case that A has n distinct eigenvalues.

Proof

Since A has n distinct eigenvalues, we know that it is diagonalizable and we can write $A = VDV^{-1}$ where D is a diagonal matrix of our unique eigenvalues, and V is a matrix of the corresponding eigenvectors. We can write $A^k = VD^kV^{-1}$. Since

$$D^k = \begin{bmatrix} \lambda_1^k & & 0 \\ & \lambda_2^k & \\ & & \ddots \\ 0 & & \lambda_n^k \end{bmatrix}$$

and $|\lambda| \leq 1$, then D^k must converge to a matrix consisting of 1's and 0's. Therefore A^k must also converge and $\lim_{k\to\infty}(A)^k = W$ exists.

To explain why every column of W must be an eigenvector of A with an eigenvalue 1, we denote the columns of W by $w_1, w_2, ..., w_n$, and consider the product $AW = A \cdot [w_1, w_2, ..., w_n] = [w_1, w_2, ..., w_n]$.
By looking at an arbitrary column in W, w_i, the matrix vector product will be: $Aw_i = w_i$.
This shows that the i'th column in W is an eigenvector of A with eigenvalue 1, and this applies to all the columns of W.

We know that A has an eigenvalue of 1 and that there is a corresponding eigenvector w_i. In Problem 5 we have shown that the eigenspace $\mathscr{E}_1 = span\{\mathbf{e}\}$ for A^T. In this problem we then have $\mathscr{E}_1 = span\{w_i\}$.
This means that the corresponding eigenvectors of A to the eigenvalue 1 is multiple of each other. One of these eigenvectors will sum to 1, which will be a probability vector denoted by $\mathbf{x}$.
By using the probability vector $\mathbf{x}$ as a basis, all the columns of W will then be a scalar multiple of $\mathbf{x}$.

From Problem 2 we know that A^k is a Markov matrix, therefore W must be a Markov matrix. And furthermore W must be a regular Markov matrix, as A is a regular Markov matrix, and every entry of the product of two regular Markov matrices can be descripted by: $(AB)_{ij} = \sum_k (A_{ik}B_{kj})$, where all terms in the sum are strictly positive, thus the sum is strictly positive.

Every column in a Markov matrix sum to 1, therefore every column in W, w_i, is a probability vector with strictly positive entries. The only scalar multiple of $\mathbf{x}$, which sum to 1, is $\mathbf{x}$ itself. All the columns in W must therefore be identical to $\mathbf{x}$. Therefore W takes the form: $W = [w_1, w_2, ..., w_n] = [\mathbf{x}, \mathbf{x}, ..., \mathbf{x}]$. $\square$

Problem 8

Prove Theorem 3.4 for all cases, using the Jordan Canonical Form.

Proof

Since A is an $n \times n$ matrix, A has n eigenvalues if counted with multiplicity. Since A is a square matrix we can write $A = PJP^{-1}$, for some matrix P, where $J = \begin{bmatrix} J_1 & & 0 \\ & J_2 & \\ & & \ddots \\ 0 & & J_r \end{bmatrix}$ and each block J_m is a square matrix that has an eigenvalue of A down the diagonal, 1's immediately above the diagonal and 0's everywhere else: $J_m = \begin{bmatrix} \lambda & 1 & & 0 \\ & \lambda & 1 & \\ & & \ddots & 1 \\ 0 & & & \lambda \end{bmatrix}$. We can write $A^k = PJ^kP^{-1}$, where $J^k = \begin{bmatrix} J_1^k & & 0 \\ & J_2^k & \\ & & \ddots \\ 0 & & J_r^k \end{bmatrix}$ and each block with a size

$$l \times l\text{: } J_m^k = \begin{bmatrix} \binom{k}{0}\lambda^k & \binom{k}{1}\lambda^{k-1} & \binom{k}{2}\lambda^{k-2} & \cdots & \binom{k}{l-1}\lambda^{k-(l-1)} \\ & \binom{k}{0}\lambda^k & \binom{k}{1}\lambda^{k-1} & \ddots & \vdots \\ & & \binom{k}{0}\lambda^k & \ddots & \binom{k}{2}\lambda^{k-2} \\ & & & \ddots & \binom{k}{1}\lambda^{k-1} \\ 0 & & & & \binom{k}{0}\lambda^k \end{bmatrix}$$

We want to show that W exist for A using the Jordan normal form. For W to exist the matrix A^k has to be converging. We split it up into two cases.

Case 1: $|\lambda| < 1$
First we look at all the eigenvalues that have absolute values smaller than 1. The non-zero values in the Jordan blocks with these eigenvalues will have values: $\binom{k}{0}\lambda^k$ which converges to 0, $\binom{k}{1}\lambda^{k-1}$ which also converges to 0,..., $\binom{k}{l-1}\lambda^{k-(l-1)}$, which converges to 0 for $k > l$. Thus the Jordan blocks with eigenvalues that have absolute values smaller than 1 will all converge to 0.

Case 2: $|\lambda| = 1$
We will prove by contradiction. Suppose that $am(1) > 1$. We know that $gm(1) = 1$ from Problem 5, then there will be a Jordan block, where at least one value is $k\lambda^{k-1}$ when $k \to \infty$, that will go to infinity. This cannot be true since it is a Markov matrix, which by definition has columns that sum to 1. A column with a value that go to infinity wont have this property of a Markov matrix.

Thus $am(1) = 1$ which means that it converges to 1, as the Jordan block only consists of the non-zero value λ^k. Therefore J^k converges so A^k must also converge, and the limit exists.

Now we have the same case as in Problem 7, and the rest of the proof follow from there. □

3 Further questions

Damping factor

The damping factor has an effect on the range of values the algorithm gives. If the damping factor is very low, it is more likely that a page will be chosen at random, than it will be reached via a link. The standard value used in our project is 0.85.

Example: Comparing PageRank calculation with damping factor 0.2, 0.85 and 0.99:

```
>>> web={0: {1, 7}, 1: {3, 6}, 2: {0, 1, 3}, 3: set(), 4: set(), 5: {3, 4, 6}, 6: {0}, 7: {4,
↪  6}}
>>> ranking1, iterations = recursive_pagerank(web,0.00001, 200, 0.2)
PageRank: {0:  0.1413,  1:  0.1277,  2:  0.1065,  3:  0.1334,  4:  0.1256,  5:  0.1065,  6:
↪  0.1384,  7:  0.1206}
Iterations: 3
>>> ranking2, iterations = recursive_pagerank(web,0.00001, 200, 0.85)
PageRank: {0:  0.2077,  1:  0.146,  2:  0.045,  3:  0.1325,  4:  0.1144,  5:  0.045,  6:
↪  0.1764,  7:  0.1332}
Iterations: 21
>>> ranking3, iterations3 = recursive_pagerank(web,0.00001, 200, 0.99)
PageRank: {0:  0.2247,  1:  0.1519,  2:  0.0305,  3:  0.1259,  4:  0.1108,  5:  0.0305,  6:
↪  0.186,  7:  0.1418}
Iterations: 198
```

The PageRank has a more even distribution when the damping factor is set to 0.2 compared to the ones set to 0.85 and 0.99. The PageRank of a page will depend more on the amount of inbound links and the structure of

links in the web, if the damping factor is closer to 1. The algorithm also requires more iterations to calculate if the damping factor is closer to 1, but the algorithm captures the link structure better for higher values of d.

For $d = 1$ the algorithm doesn't necessarily give a unique pagerank.
Example: For pages 0,1,2,3,4, where 1 and 4 links to each other, 0 and 2 links to each other, 0 links to 3, and 3 links to 2. The pagerank can for this example be computed to two different outcomes:

```
>>> web = {0: {2,3}, 1: {4}, 2: {0}, 3: {2}, 4: {1}}
>>> ranking1 = random_surf(web, 100000, 1)
Pagerank: {0:  0.40,  1:  0.0,  2:  0.40,  3:  0.20,  4:  0.0}
>>> ranking2 = random_surf(web, 100000, 1)
Pagerank: {0:  0.0,  1:  0.50,  2:  0.0,  3:  0.0,  4:  0.50}
```

Extending our algorithm

We want to create a function that updates the web and page ranking of a site simultaneously. That means we want to be able to add pages and links, as well as delete them (see appendix 4.4 for the function). The parameters of the function is the web, a dictionary of new pages and links you wish to add, a dictionary of links you want to remove from the web, and a list of pages you want to remove.

We then start by fetching the initial pagerank through the `matrix_pagerank` implementation, and then go through each case if there are elements in the respective lists. When we add the new dictionary, if a page is not already in the web, we also initialize a pagerank of $1/(N + 1)$. Once we have added/removed from the web, we use the recursive implementation to loop through and update the rankings of each page based on the previous ranking. We also make sure to save the pagerank before going through each page, and push it into the pagerank once we have gone through each page.

Comparing our implementations

During the comparisons, our implementations `recursive_pagerank`, `eigenvector_pagerank` and `matrix_pagerank` have the same precision when rounding off at 4 decimal places. Because of the nature of our `random_surf` simulation, we can expect it to varie slightly from the rest. We can calculate the margin of error by comparing each implementation against `eigenvector_pagerank` as the true evaluation (as it has no parameters other than web and damping), and summing up the absolute value of the difference. Looking at the error on a web of 100 pages we see that

```
Error:  random_surf: 0.0242,  recursive_pagerank: 0.0000,  matrix_pagerank: 0.0000
```

We can see that the difference is minimal and within an acceptable margin of error of ≤ 0.025. With this in mind, let us compare our four implementations on larger networks. We will use the time module, and log `time.time()` from the beginning and end of the functions, and return the difference. Let us first define the high precision parameters:

```
>>> random_surf(web, 100000)
>>> recursive_pagerank(web, 0.0000001)
>>> eigenvector_pagerank(web)
>>> matrix_pagerank(web, 20)
```

Similarly, here are our low precision parameters:

```
>>> random_surf(web, 10000)
>>> recursive_pagerank(web,0.00001)
>>> eigenvector_pagerank(web)
>>> matrix_pagerank(web,10)
```

Now, let us first take a look at our web of 10.000 pages with links between 0-2000 for each:

Method	Runtime (High)	Error (High)	Runtime (Low)	Error (Low)	Relation
random surf	186.98s	0.2507	18.49s	0.7416	10.11x
recursive	39.12s	0.0001	19.34s	0.0001	2.02x
eigenvectors	135.87s	-	135.42s	-	1.00x
matrix	35.46s	0.0000	30.29s	0.0000	1.17x

Table 1: Runtimes for 10.000 pages varying in precision (high vs low).

The relative time between `random_surf` in high vs low precision is significant, but we also see a large increase in error. This is because of a scaling problem we will touch more on below. Notice the matrix method is faster than the recursive method at high precision, but on low precision the recursive method becomes faster. We also see that the relation of recursion is $2.02x$ faster, but still maintaining the same error in both high and low precision. This means we could have probably lowered the precision slightly to reduce the overall runtime of the method whilst maintaining accurate results.

Below is a comparison for a web of 20.000 pages and between 0-4000 links each:

Method	Runtime (High)	Error (High)	Runtime (Low)	Error (Low)	Relation
random surf	384.68s	0.35042	37.68s	1.2180	10.21x
recursive	155.00s	0.0000	95.20s	0.0000	1.62x
eigenvectors	988.65s	-	964.14s	-	1.02x
matrix	275.41s	0.0000	214.73s	0.0000	1.28x

Table 2: Runtimes for 20.000 pages varying in precision (high vs low).

Notice a scaling problem we have for the random surf implementation. The margin of error is significantly higher when run on a larger network, considering the same amount of iterations. We also observe that, upon lowering the precision (less iterations), the error margin becomes so high that the ranking is completely wrong. This is because the number of iterations needed for the method to converge depends on the amount of pages we have. If there are 100.000 iterations on a web of 100 pages it may be sufficient with a 0.0242 error margin. This does not scale for a web of 10.000 or 20.000 pages, as we have the same amount of iterations, but the amount of pages are much larger, so we aren't able to go through each page enough times to calculate the pagerank. Therefore it is evident that the random surf model would not work well on large networks, as the runtime needs to be exceedingly higher for larger webs, otherwise we lose precision. This is very inefficient.

We also see that the recursive implementation becomes faster than the matrix implementation on a larger network. This likely happens because the recursive method only updates the pages and links necessary, but the matrix method has to multiply incredibly large matrices together which might take a lot of memory on larger networks. We can see in Table 2, when we lower the precision, that the recursive method runs $1.62x$ faster whereas the matrix method only runs $1.28x$ faster than at high precision, but with the exact same error margins.

The matrix method is indeed faster than the eigenvector method as expected from our solutions, but the overall best performance is from the recursive model on larger networks as seen in Table 2.

4 Appendix

4.1 pagerank_script1.py

```
1
2  import numpy as np
3  ########################  Helper Functions ###############
4  def make_web(n,k,kmin=0):
5      assert(k<n)
6      keys = np.array(range(n))
7      web = dict()
8      for j in keys:
9          numlinks = np.random.choice(range(kmin,k+1))
10         web[j] = set(np.random.choice(keys[keys!=j],numlinks,replace=False))
11     return web
12
13 ########       Programming Task 1   ############################
14 def surf_step(web, page, d=0.85):
15     """
16     Return a probability distribution over which page to visit next,
17     given a current page.
18     Input: -web is a dictionary of webpages and links
19            - page is a key of the dictionary, the page from which to
20            compute the next step
21            - d is the damping factor
22
23     - If page does not link to any page, then choose any page in web at random
24     - Otherwise:
25           With probability `d`, choose a page at random linked to by `page`,
26           With probability `1 - d`, choose  a page at random  from all pages in the web.
27     Otuput: is a dictionary with the same keys as web, and the
28             value for each key is the probability of choosing that page next.
29             (The sum of all these should be 1)
30     """
31     distribution = dict() # the distribution dictionary
32     N = len(web.keys())
33
34     # initialize that every key in web needs a place in distribution
35     for key, val in web.items():
36         distribution[key] = 0
37
38     # If page does not link to any page, then choose any page in web at random
39     if len(web[page]) < 1:
40         for key, val in web.items():
41             distribution[key] += 1 / N
42
43     # Otherwise
44     if len(web[page]) > 0:
45         # With probability `d`, choose a page at random linked to by `page`,
46         for link in web[page]:
47             distribution[link] += d / (len(web[page]))
48         # With probability `1 - d`, choose  a page at random  from all pages in the web.
49         for key, val in web.items():
50             distribution[key] += (1-d) / N
```

```
    return distribution


def random_surf(web,n,d=0.85):
    """
    Return PageRank values for each page by sampling `n` pages
    according to surf_step.
    Input: web is a dictionary of webpages and links,
           n is an integer, the number of steps in the simulation
           d is the damping factor,

    Returns a dictionary with thes same keys as web (the pages), and
    the value for key k is the page rank of page k. The sum of all PageRank values
    should be 1.
    """
    ranking = dict() # the ranking for each page
    all_pages = list(web.keys())

    # initialize that every key in web needs a place in distribution
    for key, val in web.items():
        ranking[key] = 0

    # Choose the first page randomly
    p = np.random.choice(all_pages)
    # Loop through the simulation n times
    for _ in range(n):
        # Add 1 each time we visit a site
        ranking[p] += 1
        # Get our probability distribution of the page p
        probdist = surf_step(web, p, d)
        # Make a list of the probabilities of visiting the next site, given we're at page
        ↪ p
        probs = []
        for key, val in probdist.items():
            probs.append(val)
        # Choose a new page based on that probability
        p = np.random.choice(all_pages, p=probs)

    # Divide by n to get the pagerank
    for key, val in ranking.items():
        ranking[key] /= n

    return ranking
```

4.2 pagerank_script2.py

```
###### Programming Task 2

def rank_update(web,pageranks,page,d):
    ''''
    Updates the value of the pagerank for page based on the formula
        PR(p)= (1-d)/N + d*sum_j (PR(q)/OB(q))
    where the sum is over all pages q that link to page, PR(q) is the current
```

```
    pagerank of page q (from "pageranks") and OB(q) is the number of pages
    outbound from page q.  Sinks are treated as linking to all pages in web.

    Input: web and pageranks are dictionaries as in the output of "make_web" and
     "random_surf", page is the key to the page whose rank we wish to update,
         and d is the damping factor.
    Output: The (mutable) dictionary "pageranks" is updated according to the above
    ↪  formula,
            and this function returns a float "increment", the (absolute) difference
            between the previous value and the updated value of PR(p).
     '''
    increment = 0
    # Pagerank of initial page
    PR_old = pageranks[page]
    inbound_p = set()
    N = len(web.keys())

    # add elements to inbound_p
    # loop through and check if each links to p
    for key, val in web.items():
        # if the page is in the links of key, add it to inbound
        if page in val:
            inbound_p.add(key)

        # sink, if page does not link to any pages, then add to inbound_p
        elif not web[key]:
            inbound_p.add(key)

    # split the equation up between first and second part
    del1 = (1 - d) * (1 / N)
    del2 = 0

    for q in inbound_p:
        # pagerank and outbound of q
        PR_q = pageranks[q]
        OB_q = len(web[q])

        # sink for sinks q, outbound is N
        if OB_q < 1:
            OB_q = N

        del2 += (PR_q) / (OB_q)

    # modify the existing pageranks dictionary
    pageranks[page] = del1 + (d * del2)
    increment = abs(PR_old - (del1 + (d * del2)))

    return increment

def recursive_pagerank(web,stopvalue,max_iterations=200,d=0.85):
    """
    Implements the recursive version of the PageRank algorithm by first creating a
    pagerank of 1/N to all pages (where N is the total number of pages)
    then applying "rank_update" repeteadly until either of two stopping conditions is
```

```
    reached:
    stopping condition 1: the maximum change from step n to step (n+1) over all pageranks
    is less than stopvalue,
    Stopping condition 2: the number of iterations has reached "max_iterations"

    Input: web is a dictionary as in the output of "make_web", d is the damping constant,
    stop value is a positive float, max_iterations is a positive integer
    """
    pageranks = dict()
    iteration = 0
    N = len(web.keys())

    for key, val in web.items():
        pageranks[key] = 1 / N

    # Loop through until max iterations
    for n in range(max_iterations):
        # Log the change diffs
        max_change = 0

        # Go through each page and update the rank
        for page in web.keys():
            change = rank_update(web, pageranks, page, d)
            if change > max_change:
                max_change = change

        # Break out if it reaches below stopvalue
        if(max_change < stopvalue):
            break

        iteration += 1

    return pageranks, iteration
```

4.3 pagerank_script3.py

```
#################### Programming Task 3  ################

def modified_link_matrix(web,pagelist,d=0.85):
    """
    Create a modified link matrix from web
    Input: web is a dictionary whose keys are contained in the list pagelist,
           pagelist is just a list of the keys (to give the keys and ordering)
           d is the damping factor
    Output: d*A^T+(1-d)*E/N
          where A is an NxN numpy array, for which
          row j has non-zero entries only in the columns corresponding
          to pagelist to which page j links.  In a given row, all non-zero entries have
          ↪  the
          same value, and these values sum to 1. Special case: if page j does not link
          to any page, then all entries in row j are given the same value (1/N).
          E is np.ones([N,N])
    """
    N = len(web.keys())
```

```
    E = np.ones([N,N])
    A = np.zeros([N,N])

    # Modified adjacency matrix
    for j in range(N):          # cols
        for i in range(N):      # rows
            # if p_j links to p_i
            if pagelist[i] in web[pagelist[j]]:
               N_j = len(web[pagelist[j]])
               A[j,i] = 1 / N_j

            # if p_j is a sink
            if len(web[pagelist[j]]) < 1:
               A[j, i] = 1 / N

    return d * A.T + (1 - d) * (E / N)

def eigenvector_pagerank(web,d=0.85):
    """
    Returns the pagerank of web as the eigenvector of the modified link matrix
    Input: web is a dictionary of web pages and lines.
           d is a positive float, the damping constant
    Output: A dictionary with the same keys as web, and the values of the pageranks of
    ↪  the keys
    """
    ranking = dict()

    # Init dict with same keys as web
    for key, val in web.items():
        ranking[key] = 0

    pages = list(web.keys())
    # Create our adjacency matrix M from function before
    M = modified_link_matrix(web,pages, d)

    # Use np eigenvector function to find our eigenvals and vects
    lamda, V = np.linalg.eig(M)
    # Choose the first vector
    V1 = V[:,0:1]    # the eigenvalue method
    # normalize the eigenvector so that it's components sum to 1
    V1 = np.real(V1)
    ranking3 = V1/(np.sum(V1))  # ranking from eigenvector 1

    # Loop through the vector and input them into the ranking dict.
    for k in range(len(pages)):
        ranking[k] = float(ranking3[k])

    return ranking

def matrix_pagerank(web,power,d=0.85):
    """
    Returns the pagerank as the first column of the power'th power of the modified link
    ↪  matrix

```

```
    Input: web is a dictionary of web pages and lines.
           d is a positive float, the damping constant
    Output: A dictionary with the same keys as web, and the values the pageranks of the
    ↪  keys
    """
    ranking = dict()

    for key, val in web.items():
        ranking[key] = 0

    # Create pages and M adjacency matrix
    pages = list(web.keys())
    M = modified_link_matrix(web,pages, d)

    # Use method (b) by computing powerth power of M
    K = np.linalg.matrix_power(M,power)
    # Choose the first col
    new_rank = K[:,0]

    # Loop through the vector and input it into the ranking dict.
    for k in range(len(pages)):
        ranking[k] = float(new_rank[k])

    return ranking
```

4.4 update_web function (inside pagerank_script3.py)

```
def update_web(web, webadd={}, removelinks={}, removepages=[], d=0.85):
    pageranks = matrix_pagerank(web,20)
    N = len(web.keys())

    # Add webadd to web
    # Initialize a pagerank if not already in web
    if len(webadd) > 0:
        for key, val in webadd.items():
            if key not in web:
                web[key] = set()
                pageranks[key] = 1 / (N + 1)

            for num in val:
                web[key].add(num)

    # Go through removelinks dict
    # and remove the links associated with the key
    if len(removelinks) > 0:
        for key, val in removelinks.items():
            for num in val:
                web[key].remove(num)

    # Go through removepages list and remove each page
    # and all its inbound and outbound links
    if len(removepages) > 0:
        for page in removepages:
            # Remove page from web and pageranks
```

```
            if page in web:
                del web[page]
            if page in pageranks:
                del pageranks[page]

            # Go through every single link in the web and remove page
            for key, val in web.items():
                if page in val:
                    web[key].remove(page)

    # Now we loop through (from recursive model)
    # and update rank based on previous rankings
    for _ in range(200):
        # Log the change diffs to break out if it reaches below stopvalue
        max_change = 0
        # Remember to save the ranks whilst going through each page
        new_ranks = pageranks.copy()
        # Go through each page and update the rank
        for page in web.keys():
            change = rank_update(web, new_ranks, page, d)
            if (change > max_change):
                max_change = change

        pageranks = new_ranks
        # Break out if it reaches below stopvalue
        if(max_change < 0.0000001):
            break
```