Efficient Sequential Algorithms, Comp309

University of Liverpool

2010–2011
Module Organiser, Igor Potapov

Supplementary Material: String Algorithms.

Lempel-Ziv Welch (LZW) Compression

We will now look at the Lempel-Ziv Welch compression algorithm, which is a lossless compression algorithm that does particularly well on data with repetitions.

A useful feature of LZW compression is that the dictionary is built adaptively during encoding. The dictionary does not need to be passed with the compressed text — the decoding algorithm produces the same dictionary from the compressed text.

The original paper that describes the LZW algorithm is:

> *Terry A. Welch. A Technique for High Performance Data Compression. IEEE Computer, Vol. 17, No. 6, 1984, pp. 8-19.*

This paper describes an improvement to a compression method introduced by Ziv and Lempel in 1977 and 1978.

LZW and variants have been used in popular software such as Unix compress and GIF compression.

Sources

There is a lot of information about LZW on the web. See, for example, Wikipedia, or the nice animation at

`http://www.data-compression.com/lempelziv.shtml`

Also, see Dave Marshall's notes.

`http://www.cs.cf.ac.uk/Dave/Multimedia`

Compression

The dictionary is initialised so that there is a codeword for every extended ASCII character.

character	code word
null	0
...	...
A	65
B	66
C	67
D	68
...	...
□	255

The compression algorithm

```
Initialise dictionary
w ← NIL
while there is a character to read
        k ← next character in text
        If wk is in the dictionary
                w ← wk
        Else
                Add wk to the dictionary
                Output the code for w
                w ← k
Output the code for w
```


Example

$T = ABACABA$

w	k	rest of text	dictionary		output
			...	...	
			A	65	
			B	66	
			C	67	
			...	...	
NIL	A	BACABA			
A	B	ACABA	AB	256	65
B	A	CABA	BA	257	66
A	C	ABA	AC	258	65
C	A	BA	CA	259	67
A	B	A			
AB	A		ABA	260	256
A					65

```
Initialise dictionary
w ← NIL
while there is a char to read
    k ← next char in text
    If wk is in the dictionary
        w ← wk
    Else
        Add wk to dictionary
        Output code for w
        w ← k
Output the code for w
```

The Decompression Algorithm

The basic decompression algorithm is as follows.

Initialise dictionary
$c \leftarrow$ first codeword
output the translation of c
$w \leftarrow c$
While there is a codeword to read
    $c \leftarrow$ next codeword
    output the translation of c
    $s \leftarrow$ translation of w
    $k \leftarrow$ first character of translation of c
    Add sk to dictionary
    $w \leftarrow c$

Example

Code = 65,66,65,67,256,65

c	w	s	k	dictionary		output
				...	...	
				A	65	
				B	66	
				C	67	
				...	...	
65	65					A
66		A	B			B
	66			AB	256	
65		B	A			A
	65			BA	257	
67		A	C			C
	67			AC	258	
256		C	A			AB
	256			CA	259	
65		AB	A			A
	65			ABA	260	

Initialise dictionary
$c \leftarrow$ first codeword
output the translation of c
$w \leftarrow c$
While there is a codeword to read
    $c \leftarrow$ next codeword
    output the translation of c
    $s \leftarrow$ translation of w
    $k \leftarrow$ first character of translation of c
    Add sk to dictionary
    $w \leftarrow c$

Refinement

The decoding algorithm as stated does not always work as it fails if c is not in the dictionary.

Example

w	k	dictionary		output
		...	...	
		A	65	
		B	66	
		C	67	
		...	...	
NIL	A			
A	B	AB	256	65
B	C	BC	257	66
C	A	CA	258	67
A	B			
AB	A	ABA	259	256
A	B			
AB	A			
ABA				259

Initialise dictionary
$w \leftarrow \mathrm{NIL}$
while there is a char to read
    $k \leftarrow$ next char in text
    If wk is in the dictionary
        $w \leftarrow wk$
    Else
        Add wk to dictionary
        Output code for w
        $w \leftarrow k$
Output the code for w

Example

Code = 65,66,67,256,259

c	w	s	k	dictionary		output
				...	...	
				A	65	
				B	66	
				C	67	
				...	...	
65	65					A
66		A	B			B
	66			AB	256	
67		B	C			C
	67			BC	257	
256		C	A			AB
	256			CA	258	
259						?

```
Initialise dictionary
c ← first codeword
output the translation of c
w ← c
While there is a codeword
to read
    c ← next codeword
    output the translation of c
    s ← translation of w
    k ← first character of
        translation of c
    Add sk to dictionary
    w ← c
```


The problem arises when the dictionary has *cs* in the dictionary for a character *c* and a string *s* and then the input contains *cscsc*.

The decompression algorithm can be modified to deal with this case.


```
Initialise dictionary
c ← first codeword
output the translation of c
w ← c
While there is a codeword to read
    c ← next codeword
    s ← translation of w
    If c is in dictionary
        k ← first character of translation of c
        output the translation of c
    Else (* c is the code for what we add here*)
        k ← first character of s
        output sk
    Add sk to dictionary
    w ← c
```


Example

Code = 65,66,67,256,259

c	w	s	k	dictionary		output
				...	...	
				A	65	
				B	66	
				C	67	
				...	...	
65	65					A
66		A	B			B
	66			AB	256	
67		B	C			C
	67			BC	257	
256		C	A			AB
	256			CA	258	
259		AB	A			ABA
	259			ABA	259	

```
Initialise dictionary
c ← first codeword
output the translation of c
w ← c
While there is a codeword
    c ← next codeword
    s ← translation of w
    If c is in dictionary
        k ← 1st char of trans c
        output trans c
    Else (* c is next added*)
        k ← first character of s
        output sk
    Add sk to dictionary
    w ← c
```

There are lots of interesting implementation issues. For example, what if the dictionary runs out of space?

Also, if we start re-using dictionary space, what data structure do we use to make dictionary access efficient?

GIF compression solves the problem of dictionary overflow by having variable-length codes. We will not cover the details.