Analysis of Algorithms

Piyush Kumar

(Lecture 5: Compression)

Welcome to 4531

Source: **Guy E. Blelloch, Emad, Tseng …**

Compression Programs

- File Compression: Gzip, Bzip
- Archivers :Arc, Pkzip, Winrar, …
- File Systems: NTFS

Multimedia

- HDTV (Mpeg 4)
- Sound (Mp3)
- Images (Jpeg)

Compression Outline

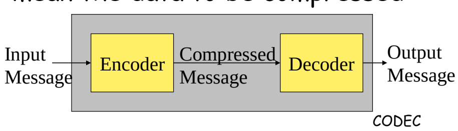
Introduction: Lossy vs. Lossless

Information Theory: Entropy, etc.

Probability Coding: Huffman + Arithmetic Coding

Encoding/Decoding

Will use "message" in generic sense to mean the data to be compressed



The encoder and decoder need to understand common compressed format.

Lossless vs. Lossy

__Lossless__: Input message = Output message

__Lossy__: Input message ≈ Output message

Lossy does not necessarily mean loss of quality. In fact the output could be "better" than the input.

- Drop random noise in images (dust on lens)
- Drop background in music
- Fix spelling errors in text. Put into better form.

Writing is the art of lossy text compression.

Lossless Compression Techniques

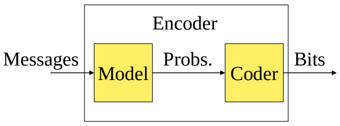
- LZW (Lempel-Ziv-Welch) compression
  - Build dictionary
  - Replace patterns with index of dict.
- Burrows-Wheeler transform
  - Block sort data to improve compression
- Run length encoding
  - Find & compress repetitive sequences
- Huffman code
  - Use variable length codes based on frequency

How much can we compress?

For lossless compression, assuming all input messages are valid, if even one string is compressed, some other must expand.

Model vs. Coder

To compress we need a bias on the probability of messages. The model determines this bias



Example models:
- Simple: Character counts, repeated strings
- Complex: Models of a human face

Quality of Compression

Runtime vs. Compression vs. Generality

Several standard corpuses to compare algorithms

Calgary Corpus

- 2 books, 5 papers, 1 bibliography,
  1 collection of news articles, 3 programs,
  1 terminal session, 2 object files,
  1 geophysical data, 1 bitmap bw image

The **Archive Comparison Test** maintains a comparison of just about all algorithms publicly available

Comparison of Algorithms

Program	Algorithm	Time	BPC	Score
BOA	PPM Var.	94+97	1.91	407
PPMD	PPM	11+20	2.07	265
IMP	BW	10+3	2.14	254
BZIP	BW	20+6	2.19	273
GZIP	LZ77 Var.	19+5	2.59	318
LZ77	LZ77	?	3.94	?

Information Theory

An interface between modeling and coding

- **Entropy**
  - A measure of information content
- **Entropy of the English Language**
  - How much information does each character in "typical" English text contain?

Entropy (Shannon 1948)

For a set of messages *S* with probability *p(s)*, $s \in S$, the **self information** of *s* is:

$$i(s) = \log \frac{1}{p(s)} = -\log p(s)$$

Measured in bits if the log is base 2.

The lower the probability, the higher the information

Entropy is the weighted average of self information.

$$H(S) = \sum_{s \in S} p(s) \log \frac{1}{p(s)}$$

Entropy Example

$$p(S) = \{.25, .25, .25, .125, .125\}$$
$$H(S) = 3 \cdot .25 \log 4 + 2 \cdot .125 \log 8 = 2.25$$

$$p(S) = \{.5, .125, .125, .125, .125\}$$
$$H(S) = .5 \log 2 + 4 \cdot .125 \log 8 = 2$$

$$p(S) = \{.75, .0625, .0625, .0625, .0625\}$$
$$H(S) = .75 \log(4/3) + 4 \cdot .0625 \log 16 = 1.3$$

Entropy of the English Language

How can we measure the information per character?

ASCII code = 7

Entropy = 4.5 (based on character probabilities)

Huffman codes (average) = 4.7

Unix Compress = 3.5

Gzip = 2.5

BOA = 1.9 (current close to best text compressor)

Must be less than 1.9.

Shannon's experiment

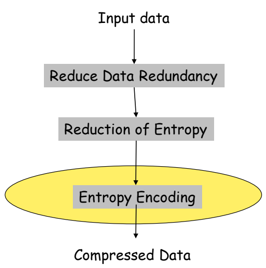
Asked humans to predict the next character given the whole previous text. He used these as conditional probabilities to estimate the entropy of the English Language.

The number of guesses required for right answer:

# of guesses	1	2	3	3	5	> 5
Probability	.79	.08	.03	.02	.02	.05

From the experiment he predicted **H(English) = .6-1.3**

Data compression model



Coding

How do we use the probabilities to code messages?

- **Prefix codes and relationship to Entropy**
- **Huffman codes**
- **Arithmetic codes**
- **Implicit probability codes…**

Assumptions

Communication (or file) broken up into pieces called messages.

Adjacent messages might be of a different types and come from a different probability distributions

We will consider two types of coding:

- **Discrete**: each message is a fixed set of bits
  - Huffman coding, Shannon-Fano coding
- **Blended**: bits can be "shared" among messages
  - Arithmetic coding

Uniquely Decodable Codes

A **variable length code** assigns a bit string (codeword) of variable length to every message value

e.g. `a = 1, b = 01, c = 101, d = 011`

What if you get the sequence of bits `1011` ?

Is it `aba, ca,` or, `ad`?

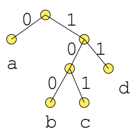
A **uniquely decodable code** is a variable length code in which bit strings can always be uniquely decomposed into its codewords.

Prefix Codes

A **prefix code** is a variable length code in which no codeword is a prefix of another word

e.g a = 0, b = 110, c = 111, d = 10

Can be viewed as a binary tree with message values at the leaves and 0 or 1s on the edges.



Some Prefix Codes for Integers

n	Binary	Unary	Split
1	..001	0	1|
2	..010	10	10|0
3	..011	110	10|1
4	..100	1110	110|00
5	..101	11110	110|01
6	..110	111110	110|10

Many other fixed prefix codes:
Golomb, phased-binary, subexponential, ...

Average Bit Length

For a code *C* with associated probabilities *p(c)* the **average length** is defined as

$$ABL(C) = \sum_{c \in C} p(c) l(c)$$

We say that a prefix code *C* is **optimal** if for all prefix codes *C'*,

$ABL(C) \leq ABL(C')$

Relationship to Entropy

Theorem (lower bound): *For any probability distribution p(S) with associated uniquely decodable code C,*

$$H(S) \leq ABL(C)$$

Theorem (upper bound): *For any probability distribution p(S) with associated optimal prefix code C,*

$$ABL(C) \leq H(S) + 1$$

Kraft McMillan Inequality

Theorem (Kraft-McMillan): *For any uniquely decodable code C,*

$$\sum_{c \in C} 2^{-l(c)} \le 1$$

Also, for any set of lengths L such that

$$\sum_{l \in L} 2^{-l} \le 1$$

there is a prefix code C such that

$$l(c_i) = l_i \; (i = 1, \ldots, |L|)$$

Proof of the Upper Bound (Part 1)

Assign to each message a length $l(s) = \lceil \log(1/p(s)) \rceil$

We then have

$$\begin{aligned} \sum_{s \in S} 2^{-l(s)} &= \sum_{s \in S} 2^{-\lceil \log(1/p(s)) \rceil} \\ &\le \sum_{s \in S} 2^{-\log(1/p(s))} \\ &= \sum_{s \in S} p(s) \\ &= 1 \end{aligned}$$

So by the Kraft-McMillan ineq. there is a prefix code with lengths *l(s)*.

Proof of the Upper Bound (Part 2)

Now we can calculate the average length given *l(s)*

$$\begin{aligned} ABL(S) &= \sum_{s \in S} p(s) l(s) \\ &= \sum_{s \in S} p(s) \cdot \lceil \log(1/p(s)) \rceil \\ &\le \sum_{s \in S} p(s) \cdot (1 + \log(1/p(s))) \\ &= 1 + \sum_{s \in S} p(s) \log(1/p(s)) \\ &= 1 + H(S) \end{aligned}$$

And we are done.

Another property of optimal codes

Theorem: *If C is an optimal prefix code for the probabilities $\{p_1, \ldots, p_n\}$ then $p_i > p_j$ implies $l(c_i) \le l(c_j)$*

Proof: (by contradiction)

Assume $l(c_i) > l(c_j)$. Consider switching codes c_i and c_j. If l_a is the average length of the original code, the length of the new code is

$$\begin{aligned} l_a' &= l_a + p_j(l(c_i) - l(c_j)) + p_i(l(c_j) - l(c_i)) \\ &= l_a + (p_j - p_i)(l(c_i) - l(c_j)) \\ &< l_a \end{aligned}$$

This is a contradiction since l_a was supposed to be optimal

Corollary

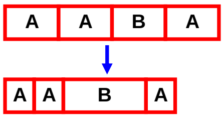
- The p_i is smallest over the code, then $l(c_i)$ is the largest.

Huffman Coding

Binary trees for compression

Huffman Code

- Approach
  - Variable length encoding of symbols
  - Exploit statistical frequency of symbols
  - Efficient when symbol probabilities vary widely
- Principle
  - Use fewer bits to represent frequent symbols
  - Use more bits to represent infrequent symbols



Huffman Codes

Invented by Huffman as a class assignment in 1950.

Used in many, if not most compression algorithms

- gzip, bzip, jpeg (as option), fax compression,…

Properties:

- Generates optimal prefix codes
- Cheap to generate codes
- Cheap to encode and decode
- $l_a = H$ if probabilities are powers of 2

Huffman Code Example

Symbol	Dog	Cat	Bird	Fish
Frequency	1/8	1/4	1/2	1/8
Original Encoding	00	01	10	11
	2 bits	2 bits	2 bits	2 bits
Huffman Encoding	110	10	0	111
	3 bits	2 bits	1 bit	3 bits

- Expected size
  - Original ⇒ 1/8×2 + 1/4×2 + 1/2×2 + 1/8×2 = 2 bits / symbol
  - Huffman ⇒ 1/8×3 + 1/4×2 + 1/2×1 + 1/8×3 = 1.75 bits / symbol

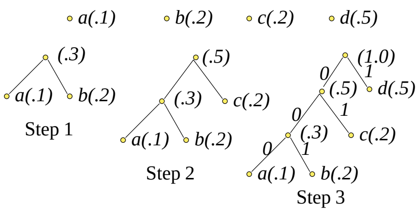
Huffman Codes

Huffman Algorithm

- Start with a forest of trees each consisting of a single vertex corresponding to a message s and with weight $p(s)$
- ***Repeat:***
  - Select two trees with minimum weight roots p_1 and p_2
  - Join into single tree by adding root with weight $p_1 + p_2$

Example

$p(a) = .1,\ p(b) = .2,\ p(c) = .2,\ p(d) = .5$



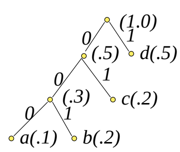
$a=000,\ b=001,\ c=01,\ d=1$

Encoding and Decoding

Encoding: Start at leaf of Huffman tree and follow path to the root. Reverse order of bits and send.

Decoding: Start at root of Huffman tree and take branch for each bit received. When at leaf can output message and return to root.

There are even faster methods that can process 8 or 32 bits at a time



Lemmas

- L1 : Let p_i be the smallest over the code, then $l(c_i)$ is the largest and hence a leaf of the tree. (Let its parent be u)
- L2 : If p_j is second smallest over the code, then $l(c_j)$ is the child of u in the optimal code.
- L3 : There is an optimal prefix code with corresponding tree T*, in which the two lowest frequency letters are siblings.

Huffman codes are optimal

Theorem: *The Huffman algorithm generates an optimal prefix code.*

In other words: It achieves the minimum average number of bits per letter of any prefix code.

Proof: By induction

Base Case: Trivial (one bit optimal)

Assumption: The method is optimal for all alphabets of size k-1.

Proof:

- Let y* and z* be the two lowest frequency letters merged in w*. Let T be the tree before merging and T' after merging.
- Then : ABL(T') = ABL(T) – p(w*)
- T' is optimal by induction.

Proof:

- Let Z be a better tree compared to T produced using Huffman's alg.
- Implies ABL(Z) < ABL(T)
- By lemma L3, there is such a tree Z' in which the leaves representing y* and z* are siblings (and has same ABL as Z).
- By previous page ABL(Z') =ABL(Z) – p(w*)
- Contradiction!

Adaptive Huffman Codes

Huffman codes can be made to be adaptive without completely recalculating the tree on each step.

- Can account for changing probabilities
- Small changes in probability, typically make small changes to the Huffman tree

Used frequently in practice

Huffman Coding Disadvantages

- Integral number of bits in each code.
- If the entropy of a given character is 2.2 bits,the Huffman code for that character must be either 2 or 3 bits , not 2.2.

Towards Arithmetic coding

- An Example: Consider sending a message of length 1000 each with having probability .999
- Self information of each message

  -log(.999)= .00144 bits
- Sum of self information = 1.4 bits.
- Huffman coding will take at least 1k bits.
- Arithmetic coding = 3 bits!

<u>Arithmetic Coding: Introduction</u>

Allows "blending" of bits in a message sequence.

Can bound total bits required based on sum of self information:

$$l < 2 + \sum_{i=1}^{n} s_i$$

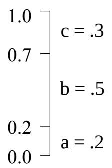
Used in PPM, JPEG/MPEG (as option), DMM

More expensive than Huffman coding, but integer implementation is not too bad.

Arithmetic Coding (message intervals)

Assign each probability distribution to an interval range from 0 (inclusive) to 1 (exclusive).

e.g.



$$f(i) = \sum_{j=1}^{i-1} p(j)$$

$f(a) = .0, \quad f(b) = .2, \quad f(c) = .7$

The interval for a particular message will be called the **<u>message interval</u>** (e.g for b the interval is [.2,.7))

Arithmetic Coding (sequence intervals)

To code a message use the following:

$$l_1 = f_1 \quad l_i = l_{i-1} + s_{i-1} f_i$$
$$s_1 = p_1 \quad s_i = s_{i-1} p_i$$

Each message narrows the interval by a factor of p_i.

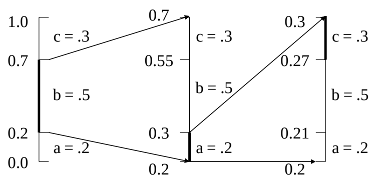
Final interval size:

$$s_n = \prod_{i=1}^{n} p_i$$

The interval for a message sequence will be called the **<u>sequence interval</u>**

Arithmetic Coding: Encoding Example

Coding the message sequence: **bac**



The final interval is **[.27,.3)**

Uniquely defining an interval

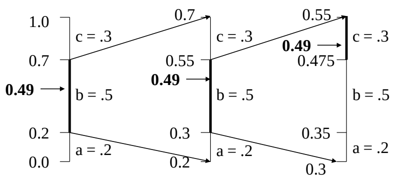
Important property: The sequence intervals for distinct message sequences of length *n* will never overlap

Therefore: specifying <u>any number in the final interval</u> uniquely determines the sequence.

Decoding is similar to encoding, but on each step need to determine what the message value is and then reduce interval

Arithmetic Coding: Decoding Example

Decoding the number .49, knowing the message is of length 3:



The message is **bbc**.

RealArith Encoding and Decoding

RealArithEncode:

- Determine *l* and *s* using original recurrences
- Code using *l + s/2* truncated to $1+\lceil -\log s \rceil$ bits

RealArithDecode:

- Read bits as needed so code interval falls within a message interval, and then narrow sequence interval.
- Repeat until *n* messages have been decoded .

Bound on Length

***Theorem:** For n messages with self information $\{s_1,\ldots,s_n\}$* RealArithEncode *will generate at most*

$$2+\sum_{i=1}^{n} s_i$$ *bits.*

$$\begin{aligned}
1+\lceil -\log s \rceil &= 1+\left\lceil -\log\left(\prod_{i=1}^{n} p_i\right)\right\rceil \\
&= 1+\left\lceil \sum_{i=1}^{n} -\log p_i \right\rceil \\
&= 1+\left\lceil \sum_{i=1}^{n} s_i \right\rceil \\
&< 2+\sum_{i=1}^{n} s_i
\end{aligned}$$

Applications of Probability Coding

How do we generate the probabilities?

Using character frequencies directly does not work very well (e.g. 4.5 bits/char for text).

Technique 1: transforming the data

- Run length coding (ITU Fax standard)
- Move-to-front coding (Used in Burrows-Wheeler)
- Residual coding (JPEG LS)

Technique 2: using conditional probabilities

- Fixed context (JBIG…almost)
- Partial matching (PPM)

Run Length Coding

Code by specifying message value followed by number of repeated values:

e.g. **abbbaacccca => (a,1),(b,3),(a,2),(c,4),(a,1)**

The characters and counts can be coded based on frequency.

This allows for small number of bits overhead for low counts such as 1.

Facsimile ITU T4 (Group 3)

Standard used by all home Fax Machines

ITU = International Telecommunications Standard

Run length encodes sequences of black+white pixels

Fixed Huffman Code for all documents. e.g.

Run length	White	Black
1	000111	010
2	0111	11
10	00111	0000100

Since alternate black and white, no need for values.

Move to Front Coding

Transforms message sequence into sequence of integers, that can then be probability coded

Start with values in a total order:
e.g.: [a,b,c,d,e,….]

For each message output position in the order and then move to the front of the order.
e.g.: c => output: 3, new order: [c,a,b,d,e,…]
a => output: 2, new order: [a,c,b,d,e,…]

Codes well if there are concentrations of message values in the message sequence.

Residual Coding

Used for message values with meaningfull order
e.g. integers or floats.

Basic Idea: guess next value based on current context. Output difference between guess and actual value. Use probability code on the output.

JPEG-LS

JPEG Lossless (not to be confused with lossless JPEG)
Just completed standardization process.

Codes in Raster Order. Uses 4 pixels as context:

NW	N	NE
W	*	

Tries to guess value of * based on W, NW, N and NE.

Works in two stages

JPEG LS: Stage 1

Uses the following equation:

$$P = \begin{cases} \min(N,W) & \text{if } NW \geq \max(N,W) \\ \max(N,W) & \text{if } NW < \min(N,W) \\ N + W - NW & \text{otherwise} \end{cases}$$

Averages neighbors and captures edges. e.g.

40	3	*
40	3	

30	40	*
20	30	

3	3	*
40	40	

JPEG LS: Stage 2

Uses 3 gradients: W-NW, NW-N, N-NE

- Classifies each into one of 9 categories.
- This gives 9^3=729 contexts, of which only 365 are needed because of symmetry.
- Each context has a bias term that is used to adjust the previous prediction

After correction, the residual between guessed and actual value is found and coded using a Golomblike code.

Using Conditional Probabilities: PPM

Use previous *k* characters as the context.

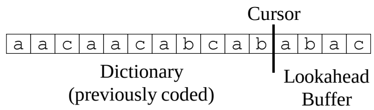
Base probabilities on counts:
e.g. if seen **th** 12 times followed by **e** 7 times, then the conditional probability ***p(e|th)***=7/12.

Need to keep *k* small so that dictionary does not get too large.

Ideas in Lossless compression

- That we did not talk about specifically
  - Lempel-Ziv (gzip)
    - Tries to guess next window from previous data
  - Burrows-Wheeler (bzip)
    - Context sensitive sorting
    - Block sorting transform

LZ77: Sliding Window Lempel-Ziv

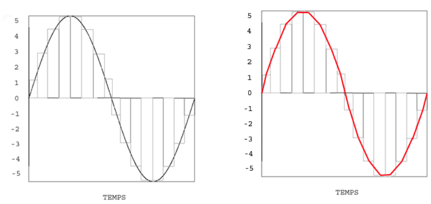


Dictionary and buffer "windows" are fixed length and slide with the cursor

On each step:

- Output (p,l,c)
  p = relative position of the longest match in the dictionary
  l = length of longest match
  c = next char in buffer beyond longest match
- Advance window by l + 1

Lossy compression



Scalar Quatization

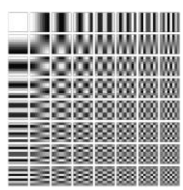
- Given a camera image with 12bit color, make it 4-bit grey scale.
- Uniform Vs Non-Uniform Quantization
  - The eye is more sensitive to low values of red compared to high values.

Vector Quantization

- How do we compress a color image (r,g,b)?
  - Find k – representative points for all colors
  - For every pixel, output the nearest representative
  - If the points are clustered around the representatives, the residuals are small and hence probability coding will work well.

Transform coding

- Transform input into another space.
- One form of transform is to choose a set of basis functions.
- JPEG/MPEG both use this idea.



Other Transform codes

- Wavelets
- Fractal base compression
  - Based on the idea of fixed points of functions.