ITCT Lecture 8.2: Dictionary Codes and Lempel-Ziv Coding

Huffman codes require us to have a fairly reasonable idea of how source symbol probabilities are distributed. There are a number of applications where this is possible but there are also many applications where such a characterization is impractical. One common example can be found in compression of data files in a computer. Many different kinds of data are stored by computer systems, and even if the data are represented in ASCII format, the ASCII symbol probabilities can vary from one file to the next.

Dictionary codes are compression codes that dynamically construct their own coding and decoding tables "on the fly" by looking at the data stream itself. ➔ It is not necessary for us to know the symbol probabilities before hand.

These codes take advantage of the fact that, quite often, certain strings of symbols are "frequently repeated" and these strings can be assigned code words that represent the "entire string of symbols".

Remark:

The disadvantage of dictionary codes is that these codes are usually efficient only for long files or messages. Short files or messages can actually result in the transmission (or storage) of more bits, on the average, than were contained in the original message.

Lempel-Ziv (LZ) codes, a particular class of dictionary codes, are known to asymptotically approach the source entropy for long messages. They also have the advantage that the receiver does not require prior knowledge of the coding table constructed by the transmitter.

Sufficient information is contained in the transmitted code sequences to allow the receiver to construct its own decoding table "on the fly." Most of the information required to do this is transmitted early in the coded messages.

Remark:

These codes initially "expand" rather than "compress" the data: extra information is being supplied to the receiver so that it can construct its decoding table.

As time progresses, less and less information need be sent to aid the receiver and the code can get down to the business of compressing the transmitted data.

LZ codes also suffer “no significant decoding delay” at the receiver in the sense that, whatever code symbol is currently being received, the receiver already contains the decoding information in its local dictionary to decode what is being received as it comes in. ➔ It eliminates the need for large buffers to store the received code words until such time as the decoding dictionary is complete enough to decode them. Every received code word results in decoding right away. ➔

“Just in time” principle for date compression!

There are a number of LZ coding algorithms but they tend to be of a rather similar nature. In this talk, we will take an in-depth look at the Lempel-Ziv-Welch (LZW) algorithm.

This algorithm is better known as the "Compress" command in UNIX-based computers and is also the compress algorithm used by most PCs.

The Structure of the Dictionary

Each entry in the dictionary is given an address m.

Each entry consists of an ordered pair $<n, a_i>$, where n is a pointer to another location in the dictionary and a_i is a symbol drawn from the source alphabet A. ➔The ordered pairs in the dictionary make up a "linked list."

The pointer variables *n* also serve as the transmitted code words.

The representation of n is a "fixed-length binary word" of *b* bits such that the dictionary contains a total number of entries less than or equal to 2^b.

Since the total number of dictionary entries is going to exceed the number of symbols, M, in the source alphabet, each transmitted code word is actually going to contain more bits than it would take to represent the alphabet A. ➔ Where the compression gain comes from?

The secret of LZ coding is that most of the code words actually represent "strings of source symbols" and in a long message it is more economical to encode these strings than it is to encode the individual symbols.
➔ "Fixed codeword length but variable string (symbol) length!"

The algorithm is initialized by constructing the first M+1 entries in the dictionary as follows.

address	dictionary	entry
0	0,	null
1	0,	a_0
2	0,	a_1
⋮	⋮	⋮
m	0,	a_{m-1}
⋮	⋮	⋮
M	0,	a_{M-1}

The o-address entry in the dictionary is a null symbol. It is used to let the decoder know where the end of the string is, i.e., it is a kind of "Punctuation mark."

The pointers n in these first M+1 entries are zero. They "point" to the null entry at address 0. The initialization also initializes pointer variable n=0 and address pointer m=M+1. The address pointer m points to the next "blank" location in the dictionary.

Encoding Procedures:

```
1.  Fetch next source symbol a;
2.  If the ordered paired <n,a> is already in the
    dictionary then
            n=dictionary address of entry <n,a> ;
    else
            transmit n;
            create new dictionary entry <n,a> at
            dictionary m;
            m=m+1;
            n=dictionary address of entry <0,a> ;
3.  Return to step 1.
```

If <n,a> is already in the dictionary in step 2, the encoder is processing a string of symbols that has occurred at least once previously. Setting the next value of n to this address constructs a linked list that allows the string of symbols to be traced.

If <n,a> is not already in the dictionary in step 2, the encoder is encountering a new string that has not been previously processed. It transmits code symbol n, which lets the receiver know the dictionary address of the last source symbol in the previous string.

Whenever the encoder transmits a code symbol, it also creates a new dictionary entry <n,a>. In this entry, n is a pointer to the last source symbol in the previous string of source symbols and a is the “root symbol” which begins a new strings. Code word n is then reset to the address of <0,a>. The “0” pointer in this entry points to the null character which indicates the beginning of the new string.

Example 1: (HW)

A binary information source emits the sequence of symbols 110 001 011 001 011 100 011 11 etc. Construct the encoding dictionary and determine the sequence of transmitted code symbols.

Solution: Initialize the dictionary as shown above, Since the source is binary, the initial dictionary will contain only the null entry and the entries 〈0,0〉 and 〈0,1〉 at dictionary addresses 0, 1, and 2, respectively. The initial values for n and m are n=0 and m=3. The encoder’s operation is then described in the following table.

source symbol	present *n*	present *m*	transmit	next *n*	dictionary entry
1	0	3		2	
1	2	3	2	2	2, 1
0	2	4	2	1	2, 0
0	1	5	1	1	1, 0
0	1	6		5	
1	5	6	5	2	5, 1
0	2	7		4	
1	4	7	4	2	4, 1
1	2	8		3	
0	3	8	3	1	3, 0
0	1	9		5	
1	5	9		6	
0	6	9	6	1	6, 0
1	1	10	1	2	1, 1
1	2	11		3	
1	3	11	3	2	3, 1

source symbol	present *n*	present *m*	transmit	next *n*	dictionary entry
0	2	12		4	
0	4	12	4	1	4, 0
0	1	13		5	
1	5	13		6	
1	6	13	6	2	6, 1
1	2	14		3	
1	3	14		11	

The encoder's dictionary to this point is as follows：

dictionary address	dictionary entry
0	0, *null*
1	0, 0
2	0, 1
3	2, 1
4	2, 0
5	1, 0
6	5, 1
7	4, 1
8	3, 0
9	6, 0
10	1, 1
11	3, 1
12	4, 0
13	6, 1
14	no entry yet

The Decoding Process

The decoder at the receiver must also construct a dictionary identical to the one at the transmitter and it must decode the received code symbols. Notice from the previous example how the encoder constructs code symbols for strings and that the encoder does not transmit as many code words as it has source symbols. Operation of the decoder is governed by the following observations：

1. Reception of any code word means that a new dictionary entry must be constructed;
2. Pointer *n* for this new dictionary entry is the same as the received code word *n*;
3. Source symbol *a* for this entry is not yet known, since it is the *root* symbol of the next string (which has not yet been transmitted b the encoder).

If the address of this next dictionary entry is *m*, we see that decoder can only construct a partial entry $\langle n, ? \rangle$ since it must await the next received code word to find the root symbol *a* for the entry. It can, however, fill in the missing symbol *a* in its *previous* dictionary entry at address $m-1$. It can also decode the source symbol string associated with received code word *n*. To see how this is done, let us *decode* the transmissions sent by the encoder of Example 1.

Example 2
Decode the received code words transmitted in Example 1.
(*Note*：As you follow along with this example, you may find it useful to write down the dictionary construction as it occurs; that will help you better see what’s going on.)

Solution： The decoder begins by constructing the same first three entries as the encoder. It can do this because the source alphabet is known *a priori* by the decoder. The decoder's initialized value for the next dictionary entry is $m = 3$.

The first received code word is $n = 2$. The decoder creates a dictionary entry of $\langle 2, ? \rangle$ at address $m = 3$ and then increments *m*. Since this is the first received code word, there is no previous partial dictionary entry to "fill in," so the decoder decodes the received symbol. Code word $n = 2$ points to address 2 of the table, which contains the entry $\langle 0, 1 \rangle$.The last symbol of coded string is therefore "1" and the pointer "0" indicates that this is also the first symbol of the string. Therefore, the entire "string" consists only of the symbol "1

The next received code word is $n = 2$. The decoder creates a partial dictionary entry at location $m = 4$ of $\langle 2, ? \rangle$. It must now complete the previous entry at location $m = 3$. It does this by looking at the dictionary entry at address $n = 2$. This entry is $\langle 0, 1 \rangle$ and the “0” pointer tells it that the symbol “1” is the root symbol of the string. Therefore, the entry at $m = 3$ is $\langle 2, 1 \rangle$. The decoder then increments *m* to $m = 5$.

The next code word is $n = 1$ and the decoder constructs a new partial dictionary entry at $m = 5$ of $\langle 1, ? \rangle$. Address $n = 1$ of the dictionary contains $\langle 0, 0 \rangle$. This is a root symbol so the dictionary entry at address 4 is updated to $\langle 2, 0 \rangle$. The string designated by the code word is decoded as “0,” since the source symbol at address $n = 1$ is a root symbol. Pointer *m* increments to 6.

The next code word, $n=5$, is used to create the partial dictionary entry 6: $\langle 5, ?\rangle$. The decoder completes the entry at dictionary address $m-1=5$ by accessing the dictionary entry at address $n=5$. This entry is 5: $\langle 1, ?\rangle$. Since this entry has a non-zero pointer, the symbol (which does not happen to be known yet !) is *not* the root symbol of the string. The decoder therefore links to the address specified by the pointer. This is address 1 which contains the entry $\langle 0, 0\rangle$. The zero pointer for this entry indicates that symbol "0" is the root of the string so the entry at dictionary location 5 is updated to 5: $\langle 1, 0\rangle$. The decoder then decodes the received string. It begins by looking at the entry in location $n=5$. This entry indicates that the last character in the string is a "0" and that there is a previous character at dictionary address 1. his address contains a "0" symbol with a pointer of zero. Therefore, the decoded string is "00".

The next received code word is $n = 4$ which produces partial dictionary entry 7: $\langle 4, ? \rangle$. The entry at address 6 is completed by looking up the entry 4: $\langle 2, 0 \rangle$. This is *not* a root symbol so the decoder links to dictionary entry 2: $\langle 0, 1 \rangle$ This *is* a root symbol, so address 6 gets 6: $\langle 5, 1 \rangle$. The decoded symbol string is specified by 4: $\langle 2, 0 \rangle$ and 2: $\langle 0, 1 \rangle$ as “10”. (Note that the root symbol comes first in the string!).

This process is repeated with $n = 3$, producing 8: $\langle 3, ? \rangle$, 7: $\langle 4, 1 \rangle$, and decoded output string “11”. Code word $n = 6$ produces 9: $\langle 6, ? \rangle$ and 8: $\langle 3, 0 \rangle$. The decoded string has three links this time, 6: $\langle 5, 1 \rangle$, 5: $\langle 1, 0 \rangle$, and 1: $\langle 0, 0 \rangle$, which define the string “001”. You are invited to continue this decoding process to complete the dictionary and decoded outputs for the remaining received code words. You will find that the encoder’s dictionary is reproduced by this process and that the decoded output symbols are the same as those originally emitted by the information source.

The LZ algorithm works by parsing the source's *t*-symbol sequence $s_0\ s_1 \dots s_{t-1}$ into smaller strings of symbols. These strings are commonly called *phrases*. Furthermore, every such phrase is *unique*. No two phrases in the dictionary are alike. The total number of these phrases is equal to the number of dictionary entries. Let *c(t)* be the total number of phrases resulting from a *t*-symbol source sequence. The number of bits required to represent the pointer/code word *n* is therefore $\log_2(c(t))$ and the total number of bits stored in the dictionary is therefore

$$b = c(t)\ [\log_2 (c(t)) + 1],$$

where the "+1" in this expression accounts for the fact that each dictionary entry must include a one-bit source symbol.

The number of phrases depends on the particular source sequence. Lempel and Ziv have proven that *c(t)* is upper bound by

$$c(t) \le \frac{t}{(1-\varepsilon_t)\log_2(t)}$$

where

$$\varepsilon_t = \min[\frac{\log_2[\log_2(t)]+4}{\log_2(t)}]$$

for *t≧4.* Note that

$$\lim_{t\to\infty} \varepsilon_t \to 0.$$

Therefore, the asymptotic upper bound on *c*(*t*) for large *t* gives us

$$b = c(t)[\log_2(c(t)) + 1] \le \frac{t}{\log_2(t)}[\log_2(t) - \log_2[\log_2(t)]] < t.$$

This tell us that compression will eventually occur if *t* is large enough.

How about the efficiency of the compression? To answer this, recall that the entropy of the sequence s_0 s_1 … s_{t-1} is upper-bounded by

$$H(s_0 s_1 \cdots s_{t-1}) \le tH(A),$$

where A is the alphabet of out information source.

The quantity

$$\lim_{t \to \infty} \frac{H(s_0\, s_1 \cdots s_{t\text{-}1})}{t}$$

is called the entropy rate of the source and is a measure of the average information per transmitted symbol in the sequence s_0 s_1 … s_{t-1} (This assumes the information source probabilities are not function of time.)

It has been shown, i.e., a theorem has been proved, that LZ codes have the property that if the information source is a stationary ergodic process, then

$$\lim_{t\to\infty}\sup\frac{1}{t}c(t)\log_2[c(t)] \le \lim_{t\to\infty}\frac{1}{t}\mathrm{H}(\mathrm{s}_0\ \mathrm{s}_1 \cdots \mathrm{s}_{\mathrm{t\text{-}1}})$$

with probability 1. This limit tells us that, asymptotically, the average code-word length per source symbol of the LZ code is no greater than the entropy rate of the information source.

This is another way of saying that the LZ code is asymptotically as efficient as any algorithm can get. The “investment” it makes in expanding the number of bits sent at the beginning of the message sequence is paid back later if only the message sequence is long enough. Compression studies carried out by Welch on typical computer files have demonstrated that ASCII text files can typically be compressed by a factor of 2 to 1 for files containing 10,000 or more characters using the LZW algorithm.