TABLE XI
WEIGHT ENUMERATOR OF $C_0^{\perp}$, A [52, 27, 9]-CODE

$$\begin{aligned}&1 + 170y^{9} + 442y^{10} + 714y^{11} + 6188y^{12} + 28560y^{13}\\&+53040y^{14} + 77520y^{15} + 308958y^{16} + 879480y^{17}\\&+1270360y^{18} + 1661240y^{19} + 3754569y^{20} + 6824752y^{21}\\&+8065616y^{22} + 9306480y^{23} + 12707500y^{24} + 14775516y^{25}\\&+14775516y^{26} + 14775516y^{27} + 12707500y^{28} + 9306480y^{29}\\&+8065616y^{30} + 6824752y^{31} + 3754569y^{32} + 1661240y^{33}\\&+1270360y^{34} + 879480y^{35} + 308958y^{36} + 77520y^{37}\\&+53040y^{38} + 28560y^{39} + 6188y^{40} + 714y^{41} + 442y^{42}\\&+170y^{43} + y^{52}\end{aligned}$$

TABLE XII
A GENERATOR MATRIX AND WEIGHT DISTRIBUTION OF T54, A [54, 27, 10]-CODE

```
100000000000000000000000000000111010111100000111100010
010000000000000000000000100000100111011101000011100100
001000000000000000000000100011100101001001011001001111
000100000000000000000000100001010100111111011101011111
000010000000000000000000000010101001001101000110011000
000001000000000000000000100010100000001010100001010010
000000100000000000000000100000011011101111110100100100
000000010000000000000000000010110010110001011100000001
000000001000000000000000000011110000010101101000011000
000000000100000000000000100000111101010000110010000010
000000000010000000000000000010100001101110111111010010
000000000001000000000000000010111001111010011001110001
000000000000100000000000000011110101110000001010100000
000000000000010000000000100000111111100010000011011110
000000000000001000000000000010100000110111100111111100
000000000000000100000000100000101010000001110101110000
000000000000000010000000100000011001110001011100111101
000000000000000001000000100001011111101001001000011011
000000000000000000100000100001111100100101000010001000
000000000000000000010000000010000001010100000111010111
000000000000000000001000000011101001100111000101110011
000000000000000000000100000011011101111110100100100001
000000000000000000000010000011000111110010010100001000
000000000000000000000001000010001010110100001100011100
000000000000000000000000100010111111000110100110010011
000000000000000000000000001011110110101110010101010001
000000000000000000000000000111010010011010001100110000
```

Weight	Frequency
0	1
10	255
12	5831
14	44268
16	330174
18	1414284
20	4802364
22	11681193
24	20802220
26	28028274
28	28028274
30	20802220
32	11681193
34	4802364
36	1414284
38	330174
40	44268
42	5831
44	255
54	1

MacWilliams identities, the weight enumerator of $C_0^{\perp}$ is shown in Table XI. Let $\tilde{s}$ be a codeword in C that is not orthogonal to s, where

$$\tilde{s} = (100000000000000000000000000011111\;000101000011101001001).$$

TABLE XIII
WEIGHT ENUMERATOR OF $(C_0^{*})^{\perp}$, A [54, 28, 3]-CODE

$$\begin{aligned}&1 + y^{3} + 255y^{10} + 2754y^{11} + 5831y^{12} + 44268y^{14}\\&+258944y^{15} + 330174y^{16} + 1414284y^{18} + 5469039y^{19}\\&+4802364y^{20} + 11681193y^{22} + 32372352y^{23} + 20802220y^{24}\\&+28028274y^{26} + 58011548y^{27} + 28028274y^{28} + 20802220y^{30}\\&+32372352y^{31} + 11681193y^{32} + 4802364y^{34} + 5469039y^{35}\\&+1414284y^{36} + 330174y^{38} + 258944y^{39} + 44268y^{40}\\&+5831y^{42} + 2754y^{43} + 255y^{44} + y^{51} + y^{54}\end{aligned}$$

By Theorem 1, we get a new code C^{*}, which we denote by T54, a [54, 27, 10]-code, which is generated by $(1, 0, s)$, $(1, 1, \tilde{s})$ and 25 basis vectors of C_0, (0, 0, basis vector of C_0). A generator matrix and the weight distribution of T54 are shown in Table XII. Hence, we get W as given in (4) with $\beta = 12$.

Let C_0^{*} be the subcode of C^{*} consisting of all codewords of weight 0(mod 4). Then $(C_0^{*})^{\perp}$ has the weight enumerator, by the MacWilliams identities, as shown in Table XIII. Therefore, we get S in (4) with $\beta = 12$.

ACKNOWLEDGMENT

The author would like to thank Prof. J. Leon for his fast program help and for comments on the manuscript. The author would also like to thank Prof. V. Pless for her lecture discussions.

REFERENCES

[1] J. H. Conway and N. J. A. Sloane, "A new upper bound on the minimal distance of self-dual codes," *IEEE Trans. Inform. Theory*, vol. 36, pp. 1319–1333, Nov. 1990.

[2] R. A. Brualdi and V. S. Pless, "Weight enumerators of self-dual codes," *IEEE Trans. Inform. Theory*, vol. 37, pp. 1222–1225, July 1991.

[3] H. P. Tsai, "Existence of certain extremal self-dual codes," *IEEE Trans. Inform. Theory*, vol. 38, pp. 501–504, Mar. 1992.

Sequential Decoding of Low-Density Parity-Check Codes by Adaptive Reordering of Parity Checks

Branko Radosavljevic, *Student Member, IEEE*, Erdal Arikan, *Member, IEEE*, and Bruce Hajek, *Fellow, IEEE*

***Abstract*—Decoding algorithms are investigated in which unpruned codeword trees are generated from an ordered list of parity checks. The order is computed from the received message, and low-density parity-check codes are used to help control the growth of the tree. Simulation results are given for the binary erasure channel. They suggest that for**

Manuscript received November 15, 1990; revised November 15, 1991. This work was supported by the Joint Services Electronics Program under Grant N00014-90-J-1270 and by an Office of Naval Research graduate fellowship. This work was presented in part at the IEEE International Symposium on Information Theory, Budapest, Hungary, June 24–28, 1991.

B. Radosavljevic and B. Hajek are with the Coordinated Science Laboratory and the Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL 61801.

E. Arikan is with the Department of Electrical Engineering, Bilkent University, P.K. 8, 06572, Maltepe, Ankara, Turkey.

IEEE Log Number 9203035.

0018-9448/92$03.00

small erasure probability, the method is computationally feasible at rates above the computational cutoff rate.

Index Terms—**Low-density codes, sequential decoding, computational cutoff rate.**

I. Introduction

Sequential decoding is a general method for decoding tree codes. The amount of computation depends on the channel noise and the expected computation per decoded digit is finite only at code rates below R_0, the computational cutoff rate. The present scheme is a modification of standard sequential decoding in an attempt to operate at rates greater than R_0.

With standard sequential decoding, a long burst of noise requires a great deal of computation to resolve. This is illustrated in the following example, taken from [7]. Suppose we have a binary erasure channel and use a convolutional code with rate R. If the first L symbols are erased, then each path with length L symbols in the codeword tree will appear equally likely to the decoder, until they are extended further into the tree. Since there are approximately 2^{LR} such paths, and each path has probability $1/2$ of being extended before the decoder finds the correct one, the expected number of paths searched by the decoder is 2^{LR-1}. A long burst of noise will cause a sequential decoder to perform a great deal of computation even on more general communication channels. For example, consideration of bursts allowed Jacobs and Berlekamp [7] to prove their probabilistic lower bound to the amount of computation on any discrete memoryless channel.

This leads one to consider adaptively reordering the codeword tree, that is, changing the order of the digits used to generate the tree. In this way, one can try to limit the occurrence and duration of bursts of noise. More generally, the goal of reordering the tree is to decrease the computation performed by the sequential decoder.

Apparently, however, one cannot significantly reorder the codeword tree associated with a convolutional code and still obtain a tree that is practical to search. Instead, in this paper we consider decoding linear block codes by sequential decoding. Our method of generating codeword trees for block codes differs from that taken by Wolf [10], Forney [3], and others. The nodes of the trees that we construct are partially specified n-tuples which *may* correspond to valid codewords. If the trees were suitably pruned, the nodes would always correspond to partially specified codewords, and thus, we call our trees *unpruned codeword trees*.[1] In contrast, the nodes in Wolf's trellis correspond to parity check sums of partially specified n-tuples. A similar pruning operation in his case leaves only those paths in the trellis that correspond to partially specified codewords. In addition, the resulting trellis is typically decoded using Viterbi decoding instead of sequential decoding.

Specifically, we consider using low-density parity-check codes. These codes, devised by Gallager [5], [6], are linear block codes characterized by three parameters. A binary (n, q, r) low-density code has blocklength n, and a parity matrix with exactly q 1's in each column and r 1's in each row. Typically, q and r are much smaller than n, resulting in a sparse parity matrix. For these reasons, low-density codes have the following two features. First, given any ordering of the parity checks used to define a low-density code, one can generate an unpruned codeword tree that grows slowly compared to the growth for a dense parity matrix. Second, different parity check orderings yield different trees.

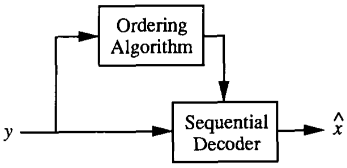

Fig. 1. The general structure of SDR algorithms.

These two features are used by the decoding algorithms presented in this correspondence. The algorithms, which we call sequential decoding with reordering (SDR) algorithms, all have the structure shown in Fig. 1. The ordering algorithm observes the channel output and uses this information to choose a parity check ordering. The sequential decoder then searches the corresponding tree to obtain the decoder output. Decoding algorithms are discussed for the binary erasure and binary symmetric channels. Simulations were performed to investigate the possibility of using this approach at rates greater than R_0.

There exist other methods related to sequential decoding that can operate at rates greater than R_0. In one approach, due to Falconer [2], codewords of a block code with blocklength N are fed in parallel to N convolutional encoders. The result is decoded by N parallel sequential decoders, and the structure of the block code is used to push forward the slowest sequential decoders. A second approach, considered by Pinsker [8], uses concatenated codes, where the inner code is a block code and the outer code is a convolutional code. Both approaches can operate at rates up to capacity. In related research, other decoding algorithms for low-density codes have been presented by Gallager [5], [6, pp. 41–52, 57–59] and Zyablov and Pinsker [11], [12].

We use the following terminology. The vector $\boldsymbol{x} = (x_i)_{i=1}^n$ is the transmitted codeword and $\boldsymbol{y} = (y_i)_{i=1}^n$ is the channel output, where n is the blocklength of the code being used. For simplicity, we assume the channel input is binary.

II. Codeword Trees for Low-Density Codes

To begin with, we mention some relevant properties of a binary (n, q, r) low-density code. There are nq/r rows in the parity matrix, but the rows need not be linearly independent. Thus, the rate R satisfies

$$R \geq 1 - q/r.$$

We use this bound to approximate R. One expects this approximation to be good for the codes used in this study because the parity checks are randomly chosen (using a construction similar to that given in [5], [6, pp. 12–13]). A second property, obtained by Gallager [5], [6, p. 16], is that with q and r fixed with $q \geq 3$, the typical minimum distance of a random low-density code grows linearly with n. Finally, codes in this study are chosen to satisfy two constraints: n is a multiple of r, and no two parity check sets have more than one digit in common. The first constraint is required by the code construction procedure. The second constraint was desirable when using the decoding algorithms considered for the binary symmetric channel, and in addition prevented the degenerate case of having a code with minimum distance equal to two.

To generate an unpruned codeword tree, we start with an ordered list of the nq/r parity equations used to define the

[1] Although we do not explicitly prune the tree, the decoding algorithm essentially does this automatically, by choosing trees in which invalid turns quickly lead to dead ends.

Parity equations

$$\begin{array}{lllllllr} x_1 & + x_2 & + x_3 & & & & & = 0 \\ & & x_3 & + x_4 & + x_5 & & & = 0 \\ x_1 & & + x_3 & & + x_5 & & & = 0 \\ & & & & x_5 & + x_6 & + x_7 & = 0 \end{array}$$

Example:

Binary erasure channel

$y = 0eee101$

Level	0	1	2	3	4
M_i	1	2	2	1	1
e_i		2	1	0	0
δ_i		0	0	0	1

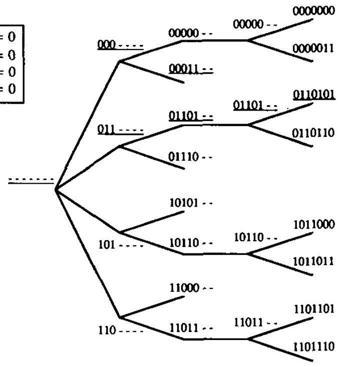

Fig. 2. An unpruned codeword tree for a linear block code defined by four parity equations. (To conserve space, the code is not a low-density code.) Nodes that agree with the received message have underlined labels. The quantities M_i, e_i, and δ_i are defined in Section IV.

code. (See Fig. 2.) Any ordering will work; at this point it is arbitrary. Each parity check corresponds to a level in the tree, and the nodes of the tree correspond to partially specified n-tuples.

At the root node, the n-tuple is completely unspecified. In moving from a node to one of its children, the set of specified digits is increased by those "new" digits (if any) that appear in the current parity check but not in any previous parity check. A different child is created for each of the ways to assign values to these new digits that satisfies the current parity check. Note the possibility that all the digits involved in the current parity check have already been assigned values earlier in the tree (i.e., the current parity check contains no new digits). In this case, a node will have no children if it does not satisfy the current parity check. For this reason, the tree is called "unpruned," to stress the fact that some nodes at intermediate levels may not correspond to valid codewords.

Thus, a given level in the tree consists of all the subsequences that satisfy the current and all previous parity checks. At the final level, all the n-tuples are completely specified, and they all correspond to valid codewords.

This procedure can be used with any linear block code. However, in the general case there is no limit on the number of digits involved in a parity check, other than the blocklength n. A typical parity check may involve $n/2$ digits. If this is the first parity check used to generate the tree, there will be $2^{n/2-1}$ nodes in the first level, and this is too large to be searched by a practical decoder. However, in an (n, q, r) low-density code, each parity check used to define the code involves exactly r digits. The parameter r is typically small, and may be chosen independently of n. All low-density codes considered here have $r \leq 10$. As discussed next, trees for low-density codes have fanout limited by 2^{r-1}, and typically the fanout is much less.

One property of the trees obtained using the method previously given is the variability of their fanout, or growth rate. This contrasts with codeword trees for convolutional codes, which grow at the same rate at every level. To characterize the growth rate, we first present some definitions. The set of digits involved in a parity check is referred to as a parity check set. Given an ordering of the parity checks, each parity check set is partitioned into two groups, the n-set and the o-set. The n-set, or new set, contains the "new" digits (as previously defined), and the o-set, or old set, contains the rest.

Given a tree for an (n, q, r) low-density code, let n_i be the number of digits in the n-set associated with level i in the tree. Also, let N_i be the total number of nodes at level i, with $N_0 \equiv 1$. Then it is not hard to check that, for $1 \leq i \leq nq/r$,

1) if the ith parity check is independent of the preceding parity checks,

$$N_i = 2^{n_i - 1} N_{i-1};$$

2) otherwise,

$$N_i = N_{i-1}.$$

In general,

$$N_i \leq 2^{r-1} N_{i-1}.$$

This result implies that codeword trees for low-density codes tend to grow quickly near the beginning of the tree and more slowly toward the end. This occurs because the growth rate at a given level depends on the size of the corresponding n-set. At a level close to the origin, few digits will have had a chance to appear in previous parity check sets, hence the corresponding n-set tends to be large. The opposite situation holds near the end of the tree. As a result, n_i will roughly be a decreasing function of i. This is not a hard and fast rule, but a good qualitative description that holds regardless of which parity check ordering is used.

In addition to the variable growth rate, codeword trees for low-density codes and those for convolutional codes differ in another important way. With a low-density code, two different subtrees descending from a common parent node can be identical for many levels. This happens because the set of digit values assigned to a node's children depends only on the parity of the child level's o-set. Specifically, two subtrees with a common parent node will agree on the values they assign to codeword digits until a level is reached where the o-set in one subtree has different parity than the o-set in the other subtree. To determine where such a level can occur, consider the root nodes of the two subtrees. The labels of these two nodes will differ only in the values assigned to the digits in the n-set of the level containing the root nodes. Denote this n-set by S. Then the first level at which the two subtrees can differ must have an o-set that contains one or more digits in S.

This property can significantly affect the behavior of a sequential decoder. If the decoder diverges from the correct path in the tree, it may not be able to detect its error for many levels. In this case, backtracking one level at a time would be very inefficient. For example, when using a $(396, 5, 6)$ low-density code and a typical parity check ordering, ten percent of the digits will have 50 or more levels between their appearance in an n-set and their first appearance in an o-set. For this reason, a new sequential decoding algorithm was developed for use on the binary symmetric channel.

III. General Form of the Ordering Algorithm

As a basic heuristic for choosing parity check orderings, consider the following structure. First, let $f(a)$ be some nonnegative measure of the unreliability of symbol y_a. To limit computation, $f(a)$ is constrained to depend on y only through the set $\{y_i, i \in S_a\}$, where $S_a \subseteq \{1, \cdots, n\}$. Specific definitions for f and S_a are given in Section IV for the binary erasure channel.

The algorithm generates an order by choosing parity checks one at a time. At each step, it chooses one of the remaining parity checks that minimizes E, where E is a function chosen to reflect the total noise in the "new" digits involved in a parity check. E is defined by

$$E(C) = \sum_{a \in C \cap A} f(a),$$

where $C \subseteq \{1,\cdots,n\}$ is a parity check set (i.e., C corresponds to a row of the parity matrix) and $A \subseteq \{1,\cdots,n\}$ is the set of digits not involved in any previously chosen parity check.

This heuristic has some desirable properties. First, it requires much less computation than searching through all possible orderings; the specific amount depends on the definition of f and S_a. Second, it tends to push unreliable digits to the rear of the codeword tree. As mentioned in Section II, codeword trees for low-density codes tend to grow more slowly toward the end. Thus, a wrong move by the sequential decoder is less likely and less costly toward the end of the tree than toward the beginning. Third, the heuristic tends to minimize the growth of the tree. At a given level, the growth of the tree depends on the number of new digits in the corresponding parity check. Since f is nonnegative, a parity check with fewer new digits is more likely to be chosen than one with more. A slowly growing tree is desirable because it is easier to search by the sequential decoder.

IV. The Binary Erasure Channel

For the binary erasure channel (BEC), we propose the minimum new erasure (MNE) algorithm. It has the structure described in Section III, with the following definitions. Each set S_a equals $\{a\}$. The function $f(a)$ equals 1 if y_a is an erasure, and equals 0, otherwise. With this definition of f, the function E counts the number of new erasures, hence the algorithm's name. S_a has minimum size; this is appropriate because observing y_a alone completely characterizes its reliability.

The MNE algorithm is related to an upper bound on the number of nodes visited by the sequential decoder, for a given parity check ordering and erasure pattern. Here, the sequential decoder "visits" a node if at some time that node is hypothesized to be on the path corresponding to the transmitted codeword. This upper bound is found by counting the nodes in the codeword tree that agree with the unerased portion of the received message. (See Fig. 2.) Let M_i be the number of such nodes at level i in the codeword tree for an (n, q, r) low-density code. Then,

$$M_0 = 1,$$
$$M_i = 2^{a_i} M_{i-1}, \qquad 1 \le i \le m,$$

where

$$a_i = e_i + \delta_i - 1,$$
$$m = nq/r = \text{the number of levels in the codeword tree},$$

and e_i is the number of new erasures at level i, which is the number of erasures in the n-set at level i. The terms n-set and o-set are defined in Section II. The quantity δ_i is defined as follows. We assign $\delta_i = 1$ if $e_i = 0$ and in addition, either: 1) there are no erasures in the o-set at level i, or 2) the parity of the erasures in the o-set at level i, taken as a group, assumes only one value among the surviving nodes at level $i - 1$. Otherwise, we assign $\delta_i = 0$.

An upper bound to the number of nodes visited by the sequential decoder is given by

$$M = \sum_{i=0}^{m} M_i = \sum_{i=0}^{m} \prod_{l=1}^{i} 2^{a_l},$$

where M is a function of the channel output and the parity check ordering. The MNE algorithm can be interpreted as a greedy heuristic for minimizing M, ignoring the δ_i term. The first parity check is chosen to minimize e_1; ignoring δ_1, this would minimize M_1. Similarly, the second parity check is chosen to minimize e_2, and so on.

A straightforward implementation of this algorithm searches the entire set of unused parity checks every time a new parity check is chosen. Also, the number of new erasures involved in each parity check is recalculated for every search, since the definition of "new" changes. For an (n, q, r) low density code, where n is the blocklength, this implementation requires $O(q^2n^2/r)$ computation and $O(qn)$ memory. A more efficient implementation [9, p. 112–118] requires $O(qn)$ computation and $O(qn)$ memory.

The MNE ordering algorithm was implemented on a computer. The objectives were to estimate the maximum number of erasures the decoding algorithm can handle and to compare this number with what can theoretically be achieved with standard sequential decoding. To form a complete decoder, the MNE algorithm was coupled with the stack algorithm [1, pp. 326–328], a standard sequential decoding algorithm. Fixed weight pseudorandom erasure patterns were generated and decoded by the computer, and the results were used to estimate the expected amount of computation and the probability of decoding failure (p_{DF}). Graphs of these quantities versus number of erasures are shown in Fig. 3. Data points with no decoding failures were omitted from the graph of p_{DF}. To gauge the effectiveness of the MNE algorithm, simulations were also performed with a random ordering algorithm, which chooses parity check orderings with equal probability for each possible ordering. The results of using the two algorithms with a (395, 5, 6) low-density code are shown in Fig. 4.

The measure of computation was the number of steps performed by the sequential decoder; specifically, the number of times the entry at the top of the stack was extended. The all-zeros codeword was always used. To compensate for this, the sequential decoder searched the codeword tree branches in reverse numerical order, and thus the all-zeros branch was always searched last. As a result, the expected computation and probability of decoding failure are upper bounds to what would be expected with random codewords. After each trial, which consisted of decoding a fixed weight pseudorandom erasure pattern, the measured average computation, c, and the measured standard deviation of computation, s, were calculated. The standard deviation of c, denoted by $\hat{s}$, was estimated by $\hat{s} = s/\sqrt{t}$, where t is the number of trials. This relation assumes independent trials. For each data point, the computer performed at least 200 trials and at most 3000; within this range, it stopped if $\hat{s} < (0.025)c$. The number of trials resulting in decoding errors, N_E, and the number of aborted trials, N_A, were recorded as well. A trial was aborted if the number of steps performed by the sequential decoder reached ten thousand, or if the stack size reached 200. However, the limit on stack size was unnecessary, because none of the trials aborted for this reason. The estimated probability of decoding failure was given by $p_{\text{DF}} = (N_E + N_A)/T$, where T is the total number of trials. Note that all codes were chosen to have the property that no two parity check sets contain more than one digit in common, as mentioned in Section II.

The results shown in Fig. 4 indicate that the MNE algorithm performs significantly better than random ordering. However, to compare this approach with standard sequential decoding, we

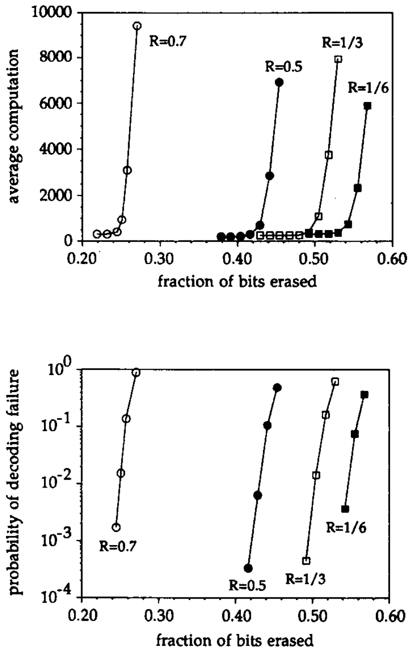

Fig. 3. Simulation results for the MNE algorithm. The code rates were 0.7, 0.5, 1/3, and 1/6, with parameters (1000, 3, 10), (396, 3, 6), (396, 4, 6), and (396, 5, 6).

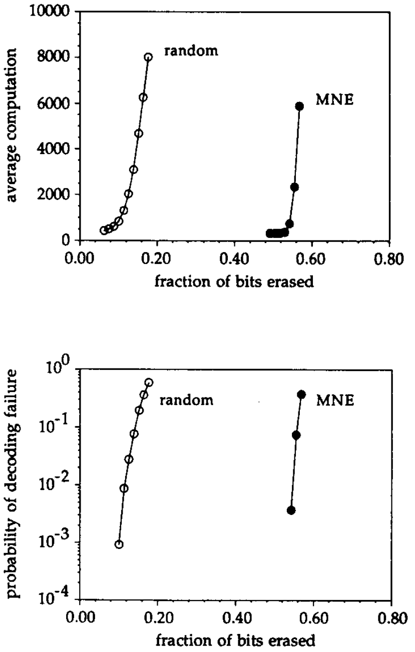

Fig. 4. Simulation results for random ordering and the MNE algorithm, used with a (396, 5, 6) code.

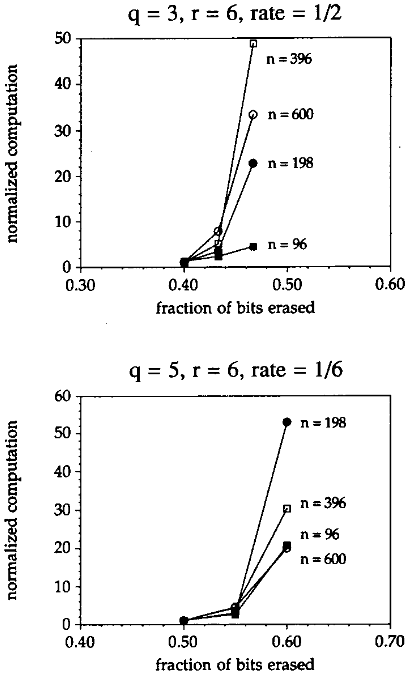

Fig. 5. Normalized computation using the MNE algorithm, for two sets of (q, r) values and various values of n.

must consider the asymptotic behavior. Simulations were performed using codes with fixed q and r values, but varying blocklengths. This is shown in Fig. 5, which contains graphs of normalized computation versus fraction of bits erased for $q = 3$, $r = 6$ and $q = 5$, $r = 6$. Here, the normalized computation is the amount of computation divided by $nq/r + 1$, the number of levels in the codeword tree with the root node. Both graphs seem to indicate that the cutoff point, where the average computation starts to increase, occurs at roughly the same value of the fraction of bits erased, as n varies with q and r fixed. In other words, for fixed q and r, the number of decodable erasures when using the MNE algorithm appears to increase linearly in n. This behavior is expected, since the typical minimum distance of an ensemble of (n, q, r) low-density codes grows asymptotically linearly in n, for $q \geq 3$ [5], [6, p. 16]. In addition, the cutoff becomes sharper with increasing n, in the following sense. For both graphs, at the first data point, normalized computation decreases with increasing n (for example, 1.18, 1.095, 1.0056, and 1.0050 for n equals 96, 198, 396 and 600, respectively, with $q = 3$, $r = 6$). This represents a feasible operating point. At the second and third data points, normalized computation tends to increase with increasing n. This trend is somewhat obscured because trials were aborted if the computation reached 10 000. Thus, normalized computation cannot exceed $10\,000/(nq/r + 1)$.

What can we conclude about using the MNE algorithm at rates above R_0, the computational cutoff rate for sequential decoding? Estimates of ε_{MNE}, the maximum erasure frequency handled by the MNE algorithm, are shown in Table I. The upper range is determined by the data point in Fig. 3 where the average computation first exceeds two times its minimum possible value, $nq/r + 1$. The lower range is given by the last data point with average computation less than $1.5(nq/r + 1)$. For example, for the (396, 5, 6) code, the relevant data points have 215 and 210 erasures out of 396 digits. The well-known formula of R_0 on a discrete memoryless channel [4, p. 279] applied to the BEC yields

$$R_0 = 1 - \log_2(1 + \varepsilon),$$

TABLE I
Cutoff Points for the MNE Algorithm and Standard Sequential Decoding on the BEC

Code Parameters			Rate		
n	j	k	R	$\varepsilon_{\mathrm{MNE}}$	$\varepsilon_{\mathrm{SD}}$
1000	3	10	0.7	0.245–0.251	0.2311
396	3	6	0.5	0.404–0.429	0.4142
396	4	6	0.333	0.480–0.505	0.5874
396	5	6	0.167	0.530–0.543	0.7818

where ε is the channel erasure probability and R_0 is measured in bits. Inverting this formula yields $\varepsilon_{\mathrm{SD}}$, the maximum erasure probability that sequential decoding can handle for a given code rate. Table I shows values of $\varepsilon_{\mathrm{SD}}$ corresponding to the code rates used in the simulations. For the $(396, 4, 6)$ and $(396, 5, 6)$ codes, $\varepsilon_{\mathrm{MNE}}$ for the MNE algorithm is significantly less than $\varepsilon_{\mathrm{SD}}$. Thus, it seems that one cannot exceed R_0 using the MNE algorithm with these values of q and r. The $(396, 3, 6)$ code looks more promising. Its value of $\varepsilon_{\mathrm{MNE}}$ nearly equals $\varepsilon_{\mathrm{SD}}$.

Finally, for the $(1000, 3, 10)$ code, $\varepsilon_{\mathrm{MNE}}$ is estimated to be greater than $\varepsilon_{\mathrm{SD}}$. To interpret this, recall that the simulations were performed with fixed weight erasure patterns, not independent erasures. At $n = 1000$, with 245 erasures, normalized computation was 1.33. Extending this point proportionately out to $n = 5000$ resulted in normalized computation equal to 1.0013. If the conjecture about a linearly increasing operating region is true, then it seems that an erasure frequency of 0.245 is within the operating region for all n (with $q = 3$, $r = 10$). This implies that for sufficiently large n, the MNE algorithm can handle independent erasures with erasure probability at least up to 0.245. This would exceed the performance of standard sequential decoding, which has $\varepsilon_{\mathrm{SD}} = 0.2311$. Though this seems plausible, nevertheless a finite number of simulations cannot prove an asymptotic result.

Decoding errors occurred much less often than decoding failures. A decoding error occurs when the sequential decoder completes its search but outputs the wrong codeword. As defined here, a decoding failure occurs if a trial results in a decoding error or is aborted because of too much computation. Out of 31 069 trials used to generate the graphs in Fig. 3, only 8 resulted in decoding errors. Thus, if the MNE algorithm is used on a channel with feedback and retransmission capabilities, one can achieve decoding error rates several orders of magnitude smaller than p_{DF}. However, the effective information rate would be lower than the code rate, because of retransmissions. For the results shown in Fig. 5, a significant number of decoding errors occurred when using the $(96, 3, 6)$ and $(198, 3, 6)$ codes. It seems one should avoid using blocklengths this small because, for the trials used to generate Fig. 5, none of the other codes had any decoding errors.

This discussion implicitly assumes that the codes used in the simulations are "typical" low-density codes. The assumption is reasonable in light of Gallager's result that almost all the low-density codes in his ensemble have minimum distance greater than a single lower bound, $\delta_{qr}n$. In addition, for the binary symmetric channel, simulation results obtained using two randomly chosen $(396, 5, 6)$ low-density codes are found to be very close.

Given the extra work performed by the ordering algorithm, why doesn't the MNE algorithm outperform standard sequential decoding in all cases? One feature that limits the SDR approach, not encountered in standard sequential decoding, is that codeword trees for low-density codes tend to grow more rapidly near the beginning than near the end. This happens regardless of the parity check ordering, as discussed in Section II. Thus, the early part of the tree looks like a code with higher rate than the original code. This is a drawback, since the sequential decoder works well only at rates less than R_0. However, the MNE algorithm tends to push erased digits toward the end of the tree, and this helps the sequential decoder. The net result of these effects determines whether the overall performance is better than standard sequential decoding. Another limitation of the SDR approach is that the distance properties of low-density codes are probably not the best possible.

V. The Binary Symmetric Channel

Results for the binary symmetric channel (BSC) were not as promising as for the BEC. Several ordering algorithms were implemented [9], but several problems were encountered. First, the same features that limit the SDR approach for the BEC apply to the BSC as well. The trees searched by the sequential decoder tend to grow more quickly near the root node than at later levels, and low-density codes may have suboptimal distance properties. Second, as discussed in Section II, the trees used here are such that if the sequential decoder diverges from the correct path in the tree, it may not be able to detect its error for many levels. This is not a problem on the BEC because the unerased portion of a received message serves to eliminate from consideration many nodes in the tree. Since conventional sequential decoders do not work well for this application on the BSC, a new sequential decoder [9] was designed, but the resulting performance was still not good enough to approach the cutoff rate. Third, the BSC is, in a sense, the worst possible channel for SDR algorithms. Each digit at the channel output, taken by itself, is equally unreliable—unlike, for example, the BEC or the Gaussian noise channel. This makes it difficult to generate a good parity check ordering.

VI. Conclusion

Sequential decoding with reordering (SDR) algorithms were designed for the binary erasure and binary symmetric channels. For both channels, with respect to standard sequential decoding, relative performance improved with increasing code rate. For the erasure channel, simulations suggest that one can use rate 0.7 low-density codes to handle a higher probability of erasure than standard sequential decoding. A similar result did not hold for the errors channel.

For future work, one could consider different parity-check ordering algorithms. In addition, SDR algorithms could be generated for other communication channels. For example, one could define $f(a) = h(P(x_i = 0 | y_i))$, where h is the entropy function. The resulting SDR algorithm could be used on channels with real-valued outputs, such as the Gaussian noise channel.

Acknowledgment

The authors thank the reviewers for their useful comments.

References

[1] G. C. Clark and J. B. Cain, *Error-Correction Coding for Digital Communications.* New York: Plenum Press, 1981.

[2] D. D. Falconer, "A hybrid sequential and algebraic decoding scheme," Ph.D. dissert., Dept. of Elect. Eng., Massachusetts Inst. of Technol., Cambridge, MA, 1967.

[3] G. D. Forney, Jr., "Coset codes—Part II: Binary lattices and related codes," *IEEE Trans. Inform. Theory*, vol. 34, pp. 1152–1187, Sept. 1988.

[4] R. G. Gallager, *Information Theory and Reliable Communication.* New York: Wiley, 1968.

[5] ——, "Low-density parity-check codes," *IRE Trans. Inform. Theory*, vol. IT-8, pp. 21–28, Jan. 1962.

[6] ——, *Low-Density Parity-Check Codes.* Cambridge, MA: M.I.T. Press, 1963.

[7] I. M. Jacobs and E. R. Berlekamp, "A lower bound to the distribution of computation for sequential decoding," *IEEE Trans. Inform. Theory*, vol. IT-13, pp. 167–174, Apr. 1967.

[8] M. S. Pinsker, "On the complexity of decoding," *Probl. Inform. Trans.* (translation of *Probl. Peredach. Inform.*), vol. 1, pp. 84–86, Jan.–Mar. 1965.

[9] B. Radosavljevic, "Sequential decoding with adaptive reordering of codeword trees," M.S. thesis, Tech. Rep. UILU-ENG-90-2209, Coor. Sci. Lab., Univ. of Illinois, Urbana, IL, Mar. 1990.

[10] J. K. Wolf, "Efficient maximum likelihood decoding of linear blocks codes using a trellis," *IEEE Trans. Inform. Theory*, vol. IT-24, pp. 76–80, Jan. 1978.

[11] V. V. Zyablov and M. S. Pinsker, "Decoding complexity of low-density codes for transmission in a channel with erasures," *Probl. Inform. Trans.* (translation of *Probl. Peredach. Inform.*), vol. 10, pp. 10–21, Jan.–Mar. 1974.

[12] ——, "Estimation of the error-correction complexity for Gallager low-density codes," *Probl. Inform. Trans.* (translation of *Probl. Peredach. Inform.*), vol. 11, pp. 18–28, Jan.–Mar. 1975.

The Lempel–Ziv Algorithm and Message Complexity

E. N. Gilbert, *Fellow, IEEE*, and T. T. Kadota, *Fellow, IEEE*

***Abstract*—Data compression has been suggested as a non-parametric way of discriminating between message sources (e.g., a complex noise message should compress less than a more redundant signal message). Compressions obtained from a Lempel–Ziv algorithm for relatively short messages, such as those encountered in practice, are examined. The intuitive notion of message complexity has less connection with compression than one might expect from known asymptotic results about infinite messages. Nevertheless, discrimination by compression remains an interesting possibility.**

***Index Terms*—Lempel–Ziv parsing, data compression, nonparametric discrimination.**

I. Introduction

A discrimination question asks which of two possible sources (e.g., signal or noise) produced a received message. If good probabilistic descriptions of the sources were known, a likelihood ratio might decide. Often the sources are not well understood and nonparametric methods are needed. If the two sources differ in redundancy, then data compression may be used ([1]–[3]). Thus, redundant signal messages will compress more than non-redundant noise messages. Here, we examine Lempel–Ziv compression for properties that may be useful in discriminating binary sources.

There are several Lempel–Ziv algorithms. The one we study scans the message digits in order and creates a new word whenever the current block of unparsed digits differs from all words already found. Suppose the first m words from the message contain a total of s_m digits. Then, s_m will tend to be large for repetitious or redundant messages. As m goes to infinity, a well-known result ([2], [4]) shows that s_m increases like $(m \log m)/H$, where H is the entropy of the source. Thus, sources with different entropies could be perfectly discriminated by parsing infinite messages.

Long messages are not available in real discrimination problems; moreover the asymptotic formula is found to be inaccurate until m is very large indeed. For these reasons, we avoid asymptotic results and concentrate on finite values of m. Even small values like $m = 3$ or 4 provide interesting counterexamples to show that properties holding for large m need not hold in general. If m is small, and especially for highly redundant sources, compression is found to be only very roughly related to the intuitive notion of message complexity. For instance, two messages that are just reversals of one another may compress very differently (Section V). However, compression does give a basis for discriminating. The main analytical results follow from a useful identity (1) that enumerates parsed messages.

II. Enumeration

A list L_m of all possible messages composed of m parsed words will be a basic tool (L_2 lists six messages 0,00; 0,01; 0,1; 1,0; 1,10; 1,11). In principle, any probability related to the first m words (e.g., the probability distribution of s_m) from any known source can be computed by evaluating the probability of each message in L_m. Since L_m contains $(m+1)!$ messages [5], such computations become impractical with large m but here, with smaller m, they give some insight. Memoryless sources, considered in Section III, allow simplifications for larger m.

An inductive procedure constructs L_m. Suppose $L_1, \cdots, L_{m-1}$ are known. Construct L_m as follows. One message in L_m is $0, 00, 000, \cdots$ with $m(m+1)/2$ digits, all zeros. Another message has all ones. Other messages have a number $k \geq 0$ of words beginning 0 and $m - k \geq 0$ words beginning 1. The first word beginning with 0 is 0 itself. Any remaining words beginning with 0 are, in parsing order, the words $A_1, A_2, \cdots$ of one member of L_{k-1} with an extra 0 added as prefix to each word. Similarly, the words beginning with 1 will be $1, 1B_1, 1B_2, \cdots$ where $B_1, B_2, \cdots$ are the words of a parsed message in L_{m-k-1}. Thus, with $m = 5$ and $k = 2$, the choices 1 from L_1 and 0,01 from L_2 determine the words 0,01 and 1,10,101. These two parsings can be interleaved, keeping the words $0A_i$ and $1B_j$ in the same order, to produce parsed messages like $1, 10, 0, 101, 01$. In general, for given m, k, and given choices from L_{k-1} and L_{m-k-1}, there are $\binom{m}{k}$ possible interleavings.

One can easily verify that this procedure lists only Lempel–Ziv parsings. That is, 1) the words of each parsing are distinct and 2) each parsing contains, as words, all proper prefixes of each of its words. Conversely, every Lempel–Ziv parsing into m words is found by this procedure.

Counting the choices that arise in this procedure leads to a count of the number $f(m, i, j)$ of parsings with m words, i zeros, and j ones. The result is stated most simply in terms of the generating function $F_m(x, y) = \sum_{i,j} f(m, i, j) x^i y^j$, which is found to satisfy the recurrence

$$F_m(x, y) = \sum_k \binom{m}{k} x^k y^{m-k} F_{k-1}(x, y) F_{m-k-1}(x, y). \quad (1)$$

Manuscript received October 7, 1991; revised March 3, 1992. This work was presented in part at the IEEE International Symposium on Information Theory, Budapest, Hungary, June 24–28, 1991.

The authors are with AT & T Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974.

IEEE Log Number 9201221.

0018-9448/92$03.00 © 1992 IEEE