A COMPARISON OF THE FANO AND JELINEK SEQUENTIAL DECODING ALGORITHMS

by John M. Geist

Technical Report No. EE-701

January 31, 1970

Department of
ELECTRICAL ENGINEERING

UNIVERSITY OF NOTRE DAME, NOTRE DAME, INDIANA

A COMPARISON OF THE FANO AND JELINEK SEQUENTIAL DECODING ALGORITHMS

by John M. Geist

Technical Report No. EE-701

January 31, 1970

Department of Electrical Engineering

University of Notre Dame, Notre Dame, Indiana 46556

This work was supported by the National Aeronautics and Space Administration under NASA Grant NGL 15-004-026 in liaison with the Flight Data Systems Branch of the Goddard Space Flight Center.

Introduction

The purpose of this report is to present the results of a comparison by simulation of the performances of the Fano[1] sequential decoding algorithm and an alternative algorithm recently proposed by Jelinek[2]. In section I we give a description of the Jelinek algorithm (the reader's familiarity with the Fano algorithm, in the form of Figure 1, is assumed), and compare some search properties of the two algorithms. In section II the simulation results are presented and discussed. Some details of the Jelinek decoder program are given in an appendix.

I. The Jelinek Algorithm

We will restrict consideration to binary convolutional codes of rate 1/N (although Jelinek gives a simple extension of the algorithm to arbitrary rates) and to transmitted frames of finite length L. The received sequence, together with the code tree, gives rise to a value tree in which each node s is assigned a value V(s) determined by comparing the code symbols along the path specified by node s with the corresponding segment of the received sequence. The values have the form

$$V(s) = \sum_{i=1}^{d(s)} \sum_{j=1}^{N} \left(\log \frac{\Pr\{r_i^{(j)}, x_i^{(j)}\}}{f(r_i^{(j)})} - B \right) \qquad (1)$$

where d(s) is the length of the path leading to node s and the first sum is

over all branches in that path, $x_i^{(j)}$ is the jth symbol on the ith branch of the path in the code tree specified by s, $r_i^{(j)}$ is the corresponding symbol in the received sequence, $f(\cdot)$ is the nominal received symbol probability function, and B is a bias chosen such that, on the average, node values along the correct path tend to increase with depth into the tree, while node values along incorrect paths tend to decrease sharply. Thus the decoding strategy is to look for a path whose values are increasing.

The decoder consists of a "stack" or ordered list of nodes, appearing in decreasing order of likelihood values. Thus the top node, or first node, is the node with greatest likelihood among the nodes in the stack. The stack is initially loaded with only the origin node, whose value is taken to be zero, and is processed according to the following rules:

(1) Compute the likelihood values of the two successors of the top node and add them to the stack in the places determined by their likelihoods.

(2) Delete the node whose successors were just added.

(3) If the new top node is in the last level of the tree, stop. Otherwise, go to (1).

The path selected by the decoder is the path specified by the node which is at the top of the stack when the algorithm halts. Taking a computation to be an execution of step (1), the number of computations required to decode a frame is one less than the size of the stack when the decoder halts.

It is clear that if we insist on exact ordering among the nodes in the stack, execution of step (1) can be quite time-consuming, since for each successor to be inserted, the decoder must scan through the stack until it encounters a node whose value does not exceed that of the node to be inserted. When the

stack size is large, this may require a great deal of searching and testing. To avoid this difficulty, Jelinek suggests an inexact ordering of node likelihoods. For some number $H > 0$, which we call the "bin spacing", a node s is assigned to bin k if

$$kH \leq V(s) < (k+1)H.$$

Thus a node can be assigned to a bin on the basis of its own likelihood alone, without comparing it to other node likelihoods. The algorithm is then modified as follows:

(1) Select any node from the highest non-vacant bin, compute the likelihoods of its successors, and insert them in the appropriate bins.

(2) Delete the node whose successors were just added.

(3) If any node in the highest non-vacant bin is in the last level of the tree, stop. Otherwise, go to (1).

We now describe the path selected by the Jelinek algorithm and by the modified version just mentioned. Let S_d be a node in level d of the value tree, and let $S_d, S_{d+1}, \ldots, S_{L-1}, S_L$ be the nodes along some path from S_d to the end of the tree. Define the minimum likelihood of this path to be $\min_{d \leq j \leq L} V(S_j)$

It can be shown that the path selected by the Jelinek algorithm consists of nodes $S_0, S_1, \ldots, S_L$ which satisfy the following conditions:

(1) S_0 is the origin node;

(2) For $i=0,1,\ldots,L-1$, the branch leading from S_i to S_{i+1} is the first branch of that path emanating from S_i which has greatest minimum likelihood.

For good codes, and in the absence of severe bursts of channel noise, only one path, namely the correct one, will satisfy these conditions. In the unlikely event that more than one path satisfies these conditions, which of the paths is selected depends on how ties are resolved in step (1) of the algorithm.

For the modified version, we replace each exact node value $V(s)$ by its approximation kH, where $kH \leq V(s) < (k+1)H$. The path chosen is a path through the approximate value tree satisfying the same conditions as before.

Massey and Sain[3] have similarly characterized the path chosen by the Fano algorithm. Let Δ be the threshold increment. If in the value tree we replace node values $V(s)$ by thresholds $k\Delta$, where $k\Delta \leq V(s) < (k+1)\Delta$, the path chosen is again one which satisfies conditions (1) and (2).

It is clear from these descriptions that if $H=\Delta$, then the modified Jelinek algorithm and the Fano algorithm select the same path, except perhaps in the cases when more than one path satisfies our conditions. Moreover, we may choose $H=\Delta$ sufficiently small that $kH \leq V(s_1) < (k+1)H$ and $kH \leq V(s_2) < (k+1)H$ if and only if $V(s_1) = V(s_2)$. For these choices of the parameters, all three algorithms will select the same path, again excepting the occurrences of multiple paths satisfying conditions (1) and (2).

More generally, we can describe the sets of nodes which are examined during decoding. Let $S_0^*, S_1^*, \ldots, S_L^*$ be the nodes in the path ultimately selected by the Jelinek algorithm, and for $0 \leq b < L$ define

$$Q_b^* = \min_{b \leq k \leq L} V(S_k^*)$$

Let S_d be any node in level d of the tree not on the chosen path, and let S_b^* be the deepest node shared by the correct path and the path to S_d.

Then the path to S_d is $S_0^*, S_1^*, \ldots, S_b^*, S_{b+1}, \ldots, S_d$. We may then state the following facts: S_d reaches the top of the stack if $V(S_k) > Q_b^*$ for all $k=b+1,\ldots,d$; S_d does not reach the top of the stack if for some k, $b+1 \leq k \leq d$, $V(S_k) < Q_b^*$. The remaining possibility, namely $V(S_k) \geq Q_b^*$, $k=b+1,\ldots,d$, with equality for at least one k, depends upon the method of resolving of ties in the stack. For the modified Jelinek algorithm, define for any node s, $\hat{V}(s)=kH$, where $kH \leq V(s) < (k+1)H$. Let $S_0^*, \ldots, S_L^*$ be the path ultimately selected and let

$$\hat{Q}_b^* = \min_{b \leq k \leq L} \hat{V}(S_k^*) \quad , \quad 0 \leq b < L.$$

Then for any node S_d specifying the path $S_0^*, \ldots, S_b^*, S_{b+1}, \ldots, d$, S_d reaches the top of the stack if $\hat{V}(S_k) > \hat{Q}_b^*$ for all $k=b+1,\ldots,d$, and does not if $\hat{V}(S_k) < Q_b^*$ for some such k. When we replace H by Δ and let $S_0^*, \ldots, S_L^*$ be the path selected by the Fano algorithm, the corresponding statement is: The decoder is positioned at S_d at least once if $\hat{V}(S_k) > \hat{Q}_b^*$ for all $k=b+1,\ldots,d$, and is never positioned at S_d if for some k, $\hat{V}(S_k) < \hat{Q}_b^*$.

Therefore, if we select $H=\Delta$ sufficiently small that $\hat{V}(s_1)=\hat{V}(s_2) \iff V(s_1)=V(s_2)$, the sets of nodes examined by the three algorithms coincide, except for the effect of ties among node values. (These are not optimum choices for H and Δ; empirical determination of the optimum values is discussed in section II.) For larger values of H and Δ, the sets of nodes examined still coincide except for ties among the adjusted node values $\hat{V}(s)$. Of course, such ties become more numerous as H and Δ increase. For values in the range of interest, however, the regions of the tree which are searched by the three algorithms coincide almost exactly in nearly all cases. Note that since the Fano decoder may move to a node several times, while a node can reach the top of the Jelinek stack

only once, the Fano decoder will usually perform many more computations than the Jelinek decoder, even though the number of distinct nodes involved in these operations are almost equal.

II. Results

Both versions of Jelinek's algorithm were programmed for the UNIVAC 1107 computer at the University of Notre Dame Computing Center. These programs were used in conjunction with existing programs for the Fano decoder and for simulating channel disturbances to study the performance of the various decoding systems. In this section, we review the results of this study.

All the results to be presented were obtained with a rate $\frac{1}{2}$ non-systematic convolutional code of memory 35 constructed by Costello[4]. This code has a number of important and useful properties, one of which is large free distance, which makes it well-suited to sequential decoding. In all the frames run to date, no decoding error has been made with this code. The generator polynomials (in octal form) are:

$$G^{(1)} = 533533676737$$

$$G^{(2)} = 733533676737$$

Runs were made on the binary symetric channel at three noise levels and on the binary-input Gaussian noise channel with eight-level output quantization (see Figure 2). For a discrete memoryless channel with K inputs and J outputs, define

$$E_0(\rho) = \max_{Q} \; -\log \sum_{j=1}^{J} \left(\sum_{k=1}^{K} Q(k)P(j/k)^{\frac{1}{1+\rho}} \right)^{1+\rho}$$

$$\rho > 0$$

and $$R_\rho = \frac{E_0(\rho)}{\rho}$$

The computation a sequential decoder must perform to decode an information digit (assuming infinite frame length) is a random variable, and R_ρ is the rate above which the ρth moment of this random variable fails to exist. When $\rho=1$ the symbol R_{comp} is customary. Some parameters of interest for the channels used are given below in Table 1. In this table, ρ is the solution of $R_\rho = R = \frac{1}{2}$.

Table 1

Channel	R_{comp}	R/R_{comp}	ρ
BSC. = .033	.5593	.89	1.354
BSC. = .045	.4996	1.00	.998
BSC. = .057	.4504	1.10	.729
Gaussian	.4913	1.02	.944

Branch values are computed by adding the values for the two digits on the branch. The digit values are determined by comparing the code symbols with the corresponding received symbol. The digit values used are listed in Table 2. These values correspond to the terms $\log \frac{Pr\{r|x\}}{f(r)} + B$ in equation (1), rounded to integer values.

Table 2.

Binary Symmetric Channel

		Received Symbol	
p	Code Symbol	0	1
.033	0	2	-18
	1	-18	2
.045	0	2	-16
	1	-16	2
.057	0	4	-35
	1	-35	4

Gaussian Channel

	Received Symbol							
Code Symbol	0	1	2	3	4	5	6	7
0	4	4	2	0	-8	-20	-34	-58
1	-58	-34	-20	-8	0	2	4	4

It was assumed throughout that the all-zero sequence was transmitted. This assumption entails no loss of generality, but it offers the decoders an unfair advantage if the 0 branch is preferred in case of ties. Therefore, both decoders were biased to choose the 1 branch in case of ties. Thus decoding is actually somewhat more difficult than we would expect for a random information sequence.

Computation Time for the Jelinek Algorithm. As pointed out in section I , the necessity for a stack search after each extension renders the Jelinek computation time unacceptable, especially since the time increases faster than linearly with the number of computations. Figure 3 exhibits the behavior of computing time with number of computations. Each point on the graph represents the average time and computation for several frames whose computation counts fall in a specified range. The points are labelled with the number of frames included in the average. These data were taken on the BSC with $p = .045$.

Optimum Bin Spacing and Threshold Increment. Extensive work with the Fano decoder* has indicated that the best choice for Δ on the BSC is a value equal to the magnitude of the branch value for a branch having a single discrepancy for $R = \frac{1}{2}$ binary codes. This is in agreement with an intuitive feeling that the decoder should be allowed to skip quickly over single errors and undertake extensive searches only in regions of severe noise. Single-error branches have values -16, -14, and -31 on the three BSC's, and the decoder program requires Δ to be a power of 2, so Δ of 16, 16 and 32 respectively, was used

* This work is beyond the scope of the present report, but some results are given by Costello[4].

in the runs of the BSC. For the Gaussian channel, "single errors" are not well-defined, but since digit metrics are roughly double those used on the first two BSC's, $\Delta = 32$.

In an effort to determine empirically the optimum bin spacing for the modified Jelinek algorithm, runs were made with H = 4 ,8,16, and 32 on the BSC with p = .033 and p = .045. (The Jelinek decoder requires H to be a power of 2 and to be at least as large as the value of a branch which agrees with the received sequence in both digits. See Appendix for details.) Table 3 gives average values of the number of computations in 100-frame samples at two noise levels. Figure 4 shows the distribution of computation for H = 4,8, and 16 at p = .045. In Figure 5 the distribution of computation is plotted for H = 8,16, and 32 on the Gaussian channel. On the basis of these results we conclude that H = 4 is the optimum choice for the first two BSC's and H = 8 for the third BSC and the Guassian channel. These values are used in the remaining runs.

Table 3

p	H	Average Comp.
.033	4	353.9
	8	354.3
	16	360.2
	32	411.8
.045	4	469.0
	8	471.1
	16	517.6
	32	607.3

Distribution of Computation for the Modified Jelinek Algorithm. It is a well-documented property of sequential decoding systems that the computation C required to decode a digit is a random variable whose distribution (asymptotically) has the form

$$\Pr\{C \geq N\} = K N^{-\rho}$$

where ρ is the solution of $R = R_\rho = E_0(\rho)/\rho$. In Figures 6 (BSC) and 7 (Gaussian) are plotted the observed distributions of the average computation per digit, that is, the total computation done in decoding a frame divided by the frame length. The distribution of this random variable $\overline{C}$ differs from that of C, but it can be shown that the tail of the distributions have the same form. Since Figures 6 and 7 are plotted on log-log scale, the curves will tend asymptotically to straight lines whose slopes are the negatives of the values of ρ given in Table 1. These asymptotes are displayed as dashed lines in Figures 6 and 7.

Comparison of Jelinek and Fano Algorithms. We will define a computation for the Fano algorithm to be an entry of either of the "Look Forward" boxes of Figure 1, and a Jelinek computation to be an execution of step (1) of the algorithm. As we remarked in section I, the Fano decoder always requires more computation to decode a frame than does the Jelinek decoder, so on that basis alone, the Jelinek decoder is superior. However, Jelinek computations are inherently more complicated than Fano computations, since with each node examined, the decoder must store enough information to determine the path back to the origin and to resume searching forward from that node if necessary. The Fano algorithm on the other hand, since it only moves from a node to an adjacent node, can easily maintain this

information as it proceeds through the tree. The question is, therefore, whether the additional complexity of the Jelinek computation is offset by the smaller number of computations required for decoding.

An added complication precludes a simple answer to this question. The number of "forward looks" the Fano decoder must perform grows faster than linearly with the number of nodes examined; that is, the ratio of Fano computations to Jelinek computations is not constant but grows as the number of Jelinek computation is increased, because of the effect of repeated searches by the Fano decoder. It is therefore possible that, for relatively quiet frames, the Fano decoder is superior, while for noisier frames the Jelinek decoder is superior. This is, in fact, the behavior which was observed in the simulations.

We choose as a measure of decoding effort, the time to decode a frame. Since the time required is roughly propotional to the number of computations, the distribution of computing time has the same shape as the distribution of computation. The behavior observed in the simulation is plotted in Figures 8 (BSC) and 9 (Gaussian).

The reader is cautioned against attributing too much significance to the position of the crossover point in Figures 8 and 9, since it depends strongly on the relative complexity of the two kinds of computations, and may be altered by differences in available hardware or programming technique What can be safely concluded is that the Jelinek decoder is at least competitive with the Fano decoder, and is faster, except on fairly quiet frames. The noisier the channel, the less likely are quiet frames, and hence the Jelinek decoder is more likely to be the faster of the two.

We can substantiate these observations in an alternative manner. Any practical sequential decoder must be time-limited; that is, a restriction must be put on the amount of time it may spend decoding a frame. If at the end of that time the decoder has not completed its search, the frame is considered an erasure. In general, the minimum amount of time that may be allowed depends on the erasive probability the user is willing to tolerate. For a given erasure probability, the average time required to decode a frame (including erased frames) can be determined from the distribution of computing time. These averages, as functions of erasive probability, are plotted in Figures 10A, 10B, and 10C for the three BSC's. We can conclude, for example, that if an erasive probability of 0.1 is required on the BSC with $p = .045$ ($R = R_{comp}$), then the Jelinek decoder is, on the average, about 25% faster than the Fano, and this advantage increases as we decrease the allowable erasive probability. In fact, to operate in the range where the Fano decoder is faster, we would have to settle for a 50% erasive rate! Thus, in the vicinity $R = R_{comp}$, the Jelinek algorithm offers considerable practical advantage. Note, however, that for the low noise case $p = .033$ ($R = .9R_{comp}$), the advantage goes to the Fano algorithm. In general, the noisier the channel, the more favorably does the Jelinek algorithm perform relative to the Fano algorithm.

Summary

The results reported here indicate that the unmodified Jelinek algorithm is of no practical value; however, it is of interest theoretically because of the insight it provides into the nature of sequential decoding, and because of its conceptual simplicity. When modified to avoid exact node ordering, the Jelinek algorithm is practicable, having approximately the same computational complexity as the usual Fano algorithm. Which of the two algorithms is faster depends strongly on the noise level of the channel, the Jelinek decoder being better on noisier channels, Fano better on quiet channels.

Acknowledgement

I am grateful to Professor James L. Massey for his encouragement and his many helpful suggestions during the course of this work.

I wish to acknowledge that the Fano decoder program used in the simulation was written by Daniel J. Costello, Jr., and the channel noise generator programs were written by K. Vairavan.

Finally, during the preparation of this report it came to my attention that an algorithm equivalent to the unmodified Jelinek algorithm has been developed independently by Dr. Kamil Zigangirov [5].

References

1. R. M. Fano, "A Heuristic Discussion of Probabilistic Decoding", IEEE Trans. on Information Theory, IT-9, pp 64-74, April 1963.

2. F. Jelinek, "A Fast Sequential Decoding Algorithm Utilizing a Stack", IBM J. Res. Div., Vol. 13, November 1969.

3. J. L. Massey & M. K. Sain, "Trunk and Tree Searching Properties of the Fano Sequential Decoding Algorithm", Proc. Sixth Allerton Conf., U. of Illinois, August 1968.

4. D. J. Costello, Jr., "Construction of Convolutional Codes for Sequential Decoding", Technical Report EE-692, Dept. of Electrical Engineering, University of Notre Dame, August 1969.

5. K. Zigangirov, "Some Sequential Decoding Procedures", Problemy Peredachi Informatsii, Vol. 2, No. 4, pp. 13-25, 1966.

Appendix

We now describe in some detail the operation of the modified Jelinek decoder as we have programmed it on the UNIVAC 1107, in the hope that our techniques may be of use to other investigators.

In order to extend from a node, three items must be known: the value of the node, its depth into the tree, and the encoder state at the node. The encoder state is necessary to produce the code symbols on the branches emanating from the node; the depth is needed to compare the code symbols with the proper span of symbols in the received sequence; and the branch values arising from this comparison must be added to the value of the extended node to yield the new node values. Therefore, for every node the decoder encounters, the value, depth, and encoder state must be saved so that the node can be extended, if necessary.

For the top node on the stack (i.e., one of the stack nodes in the highest non-empty bin) these items are kept in three accumulators which are given the symbolic labels of VALUE, DEPTH, and STATE. For other nodes on the stack, the information is contained in what we call node descriptions, which occupy six contiguous words of computer memory, in which are stored, respectively, the node's value, depth, encoder state, a stack pointer, a path pointer, and a flag. (The last three items will be discussed later.)

Initially the top node is the origin node, which is at depth zero, and the encoder state is zero. Thus DEPTH and STATE are initially set to zero. The value of the origin node is usually considered to be zero, but it is convenient to avoid the possibility of encountering nodes of negative

value. We therefore bias the node values by initializing VALUE to a sufficiently large number so that, with overwhelming probability, no node placed on the stack has negative value.

As the search through the tree proceeds, the nodes encountered are added to the stack, and the necessary information saved, either as the contents of the three accumulators or in node descriptions (or both). Once a node description is stored it is never physically deleted from memory, even if the node reaches the top bin and is extended. Therefore, there are many node descriptions resident in memory representing nodes which are no longer on the stack, by virtue of step (2) of the algorithm. On the other hand, every node on the stack (except possibly the top node) is represented by a node description. The determination of which of the resident node descriptions represent nodes still on the stack, and the ordering of the stack contents into bins, are the functions of the stack pointers and an array called the bin index. The bin index consists of two entries for each bin: the first is the number of nodes in the bin; the second is the address of the first word of the node description corresponding to one of the nodes in the bin. The stack pointer in this node description contains the address of the first word of the node description for another node in the bin, and so on. (The stack pointer in the description of the last node in the bin is meaningless.) Therefore, the contents of bin k can be found successively by using the second entry of the bin index corresponding to bin k and the stack pointers in the node descriptions referenced. In order to place an upper bound on the number of bins and hence on the size of the bin index array, a lower bound must be set on the bin spacing H. We

require H to be at least as great as the maximum positive branch value.

To see how the decoding proceeds, let us first restrict ourselves to the BSC and to the use of complementary codes, i.e., codes in which the code symbols on the 1 branch emanating from any node are complements* of the symbols on the 0 branch. Then when the top node is extended and the two branches compared to the received sequence, only two outcomes are possible: either (1) one branch agrees with the received sequence in both digits and the other branch disagrees in both, or (2) each branch disagrees in exactly one digit. We consider the two cases separately.

Case (1). We call this the typical case since almost every extension of a node on the correct path and roughly half the extensions of nodes not on the correct path are of this kind. Table 2 shows that the branch value for the branch with two agreements will be +4, so that the value for the corresponding node exceeds the value of the extended node. Then since the extended node was in the highest non-vacant bin, and since the successor along the +4 branch belongs in the same or a higher bin, we may take the successor to be the new top node and adjust VALUE, DEPTH, and STATE accordingly; there is no need to store its description. For the other successor, however, we must store a node description. The first three items in the description can be gotten easily from VALUE, DEPTH, and STATE and the branch value. To set the stack pointer, we note that a node of value V belongs to bin k if $kH \leq V < (k+1)H$. That is, the correct value of k is the integer part of V/H. We restrict H to be a power of 2, say $H = 2^r$, so that k can be found by a simple r-bit shift operation.

---

*An equivalent condition is that neither generator begin with a 0.

Having found the bin to which the node belongs, we use the bin index: increase the bin count by one, set the stack pointer of the new node description to the address presently specified in the bin index, and reset the bin index pointer to the address of the new node description. Thus after insertion, the bin index pointer points to the new description and the stack pointer of the new description points to the description which was previously referenced by the bin index pointer. After setting the path pointer and flag, which we discuss later, the decoder is ready to extend again using the updated contents of VALUE, DEPTH, and STATE.

Case (2). Since both branch values are negative, it is likely that there are nodes on the stack in bins higher than the bins to which the successors belong. Therefore the new top node is not readily available as it is in Case (1) - we must search for it. First the two new nodes are stored in the same way the one was stored in Case (1). Then the decoder scans down the bin index, starting with the bin to which the extended node belonged, looking for a bin whose count is non-zero. When the first non-empty bin is located, its count is decreased by one, VALUE, DEPTH, and STATE are loaded from the node description referenced by the bin index pointer, and the bin index pointer is reset to the address contained in the stack pointer of that node description. Thus the node which had been the second node in the bin is now referenced by the bin index. The decoder is now ready to extend again.

We turn now to the Gaussian channel, leaving in force the restriction to complementary codes*. It is no longer meaningful to use the terms "agreement" and "disagreement", but from Table 2 we see that there are still two cases: either (1) one branch value is non-negative and the other is negative, or (2) both are negative. It is clear that the same decoder actions described above are applicable on the Gaussian channel.

A transmitted frame as programmed consists of 256 branches corresponding to encoded information bits followed by a 35-branch tail corresponding to an encoded memory span of 0s, included to prevent high error probability in the last few information bits. The search in the tail differs from that in the information part of the tree in that only the successor along the 0 branch is considered. The two kinds of computations are still performed: if the branch value is non-negative, there is no storage and the successor is extended immediately; if the branch value is negative, the successor is stored and a search for a new top node undertaken.

If the contents of DEPTH is 291, this indicates that the top node is at the end of the tree and the search is completed. This brings up the problem of recovering the information symbols on the path chosen. If a description had been stored for every node encountered, then the path pointer in every node description could have been set to the address of the description of its predecessor. Then the path pointers would specify the path from the final node back to the origin. Since, typically, only one node is stored, this is impossible. However, at every extension at least one node is stored, and therefore for every node encountered, either its

*This is no sacrifice, since only complementary codes would be used in practice.

predecessor or the complement of its predecessor (or both) is stored. Thus we set the path pointer in each node description to the address of the description of the predecessor if it is stored, or, if it is not, to the address of the description of the complement of the predecessor, and we use the sixth element of the node description, the flag, to indicate whether the node referenced by the path pointer is the predecessor or its complement. Now when the top node is at the end of the tree, the decoder can step back toward the origin using the path pointers, flags, and encoder states to provide the information sequence along the path selected.

Figure 11 illustrates the pattern of extensions and storage by the the decoder program for a typical segment of a tree with the branch values for the BSC, p = .033 The extensions are numbered sequentially.

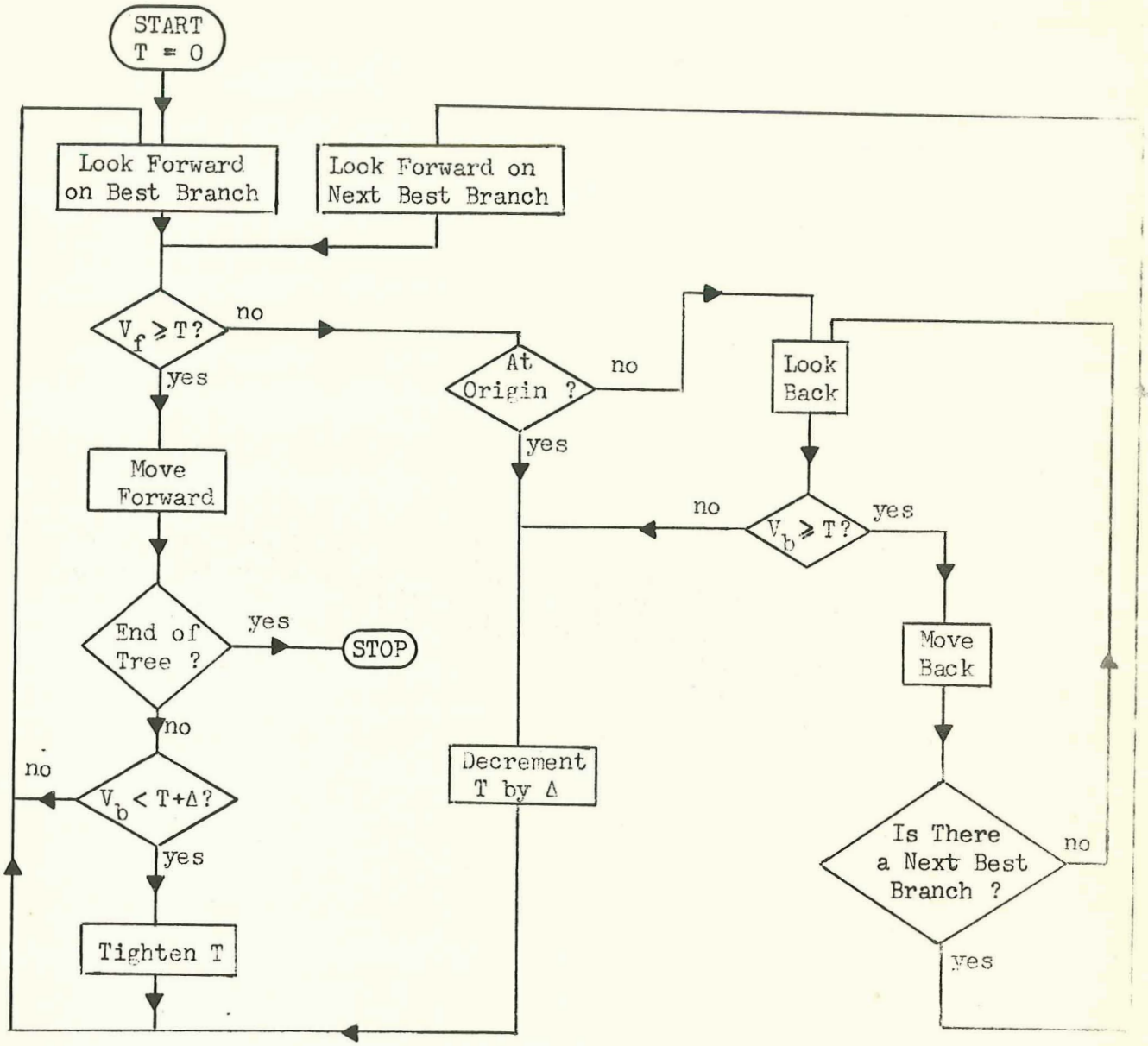
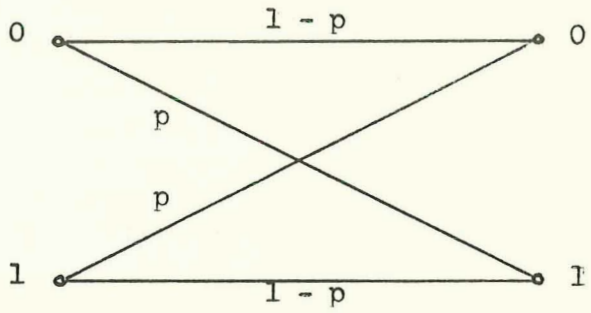


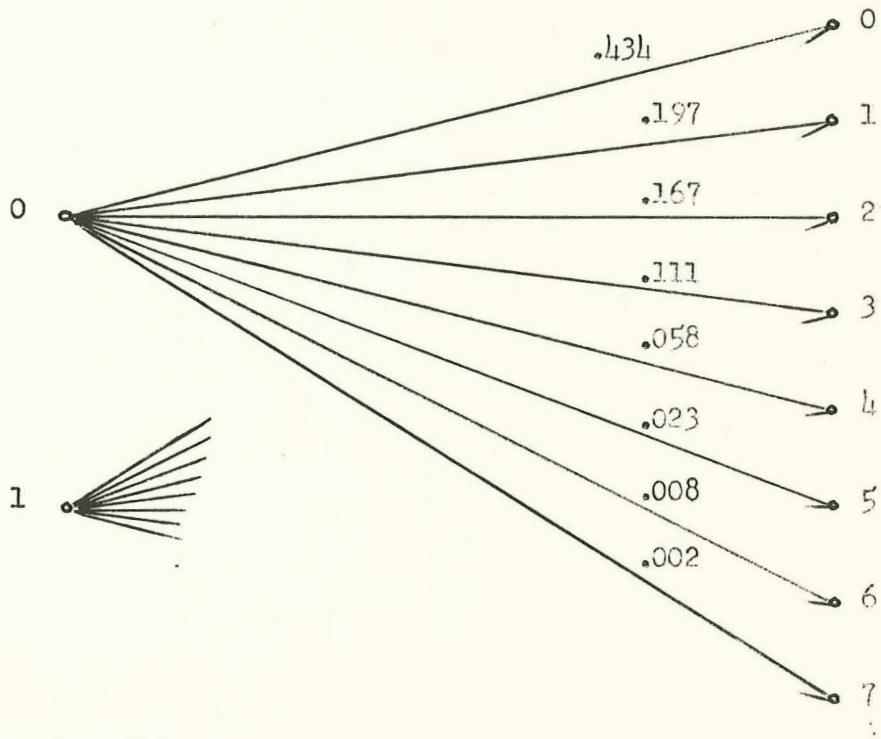
Figure 1

Fano Sequential Decoding Algorithm

V_f: Value at forward node under inspection
V_b: Value at previous node
T: Current threshold level
Δ: Threshold increment



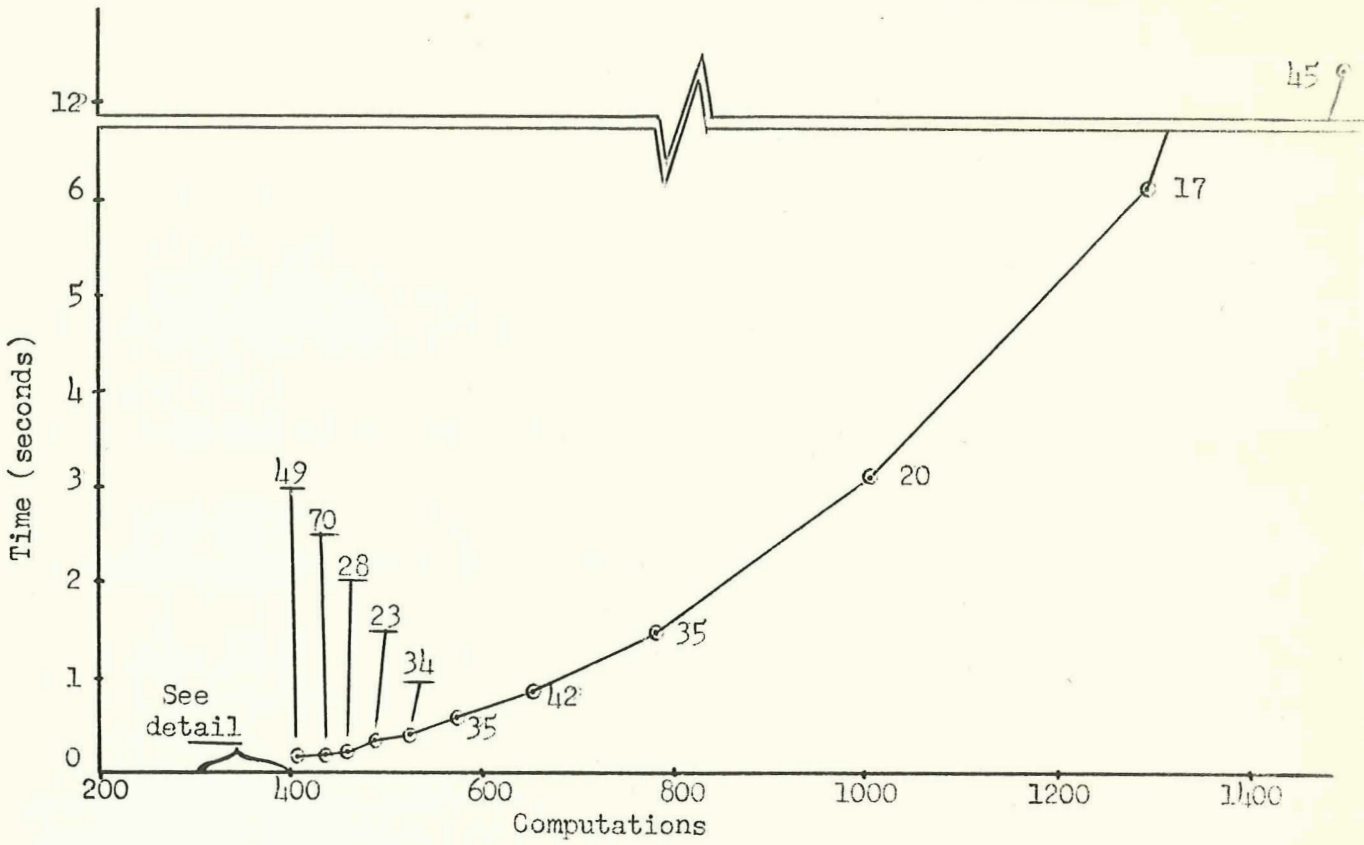
(a) Binary Symmetric Channel



(b) Gaussian Channel

Figure 2

Channels Used in the Simulation



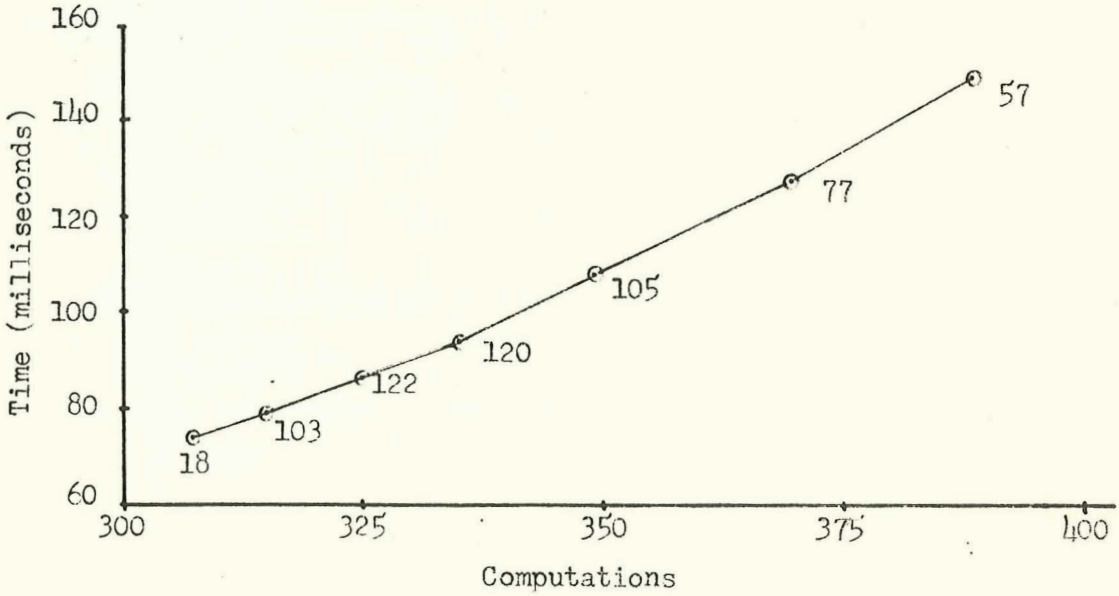


Figure 3

Variation of Computing Time with Number of Computations for the Jelinek Algorithm

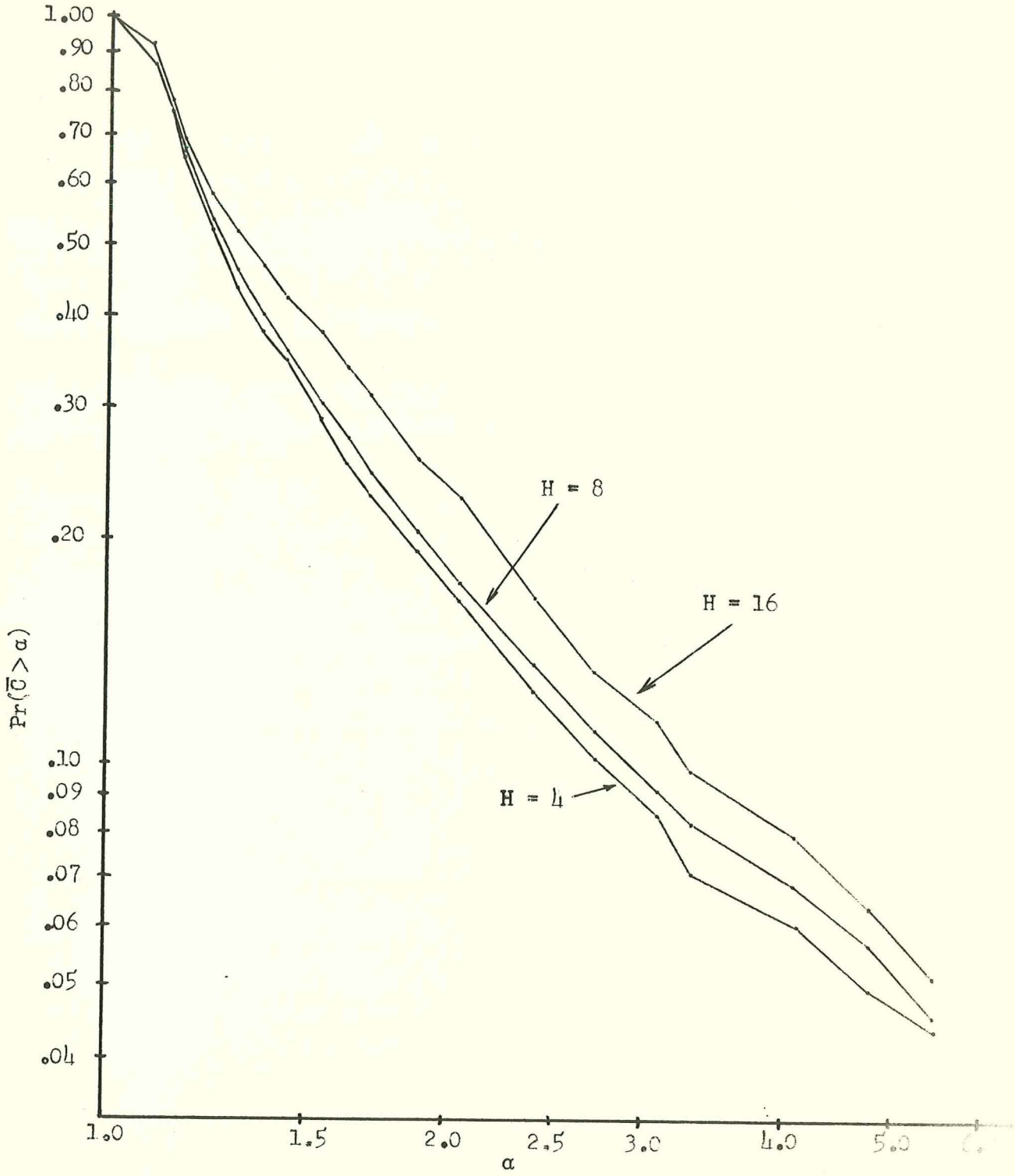


Figure 4

Modified Jelinek Algorithm

Distribution of Computation for Various Bin Spacings

Binary Symmetric Channel, p=.045

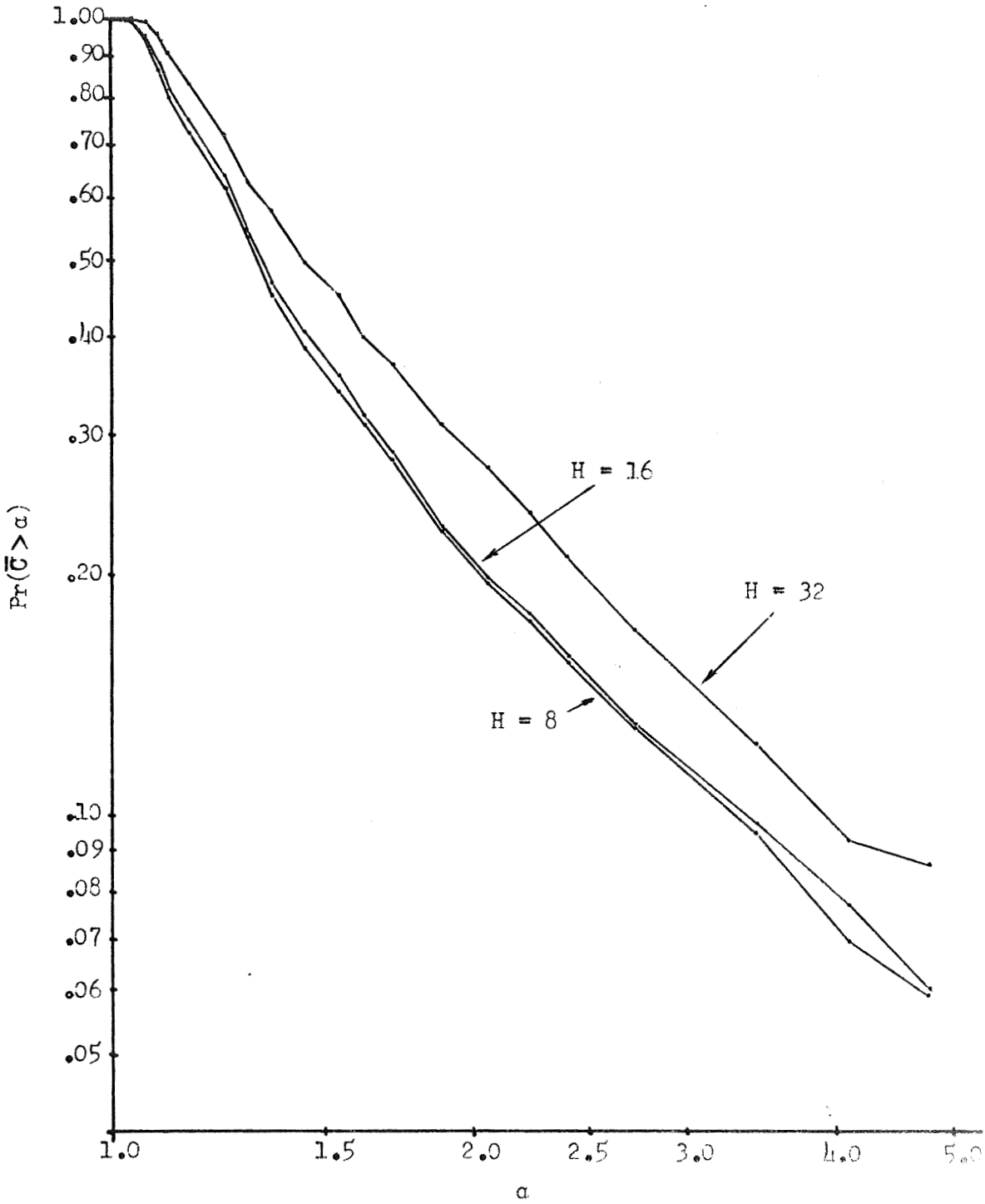


Figure 5

Modified Jelinek Algorithm

Distribution of Computation for Various Bin Spacings

Gaussian Channel

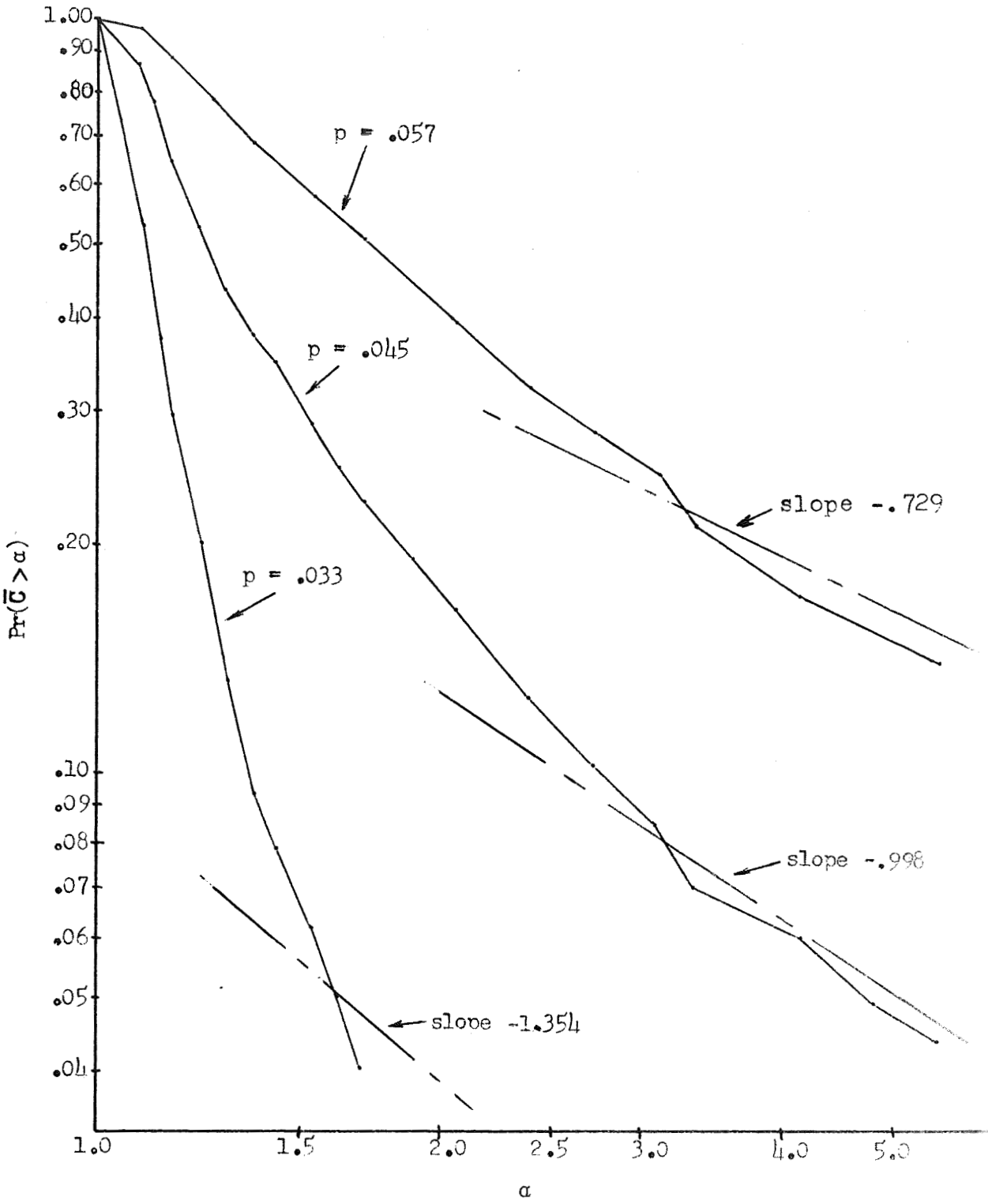


Figure 6

Modified Jelinek Algorithm

Distribution of Computation on the
Binary Symmetric Channel

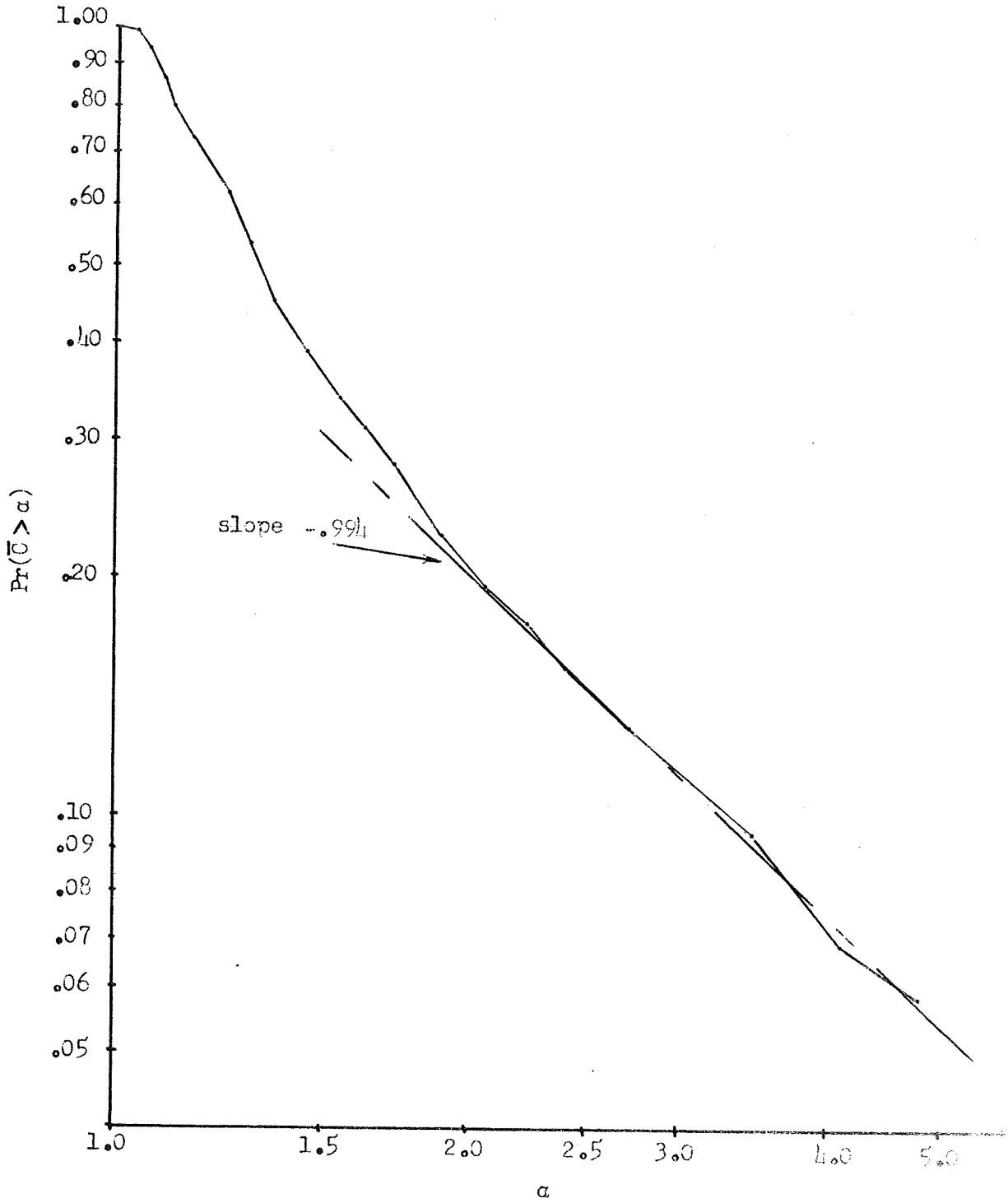


Figure 7

Modified Jelinek Algorithm

Distribution of Computation on the Gaussian Channel

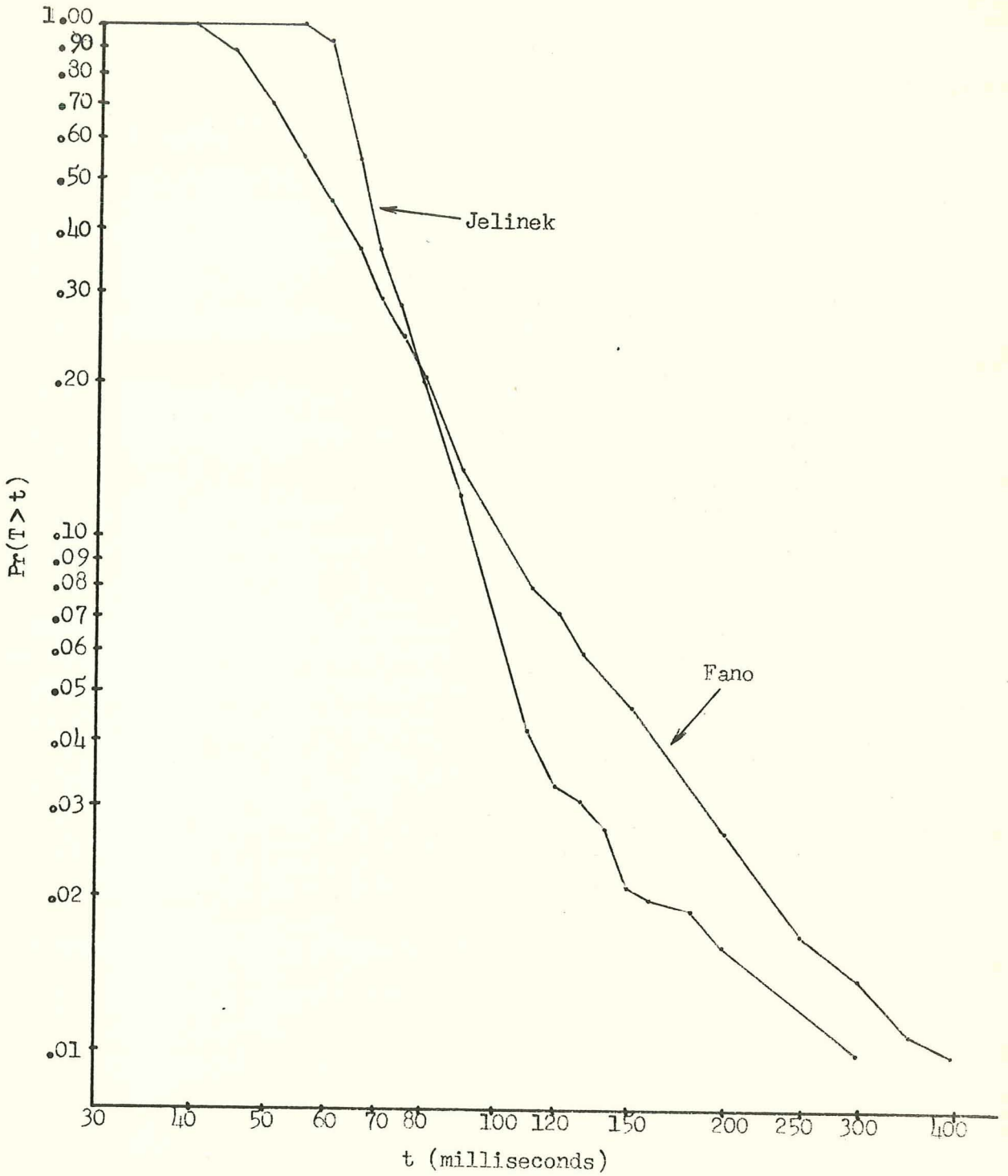


Figure 8A

Distribution of Computing Time

Binary Symmetric Channel, p=.033

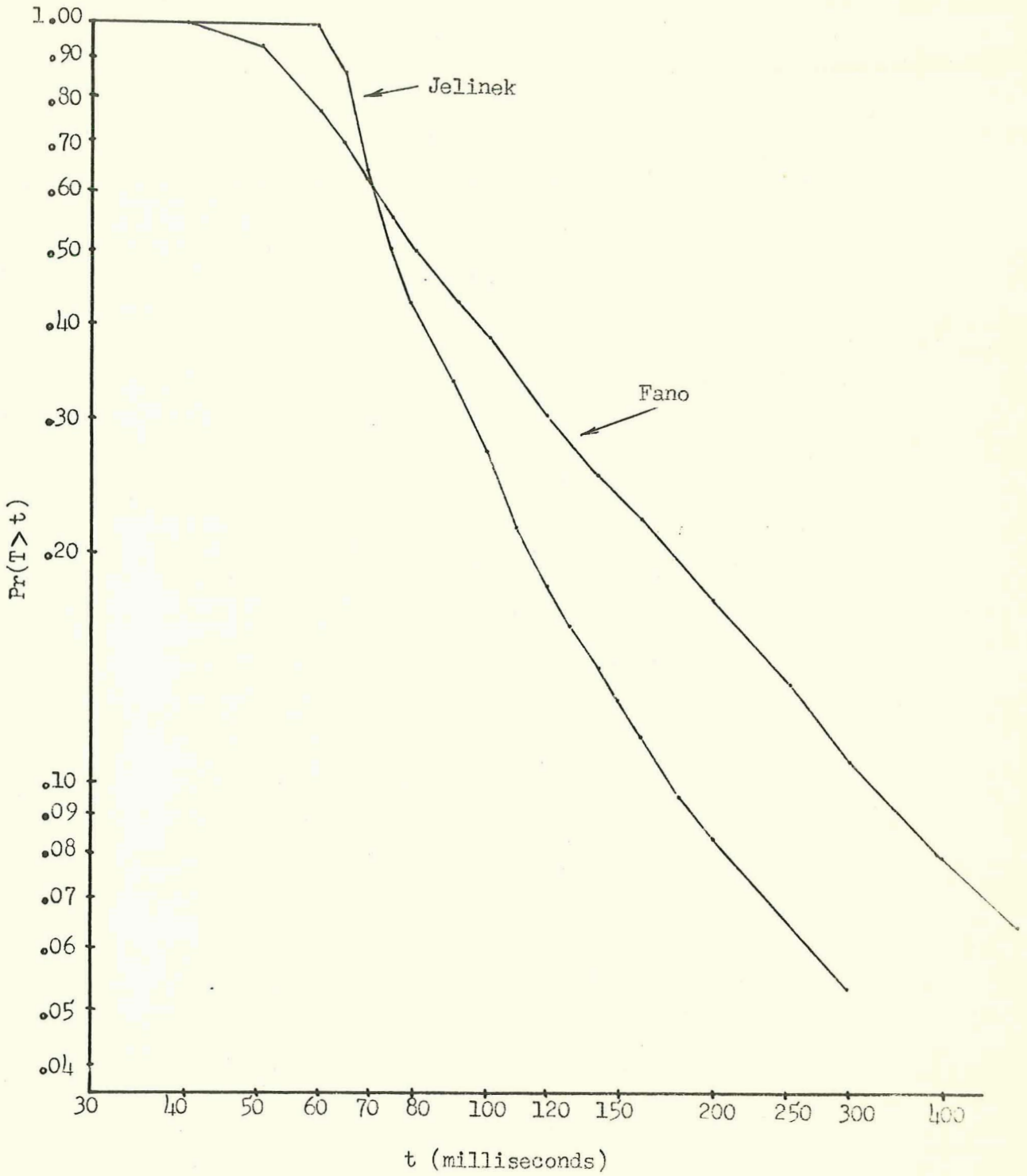


Figure 8B

Distribution of Computing Time

Binary Symmetric Channel, p=.045

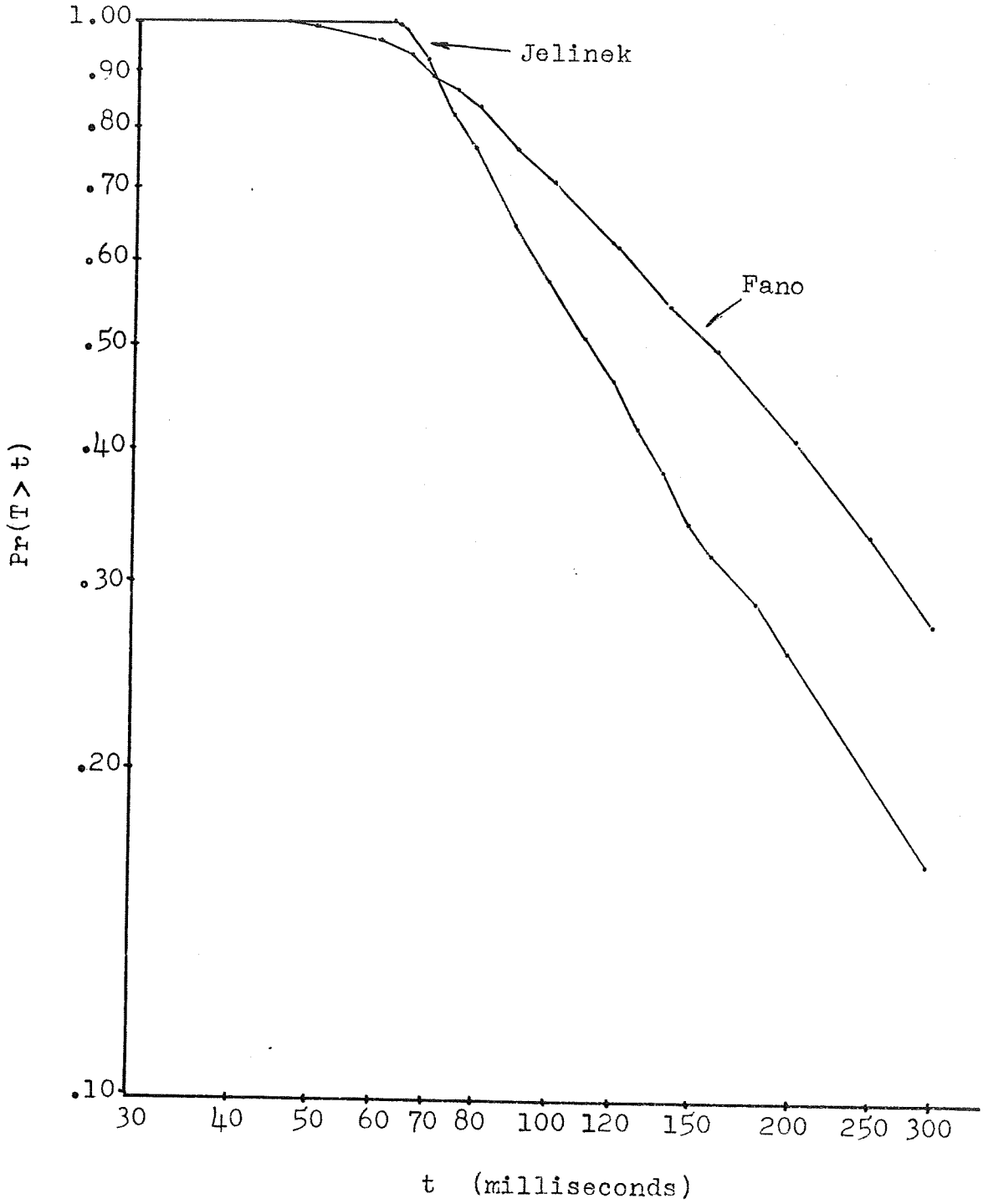


Figure 8C

Distribution of Computing Time

Binary Symmetric Channel, p=.057

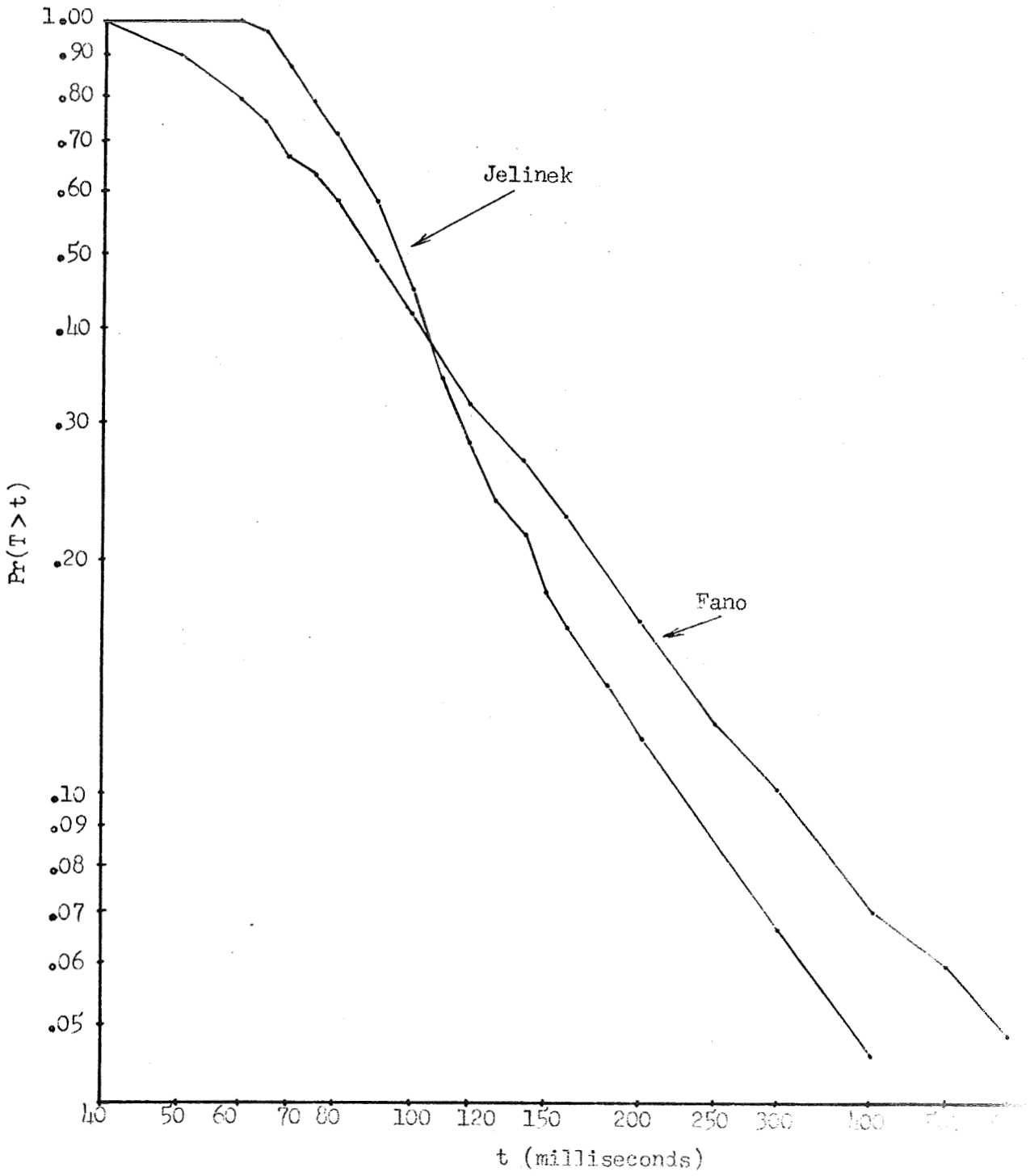


Figure 9

Distribution of Computing Time

Gaussian Channel

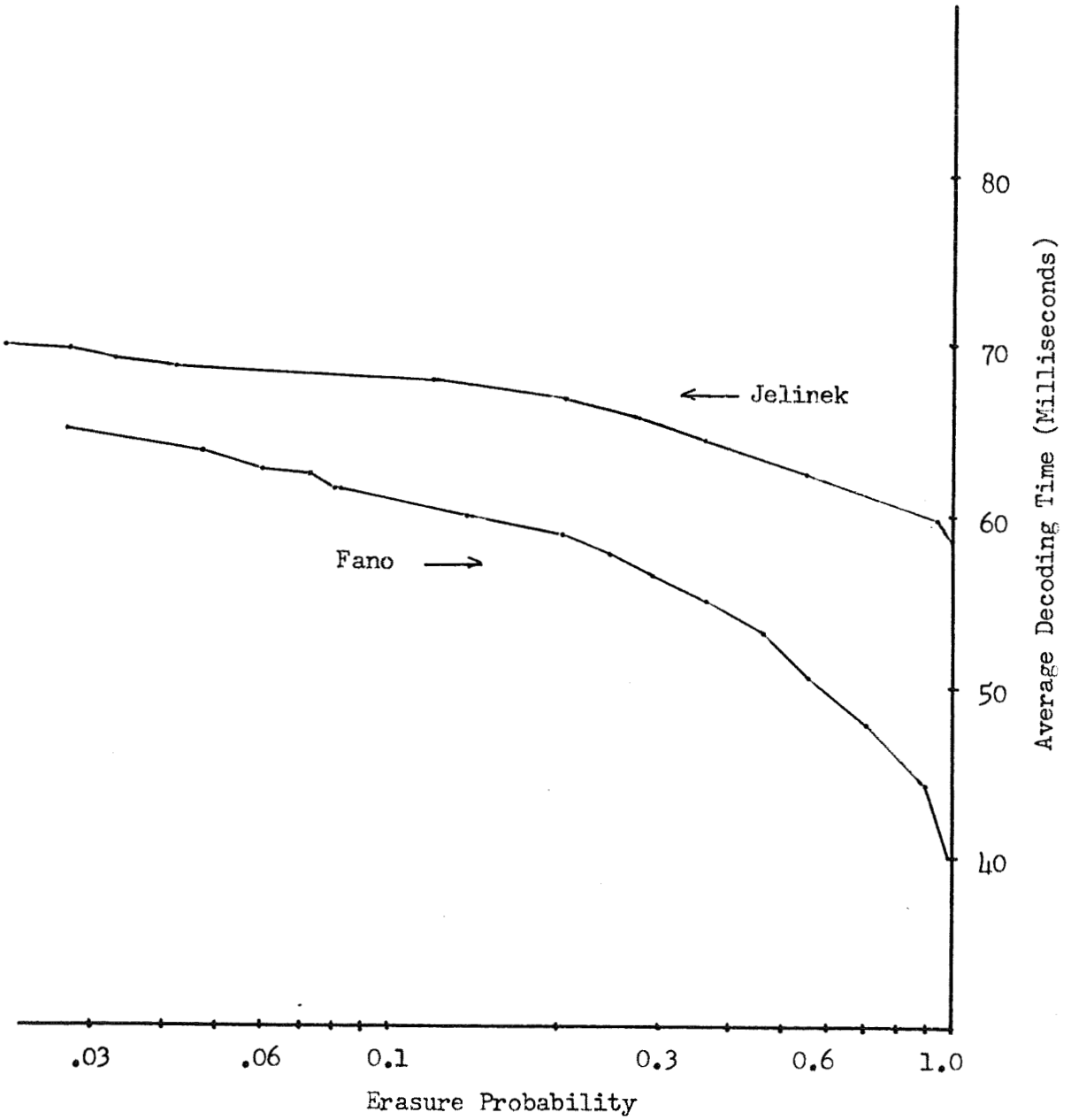


Figure 10A

Performance of Time-Limited Decoders
Binary Symmetric Channel, p=.033

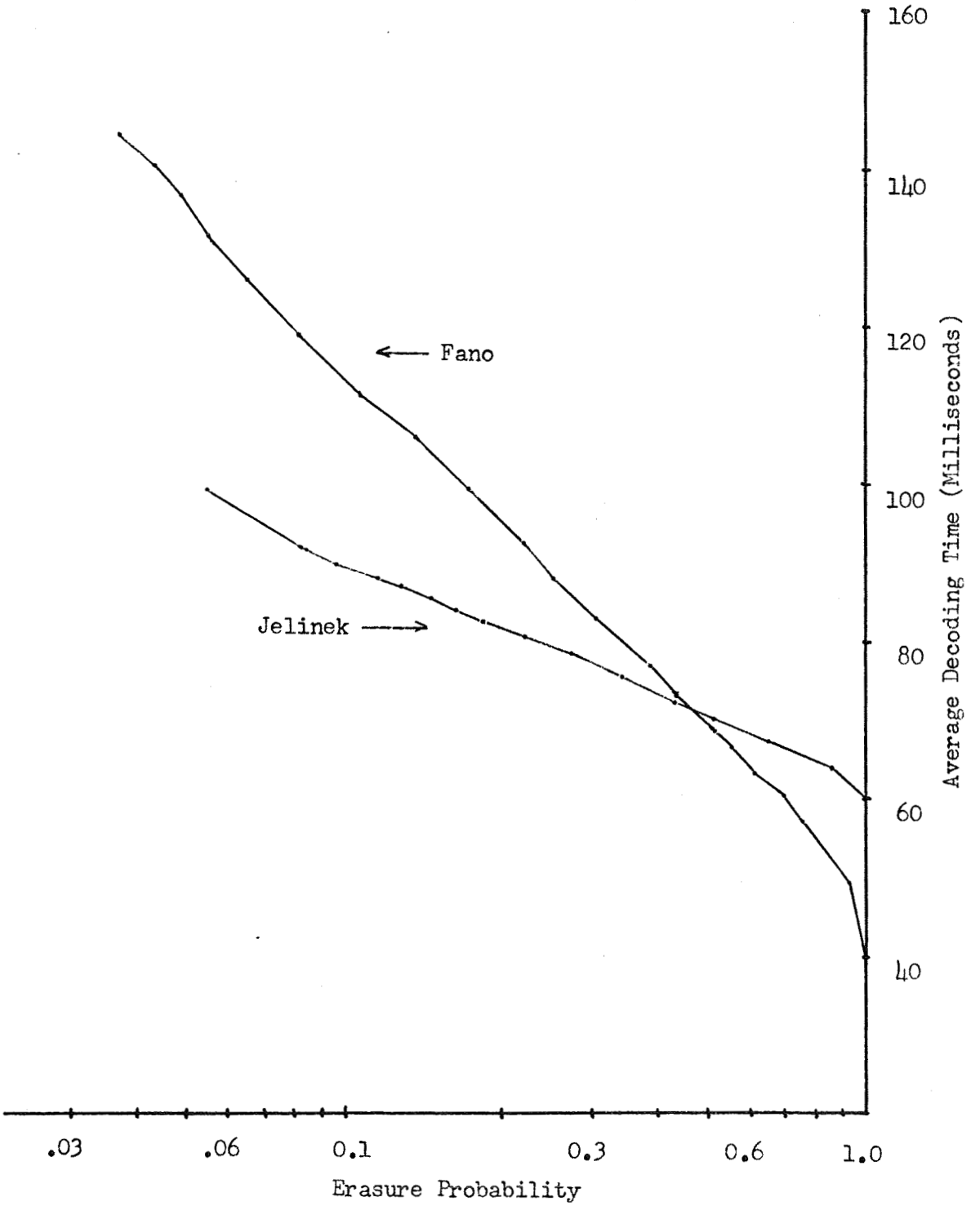


Figure 10B

Performance of Time-Limited Decoders
Binary Symmetric Channel, p=.045

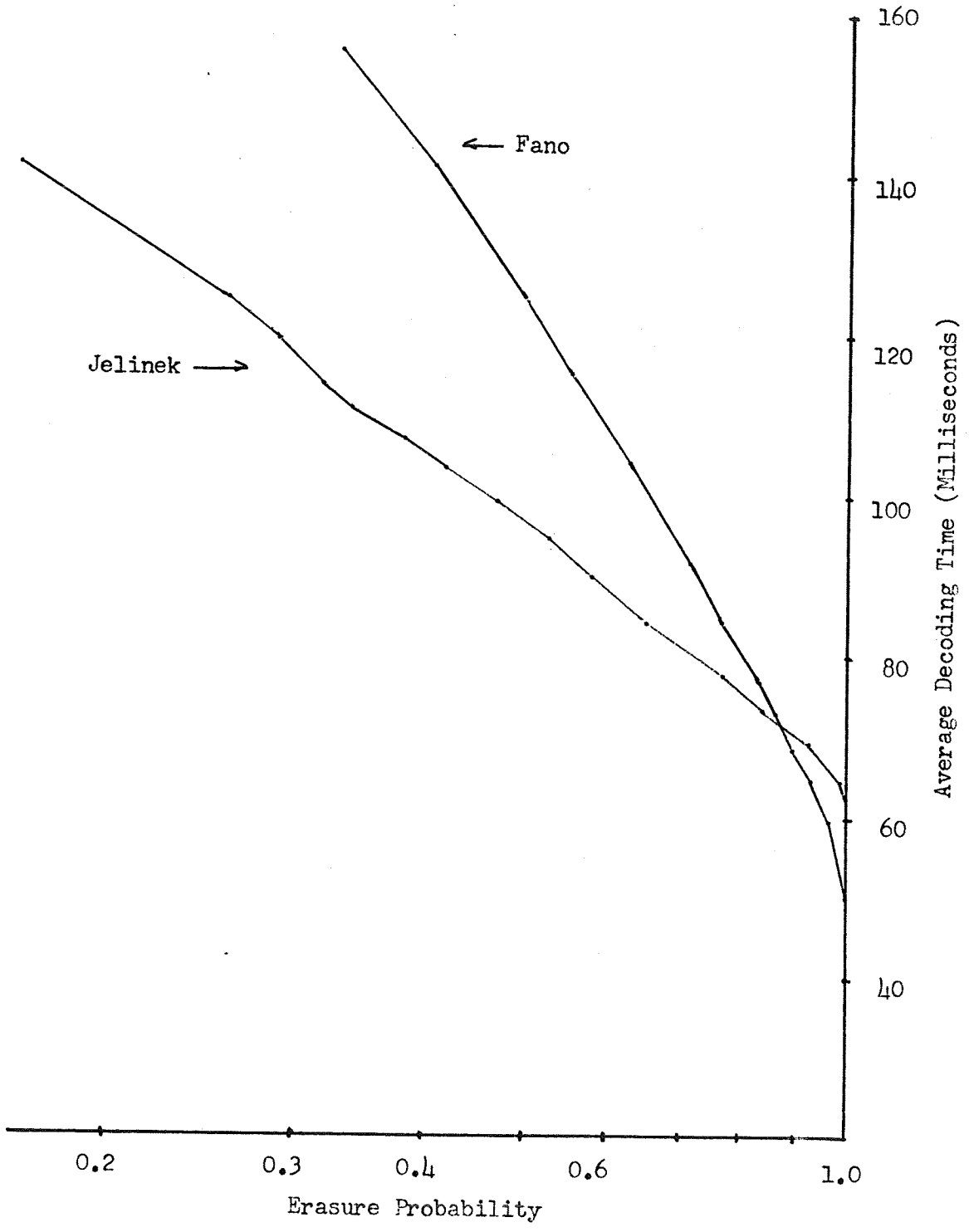


Figure 10C

Performance of Time-Limited Decoders

Binary Symmetric Channel, p=.057

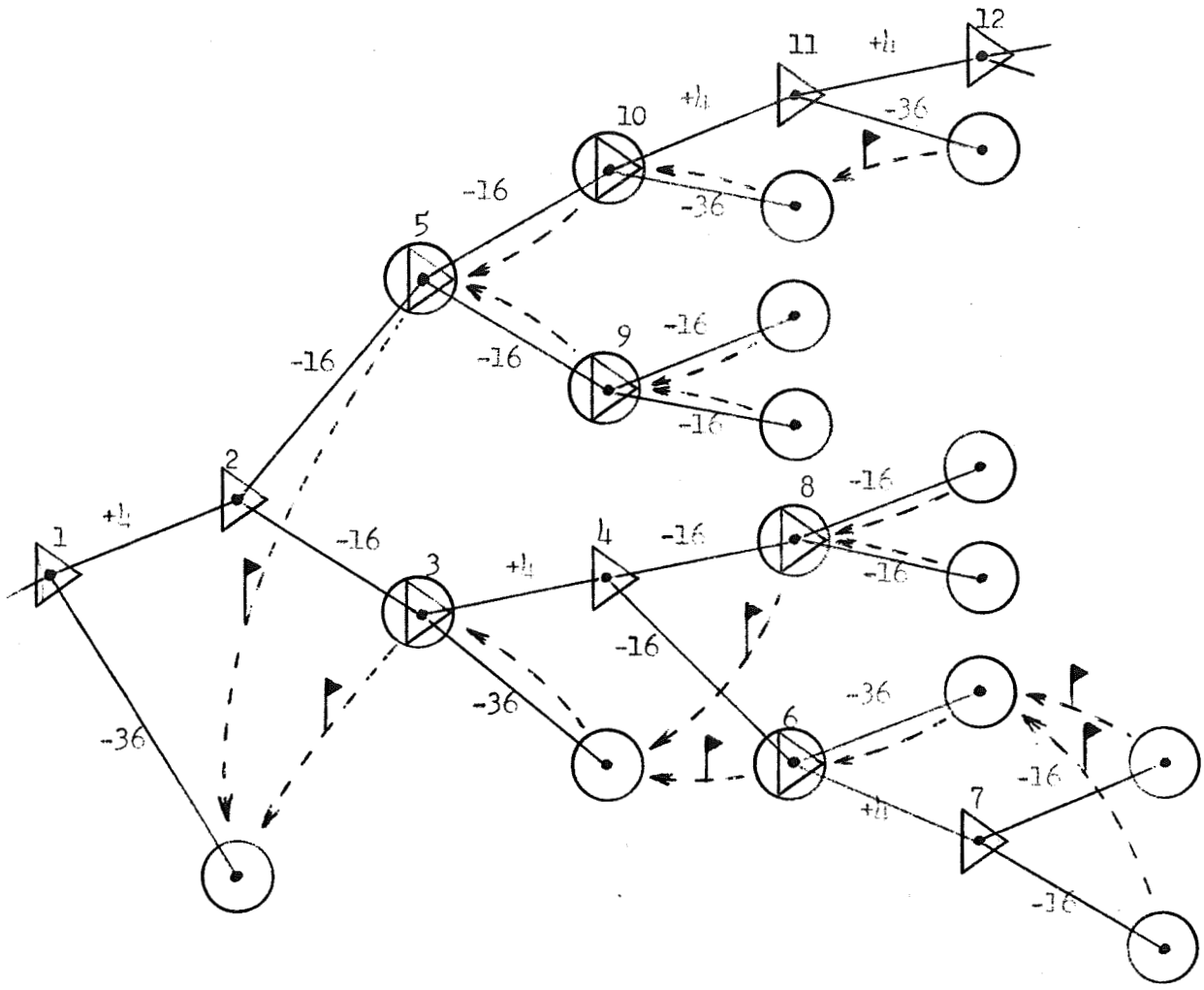


Figure 11

A Typical Search by the Jelinek Decoder

-- A node which is extended

-- A node for which a description is stored

-- A path pointer

-- An indicator that the path pointer references the complement of the predecessor