Data Compression

(For Images)

Types of Compression

- Pixel packing
- RLE (run-length encoding)
- Dictionary-based methods
- JPEG compression
- Fractal Image Compression

Factors to look out for:

- *Lossy* or *lossless* compression?
- What sort of data is a method good at compressing?
- What is its compression ratio?

Richardson, Chapter 6; Chapman & Chapman, Chapter 5

Pixel Packing

- Not a standard “data compression technique” but nevertheless a way of not wasting space in pixel data
- e.g.
  - suppose pixels can take grey values from 0-15
  - each pixel requires half a byte
  - but computers prefer to deal with bytes
  - two pixels per byte doesn’t waste space
- *Pixel packing* is simply ensuring no bits are wasted in the pixel data

Run Length Encoding (RLE)

Basic idea is this:

- AAAAAAAAAAAAAAA would encode as 15A
- AAAAAAbbbXXXXXt would encode as 6A3b5X1t

So this compression method is good for compressing large expanses of the same colour - or is it?

RLE compression ratio

- Of the 110 pixels in the 10 ´ 11 pixels sample taken from the previous image, 59 different colours altogether!
- RLE compression ratios not good in general, because there are rarely repeat runs of pixels

Full image: 371 ´ 247 bitmap

275Kb raw data
(274911 = 371 ´ 247 ´ 3) bytes

91K RLE encoded

Compression ratio approx 3:1 in this case

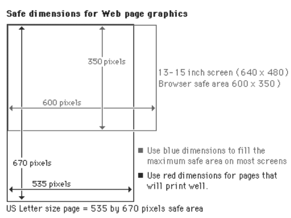
RLE compression ratio

- Another example, with a diagram this time

Full image: 350 ´ 264 bitmap

277Kb raw data
(277200 = 350 ´ 264 ´ 3) bytes

46.5K RLE encoded

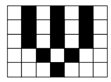


Compression ratio approx 6:1 in this case

Dictionary Methods

- A common way to compress data (pixels, characters, whatever!) is to use a *dictionary*
- The dictionary contains strings of bytes
  - e.g. particular pixel patterns
  - not limited to patterns of one colour, as with RLE
- Data is encoded by replacing each data string that has an entry in the dictionary with its index number in the dictionary
- Shorter to write an index number than a whole string!
- Dictionary may be particular to the image, or may be "standard" for particular image types

Patterns of Pixels



- Poor results with RLE as runs of pixels with same colour are very short
- But there are repeating patterns with two colours that could be included in a dictionary
- Hence, could replace each byte pattern with a pointer to it (or its index number in the dictionary)

Huffman and CCITT Compression

- Developed for fax machines and document scanners
- Uses a predefined dictionary of commonly occurring byte patterns from B&W documents containing large amounts of text in a variety of languages and typical examples of line art
- Commonly occurring patterns are given low (short) indices (codes) in the dictionary
- Data is encoded by replacing each image string that occurs in the dictionary with its index number
- Dictionary is not part of the compressed file.

The Lempel-Ziv-Welch Algorithm

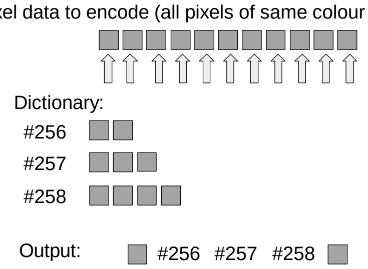
- The Lempel-Ziv-Welch method is another such dictionary algorithm, in which the dictionary is constructed as the encoding (compression) progresses
  - (actually Ziv was the first author on the original papers!)
- LZW starts with a dictionary:
  - Entries 0-255 refer to those individual bytes
  - Entries 256 onwards will be defined as the algorithm progresses
- Each time a new code is generated it means a new string of bytes has been found.
- New strings are generated by appending a character `c` to the end of an existing string `w`.

The LZW Algorithm (2)

```
set w = "";
while (not EOF)
  read a character c;
  if w+c exists in the dictionary
    w = w+c;
  else {
    output the code for w;
    add w+c to the dictionary;
    w = c;
  }
endwhile
```


A Pixel Example

Pixel data to encode (all pixels of same colour in this example):



When Is LZW Useful?

- Good for encoding pixel data with a limited palette, and/or repetitive data
  - line drawings
  - diagrams
  - plain text on a plain background
- Not good for photographic images
  - large colour range and complex features results in few repeating patterns to include in a dictionary
- see related tutorial question on character string compression using LZW

13

JPEG

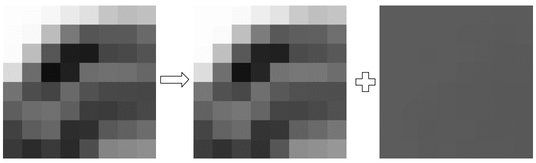
- **J**oint **P**hotographic **E**xperts **G**roup
- Designed to compress photographs
  - colour or greyscale
  - good at compressing "real" scenes
  - not good for line drawings, diagrams, lettering, cartoons
- Designed for human viewing, exploits our inability to see a full range of colours (& to perceive high frequencies)
  - **Lossy algorithm**
  - Not good for computer analysis of data
    - e.g. medical imaging

14

JPEG: How it works

Step 1:

- If a colour image, transform the image into a suitable colour space, with a g (luminance or brightness, Y) component
  - e.g. from RGB to HSV / XYZ / Lab / YCbCr
  - not necessary for greyscale images

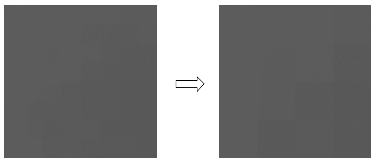


15

JPEG: How it works (2)

Step 2 (optional):

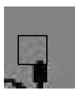
- Leave the g data alone, but "downsample" both lots of colour data
- Reduce resolution by 2, in the y (and maybe x) directions



16

JPEG: How it works (3)

Step 3:



- Divide the image (both g and colour data) into 8 ´ 8 pixel blocks

Step 4:

- For each block, perform a DCT (Discrete Cosine Transform) on the data
- This takes the 64 (integer) values to a different 64 (non-integer) values
  - amplitudes of spatial frequencies

17

JPEG: How it works (4)

Step 5:

- Use quantization on these 64 values (divide by a specially chosen number, and round to the nearest integer)
  - e.g. amplitudes of lowest frequencies may range from 0-255
  - slightly higher frequencies have amplitudes divisible by 4
  - highest frequencies may only have amplitudes of 0 or 128

Step 6:

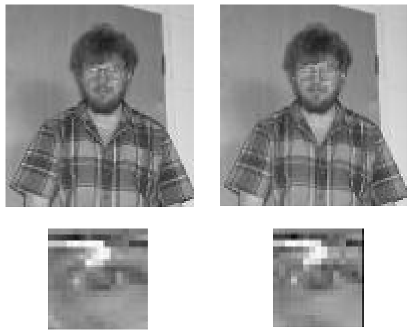
- Store these numbers in a space-efficient way
  - RLE and Huffman coding
  - long strings of 0 coefficients

18

JPEG (contd)

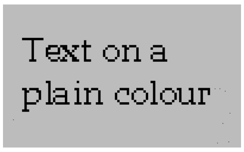
- By choosing a different number in step 5 (the *quantization coefficient*), we get different amounts of compression
- Trade-off of quality versus size of compressed data
- Decoding a JPEG is the reverse process:
  - unpack the efficiently-stored data
  - do a reverse DCT on both the colour data and the g to get the 8 ´ 8 pixel blocks
  - combine the colour data with the g and display the result
  - BUT no recovery from the *quantization* processes

19



20

Text / background boundary is a high spatial frequency - JPEG attempts to smooth this!!



21

JPEG Compression

How good is it?

- For full-colour images, the best-known lossless compression about 2:1
- For reasonable quality, compression ratios of 10:1 or 20:1 quite feasible for JPEGs
  - Clive's picture compressed with a ratio of 15:1
- For poor quality images (thumbnails?), 100:1 possible

- Re-encoding loses more information

22

How Do We Compress Movies?

- Compress individual frames using any of the techniques mentioned already
  - spatial compression
- High*, lossy* compression is OK as the quality of individual frames can be lower than for still images as our perception is dominated by motion
- Make use of limited changes between frames
  - key frames
  - difference frames
  - temporal compression
- More on this in a later lecture!

23

End of Lecture

Next lecture will look at the multitude of different graphic file formats.

24