CSC 310S -- Information Theory

Lempel-Ziv-Welch Coding (based on LZ78)

Simplified Pseudocode for Encode and Decode

[Revised: March 13, 2003]

Notation:

msg	-	a string of source symbols
seq	-	a sequence of codes

Here we give a high-level (somewhat simplified) view of how the LZW encoding and decoding algorithms work.

1. Based on Dictionary Abstract Data Type

In the first version, we assume a *dictionary* that supports the following two operations:

code (phrase) - return the fixed length code for the phrase
phrase (code) - return the phrase corresponding to the code

```
Initialize:   (at both the sender and the receiver)
    Enter the D symbols of the encoding alphabet into
     the dictionary as the phrases corresponding to codes 1 to D.

Encode (msg)
    loop
        Let match be the longest initial string in the input buffer
            that appears in the dictionary, and ...
        Let break be the following character in the input buffer.
        Send the code (match).
        Enter match || break as a new entry in the dictionary.
        Shift match out of the input buffer.
    end loop

Decode (seq)
    loop
        Receive next code of the sequence.
        Set str to phrase (code) from the dictionary.
        Emit str.
        Except for the 1st iteration,
            Replace "_" in the last dictionary entry
          by the 1st sym in str.
        Add a new last dictionary containing  str || "_" .
    end loop
```

2. Based on a Linked List Implementation

In the second version, we assume that the dictionary is implemented based on a linked list. For each "code", there is a pair consisting of a "link" and a "symbol" (each fixed length). The phase corresponding to a code *ends* with "symbol", and the preceding symbols of the phrase are obtained *in reverse order* by tracing links back until a zero link is encountered (which means that the corresponding symbol is the *first* one in the phrase. Here is a recursive function that extracts the full phrase from the linked-list representation:

```
phrase(c) =  phrase(link (c)) || symbol(c)  if link (c) > 0
          =   symbol (c)                    if link (c) = 0
```

Here are the procedures for initialization, encode and decode:

```
Initialize    (at both sender and receiver)
                                  Dictionary
                                  code   link   symbol
      c := 0                        1      0     s1
      p := D + 1                    2      0     s2
                                               ...
                                    D      0     sD
Encode (msg)
    loop
        get symbol (s)
        if (c, s) is in dictionary then
             c := code (c, s)
             get symbol (s)
        else
             send (c)
             enter (c, s) as code p in the dictionary
             p := p + 1
             c := code (0, s)
        end if
        exit when eof
    end loop

Decode (seq)
    loop
        receive code (c)
        emit phrase (c)
        if p > D + 1 then
             symbol (p - 1) := 1st char of phrase (c)
        end if
        link (p) := c
        p := p + 1
    end loop
```

Next, we give an example of encoding the message:
"boohoohoochoochooboohoohoo"

Dictionary Initialization:

code	phrase
1	b
2	c
3	h
4	o

iteration	input[1 .. 6]	match	break	code	new dict. entry code	new dict. entry phrase
1	boohoo...	b	o	1	5	bo
2	oohooh...	o	o	4	6	oo
3	ohooho...	o	h	4	7	oh
4	hoohoo...	h	o	3	8	ho
5	oohooc...	oo	h	6	9	ooh
6	hoocho...	ho	o	8	10	hoo
7	ochooc...	o	c	4	11	oc
8	chooch...	c	h	2	12	ch
9	hoocho...	hoo	c	10	13	hooc
10	choobo...	ch	o	12	14	cho
11	oobooh...	oo	b	6	15	oob
12	boohoo...	bo	o	5	16	boo
13	ohooho.	oh	o	7	17	oho
14	oohoo	ooh	o	9	18	ooho
15	oo	oo	.	6	19	oo.

And here is an example in which we decode the message that was encoded above:

1 4 4 3 6 8 4 2 10 12 6 5 7 9 6

Dictionary Initialization:

code	phrase
1	b
2	c
3	h
4	o

iteration	code	emit	m	phrase(m-1)	phrase(m)
1	1	b	5	o	b_
2	4	o	6	bo	o_
3	4	o	7	oo	o_
4	3	h	8	oh	h_
5	6	oo	9	ho	oo_
6	8	ho	10	ooh	ho_
7	4	o	11	hoo	o_
8	2	c	12	oc	c_
9	10	hoo	13	ch	hoo_
10	12	ch	14	hooc	ch_
11	6	oo	15	cho	oo_
12	5	bo	16	oob	bo_
13	7	oh	17	boo	oh_
14	9	ooh	18	oho	ooh_
15	6	oo	19	ooho	oo_

The structure of the LZW algorithm leads to some significant relationships of the following quantities:

T_n	# of source symbols matched after n iterations
C_n	# of codes transmitted after n iterations
E_n	# of dictionary entries after n iterations
S_n	# of symbol occurrences in dictionary entries after n iterations
L_n	length of match in iteration n
D	# of symbols in the source alphabet
B_c	# bits to represent a dictionary code
B_s	# bits to represent a source symbol

The initial value, the increment, and the aggregate values for the first four quantities are as follows:

Initialization	Increment	Aggregate Value
$T_0 = 0$	$T_n = T_{n-1} + L_n$	$T_n = \sum_{j=1}^{n} L_j$
$C_0 = 0$	$C_n = C_{n-1} + 1$	$C_n = n$
$E_0 = D$	$E_n = E_{n-1} + 1$	$E_n = n + D$
$S_0 = D$	$S_n = S_{n-1} + L_n$	$S_n = n + D + \sum_{j=1}^{n} L_j$

Assuming that source symbols are encoded in $B_s = 8$ bits each and the dictionary codes are $B_c + 12$ bits each, then, for the same example from the previous page, we can add three columns to the table:

bits encoded	bits sent	compression ratio
$T \times B_s$	$C \times B_c$	$\frac{C \times B_c}{T \times B_s}$

Dictionary Initialization:

code	phrase
1	b
2	c
3	h
4	o

iter	input	match	break	code	new dict. entry code	new dict. entry phrase	codes sent	bits enc.	bits sent	compr. ratio
0							0	0	0	-----
1	boohoo..	b	o	1	5	bo	1	8	12	1.500
2	oohooh..	o	o	4	6	oo	2	16	24	1.500
3	ohooho..	o	h	4	7	oh	3	24	36	1.500
4	hoohoo..	h	o	3	8	ho	4	32	48	1.333
5	oohooc..	oo	h	6	9	ooh	5	48	60	1.250
6	hoocho..	ho	o	8	10	hoo	6	64	72	1.125
7	ochooc..	o	c	4	11	oc	7	72	84	1.167
8	chooch..	c	h	2	12	ch	8	80	96	1.200
9	hoocho..	hoo	c	10	13	hooc	9	104	108	1.038
10	choobo..	ch	o	12	14	cho	10	120	120	1.000
11	oobooh..	oo	b	6	15	oob	11	136	132	0.971
12	boohoo..	bo	o	5	16	boo	12	152	144	0.947
13	ohooho	oh	o	7	17	oho	13	168	156	0.929
14	oohoo	ooh	o	9	18	ooho	14	192	168	0.875
15	oo	oo	?	6	19	oo?	15	208	180	0.865