

# COMP 360 Study guide

Francis Piché

January 31, 2019

# Contents

<b>I</b>	<b>Preliminaries</b>	<b>4</b>
1	License Information	4
2	About This Guide	4
<b>II</b>	<b>Analyzing Key Algorithms</b>	<b>4</b>
<b>3</b>	<b>Data Compression</b>	<b>4</b>
3.1	Information Theory . . . . .	5
3.2	Variable Length Encoding . . . . .	5
3.2.1	Shannon-Fano Code . . . . .	6
3.2.2	Huffman Coding . . . . .	6
3.2.3	Entropy . . . . .	6
<b>4</b>	<b>JPEG Compression</b>	<b>6</b>
4.1	Discretization of Images . . . . .	6
4.2	JPEG Compression Algorithm . . . . .	7
4.2.1	Step 1: RGB to YCC Encoding . . . . .	7
4.2.2	Step 2: Compress Chromaticity . . . . .	7
4.2.3	Step 3: Discrete Cosine Transform . . . . .	8
4.2.4	Step 4: Quantization . . . . .	8
4.2.5	Step 5: Huffman Encoding . . . . .	9
4.3	Discrete Fourier Transform . . . . .	9
4.3.1	Roots of Unity . . . . .	9
4.3.2	The Transform . . . . .	10
4.3.3	Fast Fourier Transform . . . . .	10
<b>5</b>	<b>Fast Matrix Multiplication</b>	<b>11</b>
5.0.1	Slow Matrix Multiplication . . . . .	11
5.1	Strassens Algorithm . . . . .	12
5.2	Matrices with Repeated Rows . . . . .	13
5.3	Sparse Matrices . . . . .	13
<b>6</b>	<b>Google Page Rank</b>	<b>13</b>
6.1	Search Engine Basics . . . . .	14
6.2	What is PageRank? . . . . .	14
6.3	Dividing Trust . . . . .	14
6.4	The Problem . . . . .	15
6.5	Calculating PageRank . . . . .	15
6.6	The Runtime . . . . .	16

<b>7</b>	<b>Simplex Algorithm for Linear Programming</b>	<b>16</b>
----------	-------------------------------------------------	-----------

## Part I

# Preliminaries

## 1 License Information

These notes are curated from Professor Bruce Reed COMP360 lectures at McGill University. They are for study purposes only. They are not to be used for monetary gain.

*This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 2.5 Canada License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc-sa/2.5/ca/> or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.*

## 2 About This Guide

I make my notes freely available for other students as a way to stay accountable for taking good notes. If you spot anything incorrect or unclear, don't hesitate to contact me via Facebook or e-mail at <http://francispiche.ca/contact/>

## Part II

# Analyzing Key Algorithms

## 3 Data Compression

As humans utilize more and more data, techniques are required for handling and storing this data efficiently. Some examples of data transformations:

- Compression for storage (minimize file sizes on JPEG, mp3 etc)
- Compression & redundancy adding for secure transmission
- Sorting data
- Watermarking
- Discretizing continuous measurements
- Representing information as bits
- Natural language translation

### 3.1 Information Theory

First we need to think about what information is really needed? What information do we need to record to maintain the comprehensiveness of the original structure?

**Lossy** compression such as mp3 and jpeg files, results in *approximate* communication, since the compression is done by approximating some points. (Data is lost). In lossy compression, the more data is compressed, the more it becomes incomprehensible.

**Lossless** compression on the other hand, such as gif, gzip, or tar files. As the name implies, the original data can be recovered without any loss.

A lot of information is redundant. For example, the phrases: *The colour of the chair is red. The colour of the hat is red. The colour of the table is red.* Could all be simplified to *The chair, hat and table are red.*

The optimal strategy depends on the type of data we are dealing with. For example, transmitting numbers between 0 and 31 we need 5 bits, but for transmitting numbers between 9 and 31 we only need 4, since we can transmit  $x-9$ , and add it back on the other side. Suppose further that we know the first number is at most 31 and that all numbers differ at most 3? Transmit the first number  $x_1$  needing 5 bits, and then each following number only 3 bits, since we could transmit  $x_i - x_{i-1} + 3$ , and then only send from 0-6. If the number is 0 then the previous was 3 greater than the current, and if the number is 6 then we know that the previous was 3 less than the current.

The optimal strategy also depends on the encoding technique. For example, if we are transmitting a 50x50 2D array of bits indicating the position of 10 mines, we could send the whole array (2500) bits, or we could just send the coordinates of each mine (12 bits per mine, so 120 bits total).

### 3.2 Variable Length Encoding

Suppose we know the frequency of the symbols in our message, but nothing about the structure of the message.

Say our alphabet is  $\{A, B, C, D\}$ . With 15 A's, 7B's, 6C's, and 5 E's for 39 total. We only have 5 symbols so we could just use 3 bits per symbol by letting A=000, B=001, and so on.

We can do better by building a **prefix code**. A prefix code is one such that no two symbols  $x$  and  $y$  can be encoded such that the encoding of  $x$  is a prefix of the encoding of  $y$ . So we can't have  $x = 1000$  and  $y = 100011$

### 3.2.1 Shannon-Fano Code

In this technique, we order the symbols by frequency, decreasing. Then, we partition the set of symbols so that the sum of frequencies on each side of the partition is as close to equal as possible. Then, we use 0's to encode everything on one side of the partition, and 1's for everything on the other side. We repeat until we can progress no further.

So for our previous example, A-15, B-7, C-6, D-6, E-5, if we group A+B=22 then it's as close to equal to C+D+E=17 as we can get. Then we split A from B, as that's all we can do there. Then we split the CDE group into C, DE. And then D, E.

From this algorithm, we can see that if a node  $X$  is more frequent than  $Y$ ,  $X$  cannot be deeper than  $Y$ . Similarly, there cannot be a node at the deepest level without a sibling. (By the structure of the tree, the parent is redundant). Swapping the leaves at the same level does not change the number of bits per symbol.

### 3.2.2 Huffman Coding

Improving on the Shannon-Fano Code, we can build the tree by using what we observed in the analysis above. Since the order on the same level doesn't matter, and the leaves will always be in pairs, we can build it up by taking the least frequent two symbols, and making their parent the sum of the two frequencies. We then add nodes to the tree in this manner until we can't anymore. (See my COMP251 study guide for more on Huffman Coding).

### 3.2.3 Entropy

The entropy of a probability distribution  $p$  on the letters of an alphabet is

$$\sum_{x \in S} -p(x) \log_2(x)$$

The Huffman code is actually always within 1 bit per symbol of the entropy, and we cannot do better than entropy.

## 4 JPEG Compression

JPEG compression is less of a predefined algorithm than a selection of subroutines. Some of these subroutines are mandatory for the overall result and some are optional. Often there is a tradeoff between image quality retention and the degree to which we compress.

Often, due to these tradeoffs, websites will send multiple images of increasing quality.

### 4.1 Discretization of Images

For black and white (grey scale) images, if an image is a grid of pixels, then each pixel holds one value. This value is the intensity of the grey. 255 would be white, whereas 0 would be

black.

For color images, we need more information. We instead combine 3 primary colours Red, Green, Blue (RGB).

*Side note: if you're interested, my COMP557 (Computer Graphics) guide has a section on colour theory which I thought was really cool. It explains why we chose RGB over some other colours).*

## 4.2 JPEG Compression Algorithm

The main goal is to throw away as much information as possible while keeping the image looking as close to the original as possible.

### 4.2.1 Step 1: RGB to YCC Encoding

The first step is to convert from RGB to a different encoding, called  $YC_R C_B$  (luminance, chrominance-red, chrominance-blue). The reason for this is because the human eye picks up more on "brightness" information than colour information. So, encoding this way allows us to separate the Y ("brightness"/luminance) from the colour/chrominance. This allows for the colour information to be stored at a lower resolution (with less information) without losing too much visual information.

First, we use the following constants (determined experimentally, just take them for granted).

$$Y = 0.2990R + 0.5780G + 0.1440B$$

$$C_B = -0.1687R - 0.331G + 0.5B + 2^{BitDepth-1}$$

$$C_R = 0.5R - 0.4187G - 0.0831B + 2^{BitDepth-1}$$

### 4.2.2 Step 2: Compress Chromaticity

The idea here is to downsize the amount of information we need by averaging the  $C_B$  and  $C_R$  values of adjacent pixels together. If we do this by taking 2x2 blocks of pixels, and averaging them into 1 pixel, we save 50% space.

We save 50% since we have 3 values per pixel originally. So  $3P$  total numbers, where  $P$  is the number of pixels. Next, we cut the number of pixels into 4 for both  $C_R$  and  $C_B$  so  $2(P/4)$ . In total we now have  $3P/2$  which is half of  $3P$ .

### 4.2.3 Step 3: Discrete Cosine Transform

This is where things get messy. To explain this, I really recommend this video (hyperlink) by Computerphile. Honestly, it does a better job than I ever could, but I'll give it a short either way.

We now divide the image into 8x8 blocks of pixels. The DCT (discrete cosine transform) is applied to each block. The effect is that the spatial image is now a sort of "frequency map". The frequency being how often the luminance is changing from low to high. The idea is that by removing some of the high frequency information we can still maintain the resemblance of the original.

When two cosine waves are added, a new wave is created. So, we use 64 (8x8) cosine waves with specially determined weights to try to recreate the image. The top left of these cosine waves is considered the DC (direct current) wave, and contributes the most to the final image. (usually). All the other waves are called AC (alternating current). As you move right and down, the waves are of increasing frequency in the x direction (right) and y direction (down).

By superimposing some linear combination of these waves we can recreate the original image.

To find these values, we must first "normalize" our data. Currently the pixel values run from 0-255, so we need to subtract 128 so that they are centered about 0. -127 to 127. We do this since cosine runs from -1 to 1.

The 2x2 example is as follows:

If we have a 2x2 block, we need a combination of 4 cosine waves. This will be some linear combination

$$\begin{pmatrix} p_1 & p_2 \\ p_3 & p_4 \end{pmatrix} = x_1 \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix} + x_2 \begin{pmatrix} 1 & -1 \\ -1 & 1 \end{pmatrix} + x_3 \begin{pmatrix} 1 & 1 \\ -1 & -1 \end{pmatrix} + x_4 \begin{pmatrix} 1 & -1 \\ 1 & -1 \end{pmatrix}$$

Note that those are the 4 unit cosine matrices.

We can solve this system of equations by combining into one matrix:

$$\begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & -1 & -1 & 1 \\ 1 & 1 & -1 & -1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} p_1 \\ p_2 \\ p_3 \\ p_4 \end{pmatrix}$$

By inverting the left matrix, and multiplying on both sides, we can solve for  $x_1, x_2, x_3, x_4$ . This will give us the weights for each of the cosine waves that will give the same values as the original image  $p_1, p_2, p_3, p_4$ .

### 4.2.4 Step 4: Quantization

Here is where much of the data is thrown away (and some is lost!). This step is lossy. Up until this point, we have retained all original data and the process has been reversible.



We will now "quantize" the DCT coefficients computed in the last step to get rid of information. This is done by dividing each coefficient by some number. This number is given by a Quantization table (predetermined). Depending on how much you want to compress (and lose) you can multiple the table by a scale. So the final DCT transform we will apply is given by:

$$DCT_{quantized} = ROUND\left(\frac{DCT_{coef}}{Q * Scale}\right)$$

Where  $Q$  is from the quantization table. The Scale determines how much quality you want to retain. (100 being perfect quality, 50 being half quality etc.). The ROUND will get rid of all the almost-zero (negligible) values. This is where we lose the data (and gain the compression).

#### 4.2.5 Step 5: Huffman Encoding

The final values are now encoded using the Huffman algorithm, sent, and decoded on the other side, by reversing the process. Of course, the values that come out the other end are not exact, but close enough that the human eye probably won't pick up on it in cases of slight compression.

### 4.3 Discrete Fourier Transform

An alternative to the DCT is the Discrete Fourier Transform (DFT). It is applied the same way as described above, except that we use a different matrix (and method for finding the coefficients).

I'm going to assume you're familiar with complex numbers and the complex plane, but not with roots of unity.

#### 4.3.1 Roots of Unity

To understand the DFT, we first need to understand Roots of Unity.

A root of unity is a complex number which, when raised to a positive integer power, results in 1.

For any positive integer  $n$ , the  $n$ th roots of unity are complex solutions to  $\omega^n = 1$  and there are  $n$  solutions to the equation. These solutions (found by Eulers formula) are  $\omega_n^k = e^{\frac{2k\pi i}{n}}$  with  $k = 0, \dots, n - 1$ .

Some facts:

$$\omega_n^k \omega_n^j = \omega_n^{i+j}$$

$$\begin{aligned}\omega_n^n &= 1 \Rightarrow \omega_n^{j+n} = \omega_n^j \\ \omega_n^{\frac{n}{2}} &= -1 \Rightarrow \omega_n^{j+\frac{n}{2}} = -\omega_n^j \\ \sum_{m=0}^n -1 \omega_n^{(j+k)m} &= 0\end{aligned}$$

this last one is because for each  $\omega_n^k$  there is a corresponding  $\omega_n^j$  such that  $\omega_k = -\omega_n^j$  Where  $j = k + n/2$

### 4.3.2 The Transform

Now we need to find coefficients so that

$$b_k = \sum_{j=0}^{n-1} M_j \omega_n^{jk}$$

$$\begin{pmatrix} 1 & 1 & \dots & 1 & 1 \\ 1 & \omega_n^1 & \dots & \omega_n^{n-2} & \omega_n^{n-1} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 1 & \omega_n^{n-2} & \dots & \omega_n^{(n-2)(n-2)} & \omega_n^{(n-1)(n-2)} \\ 1 & \omega_n^{(n-1)} & \dots & \omega_n^{(n-2)(n-1)} & \omega_n^{(n-1)(n-1)} \end{pmatrix} \begin{pmatrix} M_0 \\ M_1 \\ \vdots \\ M_{n-1} \end{pmatrix} = \begin{pmatrix} b_1 \\ \vdots \\ b_{n-1} \end{pmatrix}$$

Similarly DCT, we can solve for the matrix inverse and multiply it by  $M$  to find  $b$ .

The inverse is:

$$\frac{1}{n} \begin{pmatrix} 1 & 1 & \dots & 1 & 1 \\ 1 & \omega_n^{n-1} & \dots & \omega_n^{(n-2)(n-1)} & \omega_n^{(n-1)(n-1)} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 1 & \omega_n^2 & \dots & \omega_n^{(n-2)2} & \omega_n^{(n-1)2} \\ 1 & \omega_n^{(1)} & \dots & \omega_n^{(n-2)} & \omega_n^{(n-1)} \end{pmatrix}$$

Notice that the inverse is just the vertically flipped version of the original, with its entries multiplied by  $1/n$ . So it can be quite fast to compute this inverse.

Now, because of the special structure of the inverse, we can compute the multiplication of by a vector in  $n \log(n)$  time in what's known as the Fast Fourier Transform.

### 4.3.3 Fast Fourier Transform

The goal of this divide and conquer algorithm is to quickly multiply some matrix  $A$  by some vector  $\langle a_0, \dots, a_{n-1} \rangle$ . Note that this is going to give a resulting vector  $A(x)$ . We can try to split up the problem into dealing with odd and even indices for the vector coefficients.

Then the final vector  $A(x) = xA_{\text{odd}}(x^2) + A_{\text{even}}(x^2)$  Why these terms? Because matrix multiplication by a vector is just like a polynomial. So the polynomial version of this is:

$$A(x) = \sum_{j=0}^{n-1} a_j x^j$$

Then, we can split this into:

$$A_{\text{even}}(x) = \sum_{j=0}^{\frac{n}{2}-1} a_{2j} x^{2j} = A(x^2)$$

and

$$A_{\text{odd}}(x) = \sum_{j=1}^{\frac{n}{2}-2} a_{2j+1} x^{2j+1} = xA(x^2)$$

However, we also are now operating on  $x^2$  not  $x$ . In order to get the runtime down from  $O(n^2)$  to  $O(n \log(n))$  we need to have inputs  $x^2$  to our polynomial function  $A(x^2)$  such that  $x^2 = x$ .

This is where the roots of unity come in. Since  $\omega_n^{j+n/2} = -\omega_n^j$ , we have that  $\omega_n^{(j+n/2)^2} = \omega_n^{j^2}$  which cuts our input in half.

We are now operating on a set of size  $n/2$ , with linear processing time at each step, which is  $O(n \log(n))$  by the master theorem.

## 5 Fast Matrix Multiplication

Regular matrix multiplication is quite slow. In the The brute force method we would need up to  $n^3$  computations on square  $n \times n$  matrices.

In the previous chapter, we saw that in the special case of the Fast Fourier transform, we can multiply the special matrix by a vector in  $n \log(n)$  time.

But what if there is no special structure?

### 5.0.1 Slow Matrix Multiplication

In regular matrix multiplication  $AB$ ,  $A_{m \times n} B_{n \times p}$ , we need to compute:

---

```
for(i=1; i<=m; i++){
    for(j=1; j<=p; j++){
        for(k=1; k<=n; k++){
            C[i][j] += A[i, k]B[k, j]
        }
    }
}
```

---

In other words, each entry of  $C$  is computed by the dot product of each row of  $A$  with each column of  $B$ .

As you can see, we have a triple for-loop with upper bounds,  $m$ ,  $n$ , and  $p$  so  $O(mnp)$

We can split this up using divide and conquer, which may improve the running time. We can split the rows and columns in two, and compute each half individually.

More specifically, each entry of  $C$ ,

$$\begin{aligned} c_{i,j} &= a_{0,\dots,i/2} \cdot b_{0,\dots,j/2} + a_{i/2,\dots,m} \cdot b_{j/2,\dots,n} \\ &= \sum_{k=0}^{m/2} a_{i,k} b_{k,j} + \sum_{k=m/2}^m a_{i,k} b_{k,j} \end{aligned}$$

So we will split our matrix up into 4 blocks.

$$\begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix}$$

All we then have to do is recursively split up the matrix into smaller blocks, then compute the multiplications and additions on the way up. This is still  $O(n^3)$  however.

This can actually be done with 7 multiplications instead of 8, using a trick.

## 5.1 Strassen's Algorithm

We can get to  $O(n^{\log(7)})$  by doing more additions and less multiplications. Normally, we have 8 multiplications, and 4 additions.

$$\begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} = \begin{pmatrix} A_{1,1}B_{1,1} + A_{1,2}B_{2,1} & A_{1,1}B_{1,2} + A_{1,2}B_{2,2} \\ A_{2,1}B_{1,1} + A_{2,2}B_{2,1} & A_{2,1}B_{1,2} + A_{2,2}B_{2,2} \end{pmatrix}$$

The trick is to do:

$$\begin{aligned} S_1 &= B_{1,2} - B_{2,2}, \quad S_2 = A_{1,1} + A_{1,2} \\ S_3 &= A_{2,1} + A_{2,2}, \quad S_4 = B_{2,1} - B_{1,1} \\ S_5 &= A_{1,1} + A_{2,2}, \quad S_6 = B_{1,1} + B_{2,2} \\ S_7 &= A_{1,2} - A_{2,2}, \quad S_8 = B_{2,1} + B_{2,2} \\ S_9 &= A_{1,1} - A_{2,1}, \quad S_{10} = B_{1,1} + B_{1,2} \end{aligned}$$

Then by letting:

$$P_1 = A_{1,1}S_{1,1}, \quad P_2 = S_2B_{2,2}$$

$$P_3 = S_3 B_{1,1}, P_4 = A_{2,2} S_4$$

$$P_5 = S_5 S_6, P_6 = S_7 S_8, P_7 = S_9 S_{10}$$

$$\begin{pmatrix} P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\ P_3 + P_4 & P_5 + P_1 - P_3 + P_7 \end{pmatrix}$$

We now have 18 additions, and only 7 multiplications.

Each multiplication is a recursive call on an  $n/2$  square matrix, so we have the recurrence:

$$T(n) = O(n^2) + 7T\left(\frac{n}{2}\right)$$

Which gives  $O(n^{\log(7)})$  by the master theorem.

## 5.2 Matrices with Repeated Rows

If a matrix has several repeated rows (or few distinct rows), then we can compute  $x' = Mx$  for  $x_{n \times 1}$ ,  $M_{n \times n}$  in  $O(kn)$  time, where  $k$  is the number of distinct rows.

The idea is to keep track of which rows are duplicate, and only compute the distinct rows. Then, we can just plug the repeated rows into the result.

For example:

$$\begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 2 & 3 & 2 \end{pmatrix} \begin{pmatrix} 2 \\ 3 \\ 2 \\ 2 \end{pmatrix} = \begin{pmatrix} 9 \\ 9 \\ 9 \\ 20 \end{pmatrix}$$

This is  $O(nk)$  since for each entry of  $x'$  ( $n$  such entries), we need to find which of the  $k$  rows this row is a copy of ( $k$  work)

## 5.3 Sparse Matrices

Suppose we have a matrix that is mostly zeroes. It has  $k$  non-zero entries. For each non-zero entry of the matrix we keep track of it's row,  $i$ , its column,  $j$  and its value,  $a_{i,j}$ . Then we can initialize all entries of our result vector to 0, and iterate through the non-zero entries, only updating the necessary entries by updating to:

$$x'_i + = a_{i,j} x_j$$

Then this algorithm is  $O(k + n)$  since we do  $n$  work to figure out the  $k$  entries that are full, then  $k$  work to update the values of  $x'$

# 6 Google Page Rank

With so much information available, how can we rank Google search results by relevance?

## 6.1 Search Engine Basics

Three parts:

- 1. Crawlers collect the information on pages, links between them and collect as much data as possible. This is a continuous process.
- 2. A database indexes all this information
- 3. A person makes a query, and relevancy software figures out what the best results are. (Which are relevant to the user).

Interpreting a query involves applying natural language processing to extract the meaning, and figure out the type of information you want to generate keywords with which to query the index.

## 6.2 What is PageRank?

There are many important (200+) factors which Google uses to rank results. **PageRank** is the factor of *trust* the web community has in the page. Trust is measured based on a few factors.

We could try to just see how many links point to a given page. This is called the **in-degree** of a page. We can refine this with a *weighted indegree*, in which if someone with a high indregree points to someone, then that link is worth more.

We can continue to add approximations by, having computed the score under the  $i$ -th approximation, we can compute the  $i + 1$  as the sum of the  $i$  scores of the web pages that link to it.

We must still be careful however, since if a page links to a lot of other pages, it might mean it's not careful about who it links to, and may not be as trust worthy. So we can divide it's weight by the number of links leaving it.

## 6.3 Dividing Trust

A first approach to dividing trust amongst page is to initially consider each page as equally trustworthy. We then repeatedly divide the trust currently assigned to a webpage equally amongst all the pages it links to.

So if you have 4 pages, then initially each page will have  $\frac{1}{4}$  trust. Then, say the first page links to two others, then in the second iteration, that page now has  $\frac{1}{4} - \frac{2}{8} = \frac{1}{12}$  Trust.

Over many iterations, this will lead to an equilibrium state, in which the values no longer change.

$$r(p) = \sum_{p' \text{ linkstop } p} \frac{r(p')}{d^+(p')}$$

where  $d^+(p)$  is the number of links out of  $p$ .

## 6.4 The Problem

This doesn't always work, however. The problem arises when a page has no outward links. This is because normally, a page has its trust divided equally amongst its outgoing arcs. This means that the total amount flowing into all the pages is  $\sum_{p \in \text{Pages}} r(p)$ . By our equation in the previous subsection, we have that the sum of all rank flowing out of pages is  $\sum_{p \in \text{Pages} \wedge d^+(p) > 0} r(p)$ . But this means that the rank for a page with no rank has to be 0. It turns out that any page which is in the same path as a page with 0 outgoing links will have rank 0. Also, the ranks need not be unique, which poses a problem for ranking.

The solution is to divide the rank of a page with no outgoing links equally amongst all pages. Also, we take every page and divide 15% of its rank amongst all pages. The remaining 85% is divided as before. By some black magic, this gives each page a unique rank.

## 6.5 Calculating PageRank

PageRank for all pages can be seen as a vector,  $R = \begin{pmatrix} r_1 \\ \vdots \\ r_N \end{pmatrix}$ . We can express one iteration

of our dividing of the trust as a matrix  $P$ , where  $p_i$  is the  $i$ th column of  $P$ , then:

If  $d^+(p_k) = 0$  then all  $p_{kj} = \frac{1}{N}$ .

Otherwise, if there is no link from  $p_i$  to  $p_j$ ,  $p_{ji} = \frac{0.15}{N}$

Otherwise, if there is a link from  $p_i$  to  $p_j$ ,  $p_{ji} = \frac{0.15}{N} + \frac{0.85}{d^+(p_i)}$

To find the equilibrium value of  $R$ , we need to find  $R$  such that:

$$PR = R$$

$$\Rightarrow PR - R = 0$$

$$\Rightarrow (P - I)R = 0$$

for  $I$  is the identity matrix.

Finding the solution to this using Gaussian elimination is too slow, so we can approximate it using:

$$\lim_{t \rightarrow \infty} P^t X = R$$

Where  $X$  is any non-negative vector whose entries sum to 1.

Turns out that this tends to  $R$  quickly.

We can compute  $P^t$  quickly by splitting it up into three parts  $P = J + H + Q$  where:

Every entry of  $J$  is  $\frac{0.15}{N}$ .

Every entry of the  $i$  column of  $H$  is  $\frac{0.85}{N}$  if  $i$  has no link to  $j$  and 0 otherwise

And every entry of  $Q$  is 0 if there is no link from  $i$  to  $j$  and  $\frac{0.85}{N}$  otherwise.

Notice how  $Q$  and  $H$  are opposites.

Now:

$$PX = (J + H + Q)X = JX + HX + QX$$

Then, we can calculate  $JX$  and  $HX$  in  $O(N)$  time, since all the rows of  $JX$  are the same, and  $HX$  has many repeated rows (mostly 0's). We can compute  $QX$  in time proportional to the number of links, which is much less than the number of pages.

Adding all of these together takes  $3N$  additions.

## 6.6 The Runtime

Each iteration takes  $O(N) + O(M)$  where  $M$  is the number of hyperlinks,  $N$  is the number of pages. This is because it takes  $O(N)$  to compute  $JX + HX$  and  $O(M)$  to compute  $QX$ . There are  $40\log(N)$  iterations.

## 7 Gaussian Elimination

Gaussian elimination is an efficient algorithm for solving a system of linear equations. I'm going to assume you know how it works from MATH133, but in case you need a refresher, Khan Academy has lots of videos.

## 8 Simplex Algorithm for Linear Programming