Computer Science Department of The University of Auckland
CITR at Tamaki Campus (http://www.tcs.auckland.ac.nz**)**

CS-TR-159 / CITR-TR-16 September 1997

A Low-Cost Decoder for Arbitrary Binary Variable-Length Codes

Ulrich Günther and Radu Nicolescu*

Abstract

Encoders and decoders for variable-length codes such as Huffman Codes can be costly to implement. This paper describes low-cost encoder and decoder for binary variable-length codes that is simple to implement when decoding speed is not an issue.

* The University of Auckland, Tamaki Campus, Computing and Information Technology Research, Computer Vision Unit, Auckland, New Zealand

A Low-Cost Decoder for Arbitrary Binary Variable-Length Codes

Ulrich Günther and Radu Nicolescu

Computing and Information Technology, Department of Computer Science,
Tamaki Campus, University of Auckland, Auckland
New Zealand
E-mail: {u.guenther, r.nicolescu} @auckland.ac.nz

Abstract

Encoders and decoders for variable-length codes such as Huffman Codes can be costly to implement. This paper describes low-cost encoder and decoder for binary variable-length codes that is simple to implement when decoding speed is not an issue.

Keywords*: encoder, decoder, variable-length code, Huffman code, tree*

1. Introduction

Variable-length Huffman codes [1] are widely used in data compression, e.g., in the Unix **`pack`** command or as a low-level part of the popular JPEG image data compression scheme. Encoding and decoding variable-length codes presents an important problem in an environment dominated by the fixed word length data representation in modern computers.

For a given variable-length code, an encoder must thus convert some fixed-length input into its variable-length representation, and vice-versa, a decoder converts variable-length input into its corresponding fixed-length output.

Little interest, if any, seems to be paid to variable-length encoders in literature, perhaps because they are generally regarded as the ``easier half''. This is different for decoders. Tanaka [2] presents a decoder based on an automaton. The decoding tree in Tanaka's decoder is stored in a transition matrix, whose row indices correspond to the branch or leaf nodes in the decoding tree. The column indices correspond to the next bit received by the decoder, and the matrix entries indicate the node number to which this bit takes the decoder.

Hirschberg and Lelewer [3] present a somewhat smaller decoder for canonical Huffman codes, which put a constraint on the shape of the decoding tree. While canonical Huffman codes have the same coding efficiency as general Huffman codes, other aspects such as spectral characteristics or synchronisation are not addressed by them.

Higgie [4] introduced a decoder based on a pointer look-up table where subtrees of the decoding tree are ``skipped'' in the decoding process by way of a ``next-1-pointer''. Recently, Chung [5] presented a decoder based on a recursive description of the encoding tree. Chung's decoder also promotes the idea of skipping subtrees.

Higgie's and Chung's idea of skipping subtrees does also find application in the decoder proposed in our paper. Our decoder, however, uses a very cost-efficient encoding of the decoding tree and derives the skipping

information from this encoding rather than from dedicated side information.

This representation will be introduced in Section 2, followed by our proposal for an encoder and a decoder based on this model in Sections 3 and 4. Section 5 discusses cost aspects, and Section 6 explains how we may obtain a tree prescription from the statistics of an information source.

2. Representing the decoding tree

The enumeration of binary trees by a stack is a well known representation. For a complete binary tree, we may choose the following representation using a binary string:

1. draw the tree such that all branches corresponding to a 0 point to the left, and all those corresponding to a 1 point to the right.
2. number all leaf nodes on the tree in lexicographical order from ``left to right'', starting with the leaf node furthest to the left, i.e., the all-zero codeword.
3. begin a prescription string, which is initially empty (λ).
4. traverse all leaf nodes in lexicographical order, starting from the root node as follows:
5. for all 0-branches encountered, add a 0 to the string.
6. upon reaching a leaf node other than the last one to the right, go back up the tree to the nearest incomplete branch node, add a 1-branch to it and append a 1 to the string. Continue the traversal from that 1-branch.

Example 1: consider the tree in Figure 1. It may be encoded as follows:

- begin a new string `s:=`λ.
- Number the codewords as follows, in lexicographical order (preorder): **codeword (number)**: 0 (0), 100 (1), 1010 (2), 10110 (3), 10111 (4), 110 (5), 111 (6).
- now traverse the tree's leaf nodes. We start from the root node, adding a 0 to the string as we move left to leaf node number 0: `s:=0.`
- next, move back up the tree to the next incomplete branch node, in this case the root. Take the 1-branch to the right, and add a 1 to the string: `s:=01.`
- now we have reached a new branch node, so take the 0-branch from there and add a 0 to the string: `s:=010.`
- add another 0 to the string to reach codeword 100 (leaf node no. 1): `s:=0100.`
- move back up the tree to the next incomplete branch node (the node corresponding to 10). Take the 1-branch to the right, and add a 1 to the string: `s:=01001.`
- take the 0-branch to the left to reach 1010 (leaf node number 2). Now `s:=010010.`
- complete the traversal in the same fashion. Eventually, we obtain

`s=010010101101.`

Note that each occurrence of 1 in `s` indicates the return from a leaf node. The first 1 indicates that we have passed through leaf node 0, the second 1 indicates that we have passed through leaf node 1 etc.

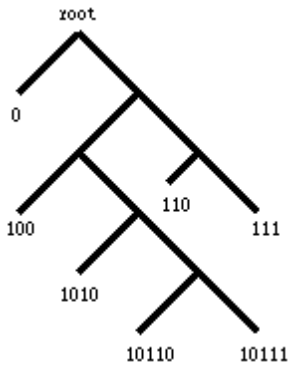

Figure 1: the code tree used in our examples

We shall briefly show that this prescription is unique for and applicable to all finite, complete, and prefix-free binary decoding trees. For this, we note that any such tree may be constructed from

the tree corresponding to the code set {0,1}, by **leaf node expansion**, i.e., by replacing an existing leaf node with a two-branch subtree. Thus, any finite, complete, and prefix-free binary decoding tree may be derived from the tree for {0,1} by a series of leaf node expansions. For example, the tree in Figure 1 may be constructed as follows:

Example 2: start with the elementary tree {0,1}. Expand leaf nodes in the order 1, 11, 10, 101, 1011 to obtain final code set {0,100,1010,10110,10111,110, 111}.

The prescription for the initial tree {0,1} is always 01. Subtree prescriptions are simply inserted after the bit corresponding to the leaf node the subtree's root is appended to. Again, illustrated with the same steps as above:

Example 3: start with the elementary tree {0,1}, represented by 01. Continue as follows (the dots enclose the newly added subtree):

- 01.01. represents {0,10,11}.
- 0101.01. represents {0,10,110,111}.
- 010.01.101 represents {0,100,101,110,111}.
- 01001.01.101 represents {0,100,1010,1011,110,111}.
- 0100101.01.101 represents {0,100,1010,10110,10111,110,111}.

As the reader may wish to verify, the order in which the subtrees are added does not affect the final prescription.

In the opposite direction, we may interpret the prescription as a computer program with the two commands:

- 0: draw a 0-branch, starting from the current node, and make the 0-node at the end of the branch the current node.
- 1: starting from the current node, go back up the tree to the nearest incomplete branch node. Draw a 1-branch, completing that branch node, and make the 1-node at the end of the branch the current node.

Hence, our prescription is unique. However, not all finite binary strings are prescriptions of complete, prefix-free trees. In particular, we require that prescriptions start with a 0 and end with a 1. As all branch nodes must be complete, the number of 0-branches equals the number of 1-branches, and thus a prescription must contain the same number of zeros and ones. Also, since the 0-branches of the elementary 01-subtrees are traversed first, the number of 0-bits in any proper left prefix of the prescription exceeds the number of 1-bits in that prefix.

Furthermore, we note that every complete subtree of a complete tree must in itself comply with these rules, and (unless it includes the rightmost leaf node in the tree) the substring corresponding to the subtree must be followed by a 1.

We shall now show how prescriptions may be utilised in a low-cost encoder and decoder.

3. The encoder

In both the encoder and decoder applications, we implicitly assume that the codewords that are to be the encoder input or decoder output are in lexicographical order with respect to their corresponding leaf nodes of the tree, i.e., that they "equal" the leaf node numbers.

If this direct correspondence does not exist, it may be achieved by a bijective mapping of the N fixed-length codewords onto the the set $\{0,1,\ldots,N-1\}$. In practice, this bijective mapping may be implemented as a look-up table/array. Its cost needs to be taken into account when comparingencoders/decoders.

Suppose that we wish to encode codeword number n. We did note above that, in our tree description, each 1 indicates the return from a leaf node. This feature may be utilised to construct an encoder, since we can find the return from the n'th leaf node by counting the 1's and stopping at the $(n+1)$'th, or the end of the tree description string, whichever occurs first. The actual algorithm works as follows:

Presume that the tree for which we wish to implement the encoder has been encoded into a prescription string s=s[1]s[2]…s[i]… The following algorithm may be used to emit the n'th codeword, where $n \in \{0,1,\ldots,N-1\}$:

1. initialise a binary stack, a counter m for the 1's in the tree string, and a tree string position pointer i. Set m=0 and i=1. For the purpose of terminating the algorithm, we also append a 1 to the end of the tree prescription string s.

2. while i ≤ |s| and m ≤ n do

   - if s[i] =1, increment m. If now m ≤ n, pop elements off the stack until the element popped is a 0.
   - push s[i] onto the stack.
   - increment i.

3. pop one element off the stack (this is necessary to get rid of the terminating 1)

4. the stack now contains the codeword. Output the stack contents in FIFO order.

Example 4: Consider the tree from Figure 1, with prescription s=0100101011101. Presume that we wish to encode the codeword with number n=5, whose variable-length form is 110. The encoding process will work as follows:

1. create an empty binary stack σ=[ ] , and set i=1 and m=0.
2. s[1] =0, hence σ=[0] and i=2.
3. s[2] =1, hence m=1, σ=[1] , and i=3.
4. s[3] =0, hence σ=[1,0] and i=4.
5. s[4] =0, hence σ=[1,0,0] and i=5.
6. s[5] =1, hence m=2, σ=[1,0,1] , and i=6.
7. s[6] =0, hence σ=[1,0,1,0] and i=7.
8. s[7] =1, hence m=3, σ=[1,0,1,1] , and i=8.
9. s[8] =0, hence σ=[1,0,1,1,0] and i=9.
10. s[9] =1, hence m=4, σ=[1,0,1,1,1] , and i=10.
11. s[10] =1, hence m=5, σ=[1,1] , and i=11.
12. s[11] =0, hence σ=[1,1,0] and i=12.
13. s[12] =1, hence m=6, σ=[1,1,0,1] and i=13. Exit loop.
14. get rid of terminating 1 that has landed on the stack: now σ=[1,1,0] .
15. output σ in FIFO order, i.e., starting with the leftmost 1.

---

4. A low-cost decoder

We presume that the tree for which we wish to implement the decoder has again been encoded into a prescription s=s[1] s[2] …s[i] …. Again, we append an extra 1 to the end of s.

Presume further that the decoder receives a bitstream x=x[1] x[2] … . We wish to decode the first codeword in x by finding its leaf node number n:

1. set i=1 and j=1, and n=0.
2. if s[i] =1, increment n.
3. compare s[i] and x[j] :

   - if s[i] <>x[j] , skip the subtree starting at s[i] . This involves incrementing i and possibly reading in further s[i] until the complete subtree at this branch has been skipped. As mentioned above, the substring corresponding to the subtree is characterised by an equal number of 0's and 1's, followed by a 1. Each time a 1 is read from s in this process, increment n. Set i to index the first bit following the subtree.
   - if s[I] =x[j] , increment both i and j. If now s[i] =1, output n and exit.

4. continue at step 2.

Example 5: Consider the decoding tree from Figure 1, whose prescription (with a 1 appended) is s=0100101011011. Presume that we wish to decode a bitstream starting with the codeword 1010, which carries leaf node number 2.

According to the instructions above, we proceed as follows:

1. set i=1 and j=1, and n=0.
2. s[1] =0, i.e., n does not change.
3. s[1] =0 and x[1] =1. Skip the subtree: this subtree ends with s[2] , hence we get i=2.
4. s[2] =1, so n=1.
5. s[2] =1 and x[1] =1. Thus set i=3 and j=2. Now s[3] <>1 , so continue.
6. s[3] =0, n does not change.
7. s[3] =0 and x[2] =0. Set i=4 and j=3. The exit conditions are not met, so continue.
8. s[4] =0, n does not change.
9. s[4] =0 and x[3] =1. Skip the subtree: this subtree is also elementary and ends with s[5] , hence we get i=5.
10. s[5] =1, so n=2.
11. s[5] =1 and x[3] =1. Set i=6 and j=4. The exit conditions are not met, so continue.
12. s[6] =0, n does not change.
13. s[6] =0 and x[4] =0. Set i=7 and j=5. However, now s[7] =1, hence exit and output n=2.

5. Encoder and decoder cost

When implementing an encoder or decoder in hardware or software, there is a cost in terms of storage space.

The encoder presented above generally requires the following storage:

- a look-up table. For a code with N codewords, this table requires $N\lceil \log_2 N \rceil$ bits of storage.
- s storage for the tree prescription. This string contains one bit for every branch in the tree, i.e., two for every (complete) branch node. We also require a bit for the termination. In a binary tree, there are N-1 branch nodes. Hence, the string requires 2N-1 bits of storage.
- i: a register capable of counting up to the string's length plus one, which thus requires $\lceil \log_2 (2(N-1)+1) \rceil$ bits.
- m: requires $\lceil \log_2 (N+1) \rceil$ bits.
- the binary stack. Maximum stack depth here is the same as the maximum codeword length. This is constrained by N-1. Hence, the stack needs $\lceil \log_2 (N-1) \rceil$ bits storage plus $\lceil \log_2 \lceil \log_2 (N-1)+1 \rceil \rceil$ bits for a stack pointer.

On the decoder side, the following storage is required:

- look-up table, tree prescription string, and index i as above.
- j depends on the codeword length and hence requires $\lceil \log_2 (N-1) \rceil$ bits.
- n requires $\lceil \log_2 N \rceil$ bits.
- a counter for the ones/zeros in the subtrees: $\lceil \log_2 N \rceil$ bits.

For large N, the look-up table is the most dominant item in both the encoder and decoder. In this respect, our proposals compare favourably to Tanaka's decoder, where a transition matrix roughly combines the functions of both the decoder tree representation and the look-up table.

This transition matrix requires $\lceil \log_2 N \rceil+1$ bits for every entry in the state transition matrix. This is about one bit more per entry than the look-up table entries in our case because Tanaka includes not only the leaf nodes, but also all branch nodes in his representation. With hence 4N-2 entries in the matrix, Tanaka also requires almost four times as many entries in the transition matrix as we require for the look-up table.

Higgie's and Chung's decoders require a similar amount of storage as Tanaka's automaton.

Another cost is the encoding and decoding speed. In this respect, the other decoders

mentioned are superior to our model, which requires a traversal of the whole decoding tree. However, not all applications require speedy decoding. This is the case, for example, in communication applications where the data processing bandwidth at the transmitter and receiver is much larger than the bandwidth of the communication channel.

6. Obtaining a Huffman tree prescription from an information source

Typically, Huffman code decoding trees are obtained from the statistics of source symbols that we wish to encode. For this purpose, Huffman's algorithm recursively combines the two smallest probabilities to form a new (branch node) symbol until only the root node is left. It then recursively builds the tree by expanding the branch nodes [1],[6].

The same algorithm, with a minor modification, may be used to produce the tree prescription used in our encoder and decoder. When the tree is built from the root node, we start with an empty string and insert a new subtree string 01 at the proper position with every branch node expansion that we perform.

7. Conclusion

The encoder and decoder models proposed in this paper extend the range of available encoders and decoders for variable-length codes such as Huffman codes. In particular, they may find application in circumstances where the decoding/encoding speed is uncritical while the storage cost is important.

Decoder, encoder, and an algorithm for deriving the tree prescripton from source probabilities have been successfully implemented and tested in MATLAB.

8. Acknowledgements

The authors would like to thank their colleagues at the Computing and Information Technology Research group at Tamaki Campus for their helpful comments.

6. References

[1] D. Huffman, “A Method for the Construction of Minimum Redundany Codes”, Proc. Inst. Radio Eng., vol. 40, pp.1098-1101, 1952.

[2] H. Tanaka, “Data Structure of Huffman Codes and its Application to Efficient Encoding and Decoding”, IEEE Trans. Inform. Theory, vol. 33, no. 1, pp.154-156, 1987.

[3] D.S. Hirschberg and D.A. Lelewer, “Efficient Decoding of Prefix Codes”, Communicatuions of the ACM, vol. 33, no. 4, pp.449-459, 1990.

[4] G.R. Higgie, “Analysis of the Families of Variable-Length Self-Synchronizing Codes called T-Codes”, PhD Thesis, The University of Auckland, 1991.

[5] K.-L. Chung, “Efficient Huffman Decoding”, Information Processing Letters, vol. 61, pp.97-99, 1997.

[6] R.W. Hamming, “Coding and Information Theory”, 2nd ed., Prentice-Hall, 1986.