IPN Progress Report 42-146 August 15, 2001

A New Entropy Coding Technique for Data Compression

A. B. Kiely[1] and M. Klimesh[1]

We present a novel entropy coding technique that is based on recursive interleaving of variable-to-variable-length binary source codes. An entropy coder using this technique is a general purpose module that can be used in a wide variety of data compression algorithms. The encoding process is adaptable in that each bit to be encoded has an associated probability-of-zero estimate that may depend on previously encoded bits. This adaptability allows more efficient compression, and the technique has speed advantages over arithmetic coding, the state-of-the-art adaptable entropy coding method. The technique can achieve arbitrarily small redundancy. Much variation is possible in the choice of component codes and in the interleaving structure, yielding coder designs of varying complexity and compression efficiency. We discuss coder design and performance estimation methods. We present practical encoding and decoding algorithms, as well as measured performance results.

I. Introduction

In data compression algorithms, the need frequently arises to compress a binary sequence in which each bit has some estimated distribution, i.e., probability of being equal to zero. If long runs of bits have nearly identical distributions, then simple source codes, most notably Golomb's runlength codes [8,9], are quite efficient. However, in many practical situations, not only does the distribution vary from bit to bit, but it is desirable to have the estimated distribution for a bit depend on the values of earlier bits.

Allowing the estimated distribution to change with each new bit can result in more efficient compression because a source model can make better use of the immediate context in which a bit appears and can quickly adapt to changing statistics. For example, when compressing a bit plane of a wavelet-transformed image, one would want to use an entropy coder that can efficiently encode a bit sequence with a probability estimate that varies from bit to bit.

Accommodating a dynamically changing probability estimate is tricky because the decompressor will need to make the same estimates as the compressor. In general, before the ith bit can be decoded, the values of the first $i-1$ bits must be determined. This requirement makes it difficult to efficiently use simple source codes such as runlength codes. To our knowledge, currently the only efficient coding

---

[1] Communications Systems and Research Section.

The research described in this publication was carried out by the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

methods that accommodate a bit-wise adaptive probability estimate are arithmetic coding[2] [12,14,19] and a relatively obscure technique called interleaved entropy coding [2,5–7].

In this article, we introduce a new technique called *recursive interleaved entropy coding*, which is a generalization of interleaved entropy coding. A recursive interleaved entropy coder compresses a binary source with a bit-wise adaptive probability estimate by recursively encoding groups of bits with similar distributions, ordering the output in a way that is suited to the decoder. Much variation is possible in the choice of component codes and in the interleaving structure, yielding coder designs of varying complexity and compression efficiency.

As an indication of the interest in low-complexity encoding and decoding of sequences with adaptive probability estimates, we note that much effort has been put into reducing the complexity of arithmetic coding; see [3,4,15–23]. These complexity reductions generally involve the use of judicious approximations and typically result in a slight decrease in compression efficiency.

The functionality of our coding technique is essentially the same as that of binary arithmetic coding [10–14]; however, our coder is not an arithmetic coder, and there are many practical differences. Arithmetic encoding of one bit requires a few arithmetic operations and usually at least one multiplication. Our encoder requires no arithmetic operations except those that might be needed to choose a code index based on the estimated bit distribution; however, it requires some bookkeeping and bit manipulation operations. Our encoder requires more memory than arithmetic coding. Arithmetic decoders are generally of similar complexity to the encoders, but our decoder is much simpler than our encoder: it needs fewer operations than the encoder and requires only a small amount of memory. In a related article [1], we describe modified encoding and decoding techniques with lower encoder memory requirements.

In the remainder of this section, we give a brief overview of the entropy coding technique. Section II describes the details of encoder and decoder operation and presents practical encoding and decoding algorithms. In Section III we examine a class of binary trees that appears to be well suited for use in coder designs. Section IV gives methods for estimating the performance of a given coder design. In Section V we describe a technique for designing an encoder that meets a given redundancy constraint. Section VI provides performance results. Finally, Section VII provides a conclusion and identifies some open problems.

A. The Source Coding Problem

We examine the problem of compressing a sequence of bits $b_1, b_2, \cdots$ from a random source. For each source bit b_i we have a probability estimate $p_i = \mathrm{Prob}[b_i = 0]$ that may depend on the values of the source sequence prior to index i and on any other information available to both the compressor and decompressor. This dependence encompasses both adaptive probability estimation as well as correlations or memory in the source. Because of this dependence, efficient coding requires a bit-wise adaptable coder. We are not concerned here with methods of modeling the source, and so we make no distinction between the actual and estimated probabilities.

Without loss of generality, we assume that $p_i \geq 1/2$ for each index i. If this were not the case for some p_i, we could simply invert bit b_i before encoding to make it so; this inversion can clearly be duplicated in the decoder.

We also assume that the decompressor can determine when decoding is complete. In practice, this often occurs automatically; otherwise, a straightforward method such as transmitting the sequence length prior to the compressed sequence can be used.

---

[2] We include in the family of arithmetic coders assorted approximate arithmetic coders such as the quasi-arithmetic coder of [4] and the Z-coder [3].

Although we only discuss the compression of binary sequences, given any nonbinary source, we can assign prefix-free binary codewords to source symbols to produce a binary stream. Thus, a bit-wise adaptable coder such as the one we describe here can be applied to nonbinary sources as well.

B. The Recursive Interleaved Entropy Coder Concept

We now give an overview of how our entropy coding technique works. To simplify the explanation, we defer some of the details until Section II.

Since, by assumption, each bit has probability-of-zero at least 1/2, we are concerned with the probability region $[1/2, 1]$. We partition this region into several narrow intervals, and with each interval we associate a bin that will be used to store a list of bits. When bit b_i arrives, we place it into the bin corresponding to the interval containing p_i. Because each interval spans a small probability range, all of the bits in a given bin have nearly the same probability-of-zero, and we can think of each bin as corresponding to some nominal probability value.

Bits in the leftmost bin, whose interval contains probability 1/2, do not require further processing. For every other bin, we specify an exhaustive prefix-free set of binary codewords. When the bits collected in a bin form one of these codewords, we delete these bits from the bin and encode the value of the codeword by placing one or more new bits in other bins.[3] In effect, bits in a bin are encoded using a prefix-free variable-to-variable-length code, with the added twist that the output bits are assigned to other bins where they may be further encoded.

The mapping from codewords to encoded bits is conveniently described using a binary tree. Each codeword is assigned to a terminal node in the tree; non-terminal nodes are labeled with a probability value that determines a destination bin; and the branch labels (each a zero or one) indicate the output bits that are placed in the destination bins. For example, Fig. 1 shows a tree that might be used for a bin with nominal probability 0.9. The prefix-free codeword set for this bin is $\{00, 01, 1\}$, shown as labels of the terminal nodes in the tree. If the codeword to be processed in the bin is 00, which occurs with probability approximately 0.81, we place a zero in the bin that contains probability 0.81. If the codeword is 1, first we place a one in the bin containing probability 0.81, which indicates that the codeword is something other than 00; then we place a zero in the bin containing probability 0.53 because, given that the codeword is not 00, the conditional probability that the codeword is 1 is approximately 0.53. We can see that this process contributes to data compression because the most likely codeword is 00, which is represented using a single bit.

Bits that reach the leftmost bin form the encoder's output; we refer to this bin as the "uncoded" bin since these bits do not undergo further coding. These bits are zero with probability very close to 1/2 and are thus nearly incompressible, so leaving these bits uncoded does not add much redundancy.

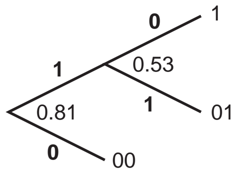

Fig. 1. Example of a tree for a bin with representative probability 0.9.

[3] The rules for collecting bits to form a codeword are not straightforward, and we save the details for Section II.B.

During the encoding process, bits arrive in various bins either directly from the source or as a result of processing[4] codewords in other bins. Our goal is to have bits flow to the leftmost bin. To accomplish this, we impose the constraint on the non-terminal node labels that all new bits resulting from the processing of a codeword must be placed in bins strictly to the left of the bin in which the codeword was formed. Apart from our desire to move bits to the left, this constraint also prevents encoded information from traveling in loops, which would make coding difficult or impossible.

As illustrated in the example above, a natural method of mapping output bits to bins is to assign each output bit of a codeword to the bin indicated by the output bit's probability-of-zero, as computed from the nominal probability of the bin in which the codeword is formed. If we use this method, then a bin with nominal probability p must necessarily use a tree that is "useful" at p according to the following definition.

Definition.

(1) *We say that a tree is* useful at probability p *if it has the property that, when all input bits have probability-of-zero equal to* p, *all output bits have probability-of-zero in the range* $[1/2, p)$.

(2) *A tree is* useful *if there exists some* p *for which the tree is useful at* p.

(3) *If the branches of a tree lack output bit labels, then we say that the tree is useful (respectively* useful at probability p) *if some assignment of output bit labels makes the tree useful (respectively useful at probability* p*).*

Perhaps surprisingly, requiring output bits to be mapped strictly to the left turns out to be a reasonable constraint—we'll see in Section III.A that for any $p \in (1/2, 1)$ there exists a tree that is useful at p.

In practice, bins are identified by indices rather than nominal probability values, starting with index 1 for the leftmost bin. At each non-terminal node in a tree, we identify the index, rather than the nominal probability value, of the bin to which the associated output bit is mapped. The constraint on encoder design is now that each output bit from the tree for a bin must be mapped to a bin with strictly lower index. No computations involving probability values are needed for encoding apart from those that may be required to map input bits to the appropriate bins.

The mapping of output bits to bins can be designed without regard for nominal probability values, so it is possible to design working coders that include trees that are not useful. In fact, good coders can be designed that do not contain useful trees. However, one expects better compression if output bit probabilities are in close agreement with their destination bins' nominal probabilities; this appears to be most easily achieved by exploiting useful trees.

Near the end of encoding of a sequence of bits, there may be bins that contain partial codewords that must be "flushed" from the encoder. When this occurs, we append one or more extra bits to the partial codeword to form a complete codeword that is then processed in the normal manner. Section II.C gives more details about flushing the encoder.

Clearly the encoder's output contains some redundancy because source bits with slightly different probabilities-of-zero are treated the same; that is, the bins' intervals have positive widths. As one might expect, by increasing the number of bins in the coder design, we can decrease the redundancy to arbitrarily small values. This result is formalized in Section III.B.

[4] We refer to the encoding of a codeword in a bin as "processing" the codeword, rather than "encoding," to avoid confusion with the overall encoding procedure.

C. Relation to Interleaved Entropy Codes

An important special case of our entropy coder arises when all output bits generated from each tree are mapped to the uncoded bin. In this case, the coder essentially interleaves several separate variable-to-variable-length binary codes. This technique was first suggested in [6], where Golomb codes are interleaved, and has also appeared in [7]. Howard [5] gives a more thorough analysis of this technique, which we refer to as non-recursive interleaved entropy coding. A non-recursive coder design allows reduced-complexity encoding; in Section VI, we show that in fact non-recursive coders can achieve very high encoding speeds.

By using an increasing number of increasingly complex variable-to-variable-length codes, it's clear that we can make asymptotic redundancy arbitrarily small with a non-recursive coder (provided that the estimates of the source distribution can be made arbitrarily accurate). With the additional flexibility of the recursive technique presented here, a given redundancy target tends to be achievable with fewer and/or simpler codes.

II. Encoding and Decoding

Section I gave an overview of how recursive interleaved entropy coding works. In this section, we describe the encoding and decoding procedures in more detail and give practical algorithms for encoding and decoding.

It should be noted that the encoding algorithm presented here requires memory resources that are proportional to the length of the source bit sequence. Alternative encoding algorithms (with corresponding decoding algorithms) that have much more modest memory requirements are described in [1].

We first state more precisely how a coder is specified. A recursive interleaved entropy coder specification consists of

(1) An integer B indicating the number of bins. The bins are indexed from 1 to B.

(2) For each bin with index greater than 1, an exhaustive prefix-free set of binary codewords and a binary tree that describes rules for processing each codeword by placing one or more bits in lower-indexed bins.

(3) A rule for mapping source bits to bins.

While we usually think of each bin in the encoder as corresponding to some probability interval, such a relationship is not required, and there need not be any implicit probability estimate used to map bits to bins.

Since we are not concerned with source modeling, we will frequently present coder designs without specifying a rule for mapping source bits to bins. In this case, we assume that each source bit is mapped to the bin that minimizes redundancy given the source bit probability-of-zero p_i.

As a running example to illustrate the encoding and decoding procedures, we use a 5-bin coder design that we refer to as C5. This coder design is illustrated in Fig. 2, where the trees shown identify the relationship between codewords and output bits. For example, Fig. 2(c) indicates that if codeword 01 is formed in bin 4, then we place bits 1,0,1 in bins 3,2,1, respectively.

To illustrate encoding and decoding using coder design C5, we also need a rule to assign a bin index to each source bit. For the examples in this section, we'll let this bin index equal one more than the number of zeros in the four most recent source bits (or in all of the preceding source bits when there are fewer than four of them). This is not a rule one would likely use in practice, but it is convenient for illustrative purposes.

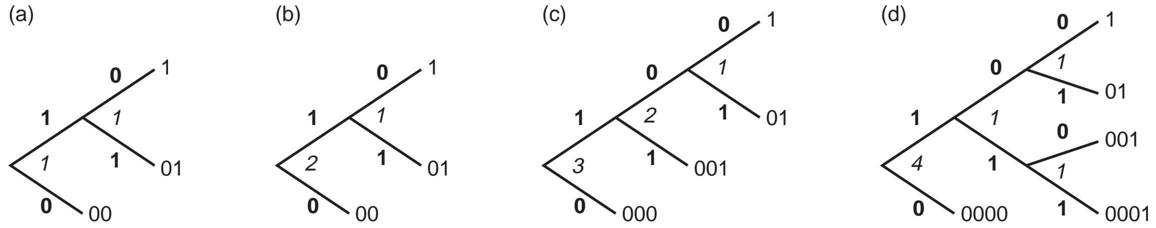

Fig. 2. The 5-bin coder design C5: (a) bin 2, (b) bin 3, (c) bin 4, and (d) bin 5. Output bits are shown in boldface; the corresponding bin indices are in italics. The input codewords are shown at terminal nodes of the trees. The first bin of a coder design does not have an associated tree.

A. Decoder Operation

We first describe the decoding procedure since it determines how encoding must be performed. We regard each bin in the decoder as containing a list of bits. To begin, all of the encoded bits are placed in the first (uncoded) bin, and all other bins are empty. At any time, each nonempty bin (with the exception of the uncoded bin) will contain a single codeword or a suffix of a codeword. To decode a source bit, we take the next bit from the bin to which the source bit was assigned. If this bin is empty, we first reconstruct the codeword in that bin by taking bits from other bins as needed. This in turn may require reconstructing codewords in those bins, and so on.

Example 1. Suppose a 4-bit source sequence is encoded using coder design C5 and the encoder output is $0, 1, 1, 1, 0, 0$. To decode, first we place the encoded bits in the first bin. From our bin assignment rule, the first source bit came from bin 1, so our first output bit equals the first encoded bit, which is a zero. Our decoder state now looks like this:

1	2	3	4	5
~~0~~				
1				
1				
1				
0				
0				

decoded sequence = 0

Since the decoded bit sequence consists of a single zero, our bin assignment rule tells us that the next source bit comes from bin 2. This bin is empty, so we must reconstruct a codeword in the bin. Examining the tree for bin 2, we see that the possible codewords are 00, 01, and 1, which produce 0, 11, and 10, respectively, in bin 1. Since the next bits in bin 1 are 11, we delete these bits and place codeword 01 in bin 2. The first bit of this codeword is the next source bit, so we output a zero and delete this zero from bin 2:

1	2	3	4	5
~~0~~	~~0~~			
~~1~~	1			
~~1~~				
1				
0				
0				

decoded sequence = 0,0

According to our bin assignment rule, the next source bit was placed in bin 3. Examining the tree for bin 3, we see that the 1 in the second bin indicates that the codeword was either 01 or 1, and the 1 in

the first bin signals that the codeword was 01, so we place the codeword 01 in the third bin, removing the bits from bins 1 and 2 as they are used. The first bit from this codeword is our next source bit, so we delete it:

1	2	3	4	5
~~0~~	~~0~~	~~0~~		
~~1~~	~~1~~	1		
~~1~~				
~~1~~				
0				
0				

decoded sequence = 0,0,0

Since the first three bits are $0, 0, 0$, the fourth and final bit comes from the fourth bin. Following the tree for bin 4, we see that the 1 in bin 3 indicates that the codeword in bin 4 is 001, 01, or 1. To determine which it is, we must take a bit from bin 2, which is empty. So we must first reconstruct a codeword in bin 2. Following the tree for bin 2, the 0 in bin 1 tells us to place the codeword 00 in bin 2. Now we can continue reconstructing the codeword in bin 4. The 0 in bin 2 indicates that the codeword in bin 4 is either 01 or 1, and the 0 in bin 1 tells us that the codeword in bin 4 is 1. (As usual, bits are deleted as they are used.) The 1 reconstructed in bin 4 is the fourth decoded bit:

1	2	3	4	5
~~0~~	~~0~~	~~0~~	~~1~~	
~~1~~	~~1~~	~~1~~		
~~1~~	~~0~~			
~~1~~	0			
~~0~~				
~~0~~				

decoded sequence = 0,0,0,1

The final decoded sequence is $0, 0, 0, 1$. We observe that an unused 0 bit remains in bin 2. This bit was added during encoding to flush a partial codeword from bin 2; it is ignored in decoding. △

Software decoding can be accomplished using two recursive procedures, `GetBit` and `GetCodeword`, shown in Fig. 3. `GetBit` simply takes the next available bit from the indicated bin. If this bin is empty, then it first calls `GetCodeword`. Given an empty bin, `GetCodeword` determines which codeword must have occupied the bin by taking bits from other bins (via `GetBit`), and then places that codeword in the bin. The `GetCodeword` procedure is similar to Huffman decoding, except that at each step we take the next bit from the appropriate bin, not (necessarily) from the encoded bit stream.

To decode the ith source bit, let `binindex` equal the index of the bin to which this source bit is assigned according to the bin assignment rule for the coder. Then the ith decoded bit is equal to `GetBit(binindex)`.

B. Encoder Operation

The encoding procedure outlined in Section I.B illustrated the codeword processing operations involved in encoding, but did not discuss the order in which bits are collected to form codewords. This order is important; we must encode in a way that allows the decoder to determine the value of source bit b_{i-1} before attempting to decode source bit b_i, since the bin to which a source bit is mapped may depend on the values of previous source bits.

`GetBit(binindex)`:

(1) If the bin with index `binindex` is empty, then call `GetCodeword(binindex)`.

(2) Remove the first bit in the bin with index `binindex` and return the value of this bit.

`GetCodeword(binindex)`:

(1) Initialize `nodepointer` to point to the root of the tree for the bin with index `binindex`.

(2) While `nodepointer` is not pointing to a terminal node, do the following:

   (a) Let `thisbinindex` equal the bin index indicated by the node at `nodepointer`.

   (b) Assign `bitvalue` = `GetBit`(`thisbinindex`).

   (c) Let `nodepointer` point to the node indicated by the branch with label equal to `bitvalue`.

(3) At this point `nodepointer` points to a terminal node, which means that we have reached a codeword. Place this codeword in bin `binindex`.

Fig. 3. The `GetBit` and `GetCodeword` procedures used for recursive decoding.

In practice, the order in which bits are collected to form codewords is specified somewhat indirectly. During encoding, each bin is viewed as containing a list of bits, but when a bit arrives in a bin as an output bit from another bin, the bit might be inserted into the list at a position other than the end of the list. In this section, we give encoding rules that specify when we can process a codeword formed by the bits at the beginning of a bin's list and at what positions in the destination bins' lists to insert the bits produced by processing the codeword. We also give practical encoding algorithms that conform to these rules.

The rules are described using an ordering of all bits in the encoder. We refer to this ordering as the "priority" order of the bits; for source bits, it corresponds to the order in which bits arrive (earlier bits have higher priority), and when encoding is complete the encoder output consists of the bits in bin 1 in priority order.

The following list summarizes the priority rules:

(1) Source bit b_{i-1} has higher priority than source bit b_i.

(2) Bits taken from a bin to form a codeword must be taken in priority order. The encoder output (i.e., the bits that fall in the uncoded bin) must be read in priority order.

(3) When a codeword is formed, the resulting output bits are assigned priority essentially equal to the highest priority of the bits forming the codeword. Among these output bits, those generated closer to the root of the tree are assigned higher priority.

(4) We cannot process a codeword if a bit with higher priority than the lowest priority bit in the codeword might enter the codeword's bin.

(5) Flush bits can be added to complete a partial codeword in a bin only if no other bits can enter the bin.

Part of the idea behind Rule (3) is that, when the decoder reconstructs codewords, it needs the values of bits generated closest to the root of the tree first. For example, suppose we formed the codeword 001

in bin 4 of an encoder using coder design C5. In this case, the output bits produced are 1,1 in bins 3,2. The bit placed in bin 3 must have higher priority, because the decoder will not know whether a bit was placed in bin 2 until it reads the value of the bit in bin 3.

Rule (4) implies that even when bits in a bin (taken in priority order) form a codeword, we may not be able to process the codeword. For example, suppose that the first three input bits to an encoder using coder design C5 are $0, 0$, and 1 and are placed in bins 3, 4, and 3, respectively,[5] so that our encoder state looks like this:

1	2	3	4	5
		0 (1)	0 (2)	
		1 (3)		

Here the bit index is written in parentheses after the bit value. Although the bits in bin 3 form a codeword, we cannot yet process this codeword because the formation of a codeword in bin 4 will result in a bit entering bin 3, which will break up the codeword in that bin.

There is more than one encoding method that conforms to the priority rules. Variations in the specific encoder implementation amount to changing the order in which certain processing operations are performed and using different methods for identifying codewords to be processed. All encoding methods that conform to the priority rules yield the same encoded bit stream, apart from differences arising due to different choices of flush bits.

To illustrate the priority rules, we can assign a label to each bit that indicates its priority. These priority labels are numbered like sections in a paper. Each source bit has a label equal to its position in the source sequence, and when a codeword is formed, the resulting output bits are given labels as though each were a "subsection" of the highest priority bit label appearing in the codeword.

Example 2.

(a) The ninth bit in a source sequence has priority label "9."

(b) A bit with priority label "3.4.2" has higher priority than a bit with label "4.1.1.2," just as section 3.4.2 would appear in a paper before section 4.1.1.2.

(c) Suppose a codeword is formed from bits with priority labels 5.1 and 9.1.2. If this codeword produces three output bits, then these bits have labels 5.1.1, 5.1.2, and 5.1.3, with higher priority labels assigned to bits generated closer to the root of the tree. △

According to our coder design rules, each output bit from a bin is mapped to a bin with lower index. Thus, Rule (4) allows us to process a codeword in a bin if the last (i.e., lowest-priority) bit in the codeword has higher priority than any bit in any higher-indexed bin. A simple way of guaranteeing this condition is to first place all source bits in the appropriate bins, then process codewords starting with those in the highest-indexed bin and working toward bin 1, so that each time we begin to process codewords in a bin, all bins with a higher index are empty. At each step we identify the nonempty bin with the largest index and we take bits (in priority order) from this bin until we have formed a codeword, appending flush bits if needed to complete the final codeword of the bin.

The following example illustrates this encoding procedure. (Shortly we will describe an alternative technique.)

[5] We assume for the sake of this example that source bits are assigned to bins using a rule different from the one used previously.

Example 3. We use coder design C5 to encode the sequence $0, 0, 0, 1$, which is the sequence just decoded in the previous section. First we place the input bits in the appropriate bins, according to our rule for assigning bin indices to input bits. We write the priority label of each bit in parentheses.[6]

1	2	3	4	5
0 (1)	0 (2)	0 (3)	1 (4)	

Next, we process codewords, starting at the highest-indexed nonempty bin. In this case, bin 4 contains a single bit that is in fact a codeword. From the tree for bin 4, we see that we should put bits 1,0,0 in bins 3,2,1, respectively. These bits are assigned labels 4.1, 4.2, 4.3, respectively. The encoder state now looks like this:

1	2	3	4	5
0 (1)	0 (2)	0 (3)	~~1 (4)~~	
0 (4.3)	0 (4.2)	1 (4.1)		

Next we process codewords in bin 3. Bin 3 contains 01, which (from the tree for bin 3) forms a codeword and produces ones in bins 2 and 1. Since the highest-priority bit in the codeword has priority label 3, the output bits are assigned labels 3.1 and 3.2, respectively, and are inserted in priority order in bins 2 and 1:

1	2	3	4	5
0 (1)	0 (2)	~~0 (3)~~	~~1 (4)~~	
1 (3.2)	1 (3.1)	~~1 (4.1)~~		
0 (4.3)	0 (4.2)			

In bin 2 we find the codeword 01, which produces a pair of ones in the first bin. The highest-priority bit in the codeword has label 2, so the output bits are labeled 2.1 and 2.2:

1	2	3	4	5
0 (1)	~~0 (2)~~	~~0 (3)~~	~~1 (4)~~	
1 (2.1)	~~1 (3.1)~~	~~1 (4.1)~~		
1 (2.2)	0 (4.2)			
1 (3.2)				
0 (4.3)				

Next, in bin 2 we have a 0 that is not a complete codeword, so we must add one or more flush bits to complete the codeword. Flush bits can be assigned priority label ∞, and any choice of flush bits that forms a complete codeword will produce a decodable sequence. In this case, we choose to append a 0, as this results in a codeword that produces a single output bit. The codeword now formed, 00, results in a 0 with label 4.2.1 in bin 1:

[6] It is a coincidence that in this case the input bit priority labels happen to equal the bin indices.

1	2	3	4	5
0 (1)	~~0 (2)~~	~~0 (3)~~	~~1 (4)~~	
1 (2.1)	~~1 (3.1)~~	~~1 (4.1)~~		
1 (2.2)	~~0 (4.2)~~			
1 (3.2)	~~0 (∞)~~			
0 (4.2.1)				
0 (4.3)				

At this point all bins except bin 1 are empty, so encoding is complete. Reading the bits in bin 1 in priority order, we see that the output sequence is 0,1,1,1,0,0, which was the input sequence to the decoder in the decoding example of the previous section. △

For encoding in software, rather than assigning priority labels to each bit, we maintain a linked list of bit values sorted in order of priority. Each record in the list stores a bit value and the index of the bin that contains the bit. Initially the list contains the entire input sequence in order of arrival. When a codeword is processed, we delete the bits that formed the codeword and insert the resulting output bits in the list at the location of the first bit in the codeword.

For example, suppose an encoder using design C5 has its linked list in the state shown in the left half of Fig. 4. The largest bin index in this list is 4, so we search through the list for bits in bin 4 until we form the codeword 01. This codeword produces output bits 1,0,1 in bins 3,2,1, respectively; the new state of the linked list is shown in the right half of Fig. 4. When all bits are in the first bin, the encoder output consists of these bits in priority order.

Instead of processing all codewords in the highest-indexed nonempty bin before moving to the next bin, we may use a recursive encoding variation that uses an alternative method of identifying codewords for processing, outlined in Fig. 5. The recursive `MakeCodeword` procedure of Fig. 5 forms and processes a codeword starting with a given bit. To do this, it scans through the bits in priority order to form a codeword, and forms and processes codewords in higher-indexed bins as needed. The procedure returns a pointer to the highest-priority output bit produced. In our experience, this recursive encoder implementation is faster than processing codewords in decreasing order of bin index.

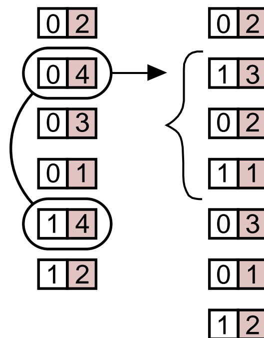

Fig. 4. One step of software encoding using coder design C5. In each pair, the left (unshaded) box indicates bit value; the right (shaded) box shows bin index.

```
Encode:
    initialization: load all source bits (with bin indices) into memory

    while (memory is not empty)
        let bitpointer point to the highest priority bit in memory
        while (bin(bitpointer) ≠ 1)
            let bitpointer = MakeCodeword(bitpointer)
        remove and output the bit indicated by bitpointer

MakeCodeword(bitpointer):
    thebin = bin(bitpointer)
    bitstring = bit(bitpointer)

    while (bitstring is not a complete codeword)
        if bitpointer points to the lowest priority bit in memory
            append flush bits as needed to make bitstring a complete codeword
        else
            assign bitpointer to the bit with next lower priority
            while (bin(bitpointer) > thebin)
                bitpointer = MakeCodeword(bitpointer)
            if (bin(bitpointer) = thebin)
                append bit(bitpointer) to bitstring

    delete from memory the bits that formed bitstring
    insert the corresponding output bits at the position occupied by the first bit in bitstring
    return a pointer to first output bit generated.
```

Fig. 5. Outline of an encoder implementation using the recursive `MakeCodeword` procedure.

C. Assigning Flush Bits

We now turn to the task of assigning flush bits when needed to complete the last codeword in a bin. When they are required, any choice of flush bits that forms a complete codeword will produce a decodable sequence. The decoder does not need to know the encoder's method of selecting flush bits—they are simply the bits remaining in the decoder after decoding is complete. This means that decoder speed is essentially unaffected by whether the encoder uses a quick or a highly optimized method to choose such bits.

We would like to choose flush bits in a way that minimizes the length of the encoded sequence. In general, however, this is not easy to do; in particular, the optimal assignment of flush bits in a given bin may depend on the contents of lower-indexed bins. For example, suppose the trees for bins 2 and 3 of a coder design are as shown in Fig. 6. If after processing as many complete codewords as possible we are left with a 0 in bin 3, and all other bins (except bin 1) are empty, then it is easily checked that we minimize encoded length by adding flush bits 00 in bin 3. On the other hand, if the encoder state includes a 0 in bin 3 as before, but now also includes a higher priority 0 in bin 2, then this time the encoded length is minimized by putting a 1 in bin 3. In fact, determining the optimal choice of flush bits could be much more complicated; for example, there could be a large number of bits in bin 2 while bin 3 contains a bit with higher priority.

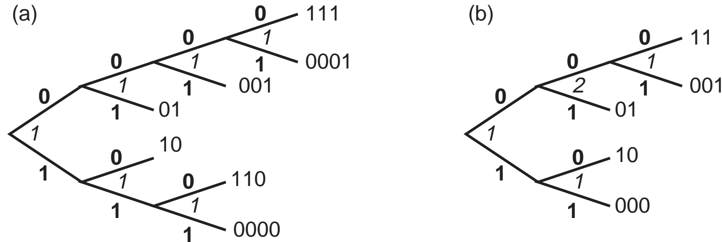

Fig. 6. Trees in a coder design illustrating the potential difficulty in optimally assigning flush bits: (a) bin 2 and (b) bin 3.

Thus, it apparently is difficult in general to optimally assign flush bits. Fortunately, the greedy assignment rule described below seems to produce an encoded length that is at worst only a few bits longer than the length obtained from optimal flush-bit assignment.

A greedy assignment selects flush bits to clear a given bin without regard for the contents of lower-indexed bins. Under this approach, we can compute flush-bit assignments in advance for each prefix of a codeword in each bin, and assign flush bits during encoding using a look-up table.

We now outline the greedy flush-bit assignment rule used for the simulations in this article. For each bin we identify a nominal probability value that closely matches the probability values of bits we expect to enter the bin. In a well-designed coder, a bit placed in a bin with nominal probability χ will ultimately result in a number of encoded bits approximately equal to

$$\begin{cases} -\log_2 \chi, & \text{if the bit is a 0} \\ -\log_2(1-\chi), & \text{if the bit is a 1} \end{cases}$$

For each codeword associated with a bin, we determine the corresponding output bits and associated bins and estimate the expected number of encoded bits using the above approximation. We think of this quantity as the "cost" of the codeword. Given a codeword prefix in a bin, we select the flush bits to minimize the cost of the codeword formed.

This greedy assignment rule is optimal for any bin that maps all output bits to the uncoded bin. This happens, for example, for the second bin in a recursive encoder, and for all bins in a non-recursive encoder. This method often, but not always, amounts to appending zeros to the partial codeword until a codeword is formed.

III. Useful Trees

Recall from Section I that a tree whose branches lack output-bit labels is useful at probability p if some assignment of output-bit labels results in all output bits having probability-of-zero in the range $[1/2, p)$ when the input bits all have probability-of-zero equal to p.

From the overview of coder operation given in Section I, we expect a bin to be associated with some probability interval, and we require that all output bits generated at a bin must be mapped to bins with strictly lower indices, which we expect to be associated with smaller probability values. Thus, if there are probability values for which useful trees do not exist, it may be difficult to produce coder designs achieving very small redundancies.

Fortunately, it turns out that there is essentially no restriction on the probability values at which a useful tree can be found. We will show that there exist families of trees such that, for any $p \in (1/2, 1)$,

there is a tree in the family that is useful at p. We call such a family *complete*. A complete family of trees is necessarily infinite.

In Section III.A we exhibit a complete family of trees. Section III.B presents a result that implies that any complete family of useful trees can be used to design a coder that meets an arbitrarily small redundancy target, provided that the source-bit probability-of-zero estimates are accurate. We give some additional results about useful trees in Section III.C.

A. Useful Trees Exist Everywhere

We now describe a particular family of useful trees. For integers $n > m \geq 1$, we form the tree $\mathcal{T}_{m,n}$ as shown in Fig. 7.[7] The assignment of output-bit labels that makes $\mathcal{T}_{m,n}$ useful at a given p depends on p, and we discuss these labelings in Appendix A.

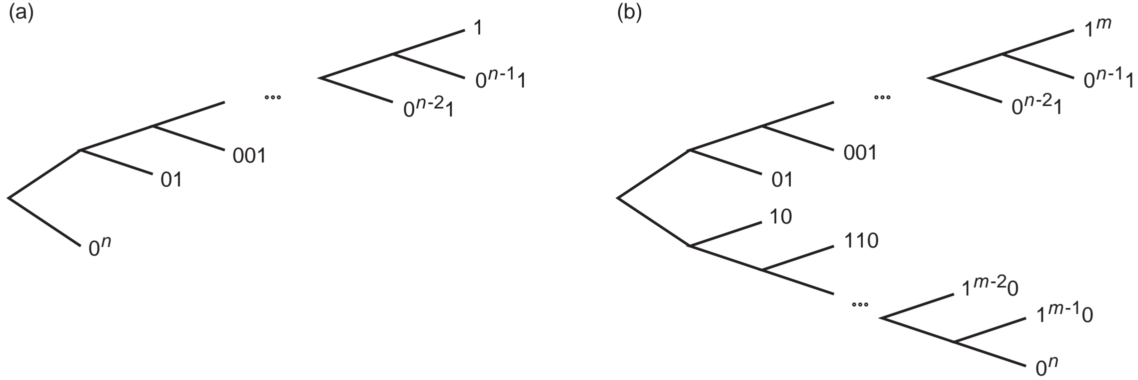

Fig. 7. The tree $\mathcal{T}_{m,n}$ for (a) m = 1 and (b) m > 1.

The following theorem identifies the range of probability values p for which $\mathcal{T}_{m,n}$ is useful. We prove an extended version of this theorem in Appendix A.

Theorem 1. *For $\alpha \in [0,1)$, let γ_α denote the root of $p = (1-p)^\alpha$ that is in the range $(1/2, 1]$. Then,*

(1) For $m = 1$ and $n > m$, $\mathcal{T}_{m,n}$ is useful in the interval $(\gamma_{m/n}, 1)$.

(2) For $n > m > 1$, $\mathcal{T}_{m,n}$ is useful in the interval $(\gamma_{m/n}, \gamma_{(m-2)/(n-2)})$.

Table 1 lists some values of γ_α. It is easily verified that $\gamma_{m/n}$ decreases as m/n approaches 1, and the following result implies that $\gamma_{m/n}$ approaches 1/2 as m/n approaches 1.

Proposition 1. *For $n > m \geq 1$,*

$$\frac{1}{2} < \gamma_{m/n} < \left(\frac{1}{2}\right)^{m/n}$$

Proof. Let $x_{m,n}(p) = p^n - (1-p)^m$, so that $\gamma_{m/n}$ is the zero in $(1/2, 1]$ of $x_{m,n}(p)$. Since $x_{m,n}(p)$ is increasing over $(0,1)$, it is sufficient to show that $x_{m,n}(1/2) < 0 < x_{m,n}\big((1/2)^{m/n}\big)$. The first inequality is immediate. To establish the second, note that $1 - (1/2)^{m/n} < 1/2$, so that

[7] The authors gratefully acknowledge Sam Dolinar for pointing out this generalization of the $\mathcal{T}_{m,m+1}$ trees.

Table 1. Some values of γ_α.

α	γ_α
0	1
1/2	$\frac{1}{2}\left(\sqrt{5}-1\right) \approx 0.61803$
2/3	0.56984
3/4	0.54970
4/5	0.53860
5/6	0.53156
6/7	0.52669
7/8	0.52312

$$x_{m,n}\left(\left(\frac{1}{2}\right)^{m/n}\right) = \left(\frac{1}{2}\right)^{m} - \left(1-\left(\frac{1}{2}\right)^{m/n}\right)^{m} > \left(\frac{1}{2}\right)^{m} - \left(\frac{1}{2}\right)^{m} = 0$$

❐

This result can be use to bound the rate at which $\gamma_{m/n}$ decays to 1/2. In particular,

$$\left(\frac{1}{2}\right)^{m/n} - \frac{1}{2} = \frac{1}{2}\left(e^{[(n-m)/n]\ln 2} - 1\right)$$

$$= \frac{1}{2}\left(\frac{n-m}{n}\ln 2 + \frac{1}{2}\left(\frac{n-m}{n}\ln 2\right)^{2} + \frac{1}{3!}\left(\frac{n-m}{n}\ln 2\right)^{3} + \cdots\right)$$

$$\approx \left(\frac{n-m}{n}\right)\frac{\ln 2}{2}$$

where the approximation is valid if m/n is close to 1.

It is clear from Proposition 1 and Theorem 1 that the set of all $\mathcal{T}_{m,n}$ trees is complete. Furthermore, for any fixed integer $d > 0$, the set $\{\mathcal{T}_{m,m+d}\}_{m=1}^{\infty}$ forms a complete family of useful trees. Figure 8 illustrates how the useful regions for the trees in the important family $\{\mathcal{T}_{m,m+1}\}_{m=1}^{\infty}$ span $(1/2, 1)$.

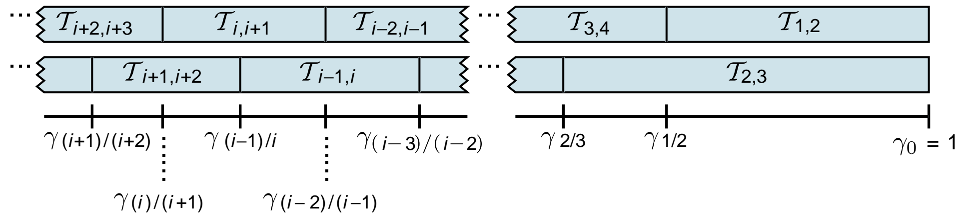

Fig. 8. Useful regions of trees in the family $\{\mathcal{T}_{m,m+1}\}_{m=1}^{\infty}$. (Not to scale.)

B. Application of Useful Trees to Coder Design

We noted in Section I.C that redundancy can be made arbitrarily small (subject to the accuracy of the source distribution estimates) using non-recursive interleaved entropy coders; however, the codes required quickly become complex. On the other hand, we saw in Section III.A that we can easily construct a complete family of useful trees with manageable complexity. The following theorem establishes that such a family of trees provides another method of producing coder designs achieving arbitrarily small redundancy when we exploit recursion in the coder.

Theorem 2. *Let $\mathcal{U}$ be a complete family of useful trees without output-bit specifications. Then for any $\varepsilon > 0$ and $\delta > 0$ there exists a coder design that uses only trees from $\mathcal{U}$, and a constant c, for which the following holds: For any n and any sequence of bits $b_1, \cdots, b_n$ whose associated probability-of-zero estimates $p_1, \cdots, p_n$ are all in the range $[\delta, 1-\delta]$, the coder will compress the sequence to at most*

$$c + (1+\varepsilon) \sum_{i=1}^{n} \left\{ \begin{array}{ll} -\log_2 p_i, & \text{if } b_i = 0 \\ -\log_2 (1-p_i), & \text{if } b_i = 1 \end{array} \right\} \quad \text{bits} \tag{1}$$

This theorem is proved in Appendix B.

It should be emphasized that this theorem is not probabilistic and there need not be any relation between the bit values and their probability-of-zero estimates. However, if, for example, we generate independent random bits $b_1, \cdots, b_n$ and let $p_i = \text{Prob}[b_i = 0]$, then the theorem implies that the expected number of encoded bits per source bit can be made to approach the source entropy. Similar results hold under more complicated assumptions, so long as the probability estimates are accurate for each bit. Even if the estimates are not accurate, the sum in Eq. (1) represents, in a rather loose sense, the best average encoded length achievable by a coder that relies on the estimates [11]. An ideal arithmetic coder that uses the same probability estimates would produce encoded length slightly larger than the sum in Expression (1).

Using Theorem 2, it can be shown that redundancy can be made arbitrarily small in other senses as well. For example, assuming accurate probability estimates, the average number of bits of redundancy per source bit can be made arbitrarily small without any restriction on the source-bit probabilities.

In the proof of Theorem 2, the coder design produced may have many more bins than necessary; no attempt is made to keep complexity low. Section V.A presents a more practical procedure to design a coder that meets a redundancy constraint using a small number of bins.

We use the following lemma in our discussion of a coder design algorithm in Section V.A. This lemma implies that if a tree is useful at p and the tree is used with input bits having probability-of-zero p, then the expected number of input bits in a codeword is greater than the expected number of output bits produced by processing a codeword; thus, the tree contributes to data compression.

Lemma 1. *Suppose a tree is useful at p. Let $\eta_k(p)$ be the expected number of output bits produced at node k per input bit when the input bits are independent and have probability-of-zero p. Then*

$$\sum_k \eta_k(p) < 1 \tag{2}$$

where the sum is over all non-terminal nodes of the tree.

Note that Expression (2) might hold even for a tree that is not useful at p.

Proof. To calculate the entropy of a discrete probability distribution, we can assign each possible outcome to a leaf of a binary tree. Then the grouping property of entropy allows us to calculate the entropy of the distribution from the entropies associated with the individual binary decisions leading to the leaves.

Here we consider the probability distribution on the input codewords given that input bits have probability-of-zero p. Let H be the entropy of this distribution and let L be the expected input codeword length. We use two binary trees associated with the distribution on the input codewords: the tree from the input codewords themselves (in which the binary decisions are input bits) and the tree of the type we use to describe coders, in which the binary decisions correspond to output bits.

For the first type of tree, the entropy of each binary decision is $\mathcal{H}_2(p)$, where $\mathcal{H}_2$ is the binary entropy function, and the expected number of decisions is L. Thus, we have

$$H = L\mathcal{H}_2(p)$$

For the second type of tree, let $q_k(p)$ denote the probability that an output bit from node k of the tree is zero. Then the entropy of the binary decision at node k is $\mathcal{H}_2(q_k(p))$, and the probability that this decision is encountered in forming a codeword is $L\eta_k(p)$. So in this case we have

$$H = \sum_k L\eta_k(p)\mathcal{H}_2(q_k(p))$$

where the sum is over all non-terminal nodes.

Equating the two expressions for H and dividing by L yields

$$\mathcal{H}_2(p) = \sum_k \eta_k(p)\mathcal{H}_2\big(q_k(p)\big) \tag{3}$$

The fact that the tree is useful at p implies that $\mathcal{H}_2(q_k(p)) > \mathcal{H}_2(p)$ for each k, thus

$$\mathcal{H}_2(p) = \sum_k \eta_k(p)\mathcal{H}_2\big(q_k(p)\big) > \sum_k \eta_k(p)\mathcal{H}_2(p)$$

and dividing by $\mathcal{H}_2(p)$ yields Expression (2). ❐

C. Useful Trees and Practical Coding

There are many useful trees in addition to those described in Section III.A. In this section, we give some examples of such trees and we provide a few results that might be useful in characterizing the class of useful trees.

For p near 1, there are several useful trees that perform runlength coding and appear to be well suited for use in practical coders. Figure 9 illustrates two such trees.

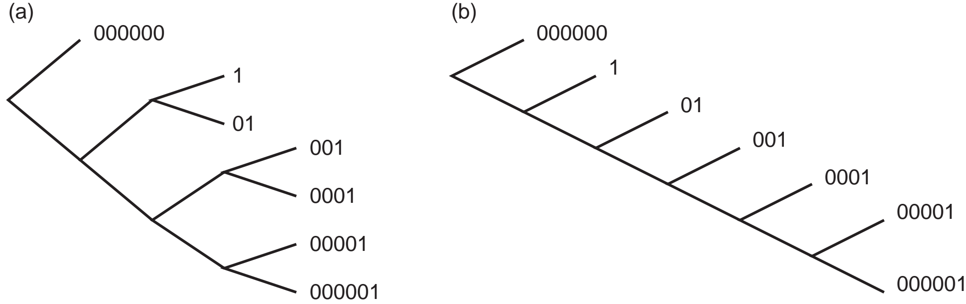

Fig. 9. Examples of useful trees that perform runlength coding. Tree (a) performs, essentially, Golomb coding [8], and tree (b) is formed by rearranging the codewords of $\mathcal{T}_{1,6}$.

Two trees that are both useful in some common region can be combined to construct a new useful tree. For example, $\mathcal{T}_{2,3}$ is useful for $p \in (\gamma_{2/3}, 1)$ and $\mathcal{T}_{1,2}$ is useful for $p \in (\gamma_{1/2}, 1)$. If we join $\mathcal{T}_{1,2}$ to $\mathcal{T}_{2,3}$ at the terminal node of $\mathcal{T}_{2,3}$ corresponding to codeword 001, then we use this codeword as a prefix of each codeword in tree $\mathcal{T}_{1,2}$. The resulting composite tree, shown in Fig. 10, is useful in the region $(\gamma_{2/3}, 1) \cap (\gamma_{1/2}, 1) = (\gamma_{1/2}, 1)$. Such composite trees do not appear to have any significant advantages over their component trees.

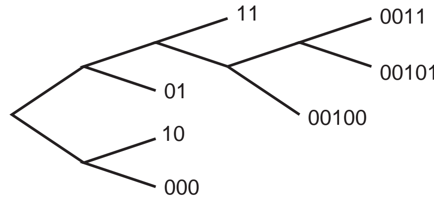

Fig. 10. A composite useful tree formed by combining the trees $\mathcal{T}_{2,3}$ and $\mathcal{T}_{1,2}$.

Useful trees can be combined in another manner to produce a second class of composite trees. Let T_a be a tree that is useful at p and for which the output probability-of-zero from the root node is within the useful region of a tree T_b. Suppose we form and process codewords with T_a until the output bits from the root node form an input codeword for T_b, and then we process this codeword using T_b. This sequence of operations is equivalent to forming and processing a codeword with a single larger tree that is derived from T_a and T_b and is useful at p.

If the bits arriving in a bin are independent and identically distributed (IID), then exchanging the positions of two codewords that have equal numbers of zeros and equal numbers of ones will have no impact on compression efficiency.[8] Thus, the resulting tree can be considered to be equivalent to the original tree. Figure 11 shows an example of two equivalent useful trees.

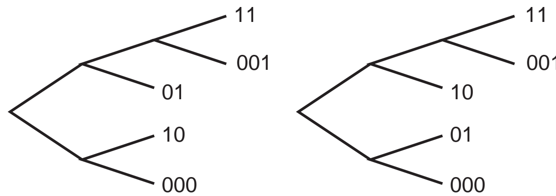

Fig. 11. Two equivalent useful trees.

[8] Note, however, that bits arriving in a bin may not be IID even when the source bits are IID.

We have not found a simple characterization of the set of all useful trees. However, we have enumerated all useful trees with seven or fewer terminal nodes. In Table 2 we list the number of useful trees with up to seven terminal nodes when we eliminate the first type of composite trees and select a single tree from each set of equivalent trees. Figure 12 shows the five useful trees with four terminal nodes.

Table 2. Number of useful trees with *n* terminal nodes, excluding the first type of composite trees and selecting a single member from each set of equivalent trees.

n	Number of useful trees with n terminal nodes
3	1
4	5
5	39
6	416
7	5368

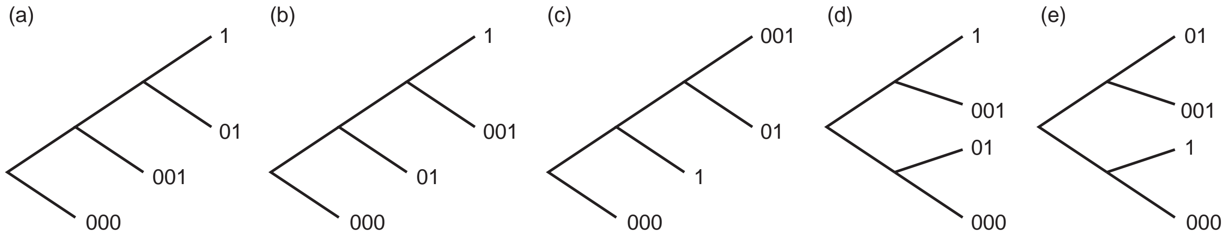

Fig. 12. The useful trees with four terminal nodes. Tree (a) is useful in the range (0.755,1); trees (b), (c), and (d) are useful in (0.682,1); and tree (e) is useful in (0.618,1).

It may be advantageous to consider trees formed from modifications to the $\mathcal{T}_{m,n}$ trees. For example, starting from $\mathcal{T}_{2,3}$, we can extend the tree at the nodes corresponding to codewords 10 and 001 (essentially by appending uncoded bits) and swap codewords 00100 and 1011 to arrive at the tree shown in Fig. 13. This tree is useful over $(\gamma_{2/3}, 1)$, like $\mathcal{T}_{2,3}$, but over this region it tends to produce output bits with probabilities-of-zero closer to 1/2. The much simpler modification of rearranging the codewords can sometimes give a similar improvement.

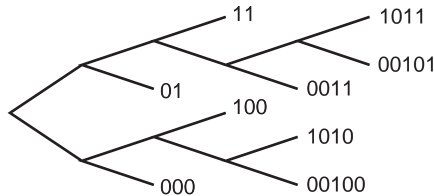

Fig. 13. A useful tree formed by extending $\mathcal{T}_{2,3}$.

IV. Estimating Compression Efficiency

In this section, we turn to the problem of quantifying the compression efficiency of a recursive interleaved entropy coder. We would like to analytically determine the redundancy resulting from a given coder design, but this depends on the source and does not appear to be easy to calculate exactly in general.

One metric that gives a good indication of performance, and for which we can find reasonable estimates, is the rate (the expected number of output bits per source bit) when the input to the encoder is an IID stream of bits into bin j, each bit having probability-of-zero equal to p. We denote this quantity by $R_j(p)$.

Because of the recursive nature of the encoder, our estimates of $R_j(p)$ rely on rate estimates for other bins. A given bin may have bits with different probabilities-of-zero arriving from higher-indexed bins. If bin j has as input λ_1 bits each with probability-of-zero q_1 and λ_2 bits each with probability-of-zero q_2, then the resulting contribution to the number of output bits might be approximated as

$$\lambda_1 R_j(q_1) + \lambda_2 R_j(q_2) \tag{4}$$

or

$$(\lambda_1 + \lambda_2) R_j\left(\frac{\lambda_1 q_1 + \lambda_2 q_2}{\lambda_1 + \lambda_2}\right) \tag{5}$$

The first approximation would tend to be more accurate when long runs of bits in bin j have the same probability-of-zero, and the second would be more accurate if the two types of bits are well mixed.

In this section, we describe two techniques for estimating $R_j(p)$ that are direct applications of the respective approximations above. Extensions of these techniques can be used to accurately estimate the rate obtained for a source that produces bits with varying (but known) distributions. The rate estimates produced are asymptotic as the input sequence length becomes large, i.e., the cost of bits used to flush the encoder is not included. We examine the cost of flush bits in Section IV.D.

The rate estimation techniques do not usually give exact results, in part because bits arriving in a bin may not be independent even when the source bits are independent. This dependence can arise, for example, when processing a single codeword results in more than one output bit being placed in the same bin. Exact rate expressions can be obtained in certain cases; for example, $R_1(p) = 1$ exactly since bits in the first bin are uncoded, and both techniques compute $R_j(p)$ exactly for any bin j that maps all output bits to the first bin. In practice, both techniques usually give quite good estimates.

As a first step for either technique, at each non-terminal node in each tree we need to compute η, the expected number of output bits at the node per input bit to the bin, and q, the probability that an output bit from the node is zero. Both quantities are functions of p, the probability-of-zero at the input to the bin. To compute η, we determine the expected number of output bits at the node per codeword and divide this by the expected codeword length.

Example 4. Consider the tree for bin 4 of coder design C5, shown again in Fig. 14. The codeword set is $\{1, 01, 001, 000\}$, and the expected codeword length is

$$L = (1-p) + 2p(1-p) + 3p^2(1-p) + 3p^3 = 1 + p + p^2$$

Each codeword produces one output bit at node a, so the expected number of output bits per input bit at this node is $\eta = 1/L$, and the probability that the output bit at node a is a zero is p^3. So at node a we

have the pair $(\eta, q) = \big(1/[1+p+p^2], p^3\big)$. A codeword produces an output bit at node b with probability $1-p^3$, so the expected number of output bits per input bit at node b is $(1-p^3)/L = 1-p$. The probability that an output bit produced at node b is zero is equal to $(1+p)/(1+p+p^2)$. After performing similar calculations for node c and for the trees for bins 2 and 3 of coder design C5, we arrive at the (η, q) pairs shown in Fig. 15. △

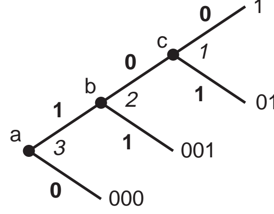

Fig. 14. The tree for bin 4 of coder design C5.

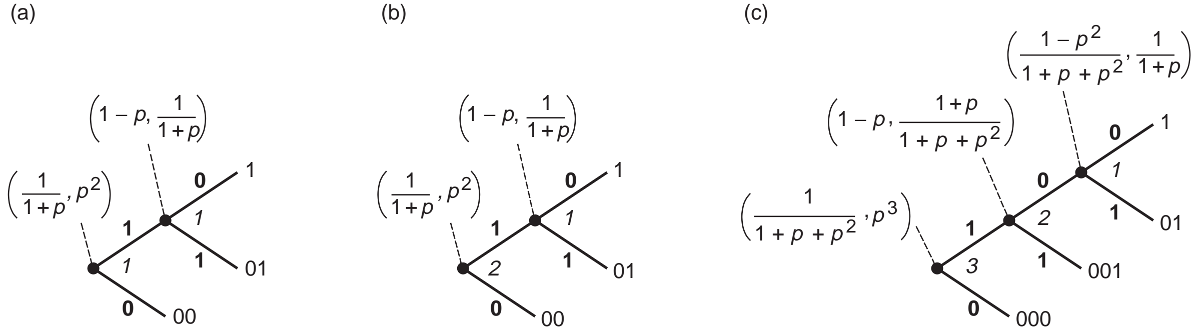

Fig. 15. Expected number of output bits per input bit and probability that an output bit is zero, at the non-terminal nodes of three of the trees used in coder design C5: (a) bin 2, (b) bin 3, and (c) bin 4.

A. First Method for Rate Estimation

In our first method of rate estimation, we estimate $R_j(p)$ from the estimates for $R_1(p)$, $R_2(p), \cdots$, $R_{j-1}(p)$. For non-terminal node k of the tree for bin j, let $\eta_k(p)$ denote the expected number of output bits per input bit, $q_k(p)$ denote the probability-of-zero of these bits, and B_k denote the destination bin of these bits. We use the estimate

$$R_j(p) = \begin{cases} \sum\limits_k \eta_k(p) R_{B_k}\big(q_k(p)\big), & \text{if } j > 1 \\ 1, & \text{if } j = 1 \end{cases} \tag{6}$$

where the sum is over all non-terminal nodes in the tree.

Example 5. We compute an estimate of $R_4(p)$ for coder design C5, making use of the (η, q) pairs exhibited in Fig. 15. We begin with $R_1(p) = 1$. Next we estimate

$$R_2(p) = \frac{1}{1+p} R_1(p^2) + (1-p) R_1\left(\frac{1}{1+p}\right) = (1-p)\left(1 + \frac{1}{1-p^2}\right)$$

Similarly, for bin 3:

$$R_3(p) = \frac{1}{1+p} R_2(p^2) + (1-p) R_1 \left(\frac{1}{1+p} \right) = (1-p) \left(2 + \frac{1}{1-p^4} \right)$$

and finally for bin 4:

$$R_4(p) = \frac{1}{1+p+p^2} R_3(p^3) + (1-p) R_2 \left(\frac{1+p}{1+p+p^2} \right) + \frac{1-p^2}{1+p+p^2} R_1 \left(\frac{1}{1+p} \right)$$

$$= (1-p) \left(4 - \frac{1+p}{2+2p+p^2} + \frac{1}{1-p^{12}} \right)$$

△

We can use a modification of this rate estimation technique to derive a compact recursive formula for an estimate of redundancy, where the redundancy $\rho_j(p)$ is the number of output bits per input bit in excess of the binary entropy function. For bin 1 the redundancy is $1 - \mathcal{H}_2(p)$, and for $j > 1$ the redundancy for bin j is approximately

$$\rho_j(p) = R_j(p) - \mathcal{H}_2(p) = \sum_k \eta_k(p) \left[R_{B_k}\big(q_k(p)\big) - \mathcal{H}_2\big(q_k(p)\big) \right] = \sum_k \eta_k(p) \rho_{B_k}\big(q_k(p)\big) \tag{7}$$

where we have used Eqs. (3) and (6). This expression motivates the coder design procedure described in Section V.A.

B. Second Method for Rate Estimation

Our second technique for estimating $R_j(p)$ is based on Expression (5). For each bin we produce a list of (λ, q) pairs, where in each pair λ represents an expected number of bits per source bit and q is the corresponding probability-of-zero. Initially each list is empty except the list for bin j, which contains the pair $(1, p)$.

We now let ℓ step through bin indices, starting with $\ell = j$ and working down through $\ell = 2$. At a given step, suppose the list for bin ℓ contains pairs (λ_1, q_1), $(\lambda_2, q_2), \cdots$, (λ_m, q_m). We compute the total expected number of bits in the bin per source bit,

$$\Lambda_\ell = \sum_{i=1}^{m} \lambda_i \tag{8}$$

and the average probability-of-zero in the bin,

$$Q_\ell = \frac{1}{\Lambda_\ell} \sum_{i=1}^{m} q_i \lambda_i$$

Treating the input to bin ℓ as Λ_ℓ bits, each with probability-of-zero Q_ℓ, we compute the expected number of bits λ'_k and associated probability-of-zero q'_k at each non-terminal node k in the tree and append (λ'_k, q'_k) to the list for the bin to which the output bit associated with node k is mapped.[9] This calculation uses the

[9] If bits generated at node k are assigned to bin 1, then computation of q'_k is not necessary.

originally computed (η, q) pairs associated with the non-terminal nodes in the tree. Then we decrement ℓ to move to the next bin.

After we finish stepping through the bins, our estimate of $R_j(p)$ equals Λ_1, the total expected number of bits in the first bin computed using Expression (8).

This technique can easily be adapted to estimate the rate associated with a source that produces bits with varying (but known) distributions. For a source that produces bits with probabilities-of-zero $\phi_1, \cdots, \phi_k$ with frequencies $f_1, \cdots, f_k$, we simply initialize our lists so that each pair (f_i, ϕ_i) is put in the list for the bin to which probability ϕ_i is mapped.

Example 6. To illustrate this rate estimation method, we compute again an estimate of $R_4(p)$ for coder design C5. Initially the list for bin 4 is $\{(1, p)\}$, and the lists for all other bins are empty. From Fig. 15(c) we see that this produces pairs $\big(1/[1+p+p^2], p^3\big)$, $\big(1-p, [1+p]/[1+p+p^2]\big)$, and $\big([1-p^2]/[1+p+p^2], 1/[1+p]\big)$ in bins 3, 2, and 1, respectively. The updated lists are

$$\text{bin } 3 : \left\{ \left(\frac{1}{1+p+p^2}, p^3 \right) \right\}$$

$$\text{bin } 2 : \left\{ \left(1-p, \frac{1+p}{1+p+p^2} \right) \right\}$$

$$\text{bin } 1 : \left\{ \left(\frac{1-p^2}{1+p+p^2}, \frac{1}{1+p} \right) \right\}$$

Moving to bin 3, we find a total expected number of bits $\Lambda_3 = 1/(1+p+p^2)$ and probability-of-zero $Q_3 = p^3$. Bin 3 produces [see Fig. 15(b)] pairs $\big(\Lambda_3/[1+Q_3], Q_3^2\big) = \big([1-p]/[1-p^6], p^6\big)$ and $(\Lambda_3[1-Q_3], 1/[1+Q_3]) = \big(1-p, 1/[1+p^3]\big)$ in bins 2 and 1, respectively. We add these pairs to the appropriate lists:

$$\text{bin } 2 : \left\{ \left(1-p, \frac{1+p}{1+p+p^2} \right), \left(\frac{1-p}{1-p^6}, p^6 \right) \right\}$$

$$\text{bin } 1 : \left\{ \left(\frac{1-p^2}{1+p+p^2}, \frac{1}{1+p} \right), \left(1-p, \frac{1}{1+p^3} \right) \right\}$$

Next, in bin 2 we compute

$$\Lambda_2 = 1-p+\frac{1-p}{1-p^6} = (1-p)\left(1+\frac{1}{1-p^6}\right)$$

$$Q_2 = \frac{1}{\Lambda_2}\left(\frac{1-p^2}{1+p+p^2} + \frac{(1-p)p^6}{1-p^6}\right) = \frac{1-p^2+p^3-p^5+p^6}{2-p^6}$$

From Fig. 15(a) we see that bin 2 produces pairs

$$\left(\frac{\Lambda_2}{1+Q_2}, Q_2^2\right) = \left(\frac{(1-p)(2-p^6)^2}{(1-p^6)(3-p^2+p^3-p^5)}, \frac{(1-p^2+p^3-p^5+p^6)^2}{(2-p^6)^2}\right)$$

and

$$\left(\Lambda_2(1-Q_2), \frac{1}{1+Q_2}\right) = \left(\frac{(1-p)(1+p+2p^2)}{1+p+p^2}, \frac{2-p^6}{3-p^2+p^3-p^5}\right)$$

both in bin 1. So the list for bin 1 becomes

$$\text{bin 1}: \left\{ \left(\frac{1-p^2}{1+p+p^2}, \frac{1}{1+p}\right), \left(1-p, \frac{1}{1+p^3}\right), \left(\frac{(1-p)(2-p^6)^2}{(1-p^6)(3-p^2+p^3-p^5)}, \frac{(1-p^2+p^3-p^5+p^6)^2}{(2-p^6)^2}\right), \right.$$

$$\left. \left(\frac{(1-p)(1+p+2p^2)}{1+p+p^2}, \frac{2-p^6}{3-p^2+p^3-p^5}\right)\right\}$$

The rate is estimated as the total expected number of bits in bin 1:

$$R_4(p) = \frac{1-p^2}{1+p+p^2} + 1 - p + \frac{(1-p)(2-p^6)^2}{(1-p^6)(3-p^2+p^3-p^5)} + \frac{(1-p)(1+p+2p^2)}{1+p+p^2}$$

$$= (1-p)\left(3 + \frac{(2-p^6)^2}{(3-p^2+p^3-p^5)(1-p^6)}\right)$$

For $p \in (1/2, 1)$, this estimate is slightly lower than the estimate computed using the first technique. △

C. Performance Example

The rate estimation techniques of Sections IV.A and IV.B allow computation of estimates of the redundancy $\rho_j(p) = R_j(p) - \mathcal{H}_2(p)$ for each bin j of a coder. We typically plot redundancy for a coder design as a function of p, under the assumption that source bits are mapped to the bin that minimizes the estimated rate. Generally, each $R_j(p)$ is nearly linear in p, so $\min_j R_j(p)$ is nearly piecewise linear. The rate function $\min_j R_j(p)$ tends to hug the binary entropy curve, as illustrated in Fig. 16, resulting in a redundancy curve with a sawtooth appearance.

To demonstrate the accuracy of the estimation techniques, we have estimated redundancy for two coder designs: C5, and a 23-bin coder that uses only $\mathcal{T}_{m,m+1}$ trees. In addition, we have measured the actual redundancy by simulation. Figures 17 and 18 show the results. The results of the two estimation techniques are indistinguishable at the scale of the figures and are in close agreement with the measured redundancy. For bins 1–3 of coder design C5, both redundancy estimates can be shown to be equal to the actual redundancy.

We remark that plots of redundancy versus p may be misleading when source bits entering a bin do not all have the same probability-of-zero. Since $\mathcal{H}_2(p)$ is convex $\cap$, the average entropy of these incoming bits will be less than the binary entropy function evaluated at their average probability-of-zero. Thus,

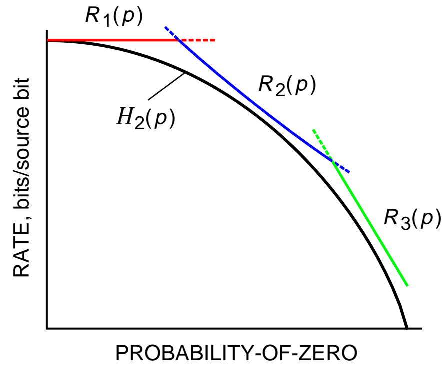

Fig. 16. Rate functions $R_j(p)$ and binary entropy $H_2(p)$.

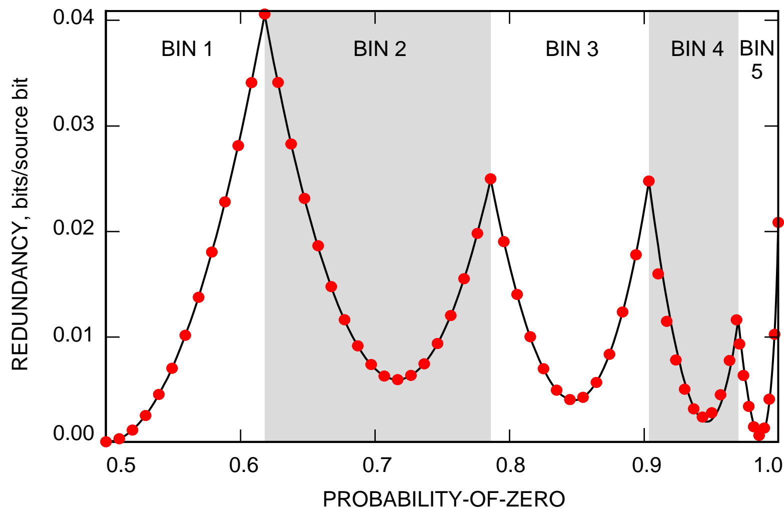

Fig. 17. Estimated (solid curve) and measured (individual points) redundancy for coder design C5. Each point was generated using 500 sequences, each of length 2^{20} bits.

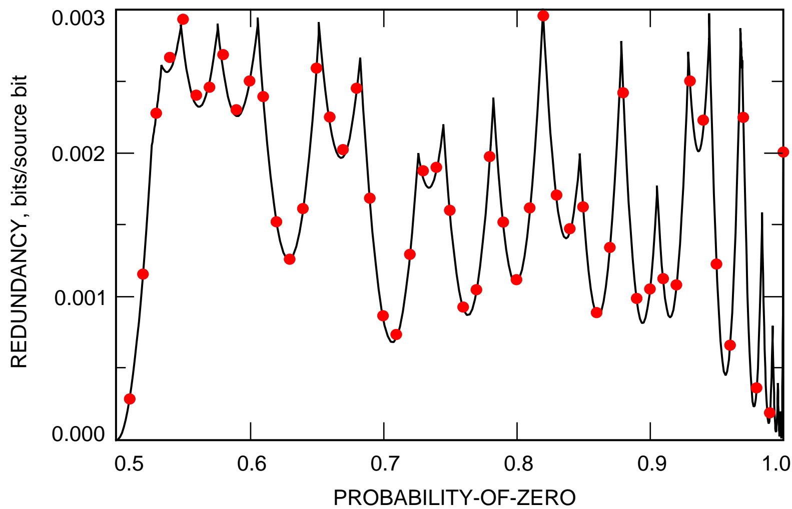

Fig. 18. Estimated (solid curve) and measured (individual points) redundancy for a 23-bin coder. Each point was generated using 1000 sequences, each of length 2^{20} bits.

the redundancy could be larger than one might expect from examining a plot of redundancy versus p; see Fig. 19. In theory, this redundancy could be larger than the redundancy at any fixed probability-of-zero in the bin.

In practice, the discrepancy is usually quite small since the intervals associated with bins are generally narrow. However, we note that a 2-bin coder with a very large tree derived from a universal source code would appear from a redundancy plot to give very low redundancy, when in fact the redundancy would be high if incoming bits had widely varying probabilities-of-zero.

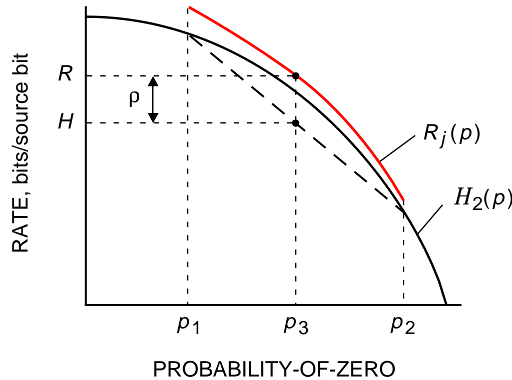

Fig. 19. If bits arriving in bin *j* have probabilities-of-zero p_1 and p_2, then the entropy, *H*, and rate achieved, *R*, might be as shown. The associated redundancy ρ could be larger than might be expected from a graph that shows redundancy as a function of *p*.

D. Rate Cost of Flush Bits

Now that we have methods of estimating the asymptotic rate of a given coder design, we turn to the non-asymptotic component of rate: the contribution due to flush bits. The cost, in bits of output, of flush bits is usually quite small unless the number of bins in the coder is very large; thus, unless the source sequence is short, the contribution to rate is generally minimal. If one wishes to compress several short sequences of bits independently, the cost of flush bits can become important. This might arise as a method of accommodating a constraint in encoder memory; note, however, that in [1] we present an alternate method of handling encoder memory limitations.

For a non-recursive coder and a given source model, we can analytically compute the probability distribution on codeword prefixes remaining in each bin before flush bits are added. This allows us to calculate the exact expected rate cost of flush bits, which is simply the sum of the expected cost in each bin. This calculation is tractable because no interactions between bins arise.

For a recursive coder, calculating the expected rate cost of flush bits is more difficult, in part because the recursive nature of the coder makes it very difficult to model the stream of bits arriving in each bin. A simpler alternative is to compute the rate cost approximately, by assuming that all bits arriving in a bin have the same nominal probability-of-zero and calculating the distribution of codeword prefixes and the bit cost of flush bits based on this probability.

Another alternative is to measure the bit cost of flush bits by simulation, measuring the encoded length for moderate-length source sequences and subtracting the asymptotic component. For most sources, we expect the distribution of codeword prefixes in the encoder before flushing to approach some steady-state distribution as the number of bits encoded becomes large. Thus, we anticipate that the expected cost of flush bits approaches a constant as the sequence length becomes long.

As an example, we have used this simulation method to find the average cost of flush bits for the 6-bin and 9-bin coders specified in Table C-2 of Appendix C when for each bit the probability-of-zero is chosen randomly according to a given distribution. We used a uniform distribution over $[1/2, 1]$ and a triangular distribution over $[1/2, 1]$ with the peak at 1. The results are shown in Fig. 20. After some erratic behavior for very short sequences, the bit cost does indeed appear to approach a constant. This constant appears to have only a small dependence on the distribution from which the probabilities-of-zero were selected; we would expect this to be the case among distributions that cause all bins to be used. The small difference that does occur can be largely attributed to the fact that the different distributions give rise to slightly different average probabilities-of-zero within bins.

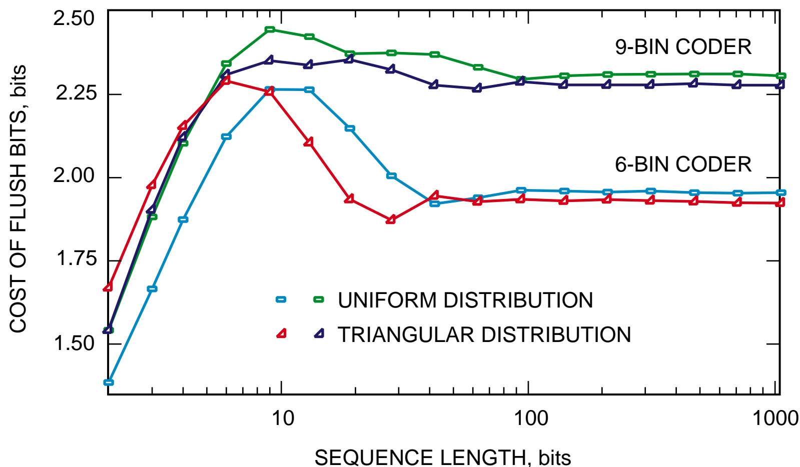

Fig. 20. Cost, in bits per encoded sequence, of the flush bits needed to complete encoding for the 6-bin and 9-bin coders specified in Table C-2 of Appendix C. The probability-of-zero of each bit is independently chosen from the indicated distribution. Each point was generated using 200,000 sequences.

V. Designing Coders

In this section, we describe a practical procedure for finding a coder design whose redundancy meets a given goal, and we exhibit some coder designs obtained using this design procedure.

A. A Coder Design Procedure

As discussed in Section IV, evaluating the exact redundancy of a given coder design is a difficult problem. Thus, we consider only the redundancy obtained for sources with a fixed probability-of-zero, and we rely on the rate estimates of Section IV, which in practice are quite close to the rates achieved.

The redundancy goal is specified by a quantity Δ that represents the maximum allowed asymptotic redundancy, in bits per source bit. The procedure also requires a set of candidate trees to be used in the coder. In this context, the trees do not include assignments of bin indices to non-terminal nodes or output bit labels to branches; these assignments will be made as part of the design procedure. To use the design procedure with arbitrarily small values of Δ, the set of candidate trees should have the property that for any $p \in (1/2, 1)$ some tree in the set is useful at p—that is, the set contains a complete family of useful trees. In practice, it is often convenient to limit consideration to a finite candidate tree set.

Each bin j will have associated with it an interval specified by the left endpoint z_{j-1} and right endpoint z_j. For $j > 1$, a tree will also be associated with bin j. No design work is required for bin 1, since $z_0 = 1/2$ and bin 1 is uncoded. To design the rest of the coder, we add bin specifications in order of increasing

bin index by selecting a tree, assigning bin indices and output bit labels to the tree, and computing the left endpoint of the bin's interval. For example, Fig. 21 shows a case where the coder design has been specified for the first three bins, and our redundancy target Δ is met for source probabilities-of-zero less than p^*. The next task is to specify a tree that meets the redundancy target for an interval that includes p^*; as a result we will have $z_3 \leq p^*$.

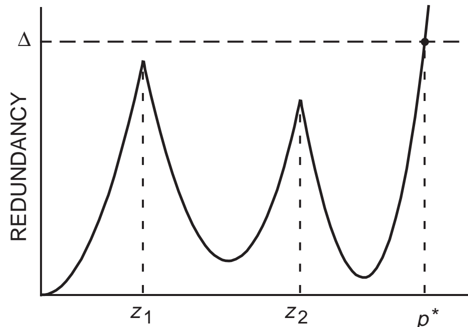

Fig. 21. Redundancy of a coder after specifying trees for bins 2 and 3.

Given that the first $j-1$ bins of the coder satisfy the redundancy target up to probability-of-zero p^*, the tree for bin j needs to yield redundancy of at most Δ at p^*. This can be accomplished by selecting a tree that is useful at p^* and, treating the input bits to the tree as all having probability-of-zero p^*, assigning branch labels so that each output bit is more likely to be a 0 than a 1. Then, at each non-terminal node, we assign the output bin index ℓ so that the probability-of-zero of the output bit is in the interval $[z_{\ell-1}, z_\ell)$ (for this assignment we temporarily let $z_j = p^*$). This construction maps output bits to bins in regions where the redundancy is less than the target Δ, and so, using the redundancy approximation of Eq. (7), the redundancy at p^* satisfies

$$\rho_j(p^*) = \sum_k \eta_k(p^*)\rho_{B_k}(q_k(p^*)) < \sum_k \eta_k(p^*)\Delta < \Delta$$

where $\eta_k(p^*)$ is the expected number of output bits generated at non-terminal node k for each input bit, $q_k(p^*)$ is the corresponding probability-of-zero, and the sums are over all non-terminal nodes. This last inequality is due to Lemma 1 of Section III. Thus, to the extent that the approximation in Eq. (7) is accurate, the tree we have selected for the new bin produces redundancy less than Δ at p^*. Since the rate function for each bin is continuous, we have extended the range where the coder meets the redundancy target.

The above procedure can always be used to find one or more trees (labeled with output bits and destination bins) that extend the range over which the coder meets the redundancy target. However, additional trees, or trees with alternate output bit and bin assignments, might also meet the redundancy goal at p^*. A possible method of finding such trees is to select a tree and assign bins and output bits as in the above procedure, but using a target probability somewhat larger than p^*.

A reasonable method of choosing among several trees that meet the redundancy goal at p^* is to pick the tree that extends the useful range of the coder the furthest. The procedure is then essentially a greedy algorithm, and there is no guarantee that the number of bins in the coder will be minimized. In addition, the design procedure does not give consideration to minimizing encoding or decoding complexity, which may be quite different from minimizing the number of bins.

B. Coder Designs

We have used the method of Section V.A to find several coder designs, many of which appear to be promising. A group of coder designs was generated with the trees $\{\mathcal{T}_{m,m+1}\}_{m=1}^{\infty}$ described in Section III.A as the set of candidate trees. Table 3 indicates some parameters and measures of redundancy of these coder designs, and Fig. 22 contains a plot of redundancy versus probability-of-zero for some of them. Table C-1 in Appendix C gives complete specifications.

Table 3. Parameters and redundancies of some coder designs that use only trees from $\{\mathcal{T}_{m,m+1}\}_{m=1}^{\infty}$.

Bins	Maximum redundancy[a]	Typical redundancy[b]	Number of terminal nodes in the trees
2	1/2	0.083	3
3	1/4	0.029	3,3
4	1/8	0.015	3,3,3
5	1/16	0.012	3,3,3,3
6	0.041	0.011	3,3,3,3,3
7	1/36	0.0067	5,5,3,3,3,3
8	0.019	0.0071	5,5,5,3,3,3,3
9	0.014	0.0060	5,5,5,3,3,3,3,3
10	3/256	0.0048	7,5,3,5,3,3,3,3,3
11	0.010	0.0043	7,5,5,3,5,3,3,3,3,3
12	0.0080	0.0034	7,3,3,5,3,3,3,3,3,3,3

[a] "Maximum redundancy" refers to the maximum estimated redundancy among sources having a fixed probability-of-zero, given that the source bits are mapped to the bin that minimizes redundancy.

[b] "Typical redundancy" refers to the measured redundancy for a source with p_i uniformly distributed over $[0, 1]$.

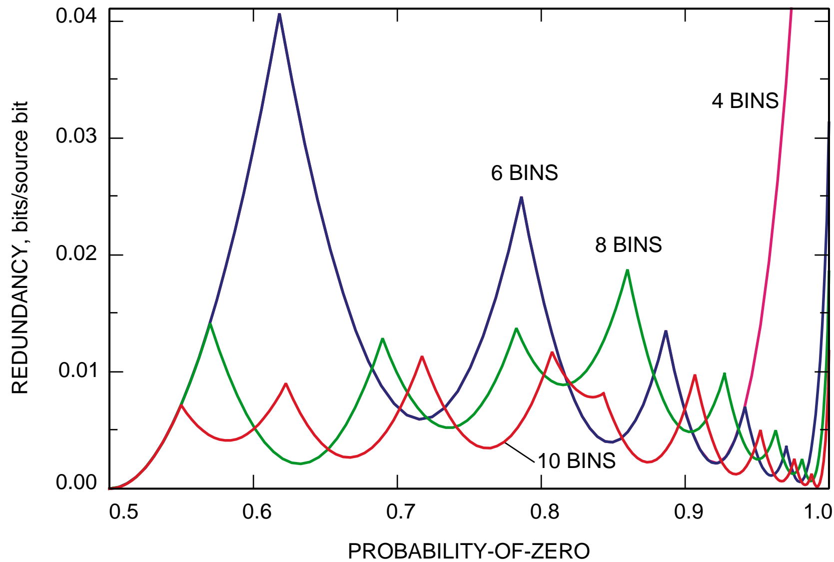

Fig. 22. Redundancy as a function of probability-of-zero for some coder designs from Table 3.

For a given number of bins, we can obtain lower redundancy when we augment our set of candidate trees to include some trees that perform runlength coding. Table 4 indicates some parameters and measures of redundancy for some of the resulting designs obtained when we use this larger set of candidate trees, and Fig. 23 contains a plot of redundancy versus probability-of-zero for some of these coder designs. Table C-2 in Appendix C gives complete specifications.

Table 4. Parameters and redundancies of some coder designs that use trees from $\{\mathcal{T}_{m,m+1}\}_{m=1}^{\infty}$ and certain trees that perform runlength coding.

Bins	Maximum redundancy[a]	Typical redundancy[b]	Number of terminal nodes in the trees
2	1/6	0.051	7
3	0.073	0.025	4,7
4	0.041	0.014	3,6,7
5	0.026	0.0084	5,5,8,7
6	0.018	0.0068	5,5,7,11,7
7	0.014	0.0055	5,5,6,8,4,7
8	0.010	0.0046	7,5,5,4,8,8,7
9	0.0090	0.0041	7,3,5,4,6,3,10,6
10	0.0071	0.0032	7,5,3,5,5,6,7,9,6
11	0.0065	0.0035	9,5,3,4,3,5,11,9,8,7
12	0.0055	0.0027	9,7,5,3,5,7,6,5,4,6,6

[a] "Maximum redundancy" refers to the maximum estimated redundancy among sources having a fixed probability-of-zero, given that the source bits are mapped to the bin that minimizes redundancy.

[b] "Typical redundancy" refers to the measured redundancy for a source with p_i uniformly distributed over $[0,1]$.

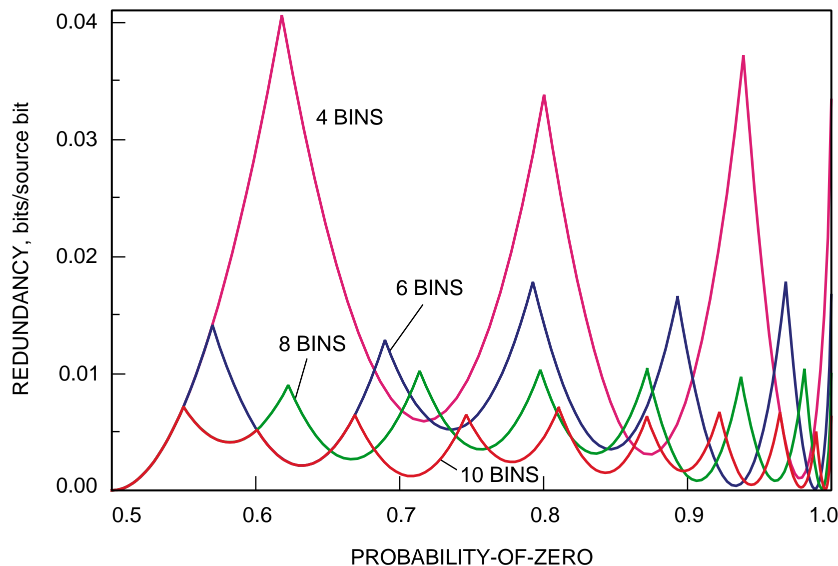

Fig. 23. Redundancy as a function of probability-of-zero for some coder designs from Table 4.

We observe from Tables 3 and 4 that we can achieve low redundancy with coder designs that use a moderate number of bins and relatively small trees. As we'll see in Section VI, implementing coder designs of similar or even somewhat greater complexity is quite practical.

VI. Encoding and Decoding Speed

We now examine the encoding and decoding speeds obtained from coders of various compression efficiencies. We have measured the encoding and decoding speeds of software implementations of our coding technique and of arithmetic coding. We have also measured the redundancy achieved by the coders. All timing tests were performed on a Sun Ultra Enterprise with a 167-MHz UltraSPARC processor.

For our test source sequences, we generated sequences of 500,000 probability values p_i from a uniform distribution on $[0, 1]$ and produced random bits b_i according to these values. We compute bin assignments outside of the timing loop as the optimal assignment given p_i to isolate the speed and efficiency of the actual coding from the source modeling.

For comparison, we also evaluated the "shift/add" binary arithmetic coder from [19] with $b = 16$ and several different values of f (see [19] for definitions of these parameters). The arithmetic coder was modified to be similarly isolated from the modeling; bit probabilities were supplied in a form convenient to the coder. We selected the coder from [19] because it is reasonably fast, it is widely used by other researchers, the source code is publicly available, and it was relatively easy to isolate the coder from the source modeling. Other arithmetic coder implementations (e.g., [18]) may be somewhat faster.

Figure 24 shows redundancy versus decoding speed for two recursive coder designs as well as the arithmetic coder. The recursive coder designs are the 6-bin coder specified in Table C-3 of Appendix C and the 10-bin coder specified in Table C-2 of Appendix C. (The 10-bin coder also appears in Table 4 and Fig. 23.) The 6-bin coder design yields a fast decoder in part because the coder design is recursive only in the last bin; note, however, that our software does not explicitly take advantage of this property. Figure 24 shows that recursive interleaved entropy coding can offer a noticeable improvement in decoding speed over arithmetic coding. For many data compression applications, e.g., image retrieval from databases, decoding speed is much more important than encoding speed, and recursive interleaved entropy coding appears to be an attractive alternative to arithmetic coding in such a situation.

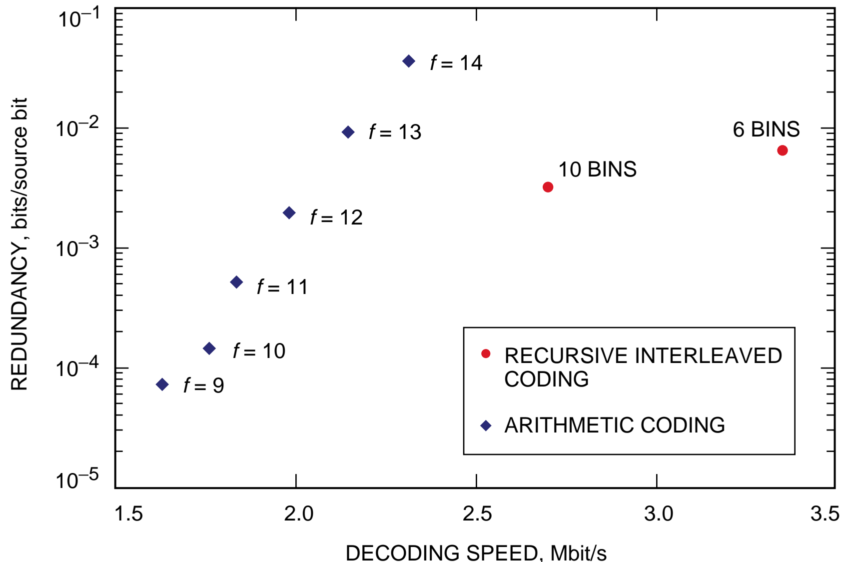

Fig. 24. Decoding performance of two recursive coder designs and the binary arithmetic coder from [19].

For spacecraft applications, however, encoding speed is paramount. Figure 25 shows redundancy versus encoding speed for the two recursive coder designs tested above, for two non-recursive coder designs with a specialized encoder, and for the arithmetic coder. We observe that the recursive coders offer encoding speed comparable to that of arithmetic coding. For a recursive coder design, we expect that encoding is inherently slower than decoding.

For the non-recursive coder designs in Fig. 25, encoding speed is measured for an encoder specifically tailored to exploit non-recursive designs. The non-recursive designs are specified in Table C-3 of Appendix C. We see from the figure that the non-recursive coder designs shown here provide encoding that is more than twice as fast as that of the arithmetic coder. As noted earlier, non-recursive interleaved entropy coders have been investigated in [5–7]; however, previous implementations have used less general component codes than our implementations, and the potential for fast encoding (and decoding) does not appear to have been fully appreciated.

A non-recursive coder design that meets a given redundancy target requires larger trees, and usually a larger number of bins, than a recursive design. For example, achieving low redundancy when the source has probability-of-zero close to one requires the use of very large trees in non-recursive coders, but not in recursive coders. However, at the present stage of development, non-recursive coders appear to have an advantage when encoding speed is our primary concern.

These results give some indication of the performance achievable using recursive and non-recursive interleaved entropy coding. We have good reason to be optimistic that even better coder designs are possible. First, note that our coder design procedure essentially ignores encoding and decoding complexity. Second, recall from Section III.C that the number of useful trees grows very quickly as we increase the size of the tree and from Section I.B that good coder designs may include trees that are not useful. Thus, the number of potential coder designs quickly becomes large as we increase the size of the candidate trees.

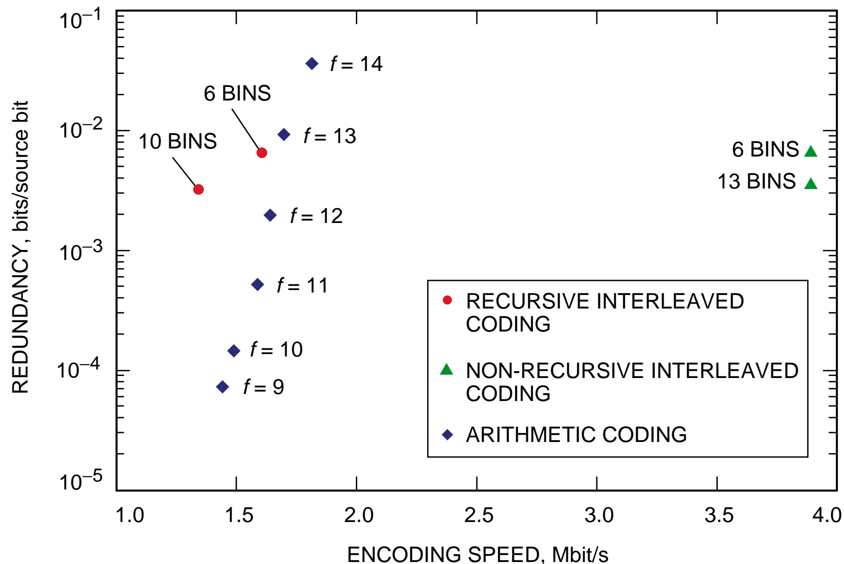

Fig. 25. Encoding performance of two recursive coder designs, two non-recursive coder designs, and the binary arithmetic coder from [19].

VII. Conclusion

We have presented a new entropy coding technique that provides the same functionality as binary arithmetic coding. The technique accommodates an adaptive probability estimate, which allows a data compression algorithm to exploit sophisticated source models, enabling efficient compression. The technique can in theory achieve arbitrarily small asymptotic redundancy as coder designs increase in complexity. We have described a rate estimation technique and a practical coder design procedure. Using the design procedure, we have found relatively simple coder designs that yield efficient compression. Compared to arithmetic coding, our technique provides competitive encoding speed and noticeably better decoding speed. For the special case of a non-recursive coder design, we can achieve significantly faster encoding than with arithmetic coding.

We see that recursive interleaved entropy coding appears to be a viable alternative to arithmetic coding. As recursive interleaved entropy coding is still a very new technique, it is reasonable to expect further improvements, perhaps from both discovering better coder designs and developing algorithmic improvements. By comparison, arithmetic coding has been maturing for over two decades. Our encoding speed tests of non-recursive interleaved entropy coding indicate that the extent of the benefits of that technique has not been previously appreciated.

Several interesting directions for future research remain. Although we have exhibited a practical coder design procedure, it is likely that this procedure could be improved upon, either through refinements or with a fundamentally different procedure. In particular, it may be possible to determine good heuristics for finding fast and efficient coder designs for a given application. A related research task is to better characterize the various aspects of complexity of coder designs. Such a characterization might allow a design procedure to take encoding and decoding speeds into account. In addition, variations in the underlying encoding and decoding procedures may yield speed improvements. Finally, we would like to identify trees (whether useful or not) that are well suited for inclusion in coder designs, and we would like to have a better understanding of useful trees. For example, we would like to identify new (or generalized) families of useful trees and to find a better characterization of useful trees.

References

[1] A. B. Kiely and M. Klimesh, "Memory-Efficient Recursive Interleaved Entropy Coding," *The InterPlanetary Network Progress Report 42-146, April–June 2001*, Jet Propulsion Laboratory, Pasadena, California, pp. 1–14, August 15, 2001. http://ipnpr.jpl.nasa.gov/progress_report/42-146/146J.pdf

[2] M. Boliek, J. D. Allen, E. L. Schwartz, and M. J. Gormish, "Very High Speed Entropy Coding," *Proc. IEEE International Conference on Image Processing (ICIP-94)*, Austin, Texas, pp. 625–629, 1994.

[3] L. Bottou, P. G. Howard, and Y. Bengio, "The Z-Coder Adaptive Binary Coder," *Proc. IEEE Data Compression Conference*, Snowbird, Utah, pp. 13–22, March 1998.

[4] P. G. Howard and J. S. Vitter, "Arithmetic Coding for Data Compression," *Proc. IEEE*, vol. 82, no. 6, pp. 857–865, June 1994.

[5] P. G. Howard, "Interleaving Entropy Codes," *Proc. Compression and Complexity of Sequences 1997*, Salerno, Italy, pp. 45–55, 1998.

[6] F. Ono, S. Kino, M. Yoshida, and T. Kimura, "Bi-Level Image Coding with MELCODE—Comparison of Block Type Code and Arithmetic Type Code," *Proc. IEEE Global Telecommunications Conference (GLOBECOM '89)*, Dallas, Texas, pp. 0255–0260, November 1989.

[7] K. Nguyen-Phi and H. Weinrichter, "A New Binary Source Coder and its Application to Bi-Level Image Compression," *Proc. IEEE Global Telecommunications Conference (GLOBECOM '96)*, London, England, pp. 1483–1487, November 1996.

[8] S. W. Golomb, "Run-Length Encodings," *IEEE Transactions on Information Theory*, vol. IT-12, no. 3, pp. 399–401, July 1966.

[9] R. G. Gallager and D. C. Van Voorhis, "Optimal Source Codes for Geometrically Distributed Integer Alphabets," *IEEE Transactions on Information Theory*, vol. IT-21, no. 2, pp. 228–230, March 1975.

[10] A. B. Kiely, "Bit-Wise Arithmetic Coding for Data Compression," *The Telecommunications and Data Acquisition Progress Report 42-117, January–March 1994*, Jet Propulsion Laboratory, Pasadena, California, pp. 145–160, May 15, 1994. http://tmo.jpl.nasa.gov/tmo/progress_report/42-117/117m.pdf

[11] G. G. Langdon, "An Introduction to Arithmetic Coding," *IBM Journal of Research and Development*, vol. 28, no. 2, pp. 135–149, March 1984.

[12] J. Rissanen and G. G. Langdon, "Arithmetic Coding," *IBM Journal of Research and Development*, vol. 23, no. 2, pp. 149–162, March 1979.

[13] J. Rissanen and G. G. Langdon, "Universal Modeling and Coding," *IEEE Transactions on Information Theory*, vol. IT-27, no. 1, pp. 12–23, January 1981.

[14] I. H. Witten, R. M. Neal, and J. G. Cleary, "Arithmetic Coding for Data Compression," *Communications of the ACM*, vol. 30, no. 6, pp. 520–540, June 1987.

[15] L. Huynh and A. Moffat, "A Probability-Ratio Approach to Approximate Binary Arithmetic Coding," *IEEE Transactions on Information Theory*, vol. 43, no. 5, pp. 1658–1662, September 1997.

[16] G. G. Langdon and J. Rissanen, "Compression of Black-White Images with Arithmetic Coding," *IEEE Transactions on Communications*, vol. COM-29, no. 6, pp. 858–867, June 1981.

[17] J. Rissanen and K. M. Mohiuddin, "A Multiplication-Free Multialphabet Arithmetic Code," *IEEE Transactions on Communications*, vol. 37, no. 2, pp. 93–98, February 1989.

[18] W. B. Pennebaker, J. L. Mitchell, G. G. Langdon, and R. B. Arps, "An Overview of the Basic Principles of the Q-Coder Adaptive Binary Arithmetic Coder," *IBM Journal of Research and Development*, vol. 32, no. 6, pp. 717–726, November 1988.

[19] A. Moffat, R. M. Neal, and I. H. Witten, "Arithmetic Coding Revisited," *ACM Transactions on Information Systems*, vol. 16, no. 3, pp. 256–294, July 1998.

[20] D. Chevion, E. D. Karnin, and E. Walach, "High Efficiency, Multiplication Free Approximation of Arithmetic Coding," *Proc. IEEE Data Compression Conference*, Snowbird, Utah, pp. 43–52, April 1991.

[21] P. G. Howard and J. S. Vitter, "Practical Implementations of Arithmetic Coding," in *Image and Text Compression*, J. A. Storer, ed., Boston: Kluwer Academic, pp. 85–112, 1992.

[22] G. G. Langdon, Jr. and J. Rissanen, “A Simple General Binary Source Code,” *IEEE Transactions on Information Theory*, vol. IT-28, no. 5, pp. 800–803, September 1982. See also correction: vol. IT-29, no. 5, pp. 778–779, September 1983.

[23] G. Feygin, P. G. Gulak, and P. Chow, “Minimizing Excess Code Length and VLSI Complexity in the Multiplication-Free Approximation of Arithmetic Coding,” *Information Processing and Management*, vol. 30, no. 6, pp. 805–816, November 1994.

Appendix A
Useful Regions for a Family of Trees

In this appendix, we prove an extended version of Theorem 1 from Section III. Recall that Theorem 1 identifies, for each $\mathcal{T}_{m,n}$ tree, the region where it is useful.

For each $\mathcal{T}_{m,n}$ tree, there is more than one output-bit labeling that produces a useful tree. For any pair of bit values A and B, let $T_{m,n}^{AB}$ denote the labeling of $\mathcal{T}_{m,n}$ shown in Fig. A-1. We'll see below that only the labelings shown in Fig. A-1 with certain choices of A and B produce useful trees. Note that for tree $T_{m,n}^{00}$ the output bits are identical to the input bits except when the input is the all-zeros or all-ones codeword.

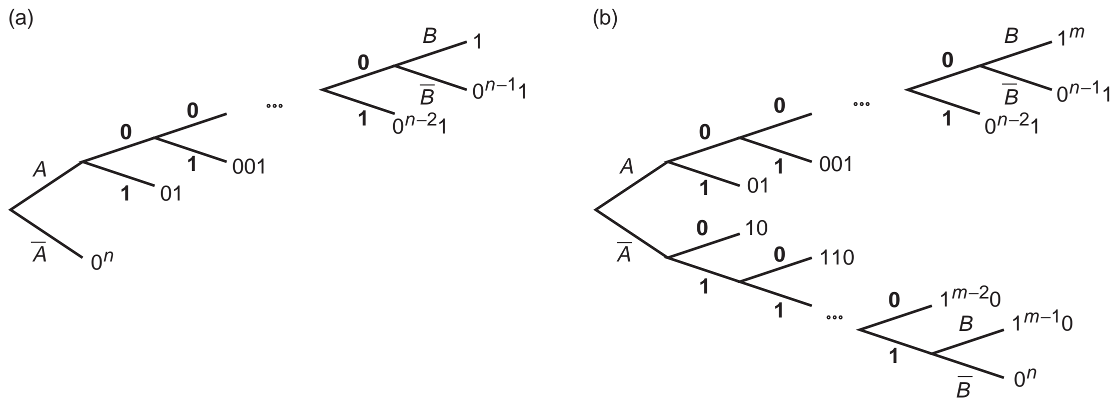

Fig. A-1. Output-bit labelings of $T_{m,n}$ that, for certain choices of bits A and B, produce useful trees for (a) $m = 1$ and (b) $m > 1$. The labeled tree is denoted by $T_{m,n}^{AB}$.

Recall from Section III that for $\alpha \in [0,1)$, γ_α denotes the real root in the interval $(1/2,1]$ of $p = (1-p)^\alpha$. We now state a more complete version of Theorem 1:

Theorem A-1.

(a) *For $n \geq 2$, $\mathcal{T}_{1,n}$ is useful in the interval $(\gamma_{1/n}, 1)$:*

- $T^{00}_{1,n}$ *is useful in the interval* $(\gamma_{1/n}, 2^{-1/n}]$.
- $T^{10}_{1,n}$ *is useful in the interval* $[2^{-1/n}, 1)$.

(b) *For $n \geq 3$, $\mathcal{T}_{2,n}$ is useful in the interval $(\gamma_{2/n}, 1)$:*

- $T^{00}_{2,n}$ *is useful in the interval* $(\gamma_{2/n}, \gamma_{1/(n-1)}]$.
- $T^{01}_{2,n}$ *is useful in the interval* $[\gamma_{1/(n-1)}, \beta_n]$.
- $T^{11}_{2,n}$ *is useful in the interval* $[\beta_n, 1)$.

Here, for $n \geq 3$, β_n is the unique zero of $1/2 - p + p^2 - p^n$ in the interval $(\gamma_{2/n}, 1)$.

(c) *For $n > m \geq 3$, $\mathcal{T}_{m,n}$ is useful in the interval $(\gamma_{m/n}, \gamma_{(m-2)/(n-2)})$:*

- $T^{00}_{m,n}$ *is useful in the interval* $(\gamma_{m/n}, \gamma_{(m-1)/(n-1)}]$.
- $T^{01}_{m,n}$ *is useful in the interval* $[\gamma_{(m-1)/(n-1)}, \gamma_{(m-2)/(n-2)})$.

To prove Theorem A-1, at each non-terminal node of $T^{00}_{m,n}$ we calculate the probability that an output bit generated is zero. This probability is a function of p, the probability-of-zero of the input bits. We can determine the useful ranges of $\mathcal{T}_{m,n}$ and its labeled versions from the output bit probabilities-of-zero; for example, $\mathcal{T}_{m,n}$ is useful when each of these probabilities is in the range $(1-p, p)$, and $T^{00}_{m,n}$ is useful when each is in the range $[1/2, p)$.

For a non-terminal node at depth i in the "upper" section of $T^{00}_{m,n}$, let $\Phi^u_i(p)$ denote the probability that a codeword produces an output bit at the node, and let $q^u_i(p)$ denote the probability that an output bit produced is 0. Let $\Phi^\ell_i(p)$ and $q^\ell_i(p)$ denote similarly defined quantities in the "lower" section of $T^{00}_{m,n}$. (We omit the dependence on m and n in this notation.) Figure A-2 identifies where each of these quantities arises in $T^{00}_{m,n}$.

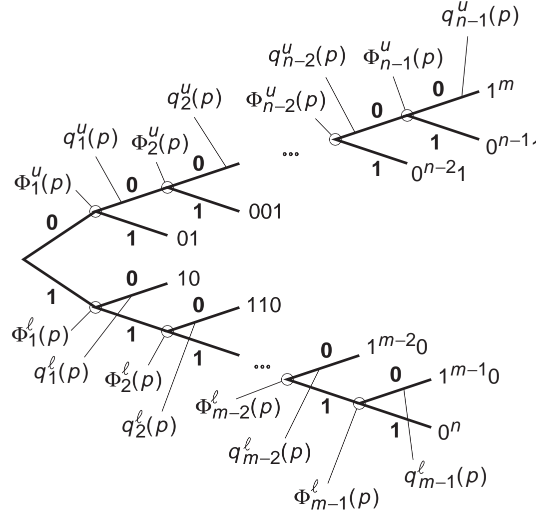

Fig. A-2. Probabilities associated with $\mathcal{T}_{m,n}$, illustrated with the output-bit labeling $T^{00}_{m,n}$; $\Phi^u_i(p)$ and $\Phi^\ell_i(p)$ denote probabilities that a codeword produces an output bit at the indicated node, and $q^u_i(p)$ and $q^\ell_i(p)$ denote probabilities that an output bit produced is zero.

We find that

$$q_i^u(p) = 1 - \frac{(1-p)p^i}{\Phi_i^u(p)}, \qquad i = 1, 2, \cdots, n-1$$

$$q_i^\ell(p) = \frac{p(1-p)^i}{\Phi_i^\ell(p)}, \qquad i = 1, 2, \cdots, m-1$$

$$\Phi_i^u(p) = p^i - p^n + (1-p)^m = p^i - x_{m,n}(p), \qquad i = 1, 2, \cdots, n-1$$

and

$$\Phi_i^\ell(p) = (1-p)^i + p^n - (1-p)^m = (1-p)^i + x_{m,n}(p), \qquad i = 1, 2, \cdots, m-1$$

where

$$x_{m,n}(p) \triangleq p^n - (1-p)^m, \qquad 1 \le m < n$$

The function $x_{m,n}(p)$ is increasing in p for $p \in (0,1)$ and has one zero in the range $(1/2, 1]$. This zero depends only on the ratio m/n and is equal to $\gamma_{m/n}$. Thus, for $p \in [0,1]$, $x_{m,n}(p) > 0$ if and only if $p > \gamma_{m/n}$.

We next present some simple results that will be used to prove Theorem A-1.

Lemma A-1. *For $n > 2$, the polynomial $1/2 - p + p^2 - p^n$ has exactly one zero in $(\gamma_{2/n}, 1)$.*

Proof. Let $y(p) \triangleq 1/2 - p + p^2 - p^n$. It is straightforward to show that $y(\gamma_{2/n}) > 0$ and $y(1) < 0$, so $y(p)$ has at least one zero in $(\gamma_{2/n}, 1)$. To show that $y(p)$ has exactly one zero in this range, it is sufficient to show that

$$y'(p) < 0 \quad \text{whenever} \quad y(p) \le 0 \tag{A-1}$$

Now, $y(p) \le 0$ implies $-p^n \le -1/2 + p - p^2$; thus,

$$\begin{aligned} py'(p) &= -p + 2p^2 - np^n \\ &\le -p + 2p^2 + n\left(-\frac{1}{2} + p - p^2\right) \\ &= -\frac{n}{2} + (n-1)p + (2-n)p^2 \end{aligned}$$

This expression is a quadratic polynomial in p whose discriminant is $2 - (n-1)^2$, which is less than zero since $n > 2$. Thus, $py'(p)$ does not change signs while $y(p) \le 0$, so $y'(p)$ is never equal to 0 when $y(p) \le 0$. But $y'(p) \le 0$ at the point where $y(p)$ first equals 0; thus, Expression (A-1) holds. ❐

Proposition A-1.

(1) $\Phi_1^u(p) > 1 - p$ *for* $p > 1/2$.

(2) $\Phi_1^u(p) < p$ *if and only if* $p > \gamma_{m/n}$.

(3) *For* $m > 1$, $q_{n-1}^u(p) > 1 - p$ *if and only if* $p \in (0, \gamma_{(m-2)/(n-2)})$. *For* $m = 1$, $q_{n-1}^u(p) > 1 - p$ *for* $p \in (0, 1)$.

(4) $q_1^u(p) < p$ *for* $p < 1$.

(5) $q_i^u(p) > q_{i+1}^u(p)$ *for* $i = 1, 2, \cdots, n-2$ *and* $p \in (\gamma_{m/n}, 1)$.

(6) $q_1^u(p) > q_1^\ell(p)$ *for* $p > 1/2$.

(7) $q_i^\ell(p) > q_{i+1}^\ell(p)$ *for* $i = 1, 2, \cdots, m-2$ *and* $p > \gamma_{m/n}$.

(8) $q_{n-1}^u(p) = q_{m-1}^\ell(p)$ *for* $m > 1$.

(9) $q_{n-2}^u(p) \geq 1/2$ *for* $p \leq \gamma_{(m-2)/(n-2)}$.

(10) $q_{m-2}^\ell(p) > q_{n-2}^u(p)$ *for* $p \in (\gamma_{m/n}, \gamma_{(m-2)/(n-2)})$.

(11) *For* $m \geq 1$, $q_{n-1}^u(p) \geq 1/2$ *if and only if* $p \leq \gamma_{(m-1)/(n-1)}$.

(12) *For* $m > 1$, $\Phi_1^u(p) > q_1^\ell(p)$ *when* $p > \gamma_{m/n}$.

Parts (1) and (6) of Proposition A-1 hold because

$$2p - 1 = p - (1-p) > p^n - (1-p)^n > p^n - (1-p)^m = x_{m,n}(p)$$

where the first inequality holds because $p^n - (1-p)^n$ is decreasing in n for $p > 1/2$. The other parts are straightforward, though somewhat tedious, to establish.

Proof of Theorem A-1. To establish the useful region of a tree, we must examine the probabilities-of-zero of the output bits. The probability-of-zero of an output bit generated at the root of $\mathcal{T}_{m,n}^{00}$ is $\Phi_1^u(p)$, and the probabilities-of-zero at the other nodes are $q_1^u(p), q_2^u(p), \cdots, q_{n-1}^u(p)$, and $q_1^\ell(p), q_2^\ell(p), \cdots, q_{m-1}^\ell(p)$.

We first establish the region where $\mathcal{T}_{m,n}$ is useful. For $m > 1$, from parts (1), (2), and (3) of Proposition A-1, we see that the useful range of $\mathcal{T}_{m,n}$ is contained in the interval $(\gamma_{m/n}, \gamma_{(m-2)/(n-2)})$ and that $1 - p < \Phi_1^u(p) < p$ throughout this range. For $m > 1$ and $p \in (\gamma_{m/n}, \gamma_{(m-2)/(n-2)})$, combining parts (4), (5), and (3) of Proposition A-1 gives

$$p > q_1^u(p) > q_2^u(p) > \cdots > q_{n-1}^u(p) > 1 - p$$

and parts (4), (6), (7), (8), and (3) of Proposition A-1 give

$$p > q_1^u(p) > q_1^\ell(p) > q_2^\ell(p) > \cdots > q_{m-1}^\ell(p) = q_{n-1}^u(p) > 1 - p$$

Thus, for $m > 1$, $\mathcal{T}_{m,n}$ is useful if and only if $p \in (\gamma_{m/n}, \gamma_{(m-2)/(n-2)})$. Similar reasoning shows that $\mathcal{T}_{1,n}$ is useful if and only if $p \in (\gamma_{1/n}, 1)$.

Next we turn our attention to the output-bit labeling. For $m > 1$ and $p \in (\gamma_{m/n}, \gamma_{(m-2)/(n-2)})$, combining parts (5) and (9) of Proposition A-1 yields

$$q_1^u(p) > q_2^u(p) > \cdots > q_{n-2}^u(p) > 1/2$$

and parts (7), (10), and (9) of Proposition A-1 give

$$q_1^\ell(p) > q_2^\ell(p) > \cdots > q_{m-2}^\ell(p) > q_{n-2}^u(p) > 1/2$$

Thus, the 0 and 1 output bit labels in Fig. A-1 must be as shown for the labeled tree to be useful, and we must simply determine the values that A and B can take on at each p. The choice of $A = 0$ works whenever $\Phi_1^u(p) \geq 1/2$ and $B = 0$ can be used whenever $q_{n-1}^u(p) \geq 1/2$.

Part (11) of Proposition A-1 implies that when $m = 1$, B needs to be zero, and when $m > 1$, B can be zero if and only if $p \leq \gamma_{(m-1)/(n-1)}$.

For $m = 1$, we find that $\Phi_1^u(p) \geq 1/2$, and hence A can be zero if and only if $p < 2^{-1/n}$. It is easily shown that $2^{-1/n} > \gamma_{m/n}$, so the point $p = 2^{-1/n}$ always occurs in the region where $\mathcal{T}_{1,n}$ is useful. Thus, $T_{1,n}^{00}$ and $T_{1,n}^{10}$ are the useful labelings of $\mathcal{T}_{1,n}$.

For $m > 2$ and $p \in (\gamma_{m/n}, \gamma_{(m-2)/(n-2)})$, combining parts (12), (7), (10), and (9) of Proposition A-1 gives

$$\Phi_1^u(p) > q_1^\ell(p) \geq q_{m-2}^\ell(p) > q_{n-2}^u(p) > 1/2$$

thus, A must be chosen to be zero.

For $m = 2$, we find that $\Phi_1^u(p) \geq 1/2$, and hence A may be zero if and only if $1/2 - p + p^2 - p^n \geq 0$. From Lemma A-1, in the range $(\gamma_{2/n}, 1)$ this function has exactly one zero, β_n. It is straightforward to show that $y(\gamma_{1/(n-1)}) > 0$ and $y(1) < 0$, hence $\beta_n > \gamma_{1/(n-1)}$. This means that trees $T_{2,n}^{00}$, $T_{2,n}^{01}$, and $T_{2,n}^{11}$ are useful but $T_{2,n}^{10}$ is not. ❒

Appendix B
Proof of Theorem 2

In this appendix, we prove Theorem 2 of Section III.B.

Proof. Without loss of generality, we assume each p_i is in the range $[1/2, 1-\delta]$. Choose $\omega > 0$ to be small enough that $1/(-\log_2(1/2+\omega)) < \sqrt{1+\varepsilon}$. The interval $[1/2, 1/2+\omega)$ will correspond to the uncoded bin.

We now specify the trees to use in the remaining region of interest, $[1/2+\omega, 1-\delta]$. The useful region for any tree is an open set and, more specifically, is the union of a finite number of open intervals. Since any open cover of a compact set admits a finite subcover, we can find a finite set $\mathcal{U}'$ of trees from $\mathcal{U}$ such that for any $p \in [1/2+\omega, 1-\delta]$ there exists a tree from $\mathcal{U}'$ that is useful at p. Let m be the size of $\mathcal{U}'$ and let $U_1, \cdots, U_m$ be the members of $\mathcal{U}'$. For any $p \in [1/2+\omega, 1-\delta]$ and each $i \in \{1, \cdots, m\}$, let $f_i(p)$ be the

distance from p to the region in which U_i is not useful. Let $I_i = \{p \in [1/2+\omega, 1-\delta] : f_i(p) = \max_j f_j(p)\}$. It is straightforward to verify that

(1) For each i, I_i is a subset of the useful region of U_i.

(2) Each I_i is the union of a finite number of closed intervals.

(3) $\bigcup_{i=1}^{m} I_i = [1/2 + \omega, 1 - \delta]$.

Since I_i is a subset of the useful region of the tree U_i, it follows that, for any $p \in I_i$, if the input probability to the tree U_i is p, then each of the output bit probabilities will be in the range $(1-p, p)$. In fact, since I_i is a compact set and the output bit probabilities are continuous functions of p, we can find $\Delta_i > 0$ such that, for any input bit probability $p \in I_i$, the output bit probabilities will all be in the range $(1-p+\Delta_i, p-\Delta_i)$. In other words, when used at a given point in I_i, the tree U_i moves probability values closer to 1/2 by at least Δ_i.

Let $\Delta = \min_{1 \le i \le m} \Delta_i$ and let $N = \lfloor 1/\Delta \rfloor$. We will see later that N serves as an upper bound to the recursion depth of the code design. Let $\varepsilon' > 0$ be such that $(1+\varepsilon')^N \le \sqrt{1+\varepsilon}$. Next let $\Delta' > 0$ be small enough that

$$\frac{-\log_2 p}{-\log_2(p + \frac{\Delta'}{2})} < 1 + \varepsilon'$$

and

$$\frac{-\log_2(1 - p - \frac{\Delta'}{2})}{-\log_2(1-p)} < 1 + \varepsilon'$$

whenever both p and $p + \Delta'/2$ are in $[1/2 + \omega, 1 - \delta]$. Let $\Delta'' = \min\{\Delta, \Delta'\}$.

To complete the coder design, we partition $[1/2+\omega, 1-\delta]$ into intervals of width at most Δ'' in such a way that each interval is a subset of some I_i; it follows from the properties of $\{I_j\}_{j=1}^{m}$ that such a partition can be found. The intervals correspond to bins in the coder design. For each bin we pick an i such that the bin's interval is a subset of I_i (as there might be more than one); the bin will use tree U_i. Each bin is assigned a nominal probability; this is taken to be the probability at the center of its interval. Output bit probabilities are computed based on the bin's nominal probability, and output bins are assigned to be the bins in which the output probabilities fall. By our choice of Δ, output bits are assigned to bins that are strictly to the left of the bin that produces the output bits.

At this point, we have completely specified a coder design and we turn our attention to bounding the size of the encoded bitstream. We consider an encoding procedure that, like the encoding procedures described in Section II.B, loads all source bits into encoder memory before processing, but which also keeps track of two additional quantities with each bit. The first quantity is the "nominal encoded length" (in bits) of the bit value. The nominal encoded length for source bit b_i is $-\log_2 p_i$ if $b_i = 0$ or $-\log_2(1-p_i)$ if $b_i = 1$. This value represents, in a sense, the ideal number of output bits that should be produced from encoding the bit; initially the sum of these values is the sum in Expression (1). The second quantity is the "ancestry depth" of the bit, which represents the maximum depth to which the ancestry of a bit can be traced. For source bits, the number is set to 0.

The formation and processing of codewords proceeds normally. Now, however, when a codeword is processed we also calculate values of the nominal encoded length and ancestry depth for each bit produced.

Each such bit has an associated probability-of-zero estimate that is computed based on the nominal input probability of the bin that produces the bit. If we call this probability q, then the nominal encoded length of the output bit is set to $-\log_2 q$ or $-\log_2(1-q)$, depending on whether the bit is 0 or 1. The bits produced are each assigned an ancestry depth equal to one more than the maximum ancestry depth of the bits that formed the codeword.

For example, suppose the tree of Fig. B-1 is used in a bin with nominal probability p. The codewords are labeled with probabilities based on this nominal probability. The non-terminal nodes are labeled with output probabilities; if W represents a random input codeword chosen according to the nominal input bit probabilities, then $q_1 = \Pr(W = 00)$ and $q_2 = \Pr(W = 1 \mid W \in \{1, 01\})$. Suppose a codeword is formed in this bin from two bits with probabilities-of-zero p' and p''. Before the codeword is processed, the affected bits with their nominal encoded lengths might appear as shown in Fig. B-2(a). Figure B-2(c) shows the situation as it might appear after processing. If the bits in Fig. B-2(a) have ancestry depths of, say, 1 and 0, then both bits in Fig. B-2(c) will have ancestry depth 2.

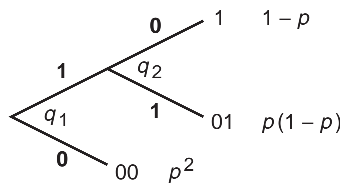

Fig. B-1. An example tree with nominal codeword probabilities shown.

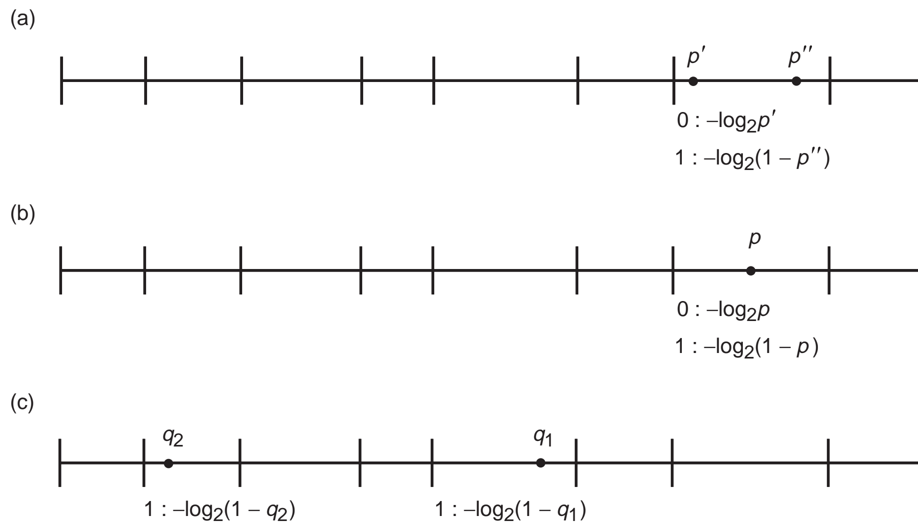

Fig. B-2. Example of processing a codeword, illustrated with bits and their nominal encoded lengths. Probabilities are shown in relation to the intervals corresponding to bins. States shown: (a) initial state, (b) conceptual intermediate state, and (c) final state.

Conceptually, the processing of codewords can be divided into two steps. First, the nominal encoded lengths of the input bits are modified to reflect the bin's nominal input probability. In the second step, the rest of the processing occurs. From our choice of Δ', the first step cannot increase the sum of the nominal encoded lengths of the bits forming the codeword by more than a factor of $1+\varepsilon'$. But a key point is that the second step does not change the total sum of nominal encoded lengths in the encoder: the product of the probabilities of the input bit values is equal to the product of the probabilities of the output bit values, from the fact that the output bit probabilities are assigned based on the nominal input bit probabilities.

Returning to our example, we show in Fig. B-2(b) the encoder state after the nominal encoded length modification. In this state, the sum of the nominal encoded lengths is $-\log_2 p(1-p)$, which directly relates to the fact that $\Pr(W = 01) = p(1-p)$, where again W is the random codeword formed if input bits all have probability-of-zero p. But we also have

$$\Pr(W = 01) = \Pr(W \in \{1, 01\}) \Pr(W = 01 \mid W \in \{1, 01\})$$

$$= (1 - q_1)(1 - q_2)$$

based on the definitions of q_1 and q_2. Thus, $(1-q_1)(1-q_2) = p(1-p)$, so $-\log_2(1-q_1) - \log_2(1-q_2) = -\log_2 p - \log_2(1-p)$. Note that probabilities are only used as a tool to establish this equality; the bits processed are not considered to be random quantities in this analysis.

Suppose that processing a codeword produces a bit with computed probability-of-zero p and ancestry depth k. Then at least one of the bits forming the codeword must have had ancestry depth $k-1$. This bit must have had a computed probability-of-zero of at least $p + \Delta/2$, since the nominal probability of its bin must have been at least $p + \Delta$ by our choice of Δ, and the step of modifying the probability changes it by at most $\Delta/2$. If a bit with ancestry depth $k > N$ is created, then at some point a bit must have existed with probability-of-zero $p + k\Delta/2$, but from our choice of N,

$$p + k\left(\frac{\Delta}{2}\right) > p + \left(\frac{1}{\Delta}\right)\left(\frac{\Delta}{2}\right) = p + \frac{1}{2} \geq 1$$

which is a contradiction; thus, no bit can have ancestry number greater than N.

Associate with each bit in the encoder the quantity $\ell/(1+\varepsilon')^k$, where ℓ is the nominal encoded length of the bit and k is the ancestry depth of the bit. We have shown that, in processing a codeword, the sum of the nominal lengths of the output bits is less than $(1+\varepsilon')$ times the nominal lengths of the bits that formed the codeword. Thus, the sum over all bits of the $\ell/(1+\varepsilon')^k$ quantity decreases each time a codeword is processed. Therefore, the sum of the nominal encoded lengths after all bits have reached the leftmost (uncoded) bin must be less than $(1+\varepsilon')^N$ times the nominal encoded lengths of the original source bits. Our choice of ε' implies that this sum must be less than

$$\sqrt{1+\varepsilon} \sum_{i=1}^{n} \begin{Bmatrix} -\log_2 p_i, & \text{if } b_i = 0 \\ -\log_2(1-p_i), & \text{if } b_i = 1 \end{Bmatrix}$$

Since the bits that reach the leftmost bin form the encoder's output, these bits are each encoded with one bit. By our choice of ω, the nominal encoded length of each bit in this bin is greater than $1/\sqrt{1+\varepsilon}$. Thus, by leaving these bits uncoded, any increase in encoded length is by a factor less than $\sqrt{1+\varepsilon}$, and the final encoded sequence length must be less than $(1+\varepsilon)$ times the original nominal encoded length.

The above analysis has neglected the fact that some bins may be left with incomplete codewords, requiring that flush bits be used. It is easily seen that the extra bits produced by flushing the encoder are bounded by some constant; let c be this constant. The number of bits in the entire output sequence must then be at most the quantity of Expression (1). ❐

Appendix C
Some Coder Designs

In this appendix, we give specifications for several coder designs that use relatively small trees and a small number of bins.

We use a shorthand notation to specify trees in a coder design. A terminal node is identified by the corresponding codeword. Since codeword sets frequently involve long runs of zeros or ones, we use, for example, 0^i to denote a run of i zeros. A non-terminal node is represented by an integer, identifying the index of the associated bin, followed by an ordered pair containing the representations of the child nodes, with the node associated with a zero output bit listed first. To specify a complete coder design, we list the trees for each coded bin. For example, coder design C5 (see Fig. 2) can be written as

$$2 : 1\big(00, 1(1, 01)\big)$$
$$3 : 2\big(00, 1(1, 01)\big)$$
$$4 : 3\left(0^3, 2\big(1(1, 01), 001\big)\right)$$
$$5 : 4\left(0^4, 1\big(1(1, 01), 1(001, 0^3 1)\big)\right)$$

Table C-1 lists coder designs that use only the complete family of useful trees described in Section III.A. Table C-2 lists coder designs that also use runlength trees. Table C-3 lists the coder designs used in Section VI.

Table C-1. Some coder designs that use only the $\mathcal{T}_{m,m+1}$ trees of Section III.A. Maximum redundancy is estimated using the second technique for estimating rate.

Bins	Maximum redundancy	Coder design	Probability range
2	1/2	$2:1(00,1(1,01))$	$(0.6180,1)$
3	1/4	$2:1(00,1(1,01))$;	$(0.6180,0.7862)$
		$3:2(00,1(1,01))$	$(0.7862,1)$
4	1/8	$2:1(00,1(1,01))$;	$(0.6180,0.7862)$
		$3:2(00,1(1,01))$;	$(0.7862,0.8867)$
		$4:3(00,1(1,01))$	$(0.8867,1)$
5	1/16	$2:1(00,1(1,01))$;	$(0.6180,0.7862)$
		$3:2(00,1(1,01))$;	$(0.7862,0.8867)$
		$4:3(00,1(1,01))$;	$(0.8867,0.9416)$
		$5:4(00,1(1,01))$	$(0.9416,1)$
6	0.04058	$2:1(00,1(1,01))$;	$(0.6180,0.7862)$
		$3:2(00,1(1,01))$;	$(0.7862,0.8867)$
		$4:3(00,1(1,01))$;	$(0.8867,0.9416)$
		$5:4(00,1(1,01))$;	$(0.9416,0.9704)$
		$6:5(00,1(1,01))$	$(0.9704,1)$
7	1/36	$2:1(1(0^3,10),1(1(001,11),01))$;	$(0.5698,0.6897)$
		$3:2(2(0^3,10),1(2(001,11),01))$;	$(0.6897,0.8010)$
		$4:3(00,1(1,01))$;	$(0.8010,0.8950)$
		$5:4(00,1(1,01))$;	$(0.8950,0.9460)$
		$6:5(00,1(1,01))$;	$(0.9460,0.9726)$
		$7:6(00,1(1,01))$	$(0.9726,1)$
8	0.01872	$2:1(1(0^3,10),1(1(001,11),01))$;	$(0.5698,0.6897)$
		$3:2(2(0^3,10),1(2(001,11),01))$;	$(0.6897,0.7826)$
		$4:3(3(0^3,10),1(3(001,11),01))$;	$(0.7826,0.8599)$
		$5:4(00,1(1,01))$;	$(0.8599,0.9273)$
		$6:5(00,1(1,01))$;	$(0.9273,0.9630)$
		$7:6(00,1(1,01))$;	$(0.9630,0.9813)$
		$8:7(00,1(1,01))$	$(0.9813,1)$
9	0.01412	$2:1(1(0^3,10),1(1(001,11),01))$;	$(0.5698,0.6897)$
		$3:2(2(0^3,10),1(2(001,11),01))$;	$(0.6897,0.7826)$
		$4:3(3(0^3,10),1(3(001,11),01))$;	$(0.7826,0.8265)$
		$5:3(00,1(1,01))$;	$(0.8265,0.8950)$
		$6:5(00,1(1,01))$;	$(0.8950,0.9460)$
		$7:6(00,1(1,01))$;	$(0.9460,0.9726)$
		$8:7(00,1(1,01))$;	$(0.9726,0.9862)$
		$9:8(00,1(1,01))$	$(0.9862,1)$

Table C-1 (cont'd).

Bins	Maximum redundancy	Coder design	Probability range
10	3/256	$2 : 1(1(1(0^4, 110), 10), 1(1(001, 1(0^3 1, 1^3)), 01));$	$(0.5497, 0.6226)$
		$3 : 1(2(0^3, 10), 1(2(001, 11), 01));$	$(0.6226, 0.7172)$
		$4 : 2(00, 2(1, 01));$	$(0.7172, 0.8075)$
		$5 : 4(4(0^3, 10), 1(4(001, 11), 01));$	$(0.8075, 0.8434)$
		$6 : 4(00, 1(1, 01));$	$(0.8434, 0.9068)$
		$7 : 6(00, 1(1, 01));$	$(0.9068, 0.9523)$
		$8 : 7(00, 1(1, 01));$	$(0.9523, 0.9758)$
		$9 : 8(00, 1(1, 01));$	$(0.9758, 0.9878)$
		$10 : 9(00, 1(1, 01))$	$(0.9878, 1)$
11	0.01046	$2 : 1(1(1(0^4, 110), 10), 1(1(001, 1(0^3 1, 1^3)), 01));$	$(0.5497, 0.6226)$
		$3 : 1(2(0^3, 10), 1(2(001, 11), 01));$	$(0.6226, 0.6994)$
		$4 : 2(3(0^3, 10), 1(3(001, 11), 01));$	$(0.6994, 0.7753)$
		$5 : 3(00, 2(1, 01));$	$(0.7753, 0.8539)$
		$6 : 5(5(0^3, 10), 1(5(001, 11), 01));$	$(0.8539, 0.8769)$
		$7 : 5(00, 1(1, 01));$	$(0.8769, 0.9290)$
		$8 : 7(00, 1(1, 01));$	$(0.9290, 0.9638)$
		$9 : 8(00, 1(1, 01));$	$(0.9638, 0.9818)$
		$10 : 9(00, 1(1, 01));$	$(0.9818, 0.9908)$
		$11 : 10(00, 1(1, 01))$	$(0.9908, 1)$
12	0.007975	$2 : 1(1(1(0^4, 110), 10), 1(1(001, 1(0^3 1, 1^3)), 01));$	$(0.5497, 0.6180)$
		$3 : 2(2(1, 01), 00);$	$(0.6180, 0.6710)$
		$4 : 1(2(1, 01), 00);$	$(0.6710, 0.7441)$
		$5 : 3(4(0^3, 10), 1(4(001, 11), 01));$	$(0.7441, 0.8070)$
		$6 : 4(00, 2(1, 01));$	$(0.8070, 0.8608)$
		$7 : 5(00, 1(1, 01));$	$(0.8608, 0.8983)$
		$8 : 6(00, 1(1, 01));$	$(0.8983, 0.9392)$
		$9 : 8(00, 1(1, 01));$	$(0.9392, 0.9691)$
		$10 : 9(00, 1(1, 01));$	$(0.9691, 0.9844)$
		$11 : 10(00, 1(1, 01));$	$(0.9844, 0.9922)$
		$12 : 11(00, 1(1, 01))$	$(0.9922, 1)$

Table C-2. Some coder designs that use the $\mathcal{T}_{m,m+1}$ trees of Section III.A and runlength trees. Maximum redundancy is estimated using the second technique for estimating rate.

Bins	Maximum redundancy	Coder design	Probability range
2	1/6	$2:1(1(1(1,01),1(1(001,0^31),1(0^41,0^51))),0^6)$	0.07262
3	0.07262	$2:1(1(1(01,001),1),0^3);$	$(0.6573, 0.8960)$
		$3:2(0^6,1(1(1(001,0^31),1(0^41,0^51)),1(1,01)))$	$(0.8960, 1)$
4	0.04058	$2:1(00,1(1,01));$	$(0.61803, 0.8002)$
		$3:1(0^5,1(2(1(0^31,0^41),001),1(1,01)));$	$(0.8002, 0.9388)$
		$4:3(0^6,2(1(1(001,0^31),1(0^41,0^51)),1(1,01)))$	$(0.9388, 1)$
5	0.02551	$2:1(1(0^3,10),1(1(001,11),01));$	$(0.5698, 0.7124)$
		$3:2(2(2(1(001,0^31),01),1),0^4);$	$(0.7124, 0.8497)$
		$4:1(1(2(1(01,001),1),1(1(0^31,0^41),1(0^51,0^61))),0^7);$	$(0.8497, 0.9556)$
		$5:4(0^6,2(1(1(001,0^31),1(0^41,0^51)),1(1,01)))$	$(0.9556, 1)$
6	0.01783	$2:1(1(0^3,10),1(1(001,11),01));$	$(0.5698, 0.6897)$
		$3:2(2(0^3,10),1(2(001,11),01));$	$(0.6897, 0.7926)$
		$4:2(3(3(2(2(1(0^41,0^51),0^31),001),01),1),0^6);$	$(0.7926, 0.8931)$
		$5:1(1(2(1(1(0^61,0^71),1(0^81,0^91)),1(0^41,0^51)),$ $1(1(1,01),1(001,0^31))),0^{10});$	$(0.8931, 0.9683)$
		$6:5(0^6,2(1(1(001,0^31),1(0^41,0^51)),1(1,01)))$	$(0.9683, 1)$
7	0.01412	$2:1(1(0^3,10),1(1(001,11),01));$	$(0.5698, 0.6897)$
		$3:2(2(0^3,10),1(2(001,11),01));$	$(0.6897, 0.7784)$
		$4:2(1(1(1,01),2(1(0^31,0^41),001)),0^5);$	$(0.7784, 0.8671)$
		$5:1(1(2(1(01,001),1),1(1(0^31,0^41),1(0^51,0^61))),0^7);$	$(0.8671, 0.9424)$
		$6:5(0^3,2(1(01,001),1));$	$(0.9424, 0.9850)$
		$7:6(0^6,2(1(1(001,0^31),1(0^41,0^51)),1(1,01)))$	$(0.9850, 1)$
8	0.01039	$2:1(1(1(0^4,110),10),1(1(001,1(0^31,1^3)),01));$	$(0.5497, 0.6226)$
		$3:1(2(0^3,10),1(2(001,11),01));$	$(0.6226, 0.7138)$
		$4:3(3(2(1,01),2(001,0^31)),0^4);$	$(0.7138, 0.7977)$
		$5:2(0^3,2(1(01,001),1));$	$(0.7977, 0.8720)$
		$6:1(1(3(1(01,001),1),2(1(0^31,0^41),1(0^51,0^61))),0^7);$	$(0.8720, 0.9371)$
		$7:4(0^7,1(1(1(0^31,0^41),1(0^51,0^61)),3(1(01,001),1)));$	$(0.9371, 0.9812)$
		$8:7(0^6,3(1(1(001,0^31),1(0^41,0^51)),1(1,01)))$	$(0.9812, 1)$
9	0.008968	$2:1(1(1(0^4,110),10),1(1(001,1(0^31,1^3)),01));$	$(0.5497, 0.6180)$
		$3:2(2(1,01),00);$	$(0.6180, 0.6860)$
		$4:2(3(0^3,10),1(3(001,11),01));$	$(0.6860, 0.7564)$
		$5:1(2(2(01,001),1),0^3);$	$(0.7564, 0.8337)$
		$6:1(1(2(1(0^31,0^41),001),1(1,01)),0^5);$	$(0.8337, 0.9045)$
		$7:6(00,1(1,01));$	$(0.9045, 0.9568)$
		$8:5(0^9,1(2(3(1(0^71,0^81),0^61),1(0^41,0^51)),$ $1(1(1,01),1(001,0^31))));$	$(0.9568, 0.9872)$
		$9:8(0^5,2(3(1(0^31,0^41),001),1(1,01)))$	$(0.9872, 1)$

Table C-2 (cont'd).

Bins	Maximum redundancy	Coder design	Probability range
10	0.007139	$2:1(1(1(0^4,110),10),1(1(001,1(0^31,1^3)),01);$	$(0.5497,0.6008)$
		$3:1(1(1(001,11),01),1(0^3,10));$	$(0.6008,0.6689)$
		$4:1(2(1,01),00);$	$(0.6689,0.7462)$
		$5:3(3(2(1,01),2(001,0^31)),0^4);$	$(0.7462,0.8106)$
		$6:1(2(1(1,01),1(001,0^31)),0^4);$	$(0.8106,0.8718)$
		$7:2(0^5,1(3(1(0^31,0^41),001),1(1,01)));$	$(0.8718,0.9221)$
		$8:4(0^6,3(1(1(001,0^31),1(0^41,0^51)),1(1,01)));$	$(0.9221,0.9643)$
		$9:6(0^8,1(1(1(1,01),1(001,0^31)),1(1(0^41,0^51),1(0^61,0^71))));$	$(0.9643,0.9895)$
		$10:9(0^5,2(3(1(0^31,0^41),001),1(1,01)))$	$(0.9895,1)$
11	0.006484	$2:1(1(1(1(0^31,1(0^41,1^4)),001),01),1(1(1(0^5,1^30),110),10));$	$(0.5386,0.5779)$
		$3:2(2(1(11,001),01),1(10,0^3));$	$(0.5779,0.6423)$
		$4:2(3(1,01),00);$	$(0.6423,0.6999)$
		$5:3(2(3(01,001),1),0^3);$	$(0.6999,0.7549)$
		$6:3(00,2(1,01));$	$(0.7549,0.8102)$
		$7:1(3(2(1,01),2(001,0^31)),0^4);$	$(0.8102,0.8703)$
		$8:4(2(2(1(1,01),1(001,0^31)),3(2(1(0^61,0^71),1(0^81,0^91)),$ $1(0^41,0^51))),0^{10});$	$(0.8703,0.9195)$
		$9:3(0^8,2(1(1(1,01),1(001,0^31)),1(1(0^41,0^51),1(0^61,0^71))));$	$(0.9195,0.9601)$
		$10:7(0^7,2(1(1(0^31,0^41),1(0^51,0^61)),4(1(01,001),1)));$	$(0.9601,0.9884)$
		$11:10(0^6,4(1(1(001,0^31),1(0^41,0^51)),1(1,01)))$	$(0.9884,1)$
12	0.005460	$2:1(1(1(1(0^31,1(0^41,1^4)),001),01),1(1(1(0^5,1^30),110),10));$	$(0.5386,0.5708)$
		$3:1(2(1(001,2(0^31,1^3)),01),1(2(0^4,110),10));$	$(0.5708,0.6249)$
		$4:1(2(0^3,10),1(2(001,11),01));$	$(0.6249,0.6763)$
		$5:1(3(1,01),00);$	$(0.6763,0.7417)$
		$6:4(4(2(1,01),2(001,0^31)),0^4);$	$(0.7417,0.7875)$
		$7:5(1(3(2(001,0^31),2(0^41,0^51)),2(1,01)),0^6);$	$(0.7875,0.8420)$
		$8:1(1(3(2(0^31,0^41),001),2(1,01)),0^5);$	$(0.8420,0.8957)$
		$9:5(0^4,1(1(1,01),1(001,0^31)));$	$(0.8957,0.9374)$
		$10:8(0^3,4(1(01,001),1));$	$(0.9374,0.9711)$
		$11:9(0^5,3(4(1(0^31,0^41),001),1(1,01)));$	$(0.9711,0.9915)$
		$12:11(0^5,3(4(1(0^31,0^41),001),1(1,01)))$	$(0.9915,1)$

Table C-3. Some coder designs used in Section VI.

Bins	Maximum redundancy	Coder design	Probability range
6-bin recursive	1/36	$2:1(1(0^3,01),1(1(001,11),10));$	$(0.56984,0.694507)$
		$3:1(1(1(010,1(0^41,011)),1(001,0^31)),1(1,0^5));$	$(0.694507,0.797317)$
		$4:1(1(1(10,1(1(11,0^311),0^310)),1(01,\ 001)),0^4);$	$(0.797317,0.886762)$
		$5:1(1(1(1(01,001),1(1,1(0^71,0^81))),1(1(0^31,\ 0^41),$ $1(0^51,0^61))),0^9);$	$(0.886762,0.959425)$
		$6:5(0^4,1(1(1,01),1(001,\ 0^31)))$	$(0.959425,1)$
6-bin non-recursive	1/30	$2:1(1(0^3,01),1(1(001,11),10));$	$(0.56984,0.694507)$
		$3:1(1(1(010,1(0^41,011)),1(001,0^31)),1(1,0^5));$	$(0.694507,0.797317)$
		$4:1(1(1(10,1(1(11,0^311),0^310)),1(01,001)),0^4);$	$(0.797317,0.886762)$
		$5:1(1(1(1(01,001),1(1,1(0^71,0^81))),1(1(0^31,0^41),$ $1(0^51,0^61))),0^9);$	$(0.886762,0.957002)$
		$6:1(1(1(1(1(1,01),1(1(001,0^31),1(0^41,0^51))),$ $1(1(1(0^61,0^71),1(0^81,0^91)),1(1(0^{10}1,0^{11}1),$ $1(0^{12}1,0^{13}1)))),1(1(1(1(0^{14}1,0^{15}1),$ $1(0^{16}1,0^{17}1)),1(1(0^{18}1,0^{19}1),1(0^{20}1,0^{21}1))),$ $1(1(1(0^{22}1,0^{23}1),1(0^{24}1,0^{25}1)),$ $1(1(0^{26}1,0^{27}1),1(0^{28}1,0^{29}1))))),0^{30})$	$(0.957002,1)$
13-bin non-recursive	1/31	$2:1(1(1(1(1(0^41,1^4),0^31),001),10),$ $1(01,1(110,1(0^5,1^30))));$	$(0.538597,0.56984)$
		$3:1(1(1(001,1(1(1101,0^311),1^3)),10),$ $1(01,1(0^4,1(1100,0^310))));$	$(0.56984,0.618034)$
		$4:1(1(0^3,01),1(10,1(001,11)));$	$(0.618034,0.66515)$
		$5:1(1(1(010,1(10^4,110)),1(1(101,011),$ $1(1(10^31,1^3),1001))),00);$	$(0.66515,0.724492)$
		$6:1(1(0^5,1),1(1(0^31,001),1(010,1(0^41,011))));$	$(0.724492,0.764976)$
		$7:1(0^3,1(1(001,010),1(100,1(11,1(011,101)))));$	$(0.764976,0.818556)$
		$8:1(0^4,1(1(001,01),1(10,1(0^310,1(0^311,11)))));$	$(0.818556,0.852992)$
		$9:1(0^5,1(1(001,01),1(1,1(0^41,0^31))));$	$(0.852992,0.881271)$
		$10:1(0^6,1(1(01,1),1(1(0^51,0^41),1(0^31,001))));$	$(0.881271,0.908155)$
		$11:1(0^9,1(1(1(1,01),1(001,0^31)),$ $1(1(0^41,0^51),1(0^61,1(0^71,0^81)))));$	$(0.908155,0.942647)$
		$12:1(0^{15},1(1(1(1,1(01,001)),1(1(0^31,0^41),$ $1(0^51,0^61))),1(1(1(0^71,0^81),1(0^91,0^{10}1)),$ $1(1(0^{11}1,0^{12}1),1(0^{13}1,0^{14}1)))));$	$(0.942647,0.968473)$
		$13:1(0^{31},1(1(1(1(1,1(01,001)),1(1(0^31,0^41),$ $1(0^51,0^61))),1(1(1(0^71,0^81),1(0^91,0^{10}1)),$ $1(1(0^{11}1,0^{12}1),1(0^{13}1,0^{14}1)))),1(1(1(1(0^{15}1,$ $0^{16}1),1(0^{17}1,0^{18}1)),1(1(0^{19}1,0^{20}1),1(0^{21}1,$ $0^{22}1))),1(1(1(0^{23}1,0^{24}1),1(0^{25}1,0^{26}1)),$ $1(1(0^{27}1,0^{28}1),1(0^{29}1,0^{30}1))))))$	$(0.968473,1)$