15-499:Algorithms and Applications

Data Compression I and II
- –Introduction
- –Information Theory
- –Probability Coding
- –Applications of Probability Coding

Compression in the Real World

Generic File Compression
- **Files**: gzip (LZ77), bzip (Burrows-Wheeler), BOA (PPM)
- **Archivers**: ARC (LZW), PKZip (LZW+)
- **File systems**: NTFS

Communication
- **Fax**: ITU-T Group 3 (run-length + Huffman)
- **Modems**: V.42bis protocol (LZW), MNP5 (run-length+Huffman)
- **Virtual Connections**

Compression in the Real World

Multimedia
- **Images**: gif (LZW), jbig (context), jpeg-ls (residual), jpeg (transform+RL+arithmetic)
- **TV**: HDTV (mpeg-4)
- **Sound**: mp3

An example

Other structures
- **Indexes**: google, lycos
- **Meshes (for graphics)**: edgebreaker
- **Databases**:

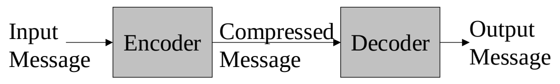
Compression Outline

➡ **Introduction**:
- Lossless vs. lossy
- Model and coder
- Benchmarks

Information Theory: Entropy, etc.
Probability Coding: Huffman + Arithmetic Coding
Applications of Probability Coding: PPM + others
Lempel-Ziv Algorithms: LZ77, gzip, compress, ...
Other Lossless Algorithms: Burrows-Wheeler
Lossy algorithms for images: JPEG, MPEG, ...
Compressing graphs and meshes: BBK

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.

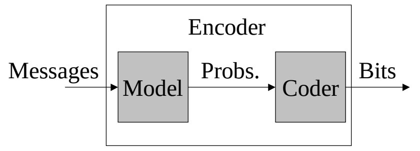
Writing is the art of lossy text compression.

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

e.g. 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
RK	LZ + PPM	111+115	1.79	430
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	?

Compression Outline

Introduction: Lossy vs. Lossless, Benchmarks, …

Information Theory:
- Entropy
- Conditional Entropy
- Entropy of the English Language

Probability Coding: Huffman + Arithmetic Coding

Applications of Probability Coding: PPM + others

Lempel-Ziv Algorithms: LZ77, gzip, compress, …

Other Lossless Algorithms: Burrows-Wheeler

Lossy algorithms for images: JPEG, MPEG, …

Compressing graphs and meshes: BBK

Information Theory

An interface between modeling and coding

Entropy
- A measure of information content

Conditional Entropy
- Information content based on a context

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 ∈ 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

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

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

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

Conditional Entropy

The **conditional probability** *p(s|c)* is the probability of *s* in a context *c*. The **conditional self information** is

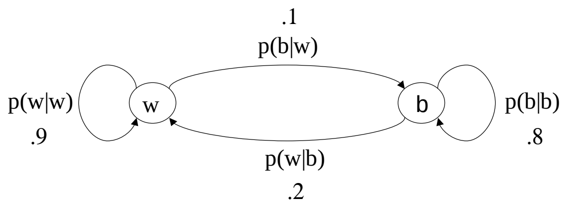
$$i(s|c) = -\log p(s|c)$$

The conditional information can be either more or less than the unconditional information.

The **conditional entropy** is the weighted average of the conditional self information

$$H(S\mid C) = \sum_{c\in C}\left(p(c)\sum_{s\in S} p(s\mid c)\log\frac{1}{p(s\mid c)}\right)$$

Example of a Markov Chain



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
- RK = 1.79 (current close to best text compressor)

Must be less than 1.79.

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	4	5	> 5
Probability	.79	.08	.03	.02	.02	.05

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

Compression Outline

Introduction: Lossy vs. Lossless, Benchmarks, …

Information Theory: Entropy, etc.

Probability Coding:
- Prefix codes and relationship to Entropy
- Huffman codes
- Arithmetic codes

Applications of Probability Coding: PPM + others

Lempel-Ziv Algorithms: LZ77, gzip, compress, …

Other Lossless Algorithms: Burrows-Wheeler

Lossy algorithms for images: JPEG, MPEG, …

Compressing graphs and meshes: BBK

Assumptions and Definitions

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

Each message come from a **message set** $S = \{s_1,\ldots,s_n\}$ with a **probability distribution** $p(s)$. Probabilities must sum to 1. Set can be infinite.

Code $C(s)$: A mapping from a message set to **codewords**, each of which is a string of bits

Message sequence: a sequence of messages

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

Discrete or Blended

We will consider two types of coding:

Discrete: each message is a fixed set of bits

- Huffman coding, Shannon-Fano coding

`01001` `11` `0001` `011`

message: 1 2 3 4

Blended: bits can be "shared" among messages

- Arithmetic coding

`010010111010`

message: 1,2,3, and 4

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

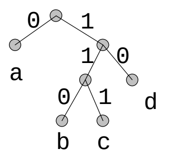
A **prefix code** is a variable length code in which no codeword is a prefix of another word.

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

All prefix codes are uniquely decodable

Prefix Codes: as a tree

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



a = 0, b = 110, c = 111, d = 10

Some Prefix Codes for Integers

n	Binary	Unary	Gamma
1	..001	0	0|
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 Length

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

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

We say that a prefix code *C* is **optimal** if for all prefix codes *C'*, $l_a(C) \leq l_a(C')$

l(c) = length of the codeword *c* (a positive integer)

Relationship to Entropy

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

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

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

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

Kraft McMillan Inequality

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

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

Also, for any set of lengths L such that

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

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 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 inequality 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)*

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

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 is not optimal

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 Codes

Huffman Algorithm:

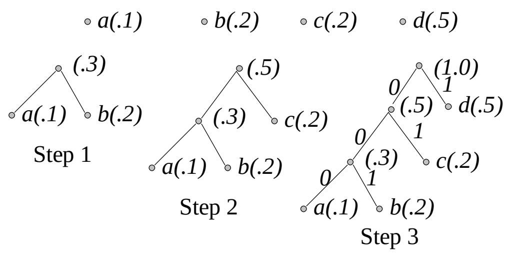
Start with a forest of trees each consisting of a single vertex corresponding to a message *s* and with weight *p(s)*

Repeat until one tree left:

- 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



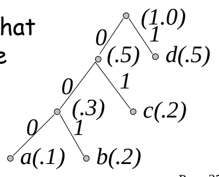
a=000, b=001, c=01, d=1

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



Huffman codes are "optimal"

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

Proof outline:

Induction on the number of messages n.

Consider a message set S with n+1 messages

1. Can make it so least probable messages of S are neighbors in the Huffman tree
2. Replace the two messages with one message with probability $p(m_1) + p(m_2)$ making *S'*
3. Show that if S' is optimal, then S is optimal
4. S' is optimal by induction

Problem with Huffman Coding

Consider a message with probability .999. The self information of this message is

$$-\log(.999) = .00144$$

If we were to send a 1000 such message we might hope to use 1000*.0014 = 1.44 bits.

Using Huffman codes we require at least one bit per message, so we would require 1000 bits.

Arithmetic Coding: Introduction

Allows “blending” of bits in a message sequence. Only requires 3 bits for the example

Can bound total bits required based on sum of self information: $l < 2 + \sum_{i=1}^{n} s_i$

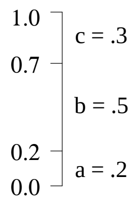
Used in PPM, JPEG/MPEG (as option), DMM

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.



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

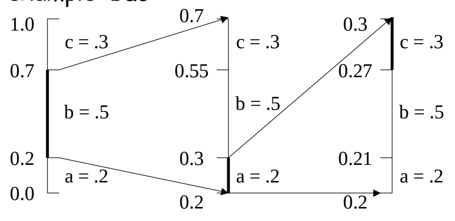
$$f(a) = .0, \quad f(b) = .2, \quad f(c) = .7$$

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

Arithmetic Coding: sequence intervals

Code a message sequence by composing intervals.

For example: **bac**



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

We call this the **sequence interval**

Arithmetic Coding: sequence intervals

To code a sequence of messages with probabilities p_i ($i = 1..n$) 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: $s_n = \prod_{i=1}^{n} p_i$

Warning

Three types of interval:

- **message interval** : interval for a single message
- **sequence interval** : composition of message intervals
- **code interval** : interval for a specific code used to represent a sequence interval (discussed later)

Uniquely defining an interval

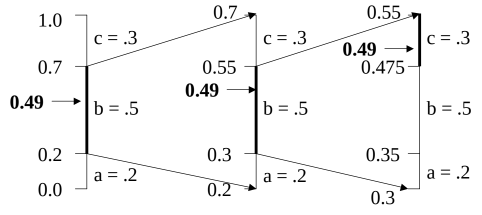
Important property: The sequence intervals for distinct message sequences of length *n* will never overlap

Therefore: specifying any number in the final interval 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.**

Representing Fractions

Binary fractional representation:

$$\begin{aligned} .75 &= .11 \\ 1/3 &= .01\overline{01} \\ 11/16 &= .1011 \end{aligned}$$

So how about just using the smallest binary fractional representation in the sequence interval. e.g. [0,.33) = .01 [.33,.66) = .1 [.66,1) = .11

But what if you receive a 1?

Should we wait for another 1?

Representing an Interval

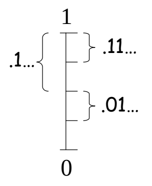
Can view binary fractional numbers as intervals by considering all completions. e.g.

	min	max	interval
.11	$.11\overline{0}$	$.11\overline{1}$	[.75,1.0)
.101	$.101\overline{0}$	$.101\overline{1}$	[.625,.75)

We will call this the **code interval.**

Code Intervals: example

[0,.33) = .01 [.33,.66) = .1 [.66,1) = .11

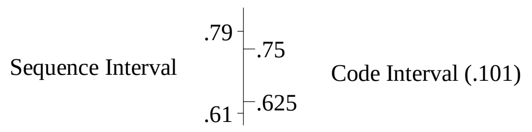


Note that if code intervals overlap then one code is a prefix of the other.

***Lemma**: If a set of code intervals do not overlap then the corresponding codes form a prefix code.*

Selecting the Code Interval

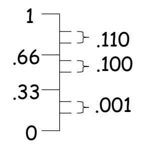
To find a prefix code find a binary fractional number whose code interval is contained in the sequence interval.



Can use the fraction *l + s/2* truncated to $\lceil -\log(s/2) \rceil = 1 + \lceil -\log s \rceil$ bits

Selecting a code interval: example

[0,.33) = .001 [.33,.66) = .100 [.66,1) = .110



e.g: for [.33…,.66…), *l = .33…, s = .33…*

l + s/2 = .5 = .1000…

truncated to $1+\lceil -\log s \rceil = 1+\lceil -\log(.33) \rceil = 3$ bits is .100

Is this the best we can do for [0,.33) ?

RealArith Encoding and Decoding

RealArithEncode:

Determine *l* and *s* using original recurrences

Code using *l + s/2* truncated to 1+⌈-log *s*⌉ 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 {$s_1,…,s_n$}* RealArithEncode *will generate at most* $2+\sum_{i=1}^{n} s_i$ *bits.*

Proof:

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

Integer Arithmetic Coding

Problem with RealArithCode is that operations on arbitrary precision real numbers is expensive.

Key Ideas of integer version:

Keep integers in range [0..R) where R=2^k

Use rounding to generate integer sequence interval

Whenever sequence interval falls into top, bottom or middle half, expand the interval by factor of 2

This integer Algorithm is an approximation or the real algorithm.

Integer Arithmetic Coding

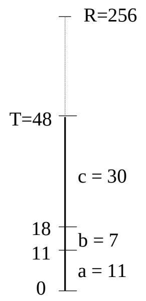
The probability distribution as integers

Probabilities as counts:
  e.g. c(a) = 11, c(b) = 7, c(c) = 30

T is the sum of counts
  e.g. 48 (11+7+30)

Partial sums f as before:
  e.g. f(a) = 0, f(b) = 11, f(c) = 18

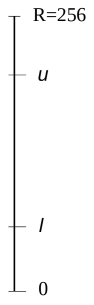
Require that R > 4T so that probabilities do not get rounded to zero



Integer Arithmetic (contracting)

l_1 = 0, s_1 = R

$$\begin{aligned} s_i &= u_i - l_i + 1 \\ u_i &= l_i + \lfloor s_i \cdot (f_i + s_i)/T \rfloor - 1 \\ l_i &= l_i + \lfloor s_i \cdot f_i / T \rfloor \end{aligned}$$



Integer Arithmetic (scaling)

If *l* ≥ *R/2* then (**in top half**)
  Output 1 followed by *m* 0s
  m = 0
  Scale message interval by expanding by 2

If *u* < *R/2* then (**in bottom half**)
  Output 0 followed by *m* 1s
  m = 0
  Scale message interval by expanding by 2

If *l* ≥ *R/4* and *u* < *3R/4* then (**in middle half)**
  Increment m
  Scale message interval by expanding by 2

Summary so far

Model generates probabilities, ***Coder*** uses them

Probabilities are related to **information**. The more you know, the less info a message will give.

More "skew" in probabilities gives lower **Entropy H** and therefore better compression

Context can help "skew" probabilities (lower H)

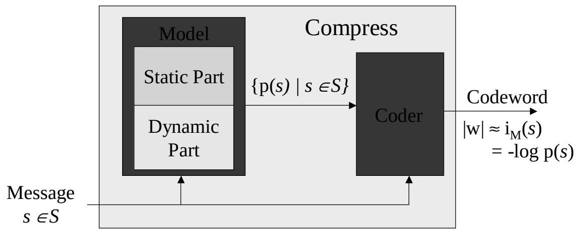
Average length l_a for **optimal prefix code** bound by

$$H \le l_a < H + 1$$

Huffman codes are optimal prefix codes

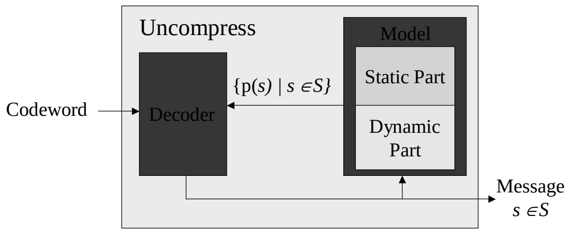
Arithmetic codes allow "blending" among messages

Encoding: Model and Coder



The **Static part** of the model is fixed
The **Dynamic part** is based on previous messages
The "optimality" of the code is relative to the probabilities.
If they are not accurate, the code is not going to be efficient

Decoding: Model and Decoder



The **probabilities** $\{p(s) \mid s \in S\}$ generated by the model need to be the same as generated in the encoder.
Note: consecutive "messages" can be from a different message sets, and the probability distribution can change

Codes with Dynamic Probabilities

Huffman codes:

Need to generate a new tree for new probabilities.
Small changes in probability, typically make small changes to the Huffman tree.
"**Adaptive Huffman codes"** update the tree without having to completely recalculate it.
Used frequently in practice

Arithmetic codes:

Need to recalculate the f(m) values based on current probabilities.
Can be done with a balanced tree.

Compression Outline

Introduction: Lossy vs. Lossless, Benchmarks, …
Information Theory: Entropy, etc.
Probability Coding: Huffman + Arithmetic Coding
➡ **Applications of Probability Coding**: PPM + others
- Transform coding: move to front, run-length, …
- Context coding: fixed context, partial matching

Lempel-Ziv Algorithms: LZ77, gzip, compress, …
Other Lossless Algorithms: Burrows-Wheeler
Lossy algorithms for images: JPEG, MPEG, …
Compressing graphs and meshes: BBK

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 the 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

Takes advantage of **temporal locality**

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

For each message

- output the position in the order
- move to the front of the order.

e.g.: b => output: 3, new order: [c,a,b,d,e,…]
a => output: 2, new order: [a,c,b,d,e,…]

Probability code the output.

The hope is that there is a bias for small numbers.

Residual Coding

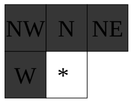
Typically used for message values that represent some sort of amplitude:
e.g. gray-level in an image, or amplitude in audio.

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:



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

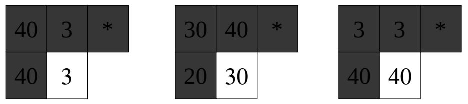
Works in two stages

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.



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 Golomb-like code. (Golomb codes are similar to Gamma codes)

Using Conditional Probabilities: PPM

Use previous *k* characters as the context.
- Makes use of conditional probabilities

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 (typically less than 8).

Note that 8-gram Entropy of English is ≅2.3bits/char while PPM does as well as 1.7bits/char

PPM: Partial Matching

Problem: What do we do if we have not seen the context followed by the character before?
- Cannot code 0 probabilities!

The key idea of PPM is to reduce context size if previous match has not been seen.
- If character has not been seen before with current context of size 3, try context of size 2, and then context of size 1, and then no context

Keep statistics for each context size < *k*

PPM: Changing between context

How do we tell the decoder to use a smaller context?

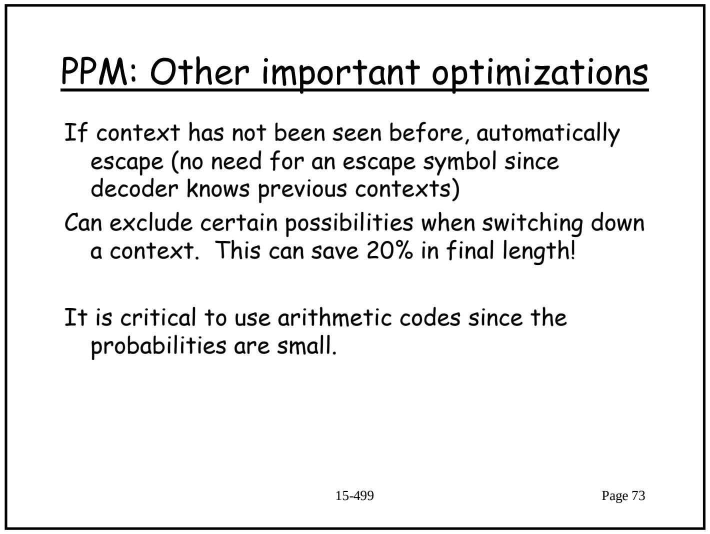
Send an **escape** message. Each escape tells the decoder to reduce the size of the context by 1.

The escape can be viewed as special character, but needs to be assigned a probability.
- Different variants of PPM use different heuristics for the probability.

PPM: Example Contexts

Context	Counts	Context	Counts	Context	Counts
Empty	A = 4	A	C = 3	AC	B = 1
	B = 2		$ = 1		C = 2
	C = 5	B	A = 2		$ = 2
	$ = 3		$ = 1	BA	C = 1
		C	A = 1		$ = 1
			B = 2	CA	A = 1
			C = 2		$ = 1
			$ = 3	CB	A = 2
					$ = 1
				CC	A = 1
					B = 1
					$ = 2

String = ACCBACCACBA k = 2



PPM: Other important optimizations

If context has not been seen before, automatically escape (no need for an escape symbol since decoder knows previous contexts)

Can exclude certain possibilities when switching down a context. This can save 20% in final length!

It is critical to use arithmetic codes since the probabilities are small.

15-499 Page 73