Dictionary-based Coding Techniques

National Chiao Tung University
Chun-Jen Tsai
10/16/2014

Rationale

- ☐ In previous two chapters, we looked at coding techniques that assume a source that generates a sequence of independent symbols.
  - Most data sources are correlated, thus, the coding step is generally preceded by a de-correlation step (i.e. model prediction).
- ☐ Alternatively, we can build a list of commonly occurring patterns and encode these patterns by transmitting their index in the list
  → dictionary techniques

Static vs. Adaptive Dictionary

- The dictionary holds a list of strings of symbols and it may be static or dynamic (adaptive)
- Static dictionary – permanent, sometimes allowing the addition of strings but no deletions
- Dynamic dictionary – holding strings previously found in the input stream, allowing for additions and deletions of strings as new input symbols are being read

Basic Idea of Dictionary Coding

- ❑ Given an input source, we want to
  - ■ Identify frequent symbol patterns
  - ■ Encode those more efficiently
  - ■ Use a default (less efficient) encoding for the rest
  - ■ Hopefully, the average bits per symbol gets smaller
- ❑ In general, dictionary-based techniques works well for highly correlated data (e.g. text), but less efficient for data with low correlation (e.g. i.i.d. sources)

Motivating Example

- Consider an ‘English’ source with 26 letters & six punctuation marks
  - Single-symbol FLC, fixed-length encoding: 5 bps
  - Four-symbol FLC, fixed-length encoding: 20 bps (32^4)
- If we assume uneven distribution of the symbols
  - Pick a dictionary witch contains the 256 most-frequent patterns (probability p) and encode them with 8 bits
  - Encode the rest with 20 bits
  - Use 1-bit prefix to distinguish the two cases

then, the average rate is $9p + 21(1 - p) = 21 - 12p$.
If $p > 0.084$, $21 - 12p < 20$.

Static Dictionary

- Using a static dictionary is less complex, but the probability p of a hit highly depends on the applications
  - For student records in a university is probably ok.
- The key for success is that the most common patterns are a small subset of all possible messages
  - Out of over 100,000 English words, only less than 2,000 words are used in most writings

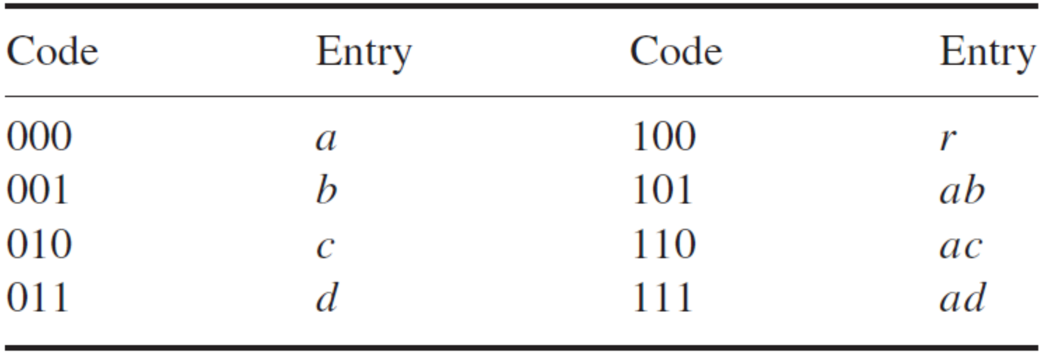
Digram Coding

- The dictionary is composed of
  - All letters from the alphabet
  - As many digrams (pairs of letters) as possible

- For example, if we want to encode pure ASCII text documents, we can design a dictionary of size 256 entries, and
  - Source alphabet: 95 printable ASCII symbols
  - Digrams: 161 most common pairs

Simple Digram Coding Example

- The source alphabet A = {a, b, c, d, r}
- Dictionary:



Code	Entry	Code	Entry
000	*a*	100	*r*
001	*b*	101	*ab*
010	*c*	110	*ac*
011	*d*	111	*ad*

- Try to code the sequence *abracadabra*, the output is 101100110111101100000.

Problem: Which Digrams to Use?

- ❑ Source 1: LaTex documents
- ❑ Source 2: C programs

Pair	Count	Pair	Count
eƀ	1128	ar	314
ƀt	838	at	313
ƀƀ	823	ƀw	309
th	817	te	296
he	712	ƀs	295
in	512	dƀ	272
sƀ	494	ƀo	266
er	433	io	257
ƀa	425	co	256
tƀ	401	re	247
en	392	ƀ$	246
on	385	rƀ	239
nƀ	353	di	230
ti	322	ic	229
ƀi	317	ct	226

Pair	Count	Pair	Count
ƀƀ	5728	st	442
nlƀ	1471	le	440
; nl	1133	ut	440
in	985	f (	416
nt	739	ar	381
= ƀ	687	or	374
ƀi	662	rƀ	373
tƀ	615	en	371
ƀ =	612	er	358
);	558	ri	357
, ƀ	554	at	352
nlnl	506	pr	351
ƀ f	505	te	349
eƀ	500	an	348
ƀ∗	444	lo	347

Adaptive Dictionary Technique

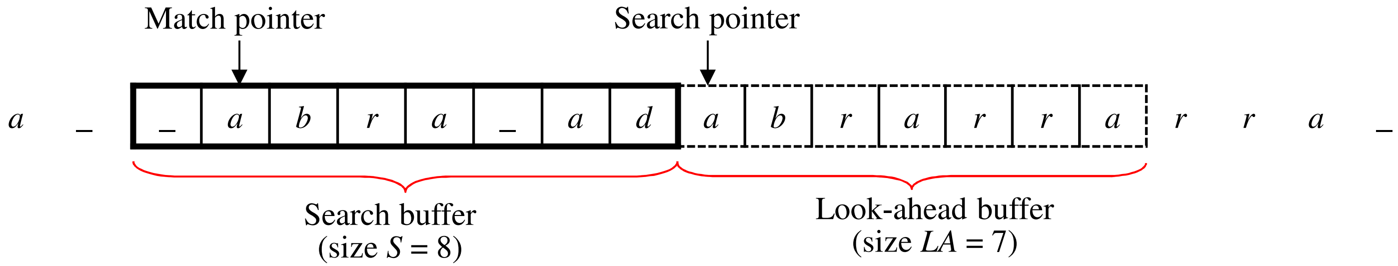
- Original ideas published by Jacob Ziv and Abraham Lempel in 1977 (LZ77/LZ1) and 1978 (LZ78/LZ2)
- The most well-known dictionary-based technique, LZW, is a modification to LZ algorithms published by Terry Welch in 1984

LZ77 (1/2)

- General approach
  - Dictionary is a portion of the previously encoded sequence
  - Use a sliding window for compression
- Mechanism
  - Find the maximum length match for the string pointed to by the search pointer in the search buffer, and encode it
- Rationale
  - If patterns tend to repeat locally, we should be able to get more efficient representation

LZ77 (2/2)

- ☐ Sliding window is composed of a search buffer and a look-ahead buffer (note: window size $W = S + LA$)



- ☐ Offset = search pointer – match pointer ($o = 7$)
- ☐ Length of match = number of consecutive letters matched ($l = 4$)
- ☐ Codeword ($c = C(r)$), where $C(x)$ is the codeword for x
- ☐ Encoding triple: $<o, l, c> = <7, 4, C(r)>$
  - ■ If FLC is used and alphabet size is $|A|$, $<o, l, c>$ can be encoded with $\lceil \log_2 S \rceil + \lceil \log_2 W \rceil + \lceil \log_2 |A| \rceil$ bits.

Possible Cases for Triples

- There could be three different possibilities that may be encountered during the coding process:
  - No match for the next character to be encoded in the window
  - There is a match
  - The matched string extends inside the look-ahead buffer
- For each of these cases, we have a triple to signal the case to the decoder

LZ77 Encoding Example

- Sequence
  - *cabracadabrarrarrad*
  - $W = 13$, $S = 7$
- $|cabraca|\underline{d}abrar|rarrad$
  - no match for *d*
  - send <0, 0, $C(d)$>
- $|abracad|\underline{a}brarr|arrad$

  $|abrac\underline{a}d|\underline{a}brarr|arrad$

  $|abr\underline{a}cad|\underline{a}brarr|arrad$

  $|\underline{abra}cad|\underline{abra}rr|arrad$
  - send <7, 4, $C(r)$>
- $|cadabrar|\underline{r}arrad|$

  $|cadabra\underline{r}|\underline{r}arrad|$

  $|cadab\underline{rar}|\underline{rar}rad|$
  - send <3, 3, $C(r)$>
- Could we do better?
  - send <3, 5, $C(d)$> instead

LZ77 Decoding Example

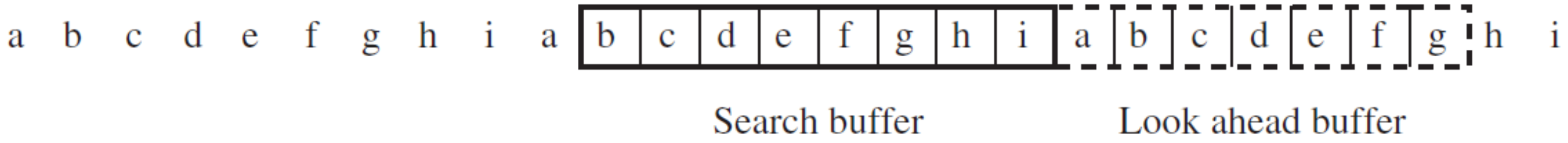
- Current input: $<0, 0, C(d)> <7, 4, C(r)> <3, 5, C(d)>$
- Current output: $cabraca$
- Decode: $<0, 0, C(d)>$
  - Decode $C(d)$: $c|abraca\underline{d}|$
- Decode: $<7, 4, C(r)>$
  - Start with the first 'a', copy four letters: $cabra|cad\underline{abra}|$
  - Decode $C(r)$: $cabrac|adabra\underline{r}|$
- Decode: $<3, 5, C(d)>$
  - Start with the first 'r', copy three letters: $cabracada|brar\underline{rar}|$
  - Copy two more letters: $cabracadabr|arrar\underline{ar}|$
  - Decode $C(d)$: $cabracadabrarrarar\underline{d}$

LZ77 Variants

- ☐ For LZ77, we have
  - Adaptive scheme, no prior knowledge
  - Asymptotically approaches the source statistics
  - Assumes that recurring patterns close to each others
- ☐ Possible improvements
  - Variable-bit encoding: PKZip, zip, gzip, …, etc., uses a variable-length coder to encode $<o, l, c>$.
  - Variable buffer size: larger buffer requires faster searches
  - Elimination of $<0, 0, C(x)>$
    - LZSS sends a flag bit to signal whether the next “token” is an $<o, l>$ pair or the codeword of a symbol

Problems with LZ77

- If the recurring patterns happens with a period larger than the search window, the performance is bad
- Example:



a b c d e f g h i a b c d e f g h i a b c d e f g h i

Search buffer

Look ahead buffer

LZ78

- LZ78 improvements from LZ77
  - No search buffer – explicit dictionary instead
  - Encoder/decoder must build dictionary in sync
  - Encoding: $<i, c>$
    - i = index in the dictionary, $i = 0$ for symbols not in the dictionary
    - c = code of the following character
- Example: encode the following contents
  - *wabbaƀwabbaƀwabbaƀwabbaƀwooƀwooƀwoo*

LZ78 Example

- Input: $wabba\not{b}wabba\not{b}wabba\not{b}wabba\not{b}woo\not{b}woo\not{b}woo$
- Dictionaries:

initial dictionary (empty)

Index	Entry

dictionary after encoding w, a, b

Encoder Output	Index	Entry
$<0, C(w)>$	01	w
$<0, C(a)>$	02	a
$<0, C(b)>$	03	b

final dictionary

Encoder Output	Dictionary Index	Dictionary Entry
$\langle 0, C(w)\rangle$	01	w
$\langle 0, C(a)\rangle$	02	a
$\langle 0, C(b)\rangle$	03	b
$\langle 3, C(a)\rangle$	04	ba
$\langle 0, C(\not{b})\rangle$	05	$\not{b}$
$\langle 1, C(a)\rangle$	06	wa
$\langle 3, C(b)\rangle$	07	bb
$\langle 2, C(\not{b})\rangle$	08	$a\not{b}$
$\langle 6, C(b)\rangle$	09	wab
$\langle 4, C(\not{b})\rangle$	10	$ba\not{b}$
$\langle 9, C(b)\rangle$	11	$wabb$
$\langle 8, C(w)\rangle$	12	$a\not{b}w$
$\langle 0, C(o)\rangle$	13	o
$\langle 13, C(\not{b})\rangle$	14	$o\not{b}$
$\langle 1, C(o)\rangle$	15	wo
$\langle 14, C(w)\rangle$	16	$o\not{b}w$
$\langle 13, C(o)\rangle$	17	oo

Remarks on LZ78

- Observation
  - If we keep on encoding, the dictionary will keep on growing
- Possible solutions
  - Stop growing the dictionary
    - Effectively switch to a static dictionary
  - Prune it
    - Based on usage statistics
  - Reset it
    - Start all over again
- The best solution depends on the knowledge of the source

LZ78 Variants: LZW

- Invented by Terry Welch in 1984
- Idea
  - Instead of <*i*, *c*>, encode *i* only
- Algorithm
  - Initial dictionary contains all alphabet letters, p = null

```
while (!done)
 read next symbol into $a$
 if ($p$*$a$) is in the dictionary    // Note: '*' stands for concatenation
    $p$ = $p$*$a$
 else
    send out index of $p$
    add $p$*$a$ to the dictionary
    $p$ = $a$
end
```

Example: LZW Encoding

- ❑ Input: *wabbaƀwabbaƀwabbaƀwabbaƀwooƀwooƀwoo*
- ❑ Dictionaries:

initial dictionary (source alphabet)

Index	Entry
1	ƀ
2	*a*
3	*b*
4	*o*
5	*w*

final dictionary

Index	Entry	Index	Entry
01	*b̸*	14	*ab̸ w*
02	*a*	15	*wabb*
03	*b*	16	*bab̸*
04	*o*	17	*b̸ wa*
05	*w*	18	*abb*
06	*wa*	19	*bab̸ w*
07	*ab*	20	*wo*
08	*bb*	21	*oo*
09	*ba*	22	*ob̸*
10	*ab̸*	23	*b̸ wo*
11	*b̸ w*	24	*oob̸*
12	*wab*	25	*b̸ woo*
13	*bba*		

- ❑ Output: 5 2 3 3 2 1 6 8 10 12 9 11 7 16 5 4 4 11 21 23 4

Problems with LZW Decoding

- ❑ Decoding of LZW is simple, in general
  - Output symbols from the dictionary as indexed by the inputs
  - Construct the dictionary on-the-fly as the encoder does

- ❑ However, if we have a message pattern $cScS$ …, where c is a character, S is a string, we may run into a situation that the indexed entry is in partial construction
- ❑ Solution: the current dictionary entry under construction is in p, we should allow reading partial data out of p during decoding

Example: Special Case in Decoding

- Alphabet $A = \{a, b\}$, input is $abababab$, encoder output is 1235 ….
- Decoding dictionaries:

initial dictionary

Index	Entry
1	a
2	b

⇒

intermediate dictionary

Index	Entry
1	a
2	b
3	ab
4	ba
5	$a\ldots$

when we reach decoding of 5, $p = ab???$, we do not have the complete output!

Application: Compress

- An early implementation of LZW
- Adaptive dictionary, starts with 2^9 entries
- User can configure max codeword length $b_{max} = 9\sim16$
- Dictionary grows up to double in size
  - When dictionary reaches $2^{b_{max}}$ entries, it becomes a static dictionary encoder
- If compression ratio falls below a threshold, dictionary is reset

Application: GIF Images

- ❑ LZW scheme, similar to compress:
- ❑ Clear code is used to reset the encoder/decoder. For b bits/pixel images, 2^b is used as the clear code
- ❑ Dictionary size is initially 2^{b+1}
- ❑ Dictionary size can grows up to 4096 entries
- ❑ Format:
  - Codewords stored in blocks of 8-bit characters
  - Each block begins with a header with a size count up to 255, and ends with a block terminator symbol (8 zero bits)
  - The last block has a end-of-information code, 2^b+1, before the block terminator

GIF Performance

- GIF vs. arithmetic coding

Image	GIF	Arithmetic Coding of Pixel Values	Arithmetic Coding of Pixel Differences
Sena	51,085	53,431	31,847
Sensin	60,649	58,306	37,126
Earth	34,276	38,248	32,137
Omaha	61,580	56,061	51,393

Application: PNG Images

- ❑ Based on LZ77, patent-free alternative to GIF
- ❑ Designed specifically for lossless image compression
- ❑ Modes: true color, grayscale, 8-bit pallette
- ❑ Two autonomous compression components
  - Deflate (RFC 1951) — LZ77-style dictionary compression algorithm plus Huffman coding
  - Filtering — lossless transformations of byte-level image data

PNG – Deflate

- Deflate = LZ77 + Huffman
- Three types of data blocks
  - Uncompressed, LZ77 + fixed Huffman, LZ77 + adaptive Huffman
- Match length is between 3 and 258 bytes
  - A sliding window of at least 3-byte long is examined
  - If match is not found, encode the first byte and slide window
  - At each step, LZ77 either outputs a codeword for a literal or a paired value of <match_length, offset>
    - Match length is encoded by index code (257~285) and a selector code (0~5 bits)
    - Offset (1~32768) is encoded using Huffman code

PNG – Filtering

- Filters are applied on a scanline-by-scanline basis
- All algorithms applied to bytes (not pixels)
- Filter types:
  - None: unmodified value
  - Sub: difference from previous byte value (mod 256)
  - Up: difference from the byte value above
  - Average: subtract average of the left and the above bytes
  - Paeth:
    - Compute initial estimate by left + above – upper_left
    - The value of left, above, or upper_left that is closest to the initial estimate is used as the estimate

PNG: Performance

- PNG vs. GIF vs. arithmetic coding

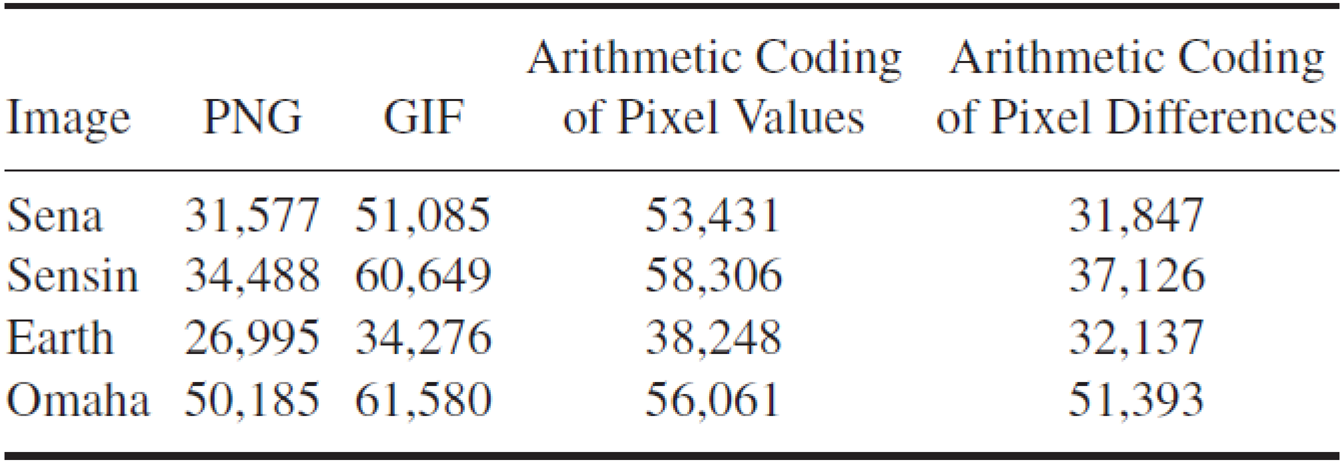


Image	PNG	GIF	Arithmetic Coding of Pixel Values	Arithmetic Coding of Pixel Differences
Sena	31,577	51,085	53,431	31,847
Sensin	34,488	60,649	58,306	37,126
Earth	26,995	34,276	38,248	32,137
Omaha	50,185	61,580	56,061	51,393