CHAPTER 6: CODE FOR DATA COMPRESSION

Introduction to Data Compression

We use binary strings to represent the characters in a symbol set. We will call the symbol set an alphabet. For example the ASCII code represents each character of the alphabet as a binary string of 8 bits. This is an example of a fixed-length code. We would like to consider code of variable length so that the average code length is as small as possible. Then the total number of bits required to represent a piece of information (a message, a document or a file) is minimal. One way of achieving this is to assign shorter codewords to represent symbols that occur more frequently. Assume that the relative frequency or probability of each symbol is known.

The objectives of data compression include:

(i) to save storage space,

(ii) to reduce transmission time when sending data.

Fixed-length code versus variable length code

In a fixed-length code all the codewords have the same length. In a variable-length code codewords may have different lengths.

Example 1:

Suppose that we have a 100,000 character data file that we wish to store. The file contains only 6 characters, appearing with the following frequencies:

	a	b	c	d	e	f
Freq in '000s	45	13	12	16	9	5
A fixed-length code	000	001	010	011	100	101
A variable length code	1	010	011	000	0010	0011

The fixed length-code requires

The variable length-code uses only

From the example above, we see that variable-length codes provide opportunity for data compression.

Unique decoding and Prefix Code

When using code of variable length, one problem is that decoding may be ambiguous. The same code string may be decoded in different ways if more than one source strings have the same code string.

For example, if A is represented by 0, B is represented by 01, C is represented by 10, then the code string 010 may be interpreted as AC or BA depending on whether it is read as 0-10 or 01-0. Such a code string is not uniquely decodable.

One way to ensure unique decoding is to require that the code is prefix free: no codeword is a prefix of another codeword. Prefix here means a left substring. In the example just now, the codeword of A is 0, which is a prefix of 01, the codeword of B.

A code is called a prefix code (or prefix-free code) if no codeword is a prefix of another codeword.

Example 2:

{a = 0, b = 110, c = 10, d = 111} is

{a = 0, b = 01, c = 10, d = 100} is

Huffman Code For Data Compression

Huffman designed the following algorithm to generate prefix-free binary code to minimise the average code length by assigning shorter codewords to represent more frequent symbols, assuming that the probability distribution of symbols in the alphabet is known.

Arrange the probabilities in descending order. Merge the two lowest values (i.e. represent the two lowest values by their sum). Continuing merging two lowest values until they are all merged into 1.

Construct a binary tree to reverse the merging process, splitting each merged group into two branches. Label the leaves of the tree with the symbols corresponding to the probabilities values.

The codeword for a symbol is obtained by starting at the root and moving down until we reach the leaf that holds the symbol. Each time we move down a left branch we add a 0 to the code, and each time we move down a right branch we add a 1.

The method is best illustrated by an example.

Example 3:

Given a set of symbols and their probability distribution:

Symbol (x_i)	A	B	C	D	E
Probability, $P(x_i)$	0.10	0.15	0.30	0.16	0.29

Find a prefix-free binary code (a set of codewords) with minimum expected codeword length (equivalently, a tree with minimum weighted path length from the root).

Solution:

Arrange the probabilities in descending order and merge the 2 lowest values:

Reverse the merging process by splitting the merged values, producing this binary tree:

From the tree, we read the Huffman code as follows:

Symbol (x_i)	A	B	C	D	E
Probability $P(x_i)$	0.10	0.15	0.30	0.16	0.29
Codeword (C_i)					
Codeword length (in bits) (L_i)					

Expected length of code $L(C) = \sum L_i P(x_i)$

Lower bound of expected code length and Entropy

Is it possible to design binary code to make the expected code length as small as we like? According to Shannon theorem, there is a lower limit to it:
The expected code length cannot be less than $-\sum P(x_i)\log_2 P(x_i)$

where $P(x_i)$ is the probability of symbol x_i, and $\sum$ means summation over all the symbols in the alphabet.

The quantity $H(X) = -\sum P(x_i)\log_2 P(x_i)$ is called the entropy of the symbol set with that probability distribution. It is a measure of the smallest expected codeword length that is theoretically possible for the given alphabet with the associated probability distribution.

As $\log_2$ is usually not included in most calculators, we may calculate it by a change of base: $\log_2 P = \log_{10} P / \log_{10} 2$ or $\ln(P) / \ln(2)$.

Then $H(X) = -\frac{\sum P(x_i)\log P(x_i)}{\log 2}$ using log of any base.

Example 4:

Referring to the same example above, find the entropy of the probability distributions on *x*.

$$H(X) = -\frac{\sum P(x_i)\log P(x_i)}{\log 2}$$

The Huffman code in example 3 has expected codeword length of 2.25 bits per symbol, only slightly larger than the calculated entropy of 2.2047 bits per symbol. Not only is this code optimal in the sense that no other feasible code performs better, but it is very close to the theoretical limit established by Shannon.

Efficiency

Every variable length code has a certain measure of how well it encodes a language. One such measure is the efficiency, which is defined as the ratio of the entropy to the expected value of the length of the code.

$$\text{Efficiency} = \frac{\text{entropy}}{\text{expected codeword length}} = \frac{H(X)}{L(C)}$$

In examples 3 and 4, the efficiency of the Huffman code is

Example 5:

Using Huffman tree, construct a Huffman code for the five letters with the given probabilities as below.

Symbol (x_i)	E	A	M	N	T
Probability, $P(x_i)$	0.40	0.30	0.20	0.05	0.05

Calculate the entropy of the probability distribution and the efficiency of the code.

Solution:

Symbol (x_i)	E	A	M	N	T
Probability, $P(x_i)$	0.40	0.30	0.20	0.05	0.05

Symbol (x_i)	E	A	M	N	T
Probability, $P(x_i)$	0.40	0.30	0.20	0.05	0.05
Codewords (C_i)					
Codeword length (in bits) (L_i)					
Expected codeword length ($L_i\ P(x_i)$)					
Entropy ($-P(x_i) \log_2 P(x_i)$)					

Entropy $H(X)$

Expected codeword length,

Efficiency

Alternative Huffman codes

A Huffman code need not be unique, but it is always one of the codes minimizing $L(C)$. Alternative codes exist when there are equal probability values at any stage of the construction process.

Example 6:

Symbol (x_i)	A	B	C	D	E
Probability, $P(x_i)$	0.2	0.4	0.1	0.1	0.2

Obtain 2 different Huffman codes for this symbol set.

Solution:

Merging process:

Now there are many ways to split those merged values.
Two possible codes are given below:

The codes are tabulated below:

Symbol	Probabilty	Huffman code 1	Huffman code 2
A	0.2		
B	0.4		
C	0.1		
D	0.1		
E	0.2		

Further considerations

The entropy is determined with the assumption that successive symbols in the source string are independent. That means the occurrence of earlier symbols does not affect the probability of the next symbol. In practice, successive symbols are quite frequently associated. For example, in English, patterns like “ing”, “ed”, “tion” and “ise” are quite common. When we see “tio”, we would guess the next symbol to be “n” which is highly likely to be correct. If we consider such blocks of symbol with high frequency, it might be possible to construct shorter codes to “beat the best”, and achieve stronger compression. Students who are interested to pursue the topic further may read books on “Data Compression” and “Information Theory”.

Extra Example:

Letter, (x_i)	a	e	s	o	g	u	m
Probability, $P(x_i)$	0.27	0.23	0.02	0.18	0.08	0.17	0.05

a) Construct a Huffman tree and hence find the Huffman code for the seven letters with the given probabilities as above.

b) Calculate the entropy of the probability distribution and the efficiency of the Huffman code obtained in part a).