COMP 360 Study guide

Francis Piché

April 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 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 $\{A, B, C, D, E\}$. With 15 A's, 7B's, 6C's, 6 D'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

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

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 shot 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:

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

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:
$P_1 = A_{1,1}S_{1,1}$, $P_2 = S_2 B_{2,2}$

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

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

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:

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

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 **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.

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

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 $\{x_1, \dots x_n\}$, 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 $x_1, \dots x_n$ 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:

$$4x_1 + 32x_2 + 13x_3 + 8x_4 + 4x_5 + 14x_6 \geq 55$$

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. Each route between cities has a cost d_{ij}.
We then minimize:

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

Subject to the "go-to" constraint: which states that for each city i, we must go to exactly one city:

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

and the come from constraint, which says that for each j, we must come from exactly one city:

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

And that all subsets $S \subset \{1, ..., n\}$, we have no cycles (similar reasoning as the MWST), since if we have a sub-cycle we might have that not every city is hit:

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

9.8 Satisfiability

Given a Boolean expression, (made up of AND, OR, NOT operators, and variables which may be TRUE or FALSE), it is satisfiable if there exists a truth assignment to the variables which makes the whole statement evaluate to true.

If we limit ourselves to only statements which are in the form:

$$\wedge_{ij}^{n}(\phi_i \vee \phi_j)$$

meaning a series of OR operators, where each ϕ can be either a variable or its negation, (aka clauses) joined by AND operators, then we can just check that each clause is true for some truth assignment to the variables. (And it turns out if you take MATH318 you'll see that any sequence of AND, OR and NOT can be written in this form so we're not limiting at all!!)

As an ILP, the inputs $X = \{x_1, ...x_n\}$ of boolean variables, $C_1, ..., C_j$ of clauses specified by $C_i^+ = \{k | x_k \in C_i\}$and $C_i^- = \{k | \neg(x_k) \in C_i\}$ (so we keep track of which clauses have NOT or regular variables.) So our variables are y_i which is 1 if x_i is true, and 0 if false.

We maximize 0 (meaning not a maximization, just finding a solution)

Subject to:

$$\forall 1 \leq i \leq j : \sum_{k \in c_i^+} y_k + \sum_{k \in c_i^-} (1 - y_k) \geq 1$$

Where every y_k is 0 or 1.

Basically this is just saying that for each clause, use y_i to show x_i, and $1 - y_i$ to show $\neg x_i$, and have that the sum of each clause is greater than one (meaning the whole clause is true) for each clause simultaneously. Just turn all the $\vee$ into $+$ and the x_i into 0's and 1's.

Might still be confusing so lets see some examples:
Given:

$$(x_1 \vee x_2) \wedge (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2)$$

We maximize 0 (just solving, not minimizing or maximizing)
Our variables, y_1, y_2.
To satisfy the first clause:

$$y_1 + y_2 \geq 1$$

and the second:

$$y_1 + (1 - y_2) \geq 1 \Rightarrow y_1 - y_2 \geq 0$$

the third:

$$(1 - y_1) + (1 - y_2) \geq 1 \Rightarrow -y_1 - y_2 \geq -1$$

and lastly:

$$y_2 + (1 - y_1) \geq 1 \Rightarrow y_2 - y_1 \geq 0$$

And y_1, y_2 are 0 or 1.

10 Bounding Solutions to LP's

We can find bounds for the solutions of variables by multiplying the constraints by some constants.

Honestly I'm really confused by this part so if anyone can help that would be really appreciated.

11 LP Standard Form

A LP is in standard form if we are maximizing the objective function subject to a set of linear constraints, a non-negativity constraint for each variable, and some constraints insisting a linear function is at most some constant.

So for n variables$x_1, ..., x_n$:

$$max(\sum_{i=1}^{n} c_i x_i)$$

subject to:
for $i = 1$ to m:

$$\sum_{j=1}^{n} a_{ij} x_j \leq b_i$$

for j=1 to n:

$$x_k \geq 0$$

If we can reduce any LP into standard form, and we can find an algorithm to solve any LP in standard form, then we can solve any LP!

12 Reducing LP's to Standard Form

Whenever we need to minimize some function, we can maximize $-z$ instead.

If x is negative, we can replace x by $(x'-x'')$ where we insist each of x' and x'' are nonnegative.

If we have a constraint $f = b$, we can replace with: $f \geq b$ and $f \leq b$.

If we have $f \geq b$, we can replace with $-f \leq b$

So as you can see, we can easily change any non-standard form linear program into a standard one using those operations.

13 Solving Standard LP's

Consider this example:
Maximize $5x_1 + 4x_2 + 3x_3$
Subject to:

$$2x_1 + 3x_2 + x_3 \leq 5$$

$$4x_1 + x_2 + 2x_3 \leq 11$$

$$3x_1 + 4x_2 + 2x_3 \leq 8$$

$$x_1, x_2, x_3 \geq 0$$

The first step is to introduce **slack variables**.

Between the left hand side of the first constraint and the right hand side, there may be "slack room". That is, $5 - 2x_1 - 3x_2 - x_3$ might be non-zero, and would be positive. Lets call it $x_4 = 5 - 2x_1 - 3x_2 - x_3$. we kow $x_4 \geq 0$ since the original constraint is a $\leq$. We doe this for each choice of $x_i's$ in the conditions:

$$x_5 = 11 - 4x_1 - x_2 - 2x_3$$

$$x_6 = 8 - 3x_1 - 4x_2 - 2x_3$$

We can then restate the problem as:
Maximize $5x_1 + 4x_2 + 3x_3$ subject to:

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

since this is equivalent to the $\leq$ relationships before.

The original variables are referred to as the **decision variables**.

Note that a feasible (and optimal) solutions of the decision variables can be extended to feasible (or optimal) solutions with the slack variables, and visa versa.

The idea with the simplex method is to repeatedly improve our solutions by finding feasible solutions to the problem subject to the constraint that the solution is larger than the previous.

To find the initial feasible solution, we can set all decision variables to 0, and the slack variables represent a feasible solution. For example, if we set the decision variables in the previous example to 0, then $x_4 = 5$, $x_5 = 11$, $x_6 = 8$, which leaves 0 as our result.

Once we've found this, we add to x_1 until we found the best solution it can offer.

But how much can we increase it? We need to ensure that none of $x_4, x_5 and x_6$ drop below 0, so we look at the equation for each and find a bound on x_1 (remember that we are leaving both x_2 and x_3 at 0.).

$$\begin{aligned} x_4 &= 5 - 2x_1 - 3x_2 - x_3 \\ &= 5 - 2x_1 - 0 - 0 \end{aligned}$$

so we have $5 - 2x_1 \geq 0$, which means $x_1 \leq 5/2$. We do the same with the others and find $x_1 \leq 11/4, and 8/3$. Of these, $5/2$ is the lowest so this is the bound we use. If we use this, then we see that $x_5 = 1$ and $x_6 = 1/2$.

To proceed, we put the newly found value to the left hand side, and express it in terms of x_2, x_3, x_4.

$$x_1 = 5/2 - 3x_2/2 - x_3/2 - x_4/2$$

Next we substitute this into the other equations for x_5, x_6:

$$\begin{aligned} x_5 &= 11 - 4(5/2 - 3x_2/2 - x_3/2 - x_4/2) - x_2 - 2x_3 \\ &= 1 + 5x_2 + 2x_4 \\ x_6 &= 8 - 3(5/23x_2/2 - x_3/2 - x_4/2) - 4x_2 - 2x_3 \\ &= 1/2 + x_2/2 - x_3/2 + 3x_4/2 \end{aligned}$$

our new solution is:

$$z = 25/2 - 7x_2/2 + x_3/2 - 5x_4/2$$

And we repeat. We can't raise x_2 pr x_4 since they are negative, so we have to raise x_3. We use the previous system of equations to find bounds for x_3 the same way we did before.

14 Simplex Method Terminology

There's a few terms that get tossed around when talking about the simplex method and it can be a little confusing.

14.1 Dictionaries

A dictionary is a special system of equations that shows different phases in the Simplex method.

It involves variables $x_1, ..., x_{n+m}$ where m is the number of slack variables, and n is the initial number of variables. It also involves an equation z which represents the solution to the objective function.

We must have that every set of equations comprising a dictionary must also be a solution of the original problem. (Including slack variables) That is:

$$x_{n+i} = b_i - \sum_{j=i}^{n} a_{ij} x_j$$

$$z = \sum_{j=1}^{n} c_j x_i$$

for all $i = 1...m$.

Also, the equations of any dictionary must express m of the $n + m$ variables and z in terms of the remaining n variables.

. The above two conditions are the the defining characteristics of a dictionary.

If, when setting the right hand size variables to zero and evaluating the left hand side variables we arrive at a feasible solution, we call the dictionary feasible. Every feasible dictionary describes a feasible solution, but not every feasible solution has a dictionary. Solutions which have a dictionary are called basic.

14.2 Basic, Non-Basic, Pivot

As mentioned above, the variables on the left hand side of a dictionary are called **basic**. The variables that appear on the right hand side are called **non-basic**. Together the basic variables are called a **basis**.

At each iteration of the method, the basis changes, with nonbasic variables becoming basic, and variables leaving.

We choose entering variables with the goal of increasing the objective function z, and choose leaving variables with the goal of keeping all values non-negative.

The **pivot row** is the row in which the leaving variable appears in the dictionary.

15 The General Simplex Method

To generalize the example we just saw:

Step 1: Introduce slack variables $x_{n+1}, ..., x_{n+m}$ and denote the object function by z. That is:

$$x_{n+i} = b_i - \sum_{j=1}^{n} a_{ij} x_j$$

As your starting feasible dictionary.

Step 2: If there is no variable with positive coefficient in z for the current feasible dictionary then return. The solution is optimal.

Step 3: Choose one of the variables x_e with a positive coefficient.

Step 4: See if there are any variables which give an upper bound on the possible values of your chosen variable, if we increase it by leaving all other variables at 0. If not, then the problem is unbounded, and we stop.

Step 5: Let x_l be one of the variables who gives the lowest upper bound on x_e.

Step 6: Write the equation for x_l as an equation for x_e.

Step 7: Replace x_e by the RHS of the equation in step 6 in all the other equations of the dictionary to obtain a new feasible dictionary. Return to step 2.

16 Ensuring Termination

There are a few ways the simplex algorithm can fail.

- We might not be able to get the initial dictionary
- Can we always choose and entering, leaving and construct the next feasible dictionary?
- Does the simplex method always terminate?

It's worth noting that there are cases where the simplex method yields several degenerate iterations in which the solution does not improve. This is common and always results in a "breakthrough" to the optimal solution.

Here we'll focus on the last one.

It's possible that the simplex method enter a cycle and never terminate.

16.1 An Example

Consider:

$$10x_1 - 57x_2 - 9x_3 - 24x_4$$

subject to:

$$x_1 - 11x_2 - 5x_3 + 18x_4 \leq 0$$

$$x_1 - 3x_2 - x_3 + 2x_4 \leq 0$$

$$x_1 \leq 1$$

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

Then a feasible slack dictionary is:

$$-x_1 + 11x_2 + 5x_3 - 18x_4 = x_5$$

$$-x_1 + 3x_2 + x_3 - 2x_4 = x_6$$

$$1 - x_1 = x_7$$

$$z = 10x_1 - 57x_2 - 9x_3 - 24x_4$$

Not gonna write out the 6 iterations of the loop but here it is copy pasted from Cvatal:

$$
\begin{aligned}
x_1 &= 11x_2 + 5x_3 - 18x_4 - 2x_5 \\
x_6 &= -\ 4x_2 - 2x_3 + 8x_4 + x_5 \\
x_7 &= 1 - 11x_2 - 5x_3 + 18x_4 + 2x_5 \\
\hline
z &= 53x_2 + 41x_3 - 204x_4 - 20x_5.
\end{aligned}
$$

After the second iteration:

$$
\begin{aligned}
x_2 &= -\ 0.5x_3 + 2x_4 + 0.25x_5 - 0.25x_6 \\
x_1 &= -\ 0.5x_3 + 4x_4 + 0.75x_5 - 2.75x_6 \\
x_7 &= 1 + 0.5x_3 - 4x_4 - 0.75x_5 - 13.25x_6 \\
\hline
z &= 14.5x_3 - 98x_4 - 6.75x_5 - 13.25x_6.
\end{aligned}
$$

After the third iteration:

$$
\begin{aligned}
x_3 &= 8x_4 + 1.5x_5 - 5.5x_6 - 2x_1 \\
x_2 &= -\ 2x_4 - 0.5x_5 + 2.5x_6 + x_1 \\
x_7 &= 1 - x_1 \\
\hline
z &= 18x_4 + 15x_5 - 93x_6 - 29x_1.
\end{aligned}
$$

After the fourth iteration:

$$
\begin{aligned}
x_4 &= -\ 0.25x_5 + 1.25x_6 + 0.5x_1 - 0.5x_2 \\
x_3 &= -\ 0.5x_5 + 4.5x_6 + 2x_1 - 4x_2 \\
x_7 &= 1 - x_1 \\
\hline
z &= 10.5x_5 - 70.5x_6 - 20x_1 - 9x_2.
\end{aligned}
$$

After the fifth iteration:

$$\begin{aligned} x_5 &= 9x_6 + 4x_1 - 8x_2 - 2x_3 \\ x_4 &= -x_6 - 0.5x_1 + 1.5x_2 + 0.5x_3 \\ x_7 &= 1 - x_1 \\ \hline z &= 24x_6 + 22x_1 - 93x_2 - 21x_3. \end{aligned}$$

After the sixth iteration:

$$\begin{aligned} x_6 &= -0.5x_1 + 1.5x_2 + 0.5x_3 - x_4 \\ x_5 &= -0.5x_1 + 5.5x_2 + 2.5x_3 - 9x_4 \\ x_7 &= 1 - x_1 \\ \hline z &= 10x_1 - 57x_2 - 9x_3 - 24x_4. \end{aligned}$$

These iterations were attained by always choosing the entering variable that has the largest coefficient in the z-row of the dictionary, and if two or more variables are the same, then the one with the smallest subscript is taken.

Notice we're now back to the beginning.

Cycling is the only reason, as given by this theorem:

Theorem 3.1: If the simplex method fails to terminate, then it must cycle.

So if we can avoid cycling, we can ensure termination.

Note that when choosing a leaving variable, we must not choose one which the most stringent upper bound on the increase of the entering variable. If there are two or more variables that give the same upper bound, then there is a choice that needs to be made.

To break the choice, we always use the leaving variable with the smallest subscript.

17 Finding a Feasible Initial Dictionary

Suppose we have the LP:

$$(Max 5x_1 - 32x_2)$$

Subject to:

$$-x_1 \leq -5$$

$$-7x_1 - 12x_2 \leq 20$$

$$x_1 - x_2 \leq 1$$

$$x_1, x_2 \geq 0$$

Since we have negative values on the right hand side of some constraints, we can't build a feasible dictionary the usual way using slack variables. This is because each equation for x_i is of the form:

$$x_{n+i} = b_i - \sum_{j=1}^{n} a_{ij}x_j$$

So clearly if any of the b_i are negative, then we break the non-negativity condition on x_{n+i}. Equivalently, we can say that if all $x_i = 0$ is a feasible solution, then all b_i must be greater than 0.

We must use an **auxiliary LP**. We construct this by adding a new variable, x_0 with the constraint $x_0 \geq 0$, coefficient -1 and maximize $-x_0$.

This LP has an optimal solution of $x_0 = 0$ if and only if there is a feasible solution to the original. To see this, suppose $x_0 \neq 0$. Then

$$\sum_{j=1}^{n} a_{ij}x_j - x_0 \leq b_i$$

Would not have the origin (all $x_i = 0$) as a feasible solution, which by the argument before, means that some $b_i \leq 0$ which means there is no feasible solution!

So suppose we obtained a non-negative solution to the auxiliary problem. Then, at some point we must have had x_0 as non-zero in the basis until the auxiliary objective function went to zero (since we are maximizing -x_0). By removing x_0 from the basis at this point (and just deleting it from the dictionary). So we just copy down the dictionary without any terms that include x_0. Then, to obtain z we just express the original objective function in terms of the non-basic variables. This is now a feasible solution of the original problem, and we can continue to use the simplex method as before.

18 Upper Bounding The Optimal Solution of an LP

The general approach here is to sum multiples of the constraints. For example, in the napsack problem, we chose the highest value/weight ratio. For example:
Maximize:

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

Subject to:

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

We would choose to take $4x_4$ with a value of $4 * 750 = 3000$, since it has the highest ratio. To prove this is optimal, suppose we multiply the constraint by the value/weight ratio (3 in this case).

$$2400x_1 + 2010x_2 + 1650x_3 + 750x_4 + 750x_5 + 1650x_6 + 120x_7 + 120x_8 \leq 3000$$

By our non-negativity constraints, each $c_i x_i$ in the new (multiplied) constraint must be greater than or equal to the corresponding $c_i x_i$ in the objective function. That is $2000x_1 \leq 2000x_1$, $1300x_2 \leq 2010x_2$, $1250x_3 \leq 1650x_3$ etc. All together we get that the objective function is bounded above by the multiplied constraint. So in this case, 3000 is an upper bound on the objective function, and we see that we have an optimal solution.

18.1 A Systematic Approach

To know that we're choosing the best multipliers, theres a trick we can use.

Suppose we have:
Maximize:

$$-x_1 - 2x_2$$

subject to:

$$-3x_1 + x_2 \leq -1$$

$$x_1 - x_2 \leq 1$$

$$-2x_1 + 7x_2 \leq 6$$

$$9x_1 - 4x_2 \leq 6$$

$$-5x_1 + 2x_2 \leq -3$$

$$9x_1 - 4x_2 \leq 6$$

$$x_1, x_2 \geq 0$$

If we take the right hand side of each constraint as a coefficient to some new variables y_i, then we can come up with a new LP:
Minimize:

$$-y_1 + y_2 + 6y_3 + 6y_4 - 3y_5 + 6y_6$$

subject to:

$$-3y_1 + y_2 - 2y_3 + 9y_4 - 5y_5 + 9y_6 \geq -1$$

$$y_1 - y_2 + 7y_3 - 4y_4 + 2y_5 - 4y_6 \geq -2$$

Where the coefficients of the first equation come from the coefficients of the x_1's in the constraints, and the second's come from the coefficients of the x_2's in the constraints. Then, as we'll see in the next section, solving this new LP will give the best multipliers for an upper bound on the original.

19 Duality Theorem

The **dual** of an LP is exactly as shown in the systematic approach to finding the best multipliers. We take the original problem (in standard form) and take the right hand side of the constraints as coefficients in the objective function. The objective function is now to minimize. Thus, there is as many variables in the dual as there are constraints in the original. There are as many constraints in the dual as there are variables in the original, since the coefficients of y_i in the dual correspond to the coefficients of all the x_j in constraint i in the original.

Note that the dual of the dual is the original.

The duality theorem for LPs in standard form states that if either of the dual or the primal have an optimal solution then they both have a solution, and the solution values are equal.

The proof is ????????

The solution to the dual can be read off the equation for z in the optimal dictionary in the original, where $y_i = -c^D_{n+i}$ where $c^D_{n+i} = 0$ if x_{n+i} is in the basis for the dictionary D.

20 Algorithmic Game Theory

20.1 Rock Paper Scissors

We can represent rock paper scissors as a "payoff" matrix. Where each row is a choice from 1 player, and each column is a choice from another. Then, each entry would be the score that the target player would receive from that combination.

$$\begin{Bmatrix} - & R & P & S \\ R & 0 & -1 & 1 \\ P & 1 & 0 & -1 \\ S & -1 & 1 & 0 \end{Bmatrix}$$

If we are interested in the player corresponding to the rows.

It seems that neither player has much of an advantage since the payoff matrix is skew-symmetric. ie:

$$P_{ij} = -P_{ji}$$

ie:

$$P = -P^T$$

So no matter what you choose, your expected winnings are 0.

20.2 Morra

In Morra, each player hides one or two gold dubloons and guesses how many gold dubloons the other player is hiding. If only one player guesses correctly, then they get all the loot. If both or neither guess correct, nothing happens.

The matrix is then:

$$\begin{Bmatrix} - & [1,1] & [1,2] & [2,1] & [2,2] \\ [1,1] & 0 & 2 & -3 & 0 \\ [1,2] & -2 & 0 & 0 & 3 \\ [2,1] & 3 & 0 & 0 & -4 \\ [2,2] & 0 & -3 & 4 & 0 \end{Bmatrix}$$

Where each pair $[x, y]$ is x =number hiding and y = number guessed. We get value above 2 or below -2 since we lose or gain both what we have, and what we the other has.

Again here there is no best strategy since the matrix is skew-symettric.

Suppose we're interested in the row-player. Can we find a lower bound on the expected winnings?
Maximize z
Subject to:

$$x_1 + x_2 + x_3 + x_4 = 1$$

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

$$-2x_2 + 3x_3 - z \geq 0$$

$$2x_1 - 3x_4 - z \geq 0$$

$$-3x_1 + 4x_4 - z \geq 0$$

$$3x_2 - 4x_3 - z \geq 0$$

Where the x_i are the probability of the other player picking column i. So, the probabilities must sum to 1 (first constraint). And each row in the matrix corresponds to a constraint. For example the [1,1] row: if you only ever pick [1,1], and the other person picks [1,1] with probability x_1, [1,2] with probability x_2, [2, 1] with prob x_3 and [2, 2] with prob x_3, then the expected value z is $0x_0 + 2x_1 - 3x_2 + 0x_3$. But we're only interested in the upper bound, so:

$$2x_1 - 3x_2 \leq z$$

. Moving things around we get:

$$-2x_1 + 3x_2 - z \geq 0$$

. We can do a similar thing, finding a lower bound on the value for the column player.

Solving each of these we get 0. That is, Morra is a **zero sum game**.

Note that the column and row player LPs are duals of each other, and by the theorem before, have the same optimal solution.

Part IV
P, NP, and NP Completeness

When classifying problems, up to now we always called their solutions efficient if it could un in polynomial time. Now we are looking for a broader classification to include problems which either have no efficient algorithm, or we cannot prove that there is no efficient algorithm and one has not yet been found.

There is a class of problems that have been proved to be equivalent in that a polynomial time algorithm for one of them would be a solution to all of them, should it exist. These are called NP-Complete problems.

I like the way it was put in Klienberg & Tardos, page 453: "From a pragmatic point of view, NP-completeness essentially means "*computationally hard for all practical purposes, though we can't prove it.*""

21 Definitions

When I say fast, or quickly I mean polynomial time. When I say decision problem, I mean a problem that has a yes or no answer. Many problems have a solving version and decision version. For example:
Does this graph have a spanning tree of weight at most k vs *Find the minimum spanning tree of this graph.*

Solving a problem vs verifying the solution are also not the same thing. Someone tells you that the answer is yes and you check, vs you finding the answer to be yes.

21.1 NP

This is the set of decision problems that can be verified quickly. (For the yes answer). Ie, there exists a fast certificate to verify the answer.

21.2 P

P is the set of decision problems that can be **solved** quickly. Since the problem can be solved quickly, we can verify a solution quickly (by just solving it ourselves using the efficient algorithm and seeing if we get the same result), therefore $P \subseteq NP$

21.3 NP-Complete

An NP problem is NP-complete $\Leftrightarrow$ every other NP problem can be reduced to the same problem. That is why a solution for any NP-complete problem is a solution for any NP problem.

21.4 Complementary Problems

The co-problem for a decision problem is the one that gives the opposite answer for the same input. So for example "is 7 prime" vs "is 7 not prime" would give opposite answers for the same input.

So, a problem is co-NP if we can verify every "no" answer in polynomial time.

22 Some theorems and conjectures

Theorem: $P \subseteq (NP \cap co - NP)$
Note that $NP \cap co - NP$ is non-empty.
Conjecture: $P = (NP \cap co - NP)$
Million dollar problem: $P = NP$

Observation: $P = NP \Leftrightarrow P = co - NP$
Proof:
$P = co - P$, since we are able to solve the problems efficiently. If $P = NP$, then necessarily $co - NP = co - P$. And since $P = co - P$, we have that $P = co - NP$.

23 Reductions

We can show that a problem is easy or (likely) hard by reducing the problem to another problem with the same answer.

23.1 Karp Reductions

A Karp reduction from problem π to π' is a fast algorithm which given an instance of π returns an instance of π' with the same answer.

If a Karp reduction exists for π, to π', then $\pi \leq_p \pi'$

For example, if a graph is k colorable, then we can surely turn it into a $k+1$ coloring by adding a vertex with an edge to every vertex of G. Then we need $k+1$ colors. So $k - colorability \leq_p (k+1) - colorability$

The point of this is that if there is a polynomial time algorithm for π' then there is one for π (if $\pi \leq_p \pi'$). So, conversely if there is no fast algorithm for π, then there is none for π'.

We can now restate our definition for NP-completeness by saying that a problem π is NP-complete if it is in NP and for every other problem π' in NP, $\pi' \leq_p \pi$, ie: every other problem can be reduced to π. So, if there were to exist an NP-complete problem in P, then we could show that every problem in NP is also in P, thus proving $P = NP$.

23.2 Some NP-complete problems

It has been shown (outside the scope of this course) that SAT is NP-complete. We also know that for LP's in standard form, we can verify than it has a feasible solution with value $\geq k$ efficiently. So, decision ILP's are in NP. From previous section, we saw how to formulate SAT as an ILP, therefore SAT $\leq_p$ ILP, and so ILP is NP-complete.

23.3 Reducing SAT to 3-SAT

3-SAT is a SAT problem in which every clause has exactly 3 distinct elements. 3-SAT is in NP, since we can use any satisfying truth assignment as a certificate of validity (which can be found using ILP).

Let $C_1, ..., C_m$ be a set of clauses, over a set of variables $X = \{x_1, ..., x_n\}$.

If A, B are clauses over X, and $y \notin X$, then we can extend a truth assignment of X to $X \cup y$ such that it satisfies $(A \vee y) \wedge (B \vee \neg(y)) \Leftrightarrow$ The original truth assignment satisfies one of A or B.

To see this, suppose y is false, then A must be true, and B can be true or false. Then if y is true, A can be true or false, and B must be true.

Now, we will replace all clauses C_j by clauses consisting of X_j new variables which appear only in the clauses which will replace C_j.

If C_j is longer than 3, it has the form:

$$z_j^1, .., z_j^{l_j}$$

where l_j is the length of clause j. We will replace it by:

$$(z_j^1 \vee z_j^2 \vee y_j^1) \wedge (\neg y_j^i \vee z_j^{i+2} \vee y_j^{i+1}) \wedge (\neg y_j^{l_j-3} \vee z_j^{l_j-1} \vee z_j^{l_j})$$

for $i \in \{1, .., l_j - 3\}$. This works by the note above, since $y_i \in X_j$ are explicitly not seen in C_j. Also notice that we have created groupings of 3, (clauses of length 3). So if we were able to satisfy the original, we can equivalently satisfy this one, and visa versa.

For clauses shorter than 3, we can replace them as follows:
Replace: $z_j^1 \vee z_j^2$ by:

$$(z_j^1 \vee z_j^2 \vee y_j^1) \wedge (z_j^1 \vee z_j^2 \vee \neg(y_j^1))$$

Replace: z_j^1 by the conjunction (AND) of:

$$(z_j^1 \vee y_j^1 \vee y_j^2)$$

$$(z_j^1 \vee y_j^1 \vee \neg y_j^2)$$

$$(z_j^1 \vee \neg z_j^1 \vee y_j^2)$$

$$(z_j^1 \vee \neg y_j^1 \vee \neg y_j^2)$$

Which is equivalent to the original.

Finally if the length is 3 we need not replace C_j.

This procedure gives us a set X' of variables of size $n' = n + \sum_{j=1}^{m} n_j$ variables, where n is the original number of variables, m is the number of clauses, and $n_j = |l_j - 3|$. Then the set of clauses has $m' = \sum_{j=1}^{m} a_j$ clauses, where $a_j = 4$ if $l_j = 1$, $a_j = 2$ if $l_j = 2$ and $a_j = l_j - 2$ otherwise.

These give the same answer since any satisfying instance of SAT, keeping the same truth assignment for X and setting y_j^i to be true if none of $z_j^1, ..., z_j^{i+1}$ are true (and false otherwise) gives a satisfying instance of 3-SAT. Also, any non-satisfying instance of SAT can be a non-satisfying instance of 3-SAT by selecting y_j to be true if it is negated, and false otherwise.

23.4 Transitivity of Reductions

If π' can be efficiently reduced to π, and π'' can be efficiently reduced to π', then π'' can be reduced to π.

We can use this to say that if $\pi \in NP$, and π' is NP-complete, and π' reduces to π, then NP-complete. This is because if π' is NP-complete, then for any other problem π'' in NP, π'' reduces to π', but we said π' is NP-complete, therefore π reduces to π' and so is NP-complete.

24 More NP-Complete Problems

24.1 Independent Set

An independent (stable) set is a set of vertices in which none of them share an edge. Note that a graph is k-colourable $\Leftrightarrow$ we can partition its vertices into k independent sets. (Since a k-colouring is a way of coloring the vertices such that no two vertices sharing an edge have the same color).

Instances of Independent Set involve determining if G has an independent set of size k.

24.1.1 Proof

First, Independent Set $\in$ NP since we can check that there are k distinct vertices with no edges joining them in $O(n+m)$ time. (By simply looping through them).

The approach is to reduce 3-SAT to Independent Set in polynomial time.

Suppose we have m clauses of length 3. Let G be a graph with $n = 3m$ vertices, each of which map to a variable in the 3-SAT input. For each vertex in a clause, create an edge between all three. Next, join each variable x to its negation $\neg x$.

This relies on the alternative interpretation of satisfiability. Instead of finding an assignment of boolean values to all variables such that each clause has at least one satisfied literal, notice that we can choose one literal from each clause, such that the set of chosen literals forms a consistent set. Consistent meaning that it does not contain a literal x and its negation $\neg x$. So from $(x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee \neg x_3$, we could pick x_1 from the first clause, x_2 from the second but not x_1 and $\neg x_1$.

By adding edges between vertices of a clause, we force the independent set to contain only one of the variables in the clause. Similarly, by forming an edge between each variable and its negation, we enforce the condition that the choices are consistent. Therefore the resulting choice of independent set will satisfy the SAT input formula.

24.2 Clique

A clique is a set of vertices in a graph such that every two distinct vertices in the clique share an edge. (The subgraph is complete).

An instance of the clique problem consists of determining if there exists a clique of size k.

We claim that clique is NP-Complete.

24.2.1 Proof

We show that Clique is NP-Complete by reducing it to Independent Set in polynomial time.

First, Clique is NP since we can check that there are is a connected subgraph of size k in $O(n+m)$ time.

Suppose (G,k) is an instance of Independent Set. Take all $V(G)$, and form an edge $e=(v_i,v_j)$ in the graph of the clique instance $\bar{G} \Leftrightarrow e \notin E(G)$. Any independent set in G will therefore be a clique in $\bar{G}$.

24.3 Vertex Cover

A vertex cover of a graph G is a set S of vertices such that every edge $e \in E(G)$ has at least one vertex in S.

An instance of Vertex Cover involves verifying a vertex cover of size k in G.

Vertex Cover is NP-Complete

24.3.1 Proof

Vertex Cover is NP since given a set of vertices we can check that at they cover every edge in $O(n+m)$.

We will show that it is NP-complete by reducing Independent Set to Vertex Cover.

We rely on the fact that for a graph $G=(V,E)$, $S \subseteq V$ is independent $\Leftrightarrow V-S$ is a vertex cover. To see this, suppose S is an independent set. Let $e=(u,v)$ be an edge in G. Since S is independent, at most one of u,v are in S. Thus the other must be in $V-S$. Then, all edges must have at least one end in $V-S$ and thus $V-S$ is a vertex cover. If $|S|=k$ then $|V-S|=n-k$ and can clearly be reduced in polynomial time.

24.4 3-Coloring

A 3-coloring is a graph coloring in which we use only 3 colors.

An instance of 3-coloring involves determining if a graph can be colored with 3 colors.

24.4.1 Proof

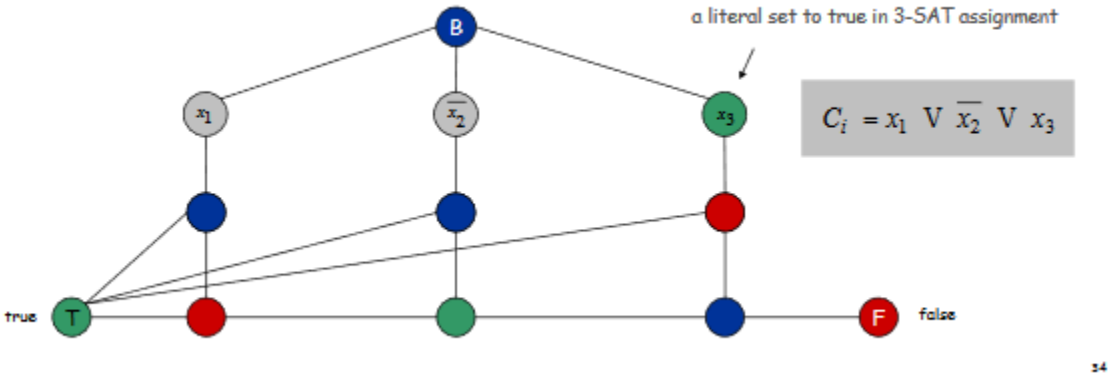
3 Coloring is NP since we can verify a 3 coloring in $O(n+m)$ time.

To prove that it is NP-complete we reduce from 3-SAT to 3-Coloring.

The way that it works is as follows: we create a vertex for each literal in the 3-SAT input. Then, we create 3 new vertices, T, F, B and connect them in a triangle. Then we connect every literal to B. Next we connect every literal to its negation. This will form a triangle between x, $\neg x$ and B for all x in the literals.

Already, we see that x and $\neg x$ must have different colors, and that T, F, B have different colors.

Now, we must also have that any clause has at least one literal evaluating to true, so we then add a six-vertex "gadget" for each clause. A gadget is a set of redundant vertices that allow us to retain this property. We must design the gadget in such a way that a 3 coloring is only possible if one of the literals in the clause has the same color as T.



24.5 Subset Sum

The subset sum problem is a special case of the knapsack problem. Given $w_1, ..., w_n$ and a target W is there a subset which sums to W.

Subset sum is NP-complete.

24.5.1 Proof

Subset sum is in NP since we can easily get a series of true false values that will tell us whether to include w_i in the subset. Then we can just sum all of them to see if we have the target W.

First notice that, due to the way base-10 numbers are set up, asking:

Can we find a subset of the following set which sums to 7035?

6018, 4032,1028,2020,3015,1003

Can we find a subset of the following set the first element which sum to 7 and the second elements of which sum to 35?

(6,18),(4,32),(1,28),(20,20),(3,1),(1,3)

Are two ways of asking the same question.

So given a 3-SAT instance with n variables and k clauses, we can form some new numbers by using the following table:

$C_1 = \bar{x} \vee y \vee z$
$C_2 = x \vee \bar{y} \vee z$
$C_3 = \bar{x} \vee \bar{y} \vee \bar{z}$

	x	y	z	C_1	C_2	C_3	
x	1	0	0	0	1	0	100,010
¬ x	1	0	0	1	0	1	100,101
y	0	1	0	1	0	0	10,100
¬ y	0	1	0	0	1	1	10,011
z	0	0	1	1	1	0	1,110
¬ z	0	0	1	0	0	1	1,001
dummies to get clause columns to sum to 4	0	0	0	1	0	0	100
	0	0	0	2	0	0	200
	0	0	0	0	1	0	10
	0	0	0	0	2	0	20
	0	0	0	0	0	1	1
	0	0	0	0	0	2	2
Target	1	1	1	4	4	4	111,444

As you can see, these new numbers will have $n+k$ digits, and we will have $2n+2k$ numbers.

The upper left quadrant contains the truth assignments for the variables. The bottom left is always just 0's. The top right is 1 if the clause can be satisfied by the corresponding truth assignment and 0 otherwise. The bottom right is simply padding so that the sum of (one of the true variables) + the padding = 4.

We can clearly see from the table that we have a new subset sum, and it corresponds to a satisfying truth assignment

24.6 Partition

Given k integers, $a_1, ..., a_k$, can we partition them into two sets such that the subset sum of each is equal?

Partition is NP-Complete

24.6.1 Proof

First, it is NP by the same logic as the subset sum problem. Next, we will show that it is NP-complete by reducing subset sum to it.

Given an instance of subset sum: $s_1, ..., s_l$, let $k = l + 1$, and $a_i = s_i$ for $i = 1, ..., l$.

There are two cases. First, if twice the target in the subset sum instance is $\geq \sum_{i=1}^{k-1} a_i$, then set the last a_k to be

$$2W - \sum_{i=1}^{k-1} a_i$$

Otherwise, we set it to be

$$\sum_{i=1}^{k-1} a_i - 2W$$

For case 1, if the subset sum is a yes-instance, then we can now just set S_1 in the partition problem to be S in the subset sum problem, where S_1 does not contain a_k. Then S_2 will be $\{s_1, .., s_l\} - S \cup \{a_k\}$ which by construction has the same sum as S_2.

In the second case, we let S_1 be $S \cup \{a_k\}$, and the reverse logic holds.

The idea here is that the assignment of a_k "compensates" for the difference between the sum of ALL values in the instance of subset sum, and twice the target value.

24.7 Decision-Knapsack

Given $w_1, ..., w_n$, $v_1, ..., v_n$, target value V and max weight W, is there a subset Z of $\{1, .., n\}$ such that the sum of $w_i \leq W$ and the sum of $v_i \geq V$.

This problem is NP-complete.

24.7.1 Proof

First, it is in NP since we can again easily verify.

We can reduce the partition problem to the decision knapsack problem as follows.

Let V=W=$\frac{\sum_{i=1}^{k} a_i}{2}$ where $k = n$, $v_i = w_i = a_i$. Basically just make the problem super easy by making the two sums equal and stealing the values directly from the partition problem.

24.8 Directed Hamiltonian Cycle

Given a directed graph $G = (V, E)$, a Hamiltonian cycle is a cycle in G that visits each vertex exactly once. An instance of DHC (directed Hamiltonian cycle) determines if a directed graph contains a Hamiltonian cycle.

DHP is NP-complete.

24.8.1 Proof

The reduction will be from 3-SAT. $3 - SAT \leq_p DHC$.

First, DHC is in NP. This is because, given a cycle C as an ordered list of vertices we can verify that it contains each vertex exactly once in $O(n)$ time.

Let $x_1, ..., x_n$, $C_1, ..., C_m$ be an instance of 3-SAT. To start, let G be a graph with 2^n vertices, representing all possible truth assignments to the variables. Make n paths like this:

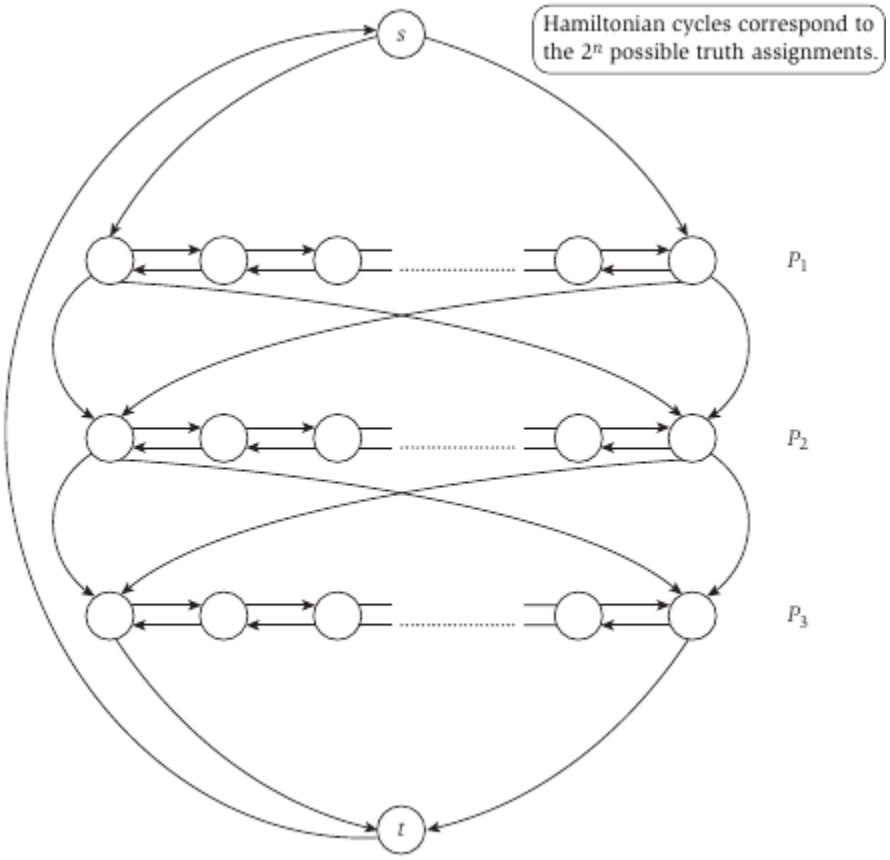


Figure 8.7 The reduction from 3-SAT to Hamiltonian Cycle: part 1.

That is, there are paths that go left to right on each level P_i, and edges connecting the levels. Add two nodes s and t.

Let $P_i = v_i1, v_i2, ..., v_ib$ be one of these paths.

All the Hamiltonian cycles in this graph must use the edge (t, s) since we need to return back to the start, and it is the only edge from t to s. At each level, we have a choice of going left along p_i or right. After traversing, we again have a choice of going left or right on the next path.

So if we traverse from left to right, we say that x_i is true, and false otherwise.

But wait! Theres more! So far we can make a Hamiltonian cycle for any 3-SAT true instance, but we need to be able to go the other way. We want to have that 3-SAT will be true if there is a hamiltonian cycle.

To do this, add a node for each clause in the 3-SAT instance. This will enforce the choice of direction of the paths.

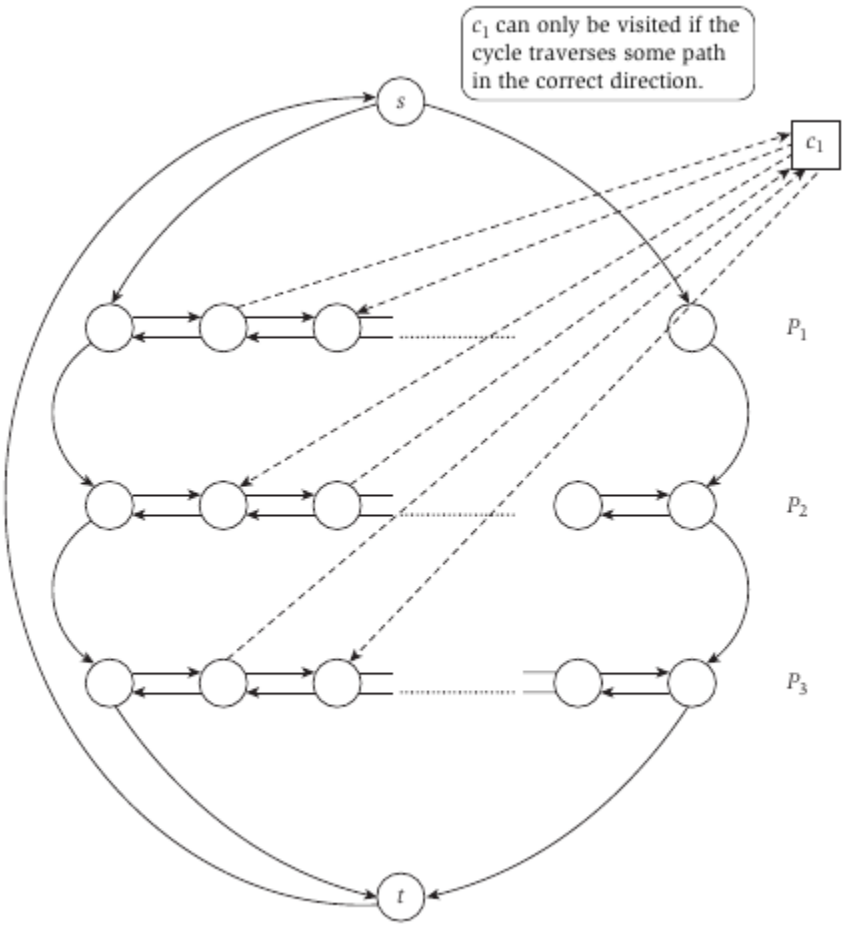


Figure 8.8 The reduction from 3-SAT to Hamiltonian Cycle: part 2.

So if you were to go the wrong way, there would be no way to get up to the clause node, and hence not every vertex would be in the cycle.

It should now be clear that a 3-SAT instance is satisfiable $\Leftrightarrow$ this graph has a Hamiltonian cycle, therefore DHC is NP-complete.

24.9 Undirected Hamiltonian Cycle (UHC)

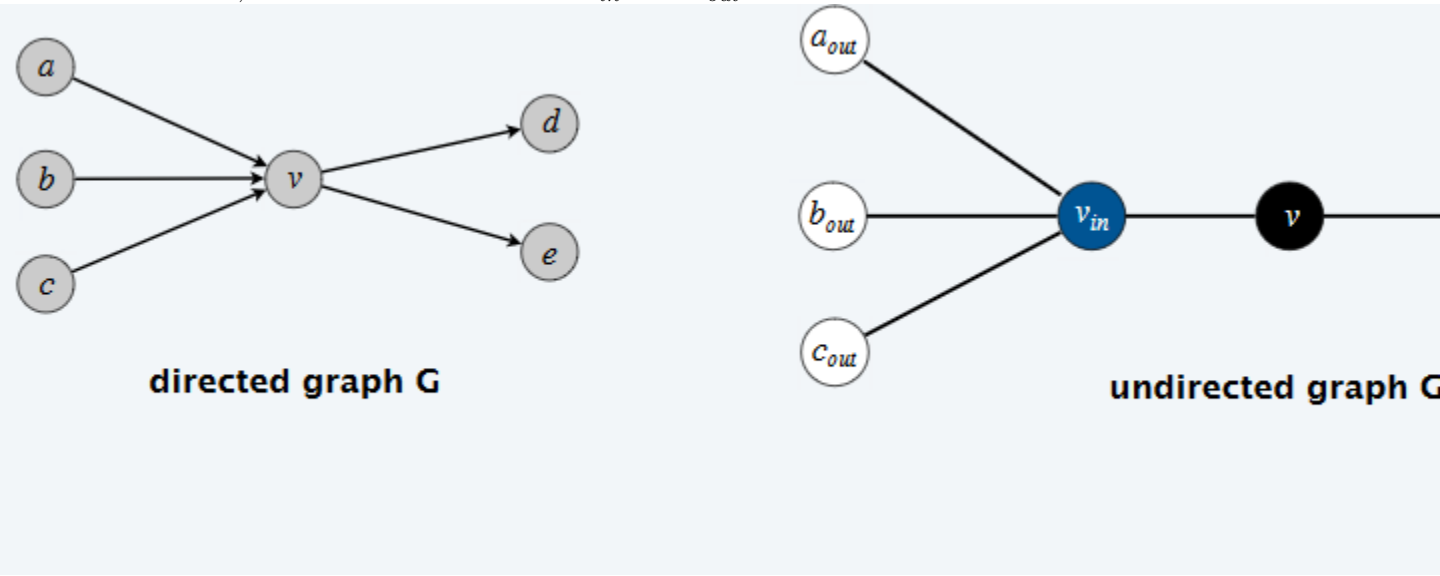
Similar to DHC, the undirected version is just on an undirected graph instead of a directed one.

UHC is NP-complete.

24.9.1 Proof

Given an instance of DHC, we will reduce to UHC. This will be done by creating a new graph G' with $3n$ vertices where n is the number of vertices in G.

For each vertex v, create two new vertices v_{in} and v_{out}. Like this:



If G had a directed Hamilton cycle, then its clear that G' does as well. (We didn't even need to create G' for this.)

The more interesting case is when we have an undirected Hamilton cycle in G'. Then either the cycle C has the order v_{in}, v, v_{out} for any $v \in C$ or v_{out}, v, v_{in}. In the former case, form a direct arc from v_i to v_{i+1} for the v vertices (ignore the in/out vertices). In the latter case form a directed arc in the opposite direction v_{i+1} to v_i.

24.10 Traveling Salesman

The traveling salesman problem (TSP) is similar to the Hamiltonian cycle in which it involves finding a Hamiltonian cycle, however, we must choose the Hamiltonian cycle with length of $\leq k$, where the length is the sum of the weights of each edge.

TSP is NP-complete.

24.10.1 Proof

Given a set edges we can easily verify that they form a Hamiltonian cycle and that the sum of the edge weights is less than k in poly-time.

To prove that it is NP-complete we will reduce from UHC.

We create a new graph G' with all the same vertices and edges of G, but now we add edges between all the pairs of vertices that were not connected in G. We then assign weights to edges of G in G' to be 0, and 1 if they are new edges.

Let k=0. Then, there is an undirected hamilton cycle in $G \Leftrightarrow$ there is some cycle in G'

passing through all vertices with length 0.

To see this, consider $\Leftarrow$ direction. If there is a cycle that passes through all vertices of G' with length 0, then we only used edges that existed in G.
On the other hand, $\Rightarrow$, if there is a Hamiltonian cycle in G then clearly by construction it will have length 0 in G'.

Part V
Dealing With NP-Complete Problems

25 Randomized Algorithms

A random algorithm is any algorithm in which we make random decisions that affect the steps in which the algorithm runs. This does not mean that the data being operated on is random.

Random algorithms can help solve NP-complete problems in roughly polynomial-time (on average) when we otherwise have no polynomial time solution. So, we *expect* the algorithm to complete in polynomial time, but this may not always be the case.

Other times random algorithms help simplify or (on average) speed up even polynomial time algorithms.

25.1 Probability Review

If you've taken probability before you can probably skip this part, but for the benefit of the others, I'll summarize.

A **random variable** is a mathematical representation of something which can assume many values. If we're flipping coins, we could let X be the number of times we see heads. Then we might be interested in the probability that out of 10 flips we see 4 heads. Then we calculate $P(X = 4)$. The event of $X = x$ means the event that the random variable X assumes a specific value x.

The expected value is the mean of a random variable.

$$E(X) = \sum_x xP(X = x)$$

Or the sum of all possible values, times the probability of X assuming that value.

The expected value is linear.

$$E(X + Y) = E(X) + E(Y)$$

$$E(cX) = cE(X)$$

where c is a constant.

$$E(c) = c$$

.

Another important fact of probability is the union of two events. If A and B are outcomes of an experiment, then:

$$P(A \cup B) = P(A) + P(B) - P(A \cap B)$$

Where $A \cap B$ is the event of both A and B occuring. The events ares said to be be disjoint if $P(A \cap B) = 0$ ie: $A \cap B = \phi$

25.2 Randomized Quicksort

Recall that quicksort is done by choosing a pivot, and splitting the list into two, based on all the numbers less than the pivot and greater. Then, recurse on each half.

The performance of quicksort depends on the choice of pivot. Clearly, if we choose the smallest or largest element as the pivot, we will need to do $n-1$ comparisons on that step. If we ALWAYS choose the worst pivot, we will do

$$(n-1) + (n+2) + ... + 1 = \frac{n(n-1)}{2} = O(n^2)$$

operations. (Geometric series).

If we randomly choose the pivot, we could avoid this problem most of the time.

How long do we expect this to take on average? Let X_{ij} be a random variable representing the number of times a_i and a_j are compared. Then the total number of comparisons would be:

$$E(\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij})$$

$$= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E(X_{ij})$$

To find $E(X_{ij})$, first note that each pair of elements is compared at most once. This is because we only compare elements if one of them is a pivot, and if this is the case, then we will place the pivot between the sub lists, and thus it is never compared to other elements. Thus $X_{ij} \in \{0, 1\}$. Also, since we only compare to the current pivot, that means the elements that exist at this stage are: $\{a_i, ..., a_j\}$(no particular order, but a_i ¡ a_j).

Then the probability of a_i being compared to a_j is $\frac{2}{j-i+1}$. The 2 is because we can choose

either a_i or a_j (two choices), and the denominator is the total number of elements to choose from at that step.

Then $E(X) = \frac{2}{j-i+1}$.

Letting $k = j - i$, then we have:

$$\sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1}$$

clearly this is less than:

$$\sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k+1}$$

thus this is $O(nln(n))$. (The inner is essentially an integral of $1/x$, the outer doesn't depend on k so its like multiplying the inner by n)

25.3 Randomized Selection

If we want to select the k-th smallest element in a list of numbers, at first glance it seems we could do this in $O(nlogn)$ using quicksort, then grabbing the k-th element.

We can do better by modifying quicksort slightly.

Let S_L be the numbers less than the pivot x_p, S_R be greater. At each iteration, set S_L and S_R based on a random x_p. Then, if there are $k-1$ elements smaller than x_p, we know that x_p is the k-th smallest element and we're done.

Otherwise, if there are more than $k-1$ elements in S_L then we recurse one S_L since the kth element must be somewhere inside that set.

If there are less than $k-1$ elements is S_L, then we "overshot" the pivot. (We picked a pivot that is less than the k-th smallest element.) So we'll recurse on S_R, but only look for the $k - 1 - |S_L|$-th element of S_R instead of the k-th. This is because we've eliminated $|S_L| + 1$ of the smallest elements (including the pivot itself).

The runtime can be computed by first realizing that half the time we will pick pivots that will separate the list into two lists of size somewhere between $\frac{n}{4}$ and $\frac{3n}{4}$. Thus we have the recurrence:

$$T(n) = \frac{1}{2}T(\frac{3n}{4}) + \frac{1}{2}T(\frac{n}{4}) + O(n)$$

Which is $O(n)$ by the master theorem and the fact that $cO(n) + dO(n) + O(n) = O(n)$.

25.4 Database Contention

Suppose we have n processes $P_1, ..., P_n$ which all want to access a database. We have several rounds in which the databases may attempt (or not) to enter the database. If more than

one enters, both are locked out. The processes cannot communicate.

A randomized approach would be to say that we assign each process a probability p of attempting to enter the database.

Well, letting $p = \frac{1}{2}$ seems reasonable, since we would then expect 1 process to access at a time on average. Lets analyze this: let $S(i,t)$ be the random variable representing that process i accesses the database on round t.

$$P[S(i,t) = 1] = p(1-p)^{(n-1)}$$

Assuming all events are independent, then $P[S(i,t) \cap S(j,t)] = P[S(i,t)]P[S(j,t)]$ (just a law of probability). We make it in when we attempt, and all other processes do not. This happens with probability $(1-p)^{n-1}$ since there are $n-1$ other process that will attempt. So naturally the compliment:

$$P[S(i,t) = 0] = 1 - p(1-p)^{n-1}$$

Summing over all T rounds, the number of accesses a process gets is:

$$S(i) = \sum_{i=1}^{T} S(i,t)$$

Thus the probability that a process never gets to access the database is the event that all $S(i,t) = 0$.

$$(1 - p(1-p)^{n-1})^T$$

So if we go back to our guess of $p = \frac{1}{n}$, we get:

$$(1 - \frac{1}{n}(1 - \frac{1}{n})^{n-1})^T$$

$$= (1 - \frac{1}{n * e})^T$$

(since $lim_{n\to\infty}(1 - \frac{1}{n})^n = \frac{1}{e}$)

So the probability that there exists a process that never accesses the database is:

$$P[fail] = P[\bigcup_{i=1}^{n} S(i) = 0] \leq \sum_{i=1}^{n} P[S(i) = 0]$$

$$= n(1 - (\frac{1}{e * n}))^T$$

if we let T be this magic number: $2ne * ln(n)$

$$\approx \frac{1}{n}$$

this is because $(1-\frac{1}{en})^{en} = (1-\frac{1}{x})^x = \frac{1}{e}$, thus:

$$n(1-(\frac{1}{e*n}))^{2ne*ln(n)}$$

$$= n(\frac{1}{e})^{2ln(n)}$$
$$= ne^{-2ln(n)}$$
$$= ne^{ln(n^{-2})}$$
$$= n*n^{-2}$$
$$= \frac{1}{n}$$

Thus the probability of a success (all $S(i) = 1$ for all i) is $\geq 1-\frac{1}{n}$ which goes to 1 as n grows large, so we can say that with high probability we will succeed with $2enln(n)$ rounds.

25.5 Max 3-SAT

Given a 3-SAT formula $\phi(x_1, .., x_n) = C_1 \wedge ..., \wedge C_k$, can we find a truth assignment that satisfies as many clauses as possible?

Well, if we just randomly set the x_i to 0, 1 as a coin flip (probability 0.5), we get a running time of $O(n)$.

How often will this work?

Each clause can be seen as a random variable who can assume values of 0 (if all it's variables are 0) or 1 (if one variable is 1).

So the probability that C_i is not satisfied is the event that all 3 variables are 0, so $(\frac{1}{2})^3 = \frac{1}{8}$.

Thus:

$$E(C_i) = 0*\frac{1}{8} + 1*(1-\frac{1}{8}) = \frac{7}{8}$$

How many clauses are satisfied? ie what is:

$$E(\sum_{i=1}^{k} C_i)$$

$$= \sum_{i=1}^{k} E(C_i) = \frac{7k}{8}$$

Note that this means that every clause has some truth assignment which satisfies $> \frac{7k}{8}$ clauses since otherwise, we would not be able to achieve an expected value of $\frac{7k}{8}$.

Also note that this means that any instance of 3-SAT with less than 7 clauses is satisfiable, since for any $1 \leq k \leq 7$:

$$\frac{7k}{8} \geq k$$

25.5.1 Johnsons Algorithm

Johnsons algorithm finds the truth assignment which satisfies $\geq \frac{7k}{8}$ clauses by choosing random truth assignments until we find one that works.

So how many times to we expect to retry?

Let p_j be the probability that exactly j clauses are satisfied. Then let p be the probability that at least $\frac{7k}{8}$ clauses satisfied. Then, naturally:

$$p = \sum_{j \geq \frac{7k}{8}} p_j$$

from before we found that the expected number of clauses satisfied is :

$$\frac{7k}{8} = \sum_{j=0}^{k} j p_j$$

by the definition of expected value. Then, splitting the sum:

$$= \sum_{j < \frac{7k}{8}} j p_j + \sum_{j \geq \frac{7k}{8}} j p_j$$

26 Pseudo-Polynomial Time

To define pseudo-polynomial time algorithms we must first define unitary encoding.

Unitary encoding is when the input integer(s) to a problem is(are) encoded as an n-long string of 1's.

For example, the number 10 is not the usual binary encoding of 1010, but rather 1111111111 (10 1's).

So under this encoding, something like the Knapsack problem would have input size:

$$n + W + V + \sum w_i \sum v_i$$

bits.

A pseudo-polynomial time algorithm has a worst-case running time which is bounded by a polynomial in the size of the unitary encoding of the input instance.

Normally when we think of a polynomial time algorithm we think of $O(n^k)$ where n is the input size. This is fine, for small numbers, but if we follow the true definition of polynomial time:
An algorithm runs in polynomial time if its runtime is $O(x^k)$ for some constant k, where x denotes the number of bits of input given to the algorithm.
we see that the definition actually depends on the input size (in bits) not just numerically.

Consider:

```
isPrime(n):
   for i = 2 to n-1:
      if (n mod i) = 0 return False
   return True
```

We would normally look at this and say its $O(n)$, assuming that the modulo operation is $O(1)$. However modulo might take longer if n is very large. Thus a more conservative estimate is more like $O(n^4)$ (as an upper bound, it's probably much smaller). Writing the number n requires $log(n)$ bits, so if we let $x = log(n)$ be the number of bits in the input for some number n. Then:

$$2^x = n$$
$$\Rightarrow (2^x)^4 = n^4$$

so our algorithm is actually $O(2^{4x})$ which is not even close to polynomial.

"An algorithm runs in pseudopolynomial time if the runtime is some polynomial in the numeric value of the input, rather than in the number of bits required to represent it"

This section was heavily inspired (and some pieces taken directly from) this Stackoverflow thread.

26.1 0-1 Knapsack

The key to making the knapsack problem polynomial is to encode our inputs as an array of bits. By doing so, we effectively translate it into a unitary encoding. Meaning any polynomial time algorithm we can find (with respect to the inputs) will be pseudo-polynomial.

For this algorithm I'd recommend section 22 of my COMP251 guide. It explains a $O(nW)$ dynamic program that is the same as the one the Prof gave in the slides.

There is also a modified $O(nV)$ algorithm that uses the same idea. But instead we have an array $A = [1, ...n, 0, ..., V]$. And now each entry keeps track of whether it exists, and the weight of the item (not the value!). Then we try to find the minimum weight required to obtain a profit of at least v, rather than highest value subsets with weight less than w. At each iteration, if we take the item i, the weight increases by w_i. If we don't take it, then the weight is unchanged.

```
A[n][V]; # Array to hold result

initialize A[n][j] = INFINITY for all j.
for i in range(n):
   for j in range(j):
      if i <= 0 or v <= 0:
         A[i][j] = 0 # Then the weight required to get value of at least 0 is 0
      else:
         A[i][j] = min(A[i-1][j], A[i-1][j-v_i] + w_i) # either take it or dont
            take it
```

Then, if there is some j for which A[n][j] $< \infty$ then there is a solution. Otherwise no solution. This algorithm clearly runs in $O(nV)$.

27 Strong NP-completeness

A decision problem in NP is strongly NP-complete if the version of it which uses unitary encodings is NP-complete.

Similar to proving problems are NP-complete, we can prove problems are strongly-NP complete by reductions. Given a problem π in NP, if we can reduce some NP-complete problem π' to unitary encodings of π with the same yes-no answer in time polynomial in the size of π'.

If π' is just the unitary encoded version of π we call this a self reduction.

27.1 Some Self Reduction Examples

27.1.1 Hamilton Cycle

Given some standard binary encoding, we will have an $N \times N$ adjacency matrix to represent the graph. We also have n vertices. So n is represented with $log(n)$ and we still need n^2 bits for the matrix so $\Theta(n^2)$

If we instead encode the input n vertices in unitary, we have $n + n^2 = \Theta(n^2)$

27.1.2 3-SAT

For binary encoding we have n variables, m clauses and $3m$ literals. So there are $log(n) + 3m(log(n) + 1)$ bits.

This can be done with unitary encoding if we use a string of n 1's for n, m 1's for m, and for each literal, use a string up up to n 1's to represent different numbers. Can do this in $O(nm)$ time, unless $3m < n$ this is because in this case, there would be unused variables in the clauses. We would then need to scan through and reindex the used ones, so it would be $O(m^2)$

27.2 Strong-NP Completeness Proofs

27.2.1 Reduction Proving TSP in Strongly NP-Complete

Similarly to Hamilton cycle the binary version has size $\Theta(n^2)$ since $log(n)$ for the size of n and n^2 for the adjacency matrix.

We can reduce Hamilton cycle to TSP by making the distances 0 or 1, and turning the hamilton cycle into a unitary encoding of a path, so n bits, with n^2 for the adjacencies, so $O(n^2)$

27.3 Note on Pseudo-Polynomial Time Algorithms and P=NP

If there is a pseudo polynomial time algorithm for a strongly NP-Complete problem, then the NP-complete problem which reduces to the strongly NP-complete problem has a polynomial time algorithm, but then P=NP.

28 Approximation Algorithms I

The goal of an approximation algorithm is to give a polynomial time solution to the NP-complete problems by giving solutions which are not always optimal. If we allow the results to be "good enough" then we can find "solutions" in polynomial time.

The measure of "good enough" is given by an approximation ratio, r. In a maximization problem, the solution is good enough if the solution is within $\frac{1}{r}$ times the optimal solution. In a minimization problem, the solution is good enough it is within a factor of r of the optimal solution.

Note that $r \geq 1$, and the closer r is to 1, the better the approximation.
,

28.1 Examples of Approximation Algorithms

28.1.1 Maximum Matching

The goal is to find a maximum matching M in a graph $G = (V, E)$. If M is maximal, that implies that the end points of M form a vertex cover of size $2|M|$.

If $|V| = n$, $|E| = m$, and we have $2m$ numbers, then we can use the fact that these $2m$ numbers can be separated into even and odd numbers $2i$ and $2i - 1$. Then any matching M will connect vertex v_{2i} to vertex v_{2i-1}. (Even to odd m)

Then, for each edge, if you haven't already added it the matching, add it, and ensure you dont reuse the endpoints.

Dead simple greedy algorithm and doesn't seem like it would work. And it doesn't always, but its never worse than half the optimal solution. Ie: it is a 2-Approximation.

To prove that this is a 2-Approximation, recall that every maximal matching must cover at least half the vertices of G, since otherwise we could extend it, since there would be some edge for which both vertices are not in the endpoints of M. This means there are $E/2$ edges in M. ie: $|M| \leq 2m$.

28.1.2 Minimum Vertex Cover

Can apply the 2-Approximation for matching above and simply return the endpoints of the matching. This will give a cover of size $2|M|$. The best a cover can possibly get is $|M|$, so this is within $1/2$ of the optimal.

28.1.3 Fast 2 -Approximation For Knapsack

First, delete all items i with weight $w_i > W$, since we'd never be able to add them anyway. Next, solve the fractional version of the new problem using the greedy algorithm. (Take the best value per weight items). This gives a set S of items, (which were not split) and then a final item l_i which would need to be split to fit into the backpacks remaining capacity.

In the integer solution of the knapsack problem, we cannot take part of an item. So we can choose to take S or l_i, but not both. In general, the fractional solution to knapsack is higher than the integer solution, so the value of S + value of l_i will be larger than twice the optimal integer solution. Thus we can take the higher value of the two, and be within half the optimal solution.

28.2 Inapproximability of Problems

Some problems can be proven to have no polynomial time approximation with a ratio less than some number r. (Unless P=NP).

In general, to show inapproximability, you can show that if such an algorithm exists, it would mean that it could solve some NP complete problem in polynomial time, thus reaching a contradiction.

28.2.1 Inapproximability of TSP

For any polynomial p, unless P=NP, there is no approximation algorithm for TSP with ratio $r \geq 1$.

Proof. Suppose we have an instance G of Hamilton cycle. We reduce to TSP by setting distances to 1 if they are adjacent in G, and $nr + 1$ otherwise.

If there is a Hamilton cycle, the shortest tour has length n, since we'd have to use each edge from the cycle in the TSP tour. Otherwise, the shortest tour has length at least $nr + 1 + (n - 1) > nr$, since we would have to use some edge to visit 1 of the vertices not in the cycle to complete the tour.

This means theat the cost of any tour that is not a hamiltonian cycle in the original graph would cost at least $rn + n$, which is a difference of rn, or factor of $r + 1$!

Suppose we have an approximation algorithm for which $r \geq 1$. If G contains an Hamilton cycle, then we must return it. If G has no Hamiltonian cycle, then our algorithm will return a cost of more than rn, which means it is not a polynomial time r-approximation algorithm. Contradiction.

□

28.2.2 Inapproximability of Chromatic Number

The chromatic number χ of a graph G is the minimum number of colors required for a proper coloring of G.

Unless P=NP, there is no polynomial time algorithm with $r < 4/3$ for the chromatic number problem.

Proof. If there were such an algorithm A that could find a minimal χ, in polynomial time, within 4/3 the optimal, then it would return colorings with $\chi \leq 3$ for 3-colorable graphs, and $\chi \geq 3$ otherwise. But then this would mean that it can solve 3-coloring which is NP-complete, which means P=NP.

□

28.3 Euclidean Symmetric TSP

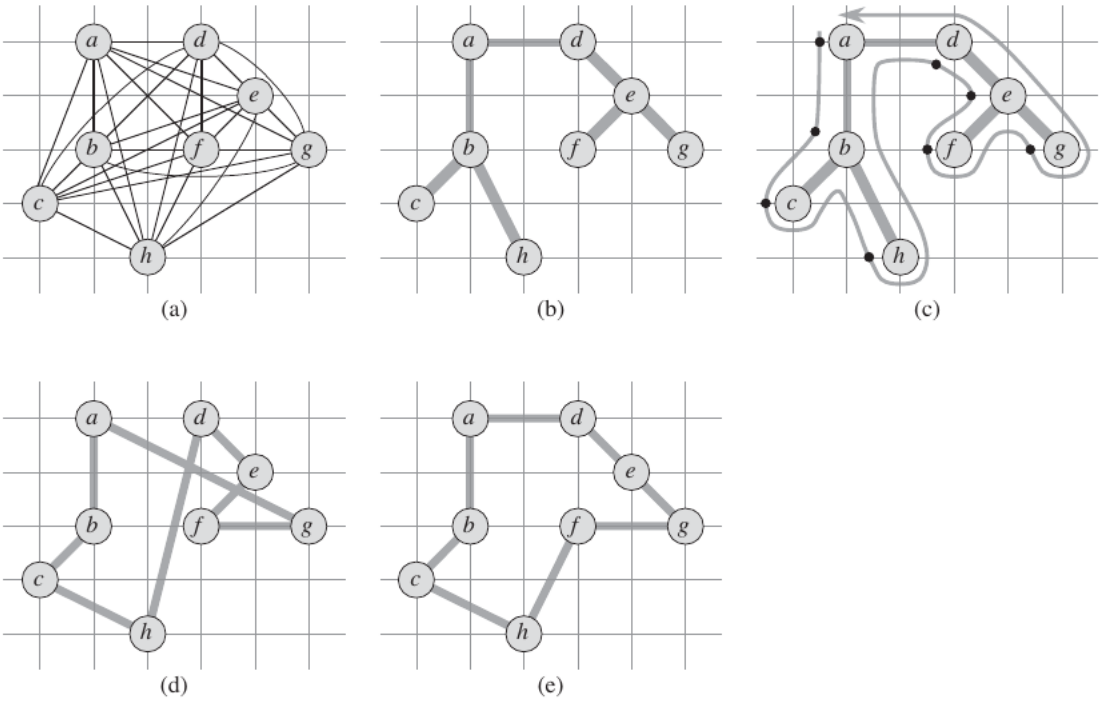
Usually, going from a city u to a city w is at most as long as going from u to some intermediate city v then to w. This is the **triangle inquality**:

$$c(u, v) \leq c(u, v) + c(u, w)$$

where $c(i, j)$ is the cost of going from city i to j.

If we think geographically this makes sense, and actually any set of points of a euclidean plane will have this property. However, in a non-euclidean, general sense this won't always work. You could have a graph which doesn't obey this property.

The idea of the approximation algorithm for Euclidean TSP is to first come up with a minimum weight spanning tree (MWST). Then, its weight gives a lower bound on the total cost of a tour, since it will cover all the vertices, without any cycles, and have minimum cost. Then, to return back to the start, you use at least 1 edge, and at most all of the edges. So the upper bound of the cost is at most twice the cost of the MWST.



Cormen et al. pg 1113 Figure 35.2

The above figure shows, starting from a TSP instance (a), make a minimum spanning tree using your favorite algorithm (Prims, Kruskals, Boruvkas, ets.) (b), then, traverse around the tree to form a cycle. Since this graph follows the triangle property, and is connected, there is always a direct edge from h to d, which will be as fast as any other path from h to d, for example. This is true for all the pairs.

Heres a more formal proof:

Proof. Clearly the above algorithm runs in polynomial time because it is a collection of polynomial algorithms (MST, and pre-order traversal). Suppose the optimal tour is H^*. We can optain a spanning tree by taking any cycle (and therefore any tour), and deleting an edge. Each cost is non-negative so the tree will have less cost than the tour. Therefore the minimum spanning tree T will have cost:

$$c(T) \leq c(H^*)$$

The walk lists the vertices when they are first visited, and when they are visited again on the way back potentially. Call this W. Since the full walk will traverse each edge of T twice, once each direction, then the cost is at most:

$$c(W) = 2c(T) \leq 2c(H*)$$

We can delete multiple visits since the triangle inequality ensures we can take the shortcut directly from one vertex to another with less than or equal to the same weight (as described above using the figure). This the tour H with no duplicate vertices, then by the triangle property:

$$c(H) \leq c(W) \leq 2c(H*)$$

This this is a 2-approximation for Euclidean TSP. □

28.4 Polynomial Time Approximation Schemes

A polynomial approximation scheme describes a set of algorithms for which there is an approximation algorithm with ration $\epsilon \in \Re$ for any $\epsilon > 0$

28.5 Knapsack Problem

Suppose we have an instance of the knapsack problem and use the dynamic approach as described in section 26.1. That will give the optimal result, but it's only pseudo polynomial.. For large n it won't do...

For the approximation scheme, suppose you "approximate" each value in the items to some smaller ranged integer value:

item	value	weight
1	934221	1
2	5956342	2
3	17810013	5
4	21217800	6
5	27343199	7

original instance (W = 11)

item	value	weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

rounded instance (W = 11)

The new values are found using this formula:

$$v_i' = ceil(\frac{v_i}{\theta})\theta$$

where $\theta = \frac{\epsilon v_{max}}{2n}$.

This rounding doesn't change the values much and so doesn't really help a whole lot. As it turns out, $v_i \leq v_i' \leq v_i + \theta$, but now we have that every v_i' has a common multiple of θ, so we can equivalently solve using $v_i'' = ceil(v_i'/\theta)$. The idea is that this won't mess up the solution too much, but will reduce the input size greatly, so that the pseudo-polynomial time algorithm is still fast.

As it turns out, if S is the solution found from our approximation algorithm, and S^* is any other feasible solution, we have that $\sum_{i \in S^*} v_i \leq (1+\epsilon)\sum_{i \in S} v_i$ ie: our approximation is within a factor of $1+\epsilon$ of the optimal.

Proof. We know that :

$$\sum_{i \in S^*} v_i' \leq \sum_{i \in S} v_i'$$

because we rounded up.

Since the values v_i and v_i' are very close (within θ we have:

$$\sum_{i \in S^*} v_i \leq \sum_{i \in S^*} v_i' \leq \sum_{i \in S} v_i' \leq \sum_{i \in S} v_i' + \theta \leq n\theta + \sum_{i \in S} v_i$$

Recall we used $\theta = \frac{\epsilon v_{max}}{2n}$, so that means that $v_{max} = 2nb/\epsilon$, and that rounding v_{max} has no effect. It's important to note that we assumed that each item fits in the knapsack, since now we can use this to have $\sum_{i \in S} v_i' \geq v_{max}' = 2nb/\epsilon$ If we had items that didn't fit in the knapsack this would not always hold.

So using the inequalities from before we now have:

$$\sum_{i \in S} v_i \geq \sum_{i \in S} v_{max}' - nb$$

$$\sum_{i \in S} v_i \geq 2nb/\epsilon - nb$$

and if ϵ is less than 1, then:

$$\sum_{i \in S^*} v_i \leq (1+\epsilon)\sum_{i \in S} v_i$$

□

Now, for the new running time, notice that the largest V can be is nv_{max}, so if our original running time was $O(nV)$, our new running time is $O(n^2 v_{max}') = O(2n^3/\epsilon)$

29 Approximation Algorithms II

29.1 Revisiting Euclidean Symmetric TSP

An Eulerian walk in a graph G is a sequence of vertices (may have repeated vertices) such that each edge is visited once. This stems from Eulers bridge problem in which we attempt to find a route in which you can visit all islands of a city, crossing each bridge only once, returning to the original island.

We'll use this theorem to help us lower the approximability bound: If G is a connected multi-graph (possibly multiple edges between vertices), then G has an Eulerian Walk $\Leftrightarrow$ every vertex degree is even. Further, we can find such a walk in polynomial time if it exists.

29.1.1 3/2 Approximability to Euclidean Symmetric TSP

Notice that Eulerian walks are very similar to the TSP problem. An Eulerian walk is simply the TSP problem with a constraint that the sum of all edge weights is less than the sum of all edge weights, where the weights of all edges are 1.

We'll use this similarity, and the fact that a polynomial time algorithm exists for Eulerian walks to reduce our bound from 2 to 3/2.

Given a Euclidean Symmetric TSP problem, let the weight of each edge $(i, j) = d_{ij}$. Let D^* be the total cost of the best tour, P^*

Similar to before, find the minimum weight spanning tree T. The best tour must at least contain a spanning tree, the optimal path P^* must contain the minimum spanning tree T.

Now, we need to find a minimum matching M inside this tree, which spans the set S, of vertices which have an odd degree in T. This set S will have an even number of vertices since the sum of the degrees in T is (one degree for each end point of an edge):

$$\sum_{v \in T} deg(v) = 2|E|$$

$$= \sum_{v \in T | deg(v) = 2k+1} deg(v) + \sum_{v \in T | deg(v) = 2k} deg(v)$$

for some k (basically split the sum into even and odd). The sum of even degrees is even, and so this means that the sum of the odd degrees must also be even. If this is true, then there must be an even number of odd degree vertices. (For the sum of odd numbers to be even you must have an even number of them)

Now that we know $|S|$ is even, we see that there is some matching M which can span

all of them. Ensuring that the matching is minimal, we can now construct a graph G' made from the edges of T and M. The vertices of $V - S$ which were even have no change in F, since they were excluded from the matching. However, the vertices of S have a new edge from M, so they now have even degree.

After this big long process we can finally use the theorem mentioned in the introduction to this section: since all the degrees are even, we know that a Eurlerian walk exists, and can be found in polynomial time.

Now, since the graph is connected, we can use the triangle property to shortcut directly from city to city (by removing vertices from the path that were visited twice). How well does this algorithm perform?

Since the cost of T is at most the cost of P^*, then using the triangle property to shortcut does not increase the length. The sum of the length of edges in M, however, is at most half the cost of the edges in T, since we used half the edges, and we chose the minimum matching withing the tree. Thus in total, the weight is at most 3/2 the optimal.

29.2 Using LP's to Approximate ILP's

Sometimes integer linear programming can be very long and difficult problem. It's often faster to approximate a solution using a linear program and rounding.

This will be illustrated using the weighted vertex cover problem.

29.2.1 Weighted Vertex Cover

To formulate weighted vertex cover as an ILP, let the objective function be:

$$min(\sum_{i \in V} w_i x_i)$$

subject to:

$$x_i + x_j \geq 1 | (i,j) \in E(G)$$
$$x_i \in \{0,1\}, i \in V(G)$$

basically the first constraint enforces the cover property, and the second just allows for choosing the vertex (1) or not choosing it (0).

If x^* is the optimal solution to this ILP, then $S = \{i \in V(G) : x_i^* = 1\}$ is a minimum weight vertex cover. (Basically just how to read off the result from the ILP).

If we instead relax this to an LP, by removing the condition that $x_i \in 0, 1$, the optimal value of this will actually be better than the integer one, since we have removed contraints.

In general, removing constraints from an LP or ILP will allow better values for the objective function.

However, our new LP is not equivalent to the weighted vertex cover, since we can take any real valued portion of a vertex. (no longer a binary decision)

However, if we round to fractional values, we can get close to the same solution.

If x^* is the optimal solution, then $S = \{i \in V : x_i^* \geq 1/2\}$ is a vertex cover whose weight is at most twice the minimum possible weight. (Take all the vertices who have value greater than 1/2 to be in the cover)

Proof. First, we need to prove that S is in fact a vertex cover.

From the constraint that $x_i^* + x_j^* \geq 1$, we have that either $x_i^* \geq 1/2$ or $x_j^* \geq 1/2$, or both. Then this means that this edge is covered.

Now to prove that it is within a factor of 2 of the optimal, let S^* be the optimal vertex cover. Then since we removed contraints (and can get a better value for the objective function, we have that)

$$\sum_{i \in S^*} w_i \geq \sum_{i \in S} w_i x_i^* \geq 1/2 \sum_{i \in S} w_i$$

where the last inequality comes from the fact that all the x_i^* are more than 1/2.

LP's can be solved in polynomial time, so this algorithm is a 2-Approximation algorithm for weighted vertex cover. □

29.3 The Primal Dual Method

29.3.1 Maximum Matching ILP and Vertex Cover

The fractional relaxation of the maximum matching ILP is:

$$max(\sum_{e \in E} x_e)$$

where for each vertex, we must have that:

$$\sum_{v : v \in e = (u,v)} x_e \leq 1$$

and

$$\forall e \in E(G)(x_e \geq 0)$$

Next, the fractional relaxation of the vertex cover ILP is:

$$min(\sum_{v \in V} y_v)$$

subject to for all $e = (u, v)$:

$$y_u + y_v \geq 1$$

$$y_v \geq 0$$

Notice that these are duals of each other!

In general, the optimal solution of duals are equal, and since the optimal solution of the relaxed problem is "better" than the ILP version, we have that the optimal solution for the vertex cover ILP is greater than the optimal solution of the maximum matching ILP.

30 Cutting Stock Problem

The formulation of the cutting stock problem is simple. We have a roll of material that we want to use to cut out a given demand of sizes. This roll, call the *raw* or *stock* roll has a given width. We then have k different type of $finals$, or cuts, that we need to make. For each final i, there is a required width w_i and number f_i copies.

Each raw can be cut into a_1 finals of width w_1, and a_2 finals of width w_2 and so on. There are many different ways of cutting each raw. Let each of these ways be represented by a vector $P_j = (a_{1j}, ..., a_{kj})$.

If we use P_j x_j times, and there are N patterns to choose from, then we must have that $\sum_{j=1}^{N} a_{ij}x_j \geq f_i$ where f_i is the number of finals we need of a given width to satisfy the order.

The problem is then, how can we minimize the number of rolls required to satisfy the order? That is, if x_j is the number of times a raws cut with pattern j, then: $\sum_{j=1}^{N} x_j$ is the number of raws cut. We then want to minimize the number of raws cut, subject to the constraint that we can't have $x_j < 0$ and we must satisfy the order.

Note that these bounds allow for fractional results, even though in some real world applications that may not be possible. Thus this algorithm is only an approximation algorithm, where we round down the number of raws used in the optimal solution. This may sometimes work, but often this not the best. However no better way is known.

There's a problem, however, preventing us from simply soling the LP and rounding. There might be exponentially many cutting patterns for a given input. (Generating all possible patterns would be very tedious). Instead, we'll generate possible patterns depending on the state of the solution as we're solving.

30.1 Delayed Column Generation

The idea here is that we will start with the simplest possible solution, and write it as a matrix. For example, suppose we have a raw width of 20, and the finals:
9, 511
8, 301
7, 263
6, 383

where the left number is the width, and the right number is the number of finals.

How can we break up each raw? Start with the absolute simplest solution: say we only use one kind of final at a time. So here we could fit 9 into 20 twice, 8 into 20 twice, 7 into 20 twice, and 6 into 20 three times.

If we represent this as a matrix we have:

$$\begin{pmatrix} 2 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 3 \end{pmatrix}$$

Where each column is a way of cutting up the raw.

If we were to use this solution, we would require 511/2 sheets to meet the demand of the first request, 301/2 for the second, and so on.

We can obviously do better.

The matrix above represents whats called **basic patterns**, or basic variables.