F. Jelinek

Fast Sequential Decoding Algorithm Using a Stack*

Abstract: In this paper a new sequential decoding algorithm is introduced that uses stack storage at the receiver. It is much simpler to describe and analyze than the Fano algorithm, and is about six times faster than the latter at transmission rates equal to R_{comp}, the rate below which the average number of decoding steps is bounded by a constant. Practical problems connected with implementing the stack algorithm are discussed and a scheme is described that facilitates satisfactory performance even with limited stack storage capacity. Preliminary simulation results estimating the decoding effort and the needed stack size are presented.

1. Introduction

Sequential decoding is a method of communication through noisy channels that uses tree codes (see Fig. 2). Several decoding algorithms have been suggested,[1-3] the one due to Fano being universally acknowledged as the best (see also Sec. 10.4 of Jelinek[4]). The first two algorithms have the common characteristic that, when used with appropriate tree codes signaling over memoryless channels, their probability of error decreases exponentially to zero at all transmission rates less than capacity C, while for all rates less than a value called R_{comp}, the average amount of decoding effort per decoded information bit is bounded above by a constant. R_{comp} is a function of the channel transmission probabilities only, and exceeds $C/2$ for all binary symmetric channels. Figure 1 contains the plot of R_{comp}/C as a function of the crossover probability p. In contrast, all widely used methods of algebraic coding achieve arbitrarily reliable performance only when the transmission rate is sufficiently reduced toward zero. (It ought to be added, however, that algebraic schemes are much simpler than sequential ones.) Furthermore, these methods work only for symmetrical channels with the same number of outputs as inputs.

In this paper we will introduce a new sequential algorithm that is faster than all competing ones, and that is very simple to describe and analyze. To realize its speed advantage without an increase in error probability, it is necessary to increase by a considerable amount the memory of the decoder. However, in a suitable environment (e.g., when a general purpose computer is used for decoding as is done in the Pioneer space program) the increase in speed will be well worth the added cost. The price of memories will continue to drop rapidly in the foreseeable future, thereby widening the applicability of our method. It is hoped that this paper will stimulate further research into tree decoding algorithms. In particular, it would be interesting to know what compromises between the present and Fano's algorithm are possible, and what the trade-off is between finite memory size and speed.

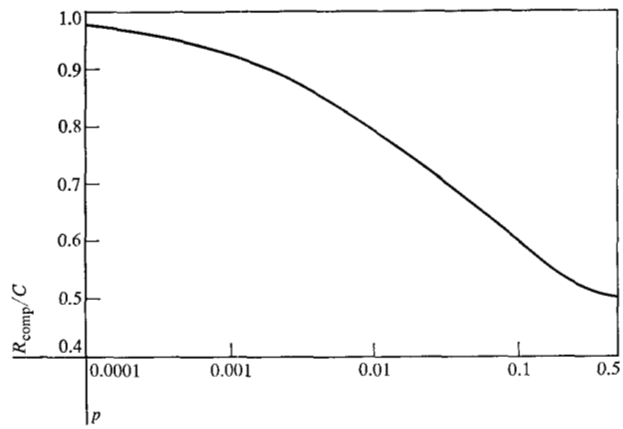


Figure 1 The graph of R_{comp}/C as a function of the crossover probability p of a binary symmetric channel.

Since no previous knowledge of sequential decoding is assumed (the exception being Section 7), we start by describing tree encoding. Then in Section 3 we give our stack decoding algorithm and provide an example. Section 4 contains an outline of the analysis of the present scheme that leads to previously known results about the average number of decoding steps and the probability of error.

The author's current address is School of Electrical Engineering, Cornell University, Ithaca, New York 14850.

* A skeletal version of this paper was presented at the Third Princeton Conference on System Science, March 1969.

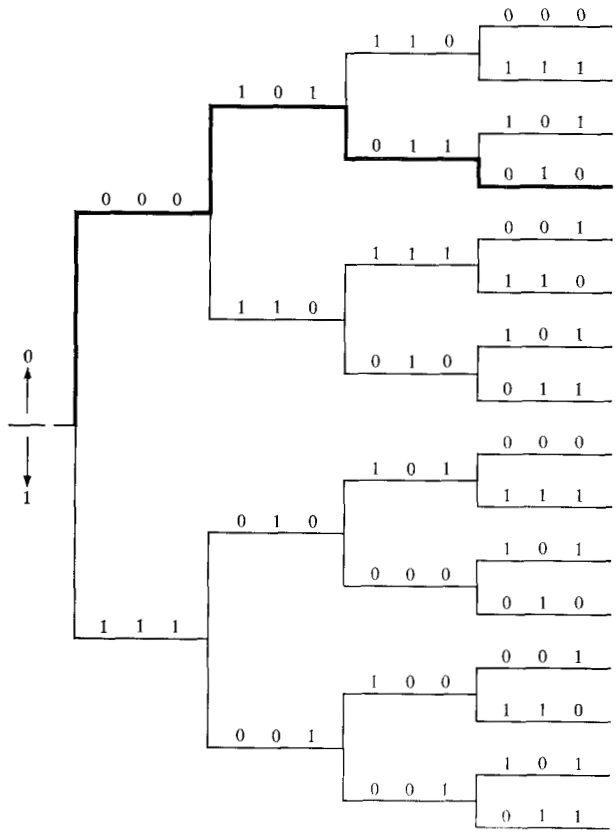


Figure 2 Example of a binary tree code of rate $R = 1/3$.

We next describe the outcome of simulations that enable us to compare the stack and Fano's algorithm (Fig. 6). In Section 6 we modify the algorithm to limit the growth of the stack, and show how effectively this can be done (Fig. 7). Section 7 contains a procedure that handles stack overflows and thereby prevents catastrophic terminations. Finally, Section 8 describes how a stack is actually implemented and in what manner it specifies the required information.

2. Tree encoding

Let us assume that we desire to communicate over a *discrete memoryless channel* characterized by a transmission probability matrix $[w_0(\eta/\xi)]$, where $\xi \in \{0, 1, \cdots, \alpha - 1\}$ are the channel inputs and $\eta \in \{0, 1, \cdots, \beta - 1\}$ are the corresponding channel outputs. Thus, for any integer n, the probability that an arbitrary output sequence η_1, η_2, $\cdots$, η_n is received, given that an arbitrary input sequence $\xi_1, \xi_2, \cdots, \xi_n$ was transmitted, is equal to

$$\prod_{i=1}^{n} w_0(\eta_i/\xi_i).$$

Let us further assume that the information source generates outputs $s \in \{0, 1, \cdots, d - 1\}$ that are to be communicated to the user located at the other end of the channel. An appropriate tree code of rate $R \equiv 1/(n_0 \log_2 d)$ bits per channel used will have d branches leaving every node, each branch being associated with a sequence of n_0 channel input digits. An example of a tree code appropriate for a binary source ($d = 2$) and a channel with binary inputs ($\alpha = 2$) is given in Fig. 2. Since $n_0 = 3$, the code has rate $R = 1/3$. The correspondence between source outputs and channel inputs (i.e., the encoding rule) is as follows.

At the start of the encoding process the encoder is "located" at the root of the tree. If the first digit s_1 generated by the source is a 0, the encoder follows the upper branch out of the root node to the next node and puts out (transmits through the channel) the sequence associated with that branch (in this case 000). If $s_1 = 1$, the encoder follows the lower branch to the next node and puts out the corresponding sequence (in this case 111). In general, just before the source generates the ith digit s_i, the encoder is located at some node $i - 1$ branches deep in the tree. If $s_i = 0$ the encoder leaves this node along the top branch and, if $s_i = 1$, it leaves the node along the bottom branch; in both cases it transmits the sequence corresponding to the branch along which it traveled. In this way a message sequence $s_1, s_2, \cdots$ traces a path through the tree and the tree sequence corresponding to that path is then transmitted. (The generalization to a d-nary tree is obvious: At time i the encoder leaves the current node along the $(s_i + 1)$th branch of the fan-out and transmits the corresponding sequence, where $s_i \in \{0, 1, \cdots, d - 1\}$.) Thus in Fig. 2 the message sequence 0011 determines the path indicated by the thick line and causes the sequence 000101011010 to be sent through the channel.

In principle, the encoding tree can be continued indefinitely and thus the total number Γ of levels it can have (the tree displayed in Fig. 2 has four levels) is arbitrary. Since a d-nary tree with Γ levels has d^Γ paths, there is a problem of how the encoder can store its tree code. We will not concern ourselves here with that issue. Let the reader be assured that no difficulty arises. The tree is never actually stored; all that is stored is a very simple algorithm that can generate the digits associated with the tree branches whenever the former are required. The usual tree codes are called *convolutional* and their description can be found on pp. 377 to 383 of Ref. 4. It will be easier for us to continue to act as if the encoder stored the entire tree.

3. The decoding algorithm

From the preceding description of tree encoding it follows that the natural transmission units we are dealing with are not channel digits themselves, but sequences

of n_0 of these that correspond to the tree branches. Accordingly, it will simplify further discussion if we henceforth restrict our attention to the *n_0-product channel* $[w(y/x)]$ whose input symbols x and output symbols y are strings of n_0 inputs and outputs of the underlying channel $[w_0(\eta/\xi)]$. From this point of view, the product channel input alphabet corresponding to the tree code of Fig. 2 is octal, and the message sequence 0011 causes 0532 to be transmitted. Let x^* represent some sequence $\xi_1^*, \xi_2^*, \cdots, \xi_{n_0}^*$ and let y^* represent $\eta_1^*, \eta_2^*, \cdots, \eta_{n_0}^*$. Then

$$w(y^*/x^*) \equiv \prod_{i=1}^{n_0} w_0(\eta_i^*/\xi_i^*). \tag{1}$$

Thus the product channel has inputs $x \in \{0, 1, \cdots, a-1\}$ and outputs $y \in \{0, 1, \cdots, b-1\}$, where $a = \alpha^n$, $b = \beta^n$, and α and β are the sizes of the input and output alphabets of the underlying channel, respectively. A tree path of length i is specified by the vector $\mathbf{s}^i \equiv (s_1, s_2, \cdots, s_i)$ (we will use boldface for vectors and superscripts will indicate their length) formed from the corresponding message digits. We will denote by $x_j(\mathbf{s}^i)$, $j \leq i$, the transmitted symbol associated with the jth branch of the path $\mathbf{s}^i$. Thus in Fig. 2 $x_3(010s_4) = 7$ for all s_4, and $x_2(10s_3s_4) = 2$ for all s_3s_4. Let us now assume that a sequence $\mathbf{y}^\Gamma$ was received through the channel (Γ is the number of levels in the tree) and that we wish to *decode* this sequence, i.e., to determine the identity of the message sequence $\mathbf{s}^\Gamma$ put out by the source. We will denote by $\hat{\mathbf{s}}^\Gamma$ the receiver's estimate of $\mathbf{s}^\Gamma$ and, of course, we aim at having $\hat{\mathbf{s}}^\Gamma$ equal to $\mathbf{s}^\Gamma$. We recall that, when $\mathbf{s}^\Gamma$ was inserted into the encoder, the latter produced the channel input sequence $\mathbf{x}^\Gamma(\mathbf{s}^\Gamma) = x_1(\mathbf{s}^\Gamma), x_2(\mathbf{s}^\Gamma), \cdots, x_\Gamma(\mathbf{s}^\Gamma)$ in the way described in the preceding section. Our problem is to specify the operation of the *decoder*.

Let $r(x)$, $x \in \{0, 1, \cdots, a-1\}$ be a suitable probability distribution (its choice will be clarified below) over the input symbols of the product channel, and define the output distribution

$$w_r(y) \equiv \sum_x w(y \mid x) r(x). \tag{2}$$

If the sequence $\mathbf{y}^\Gamma$ was received, we will be interested in the *likelihoods*

$$L(\mathbf{s}^i) \equiv \sum_{j=1}^{i} \lambda_j(\mathbf{s}^j) \tag{3}$$

of the various paths $\mathbf{s}^i = (\mathbf{s}^i, s_{i+1}, \cdots, s_j)$, where the *branch likelihood function* $\lambda_i(\mathbf{s}^i)$ of the branch leading from node $\mathbf{s}^{i-1}$ to node $\mathbf{s}^i$ (note that a path uniquely defines the tree node on which it terminates, and vice versa) is defined by

$$\lambda_i(\mathbf{s}^i) \equiv \log_2 \frac{w[y_i \mid x_i(\mathbf{s}^i)]}{w_r(y_i)} - n_0 R. \tag{4}$$

From the decoder's point of view, $L(\mathbf{s}^i)$ and $\lambda_i(\mathbf{s}^i)$ are functions of the paths only, since the received sequence $\mathbf{y}^\Gamma$ is fixed throughout the decoding process and the branch symbols $x_i(\mathbf{s}^i)$ are determined by the tree code that is known in advance.

We are now ready to describe the decoding algorithm for a binary tree ($d = 2$). This restriction will make the explanation easier to follow, but will be subsequently removed.

(1) Compute $L(0) = \lambda_1(0)$ and $L(1) = \lambda_1(1)$, the likelihoods of the two branches' leaving the root node (see Fig. 2), and place them into the decoder's memory.
(2) If $L(0) \geq L(1)$, eliminate $L(0)$ from the decoder's memory and compute $L(00) = L(0) + \lambda_2(00)$ and $L(01) = L(0) + \lambda_2(01)$. Otherwise, eliminate $L(1)$ and compute $L(10) = L(1) + \lambda_2(10)$ and $L(11) = L(1) + \lambda_2(11)$. Therefore we end with the likelihoods of three paths in the decoder's memory, two paths of length 2 and one of length 1.
(3) Arrange the likelihoods of the three paths in decreasing order. Take the path corresponding to the topmost likelihood, compute the likelihoods of its two possible one-branch extensions, and eliminate from memory the likelihood of the just extended path [e.g., if $L(0)$, $L(10)$, $L(11)$ are in the memory and, say, $L(10) \geq L(0) \geq L(11)$, then $L(10)$ is replaced in the memory by the newly computed values of $L(100) = L(10) + \lambda_3(100)$ and $L(101) = L(10) + \lambda_3(101)$. In case, say, $L(0) \geq L(11) \geq L(10)$, then $L(0)$ is replaced by $L(00)$ and $L(01)$]. At the end of this step the decoder's memory will contain the likelihoods of four paths, and either two of these will be of length 3 and one each of lengths 1 and 2, or all four paths will be of length 2.
(4) The search pattern is now clear. After the kth step, the decoder's memory will contain exactly $k + 1$ likelihoods corresponding to paths of various lengths and *different* end nodes. The $(k + 1)$th step will consist of finding the largest likelihood in the memory, determining the path to which it corresponds, and replacing that likelihood with the likelihoods corresponding to the two one-branch extensions of that path.
(5) The decoding process terminates when the path to be extended next is of length Γ, i.e., leads from the root node to the last level of the tree.

Because of the ordered nature of the decoder's memory, we refer to it as a *stack*. We next illustrate the *stack decoding algorithm* by an example. Consider the binary tree of Fig. 3 with nodes as numbered. Let paths be associated with their terminal nodes. Let the numbers written on top of the branches represent the values of the corresponding branch likelihood function for some received sequence $\mathbf{y}^3$. Thus, the likelihood of path 8 is equal to $-6 + 1 + 1 = -4$. The state of the stack

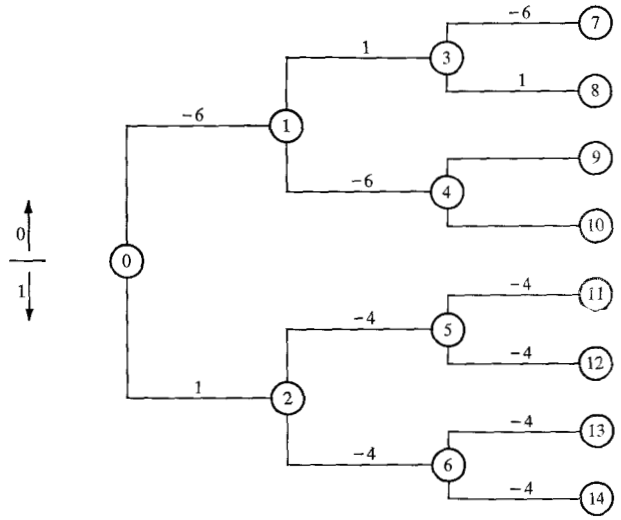
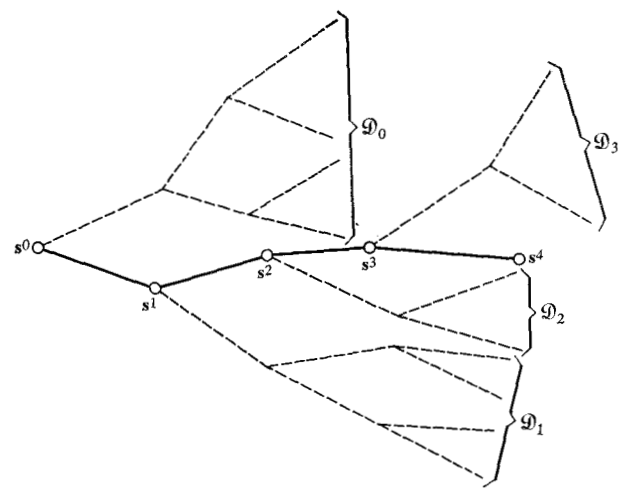


Figure 3 An example of likelihood-value assignment to tree branches induced by a code and a received sequence.

Figure 4 The $\mathfrak{D}_i$ classification of nodes of a binary tree relative to the transmitted message $(s_1, s_2, s_3, s_4, \cdots)$.



during decoding would then be as follows (topmost path on the left).

1st state:	0
2nd state:	2, 1
3rd state:	5, 6, 1
4th state:	6, 1, 11, 12
5th state:	1, 11, 12, 13, 14
6th state:	3, 11, 12, 13, 14, 4
7th state:	8, 11, 12, 13, 14, 7, 4.

When trying to perform the eighth step, the decoder would find path 8 on top of the stack and, since the latter has length 3 (= Γ), decoding would terminate.

It remains to generalize our algorithm to d-nary trees. The notion of a stack will facilitate a concise statement of the procedure.

(1) At the beginning of the decoding process, the stack contains only the root node of the tree (i.e., the empty path) with its likelihood set arbitrarily to zero.
(2) A decoding operation consists of ascertaining the path $\mathbf{s}^i$ that corresponds to the likelihood L_i at the top of the stack (i.e., the largest of the likelihoods in the stack), eliminating L_i from the stack, computing the likelihoods $\lambda_{i+1}^1, \cdots, \lambda_{i+1}^d$ of the branches that leave the end node of the path $\mathbf{s}^i$, and inserting the new path likelihoods $L_{i+1}^i = L_i + \lambda_{i+1}^i$, $i = 1, 2, \cdots, d$, into their proper positions according to size (clearly the stack contains path identifications and their likelihood values).
(3) The search ends when the decoder finds at the top of the stack a path whose length is Γ. That path is then considered to have been taken by the encoder.

In our algorithm, the likelihoods are used as a distance measure between the received sequence and the code word sequences on the various paths of the tree. The heuristic reasons for this choice of the measure were made clear in an IBM research report by the author[6] in which the stack algorithm was developed as an intuitively natural way of taking advantage of the tree structure of the code. This aspect makes the algorithm very attractive from a pedagogical point of view. The Fano method[3] can then be considered to be a particular implementation of the stack search in which the decoder's memory is eliminated for the price of an increase in the number of steps necessary for decoding (see Fig. 6). We will see in the next section that the stack algorithm lends itself to a relatively simple analysis.

4. Probabilty of error and average number of decoding steps

We now wish to compute upper bounds to the probability of decoding error and to the average number of decoding steps. Much of the argument is identical to that which applies to the Fano algorithm[4,5] and we will therefore limit ourselves to the differences between the two analyses and to a statement of results. A complete treatment can be found in our earlier IBM research report.[6]

It will be convenient to partition the nodes of the tree into sets $\mathfrak{D}_i$, $i = 0, 1, \cdots, \Gamma$, defined relative to the path $\mathbf{s}^\Gamma$ actually taken by the encoder.

Definition 1: The incorrect subset $\mathfrak{D}_i$ consists of the end node of the initial segment $\mathbf{s}^i$ of the true path $\mathbf{s}^\Gamma = (\mathbf{s}^i, s_{i+1}, \cdots, s_\Gamma)$, of the d–1 terminal nodes of the incorrect branches $s_{i+1}^* \neq s_{i+1}$ stemming from the end node of $\mathbf{s}^i$, and of all nodes lying on paths leaving these $d-1$ nodes.

A binary tree illustration of the sets $\mathcal{D}_i$ is given in Fig. 4. Let N_i be the number of nodes belonging to $\mathcal{D}_i$ that the decoder "visits." (These are end nodes of paths that have at one time appeared on top of the stack and were thus extended during the decoding process.) Then since $\bigcup_{i=0}^{\Gamma} \mathcal{D}_i$ is the set of all nodes in the tree, $\sum_{i=0}^{\Gamma} N_i$ is equal to the total number of decoding steps and we will want to estimate it.

Obviously, a path $\mathbf{s}_*^j \in \mathcal{D}_i (i < j \leq \Gamma)$ will not be extended in the decoding process unless its likelihood $L(\mathbf{s}^j)$ is such that the path $\mathbf{s}_*^j$ appears on top of the stack. But then $L(\mathbf{s}_*^j)$ must exceed the minimum of the likelihoods $\{L(\mathbf{s}^{i+1}), \cdots, L(\mathbf{s}^{\Gamma})\}$ along the true path (this condition is necessary but not sufficient). Let $\varphi(A)$ denote the *indicator function* of the event A, i.e.,

$$\varphi(A) \equiv \begin{cases} 1 \text{ if event } A \text{ takes place;} \\ 0 \text{ if event } A \text{ does not take place.} \end{cases} \tag{5}$$

Then

$$\begin{aligned} N_i &\leq 1 + \sum_{j=i+1}^{\Gamma} \sum_{\mathbf{s}_* \in \mathcal{D}_i} \varphi[L(\mathbf{s}_*^j) \geq \min L(\mathbf{s}^m)] \\ &\leq 1 + \sum_{j=i+1}^{\Gamma} \sum_{\mathbf{s}_* \in \mathcal{D}_i} \sum_{m=i+1}^{\Gamma} \varphi[L(\mathbf{s}_*^j) \geq L(\mathbf{s}^m)], \end{aligned} \tag{6}$$

where the numeral 1 accounts for the visit to the true node of $\mathcal{D}_i$.

Bounds on the γth $[\gamma \in (0, \infty)]$ decoding effort moments $\mathbf{E}[N_i^{\gamma}]$ based on the inequality (6) can be found in Ref. 5. It turns out that $\mathbf{E}[N_i^{\gamma}]$ can have a constant that is independent of Γ as an upper bound provided the coding rate R satisfies the inequality

$$n_0 R < \frac{1}{\gamma} \max_{\mathbf{r}} E_0(\gamma, \mathbf{r}), \tag{7}$$

where

$$E_0(\gamma, \mathbf{r}) \equiv -\log_2 \sum_y \left[\sum_x w(y/x)^{1/(1+\gamma)} r(x)\right]^{1+\gamma} \tag{8}$$

and the maximization is over all probability distributions $r(x)$ of the product channel input alphabet. The maximizing distribution $r^*(x)$ must then be the one used in the definition (4) of the likelihood measure $\lambda_i(\mathbf{s}^i)$. It should be stressed that the above result will hold for "good" codes whose probability of error has the behavior predicted below.

A decoding error will occur if there appears on top of the stack a path $\mathbf{s}_*^{\Gamma}$ corresponding to any of the Γ-level (incorrect) nodes of any of the sets $\mathcal{D}_0, \cdots, \mathcal{D}_{\Gamma-1}$ before the true path $\mathbf{s}^{\Gamma}$ appears. If $\mathbf{s}_*^{\Gamma} \in \mathcal{D}_i$, we say that an error took place on level $i+1$, since by Definition 1 the initial segment $\mathbf{s}^i$ of $\mathbf{s}^{\Gamma}$ is also the initial segment of $\mathbf{s}_*^{\Gamma}$, while the $(i+1)$th digit s_{i+1}^* of $\mathbf{s}_*^{\Gamma}$ differs from the corresponding digit of the true path. A necessary (but not a sufficient!) condition for an error on level $i+1$ is that the likelihood $L(\mathbf{s}_*^{\Gamma})$ [defined in (3)] be greater than or equal to the minimum value of the likelihoods $\{L(\mathbf{s}^{i+1}), \cdots, L(\mathbf{s}^{\Gamma})\}$ corresponding to the initial segments of the true path.

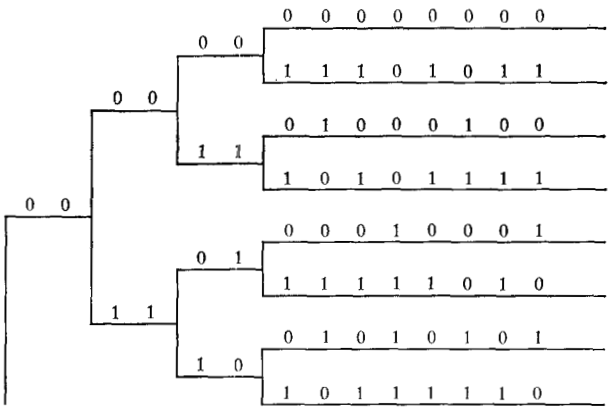


Figure 5 Portion of the last levels of a tree whose last branches have length $(t+1) = 4$ times that of regular branches.

Thus the probability $P_e(i)$ of an $(i+1)$-level error has a bound given by the inequality

$$\begin{aligned} P_e(i) &\leq \Pr\left\{\bigcup_{\mathbf{s}_* \Gamma \in \mathcal{D}_i} [L(\mathbf{s}_*^{\Gamma}) \geq \min_{i < m \leq \Gamma} L(\mathbf{s}^m)]\right\} \\ &= \Pr\left\{\bigcup_{\mathbf{s}_* \Gamma \in \mathcal{D}_i} \left[\bigcup_{m=i+1}^{\Gamma} \{L(\mathbf{s}_*^{\Gamma}) \geq L(\mathbf{s}^m)\}\right]\right\} \\ &\leq \sum_{m=i+1}^{\Gamma} \Pr\left\{\bigcup_{\mathbf{s}_* \Gamma \in \mathcal{D}_i} [L(\mathbf{s}_*^{\Gamma}) \geq L(\mathbf{s}^m)]\right\} \end{aligned} \tag{9}$$

and the probability of error P_e has the bound given by

$$P_e \leq \sum_{i=0}^{\Gamma-1} P_e(i). \tag{10}$$

It is obvious that if the tree remains regular up to the last level (i.e., always has d branches leaving each node with one symbol x corresponding to each branch), then the probability $P_e(i)$ will be a monotonically increasing function of i and, in fact, $P_e(\Gamma - 1)$ will be prohibitively large since it is equal to the probability that the branch likelihood of the last correct branch is less than the maximal likelihood of the $(d-1)$ incorrect branches' stemming from $\mathbf{s}^{\Gamma-1}$. A simple expedient to assure that $P_e(i)$ remains acceptably small even for large i (close to $\Gamma - 1$) is to associate with the last level branches not one but rather $t+1$ channel input symbols x, where t is chosen suitably large (this is illustrated in Fig. 5 with $t = 3$). The net transmission rate will thus be reduced from R to $R\Gamma/(\Gamma + t)$, which is negligibly different if $\Gamma \gg t$.

Upper bounds on P_e based on inequalities (9) and (10) can be found in Chapter 10 of Ref. 4. Let

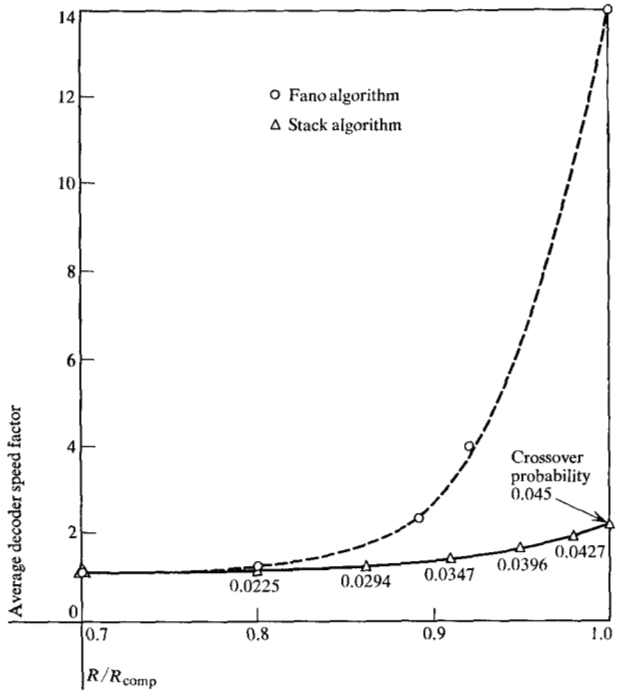


Figure 6 Average number of decoding steps necessary to decode a tree branch using the stack algorithm (solid line) and the Fano algorithm (interrupted line) as a function of the R/R_{comp} ratio. Coding rate is held constant at $R = ½$ but the crossover probabilities of the binary symmetric channel vary as indicated. Stack algorithm results are based on 1000 runs of binary information blocks of length 1000.

$$R_{comp} \equiv \frac{1}{n_0} \max_{\mathbf{r}} E_0(1, \mathbf{r}) \tag{11}$$

be the rate below which the average number of decoding steps $\mathbf{E}[N_i]$ is bounded by a constant. Then it can be shown that, for rates $R \leq R_{comp}$, there exist codes such that

$$P_e \leq K(R)2^{-tR_{comp}}, \tag{12}$$

where $K(R)$ is an increasing finite function of R. Since it can be shown that, for $R > R_{comp}$, $\mathbf{E}[N_i]$ is an exponentially increasing function of Γ (if we assume a good code), then for large Γ [which is needed to keep the rate loss factor $t/(\Gamma + t)$ small] one would ordinarily not attempt to use sequential decoding at rates that exceed R_{comp}. (The ratio of R_{comp} to channel capacity C is plotted in Fig. 1 for binary symmetric channels.)

We close this section by remarking that the argument that leads to bounds (6) and (9) for the Fano algorithm is a very elaborate one. It is not an exaggeration to state that the vast majority of students of graduate level information theory courses always remain uncomfortable with it.

5. Experimental evaluation and comparison

We have not yet fully evaluated the performance of the stack algorithm, but we can present some results of computer simulation and compare these with the published performance of the Fano sequential decoding algorithm.

We have run simulations of the performance of a rate $R = 1/2$ code when used over a binary symmetric channel whose crossover probability p is allowed to vary. In our experiment the block length $\Gamma = 1000$ and the length of the last-level branches was $t = 26$ symbols. The results of Fig. 6 are based on 1000 such blocks run for each of the different values of p. We have run the decoding as described in Section 3 with two branches leaving each node. The solid line in Fig. 6 indicates the average number of stack algorithm decoding steps necessary to decode one tree branch. The interrupted line plots the same quantity for the Fano decoding algorithm (the data is taken from Fig. 6.50 of the text by Wozencraft and Jacobs[8]). It is seen that at R_{comp} the stack algorithm has a better than sixfold advantage.

It must be stressed that Fig. 6 does not present the entire comparison and is actually unfavorable to the stack decoding algorithm: When decoding in real time with a fixed maximum delay imposed on the release of decoded information to the user (relative to the time of reception), it is the peak demands on computation that should be compared. Since the advantage of the stack algorithm grows substantially as the rate approaches R_{comp}, it is expected that this advantage will be even more pronounced during periods of high channel noise and consequent high computational demand.

Because of the above considerations the decoder *speed factor* (maximum number of decoding steps performable in the time it takes to receive one branch) must exceed substantially the averages plotted in Fig. 6. This is especially true for the Fano decoder that will be idle during the low noise time intervals when it is "caught up" with the received signal. (In such situations it can perform only one step per received branch interval, regardless of what its speed factor is!) The stack decoder need *never* be idle. If the length of the path on the top of the stack is equal to the length of the received sequence, the decoder may profitably extend the highest situated path of those paths in the stack that are shorter than the top one. This apparently premature work costs nothing and will not have to be done later should the channel noise increase.

The complete mutual independence of the stack ordering and path extending portions of the algorithm is also worthy of note. These two functions can be performed in parallel by different (but communicating) machines. The stack need not even be in order—only the top path must be available for extension. To achieve further speed-

up at the cost of greater decoder complexity, one might conceivably use several path extending machines simultaneously, the first one working on the top path in the stack, the second one on the second path, etc.

As seen from Fig. 6, the speed advantage of the stack algorithm grows as the coding rate is increased. This makes the former especially suitable for hybrid decoding[9-11] where the rate exceeds R_{comp}. A single stack may be time-shared among the m different but algebraically constrained information streams so that its cost is only a fraction of that of the total system. The problem of stack overflow (see Section 7 below) is also not crucial. Simulation for the scheme of Ref. 11 based on a stack of size 1000 was carried out with excellent results.

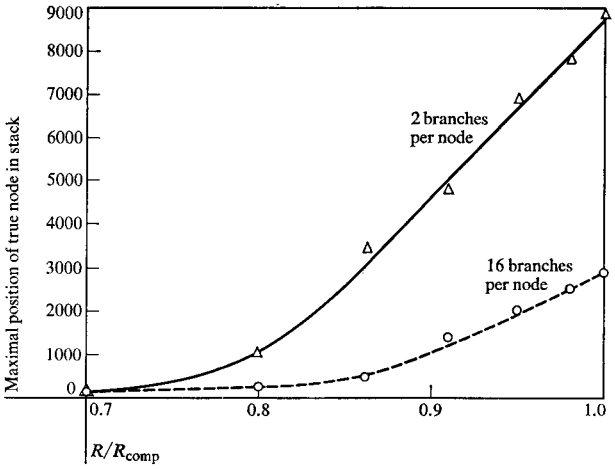


Figure 7 Plot of the maximum position of the true node in the stack obtained from 1000 runs of binary information blocks of length 1000. Solid line applies when the tree has two branches leaving each node and the stack insertion procedure is that described in Section 3. Interrupted line applies when the tree has sixteen branches leaving each node and the insertion procedure is that of Section 5.

6. Limiting the growth of the stack

As described at the end of Section 3, every decoding step of the tree search algorithm would involve the replacement of the top likelihood in the stack by d new likelihoods. This means a net growth of the stack size by $(d - 1)$ entries per decoding step. This unwanted dependence on d is entirely unnecessary, as follows from the observation (due to J. Cocke) that the path corresponding to the $(j + 1)$th branch's (in order of branch likelihood value) leaving a particular node can reach the top of the stack only after the path corresponding to the jth branch does. Hence we can modify the algorithm to limit the stack growth to at most one entry per decoding step regardless of the size of d:

(1) With each path likelihood $L(\mathbf{s}^i)$ also store the order j (by size) of the likelihood $\lambda^i = \lambda_i(\mathbf{s}^i)$ [see Eq. (4)] of the last branch s_i of the path [where $L(\mathbf{s}^i) = L(\mathbf{s}^{i-1}) + \lambda^j$, $\mathbf{s}^i = (\mathbf{s}^{i-1}, s_i)$ and $\lambda^1 \geq \lambda^2 \geq \cdots \geq \lambda^d$].
(2) If $L(\mathbf{s}^i)$ is found at the top of the stack, replace it by the likelihood $L(\mathbf{s}^{i+1})$, where s_{i+1} has the largest likelihood of all branches leaving $\mathbf{s}^i$. If the order j of the likelihood of the branch s_i is less than d, also insert into the stack the likelihood $L(\mathbf{s}_*^i) = L(\mathbf{s}^{i-1}) + \lambda^{j+1}$ corresponding to the path through the $(j + 1)$-order branch leaving the terminal node of $\mathbf{s}^{i-1}$. If $j = d$, do not insert any additional path.

With the above modification it becomes advantageous (in terms of decoding speed and stack size economy) to make the number of branches leaving a node as large as possible, provided their ordering in terms of likelihood value can be accomplished by a table look-up rather than by a direct computation followed by a comparison. If the tree code used has a convolutional structure (see Section 10.12 of Ref. 4), it is possible to construct such tables and the size of d is determined by the available storage capacity.

As an example, we give the state of the stack during the decoding of the tree of Figure 3. Paths are associated with their terminal nodes; the topmost path is stated first:

1st state: 0
2nd state: 2
3rd state: 5, 1
4th state: 6, 1, 11
5th state: 1, 11, 13
6th state: 3, 11, 13
7th state: 8, 11, 13, 4.

Comparing this with the stack development at the end of Section 3, we see that indeed the necessary stack size has been shortened.

We have tried to find out experimentally the savings in stack size obtainable from the present modification of the stack algorithm. Again, the block length $\Gamma = 1000$, $t = 26$, $R = 1/2$, and the channel is binary symmetric with varying p. Figure 7 plots the maximum position of the true path in the stack over the 1000 blocks decoded (i.e., if the stack used were longer than the number given, the true path would never have been eliminated from it). The solid line refers to the original stack algorithm of Section 3 used with a tree in which two branches leave every node. The interrupted line corresponds to the modification of this section applied to a tree with sixteen branches leaving a node. The convolutional codes used were identical. It is clear that, as expected, the second set up does allow for substantial economies in needed stack depth.

While running this experiment we found unexpectedly that the average number of decoding steps per transmitted

branch (not bit!) had virtually the same value for both versions of the stack algorithm at all values of the crossover probability p.

7. Handling of stack overflows

Obviously, any decoder, no matter how expensive, will have only a limited stack available for storing the likelihoods. When all of its locations are filled, the one containing the smallest likelihood must be purged to make room for a newly inserted path. The worry is that the purged entry may correspond to the true path, in which case a decoding error will necessarily result unless other precautions are taken. One possibility, described in some detail below, is to switch temporarily into a Fano decoding[3] mode. In this way catastrophy can be avoided when stack overflow takes place and even relatively short stacks capable of storing only several hundred paths can be used to speed up the decoding process. To keep the following discussion brief, it will be assumed that the reader is thoroughly familiar with some version of the Fano decoding algorithm (e.g., the one presented in Section 10.4 of Ref. 4).

Let T be the integral part of the largest path likelihood that was ever purged from the stack. Obviously, T is a monotonically nondecreasing function of time which might as well be defined as being equal to $-\infty$ until the first path is purged from the stack. Let us never insert a path into the stack whose likelihood is less than the current value of T. *Stack overflow* will be said to take place when the decoder finds the stack empty and is therefore unable to continue to carry out the algorithm. Let $\mathbf{s}^i$ be the last path extended before stack overflow took place. Then obviously $L(\mathbf{s}^i) \geq T$ and $L(\mathbf{s}^i, s_{i+1}) < T$ for all branches s_{i+1} leaving the node $\mathbf{s}^i$. First, the decoder backs up to the nearest preceding node $\mathbf{s}^j[j \leq i, \mathbf{s}^i = (\mathbf{s}^j, s_{j+1}, \cdots, s_i)]$ whose immediate predecessor node $\mathbf{s}^{j-1}$ has likelihood $L(\mathbf{s}^{j-1}) < T$. Next, the decoder is switched into the Fano forward mode and its cumulative threshold T_0 is set equal to $T - \tau$, where τ is the Fano decoder threshold quantum. Decoding continues in the Fano mode with T_0 varying as needed until such time as the decoder arrives for the first time at some node $\mathbf{s}^k$ at which the current value of T_0 is to be raised (until this time the variation in T_0 was downward only). At this moment $\mathbf{s}^k$ is inserted into the stack, T is set equal to $T_0 + \tau$ [note that $L(\mathbf{s}^k) \geq T_0 + \tau$] and decoder operation resumes in the stack mode. Obviously, if s^k is not the true path, with high probability all of the paths leaving it will have likelihood less than the new value of T. In such a case the stack will again be found empty and the decoder will switch back to the Fano mode as described above. It is important to note that this switchback will occur after fewer decoding steps have been completed than would have been necessary for the Fano decoder to lower its threshold from $T_0 + \tau$. Thus our strategy wastes no steps by a possibly premature switch into the stack decoding mode.

The flow chart of the complete stack-Fano algorithm is shown in Fig. 8. The notation of Section 10.4 of Ref. 4 is used.

8. Some remarks on implementation

In this section we will discuss the problem of maintaining the stack in proper likelihood order and the problem of specifying the tree paths entered into the stack. We shall describe the way our implementation has been carried out.

Although we found the *concept* of a stack useful for describing the algorithm, it turned out to be preferable to arrange the decoder's memory as a random access storage. In fact, a physical stack would necessitate a sequential comparison of its entries with the likelihood value of the path to be inserted, followed by a large relocation of data to make space for the new entry at the appropriate stack position. The amount of work connected with an insertion would vary as a function of stack content.

Instead, we found it advantageous to establish *equivalence classes* for likelihoods (e.g., all likelihoods with the same integral part belong to the same class) and to provide corresponding *class buckets* for insertion of new entries. The buckets are then ordered and a decoding step consists of selecting an arbitrary path of the top bucket, computing the likelihoods of its d extensions, computing the class membership of these likelihoods, and inserting the corresponding paths into the appropriate buckets. To be more specific, we will describe the system used by the author during simulation.

There are two sets of reserved storage locations, the *auxiliary stack* and the *stack* itself, referred to by indices $l \in \{-K, -k+1, \cdots, 0, 1, \cdots, J\}$ and $g \in \{1, 2, \cdots, M\}$, where K and J are chosen so that the likelihood values of paths visited during the decoding process lie between $-K$ and J with sufficiently high probability. M is the size of the stack, each entry of which consists of three parameters: $S(g)$—the path specification, $L(g)$—the likelihood value, and $P(g)$—a pointer to be described. The stack is filled sequentially as follows: The first path is put into location 1 (this path is then on top of the stack). Whenever a path is extended, its parameters are replaced by the corresponding parameters of the extended path. If an additional entry into the stack is made it is placed into the first unfilled location, if there is one. (When the stack is full, the bottom-of-the-stack path is replaced, as discussed below.) Let $G(l)$ be the entry at location l of the auxiliary stack. Its value is equal to the stack address of the last entered path whose likelihood has an integral part equal to l (if no such path exists in the stack, $G(l) = 0$). The pointer $P(g)$ is equal to the

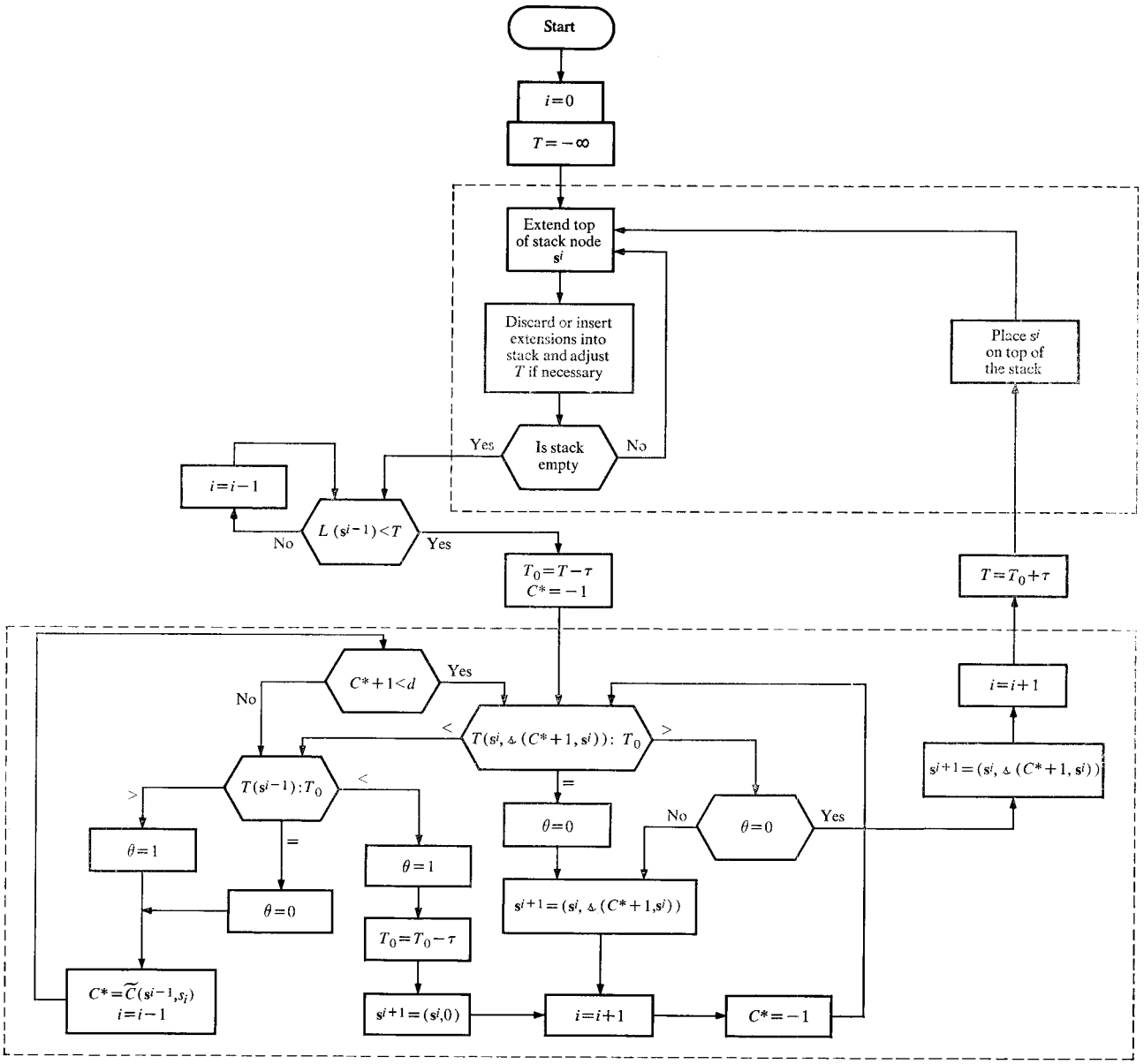


Figure 8 Flow chart of the combined stack-Fano decoding algorithm; notation as in Ref. 4.

stack address of the previously entered path whose likelihood has an integral part equal to the integral part of $L(g)$ [if no such path exists, $P(g) = 0$].

The appropriate parameter values are maintained in the two storages as follows: Suppose the path at location g of the stack is to be extended (and therefore eliminated from the stack) and suppose l equals the integral part of the likelihood $L(g)$. Then the entry $G(l)$ of the auxiliary stack will be set equal to the pointer $P(g)$. Next, suppose the path S' of likelihood L' is to be inserted into location g' of the stack. If l' is the integral part of L', then $P(g')$ is set equal to the current entry $G(l')$ of the auxiliary stack, whereupon $G(l')$ is reset equal to g'. Finally, $L(g')$ is set equal to L' and $S(g')$ to S'. Table 1 illustrates the stack maintenance procedure for the unmodified algorithm of Section 3 in the case of the likelihood situation of Fig. 3.

The auxiliary stack entries $G(l)$ are used to find the top-of-the-stack path to be extended and the bottom-of-the-stack path to be eliminated when a new entry is to be made into a full stack. The address of the top of the stack is $G(l^*)$, where l^* is such that $G(l) = 0$ for $l = l^* + 1, \cdots, J$ and $G(l^*) \neq 0$. When the stack is full, the new entry is put into location $g^+ = G(l^*)$, where $G(l) = 0$ for $l = -K, \cdots, l^+ - 1$ and $G(l^+) \neq 0$. Before this is done, $G(l^+)$ is set equal to $P(g^+)$.

Table 1 The contents of the stack and the auxiliary stack during the search of the tree of Fig. 3.

Step 1

b	S	L	P	l	g
1	0	−6	0		
2	1	1	0		
				1	2
				0	0
				−6	1

Step 2

b	S	L	P	l	g
1	0	−6	0		
2	10	−3	0		
3	11	−3	2	−3	3
				−6	1

Step 4

b	S	L	P	l	g
1	0	−6	0		
2	100	−7	4		
3	110	−7	0		
4	111	−7	3		
5	101	−7	2	−5	0
				−6	1
				−7	5
				−8	0

Step 6

b	S	L	P	l	g
1	000	−11	0		
2	100	−7	4		
3	110	−7	0		
4	111	−7	3	−4	7
5	101	−7	2		
6	01	−12	0		
7	001	−4	0	−7	5
				−11	1
				−12	6

Next, let us consider how a path is to be specified in the stack. If the tree code is convolutional (see Section 10.12 of Ref. 4), then at least the stack parameters $S(b)$ must characterize the tree depth i of the path and the sequence of ν last message digits $s_{i-\nu+1}, \cdots, s_i$, where ν is greater than or equal to the constraint length of the code. If ν is less than $\Gamma + t$, the block length of the code, then a way must be found to determine the decoded path. It would be natural to release to the user the digit $s_{i-\nu+1}$ whenever the tree depth i of the path at the top of the stack exceeds the depth of all the previously extended paths. In order to keep the probability of a wrongly released digit small, ν might have to be three times as large as the constraint length. This would make the stack storage unacceptably large and so a more subtle method of path specification must be devised.

A possible solution is to maintain in the storage a *map* of the paths contained in the stack. The construction of the map is based on the tree structure of the code and on the observation that the decoder extends a node at most once. (This is not strictly true of the modified algorithm of Section 6, but a simple adjustment of our method will take care of that problem.) The idea is simply to store, for each node of an investigated path, the digit $s \in \{0, 1, \cdots, d-1\}$ corresponding to the last branch and a backward pointer p to the preceding digit-pointer combination. When a path is extended from a node, the d newly created pointers will point to the digit-pointer combination of the former (again, we refer to the unmodified algorithm of Section 3).

The following more specific description will generalize somewhat the preceding notion.

An integer k is chosen and map storage space is allocated in which entries $V^*(j)$ and $Q^*(j)$ at location j represent a sequence of k message digits and a pointer, respectively. The stack parameter $S(g)$ itself consists of three parts: $V(g)$, capable of representing $k-1$ message digits; $I(g)$, representing the path depth; and $Q(g)$, a pointer. At the beginning $V^*(j) = Q^*(j) = V(g) = Q(g) = I(g) = 0$ for all j and g. As long as the path depth $I(g) < k$, then $Q(g) = 0$ and $V(g) = s_1, \cdots, s_{I(g)}$, the message sequence. As soon as a path $\mathbf{s}^{k-1}$ is to be extended to $(\mathbf{s}^{k-1}, s_k)$ for the first time, we set $V^*(1) = (\mathbf{s}^{k-1}, s_k)$, $V(g) = 0$, $I(g) = k$, and the pointer $Q(g)$ is set equal to 1, thus pointing to the first map location. In general, whenever path extension occurs from a path $\mathbf{s}^{i-1}$ to $\mathbf{s}^i$, where i is not a multiple of k, neither the pointer $Q(g)$ nor $Q^*(j)$ is changed, no new entry in the map is made, $I(g)$ is set to i, and s_i is added to the content of $V(g)$. Suppose that i is a multiple of k and that j is the first free location in the map. Then the pointer $Q^*(j)$ is set equal to the old pointer value $Q(g)$, $V^*(j)$ is set equal to $s_{i-k+1}, \cdots, s_i$, $Q(g)$ is made to point to the map location j, and we make $I(g) = i$ and $V(g) = 0$. It is obvious that in this way the entire path corresponding to any stack entry g is kept available in the storage. In fact, the path digits $s_i, s_{i-1}, \cdots, s_1$ are represented in the registers $V(g), V^*(j_1), \cdots, V^*(j_t)$, where $j_1 = Q(g)$, $j_2 = Q^*(j_1), \cdots, j_t = Q^*(j_{t-1})$, and $Q^*(j_t) = 0$.

If it is desired to purge the map of entries made unnecessary by path purges of the stack, this may be accomplished best by adding another register $C^*(j)$ at the various map locations. $C^*(j)$ will at all times be equal to the number v ($\leq d^k$) of pointers pointing to the map location j. Suppose the entry g is to be purged from the stack and $j_1 = Q(g)$. Then the value of $C^*(j_1)$ is lowered by 1. If the new value is not equal to 0 nothing further is done, if it is equal to 0 and $j_2 = Q^*(j_1)$ then $C^*(j_2)$ is lowered by 1 and the j_1 map location is made available to a pool for new refilling. The new value of $C^*(j_2)$ is similarly compared with 0 and, if it is 0, then $C^*(j_3)$ [$j_3 = Q^*(j_2)$] is in turn lowered by 1 and the j_2 map location becomes available, etc.

Table 2 The path identification map for stack size greater than 6 and $k = 1$.

Step 1						Step 2					
Node	*I*	*Q*	*V**	*Q**	*C**	*Node*	*I*	*Q*	*V**	*Q**	*C**
1	1	1	0	0	1	1	1	1	0	0	1
2	1	2	1	0	1	5	2	3	1	0	2
						6	2	4	0	2	1
									1	2	1
Step 4						**Step 6**					
Node	*I*	*Q*	*V**	*Q**	*C**	*Node*	*I*	*Q*	*V**	*Q**	*C**
1	1	1	0	0	1	7	3	11	0	0	3
11	3	5	1	0	4	11	3	5	1	0	4
13	3	7	0	2	2	13	3	7	0	2	2
12	3	6	1	2	2	12	3	6	1	2	2
14	3	8	0	3	1	14	3	8	0	3	1
			1	3	1	4	2	10	1	3	1
			0	4	1	8	3	12	0	4	1
			1	4	1				1	4	1
									0	1	2
									1	1	1
									0	9	1
									1	9	1

Table 3 The path identification map for stack size equal to 3 and $k = 1$. Steps 1 and 2 are the same as in Table 2.

Step 3						Step 4					
Node	*I*	*Q*	*V**	*Q**	*C**	*Node*	*I*	*Q*	*V**	*Q**	*C**
1	1	1	0	0	1	1	1	1	0	0	1
11	3	5	1	0	2	11	3	5	1	0	2
16	2	4	0	2	1	13	3	6	0	2	1
			0	3	1				1	2	1
									0	3	1
									0	4	1
Step 5						**Step 6**					
Node	*I*	*Q*	*V**	*Q**	*C**	*Node*	*I*	*Q*	*V**	*Q**	*C**
3	2	7	0	0	2	8	3	8	0	0	1
11	3	5	1	0	1	11	3	5	1	0	2
13	3	6	0	2	1	13	3	6	0	2	1
			1	2	1				1	2	1
			0	3	1				0	3	1
			0	4	1				0	4	1
			0	1	1				0	1	1
									1	7	1

Table 2 illustrates the map structure for the case of the likelihood situation of Fig. 3 if the stack size is greater than or equal to 7 and $k \geq 1$ (thus the V entries are absent). Table 3 does the same for the case in which the stack size is limited to 3 entries.

Acknowledgment

The author thanks John Cocke for his inspiration and many valuable discussions and suggestions.

References

1. J. Ziv, "Successive Decoding Scheme for Memoryless Channels," *IEEE Trans. Information Theory* **IT-9,** 97 (1963).
2. J. M. Wozencraft, *Sequential Decoding for Reliable Communication, MIT-RLE Tech. Rep.* TR325, 1957.
3. R. M. Fano, "A Heuristic Introduction to Probabilistic Decoding," *IEEE Trans. Information Theory* **IT-9,** 64 (1963).
4. F. Jelinek, *Probabilistic Information Theory,* McGraw-Hill Book Co., Inc., New York 1968.
5. F. Jelinek, "An Upper Bound on Moments of Sequential Decoding Efforts," *IEEE Trans. Information Theory* **IT-15,** 140 (1969).
6. F. Jelinek, "A Stack Algorithm for Faster Sequential Decoding of Transmitted Information," IBM Research Report RC 2441, April 15, 1969.
7. I. M. Jacobs and E. Berlekamp, "A Lower Bound to the Distribution of Computation for Sequential Decoding," *IEEE Trans. Information Theory* **IT-13,** 167 (1967).
8. J. M. Wozencraft and I. M. Jacobs, *Principles of Communication Engineering,* John Wiley & Sons, Inc., New York 1965.
9. D. Falconer, "A Hybrid Sequential and Algebraic Decoding Scheme," Sc.D. thesis, Dept. of Elec. Eng., M.I.T., 1966.
10. F. L. Huband and F. Jelinek, "Practical Sequential Decoding and a Simple Hybrid Scheme," Intl. Conf. on System Sciences, Hawaii, January 1968.
11. F. Jelinek and J. Cocke, "Adaptive Hybrid Sequential Decoding," submitted to *Information and Control.*

Received April 10, 1969