Subtleties of Source Coding*

Don Johnson

This work is produced by OpenStax-CNX and licensed under the
Creative Commons Attribution License 1.0†

Abstract

Some subtleties of coding, including self synchronization and a comparison of the Huffman and Morse codes.

In the Huffman code, the bit sequences that represent individual symbols can have differing lengths so the bitstream index m does not increase in lock step with the symbol-valued signal's index n. To capture how often bits must be transmitted to keep up with the source's production of symbols, we can only compute averages. If our source code averages $B\left(\overline{A}\right)$ bits/symbol and symbols are produced at a rate R, the average bit rate equals $B\left(\overline{A}\right) R$, and this quantity determines the bit interval duration T.

Exercise 1 ***(Solution on p. 4.)***

Calculate what the relation between T and the average bit rate $B\left(\overline{A}\right) R$ is.

A subtlety of source coding is whether we need "commas" in the bitstream. When we use an unequal number of bits to represent symbols, how does the receiver determine when symbols begin and end? If you created a source code that **required** a separation marker in the bitstream between symbols, it would be very inefficient since you are essentially requiring an extra symbol in the transmission stream.

NOTE: A good example of this need is the Morse Code: Between each letter, the telegrapher needs to insert a pause to inform the receiver when letter boundaries occur.

As shown in this example[1], no commas are placed in the bitstream, but you can unambiguously decode the sequence of symbols from the bitstream. Huffman showed that his (maximally efficient) code had the **prefix** property: No code for a symbol began another symbol's code. Once you have the prefix property, the bitstream is **partially** self-synchronizing: Once the receiver knows where the bitstream starts, we can assign a unique and correct symbol sequence to the bitstream.

Exercise 2 ***(Solution on p. 4.)***

Sketch an argument that prefix coding, whether derived from a Huffman code or not, will provide unique decoding when an unequal number of bits/symbol are used in the code.

However, having a prefix code does not guarantee total synchronization: After hopping into the middle of a bitstream, can we always find the correct symbol boundaries? The self-synchronization issue does mitigate the use of efficient source coding algorithms.

*Version 2.16: Dec 5, 2007 11:02 pm -0600

†http://creativecommons.org/licenses/by/1.0

[1] "Compression and the Huffman Code", Example 1 <http://cnx.org/content/m0092/latest/#ex3>

Exercise 3 ***(Solution on p. 4.)***

Show by example that a bitstream produced by a Huffman code is not necessarily self-synchronizing. Are fixed-length codes self synchronizing?

Another issue is bit errors induced by the digital channel; if they occur (and they will), synchronization can easily be lost even if the receiver started "in synch" with the source. Despite the small probabilities of error offered by good signal set design and the matched filter, an infrequent error can devastate the ability to translate a bitstream into a symbolic signal. We need ways of reducing reception errors **without** demanding that p_e be smaller.

Example 1

The first electrical communications system—the telegraph—was digital. When first deployed in 1844, it communicated text over wireline connections using a binary code—the Morse code—to represent individual letters. To send a message from one place to another, telegraph operators would tap the message using a telegraph key to another operator, who would relay the message on to the next operator, presumably getting the message closer to its destination. In short, the telegraph relied on a **network** not unlike the basics of modern computer networks. To say it presaged modern communications would be an understatement. It was also far ahead of some needed technologies, namely the Source Coding Theorem. The Morse code, shown in Figure 1, was not a prefix code. To separate codes for each letter, Morse code required that a space—a pause—be inserted between each letter. In information theory, that space counts as another code letter, which means that the Morse code encoded text with a three-letter source code: dots, dashes and space. The resulting source code is not within a bit of entropy, and is grossly inefficient (about 25%). Figure 1 shows a Huffman code for English text, which as we know **is** efficient.

Morse and Huffman Code Table

	%	Morse Code	Huffman Code
A	6.22	.-	1011
B	1.32	-...	010100
C	3.11	-.-.	10101
D	2.97	-..	01011
E	10.53	.	001
F	1.68	..-.	110001
G	1.65	–.	110000
H	3.63	....	11001
I	6.14	..	1001
J	0.06	.—	01010111011
K	0.31	-.-	01010110
L	3.07	.-..	10100
M	2.48	–	00011
N	5.73	-.	0100
O	6.06	—	1000
P	1.87	.–.	00000
Q	0.10	–.-	0101011100
R	5.87	.-.	0111
S	5.81	...	0110
T	7.68	-	1101
U	2.27	..-	00010
V	0.70	...-	0101010
W	1.13	.–	000011
X	0.25	-..-	010101111
Y	1.07	-.–	000010
Z	0.06	–..	0101011101011

Figure 1: Morse and Huffman Codes for American-Roman Alphabet. The % column indicates the average probability (expressed in percent) of the letter occurring in English. The entropy $H(A)$ of the this source is 4.14 bits. The average Morse codeword length is 2.5 symbols. Adding one more symbol for the letter separator and converting to bits yields an average codeword length of 5.56 bits. The average Huffman codeword length is 4.35 bits.

Solutions to Exercises in this Module

Solution to Exercise (p. 1)

$T = \frac{1}{\bar{B}(A)R}$.

Solution to Exercise (p. 1)

Because no codeword begins with another's codeword, the first codeword encountered in a bit stream must be the right one. Note that we must start at the beginning of the bit stream; jumping into the middle does not guarantee perfect decoding. The end of one codeword and the beginning of another could be a codeword, and we would get lost.

Solution to Exercise (p. 1)

Consider the bitstream ...0110111... taken from the bitstream 0|10|110|110|111|.... We would decode the initial part incorrectly, then would synchronize. If we had a fixed-length code (say 00,01,10,11), the situation is **much** worse. Jumping into the middle leads to no synchronization at all!