Lossless Compression

Multimedia Systems (Module 2 Lesson 3)

Summary:

- Dictionary based Compression
- Adaptive Mechanism
- Limpel Ziv Welch (LZW) mechanism

Sources:

- *The Data Compression Book*, 2nd Ed., Mark Nelson and Jean-Loup Gailly.
- LZW Compression Article from Dr. Dobbs Journal: *Implementing LZW compression using Java, by* ***Laurence Vanhelsuwé***

Dictionary-Based Compression

- The compression algorithms we studied so far use a statistical model to encode single symbols
  - Compression: Encode symbols into bit strings that use fewer bits.
- Dictionary-based algorithms do not encode single symbols as variable-length bit strings; they encode variable-length strings of symbols as single tokens
  - The tokens form an index into a phrase dictionary
  - If the tokens are smaller than the phrases they replace, compression occurs.
- Dictionary-based compression is easier to understand because it uses a strategy that programmers are familiar with-> using indexes into databases to retrieve information from large amounts of storage.
  - Telephone numbers
  - Postal codes

Dictionary-Based Compression: Example

- Consider the Random House Dictionary of the English Language, Second edition, Unabridged. Using this dictionary, the string:

  A good example of how dictionary based compression works

  can be coded as:

  1/1 822/3 674/4 1343/60 928/75 550/32 173/46 421/2
- Coding:
  - Uses the dictionary as a simple lookup table
  - Each word is coded as x/y, where, x gives the page in the dictionary and y gives the number of the word on that page.
  - The dictionary has 2,200 pages with less than 256 entries per page: Therefore x requires 12 bits and y requires 8 bits, i.e., 20 bits per word (2.5 bytes per word).
  - Using ASCII coding the above string requires 48 bytes, whereas our encoding requires only 20 (<-2.5 * 8) bytes: 50% compression.

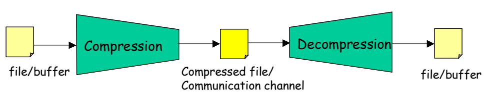
Adaptive Dictionary-based Compression

- Build the dictionary adaptively
  - Necessary when the source data is not plain text, say audio or video data.
  - Is better tailored to the specific source.
- Original methods due to Ziv and Lempel in 1977 (LZ77) and 1978 (LZ78). Terry Welch improved the scheme in 1984 (called LZW compression). It is used in, UNIX *compress*, and, GIF.
- LZ77: A sliding window technique in which the dictionary consists of a set of fixed length phrases found in a window into the previously processed text
- LZ78: Instead of using fixed-length phrases from a window into the text, it builds phrases up one symbol at a time, adding a new symbol to an existing phrase when a match occurs.

LZW Algorithm

Preliminaries:

- A dictionary that is indexed by "codes" is used.
- The dictionary is assumed to be initialized with 256 entries (indexed with ASCII codes 0 through 255) representing the ASCII table.
- The compression algorithm assumes that the output is either a file or a communication channel. The input being a file or buffer.
- Conversely, the decompression algorithm assumes that the input is a file or a communication channel and the output is a file or a buffer.



LZW Algorithm

LZW Compression:

```
set w = NIL
 loop
      read a character k
      if wk exists in the dictionary
             w = wk
      else
             output the code for w
             add wk to the dictionary
             w = k
endloop
```

The program reads one character at a time. If the code is in the dictionary, then it adds the character to the current work string, and waits for the next one. This occurs on the first character as well. If the work string is not in the dictionary, (such as when the second character comes across), it adds the work string to the dictionary and sends over the wire (or writes to a file) the code assigned to the work string without the new character. It then sets the work string to the new character.

Example of LZW: Compression

Input String: ^WED^WE^WEE^WEB^WET

w	k	Output	Index	Symbol
NIL	^			
^	W	^	256	^W
W	E	W	257	WE
E	D	E	258	ED
D	^	D	259	D^
^	W			
^W	E	256	260	^WE
E	^	E	261	E^
^	W			
^W	E			
^WE	E	260	262	^WEE
E	^			
E^	W	261	263	E^W
W	E			
WE	B	257	264	WEB
B	^	B	265	B^
^	W			
^W	E			
^WE	T	260	266	^WET
T	EOF	T		

```
set w = NIL
loop
    read a character k
    if wk exists in the dictionary
        w = wk
    else
        output the code for w
        add wk to the dictionary
        w = k
endloop
```

LZW Algorithm

LZW Decompression:

```
read fixed length token k (code or char)
output k
w = k
loop
    read a fixed length token k
    entry = dictionary entry for k
    output entry
    add w + first char of entry to
        the dictionary
    w = entry
endloop
```

The nice thing is that the decompressor builds its own dictionary on its side, that matches exactly the compressor's, so that only the codes need to be sent.

Example of LZW

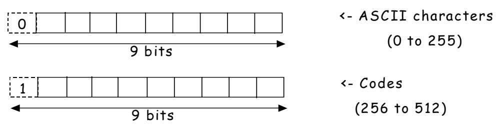
Input String (to decode): ^WED‹256›E‹260›‹261›‹257›B‹260›T

w	k	Output	Index	Symbol
	^	^		
^	W	W	256	^W
W	E	E	257	WE
E	D	D	258	ED
D	‹256›	^W	259	D^
^W	E	E	260	^WE
E	‹260›	^WE	261	E^
^WE	‹261›	E^	262	^WEE
E^	‹257›	WE	263	E^W
WE	B	B	264	WEB
B	‹260›	^WE	265	B^
^WE	T	T	266	^WET

```
read a fixed length token k
    (code or char)
output k
w = k
loop
    read a fixed length token k
        (code or char)
    entry = dictionary entry for k
    output entry
    add w + first char of entry to
        the dictionary
    w = entry
endloop
```

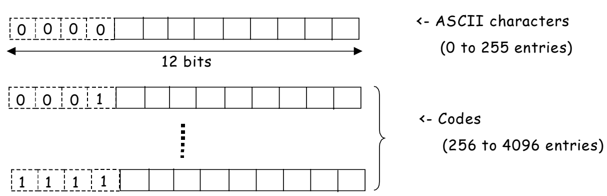
LZW Algorithm - Discussion

- Where is the compression?
  - Original String to decode : ^WED^WE^WEE^WEB^WET
  - Decoded String : ^WED<256>E<260><261><257>B<260>T
  - Plain ASCII coding of the string : 19 * 8 bits = 152 bits
  - LZW coding of the string: 12*9 bits = 108 bits (7 symbols and 5 codes, each of 9 bits)
- Why 9 bits?
  - An ASCII character has a value ranging from 0 to 255
  - All tokens have fixed length
  - There has to be a distinction in representation between an ASCII character and a Code (assigned to strings of length 2 or more)
  - Codes can only have values 256 and above



LZW Algorithm – Discussion (continued)

- With 9 bits we can only have a maximum of 256 codes for strings of length 2 or above (with the first 256 entries for ASCII characters)
- Original LZW uses dictionary with 4K entries, with the length of each symbol/code being 12 bits



- With 12 bits, we can have a maximum of 2^{12} – 256 codes.

- Practical implementations of LZW algorithm follow the two approaches:
  - Flush the dictionary periodically
    - no wasted codes
  - Grow the length of the codes as the algorithm proceeds
    - First start with a length of 9 bits for the codes.
    - Once we run out of codes, increase the length to 10 bits. When we run out of codes with 10 bits then we increase the code length to 11 bits and so on.
    - more efficient.

0	ASCII
1	Codes 256-512

Out of codes →

0	0	ASCII
0	1	Codes 256-511
1	0	Codes 512-767
1	1	Codes 768-1023

Out of codes →

0	0	0	ASCII
0	0	1	Codes 256-511
0	1	0	Codes 512-767
0	1	1	Codes 768-1023
1	0	0	Codes 1024-1279
1	0	1	Codes 1280-1535
1	1	0	Codes 1536-1791
1	1	1	Codes 1792-2047