Encoding a prefix code

Adam Gashlin

March 29, 2015

1 Introduction

In order for a receiver to interpret a prefix code, he must understand how the strings of bits in the coded message map to the source alphabet symbols. The code may be static, i.e. prearranged; dynamic, where the code is computed online as the message is received; and semi-static, where a fixed code is transmitted as a prefix to the message [13]. In the semi-static case we then have the additional problem of how to encode the code, and in particular how to do this efficiently. The efficiency of semi-static coding is the topic of this report.

Fundamentally the code consists of two parts: What are the variable-length bit strings which make up the code, and What are the source alphabet symbols that correspond to those strings. The former is highly structured: no bit string is a prefix for another. This property allows us to represent the bit strings as a binary tree, where the unique path to each leaf spells out a bit string. Moreover this binary tree should be complete, if the prefix code is optimal. We will consider this, the structure of the code, in section 2. The latter (correspondence of source alphabet symbols) is generally less structured and depends highly on the nature of the source alphabet. We can consider this a completely separate problem, though there are some gains to be had by considering it, this is addressed in section 3.

2 Code structure

2.1 Coding complete trees

A well-known approach [3, 6] to encoding a prefix code uses one bit for each of the $2n-1$ nodes, where n is the number of bit strings (and thus source symbols). This is to be expected: there are C_n (the n^{th} Catalan number) possible complete binary trees with n leaves [9]; C_n grows as 4^n, so we should expect the number of bits needed to specify a code to grow as $\log_2 4^n = 2n$. Generally if we know how many codes $f(n)$ there are for n symbols, we can take $\lceil \log_2 f(n) \rceil$ to give a lower limit on the number of bits that will be required to encode it, as any fewer bits would be unable to provide a unique encoding for each code.

2.2 Canonical codes

There are only two possible (optimal) sets of three bit strings to use for a three symbol prefix code: {0,10,11} and {1,00,01}. We can consider these as equivalent; whichever we use, we will end up with messages of the same length. The important aspect is the level sequence, i.e. how many strings there are of each length; the only possible level sequence for $n = 3$ is 1,2. Consider the possibilities for $n = 4$: {00,01,10,11}, {0,10,110,111}, {0,11,100,101}, {1,00,010,011}, {1,01,000,001}. Most of these have the level sequence 1,1,2, the exception is the first with 0,4,0.

We consider classes of codes with the same level sequence equivalent, and we choose only one of them as the "canonical" code [10, 2]. The canonical code is the one in which the bit strings in lexicographical order are also in order of increasing length. Here we see that only {00,01,10,11} and {0,10,110,111} are canonical. Similarly {0,10,11} is canonical for $n = 3$. It is also easy to generate the canonical bit strings from the level sequence. There are other aspects of the canonical code which we will see in section 3.

2.3 Coding canonical codes

Narimani and Khosravifard [8] give a coding for a level sequence that uses only $n-3$ bits, they express this as a sequence of functions applied to an initial level sequence.

An equivalent code is an unary coding of the internal node level sequence, i.e. the number of internal nodes at each level in the tree (including the root), from which the level sequence can be derived. The $n-1$ internal nodes require $n-1$ bits for the unary coding of the sequence, with 1 coded as `1`, 2 as `01`, 3 as `001`, etc. As the first bit of the code is always `1` (indicating the root, which always has exactly one internal node), and the last bit is always `1` (indicating the end of the unary string for the last level), these two bits can be elided, yielding a code that uses $n-3$ bits.

As an example, a level sequence of 0,4 corresponds to an internal node level sequence of 1,2, which is encoded as `1,01` (comma for illustration only). After removing the leading and trailing bits this is just `0`, which is sufficient to distinguish from the only other $n = 4$ level sequence 1,1,2 (internal node sequence 1,1,1, encoded as `1,1,1` or just `1`).

2.4 Theoretical limit

The number of canonical prefix codes has been studied many times [5, 1, 4, 11]. The number of codes for n symbols grows as 1.755^n, thus requiring approximately $0.8115n$ bits in the long run. Therefore there exists a coding with cn bits with $c < 1$, however there may not be an effective coding with that property.

3 Symbol assignments

3.1 Bit length per symbol

The standard approach takes advantage of the fact that we don't care (when creating the code) if we have B=10,C=11 or C=10,B=11; bit strings of the same length are equivalent. In a true canonical code, the bit strings are always assigned to the source symbols in alphabetical order (so we'd always have B=10,C=11). Therefore we can list, for each symbol of the source alphabet, how long the corresponding bit string is (e.g. 1,2,2 for A=0,B=10,C=11; 2,1,2 for A=10,B=0,C=11). We can derive the level sequence by counting the number of symbols at each length, and we can assign bit strings to symbols in lexicographical order. As bit strings can be up to $n-1$ long, we need $\lceil \log_2 n \rceil$ bits per symbol, so we end up with $n \lceil \log_2 n \rceil$ bits total.

3.2 Alphabet permutations

For another approach, consider that we have three symbols in the source alphabet and we want to assign them to the canonical bit sequences {0,10,11}. There are six possible assignments; clearly these are just the permutations of three objects. If we have prearranged an alphabet of symbols to draw from, we can encode permutations of those n symbols with $n \lceil \log_2 n \rceil - n$ bits. Recall from section 2 that we can specify a canonical set of bit sequences in around n bits. Therefore, if we want to give the complete code, bit sequences and symbol assignments, we can send the bit sequence specification followed by the alphabet permutation in $n \lceil \log_2 n \rceil$ bits total.

It is interesting that this approach achieves the same cost as that mentioned in subsection 3.1. Notably this does not take full advantage of the canonical code; by specifying the permutation of the whole alphabet we allow both B=10,C=11 and C=10,B=11. It should be possible to achieve some improvement if we take this into account. However note that in the case of a "Fibonacci" tree, with a level sequence like 1,1,1,...,2, there is no escaping the need to specify the full permutation of $n-2$ symbols. Therefore we could only gain an advantage in conjunction with a tree encoding that uses fewer bits to describe such a tree. There is only one such tree, so perhaps this is possible.

3.3 Theoretical limit

For each canonical set of bit sequences considered in 2.4, we need to decide how to partition the symbols among the levels/bit lengths. For $n = 4$, {00,01,10,11} only has one partition, with all 4 symbols at level 2; however {0,10,110,111} has 12 possibilities (any of 4 for level 1, any of remaining 3 for level 2, level 3 is then fixed); a total of 13.

Molteni [7] discusses limits and asymptotic behavior of this sequence [12], the result in bits is approximately $n \log_2 n - 1.885n$. Therefore there should

exist a coding with $n \log_2 n - n$ bits, but whether there is an effective coding is unknown.

References

[1] Clowes, J., Mitrani, I., and Wilson, L. Level Number Sequences for Binary Trees. Tech. Rep. 112, Computing Laboratory, University of Newcastle upon Tyne, 1977.

[2] Connell, J. A Huffman-Shannon-Fano code. *Proceedings of the IEEE 61*, 7 (July 1973), 1046–1047.

[3] Dreizen, H. Technical Correspondence. *Commun. ACM 29*, 2 (Feb. 1986), 149–150.

[4] Elsholtz, C., Heuberger, C., and Prodinger, H. The Number of Huffman Codes, Compact Trees, and Sums of Unit Fractions. *IEEE Transactions on Information Theory 59*, 2 (Feb. 2013), 1065–1075.

[5] Gilbert, E. Codes based on inaccurate source probabilities. *Information Theory, IEEE Transactions on 17*, 3 (1971), 304–314.

[6] Horspool, R. N., and Cormack, G. V. Technical Correspondence. *Commun. ACM 29*, 2 (Feb. 1986), 150–152.

[7] Molteni, G. Representation of a 2-Power as Sum of k 2-Powers: The Asymptotic Behavior. *International Journal of Number Theory 8*, 8 (Dec. 2012), 1923–1963.

[8] Narimani, H., and Khosravifard, M. The supertree of the compact codes. In *International Symposium on Telecommunications, 2008. IST 2008* (Aug. 2008), pp. 649–655.

[9] Otter, R. The Number of Trees. *Annals of Mathematics 49*, 3 (July 1948), 583–599.

[10] Schwartz, E. S., and Kallick, B. Generating a Canonical Prefix Encoding. *Commun. ACM 7*, 3 (Mar. 1964), 166–169.

[11] Sloane, N. J. A. The On-Line Encyclopedia of Integer Sequences. http://oeis.org/A002572. Number of partitions of 1 into n powers of 1/2.

[12] Sloane, N. J. A., Plouffe, S., Knuth, D., and van der Sanden, H. The On-Line Encyclopedia of Integer Sequences. http://oeis.org/A007178. Number of ways to write 1 as ordered sum of n powers of 1/2, allowing repeats.

[13] Turpin, A., and Moffat, A. Housekeeping for prefix coding. *IEEE Transactions on Communications 48*, 4 (Apr. 2000), 622–628.