COMP 360 Study guide

Francis Piché

February 14, 2019

Contents

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.

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 phases: *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 trasmitting 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_RC_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 concidered 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:

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

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 wont 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 nth 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 $k = 0, ..., n-1$.

Some facts:

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

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

this last one is because for each ω_n^k there is a corresponding ω_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 $nlog(n)$ 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 $< a_0, \dots, a_{n-1} >$. 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 $A(x) = xA_{odd}(x^2) + A_{even}(x^2)$ 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:

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

and

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

However, we also are now operating on x^2 not x. In order to get the runtime down from $O(n^2)$ to $O(nlog(n))$ 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 $O(nlog(n))$ 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 nxn 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 $nlog(n)$ time.

But what if there is no special structure?

5.0.1 Slow Matrix Multiplication

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

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

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,

$$c_{i,j} = a_{0,..,i/2} \cdot b_{0,...,j/2} + a_{i/2,..,m} \cdot b_{j/2,...,n}$$

$$= \sum_{k=0}^{m/2} a_{i,k}b_{k,j} + \sum_{k=m/2}^{m} a_{i,k}b_{k,j}$$

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 Strassens Algorithm

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

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

The trick is to do:
$S_1 = B_{1,2} - B_{2,2}$, $S_2 = A_{1,1} + A_{1,2}$
$S_3 = A_{2,1} + A_{2,2}$, $S_4 = B_{2,1} - B_{1,1}$
$S_5 = A_{1,1} + A_{2,2}$, $S_6 = B_{1,1} + B_{2,2}$
$S_7 = A_{1,2} - A_{2,2}$, $S_8 = B_{2,1} + B_{2,2}$
$S_9 = A_{1,1} - A_{2,1}$, $S_{10} = B_{1,1} + B_{1,2}$

Then by letting:
$P_1 = A_{1,1}S_{1,1}$, $P_2 = S_2B_{2,2}$

$P_3 = S_3 B_{1,1}$, $P_4 = A_{2,2} S_4$
$P_5 = S_5 S_6$, $P_6 = S_7 S_8$, $P_7 = S_9 S_1 0$

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

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:

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

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).

Iterpreting 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 **indegree** 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.

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

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

6.4 The Problem

This doesn't alway 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 $\sum_{p \in Pages} r(p)$. By our equation in the previous subsection, we have that the sum of all rank flowing out of pages is $\sum_{p \in Pages \wedge d^+(p)>0} r(p)$. 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 ith 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:

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

Where X is any non-negative vector who's 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 $40log(N)$ 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, I gotchu.

A system of linear equations is just a set of equations with n unknowns $\{x_1, \ldots x_n\}$, for example:

$$3x_1 - x_2 + 4x_3 = 2$$

$$17x_1 + 2x_2 + x_3 = 14$$

$$x_1 + 12x_2 - 77x_3 = 54$$

Solving this system is finding values for $x_1, \ldots x_n$ which satisfy all the equations simultaneously.

We can express these in matrix form:

$$Ax = b$$

for example:

$$\begin{pmatrix} 3 & -1 & 4 \\ 17 & 2 & 1 \\ 1 & 12 & -77 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 2 \\ 14 \\ 54 \end{pmatrix}$$

is the matrix representation of the system of equations above. Each column of the matrix A is the coefficients of the x_1 unknown, and so on. The b vector is the values on the right hand side of each equation.

There can be 0, 1 or infinitely many solutions to systems of equations. We get infinitely many if one equation is a multiple of another, none if the solutions gives something nonsensical, and one otherwise.

Normally you can solve these by isolating one variable, then plugging in that solution into the other equations.

Alternatively, multiplying by scalars, and "adding" one equation to another does not change the solution. Gaussian elimination exploits this with the matrix form to have a simple algorithm for solving linear equations.

Gaussian elimination is $O(n^3)$ since we need to isolate each unknown. Each isolation takes time $O(n^2)$, since we potentially need to add a multiple of each row of the matrix. (each row is n long and there are n rows to n^2 total operations). Then, we recurse on the substituted problem, with n levels of recursion, so $O(n^3)$ total.

Part III
Linear Programming and Applications

8 Linear Programming Formulation

Linear programming is more or less a fancy way of saying "optimization problem". Usually you'll have a set of constraints, and you're trying to maximize or minimize some value by combining amounts of various resources.

8.1 LP Formulation Example

This diet example is taken from the LP Formulation I slides.

If we need to minimize the cost of a persons food, while maintaining $\geq$ 2000 calories, $\geq$ 55g protein and $\geq$ 800mg calcium, and our options are:

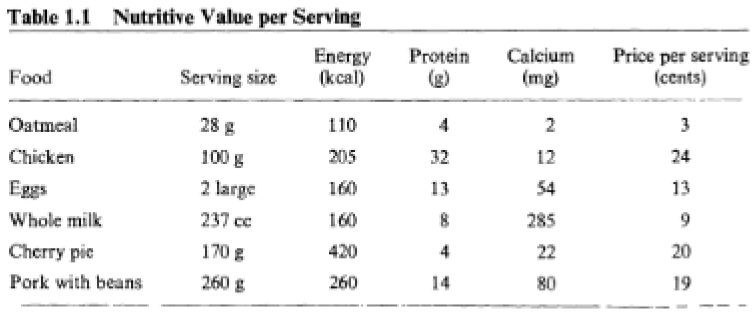


Table 1.1 Nutritive Value per Serving

Food	Serving size	Energy (kcal)	Protein (g)	Calcium (mg)	Price per serving (cents)
Oatmeal	28 g	110	4	2	3
Chicken	100 g	205	32	12	24
Eggs	2 large	160	13	54	13
Whole milk	237 cc	160	8	285	9
Cherry pie	170 g	420	4	22	20
Pork with beans	260 g	260	14	80	19

Then we can set our variables $x_1, \dots x_n$ to be (per day):

$$x_1 = oatmeal$$

$$x_2 = chicken$$

$$x_3 = eggs$$

$$x_4 = milk$$

$$x_5 = cherrypie$$

$$x_6 = pork + beans$$

Then the linear program is to minimize:

$$3x_1 + 24x_2 + 13x_3 + 9x_4 + 20x_5 + 19x_6$$

with the conditions that:

$$4x_1 + 32x_2 + 13x_3 + 8x_4 + 4x_5 + 13x_6 \geq 55$$

for the protein,

$$110x_1 + 205x_2 + 160x_3 + 160x_4 + 420x_5 + 260x_6 \geq 2000$$

for the calories and

$$2x_1 + 12x_2 + 54x_3 + 285x_4 + 22x_5 + 80x_6 \geq 800$$

for the calcium.
We must also have that

$$x_1, x_2, x_3, x_4, x_5, x_6 \geq 0$$

8.2 LP Definitions

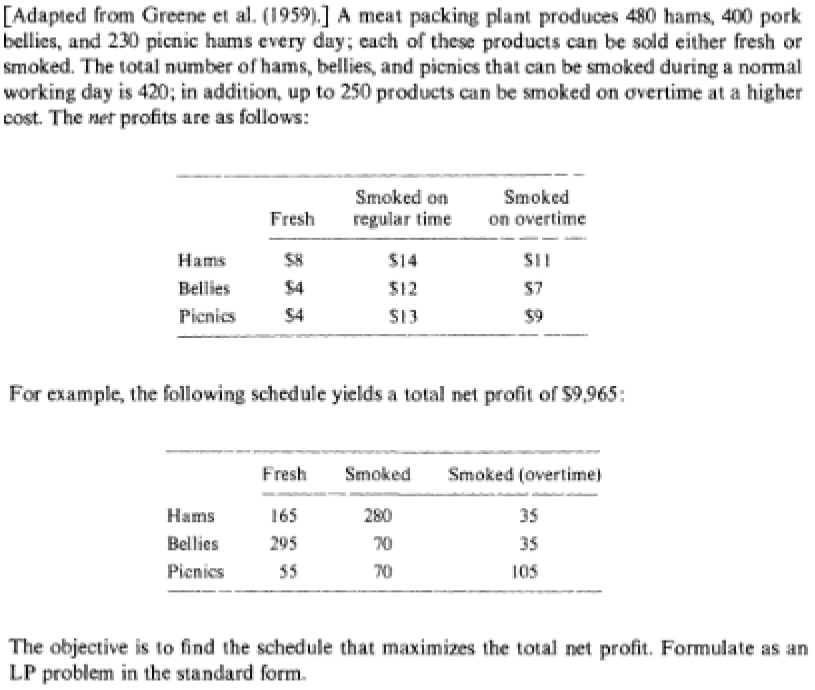
A **linear function** is a function of the form: $\sum_{i=1}^{n} c_i x_i, c_i \in \Re$
A **linear equation** is any equation of a linear function and a real number b. $f(x_1, \dots x_n) = b$
A **linear inequality** is a linear equation but with an inequality
A **linear constraint** is a linear equation or inequality which serves as a constraint to other linear equation or inequality.

A **linear program** is a problem to minimize or maximize a linear function subject to a set of linear constraints.
Standard form of a linear program is when we are maximizing a linear function where all variables are nonnegative and the set of linear constraints are of the form $f(x_1, \ldots x_n) \leq b$
A **feasible solution** to a LP is an assignment of values to the variables which satisfies all constraints.
An **Infeasible solution** tp a LP is one where there are no feasible solutions.
Unbounded maximization is when the result can be arbitrarily large.
Integer linear program (ILP) is a linear program with the added constraint that solutions are only integer assignments to the variables.

9 Examples of LP Formulation

9.1 Hams

We have the following information: (Chvatal Problem 1.6)



[Adapted from Greene et al. (1959).] A meat packing plant produces 480 hams, 400 pork bellies, and 230 picnic hams every day; each of these products can be sold either fresh or smoked. The total number of hams, bellies, and picnics that can be smoked during a normal working day is 420; in addition, up to 250 products can be smoked on overtime at a higher cost. The *net* profits are as follows:

	Fresh	Smoked on regular time	Smoked on overtime
Hams	\$8	\$14	\$11
Bellies	\$4	\$12	\$7
Picnics	\$4	\$13	\$9

For example, the following schedule yields a total net profit of \$9,965:

	Fresh	Smoked	Smoked (overtime)
Hams	165	280	35
Bellies	295	70	35
Picnics	55	70	105

The objective is to find the schedule that maximizes the total net profit. Formulate as an LP problem in the standard form.

There are 9 variables, since we have 3 options: fresh, smoked, oversmoked for each of 3 items. Call these $x_1, ..., x_9$ Each has an associated profit, so we maximize:

$$8x_1 + 14x_2 + 11x_3 + 4x_4 + 12x_5 + 7x_6 + 4x_7 + 13x_8 + 9x_9$$

with the following constraints:
We can only have 480 total hams, so

$$x_1 + x_2 + x_3 = 480$$

where x_1, x_2, x_3 are fresh, smoked and oversmoked hams.
We can only have 400 pork bellies so

$$x_4 + x_5 + x6 = 400$$

and similarly for picnic hams:

$$x_7 + x_8 + x_9 = 230$$

We can only smoke up to 420 (blaze it lol) items per day so:

$$x_2 + x_5 + x_8 \leq 420$$

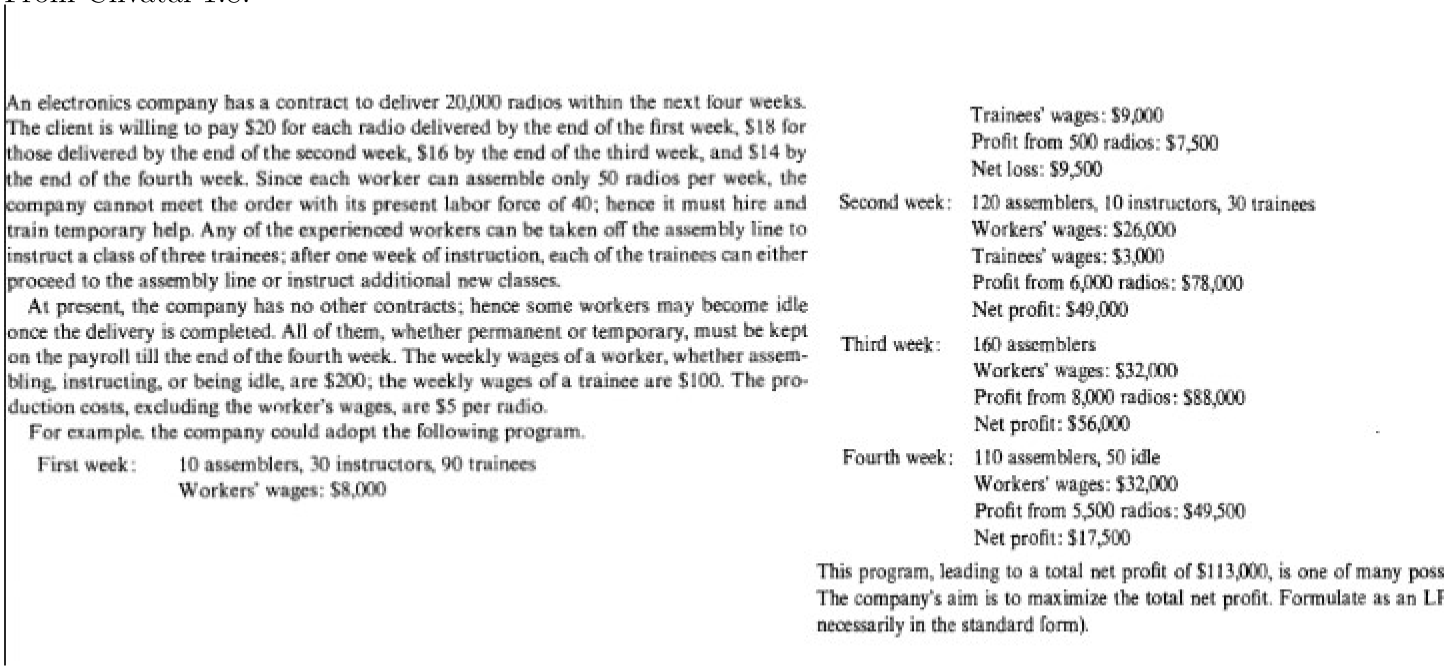
and oversmoke up to 250

$$x_3 + x_6 + x_9 \leq 250$$

Since we do not allow floating point solutions (can't have half a pork belly!) This is actually an ILP, not LP, so we have the added constraint that $x_1, ..., x_9 \in \mathbb{N}$

9.2 Radios

From Chvatal 1.8:



An electronics company has a contract to deliver 20,000 radios within the next four weeks. The client is willing to pay $20 for each radio delivered by the end of the first week, $18 for those delivered by the end of the second week, $16 by the end of the third week, and $14 by the end of the fourth week. Since each worker can assemble only 50 radios per week, the company cannot meet the order with its present labor force of 40; hence it must hire and train temporary help. Any of the experienced workers can be taken off the assembly line to instruct a class of three trainees; after one week of instruction, each of the trainees can either proceed to the assembly line or instruct additional new classes.

At present, the company has no other contracts; hence some workers may become idle once the delivery is completed. All of them, whether permanent or temporary, must be kept on the payroll till the end of the fourth week. The weekly wages of a worker, whether assembling, instructing, or being idle, are $200; the weekly wages of a trainee are $100. The production costs, excluding the worker's wages, are $5 per radio.

For example, the company could adopt the following program.

First week:	10 assemblers, 30 instructors, 90 trainees
	Workers' wages: $8,000
	Trainees' wages: $9,000
	Profit from 500 radios: $7,500
	Net loss: $9,500
Second week:	120 assemblers, 10 instructors, 30 trainees
	Workers' wages: $26,000
	Trainees' wages: $3,000
	Profit from 6,000 radios: $78,000
	Net profit: $49,000
Third week:	160 assemblers
	Workers' wages: $32,000
	Profit from 8,000 radios: $88,000
	Net profit: $56,000
Fourth week:	110 assemblers, 50 idle
	Workers' wages: $32,000
	Profit from 5,500 radios: $49,500
	Net profit: $17,500

This program, leading to a total net profit of $113,000, is one of many poss
The company's aim is to maximize the total net profit. Formulate as an LF
necessarily in the standard form).

Our variables are: r_1, r_2, r_3, r_4 for the amount the profits of each radio on a given week. w_1, w_2, w_3, w_4 for the amount of workers used in each week. t_1, t_2, t_3, t_4 for the amount of trainees trained in a given week. i_1, i_2, i_3, i_4 for the amount of instructors used on a given week, and a_1, a_2, a_3, a_4 for the amount of assemblers used on a given week.
Since each radio costs 5$ to make, and the customer will pay 20, 18, 16, and 14 for the radios depending on the week delivered. We want to maximize $15r_1 + 13r_2 + 11r_3 + 9r_4$. But we must also factor in wages, so our final problem is to maximize:

$$15r_1 + 13r_2 + 11r_3 + 9r_4 - 200w_1 - 200w_2 - 200w_3 - 200w_4 - 100t_1 - 100t_2 - 100t_3 - 100t_4$$

subject to the following constraints:
We must produce 20,000 radios, so

$$r_1 + r_2 + r_3 + r_4 = 20,000$$

and each assembler can only make 50 radios per week so:

$$r_i \leq 50a_i, i \in \{1,2,3,4\}$$

And each instructor can only teach 3 trainees, so:

$$t_j \leq 3i_j, j \in \{1,2,3,4\}$$

We also have that all workers and instructors cannot be trainees, so:

$$a_j + i_j = w_j, j \in \{1,2,3,4\}$$

Also, the amount of workers on a given week is limited by how many were trained the previous week, with an initial workforce of 40.

$$w_1 = 40$$

$$w_2 = w_1 + t_1$$

$$w_3 = w_2 + t_2$$

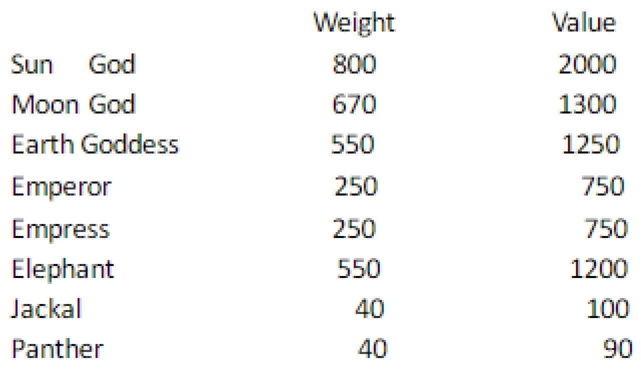
$$w_4 = w_3 + t_3$$

And since we cant have half or negative people:

$$all \in \mathbb{N}$$

9.3 Knapsack Problem

Suppose you have a backpack and can place items in it. Each item has a weight, and a value. The list of items is fixed and finite. In this case, we are trying to fill our backpack with ancient temple relics:



	Weight	Value
Sun God	800	2000
Moon God	670	1300
Earth Goddess	550	1250
Emperor	250	750
Empress	250	750
Elephant	550	1200
Jackal	40	100
Panther	40	90

and we can only hold 1000 pounds.

We need to maximize the value of each item:

$$2000x_1 + 1300x_2 + 1250x_3 + 750x_4 + 750x_5 + 1200x_6 + 100x_7 + 100x_8$$

with the constraint that the weights are less than 1000

$$800x_1 + 670x_2 + 550x_3 + 250x_4 + 250x_5 + 550x_6 + 40x_7 + 40x_8 \leq 1000$$

In general, given a list of items $i_1, ..., i_n$ where i_j has weight w_j and value v_j, with maximum weight W, we want to maximize

$$\sum_{i=1}^{n} v_i x_i$$

with the constraints:

$$\sum_{i=i}^{n} w_i x_i \leq W$$

, and only carrying at most one of each item:

$$1 \geq x_i \geq 0$$

There are variations such as the multiknapsack where we can carry multiple of each item, where we just change the last constraint to allow more than one of each x_i:

$$x_i \geq 0$$

And there is also the fractional knapsack in which we extend the ILP to an LP by allowing certain rational solutions, or even any real valued solutions for x_i. Say, we are able to take half, quarter or continuous (like infinitely many decimals) pieces of an object.

9.4 Maximum Volume s-t Flow

Recall the max flow problem from COMP 251, if you want to check out my study guide for that class I have some notes about it. I'll cover it here too though.

Let $G = (E, V)$, be a directed graph with edges E and vertices V. Let each edge $e \in E$ is given a capacity $u(e)$. G has a source vertex s which has no inward arcs and a sink vertex t which has no outward arcs. An s-t flow is a function $f : E(G) \to \mathbb{R} \geq 0$ such that for every edge e, the flow along e is at most $u(e)$, and for every node $v \neq s, v \neq t$, $flow - in = flow - out$. The volume of a flow is the total $flow - out$ on all outward arcs from s.

To express this as a LP, our variables are f_e which signify the amount of flow through each edge e.

So to get a maximum flow we need to maximize the amount flowing out of s, since no flow is "lost" along the way to t, since $flow - in = flow - out$, so we maximize the sum of flows from s to all its adjacent vertices v.

$$\sum_{e \in E | e = sv} f_e$$

With the constraint that for all edges involved:

$$\sum_{e \in E | e = uv} f_e - \sum_{e \in E | e = vw} f_e = 0$$

, for vertices u, v, and w. (Flow conservation). We must also have that

$$f_e \leq u(e), \forall e \in E$$

And that each flow is non-negative:

$$f_e \geq 0$$

9.5 Minimum Capacity s-t Cut

Again see my COMP251 guide for more on this, but:
A cut is a subset of the vertices $C \subseteq V(G) - t$ for an s-t graph G, such that $s \in C$ and $t \notin C$.
The capacity of a cut is the sum of the capacities of the edges leaving C.

We want to find the minimum capacity cut of an s-t flow.
Our variables are inC_v for each $v \in V(G)$ where inC_v is 1 if $v \in C$ and 0 otherwise. We also have x_e for each edge in $E(G)$ and is equal to 1 if e is an edge leaving C, and may be 0 or 1 otherwise.
We minimize:

$$\sum_{e \in E(G)} u(e) x_e$$

subject to:

$$inC_s = 1$$

$$inC_t = 0$$

if $e = (v, w)$ leaving C then $x_e - inC_v + inC_w \geq 0$

9.6 Minimum Weight Spanning Tree

Recall from COMP251 that a minimum spanning tree is an undirected, connected tree T, which contains all vertices $v \in V(G)$ for a connected graph G. Each edge of G has a weight, and we want to construct T (by choosing edges of G) such that the total weight is minimized.

Our variables are x_e for each edge $e \in E(G)$ where x_e is 1 if $e \in T$ and 0 otherwise.
We then minimize:

$$\sum_{e \in E(G)} w(e)x_e$$

where $w(e)$ is the weight of a given edge.
Subject to spanning all vertices of G, meaning there must be at least as many edges as number of vertices (minus one since there are $n - 1$ vertices in a tree with n edges).

$$\sum_{e \in E(G)} x_e = |V(G)| - 1$$

And also subject to ensuring that all cycles C of G,

$$\sum_{einE(C)} x_e \leq |V(C)| - 1$$

to ensure that there are no cycles.

9.7 The Traveling Salesman Problem

From wikipedia: *Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city and returns to the origin city?*

Our variables are x_{ij} for $1 \leq i \neq j \leq n$ which is 1 if the tour goes from city i to city j and 0 otherwise.
We then minimize:

$$\sum_{i=1}^{n} \sum_{j \in \{1,...i-1,i+1...n\}} d_{ij}x_{ij}$$

Subject to:

$$\sum_{j \in \{1,...i-1,i+1...n\}} x_{ij} = 1$$

and:

$$\sum_{i \in \{1,...j-1,j+1...n\}} x_{ij} = 1$$

And that all proper subsets $S \subseteq \{1, ..., n\}$,

$$\sum_{i \in S} \sum_{j \in S - \{i\}} x_{ij} \leq |S| - 1$$