ECE 8873
Data Compression and Modeling

Lecture 4:
Lossless Compression – Arithmetic Code and Dictionary Techniques

School of Electrical and Computer Engineering
Georgia Institute of Technology
Spring, 2004

Arithmetic Coding

- A skewed source often has low entropy.
- A source with small alphabet also has low entropy in most practical situations.
- Huffman codes are inefficient for skewed sources; its deviation from optimality depends on $P_{\max}$, maximum of symbol probabilities.
- Extended Huffman code may improve efficiency, but implementation requires large table.
- Is it possible to sequentially construct only the part of the table that is needed?

Arithmetic Code

- Key mechanism:
  - Code k-tuples of input symbols.
  - For each k-tuple input, generate a tag, a real number in (0,1), according the probabilities in the alphabet; the tag can be generated sequentially.
  - Represent each tag in binary code with length commensurate (inversely) with the symbol's probability; the binary code can be truncated without compromising the decodability due to the tag structure.

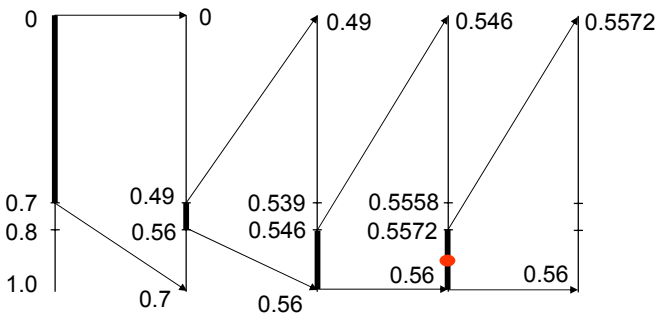
Let $A = \{a_1, a_2, \cdots, a_m\}$, and $X(a_i) = i$

$P(X = i) = P(a_i)$ and $F_X(i) = \sum_{k=1}^{i} P(X = i)$

Mapping a Symbol Sequence into a Tag

Let $A = \{a_1, a_2, a_3\}$, and $P(a_1) = 0.7, P(a_2) = 0.1, P(a_3) = 0.2$

For sequence (a_1, a_2, a_3, a_3), $T[(a_1, a_2, a_3, a_3)] = 0.5586$



Generating and Deciphering the Tag

Let $l^{(n)}$ and $u^{(n)}$ be the lower and upper bound of the tag associated with a sequence of n symbols, $\mathbf{x} = (x_1, x_2, \cdots, x_n)$

These bounds can be computed via the following recursion:

$$l^{(n)} = l^{(n-1)} + (u^{(n-1)} - l^{(n-1)})F_X(x_n - 1)$$
$$u^{(n)} = u^{(n-1)} + (u^{(n-1)} - l^{(n-1)})F_X(x_n)$$

with $l^{(0)} = 0$ and $u^{(0)} = 1$

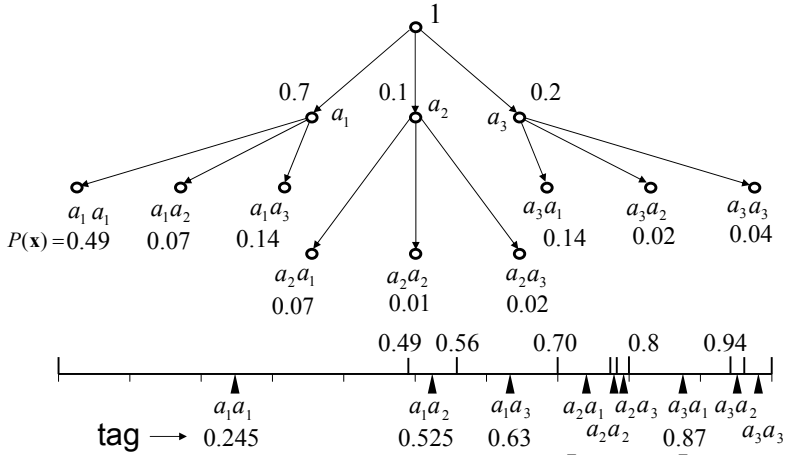
The same recursion is used to convert a tag back into the sequence of symbols.

Need to know the cumulative probabilities, $F(x)$.

The Deciphering Algorithm

1. Initialize $l^{(0)} = 0$ and $u^{(0)} = 1$;
2. For each k, $k = 1, 2, \cdots, n$ find
$$t^* = (\text{tag} - l^{(k-1)})/(u^{(k-1)} - l^{(k-1)})$$
3. Find the value of x_k for which $F_X(x_k - 1) \le t^* < F_X(x_k)$
4. Update $l^{(k)}$ and $u^{(k)}$
5. Repeat steps 2-4 until the entire sequence is found.

Another Way to Look at Arithmetic Code



Use binary format for tag but truncated it to $l(\mathbf{x}) = \lceil -\log P(\mathbf{x}) \rceil + 1$ bits.

The truncation while causes deviation from the original tag value will not however move the value out of the range the tag was originally in, thereby maintaining the decodability.

Binary Code Assignment

Each tag in [0,1) is represented by a binary fractional number and truncated to $l(\mathbf{x}) = \lceil -\log P(\mathbf{x}) \rceil + 1$ bits.

This binary code is uniquely decodable because the truncated tag always remains in $[F_X(\mathbf{x}-1), F_X(\mathbf{x}))$ and it is a prefix code.

Symbol	F_X	T_X	In Binary	$\lceil -\log P(x) \rceil + 1$	Code
1	.500	.25	.010	2	01
2	.750	.625	.101	3	101
3	.875	.8125	.1101	4	1101
4	1.000	.9375	.1111	4	1111

Average code length: $H(X) \le I_A \le H(X) + \frac{2}{m}$, $m = \text{block length}$

Efficiency of Arithmetic Code

- Sayood's derivation on p.90-91.

$$
\begin{aligned}
l_{A^m} &= \sum P(\mathbf{x}) l(\mathbf{x}) = \sum P(\mathbf{x}) \left[\left\lceil \log \frac{1}{P(\mathbf{x})} \right\rceil + 1 \right] \\
&\le \sum P(\mathbf{x}) \left[\log \frac{1}{P(\mathbf{x})} + 1 + 1 \right] \\
&= -\sum P(\mathbf{x}) \log P(\mathbf{x}) + 2 \sum P(\mathbf{x}) \\
&= H(X^m) + 2
\end{aligned}
$$

What's wrong with this derivation? Or, what has been assumed or implied in the derivation? What has been compromised by the algorithm?

Comparison of Huffman and Arithmetic Codes

1. Code efficiency:

$$\text{Huffman}: \quad H(X) \le I_A \le H(X) + \frac{1}{m}, \quad m = \text{block length}$$

$$\text{Arithmetic}: \quad H(X) \le I_A \le H(X) + \frac{2}{m}, \quad m = \text{block length}$$

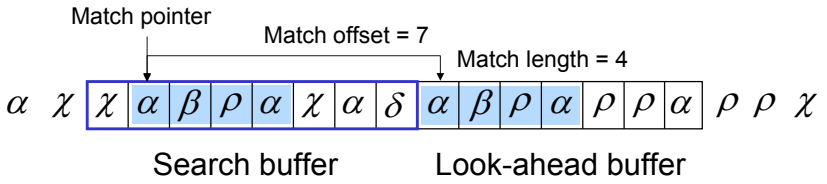
Be careful about the entropy issue.

2. Implementation efficiency:
   Huffman: needs to build entire code table first; code table may be large (as m grows for higher code efficiency)
   Arithmetic: can be implemented incrementally and sequentially; no need to store a pre-fab code table; suitable for large m. (But remember the independence issue.)

Dictionary Techniques

- Variable length coding; recall Tunstall codes.
- Consider something even simpler – the 4-letter example in Sayood.
  - Under a certain condition, the benefit outweighs the overhead bit (for code parsing).
- Dictionary can be
  - Static: when source is (relatively) strong mixing and sufficient prior knowledge is available; example – digram or bigram, trigram, …
  - Dynamic: when source is not rapidly changing (with obvious local behavior) and the change is adaptable.

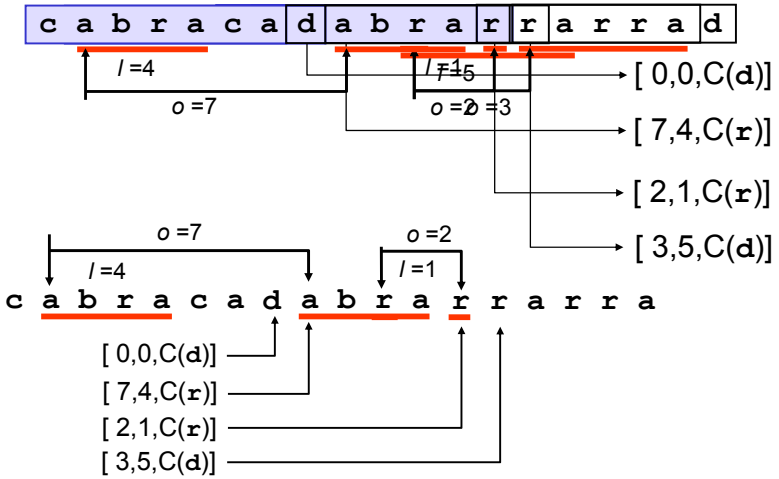
The Ziv-Lempel 77 Algorithm



When a match is found, the encoder sends a triple <*o, l, c*> where *o* is the match offset (=7), *l* is the match length (=4), and *c* is the character in look-ahead buffer that follows the match (= ρ).

What if there is no match?
What if the match extends beyond search buffer?

LZ77 Example



LZ77

- Asymptotic optimality: approached entropy.
- Recurrence of codewords happens in recent memory.
- Variations:
  - Encode the triple with variable length code; PKZip, Zip, Lharc, PNG, gzip, and ARJ
  - Improved buffer search algorithm; hash table,…
  - Use a flag bit in the case of no match instead of the original triple.

LZ78

- Avoid performance dependency on the buffer length as in LZ77.
- Codebook may grow unbounded unless constrained.

Input sequence:

*wabba_wabba_wabba_wabba_woo_*

	Dictionary	
Encoded output	Index	Entry
<0,C(*w*)>	1	*w*
<0,C(*a*)>	2	*a*
<0,C(*b*)>	3	*b*
<3,C(*a*)>	4	*ba*
<0,C(_)>	5	_
<1,C(*a*)>	6	*wa*
<3,C(*b*)>	7	*bb*
<2,C(_)>	8	*a_*
<6,C(*b*)>	9	*wab*
<4,C(_)>	10	*ba_*
<9,C(*b*)>	11	*wabb*
<8,C(*w*)>	12	*a_w*
<0,C(*o*)>	13	*o*
<13,C(_)>	14	*o_*

Initial codebook

The Lempel-Ziv-Welch (LZW) Algorithm

- 2nd element in code not transmitted. Only index is sent.
- If p is in dictionary but p*a is not, augment the dictionary with p*a.

wabba_wabba_wabba_wabba_woo_woo_woo

Initial primed dictionary

index	entry
1	_
2	*a*
3	*b*
4	*o*
5	*w*

Encoded sequence:

52332168(10)(12)9(11)7(16)5 44(11)(21)(23)4

index	entry	index	entry
1	_	14	*a_w*
2	*a*	15	*wabb*
3	*b*	16	*ba_*
4	*o*	17	*_wa*
5	*w*	18	*abb*
6	*wa*	19	*ba_w*
7	*ab*	20	*wo*
8	*bb*	21	*oo*
9	*ba*	22	*o_*
10	*a_*	23	*_wo*
11	*_w*	24	*oo_*
12	*wab*	25	*_woo*
13	*bba*		

Predictive Coding

- Symbols or letters in the sequence may have recursive dependency.
- Conventional variable length coding may not fully capture this type of redundancy.
- Instead of coding each symbol in a memoryless fashion (even when sophisticated parsing is involved), predict the symbol based on information that the decoder (also) possesses and can use, obtain the discrepancy between the input and the predicted one, and code such discrepancy. (Recall the structure of information and representation of it.)
- Decoder combines the decoded discrepancy with its version of the predicted result to recover the original.
- Cleary & Witten (1984) – predictive coding with partial match

Structure in Information

- Again, work on (or find) the structure first.
- At times, information structure is recursive.

(1 2 5 7 1 3 0 -5 -3 -1 1 -2 -7 -4 -2 1 3 4)

$$y_i = x_i - x_{i-1} - 2$$

(1 -1 1 0 -8 0 -5 -7 0 0 0 -5 -7 -1 0 1 0 -1)

Two options to transform the coding task:

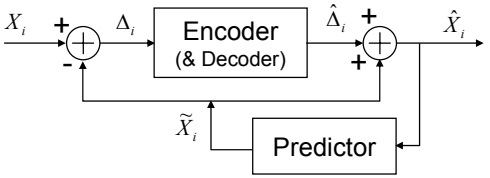
- "Predict" current symbol based on past symbols; code residual

$$\Delta_i = x_i - \tilde{x}_i = x_i - f(x_{i-1}, x_{i-2}, \cdots) = x_i - f(H_i)$$

- "Adapt" probability distribution based on past symbols; i.e., use

$$\{P(x_i \mid x_{i-1}, x_{i-2}, \cdots)\}$$

General Block Diagram



$$\Delta_i = x_i - \tilde{x}_i = x_i - f(\hat{x}_{i-1}, \hat{x}_{i-2}, \cdots) = x_i - f(H_i)$$

$$\hat{\Delta}_i = \beta(\alpha(\Delta_i)) = \Delta_i \quad \text{in case of lossless coding}$$

$$\hat{x}_i = \hat{\Delta}_i + \tilde{x}_i$$

But, the notion of prediction can also be applied to probability measures – foresee change in distribution based on what is or are already observed.

Run Length Encoding

- Simple but still useful.
- Binary (e.g., FAX): Instead of sending 1's and 0's (mostly 1's for white with 0's for black), send length of runs of white.
- Often used for data with many repeated symbols, e.g., FAX and quantized transform coefficients (DCT, wavelet)
- Can combine with other methods, e.g., JPEG does lossless coding (Huffman or arithmetic) of combined quantizer levels/runlengths.

Lossless Compression

- Known source distribution and its use – Huffman, Tunsdall
- Estimate distribution adaptively by recording the frequency on code tree – adaptive Huffman
- Extension to m-tuple code
  - m-tuple Huffman: with or without independence assumption. (What is the difference?)
  - Arithmetic: implicit independence assumption, elimination of exponential growth of codebook; sequential implementation. (Can we do away with independence assumption?)
- Dictionary methods: variable length coding

Other Methods & Remarks

- **Rice machine**
  - multiple codebooks, pick one that does best over some time period (4 Huffman codes in Voyager, 1978-1990; 32 Huffman codes in MPEG-Audio Layer 3).
- **Rissanen Minimum discription length (MDL)**
  - send prefix describing estimated model, followed by optimal encoding of data assuming model.
- **Lempel-Ziv Coding**
  - Inherently universal.
  - Variable numbers of input symbols are required to produce each code symbol – variable length coding.
  - Unlike both Huffman and arithmetic codes, the code does not use any knowledge of the probability distribution of the input. The Ziv-Lempel code achieves the entropy lower bound when applied to a suitably well-behaved source.