Dictionary-Based Coding

(Chapter 5)

Chapter Overview

This chapter is the second that deals with Markov sources. After this chapter you should:

- have understood several algorithms, to the point that:
  - you are able to execute by hand a (compression or decompression) algorithm on a given input.
  - you are able to describe the (compression or decompression) algorithm in your own words, in a reasonably precise manner.
  - you are able to understand and explain, why these algorithms perform well. Your explanation and understanding should convey the intuition in a reasonably precise manner.
  - you should know examples of the use of these algorithms in software products, including an explanation of why they are effective in these application areas.
- The algorithms you should have understood are LZ77 and LZ78 (particularly the LZW variant).

Outline

- Overview of dictionary-based methods.
- LZ77.
- LZ78/LZW.

Dictionary-based coding

Maintain dictionary of frequently-occurring sequences of symbols (called *phrases*) each associated with a number (called *tokens*).

the	1
for	2
ing	3
data	4

$\boxed{\text{for compressing the data}}$

$\downarrow$

$\boxed{\text{2 compress3 1 4}}$

NOTE: Dictionary must be agreed between coder and decoder.

Dictionary coding: issues

- How to choose a dictionary?
  - Fix a dictionary (non-adaptive).
    - The dictionary may not suit the text (e.g. dictionary is for English text but the file is Java code).
  - Choose a dictionary by looking at the entire raw text (semi-adaptive).
    - The dictionary may have to be sent with the file, reducing compression performance.
- The popular choice: *adaptive* algorithms, i.e. start with an empty dictionary and build it as we read the raw text.
- Two popular algorithms: LZ77 and LZW (a variant of LZ78).
  - L = Lempel, Z = Ziv, W = Welch.
- These compress data from Markov sources well almost optimally.

Dictionaries and Markov sources

- The probabilities of sequences of symbols are very different from their memoryless probabilities
- E.g. "the" appears about 1/20th of the time in English text.
- The probablity of "the" according to a MLS is:

$$\Pr[t] \times \Pr[h] \times \Pr[e] = 0.078 \times 0.043 \times 0.113 = 0.00038$$

- Breaking a text into words and applying Huffman coding on the *words* gives much better performance.
- Dictionary-based are superior to Huffword because:
  - They work when the "words" are unknown.
  - Words don't capture all the context information.
  - Files contain patters that are non-Markovian: e.g. first half of file is almost the same as the second half.

LZ77 (setup)

Model: *recently-seen* sequences likely to reappear.
Dictionary: Sliding window of recently-seen input.
LZ77 has two buffers:

- HISTORY buffer of size H symbols.
  - ▷ Text in HISTORY already coded.
- LOOKAHEAD buffer of size L symbols.
  - ▷ Text in LOOKAHEAD buffer to be coded.
- Initialise HISTORY full of blanks (e.g.)
- Before each compression step:
  - HIST will have last H symbols seen.
  - LOOK will have $< L$ symbols which have been seen but not yet encoded.

LZ77 (coding)

1. Read input symbols until $\textsc{Look}$ full.
2. Find a match for some prefix of $\textsc{Look}$ in $\textsc{Hist}$.
3. If match $k \geq 2$ symbols is found with an offset of o bytes then output the token $\langle o, k \rangle$.

   Move matched k symbols in $\textsc{Look}$ into $\textsc{Hist}$, remove the leftmost k symbols from $\textsc{Hist}$.
4. If match of < 2 symbols found, output the token $\langle 0, \mathrm{ASCII}(c) \rangle$. ($c$ is the next symbol in $\textsc{Look}$.)

   Slide $\textsc{Look}$, $\textsc{Hist}$ by 1 symbol.

LZ77 (example)

$H = 16$, $L = 12$.

- Initially HB contains:
  `data␣compression`
- LB is empty.
- Next text is:
  `␣compresses␣data␣and␣it␣is␣my␣favourite␣subject.`

LZ77 example, cont'd

(1) `data_compression` `_compresses_data_and ....`

(2) `data_compression` `_compresses_data_and ....`

(3) `data_compression` `_compresses_data_and ....`

12

9

(4) `ression_compress` `es_data_and_it_is ...`

Output $\langle 12, 9\rangle$, slide by 9.

LZ77 example, cont'd

1. In the second step the situation is:
   HISTORY `ression␣compress`
   LOOKAHD `es␣data␣and␣`
   Output $\langle 3, 2 \rangle$, shift HB by 2.
2. Now the situation is:
   HISTORY `ssion␣compresses`
   LOOKAHD `␣data␣and␣it`
   Output the pointer $\langle 0, \mathrm{ASCII}(\text{' '}) \rangle$, shift HB by 1.

LZ77 Decoding

- Given offset, length, read off the chars from the HB!
- Output these characters, and slide into HB (easy!).

Example:

<12,9><3,2>...

data_compression

12

9

<3,2><0,ASCII(_)>...

ression_compress

LZ77: General Points

- A history buffer is a dictionary. E.g. $H = 4$ and HB contains `abcd`:

phrase	token
`ab`	$\langle 4, 2\rangle$
`bc`	$\langle 3, 2\rangle$
`cd`	$\langle 2, 2\rangle$
`abc`	$\langle 4, 3\rangle$
`bcd`	$\langle 3, 3\rangle$
`abcd`	$\langle 4, 4\rangle$

- Counting offsets from the right gives *smaller* offsets.
    - Frequent strings re-occur sooner rather than later.
- Can code tokens using Elias-γ codes e.g.

LZ77: Parameter Choices

How to choose H and L?

- H large: more likely that matches are found, but the offsets will be larger.
- L small: searching will be quick, size of the match found will be limited.

The algorithm is *asymmetric* since compression is slower than decompression; searching for a match is computationally intensive.

LZ77 applications: QIC-122

- Standard for compressed quarter-inch-cartridge (QIC) tapes.
- Raw character is coded as `0<ASCII code>` (1 byte)
- Pointer is coded as `1<Offset><Length>`, where `Offset` and `Length` may each be chosen from a pre-arranged set of values that add up to 15 bits. E.g. `Length` = 4 bits and `Offset` = 11 bits.
- Many others including `zip` family, see e-lecture.

Outline

- LZW algorithm.
- LZW applications.

The LZW algorithm

- **LZ77 Model:** *recently-seen* sequences likely to reappear.
- **LZW Model:** old sequences of symbols are also useful!

The dictionary D contains a set of strings,

- Initially D is loaded with all 256 single-character strings, and the token of each single-character string simply its character code.
- All subsequent strings are given token numbers 256 or more.

LZW Compression

```
/* x is the currently matched string. Initially empty */
  c := next_char();
  while (xc ∈ D) do  /* xc == x followed by char c */
    begin
      x := xc;
      c := next_char();
    end
  output token corresponding to x;
  add xc to D, and give it the next available token.
  x := c;
```

- This is a “greedy” algorithm: it tries to match the longest prefix of the remaining input with something in the dictionary. It outputs the token for what it has matched, and adds what it fails to match to the dictionary.

LZW Example

INPUT: data␣at␣a␣date...

read d	d is in D		
read a	da is not in D	ADD da = 256	OUTPUT d
read t	at is not in D	ADD at = 257	OUTPUT a
read a	ta is not in D	ADD ta = 258	OUTPUT t
read ␣	a␣ is not in D	ADD a␣ = 259	OUTPUT a
read a	␣a is not in D	ADD ␣a = 260	OUTPUT ␣
read t	at is in D		
read ␣	at␣ is not in D	ADD at␣ = 261	OUTPUT 257
read a	␣a is in D		
read ␣	␣a␣ is not in D	ADD ␣a␣ = 262	OUTPUT 260
read d	␣d is not in D	ADD ␣d = 263	OUTPUT ␣
read a	da is in D		
read t	dat is not in D	ADD dat = 264	OUTPUT 256
read e	te is not in D	ADD te = 265	OUTPUT t

LZW Decoding

```
/* Initial code */
 t := next_token();
 output(t);
 last_str := t;

/* Subsequent code */
 t := next_token();
 curr_str := lookup(t);
 output(curr_str);
 ADD last_str followed by first character
    of curr_str into dictionary;
 last_str := curr_str;
```

- Decoder reads token from input, looks it up in the dictionary and outputs. It updates the dictionary based on what the coder did in the last step.

LZW Decoding

```
d                          OUTPUT d
a    lookup(a) = a         OUTPUT a    ADD da = 256
t    lookup(t) = t         OUTPUT t    ADD at = 257
a    lookup(a) = a         OUTPUT a    ADD ta = 258
␣    lookup(␣) = ␣         OUTPUT ␣    ADD a␣ = 259
257  lookup(257) = at      OUTPUT at   ADD ␣a = 260
260  lookup(260) = ␣a      OUTPUT ␣a   ADD at␣ = 261
␣    lookup(␣) = ␣         OUTPUT ␣    ADD ␣a␣ = 262
256  lookup(256) = da      OUTPUT da   ADD ␣d = 263
t    lookup(t) = t         OUTPUT t    ADD dat = 264
...
```

Decoder lags: it adds `␣d` after it outputs: `␣data␣at␣a␣da`

After reading `␣data␣at␣a␣da` coder has matched `da`.

Another Example

`!ow!o!o!yow!`

The output is:

```
INPUT     OUTPUT
!         none        ! is in D
o         !           ADD !o = 256
w         o           ADD ow = 257
!         w           ADD w! = 258
o         none        !o is in D
!         256         ADD !o! = 259
o         none        !o is in D
!         none        !o! is in D
y         259         ADD !o!y = 260
....
```

A Bug

INPUT	OUTPUT	ACTION
`!`	`!`	
`o`	`o`	`!o = 256`
`w`	`w`	`ow = 257`
256	`!o`	`w! = 258`
259	`???`	

INPUT	`!`	`o`	`w`	`!`	`o`	`x`	`y`	`z`
COMPRESSED	`!`	`o`	`w`	256		259		

```
259 = !ox  (3 characters)
259 = xyz
```

```
x = !
```

Bug Fix: if token not in dictionary, it must be `last_str` followed by first character of `last_str`.

LZW "Bug"

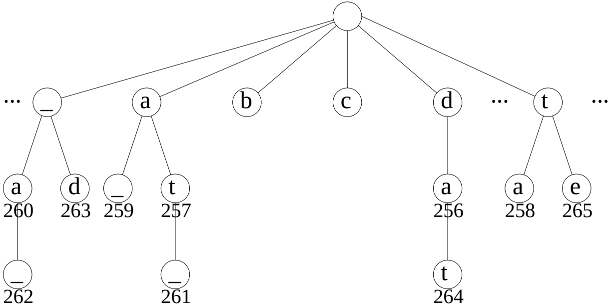
- Problem can occur if raw data contains a string of the form $xwxwx$ where x is a single character and w is a string.
  - E.g. `....thoothoot....`, where $x =$ `t` and $w =$ `hoo`.
- If xw is in the dictionary and xwx is not.
  - E.g. `thoo` is in the dictionary but `thoot` is not.
- Then when reading $xwxwx$ the coder outputs the token for xw and adds xwx to the dictionary.
  - E.g. output token for `thoo`, add `thoot` to dictionary.
- The coder then reads $xwxwx$, outputs the token for xwx but the decoder is not ready for it.
- To fix:

```
curr_str := lookup(t);
if(t not in D)
  curr_str := last_str++first char of last_str
output(curr_str);
```

LZW Speed

Compression and decompression almost equally fast.

- Compression uses *trie* data structure:



  - When compressing, LZW goes from a node in the trie to one of its children with each character read.
  - New tokens to be inserted are as children of last node visited.

- Decompression uses an array of strings.

LZW: UNIX utility `compress`

This is a widely-used LZW variant. The key features are:

- The number of bits used for representing tokens is gradually increased during encoding, so that only just enough bits are used to encode the entries in the dictionary.
  (Move from 9-bit codes to 10-bit codes when adding 512 to dictionary.)
- Limit size of dictionary & monitor performance. Rebuild from scratch if need be.
- The dictionary is represented using a *trie* data structure.