Weight Distributions for Turbo Codes Using Random and Nonrandom Permutations

S. Dolinar and D. Divsalar
Communications Systems and Research Section

This article takes a preliminary look at the weight distributions achievable for turbo codes using random, nonrandom, and semirandom permutations. Due to the recursiveness of the encoders, it is important to distinguish between self-terminating and non-self-terminating input sequences. The non-self-terminating sequences have little effect on decoder performance, because they accumulate high encoded weight until they are artificially terminated at the end of the block. From probabilistic arguments based on selecting the permutations randomly, it is concluded that the self-terminating weight-2 data sequences are the most important consideration in the design of the constituent codes; higher-weight self-terminating sequences have successively decreasing importance. Also, increasing the number of codes and, correspondingly, the number of permutations makes it more and more likely that the bad input sequences will be broken up by one or more of the permuters.

It is possible to design nonrandom permutations that ensure that the minimum distance due to weight-2 input sequences grows roughly as $\sqrt{2N}$, where N is the block length. However, these nonrandom permutations amplify the bad effects of higher-weight inputs, and as a result they are inferior in performance to randomly selected permutations. But there are "semirandom" permutations that perform nearly as well as the designed nonrandom permutations with respect to weight-2 input sequences and are not as susceptible to being foiled by higher-weight inputs.

I. Introduction

Turbo codes are constructed by applying two or more simple-to-decode codes to differently permuted versions of the same information sequence. The corresponding turbo decoding algorithm iterates the outputs of simple decoders acting on each component code. Recent work in this area by researchers all over the world (e.g., [1–5]) has shown that low bit-error rates can be achieved by turbo decoders at astonishingly low bit signal-to-noise ratios (SNRs) if the information block is large and the permutations are selected randomly. In this article, we also consider turbo code structures based on nonrandom and semirandom permutations.

Good turbo codes have been constructed using short constraint length, infinite impulse response (IIR) convolutional codes as components instead of the more familiar finite impulse response (FIR) convolutional codes. These IIR convolutional codes are also referred to as recursive convolutional codes, because previously encoded information bits are continually fed back to the encoder's input. For both IIR and

FIR encoders, a single isolated information-bit error will produce the same convolutionally encoded sequence (i.e., the encoder's "impulse response") no matter where it is permuted within the information sequence. For an IIR encoder, this impulse response has infinite weight (for a never-ending information stream), while for an FIR encoder, the weight of its response to a single isolated bit error cannot be much larger than the code's free distance. Thus, the IIR property is important for building turbo codes, because it avoids low-weight encodings that are impervious to the action of the permuters.

In contrast to single bit-error inputs, two or more bit errors in an information sequence can be permuted into different bit patterns whose encoded output bears no resemblance to the encoding of the unpermuted information. The trick in turbo coding is to match low-weight encodings of one permutation with high-weight encodings of the other(s), thus producing total weights significantly higher than the low weights that are possible from each of the simple component codes individually. In this article, we take a preliminary look at the weight distributions achievable for turbo codes using random, nonrandom, and semirandom permutations. Some of this material was included in [5] but is repeated here for a coherent presentation.

II. Turbo Code Structure

Figure 1 shows a particular example of a turbo code using three component codes that will be used throughout this article to illustrate some fundamental concepts. This is the same example used in [5], which derives the iterative algorithm for decoding such a code and evaluates the resulting performance of the iterative decoder. The figure shows three simple recursive convolutional encoders with constraint length $K = 3$ (i.e., memory $M = 2$). The overall code is a rate 1/4 code with four output streams. One of the output streams is the information sequence (uncoded). The other three output streams in this example are parity sequences corresponding to a ratio of generator polynomials g_b/g_a, where $g_a(D) = 1 + D + D^2$ and $g_b(D) = 1 + D^2$. These three parity streams would be identical if no permutations π_1, π_2 were used.

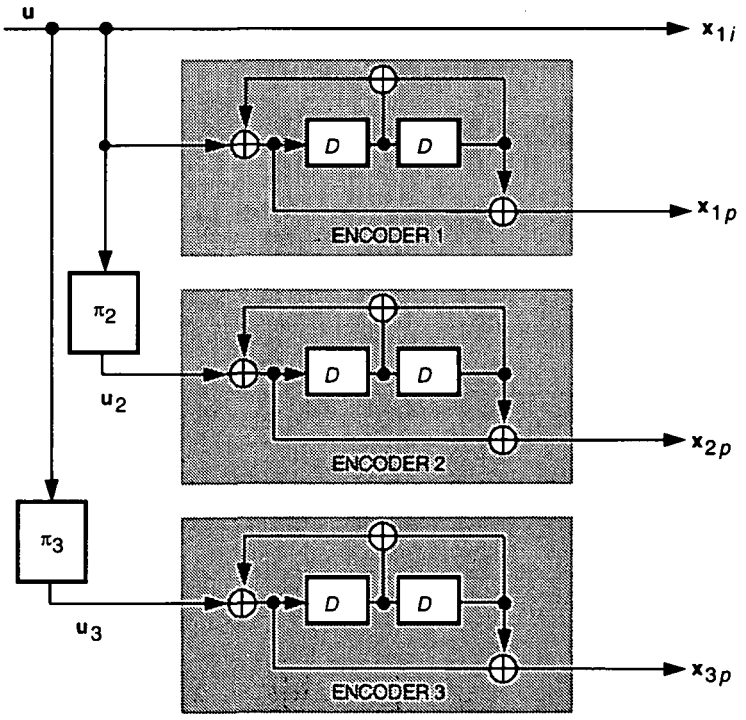

Fig. 1. Example of turbo encoder with three codes.

The encoder in Fig. 1 is used to generate an $(n(N+M), N)$ block code, where N is the information block size. Following the information bits, an additional M tail bits are appended in order to drive the encoder to the all-zero state at the end of the block. Due to the encoders' recursiveness, the required M tail bits cannot be predetermined, but they can be automatically computed at the encoders using a trick suggested in [4]. This action of returning the encoders to the all-zero state is called trellis termination.

Some input data sequences of length N are *self-terminating*, because the encoders are already in the all-zero state after encoding N information bits before any tail bits are appended. All M tail bits are zero for a self-terminating input sequence. Non-self-terminating input sequences require one or more nonzero tail bits for proper trellis termination.

III. Weight Distributions for Turbo Codes

The weight distribution for the codewords produced by the turbo decoder depends on how the codewords from one of the simple component encoders are teamed with codewords from the other encoder(s). In the example of Fig. 1, the component codes have minimum distances 5, 2, and 2. If the codes were not recursive (but used the same generator polynomials g_a, g_b), the minimum weight word for all three encoders would be generated by the weight-1 input sequence $(00\cdots0000100\cdots00)$ with a single "1." This will produce a minimum distance of 9 for the overall turbo code because the weight-1 input sequence will always appear again in the other encoders regardless of the choice of permutations. This motivates the use of recursive encoders, where the key ingredient is the recursiveness and not the fact that the encoders are systematic. For the recursive encoder in Fig. 1, the weight-1 input sequence generates the code's infinite impulse response, and the output encoded weight is kept finite only due to the trellis termination at the end of the block. In the recursive case, the minimum weight word for all three encoders is generated by the weight-3 input sequence $(00\cdots000011100\cdots00)$ with three consecutive 1's. However, after random permutations, a pattern of three consecutive 1's is not likely to appear again at the input to the second and third encoders, so it is unlikely that all three encoders will simultaneously emit minimum-weight words. In this case, the minimum distance will be higher than 9 for the turbo code structure in Fig. 1.

Recursive encoders do not modify the output weight distributions of the individual component codes. They only change the mapping between the input data sequences and the output encoded sequences. As the previous example illustrates, having the same output sequence mapped from a weight-3 input sequence instead of a weight-1 input sequence gives the permuters a chance to break up the offending input before feeding it to the second and third encoders. An encoder whose low output weights come from high input weights is generally undesirable for conventional use where the goal is minimizing the decoded bit-error rate. However, it is precisely this characteristic that makes an encoder very desirable for use as a component of a turbo code structure! The next section gives a more detailed examination of the link between the weight of the input sequence and the ability of the permuters to disperse it into a sequence with higher output weight.

Now let us consider another input sequence that produces fairly low weights at the outputs of each of the encoders in Fig. 1, the weight-2 sequence $(00\cdots00100100\cdots00)$. This sequence is self-terminating, i.e., it forces the encoder back to the all-zero state without any help from the trellis termination scheme applied at the end of the block. The four output streams that are encoded from this input are of the form $(00\cdots00100100\cdots00)$ and $(00\cdots00111100\cdots00)$, with the latter sequence repeated three times. In this case, the nonzero portion of the output has a duration of four bit times, the same as the nonzero portion of the input. If the permuters do not "break" this sequence before the input to the second and third encoders, the resulting total encoded weight will be 14.

For comparison, let us examine the weight-2 sequence $(00\cdots0010100\cdots00)$. This sequence is not self-terminating because the encoder never returns to the all-zero state until it is forced to do so at the end of the block. Now the parity sequence (repeated three times) is of the form $(00\cdots001111\cdots)$, and the

sequence of all 1's continues until the trellis is terminated. This sequence accumulates a very large weight unless it starts near the end of the block. If the permuters could map all self-terminating sequences into non-self-terminating sequences, the turbo code could have a minimum weight that grows linearly with block size N.

For a nonrecursive encoder, nearly all low-weight input sequences are self-terminating. The only non-self-terminating sequences are those whose last 1 occurs near the end of the block. As a result, the output weight is very strongly correlated with the input weight for all possible input sequences. For instance, there are no weight-2 sequences that produce a very large output weight, in contradistinction to the previous example for the recursive encoders in Fig. 1.

The weight-2 input sequence $(00\cdots00100100\cdots00)$ is not the only self-terminating weight-2 sequence for the recursive encoders in Fig. 1. In general, weight-2 sequences with their 1's separated by $d = 3\delta$ bit positions, where $\delta = 1, 2, \cdots$, are also self-terminating unless the last 1 occurs near the end of the block. However, the weight of the encoded output increases linearly with the separation. The total encoded weight of such a sequence would be $14 + 6\delta$ if there were no permutations. With permutations before the second and third encoders, a weight-2 sequence with its 1's separated by $d_1 = 3\delta_1$ bit positions will be permuted into two other weight-2 sequences with 1's separated by $d_i = 3\delta_i$ bit positions, $i = 2, 3$, where each δ_i is defined as a multiple of $1/3$. If any δ_i is not an integer, the corresponding encoded output will have a high weight because then the convolutional code output is non-self-terminating. If all δ_i's are integers, the total encoded weight will be $14 + 2\sum_{i=1}^{3}\delta_i$. This weight grows linearly with the separation between the 1's in the input sequences (after permutations). When the 1's are far apart, the encoded sequence looks like the code's infinite impulse response up to the point where the second strategically placed 1 terminates the further accumulation of weight.

The coefficient "2" in the expression for the total encoded weight of the self-terminating weight-2 sequences is a characteristic of the particular codes chosen in Fig. 1. It measures the rate of growth of the weight of each encoder's output as a function of the separation between the two 1's in the input sequence. This coefficient cannot be made any larger than 2 using constraint-length-3 codes as in Fig. 1. It can, however, be made smaller, and the result is an inferior turbo code. To illustrate this, we consider the same encoder structure in Fig. 1, except with the roles of g_a and g_b reversed. Now the minimum distances of the three component codes are 5, 3, and 3, producing an overall minimum distance of 11 for the total code without any permutations. This is apparently a better code, but it turns out to be inferior as a turbo code. This paradox is explained by again considering the critical weight-2 data sequences. For this code, weight-2 sequences with $d_1 = 2\delta_1$ bit positions separating the two 1's produce self-terminating output and, hence, low-weight encoded words if $\delta_1 = 1, 2, \cdots$. In the turbo encoder, such sequences will be permuted to have separations $d_i = 2\delta_i, i = 2, 3$, for the second and third encoders, where now each δ_i is defined as a multiple of $1/2$. But now the total encoded weight for integer triplets $(\delta_1, \delta_2, \delta_3)$ is $11 + \sum_{i=1}^{3}\delta_i$. Notice how this weight grows only half as fast with $\sum_{i=1}^{3}\delta_i$ (three-fourths as fast with $\sum_{i=1}^{3}d_i$) as the previously calculated weight for the original code. Clearly, it is important to choose component codes that cause the overall weight to grow as fast as possible with the individual separations d_i or δ_i. This consideration outweighs the criterion of selecting component codes that would produce the highest minimum distance if unpermuted.

The summation $\sum_{i=1}^{3}\delta_i$ in the expression for the total encoded weight of the self-terminating weight-2 sequences depends mostly on the choice of permutations and less on the choice of code. For a given set of permutations, this summation would be the same for any code whose self-terminating weight-2 sequences have separations $d = 3\delta$ between the 1's. This approach provides a method to partially separate the problem of picking good permutations from the problem of picking good component codes. If the weight-2 sequences are the only ones that matter, the permutations should be designed to avoid integer triplets $(\delta_1, \delta_2, \delta_3)$ that are simultaneously small in all three components. In fact, it would be nice to design permutations to guarantee that the smallest value of $\sum_{i=1}^{3}\delta_i$ (for integer δ_i) grows with the block size N.

Alas, things are never so simple. There are also many weight-$n, n = 3, 4, 5, \cdots$, data sequences that produce self-terminating output and, hence, low encoded weight. The criterion for optimally choosing the permutations also depends on similar expressions for the separations between the 1's of the higher-weight sequences. Achieving a global optimum, by considering input sequences of all weights simultaneously, is still an unsolved problem.

As argued in the next section, higher-weight input sequences are much more likely than weight-2 input sequences to be broken up by randomly chosen permuters. If the block size is large, they are likely to produce non-self-terminating output from at least one of the encoders. But a purely random permuter is also likely to reproduce a few of the weight-2 input sequences with the lowest output weights. It is easy to design nonrandom permutations that combat the weight-2 sequences more effectively than random permutations, but such designs tend to unnaturally amplify the effects of higher-weight sequences. Section IV explores these issues in more detail.

IV. Choosing the Permutations

The performance of a turbo code depends on how effectively the data sequences that produce low encoded weights at the output of one encoder are matched with permutations of the same data sequence that yield higher encoded weights at the outputs of the others. Random permutations do a very good job of teaming low weights with high weights for the vast majority of possible information sequences. It has been demonstrated empirically that the performance of these codes is astonishingly good as compared to everyday convolutional codes that do not use any permuters. In this section, we try to analyze why random permutations work so well and whether nonrandom or semirandom permutations can be designed to outperform them.

If randomly chosen permutations perform well, then in principle it is possible to design deterministic permutations that work even better. Unfortunately, this has proved to be an elusive goal. To date, most deterministic permutation designs have produced turbo codes that do not match the decoded error rates of corresponding codes using random permutations. In this section, we describe some suggested nonrandom permutations, and we try to develop design criteria that mimic or improve on the good features of random permutations.

A. Random Permutations

Now we briefly examine the issue of whether one or more random permutations can avoid matching small separations between the 1's of a weight-2 data sequence with equally small separations between the 1's of its permuted version(s). Consider, for example, a particular self-terminating weight-2 data sequence $(00\cdots001001000\cdots00)$ that corresponds to a low-weight codeword in each of the encoders of Fig. 1. If we select one permutation at random, the probability that this sequence will be permuted into another sequence with the same distance 3 between its two 1's is roughly $2/N$ (assuming that N is large and ignoring minor edge effects). The probability that such an unfortunate pairing happens for at least one possible position of the original sequence $(00\cdots001001000\cdots00)$ within the block of size N is approximately $1-(1-2/N)^N \approx 1-e^{-2}$. This implies that the minimum distance of a two-code turbo code constructed with a random permutation is not likely to be much higher than the encoded weight of such an unpermuted weight-2 data sequence, e.g., 14 for the code in Fig. 1. By contrast, if we use three codes and two different random permutations, the probability that a particular sequence $(00\cdots001001000\cdots00)$ will be reproduced by both permuters is only $(2/N)^2$. Now the probability of finding such an unfortunate data sequence somewhere within the block of size N is roughly $1-\left[1-(2/N)^2\right]^N \approx 4/N$. Thus, it is not probable that the minimum distance of a three-code turbo code using two random permutations will be limited roughly by the lowest encoded weight of an unpermuted weight-2 data sequence. This argument can be extended to account for other weight-2 data sequences that may also produce low-weight codewords, e.g., $(00\cdots00100(000)^\delta 1000\cdots00)$, for the code in Fig. 1.

For comparison, let us consider a weight-3 data sequence such as $(00\cdots0011100\cdots00)$ that, for our example, corresponds to the minimum distance of the code (using no permutations). The probability that this sequence is reproduced with one random permutation is roughly $6/N^2$, and the probability that some sequence of the form $(00\cdots0011100\cdots00)$ is paired with another of the same form is $1-(1-6/N^2)^N \approx 6/N$. Thus, for large block sizes, the bad weight-3 data sequences have a small probability of being matched with bad weight-3 permuted data sequences, even in a two-code system. For a turbo code using 3 codes and 2 random permutations, this probability is even smaller, $1-\left[1-(6/N^2)^2\right]^N \approx (36/N^3)$. This implies that the minimum-distance codeword of the turbo code in Fig. 1 is more likely to result from a weight-2 data sequence of the form $(00\cdots001001000\cdots00)$ than from the weight-3 sequence $(00\cdots0011100\cdots00)$ that produces the minimum distance in the unpermuted version of the same code. Higher-weight sequences have an even smaller probability of reproducing themselves after being passed through the random permuters. These probabilistic arguments do not guarantee that all possible choices of permutations will increase the unpermuted code's minimum distance. For the worst-case permutations, the minimum distance of the turbo code is still 9, but these permutations are highly unlikely if chosen randomly.

For a turbo code using q codes and $q-1$ permutations, the probability that a weight-n data sequence will be reproduced somewhere within the block by all $q-1$ permutations is of the form $1-\left[1-(\beta/N^{n-1})^{q-1}\right]^N$, where β is a number that depends on the weight-n data sequence but does not increase with block size N. For large N, this probability is proportional to $(1/N)^{nq-n-q}$, which falls off rapidly with N, when n and q are greater than 2. Furthermore, the symmetry of this expression indicates that increasing either the weight of the data sequence n or the number of codes q has roughly the same effect on lowering this probability.

B. Designed Permutations

In the discussion of random permutations, we identified self-terminating weight-2 data sequences as the most likely to contribute low-weight codewords that might not be successfully permuted into sequences that contribute high weights for the other encoders. Therefore, the first design criterion for a good permutation ought to be how well it breaks up the self-terminating weight-2 input sequences. We have seen how the encoded weights of self-terminating weight-2 input sequences increase linearly with the distance between their two 1's, with a slope that depends on the characteristics of the constituent codes. The first job of the permuter(s) then is to produce a good weight distribution for self-terminating weight-2 inputs. For the example in Figure 1, this is accomplished by making sure that the sum $\sum_{i=1}^{3}\delta_i$ is never (or seldom) very small for integer values of δ_i.

1. Nonrandom Permutations Based on Block Interleavers. One method for the design of nonrandom permutations is based on block interleavers. Interleavers should be capable of spreading low-weight input sequences so that the resulting codeword has high weight. Block interleavers, defined by a matrix with ν_r rows and ν_c columns such that $N=\nu_r\times\nu_c$, may fail to spread certain sequences. For example, the weight-4 sequence shown in Fig. 2 cannot be broken by a block interleaver. Block interleavers are effective if the low-weight sequence is confined to one row. If low-weight sequences (which can be regarded as the combination of lower-weight sequences) are confined to several consecutive rows, then the ν_c columns of the interleaver should be sent in a specified order to spread as much as possible the low-weight sequences. A method for reordering the columns is given in [6]. This method guarantees that for any number of columns $\nu_c=ab+r$ $(r\le a-1)$, the minimum separation between data entries is $b-1$, where a is the number of columns affected by a burst. However, as can be observed in the example in Fig. 2, the sequence $(00\cdots00100100\cdots00)$ will still appear at the input of the encoders for any possible column permutation. Only if we permute the rows of the interleaver in addition to its columns is it possible to break these weight-4 sequences. The method in [6] can be used again for the permutation of rows. Appropriate selection of a and b for rows and columns depends on the particular set of codes used and on the specific low-weight sequences that we would like to break.

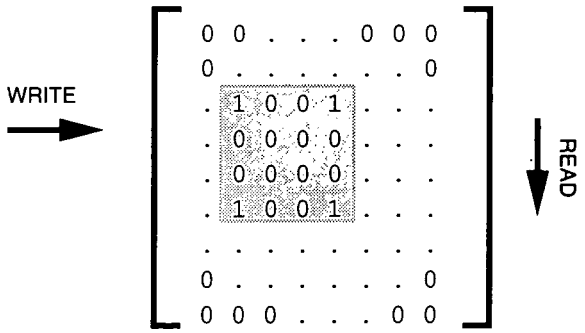

Fig. 2. Example where a block interleaver fails to "break" the input sequence.

2. Nonrandom Permutations Based on Circular Shifting. Another class of nonrandom permutations is derived from circular shifting. The basic formula is

$$\pi(j) = (aj + r) \bmod N \tag{1}$$

where $r < N$ is an offset and $a < N$ is a step size that is relatively prime to N. The choice of offset is unimportant if edge effects can be ignored. Here we take $r = 0$ and concentrate on selecting the step size a. For a concrete example, we take $N = 32$ and $a = 7$. This generates the following permutation:

j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
$\pi(j)$	0	7	14	21	28	3	10	17	24	31	6	13	20	27	2	9
	16	23	30	5	12	19	26	1	8	15	22	29	4	11	18	25

Note that for consecutive values of j, the corresponding values of $\pi(j)$ are always separated by either 7 or 25. In other words, if the distance between a pair of 1's in an unpermuted weight-2 input sequence is $d_1 = 1$, then the corresponding distance after permutation by π is either $d_2 = 7$ or $d_2 = 25$. Similarly, if $d_1 = 2$, then either $d_2 = 14$ or $d_2 = 18$. Continuing in this way, we can verify that $d_1 + d_2 \geq 8$ for any possible combination of d_1 and d_2.

If one tries to improve on this by using a larger step size, things do not always get better. For example, consider the permutation generated by taking $a = 11$:

j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
$\pi(j)$	0	11	22	1	12	23	2	13	24	3	14	25	4	15	26	5
	16	27	6	17	28	7	18	29	8	19	30	9	20	31	10	21

In this case, $d_1 + d_2 \geq 12$ whenever $d_1 = 1$ or $d_1 = 2$, but $d_1 + d_2 = 4$ when $d_1 = 3$.

This example generalizes in the following manner when the block size N is half of a perfect square. Take the step size $a = \sqrt{2N} - 1$. Then $d_1 + d_2 \geq \sqrt{2N}$ for all possible combinations of d_1 and d_2. The

step size a that achieves this inequality is not unique. For example, $a = \ell\sqrt{2N} \pm 1$ also gives the same lower bound on $d_1 + d_2$ for positive integers $\ell < \sqrt{N/2}$ that are relatively prime to $\sqrt{N/2}$. When $\sqrt{2N}$ is not an integer, it can be demonstrated empirically that the same bound holds approximately, i.e., for a properly optimized step size, $d_1 + d_2 \geq \nu$, where ν is only slightly smaller than $\sqrt{2N}$. Figure 3 shows the maximum over step sizes of the minimum value of $d_1 + d_2$ as a function of N.

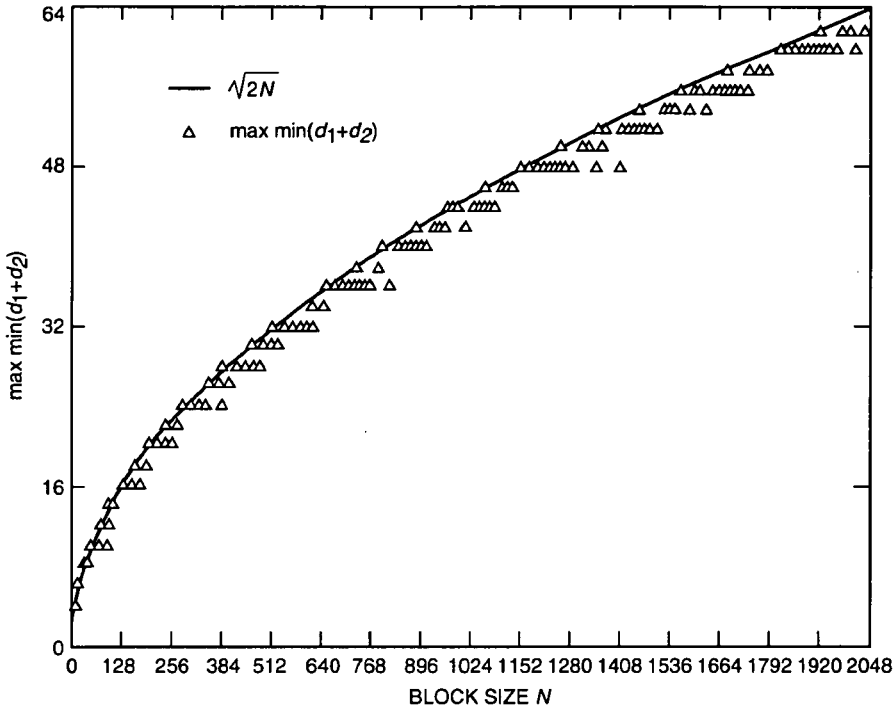

Fig. 3. Maximum over step size of the minimum value of d_1+d_2, as a function of block size N, for nonrandom permutations based on circular shifting.

Section III gave a rationale for trying to ensure that, for weight-2 data sequences, the sum of separations between the 1's induced by different permutations is as large as possible. This correlates with our objective here of choosing a permutation to make the minimum value of $d_1 + d_2$ as large as possible. In fact, we have just shown that a nonrandom permutation can be designed such that this minimum value increases without bound as N gets large. But the relevant summation in Section III was $\sum_i \delta_i$, not $\sum_i d_i$, where $\delta_i = d_i/3$ for the code illustrated in Fig. 1, and noninteger values of δ_i are of no consequence if N is large and edge effects can be ignored. If the design criterion is to maximize the minimum value of $\delta_1 + \delta_2$ for integer δ_1, δ_2 only, then the permutation can be redefined to step in units of 3 instead of 1, if 3 and N are relatively prime. But in this case, this gives the same permutation defined above. The result is that $\delta_1 + \delta_2 \geq \sqrt{2N}$ if $\sqrt{2N}$ is an integer and an optimum step size is chosen as above. For the example above with $N = 32$ and $a = 7$, we can see that the minimum value of $d_1 + d_2$ is 24 if the minimization is restricted to separations d_1, d_2 that are both integer multiples of 3; this corresponds to $\delta_1 + \delta_2 = 8 = \sqrt{2N}$.

The permutation based on circular shifting does a very good job of permuting weight-2 data sequences with low encoded weights into weight-2 sequences with high encoded weights. But this does not imply that we have found the ideal permutation for constructing turbo codes. Indeed, this permutation has an Achilles heel similar to that of block interleavers. For the example with $N = 32$ and $a = 7$, the weight-4 data sequence (10010100100000000000000000000000) is permuted into the sequence (10010000000000000000010010000000), and both of these sequences consist of two self-terminating weight-2 subsequences each characterized by $\delta = 1$. Adding the contributions from all four subsequences

gives $\sum_i \delta_i = 4$, which is only half as big as the minimum value of 8 optimized for weight-2 full sequences only.

We have not yet explored how to modify the permutation based on circular shifting so as to combat weight-4 and higher-weight input sequences while not losing too much optimality with respect to weight-2 sequences. Another open question is how to optimally select two or more different permutations when the turbo code has three or more constituent codes.

3. Semirandom Permutations. The nonrandom permutations discussed in the previous section were based on a limited design objective, namely to attack the most problematical weight-2 input sequences. According to the previous arguments for random permutations, it should be easy to break up input sequences with higher weights (if N is large) without even trying to optimize the permuter. Unfortunately, the permutations designed to optimally break up the lowest-weight sequences possess so much regularity that the random permutation analysis is not applicable when considering the higher weights. This regularity actually enhances the probability that the nonrandom permuter will reproduce bad sequences of weight-4 and higher.

To circumvent this problem, one might try to design permutations to simultaneously break up all weights of input sequences in an optimum manner. This is a formidable task that we have not yet attempted. As an alternative, we propose to evaluate some semirandom permutations, constructed so as to satisfy a limited nonrandom design objective but retaining some degree of randomness to prevent too much regularity.

We have designed one type of semirandom permutation by generating random integers i, $1 \leq i \leq N$, without replacement. An "S-random" permutation is defined as follows: Each randomly selected integer is compared to the S previously selected integers. If the current selection is equal to any of the S previous selections within a distance of $\pm S$, then the current selection is rejected. This process is repeated until all N integers are selected. The searching time for this algorithm increases with S, and it is not guaranteed to finish successfully. However, we have observed that choosing $S < \sqrt{N/2}$ usually produces a solution in reasonable time. Note that for weight-2 input sequences the separations d_1, d_2 satisfy $d_1 + d_2 \geq S + 1$, so the S-random construction (if successful) ensures that the minimum value of $d_1 + d_2$ grows with the block size N. If S is around $\sqrt{N/2}$, this growth rate is about half the growth rate achievable by the nonrandom permutation based on circular shifting. At the other extreme, for $S = 1$, the S-random permutation reduces to a purely random permutation.

V. Conclusion

We have taken a preliminary look at the weight distributions achievable for turbo codes using random, nonrandom, and semirandom permutations. We first drew a distinction between self-terminating and non-self-terminating input sequences, noting that due to the recursiveness of the encoders the latter sequences continue to accumulate encoded weight until they are artificially terminated at the end of the block. Using probabilistic arguments based on selecting the permutations randomly, we concluded that the self-terminating weight-2 data sequences are the most important consideration in the design of the component codes, and we argued that higher-weight sequences have successively decreasing importance. Also, increasing the number of codes and, correspondingly, the number of permutations, makes it more and more likely that the bad input sequences will be broken up by one or more of the permuters.

We have argued that the usual goal of minimizing the weight of the input data sequences associated with the lowest-weight output encoded sequences might be backwards in the case of turbo codes: If the input weight of the most likely error sequences is maximized rather than minimized, the permuters will have a better chance to avoid matching low-weight outputs from each encoder. We have discussed a partial separation of the problem of picking good permutations and that of picking good component

codes that arises from analyzing how fast the output weight grows as a function of the separation between the 1's of weight-2 input sequences.

We have designed nonrandom permutations that ensure that the minimum distance due to weight-2 input sequences grows roughly as $\sqrt{2N}$, where N is the block size. However, these nonrandom permutations amplify the bad effects of higher-weight inputs, and as a result they are inferior in performance to randomly selected permutations. But we have proposed semirandom permutations that perform nearly as well as the designed nonrandom permutations with respect to weight-2 input sequences and are not as susceptible to being foiled by higher-weight inputs.

References

[1] C. Berrou, A. Glavieux, and P. Thitimajshima, "Near Shannon Limit Error-Correcting Coding: Turbo Codes," *Proc. 1993 IEEE International Conference on Communications*, Geneva, Switzerland, pp. 1064–1070, May 1993.

[2] J. Hagenauer and P. Robertson, "Iterative (Turbo) Decoding of Systematic Convolutional Codes With the MAP and SOVA Algorithms," *Proc. of the ITG Source and Channel Coding Conference*, Frankfurt, Germany, pp. 1–9, October 1994.

[3] P. Robertson, "Illuminating the Structure of Code and Decoder of Parallel Concatenated Recursive Systematic (Turbo) Codes," *Proceedings GLOBECOM '94*, San Francisco, California, pp. 1298–1303, December 1994.

[4] D. Divsalar and F. Pollara, "Turbo Codes for Deep-Space Communications," *The Telecommunications and Data Acquisition Progress Report 42-120, October–December 1994*, Jet Propulsion Laboratory, Pasadena, California, pp. 29–39, February 15, 1995.

[5] D. Divsalar and F. Pollara, "Multiple Turbo Codes for Deep-Space Communications," *The Telecommunications and Data Acquisition Progress Report 42-121, January–March 1995*, Jet Propulsion Laboratory, Pasadena, California, pp. 66–77, May 15, 1995.

[6] E. Dunscombe and F. C. Piper, " Optimal Interleaving Scheme for Convolutional Codes," *Electronics Letters*, vol. 25, no. 22, pp. 1517–1518, October 26, 1989.