ANALYSES AND COST EVALUATION OF CODE TREE SEARCH ALGORITHMS

ANALYSES AND COST EVALUATION OF CODE TREE SEARCH ALGORITHMS

By

SESHADRI MOHAN, B.E. (HONS), M.Tech.

A Thesis

Submitted to the School of Graduate Studies

in Partial Fulfilment of the Requirements

for the Degree

Doctor of Philosophy

McMaster University

September 1979

DOCTOR OF PHILOSOPHY (1979)
(Electrical Engineering)

McMASTER UNIVERSITY
Hamilton, Ontario

TITLE: Analyses and Cost Evaluation of Code Tree Search Algorithms

AUTHOR: Seshadri Mohan, B.E. (HONS) (Univ. of Madras)
M.Tech. (Indian Institute of Technology, Kanpur)

SUPERVISOR: Professor J.B. Anderson

NUMBER OF PAGES: xiv, 204

ABSTRACT

Codes with a tree structure find wide use in data compression and error correction. It is generally impractical to view and weigh all the branches in a code tree, so a search algorithm is employed which considers some but not others in a predetermined fashion. Traditionally, the efficiency of code tree search algorithms has been measured by the number of tree branches visited for a given level of performance. This measure does not indicate the true consumption of resources. Cost functions are defined based on the number of code tree paths retained, S, the length of the paths, L, and the number of code tree branches searched per branch released as output, E[C]. Using these cost functions, most of the existing algorithms as well as some new algorithms proposed here are compared.

These new algorithms include three metric-first algorithms. The first one, the merge algorithm, uses, in addition to the main list used by the stack algorithm, an auxiliary list to store paths. The merge algorithm reduces the dependence on S for the product resource cost from $O(S^2)$ for the stack algorithm to $O(S^{4/3})$ for the merge algorithm. A generalization of this algorithm reduces the product cost

to O(S log S). The second algorithm uses a class of height-balanced trees, known as AVL trees, to store code tree paths, resulting in an alternate method to the merge algorithm achieving O(S log S) cost.

The third algorithm, using the concepts of dynamic hashing and trie searching, provides important modifications to the Jelinek bucket algorithm by incorporating dynamic splitting and merging of buckets. This strategy provides a balanced data structure and reduces the product cost still further compared to the first two algorithms.

We next turn to analysis of the number of nodes visited during a search. Using the theory of multitype branching processes in random environments an equation for node computation is derived for asymmetric source coding by the single stack algorithm. This equation is shown to be the stochastic analog of an equation for symmetric sources. Simulation results, obtained by encoding the Hamming source by the single stack algorithm, are used to optimize the performance of the algorithm with respect to the bias factor, stack length, and limit on computation. A modification to the algorithm that raises the barrier during forward motion provides a better distortion performance.

The metric-first stack algorithm is used to encode a voiced speech sound. From experimental evidence, it is

shown how to optimize the algorithm's SNR performance with respect to the algorithm's storage, execution time, and node computation. For each of these, the optimal parameterizing of the algorithm differs markedly. Similarities are pointed out between the results for speech and earlier theoretical results for the binary i.i.d. source with Hamming distortion measure. It is shown that metric-first algorithms may perform better with "real life" sources like speech than they do with artificial sources, and in view of this the algorithms proposed here take on added significance.

ACKNOWLEDGEMENTS

I am deeply indebted to my supervisor, Dr. John B. Anderson, for his constant encouragement, invaluable support, and continuing guidance throughout the course of this research. My association with Dr. Anderson has been a singularly rewarding experience.

I would like to thank Dr. D.P. Taylor and Dr. D. Wood for serving as members of my supervisory committee. I would also like to express my deep appreciation to McMaster University for supporting this research with a Dalley Fellowship.

The cheerful and excellent typing services of Miss Pat Dillon of the Word Processing Centre are gratefully acknowledged. Thanks are also due to Mr. G. Kappel for his excellent drawings.

TABLE OF CONTENTS

TABLE OF CONTENTS (continued)

TABLE OF CONTENTS (continued)

LIST OF FIGURES

LIST OF FIGURES (continued)

LIST OF FIGURES (continued)

LIST OF FIGURES (continued)

LIST OF TABLES

CHAPTER 1

INTRODUCTION

There is an ever increasing demand to transmit information rapidly. Shannon's paper [1], which laid the foundation for active research in information theory, showed that reliable communication with arbitrarily low probability of error was possible at rates below channel capacity. Conversely, at all rates exceeding capacity, the error probability will approach unity with increasing code word length. Excellent references [2]-[6] are available, giving both theoretical codes satisfying Shannon's existence theorem, and practical and instrumentable codes. We begin by discussing codes and code tree search algorithms used in source encoding and channel decoding, and, as a prelude, a digital communication system model.

1.1 A Digital Communication System Model

Figure 1.1 shows, in block diagram form, all the relevant functions performed in the transmission of information. The source produces outputs x_t, at time t, chosen according to a given probability distribution from the space X of possible source outputs. The entropy of the

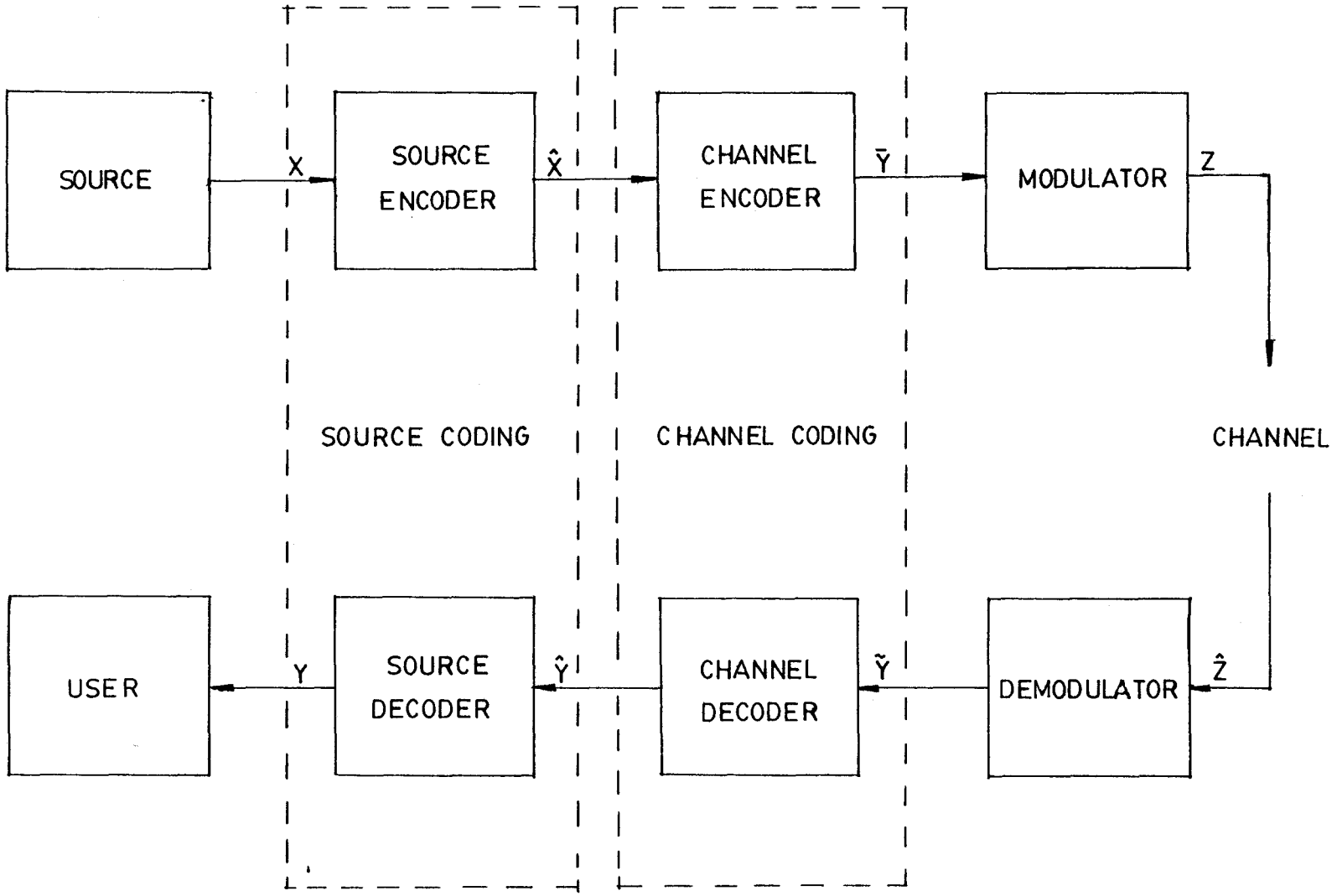


Figure 1.1 A Digital Communication System Model

source H(X) is greater than C, the channel capacity.

The source encoder transforms the source output x_t into an approximation $\hat{x}_t$ such that the entropy $H(\hat{X}) < C$, where $\hat{X}$ is the reproducer alphabet. Equivalently, the source encoder can be viewed as a device that partitions the space of possible source outputs into equivalence classes and informs the channel encoder which of these the particular source output belongs to. The source encoder is a complex device that performs a many-to-one mapping. Its complexity depends on how stringent the requirements on the mapping are.

The channel encoder receives the output of the source encoder and by means of a suitable encoding transforms it into a form suitable for efficient transmission over the channel. This is a relatively simple device that performs a one-to-one mapping.

The signal received by the channel decoder at the other end of the channel is corrupted by noise introduced in transmission through the channel and the function of the decoder is to determine from the sequence of received symbols over an appropriate period of time which of the messages was sent. Like the source encoder, the channel decoder is a complex device performing a many-to-one mapping.

Upon receiving the estimate $\hat{y}_t$ of the channel decoder, the source decoder performs a one-to-one mapping

and presents its estimate y_t of the source output x_t to the user. Like the channel encoder, the source decoder performs a relatively simple task.

The above discussion brings out the similarities between source encoders and channel decoders and also between channel encoders and source decoders. However, the channel decoder is required to find the best estimate whereas the source encoder finds an estimate of the source output that meets a certain distortion criterion. Usually, there exist several estimates that meet the criterion.

1.2 History of Source Coding and Sequential Decoding Algorithms

Huffman's codes [7] are examples of optimal variable-length, uniquely decodable, noiseless source codes. These codes are difficult to implement and they almost always involve encoder buffer overflow [8], no matter how large the buffer is. We are concerned here not with noiseless coding, but with coding with respect to a fidelity criterion; that is, with determining the least rate at which information must be transmitted in order that the total distortion does not exceed some given distortion D. Coding with a fidelity criterion was first proposed by Shannon in [9], where he defined the rate distortion function of an information source. Berger's book [10] is devoted entirely to these

rate versus distortion trade off functions R(D) for various sources. Jelinek [4] and Gallager [5] devote a chapter each to this problem while a chapter in [6] concerns itself with trellis source coding using convolutional codes and the Viterbi algorithm.

Source coding is a two-fold design problem consisting of 1) the design of good codes that guarantee performance arbitrarily close to R(D), and 2) the design of efficient algorithms that explore among the code words in order to find one with the given distortion performance. Only ten years after the birth of rate distortion theory did the first paper [11] specifically on codes appear. In [11], Jelinek showed the existence of a class of codes with a perfectly regular tree structure (henceforth known as tree codes) that achieved performance arbitrarily close to R(D), thus giving attention to the first design facet of source coding. Viterbi and Omura [21] have shown the existence of time-varying trellis codes that achieve the rate distortion bound. It is the popular belief that fixed convolutional codes will achieve the limit predicted by rate distortion theory, but this remains an open problem.

The second facet of source coding was considered by Jelinek and Anderson [12] when they proposed their (M, L) algorithm to encode the binary i.i.d. source. Later it was applied to speech as well [13]-[15]. Now there exist

several other algorithms [16]-[18] that have been proven to achieve performance arbitrarily close to R(D). All these algorithms search tree codes, hence the name code tree search algorithms. An algorithm utilizing convolutional codes is the Viterbi algorithm [19],[20].

The developments in the sequential decoding area led by almost a decade those in the field of source coding. So, the wealth of information and analysis techniques accumulated in the sequential decoding field were often exploited to advantage in the source coding area. We briefly trace the development in sequential decoding.

For sequential decoding algorithms, the counterpart of the distortion criterion for source coding is the probability of error due to incorrectly decoding a received symbol. Wozencraft [22],[23] devised the earliest sequential decoding method that achieved arbitrarily small error probability at non-zero rates. Fano [24] introduced modifications to it that made the algorithm analytically tractable. Massey [25] has shown that the Fano metric (see sec. 1.3 for its definition) proposed in [24] is optimum in that it enables the algorithm to minimize the probability of error. He further shows that the algorithm due to Zigangirov [26], and invented independently by Jelinek [27], follows naturally as a consequence of this interpretation of the Fano metric (this algorithm tries to maximize the

probability that the next step taken is along the correct path; that is, the node with the best metric is extended at any time). Recently, two other algorithms [28],[29] have been proposed, but these are variations of the Zigangirov-Jelinek stack algorithm.

1.3 Tree Codes and Convolutional Codes

By a tree code is meant a code whose words may be graphed on a perfectly regular tree structure with b branches out of each node and β symbols on each branch. Such a code is said to have a rate

$$R = \frac{\log_2 b}{\beta} \text{ bits/source symbol.} \qquad (1.1)$$

A tree code has a distinguished node called the root node from which all code words begin. Each code word corresponds to a path starting from the root node and consisting of a chain of code tree branches.

Figure 1.2 shows a rate 1/2 binary tree code with $b = \beta = 2$. A path through the tree code has a path map associated with it. The upward branch out of a node has associated with it the path map symbol 0 and the downward branch the symbol 1. For example, a path with path map 1 0 1 1 is shown dotted in Fig. 1.2.

The tree has four levels, one corresponding to each branch along a path of the code tree. Sometimes it is

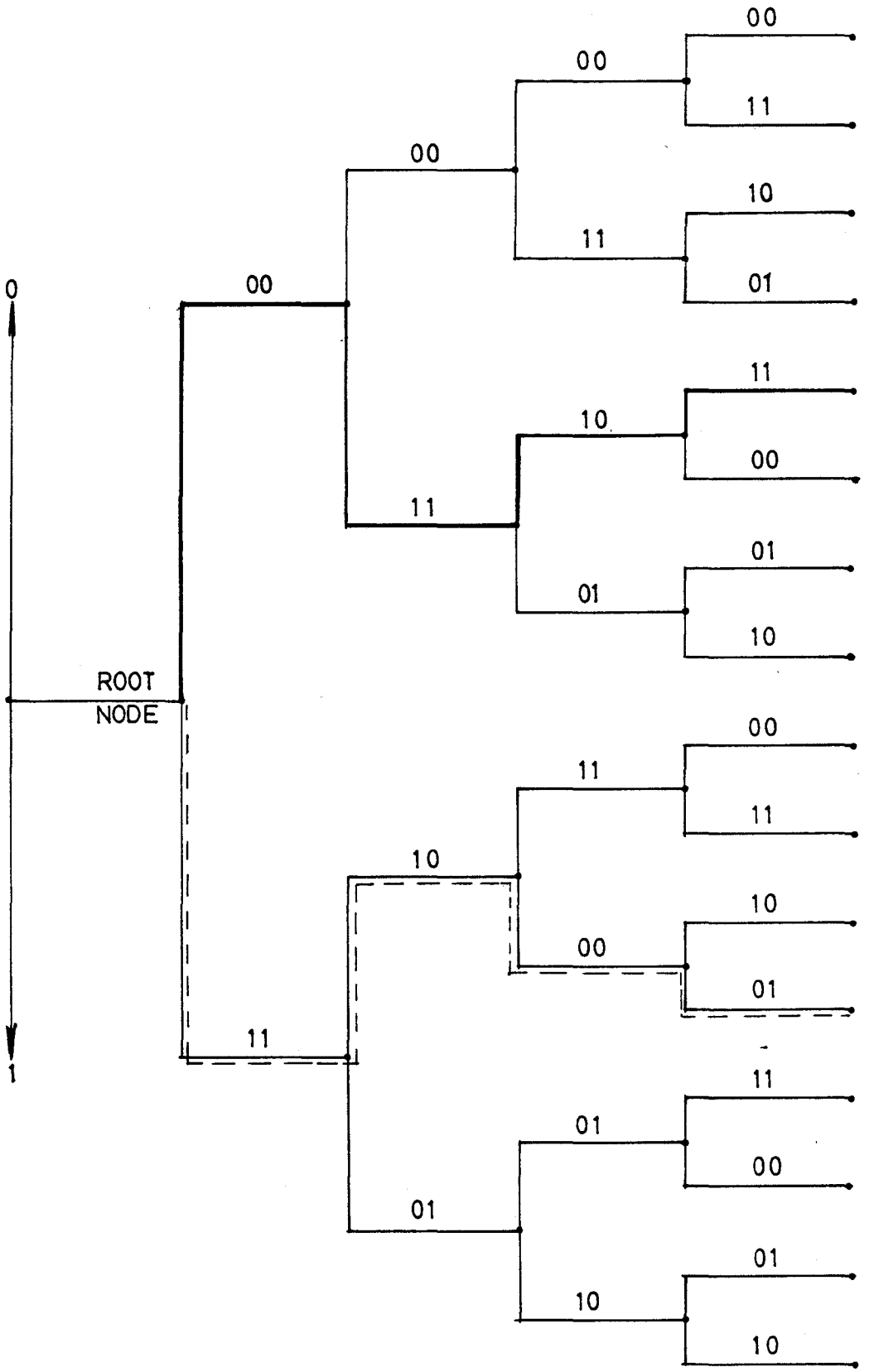


Figure 1.2 A Rate 1/2 Binary Tree Code, b = β = 2

convenient to associate levels with nodes in which case we make the convention that the root node is at level 0, all descendents of root node are at level 1, all descendents of the level 1 nodes at level 2, and so on. Theoretically, a tree code can have infinitely many levels, but any useable and instrumentable code has a trellis structure requiring only finite storage.

Assume that a binary source is to be transmitted over the channel. A channel encoder using the tree code of Fig. 1.2 will follow the upward branch from the current node at which the encoder is stationed if a 0 is to be transmitted and the downward branch if a 1 is to be transmitted, and will transmit the reproducer symbols on the branch followed. Thus, for a message sequence of 1 0 1 1, the reproducer sequence is 11 10 00 01.

A source encoder using the tree compares the source letter at a given level with the code word letters at that level and assigns a metric to code tree paths based on a distortion criterion. Define the metric $\mu(\hat{\underset{\sim}{x}}^{\ell\beta})$ of a code tree path $\hat{\underset{\sim}{x}}^{\ell\beta}$ of $\ell\beta$ code word letters (or ℓ code tree branches) by

$$\mu(\hat{\underset{\sim}{x}}^{\ell\beta}) = \ell\beta D^* - d(\underset{\sim}{x}^{\ell\beta}, \hat{\underset{\sim}{x}}^{\ell\beta}) \qquad (1.2)$$

where $\underset{\sim}{x}^{\ell\beta}$ is a sequence of $\ell\beta$ source letters, $d(.,.)$ is the distortion between $\underset{\sim}{x}^{\ell\beta}$ and $\hat{\underset{\sim}{x}}^{\ell\beta}$ and D^* is a bias factor. For additive single letter fidelity criterion

$$d(\underset{\sim}{x}^{\ell\beta}, \hat{\underset{\sim}{x}}^{\ell\beta}) = \sum_{i=1}^{\ell} d(\underset{\sim}{x}_i^{\beta}, \hat{\underset{\sim}{x}}_i^{\beta}) \tag{1.3}$$

and, consequently,

$$\mu(\hat{\underset{\sim}{x}}^{\ell\beta}) = \sum_{i=1}^{\ell} \mu(\hat{\underset{\sim}{x}}_i^{\beta}) \tag{1.4}$$

where $\underset{\sim}{x}_i^{\beta}$ and $\hat{\underset{\sim}{x}}_i^{\beta}$ are groups of β source and code word letters, respectively, at level i. By encoding a source we mean finding a path through the code tree so that the per-letter distortion of the encoded path is within a given distortion D; i.e., $d(\underset{\sim}{x}^{\ell\beta}, \hat{\underset{\sim}{x}}^{\ell\beta})/\ell\beta \leq D$. For the Hamming distortion criterion, i.e., $d(x, \hat{x}) = \delta_{x,\hat{x}}$, where δ is the Kronecker delta, the best (the least distortion) code tree path corresponding to the source letters $\underset{\sim}{x}^{8}$ = 01 11 10 11 is shown by the thick line in Fig. 1.2.

For sequential decoding, we define the metric of a code tree path $\hat{\underset{\sim}{x}}^{\ell\beta}$ as

$$\mu(\hat{\underset{\sim}{x}}^{\ell\beta}) = \sum_{i=1}^{\ell} \mu(\hat{\underset{\sim}{x}}_i^{\beta}) \tag{1.5}$$

where

$$\mu(\hat{\underset{\sim}{x}}_i^{\beta}) = \log_2 \left[\frac{p(\underset{\sim}{x}_i^{\beta} | \hat{\underset{\sim}{x}}_i^{\beta})}{p(\underset{\sim}{x}_i^{\beta})}\right] - \beta R \tag{1.6}$$

and $\hat{x}$ and x are code word letters and channel output letters (received symbols) to the decoder, respectively; $p(x_i | \hat{x}_i)$ is the distribution of the channel output conditioned on its input and

$$p(x_i) = \sum_{\hat{x}_i} q(\hat{x}_i)\ p(x_i|\hat{x}_i) \qquad (1.7)$$

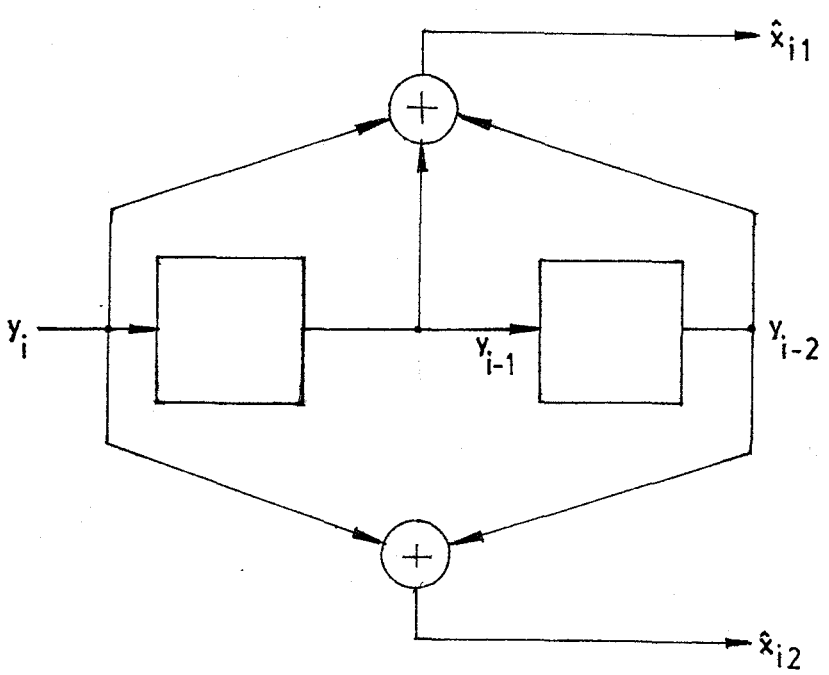
where $q(\hat{x}_i)$ is the distribution of the received symbols. Here the channel is assumed to be memoryless.

The metric defined in (1.6) and (1.7) was first introduced by Fano [24] and is known as the Fano metric. It has the property that its average per-symbol metric increment is always positive along a correct path provided R is less than C, the channel capacity, while along an incorrect path it is always negative [24]. This enabled Fano to postulate an algorithm based on these heuristic considerations, wherein, sooner or later, any incorrect path being pursued fell below a threshold, while the correct path eventually remained above it.

We now briefly consider convolutional codes and their trellis structure. A general rate $R = \nu/\beta$ convolutional code is generated by a linear sequential circuit with ν inputs and β outputs. For simplicity, we consider rate $1/\beta$ convolutional codes.

Figure 1.3(a) shows a shift register circuit that generates a rate 1/2 convolutional code of order 2 and Fig. 1.3(b) its state diagram. By state of the encoder we mean the tuple (y_{i-1}, y_{i-2}), the immediate two past inputs to the shift register. If y_i is the present input, a transformation of the state from (y_{i-1}, y_{i-2}) to (y_i, y_{i-1})

a)



b)

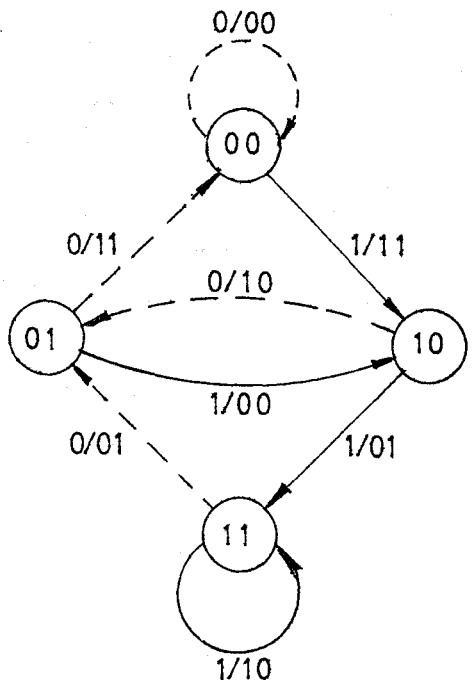


Figure 1.3(a) A Rate 1/2 Convolutional Encoder of Order 2

Figure 1.3(b) State Diagram of the Encoder of Fig. 1.3(a)

takes place. In the state diagram, the circular nodes represent the states and directed branches the transitions. The symbols $y_i/\hat{x}_{i1}$, $\hat{x}_{i2}$ indicate that if the input is y_i, the outputs are $\hat{x}_{i1}$ and $\hat{x}_{i2}$ and a state transition takes place as indicated by the branch. The addition boxes in Fig. 1.3(a) perform modulo 2 addition.

The trellis structure of the convolutional encoder of Fig. 1.3(a) is generated as follows. Assume an initial state of (0, 0). If the input is a 0, the state remains at (0, 0), and if it is a 1, a transition to (1, 0) takes place. This is indicated by the level 1 transitions in Fig. 1.4, where the horizontal axis denotes time or trellis level. From the states in level 1, again transitions take place depending on the inputs, and these are denoted by the level 2 transitions in the figure. When carried on indefinitely, this results in the trellis structure for the code. The letters on trellis branches in Fig. 1.4 indicate outputs.

In Fig. 1.4, the two paths with path maps 0, 0, 0 and 1, 0, 0 remerge into state 00 at level 3. If the metric of the path 0, 0, 0 is greater than that of 1, 0, 0, then 1, 0, 0 cannot be a prefix of the maximum-likelihood path, the path with the largest metric evaluated with respect to a given sequence of symbols. The Viterbi algorithm uses this principle of optimality of dynamic programming to encode or

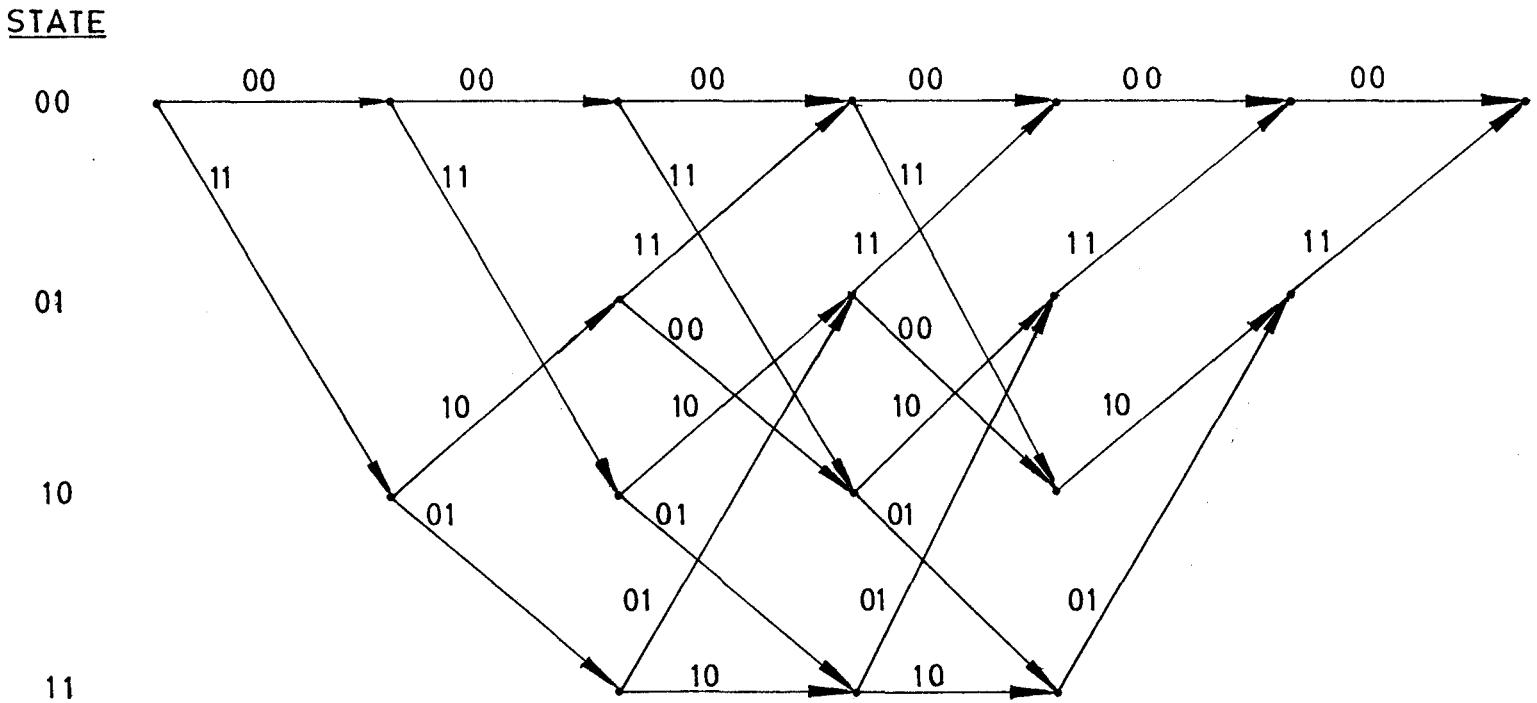


Figure 1.4 Trellis Structure of the Convolutional Encoder of Fig. 1.3(a)

decode a given sequence of symbols [20]. If code state transitions branch out without remerging, the tree code of Fig. 1.2 results and the Viterbi algorithm may not be used.

1.4 Expected Node Computation for Code Tree Search Algorithms

Define the node computation performed by an algorithm as the average number of code tree branches that the algorithm must scrutinize in order to release an output symbol, for a given level of source encoder fidelity or decoder probability of error.

The stack, M-, and 2-cycle algorithms (see Chapter 2 for a survey) for source coding obey the asymptotic formula as $D \downarrow \Delta$ [30]

$$E[C] \simeq C_1 \exp[(D-\Delta)^{-\alpha C_2}] \tag{1.8}$$

where $E[C]$ is the expected node computation, C_1 and C_2 are constants depending on the source and rate of the code, α is 1/2 or 3/4 depending on the algorithm used, D is the per-source-digit average distortion achieved by the algorithm, and $\Delta(\cdot)$ is the inverse rate distortion function. For the binary i.i.d. source, the rate distortion function is given by $R(D) = H_b(p) - H_b(D)$, where p is the probability of a 0 and $H_b(\cdot)$ is the binary entropy function. The values of α for various algorithms and sources were given in Anderson

[30].

The single stack algorithm's node computation is bounded from above by (1.8) with $\alpha = 1/2$ [18]. Viterbi and Omura [21] demonstrate the relation

$$E[C] \tilde{<} \exp[(D-A)^{-1/\gamma}] \tag{1.9}$$

for trellis source coding using the Viterbi algorithm. From experimental work for the M-algorithm, a highly truncated Viterbi algorithm, Anderson [30] conjectures a $\gamma \geq 4/3$.

For sequential decoding, asymptotic results show that the average number of computations required to decode a received symbol is Pareto distributed (see Savage [31]), i.e.,

$$P\{E[C] \geq N\} \simeq AN^{-\rho} \tag{1.10}$$

where A is a constant and ρ, called the Pareto exponent, is a function of the rate and channel.

There exists a rate R_{comp}, called the computational limit, above which the expected node computation increases exponentially with the number of levels in the code tree and below which it is bounded by a constant. R_{comp} is a function of the channel probabilities and exceeds C/2 for binary symmetric channels (BSC), for which it is given by [3, p. 399]

$$R_{comp} = 1 - \log_2 [1 + 2\sqrt{p(1-p)}]. \tag{1.11}$$

1.5 Applications of Code Tree Search Algorithms

Tree codes find wide use in source coding and sequential decoding. Algorithms to explore the code words exist and their use is well known (see [12], [16]-[18] for source coding and [20], [24], [26]-[29] for sequential decoding). They also find applications in speech and text recognition, and Forney [32] has applied the Viterbi algorithm to the intersymbol interference reduction problem. Use of this algorithm in syntactic pattern recognition [47] has also been proposed.

Recently code searching schemes have been extensively used in speech encoding [13]-[15], [33], [34]. The M-algorithm encoded speech yields 4-8 dB improvement in mean-square error over ordinary single-path searched DPCM [13]. This is a surprizing result in view of the only 1 to 1.5 dB predicted by theory for Gaussian-distributed analog sources [10, Sec. 5.1]. Wilson [15], using adaptive tree encoding, concludes that the performance of his system at 8K bits/s is as good as a non-adaptively encoded system at 16 K bits/s. Tests with a hardware tree speech encoder of rate 16 K bits/s have yielded telephone quality speech with an SNR of 18 dB and a high of 22-24 dB for voiced sounds [51].

Code searching schemes are finding increasing applications in source coding and processing pictures as well [35]-[38]. Using the M-algorithm to encode synthetic

2-D autoregressive random images, Modestino et al. [36] demonstrate that the algorithm achieves performance close to the rate-distortion bound. The single stack algorithm has been used to encode 2-D binary sources similar to facsimile images [37]. An issue here is whether to utilize 2-dimensional code searching; Stuller and Kurz [38] show that the use of 2-D code searching gains 1.2 to 2.7 bits per picture element (pixel) over the corresponding 1-D independent coding of line scans. Aside from source coding, the Zigangirov-Jelinek stack algorithm has recently been applied to the contour extraction problem [35].

Sequential decoding algorithms have found applications primarily in satellite and space communications [39]-[41]. Use of sequential decoding results in a coding gain of up to 7 dB [42]. Sequential decoding has also been successfully applied to low frequency (about 75 bits/s) and low SNR submarine communication [43].

Several other transmission-related applications have been reported in the literature. Code distance properties of convolutional codes, such as the minimum free distance d_{free} and column distance functions, have been analyzed using sequential decoding algorithms [44], [45]. Similarities between the maximum-likelihood Viterbi decoding and dynamic programming and also between the Viterbi decoding and shortest path problems of graph theory are well

known [20]. Recently, tree coding algorithms have also been formulated as mathematical programming problems and similarities have been pointed out between them and the branch and bound problems [46]. These applications point to their use not only in source coding and sequential decoding but also in a wide variety of other fields.

1.6 An Overview of the Thesis

This thesis is motivated by the increasing application of code tree search algorithms and the need to devise efficient methods. Chapter 2 surveys some existing algorithms. In Chapter 3 the inadequacy of node computation as a measure of sequential coding efficiency is pointed out and a cost measure based on the size of and number of accesses to storage is proposed.

In Chapters 4, 5, and 6 three new code tree search algorithms are proposed; the first uses multiple side lists and efficient merge techniques, the second uses a height-balanced tree data structure, and the third uses dynamic hashing concepts that provide modifications to the Jelinek bucket algorithm. Resource costs derived for these algorithms point to their cost effectiveness.

Chapter 7 analyzes an existing algorithm, the single stack algorithm, using the theory of multitype branching processes in random environments and derives an equation for

node computation. Chapters 8 and 9 report simulation results for both theoretical and "real life" sources and bring out the similarities between the results. Simulation results are used to optimize the performance of algorithms with respect to storage, execution time, and node computation.

CHAPTER 2

A SURVEY OF EXISTING CODE TREE SEARCH ALGORITHMS

2.1 Introduction

Code searching schemes may be classified as sorting or non-sorting, and as depth-first, breadth-first, or metric-first, where the "metric" is some measure of fidelity or likelihood. A number of schemes are summarized in Table 2.1. Among algorithms which sort, the well-known stack algorithm (see [26] or [27] for channel decoding or [17] for source encoding) extends code tree paths in a purely metric-first manner, meaning that the next path extended is always the one with the best metric. Sorting is used to single out the best path. The usual method is an ordering procedure. A purely breadth-first algorithm that sorts is the M-algorithm. This algorithm views all branches at once that it will ever view at a given depth, then sorts out and drops paths ending in certain branches before continuing on. Another sorting scheme is the bucket algorithm [27].

A second class of algorithms does not sort; that is, paths are never compared with one another. The simplest such method is the single stack algorithm, a purely depth-first method suggested by Gallager [18]. This scheme

Table 2.1: Search Rationales for Certain Selective Search Algorithms

	Metric-First	Breadth-First	Depth-First
Sorting	Stack Alg.	M-Alg.	
	Bucket Alg. (roughly)		
	Merge Alg.*		
	Generalized* Merge Alg.		
	AVL-Based Alg.*		
	Dynamic Bucket Alg.* (roughly)		
	-- Haccoun's Alg. (both) --		
	xx——— Multi. Stack Alg. ———xx		
Non-Sorting			Single Stack Alg.
			Fano Alg.
		-- 2-Cycle Alg. (both) --	

The asterisks indicate algorithms proposed in this thesis.

simply pursues a path until its metric falls below a discard criterion, and at any one time it stores the identity of only one path. A direct implementation is a single push-down stack. (It should be mentioned that the more widely known "stack algorithm" cited above in fact contains no stack, but only a list.) A familiar variation of the single stack method is the Fano algorithm; the peculiar character of this search stems from its method of computing the discard criterion. A more sophisticated non-sorting procedure is to set aside certain good paths for later attention as they appear in the depth-first search. This method stores a number of paths, but they are known to be good ones; an example is the 2-cycle algorithm [16].

All of the above methods are selective search algorithms, which leave some, and usually most, paths unviewed. The Viterbi algorithm [19] is an exhaustive search, which considers all possibilities inherent in the code by exhibiting all the states of the code generation structure; its cost is simply a constant times the number of generator states. Uddenfeldt and Zetterberg [48] have discussed a depth-limited exhaustive search. Here, we shall treat only selective search algorithms.

2.2 Basic Features of Searching Algorithms

It will be convenient first to define features which are common to all algorithms.

The aim of a source encoder search is to find a path with distortion as close as possible to Shannon's distortion-rate function $\Delta(R)$, where $R = \log_2 b/\beta$ is the rate of the code tree with b branches out of each node and β symbols on each branch. Searching begins at a root node and continues until some path reaches depth L. The algorithm then decides once and for all which first branch to release as output; the end node of this branch becomes the new root node. Searching resumes until some path again reaches total length L branches. The procedure continues indefinitely in this "sliding block" fashion, releasing some branch at depth ℓ and accepting a new data group at depth ℓ+L. (Such a sliding block search should not be confused with a sliding block code.) An older attitude toward searching is the "block" search, in which an L-branch path is released all at once and the search begins anew at the path's end node. We give no separate analysis for this alternative, although most of our conclusions apply.

Paths are described by path maps made up of b-ary symbols, {0, ..., b-1}, one for each branch. The code word letters on a path's terminal branch are somehow computable from its path map. Associated with each map is a metric μ,

either a likelihood of the path or a measure of its distortion, and sometimes an indication of the path's length or pointers to other storage locations. The metric is defined in (1.2) for source coding and in (1.5) to (1.7) for sequential decoding.

The following primitive functions are performed by all algorithms and will be enclosed throughout within angle brackets.

<Extend Path>: The algorithm extends a path one branch forward, "viewing" the branch. Viewing includes calculating the code word symbols on the branch, fetching the input data group corresponding to its depth (source symbols to be encoded or channel symbols to be decoded), calculating the metric increment for the branch, and forming the new metric total for the path. Branches are viewed either singly or in groups of b, depending on the algorithm. In the former case, only a single branch, say the 0-th, is viewed during the first visit to the original path's end node; if there is a later visit, the 1-st will be viewed, and so on until all b branches are viewed. Other algorithms <Extend b Paths>, and view all b at once.

<Ambiguity Check>: The algorithm checks all eldest path map symbols to determine if they are consistent with the symbol released as output. If a symbol is not, the path must be deleted. Ambiguity checks are necesary for two reasons. If

a path fails to check but is kept in storage, it may later be released as output even though its antecedent does not match earlier output. The encoder and decoder will then not develop the same path. Ambiguity checks also prevent the accidental storage of two paths with the same symbols. If two path maps once differ (as they do initially), they can become identical only when the differing symbols are deleted, and this is forewarned by the check. A separate ambiguity check is unnecessary in non-sorting algorithms, which store only a single path.

<Delete Path>: The algorithm deletes an entire path map. A deletion must occur whenever the number of paths stored exceeds S, and whenever a path fails an ambiguity check.

<Release Output Symbol>: The algorithm releases as output the earliest symbol of the best path map it has. In sorting algorithms, this triggers an ambiguity check to make certain that all path maps have the same symbol at this depth.

In non-sorting algorithms, the possibility exists that no path satisfies the constraints of the algorithm, an event we call <algorithm failure>. The cost of recovering from this event is small. The easiest method is to move one branch forward of the root node, declare the branch's end node to be the new root node, and start again; another method leading to better performance is to save the longest path ever viewed, and declare its end node to be the new

root node.

In describing the algorithms here and throughout, we use a structured language similar to that found in [49]. A procedure has the following format.

```
Procedure <name of procedure>
   begin
      statements
   end
end <name of procedure>
```

A statement may be a simple assignment statement such as i ← j (i is assigned the value of variable j) or any one of the following four control statements.

```
1)  If condition
       then statement a
       else statement b
    end if
2)  for i = initial value to final value
            in steps of increment do
        statement
    end for
3)  while condition do
        statement
    end while
4)  do until condition
        statement
```

end do until

In 1), statement a is executed if the condition is true and statement b if it is false. 2) is similar to the DO statement of FORTRAN. In 3), the statement following do is executed so long as the condition remains true. In 4), the statement is first executed and the condition evaluated. So long as the condition is false, the statement is repeatedly executed. Several statements may be enclosed between begin and end in order to avoid ambiguity. This feature also facilitates nesting of statements.

2.3 Non-Sorting Algorithms

The distinguishing feature of non-sorting schemes is that they store only a single path, extending or backtracking along the path in response to the value of the path metric.

The Single Stack Algorithm

This algorithm [18] proceeds depth-first directly through the code tree until the path metric falls below a discard criterion B; the search then backtracks to the first untried branch and proceeds depth-first again. The symbols of the path map are stored in a push-down stack, and in the steady-state operation of the algorithm, an output path map symbol is forced out the bottom whenever a depth is visited

for the first time.

By convention we assume that on first visiting a node, branch 0 out of the node is viewed, on the second visit branch 1, and so on until all branches are viewed, at which time the search must backtrack. An example of this routine appears in Fig. 2.1 for b = 2; X's indicate a path that has fallen below the discard criterion. It is more straightforward to think of the discard criterion B as a constant, although Davis and Hellman [50] show that B probably must be a function of the input data. For these cases, a constant B may increase the node computation or prevent the scheme from quite achieving the distortion-rate function.

Using a single push-down stack and a comparison to a discard criterion, the single stack algorithm is the simplest of all search algorithms.

```
Procedure <SS>
   begin
      i ← metric (root node) ← 0;
      STACK ← empty stack;
      STACK <= (metric (root node), path map of root node)
      While not all source symbols are encoded do
         begin
            While length of path in the stack < L do
```

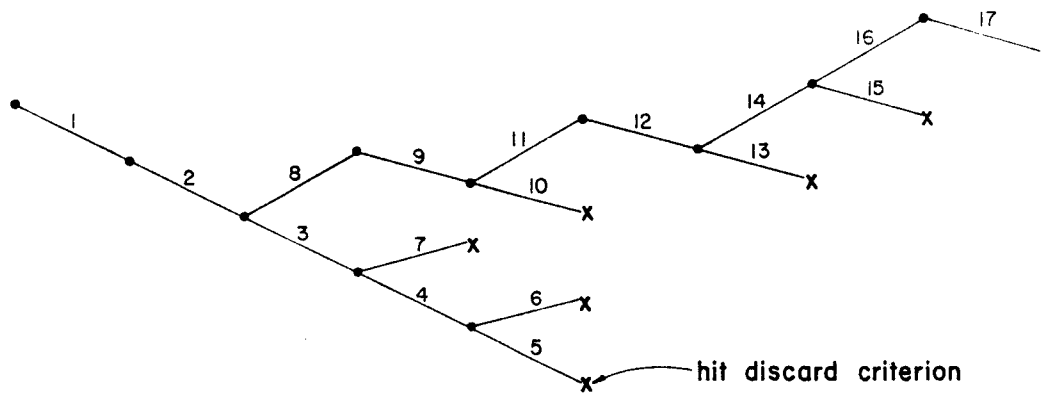


Figure 2.1 Push-Down Stack Search, b = 2. Downward Branch (0-th) Taken First, Then Upward Branch (1-st); Numbers Show Order of Visiting, x Means Path Metric Hits Discard Criterion

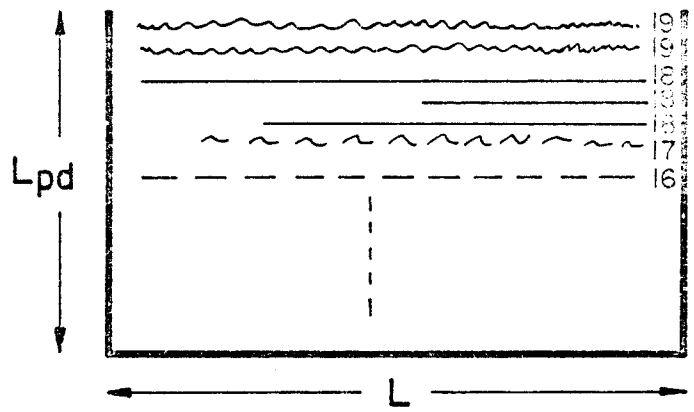


Figure 2.2 Save Stack Showing Generation Numbers

```
                begin

                   <extend path> whose map is in the stack,
                                 viewing branch i out of its
                                 end node; update μ, the
                                 metric of the path in the
                                 stack;

                   if μ > B then STACK <= (μ, i), i ← 0

                      else do until i < (b-1)

                            if STACK not empty

                               then (μ, i) <= STACK

                               else declare <algorithm
                                             failure>

                            end if

                           end do until

                           i ← i+1

                   end if

                end

             end while

             <Release output symbol>

          end

       end while

    end

end <SS>
```

In the above procedure the statement STACK <= (μ, i) is used to indicate that the tuple (μ, i) is pushed onto the stack; (μ, i) <= STACK denotes popping of the stack.

The 2-Cycle Algorithm

Algorithms similar to the single stack procedure search depth-first, backtracking only when the path falls below the discard criterion. Another method which uses the same stack structure is to search all paths lying above the criterion and having length $\ell \leq L$. Of these, only a much smaller set of "good" paths, say those with metric $\mu \geq A$, are saved for later attention. This procedure has been called the 2-cycle algorithm [16].

In viewing all paths for which $B < \mu < A$, the search (the "barrier cycle") may proceed either depth or breadth-first, but only a depth-first search uses the simple push-down stack structure. Whenever the path in the stack does penetrate the save criterion A, it is copied into a second "save" list. When the present barrier cycle search terminates, a new cycle begins from the end node of the top path in the save list. In doing so, the algorithm seeks to concatenate another good path onto an old one, and form in a depth-first manner a long chain of good paths.

The logical ordering of the save list is last-in first-out, so that this list too is a push-down stack, with entire path maps of up to L symbols as entries. The entries must be properly linked together; this can be done by pointers, but a simpler way is to identify the generation number of each path. A path in generation g is at the end

of a chain of g good paths, produced by g barrier cycles. The push-down storage regime will create a generation ordering like that shown in Fig. 2.2; a moment's thought will show that the generation 19 paths must all stem from the last saved path at generation 18, and that all the 18's stem from the single path at 17. In case barrier cycles forward of all the 19's fail to produce good paths, the top 18-th generation path must be deleted, and new 19-th generation paths attempted from the second 18-th.

If the save stack is finite with L_{pd} entries, each stacked path forces another out the bottom once the stack is full. The bottom path is discarded unless it is the last of a generation, in which case it is released as output.

```
Procedure <2-Cycle>
  begin
    Barrier stack ← Save stack ← empty stack;
    i ← 0, g ← 1;
    Barrier stack <= (metric (root node), path map of root
                                                    node)
    While not all source samples are encoded do
      begin
        While length of path in the barrier stack < L do
          begin
            <extend path> whose map is in the barrier stack,
            viewing branch i out of its end node; update μ;
            if A > μ > B then barrier stack <= (μ, i) i ← 0
```

```
else if μ ≥ A
      then
        begin copy barrier stack path into save
              stack with generation g
              do until i < (b-1)
                begin
                  if barrier stack not empty
                    then (μ,i) <= barrier stack
                    else while save stack not empty
                              and barrier stack
                              empty do
                           begin
                             if generation =g
                               then
                                 Barrier Stack <=
                                      save stack
                                 g ← g+1, i ← 0
                               else <delete path>
                                    g ← g-1
                             end if
                           end
                         end while
                         if barrier stack empty
                           then declare
                            <algorithm failure>
                         end if
                  end if
              end
```

```
                    end do until
                    i ← i+1
                end
            end if
        end if
    end
  end while
  <Release output symbol>
 end
end while
end <2-Cycle>
```

2.4 Sorting Algorithms

Sorting schemes compare paths on the basis of metric in order to decide which to extend and which to delete. These algorithms view fewer branches than non-sorting algorithms, but the cost of sorting is often very high.

The Stack Algorithm

As mentioned previously, the stack algorithm is based not on a stack, but on a list of code tree paths. The usual view is that this is an ordered list; the next path extended is always the best in terms of μ, and sufficient worst paths are deleted to keep the list at length S paths. An alternate view is that new paths are simply appended, and

that the list is probed for its best entry prior to each extension and for its worst entry prior to each deletion. The cost is similar in either case, and we shall take as the defining attributes for the stack algorithm simply a single list and metric-first extensions and deletions.

Since a path's metric indicates the likelihood that the best path in the code tree lies ahead of it, it comes as no surprise that this metric-first procedure appears to find a path at a given metric level with the least node computation of any scheme (see [17] or [27]). Despite this, the space-time cost of the stack algorithm seems to exceed that of any other scheme for binary sources; we shall see that the metric-first procedures proposed in Chapters 4, 5, and 6 have a much lower cost.

In the stack algorithm's list, paths vary in length. Once the algorithm reaches a steady state, an ambiguity check must be performed whenever the length of a path in storage exceeds L. One can show that this occurs whenever a tree depth is reached for the first time. Fig. 2.3 shows a list data structure in which each entry consists of three subwords, a path metric, an indication of path length, and a path map. The path maps are left justified, and to find an end node, the length subword must be consulted. All paths end on the left at a point L branches before the deepest tree penetration; during an ambiguity check all these

earliest branch symbols must be checked to see if they agree with the symbol released as output.

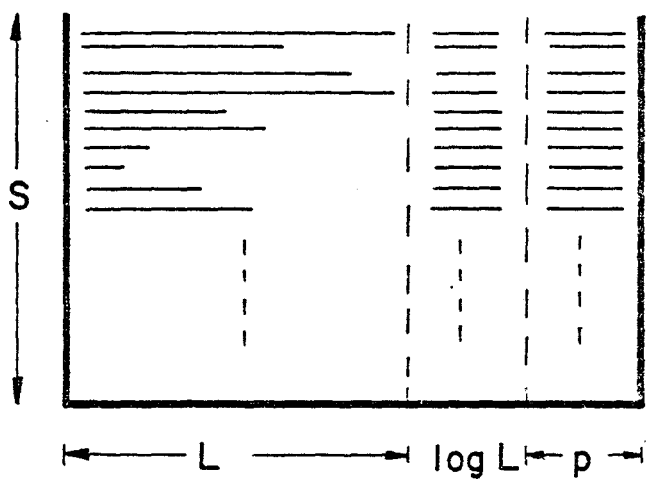


Figure 2.3 Example of Stack Algorithm List, Showing Paths, Length Indicators, and Metrics. The Top Path is About to Penetrate a New Depth, Causing an Ambiguity Check; the Fourth Path will be Deleted if its Earliest Symbol does not Pass

```
Procedure <Stack>
   begin
      list ← path map of root node;
      metric (root node) ← 0;
      While not all source symbols are encoded do
         begin
            While length of top path in the list < L do
               begin
                  <extend b paths> from the end node of the
                  topmost path in the list;
                  <Delete> this path;
                  <Order> into the list the b new paths;
               end
            end while
            Perform <ambiguity check>
            <Release output symbol>
         end
      end while
   end
end <Stack>
```

The M-Algorithm

We conclude with a breadth-first sorting algorithm. In general, such a procedure views all branches at depth ℓ that it will ever view, deletes paths according to some criterion, and then moves on to the next depth. The

M-algorithm [12] deletes all paths except a fixed number M. Breadth-first searches are synchronous (that is, all paths have the same length) and they are effective at low intensities of searching; they are thus good candidates for practical application [51], [15].

In its specific operation, the M-algorithm moves forward by extending the M paths it has retained to form bM new paths. All the terminal branches are compared to the input data corresponding to this depth, metrics computed, and the (b-1)M poorest paths deleted.

```
Procedure <M-Alg>
   begin
      Obtain root node;
      metric (root node) ← 0;
      While not all source symbols are encoded do
         begin
            While length of the retained paths < L do
               begin
                  <Extend b paths> from each retained path;
                  save these in the list;
                  <order> the list to find the best M paths;
                  <Delete> the remaining paths;
               end
            end while
            Perform <ambiguity check>
```

```
            <Release output symbol>
          end
        end while
      end
end <M-Alg>
```

2.5 Conclusions

We have surveyed three pure form algorithms (single stack, stack, and M-) and one (2-cycle) that combines breadth-first and depth-first techniques. The bucket algorithm is described in Chapter 6. The algorithms of Chevillat and Costello [29] and Haccoun and Ferguson [28] are used for decoding convolutional codes and are not surveyed here. Ng et al. [52] have described another sequential decoding algorithm.

CHAPTER 3

A COST FUNCTION FOR CODE TREE SEARCH ALGORITHMS

3.1 Introduction

The usual measure of efficiency for code searching algorithms has been the node computation, the number of branches visited during the progress of the scheme divided by the branches released as output, for a given level of source encoder fidelity or decoder probability of error. As the algorithms have come into more use, however, it has become clear that this is not a sufficient measure. Several authors ([53], [54]) have found that stack algorithm source encoding is exceptionally time consuming, even though it has the least node computation of any known method. Experience with M-algorithm hardware speech encoders [51] shows that this method is efficient despite a poor node computation. In sequential channel decoding and in sequence estimation, the situation is similarly confused. The Viterbi algorithm finds wide use despite its exhaustive nature. The selectively searching algorithms which should be more efficient suffer erasures brought on by computation overload, and find only occasional use in applications like deep space communication. What seems to be missing here is

a factor to account for the size and complexity of the required information structures, in addition to the intensity of their use. Here we develop a systematic measure of cost for code tree search algorithms, and use it to compare them.

3.2 A Definition of Algorithm Cost

A more realistic measure of algorithm cost can be based on the number of storage elements in a scheme and the number of accesses to them. The space complexity of an algorithm is the size of resources that must be reserved for its use, while the time complexity counts the number of accesses to this resource. We shall consider the product of these two, the space-time complexity, as our primary measure of a scheme's total cost.

A space-time product cost measure assumes that storage blocks "wear out" after a certain number of accesses and that the cost of blocks is proportional to their speed, assumptions that are roughly true for physical devices. Parallel processing is of no benefit under this measure, since there is no gain in trading space for time. A second measure of cost, more suited to software implementations, is the space + time complexity. The sum of space and time, this measure stresses more the opportunity cost forgone by assigning resources to a user. Different constants are

often placed before the two components, but these will have no asymptotic significance. We shall list results for both measures.

A perhaps more traditional measure of complexity for sorting methods is the number of comparisons, but this measure does not account for both space and time, and as mentioned previously, not all code searching algorithms sort.

Other measures of coding algorithm cost could be proposed than the space and time cost of storage blocks, but this kind of measure relates closely to the nature of such algorithms. Code search algorithms basically move in and out of storage data about code tree paths. Other tasks, such as computing metrics, checking for ambiguous output symbols, and generating code word letters, form a constant multiplier on the cost of storage access. The major determinants of cost remain the storage size and the pattern of accesses called for by the steps of the algorithm.

A simple building block for algorithm storage (and thus for algorithms) is the random-access memory (RAM), but it is interesting to observe that a simpler structure, the push-down stack, can generally be used. Many algorithms relate more directly to push-down stacks, and all but one (the M-algorithm) have the same asymptotic complexity based on them as on RAM's. We shall thus base our discussion in

the first instance on push-down stacks.

Three variables dominate the asymptotic cost of the algorithms we analyze, the length of path an algorithm can retain, L, the number of paths it can retain, S, and the expected node computation already defined, E[C]. We assume that L and S are finite and fixed in value. It is important to realize that algorithms differ significantly when these are so constrained; when the number of paths exceeds S, for instance, some mechanism must delete excess paths, and whenever a path exceeds length L, its oldest branch must be checked to insure that it is consistent with other paths kept to this depth. These routines may change the asymptotic cost.

3.3 A Cost Analysis of Algorithms

We turn now to a space and time cost analysis of the algorithms surveyed in Chapter 2. For clarity, we emphasize sequential source encoding schemes throughout, although the analysis applies as well to channel decoding and sequence estimation. The algorithms chosen for exposition are those which differ from each other in fundamental ways, or which demonstrate a principle in pure form. Often, a variation or a scheme combining several principles will be most effective in applications.

The Single Stack Algorithm

The space cost of the single stack algorithm is L b-ary symbols (plus a small overhead for side registers and code letter generation). The time cost is upperbounded by 2 accesses/branch viewed, since the algorithm scrutinizes a branch once during forward motion in the code tree and at most once when backtracking. The space-time product cost is thus

$$O(L\ E[C_{SS}])\quad \text{access-symbols/output branch} \tag{3.1}$$

where C_{SS} denotes the node computation of the single stack algorithm, and here and throughout $f(x) = O(g(x))$ means that $|f(x)/g(x)|$ remains bounded (we say that $f(x)$ grows no faster than $g(x)$, or that $f(x)$ is "big oh of" $g(x)$, asymptotically).

The Fano algorithm is a variant of the single stack procedure in which the discard criterion varies up and down as a function of the path metric. Searching proceeds depth-first until a path falls below B, but B is raised in increments whenever possible during visits to new nodes, and is lowered when necessary during returns to previously visited nodes. The cost of the Fano algorithm is again $O(L\ E[C_{FA}])$, but it is not clear which of C_{FA} and C_{SS} is the larger. The opportunistic changing of B undoubtedly reduces the set of nodes visited, but the algorithm may visit certain of the nodes many times. Nonetheless, if C_{FA} can be

measured, then the form (3.1) gives the total cost.

The 2-Cycle Algorithm

In this algorithm, costs stem from two sources. In the barrier stack, we have as in the single stack algorithm

$$\begin{aligned} &\underline{\text{time cost}}: && \text{about } 2\,E[C_{2C}] \text{ accesses/branch released} \\ &\underline{\text{space cost}}: && L \text{ b-ary symbols} \end{aligned} \tag{3.2}$$

In the save stack,

$$\begin{aligned} &\underline{\text{time cost}}: && N_A + 1 \text{ accesses/barrier cycle} \\ &\underline{\text{space cost}}: && L_{pd}\,(L + \log L_{pd}) \text{ b-ary symbols} \end{aligned} \tag{3.3}$$

where N_A is the number of paths that have $\mu \geq A$ during a barrier cycle. In the parameterization of the 2-cycle algorithm, EN_A is set near 1, so that the space-time cost incurred in the save stack will be smaller than that incurred in the barrier cycles if the expected number of branches viewed in a barrier cycle is less than L_{pd}. Thus far, experiments (see [16]) indicate this to be true.

The space-time cost of the 2-cycle algorithm is then still

$$O(L\ E\ [C_{2C}])\ \text{ access-symbols/branch released} \tag{3.4}$$

as it was for the single stack and Fano algorithms.

The Stack Algorithm

Most of the cost of this algorithm resides in the ordering procedure (see Chapter 2). Assume the list is instrumented with two push down stacks, the first containing the ordered list, and the second functioning as a scratch pad. To insert a path in order, the first stack is poured into the scratch stack until the location of the new path appears; then the new path is stacked in the first stack, and the scratch stack poured back. During the pour-back, excess paths are automatically deleted out the bottom of the first stack. For this ordering,

$$\begin{aligned} &\underline{\text{time cost}}: && \text{(average) } S \text{ accesses/branch viewed} \\ &\underline{\text{space cost}}: && 2S\,(L + \log L + p) \text{ b-ary symbols} \end{aligned} \tag{3.5}$$

where log L and p are the storage required for length and metric information of paths, respectively. Other steps of the algorithm form a constant overhead except for the ambiguity check, which can be combined with the deletion procedure. In other implementations it cannot, but since the check occurs with each depth penetration rather than each path extension, while still having cost similar to (3.5), its cost is of lower order and can be neglected.

Total space-time complexity is thus about $2S^2(L + \log L + p)E[C_{SA}]$, or (since p is small) $O(LS^2E[C_{SA}])$ asymptotically, where C_{SA} is the node computation of the stack

algorithm. We have continued the use of push-down storage in the implementation for consistency with the other algorithms, but a RAM-based cost estimate is similar.

Unless E $[C_{SA}]$ is very small, the added S^2 factor will make this cost much larger than that of the non-sorting algorithms. (We shall consider the experimental evidence in Chapter 10 and see that this is indeed so.) All research studies on the stack algorithm have reported computational difficulties in its use. Jelinek [27] suggests a chained storage scheme for the path maps but it is doubtful this will change the asymptotic cost. He also suggests alternative algorithms, a combining of the Fano and stack algorithms ([27], pp. 682-ff), and a bucket algorithm. The latter is a basically new scheme to which we return in Chapter 6.

The M-Algorithm

The cost of the M-algorithm is (set S = M)

$$\begin{aligned} &\underline{\text{time cost}}: \quad k''S \log S \quad \text{accesses/branch released} \\ &\underline{\text{space cost}}: \quad (k' + L + p)bS \quad b\text{-ary symbols} \end{aligned} \tag{3.6}$$

O(S log S) is the number of comparisons done by RAM-based sorting methods such as Mergesort and Heapsort ([55], Chap. 5). Here, k' and k" are small overheads to account for sophistications required in procedures like Mergesort. Implemented in push-down stacks the algorithm has space and

time cost $2(L + p)S$ and bS^2. In either case there is no separate term for node computation, since this is automatically bM. Asymptotically, the total cost is $O(LS^2 \log S)$ access symbols/branch released for RAM storage.

3.4 Conclusions

Results are summarized in Table 3.1. It is clear that there are dramatic differences among the algorithms; all depend linearly on L, but some, like the single stack algorithm, have no dependence on S, while others range as high as $S^2 \log S$. As has been shown in earlier work, there are also wide variations in E[C], and the M-algorithm in particular has no separate dependence on this factor.

It was not our intention to optimize over the choice of the three major cost factors, or over the many lesser factors and parameter settings, but only to establish the cost functions. An accurate optimization would require much further work. The experimental work thus far available does allow certain speculations about an "optimal" algorithm, however, and these appear in Chapter 10. A survey of existing code search algorithms and their resource costs were reported in [56], [78].

Table 3.1: Asymptotic Cost of Certain Algorithms in the Limit of Intensive Searching, per Output Symbol Released. Space in b-ary Symbols, Time in Accesses to Storage. C = Branches Visited/ Output Symbol Released, L = Length of Retained Paths, S = Number of Retained Paths.

Algorithm	(Space)(Time)	(Space)+(Time)
Single Stack	$O(LE[C_{SS}])$	$O(L+E[C_{SS}])$
Fano	$O(LE[C_{FA}])$	$O(L+E[C_{FA}])$
2-Cycle	$O(LE[C_{2C}])$	$O(LL_{pd}+E[C_{2C}])$
Stack	$O(LS^2E[C_{SA}])$	$O(LS+SE[C_{SA}])$
M-	$O(LS^2 \text{ Log } S)$	$O(LS + S \log S)$
Bucket	$O\{L(SE[C_{SA}]+S^2\}$ $+ H_J$	$O(LS)+O(S+E[C_{SA}])$ $+ H_J$
Merge*	$O\{L[S^{4/3} E[C_{SA}]]\}$ $+ O(S^2)$	$S + O\{(LS)^{1/2}$ $(1 + E[C_{SA}])\}$
Generalized Merge*	$L[\{O(S^{2^n/(2^n-1)}$ $+ (n-1)S\} E[C_{SA}]$ $+ S^2 + \Sigma_{i=2}^{n} A_i^2]$	$O[(L+1)^{(n-1)/n}$ $S^{1/2}] E[C_{SA}]$ $+ S + \Sigma_{i=2}^{n} A_i$
AVL-Based*	$O(LS \log S E[C_{SA}])$ $+ O(LS^2)$	$O(LS) + E[C_{SA}]$ $O(\log S)$
Dynamic Bucket*	$O(LSE[C_{SA}])$ $+ O(LS^2) + H_d$	$O(LS) + O(E[C_{SA}])$ $+ H_d$

The asterisks indicate algorithms proposed in this thesis

CHAPTER 4

THE MERGE ALGORITHM

4.1 Introduction

We propose here a new algorithm, called the merge algorithm, that has a greatly reduced resource cost compared to the stack algorithm. The merge algorithm uses in addition to a main list of size S, which serves the same purpose as in the case of the stack algorithm, an auxiliary list of size T to store paths. Both lists are of width L symbols. Sorting the much longer main list has made the metric-first stack algorithm so expensive to implement. By using a shorter auxiliary list to order paths and merging it periodically with the main list, the merge algorithm reduces the resource cost of the stack algorithm.

In this chapter, by merging we refer to the following process. Let there be two sorted arrays $x_1 \leq x_2 \leq x_3 \leq \ldots \leq x_m$ and $y_1 \leq y_2 \leq \ldots \leq y_n$. The merging process merges these two arrays into a single sorted array $z_1 \leq z_2 \leq \ldots \leq z_{m+n}$. A simple algorithm to perform the above operation is as follows.

```
Procedure <Straight Merge>
     x_{m+1} ← y_{n+1} ← ∞
     i ← j ← 1
     for k = 1 to m+n do
          if x_i < y_j
               then z_k ← x_i, i ← i+1
               else z_k ← y_j, j ← j+1
          end if
     end for
end <Straight Merge>
```

The above procedure requires m+n comparisons. Another merging algorithm known as binary merging [55, pp. 205-206] requires at most $\lceil \log \binom{m+n}{m} \rceil + \min(m,n)$, where the notation $\lceil x \rceil$ denotes the smallest integer N such that $N \geq x$. We assume the use of straight merge algorithm in computing the costs of the merge and the generalized merge algorithms.

We describe the merge algorithm for code tree searching below and in the succeeding sections derive its resource cost. The generalized merge algorithm is then described followed by its cost derivations.

4.2 The Merge Algorithm

```
Procedure merge
   [initialize]
   Assign root node to the main list;
```

```
metric (root node) ← 0;

[encode]

While not all source samples are encoded do

   begin

      While the auxiliary list can take in b more
      paths do

         begin

            If the lengths of the top paths in the main
            list and the auxiliary list < L, the width of
            the list

               then

                  begin

                     If the metric of the top path in the
                        main list < the metric of the top
                        path in the auxiliary list

                        then <extend> the top path from
                           the main list and <delete>
                           this path

                        else <extend> the top path from
                           the auxiliary list and
                           <delete> this path

                     end if

                     <Order> the newly extended paths
                     into the auxiliary list

                  end

               else perform <ambiguity check> and
                    <release ouput symbol>;

            end if

         end

      end while
```

```
            Merge the auxiliary list with the main list; {the
            auxiliary list is now empty and a new merge cycle
            starts all over again}
        end
    end while
end <merge>
```

The above procedure extends, at each time instant, either the top path from the main list or that from the auxiliary list, whichever has the largest metric. This makes the algorithm strictly metric-first. All the extended paths are reordered into the auxiliary list. Once the auxiliary list is full, it is merged with the main list. After merging, only the best S of the S+A paths remain in the main list and the worst A are dropped. The auxiliary list is now empty and is ready to take in the next set of A paths.

4.3 Cost of the Merge Algorithm

Expressions for the different cost measures, proposed in Chapter 3, are now derived for the merge algorithm. It will be shown that the optimal size of the auxiliary list is influenced by the cost criterion chosen for minimization. In the following, symbols S and A refer to the sizes of the main and auxiliary lists respectively; L is the width of the lists. Expressions for the comparisons-based and product cost measures of the merge algorithm appeared in Anderson

and Mohan [56].

Comparisons-Based Cost Measure

Define first a merge cycle as the operations done in extending paths from the main and auxiliary lists, filling the auxiliary list, and merging it with the main list. The total number of comparisons done in a merge cycle is then computed as follows. Ordering paths into the auxiliary list involves about $A^2/2$ comparisons. Merging the two lists requires another S+A comparisons. Comparing the two top paths in order to make the search metric-first requires another A comparisons. Thus a total of $S + 2A + A^2/2$ comparisons are done in a merge cycle. Note that at least A branches are viewed in a merge cycle. More than A may have been viewed since paths from the auxiliary list may also be extended. Dividing the expression for the total number of comparisons by A gives a tight upperbound of $S/A + 2 + A/2$ comparisons per branch viewed. Minimizing the above expression with respect to A yields $A \approx \sqrt{2S}$ for an optimal auxiliary list size.

Substituting the optimum A into the expression $S/A + 2 + A/2$ yeilds $\sqrt{2S} + 2$ comparisons per branch viewed for the merge algorithm compared to the S required for the stack algorithm. The increase in storage is $\sqrt{2S}/S$, an asymptotically insignificant factor. Introducing the term

for ambiguity check yields a total cost of approximately

$$\simeq (\sqrt{2S} + 2)\, E[C_{SA}] + S \text{ Comparisons/branch released} \qquad (4.1)$$

The reason for the significant reduction in the number of comparisons lies in the ability of the merge algorithm to take advantage of the already ordered main list. Note that reordering the extended paths directly into the much longer main list would dramatically increase the cost, since S accesses would be required for every branch extended. On the other hand, the merging strategy requires a cost of $\simeq 2S$ for every $\sqrt{2S}$ branches viewed or a cost of $\simeq\sqrt{2S}$ for every branch viewed.

Space-Time Product Cost Measure

In a merge cycle there are $A^2/2$ accesses in ordering a size LA storage. Finding the best path to extend next by comparing the two top paths in the main list and the auxiliary list requires A accesses to a size LS storage, and A accesses to a size LA storage. Merging the two lists requires S accesses to LS, A to LA, and (S+A) accesses to L(S+A). The total space-time product cost for a merge cycle is

$$\frac{LA^3}{2} + 3LSA + 3LA^2 + 2LS^2.$$

Dividing the above expression by A yields $LA^2/2 + 3LS + 3LA + 2LS^2/A$ as the product cost per branch viewed. Minimizing the above expression with respect to A yields $A \simeq (\sqrt{2}\ S)^{2/3}$ for an optimum auxiliary list size. Minimized product cost is about $(2.38)\ LS^{4/3}$ per branch viewed.

Note that the ambiguity check is performed each time a symbol is released as output. Cost of performing this check is $L(S^2+A^2)$ or $L(S^2+(\sqrt{2}\ S)^{4/3})$. Inserting this term and accounting for the storage of p and log L bits taken up by the metric and length arrays, respectively, yield a product cost of about

$$(L + \log L + p)\ [(2.38)S^{4/3}\ E[C_{SA}] + S^2 + (\sqrt{2}\ S)^{4/3}]$$

which is asymptotically

$$L[2.38\ S^{4/3}\ E[C_{SA}] + S^2] \text{ access-symbols per branch released,} \qquad (4.2)$$

where $E[C_{SA}]$ is the expected number of tree branches viewed per source symbol encoded by the stack algorithm. Since both the stack and the merge algorithms are strictly metric-first, E[C] is the same in both cases.

Space-Time Sum Cost Measure

The total space plus time cost in a merge cycle is $LS + LA + 2S + 4A + A^2/2$. The cost per branch viewed is $LS/A + L + 2S/A + 4 + A/2$. Minimizing the above expression yields an optimal $A \simeq (2LS)^{1/2}$. Substituting for A into the cost

expression, we get

$$\sqrt{2LS} + L + \sqrt{2S/L} + 4$$

which is

$$\simeq (2LS)^{1/2}, \text{ if } S >> L$$

or

$$\simeq (2LS)^{1/2} \, E[C_{SA}] \text{ per branch released as output} \qquad (4.3)$$

Adding the term for the ambiguity check yields a sum cost per branch released of

$$\simeq \{S + (2LS)^{1/2} \, (1 + E[C_{SA}])\} \qquad (4.4)$$

In order to compare the above three cost measures consider an example. Suppose S = 100. In order to minimize the number of comparisons, the product cost, and the sum cost one requires optimal auxiliary list sizes of 14, 27, and 71, respectively. Of the three methods of optimizations, the comparisons-based one demands the smallest sized auxiliary list and the sum cost the largest. The minimized costs are 14 $E[C_{SA}]$, L[1100 $E[C_{SA}]$ + 10,000], and {100 + 14 $L^{1/2}$ $E[C_{SA}]$}, respectively. The corresponding costs for the stack algorithm are, respectively, 100 $E[C_{SA}]$, 20,000 L $E[C_{SA}]$, and 100 L + 200 $E[C_{SA}]$. But with any measure, it is clear from the comparisons that the merge algorithm greatly reduces the asymptotic costs.

4.4 The Generalized Merge Algorithm

Next we generalize the 2-list merge algorithm (MA) of section 4.2 as follows. Recall that the MA used a side list, the auxiliary list, in order to fill the main list by periodically merging the auxiliary list with the main list. Next we ask if the resource costs of the MA can be further reduced by providing a third list which will be periodically merged with the second list. When the second list becomes full, it is merged with the main list. This provision of successive side lists when carried to its limit emerges as the generalized merge algorithm (GMA).

The GMA uses n lists; list 1 of size S is the main list and list i of size A_i, i = 2, 3, ..., n, are successively smaller auxiliary lists. A merge cycle of the GMA is defined as the operations required to fill list 2 and merge it with list 1. For reasons that will become apparent as we proceed, we call this the outer merge cycle (OMC). An OMC contains several inner merge cycles (IMC's). In order to fill list 2, list 3 is first filled and then merged with list 2. The operations carried out in so doing constitute an inner merge cycle (IMC_3) from list 3. We require $\lfloor A_2/A_3 \rfloor$ IMC_3's in order to fill all but at most $(A_3 - 1)$ locations of list 2. (The notation $\lfloor x \rfloor$ denotes the largest integer N such that N is less than or equal to x.) In general, if an inner merge cycle i (IMC_i) is defined as the operations

required to fill all but at most $A_{i+1} - 1$ locations of list i and merge it with list (i-1), then IMC_{i-1} contains $\lfloor A_i - 1 / A_i \rfloor$ IMC_i's. By definition, the OMC is the same as IMC_2.

Initially, the algorithm starts by extending the root node and ordering the newly extended paths by their metrics into list n. When list n is full, it is merged with list (n-1). Now list n is empty again, and the above operations are repeated until list (n-1) fills or is unable to accommodate another set of A_n paths. List (n-1) is then merged with list (n-2), which empties list (n-1). By repeatedly filling list (n-1) and merging it with list (n-2), all but at most $A_{n-1} - 1$ locations of list (n-2) are now occupied by code tree paths. Then list (n-2) is merged with list (n-3), and so on until all but at most $A_3 - 1$ locations of A_2 are filled. List 2 is then merged with list 1, thus completing an OMC. A new OMC then starts all over again.

The GMA extends, at any time, the best of the n top paths residing in the n lists. This makes the algorithm strictly metric-first. Selecting the best path requires at most (n-1) comparisons. For the GMA to be meaningful, n must be of order less than S. Before extending the best path a check is made to ensure that it is less than L branches in length; otherwise the eldest symbol of the best path is released and an ambiguity check is performed on all

the paths residing in the n lists.

A recursive version of the GMA is given below. This can easily be converted into an iterative version.

```
Procedure <GMA>
   begin
      main list ← root node;
      metric (root node) ← 0;
      while not all source samples are encoded do
         {fill list 2 and merge it with list 1; i.e., carry
         out an OMC, same as IMC2}
         perform <IMC2>
      end while
   end
end <GMA>

Procedure <IMCi>
   begin
      {Procedure valid for i = 2, 3, ..., n only}
      if i = n
         then
            begin
               while list n can take b more new paths do
                  select the best of the n top paths
                  residing in the lists;
               if the length of the best path < L
                  then begin
                        <extend> the best path and
```

```
                        <delete> this path;
                        <reorder> the extended paths into
                                  list n;
                    end
                else begin
                        perform <ambiguity check>;
                        <release> output symbol;
                    end
            end if
        end while
        {merge list n with list (n-1)}
        perform <merge (n-1, n)>
    end
else begin
        while list i can take in S(i+1) more paths do
            perform <IMC(i+1)>
        end while
        perform <merge (i-1, i)>
    end
end if
end
end <IMC(i)>

Procedure <merge (i, i+1)>
    begin
    if i = 1
```

```
        then merge list 2 into list 1, the main list

             {This is straight merging; excess paths are
             dropped off the bottom of the main list; list
             2 becomes empty}

        else

           if list i can take in A_{i+1} more paths

              then merge list i+1 into list i

                   {no paths are dropped; list i+1 becomes
                   empty}

              else {try to merge list i into list (i-1)}

                 perform <merge (i-1), i)>

           end if

      end if

   end

end <merge>
```

4.5 Resource Costs of the GMA Comparison-Based Cost Measure

The total number of comparisons done in an OMC may be arrived at as follows. Finding the best path for extension, which requires at most (n-1) comparisons, will be accounted for later.

Total number of comparisons in an OMC = comparisons done in merging lists 1 and 2 + comparisons done in filling list 2

$= S + A_2 + \lfloor A_2/A_3 \rfloor \times$ (comparisons done in IMC_3)

$= S + A_2 + \lfloor A_2/A_3 \rfloor (A_2 + A_3 + \lfloor A_3/A_4 \rfloor \times$ (comparisons done in IMC_4))

$$= S + A_2 + \lfloor A_2/A_3 \rfloor (A_2 + A_3 + \lfloor A_3/A_4 \rfloor (A_3 + A_4 + \lfloor A_4/A_5 \rfloor \times (\ldots + \lfloor A_{n-2}/A_{n-1} \rfloor (A_{n-1} + A_{n-2} + \lfloor A_{n-1}/A_n \rfloor \times (\text{comparisons done in } IMC_n)) \ldots)))$$

$$= S + A_2 + \lfloor A_2/A_3 \rfloor (A_2 + A_3 + \lfloor A_3/A_4 \rfloor (A_3 + A_4 + \lfloor A_4/A_5 \rfloor \times (\ldots + \lfloor A_{n-2}/A_{n-1} \rfloor (A_{n-1} + A_{n-2} + \lfloor A_{n-1}/A_n \rfloor (A_{n-1} + A_n + A_n^2/2)) \ldots))) \quad (4.5)$$

Removing the floor signs $\lfloor \rfloor$ from the above expression and dividing it by A_2 yields an upperbound to the number of comparisons per branch viewed, which is

$$1/A_2 (S + A_2 + A_2/A_3 (A_2 + A_3 + A_3/A_4 (A_3 + A_4 + A_4/A_5 (\ldots + A_{n-2}/A_{n-1} (A_{n-2} + A_{n-1} + A_{n-1}/A_n (A_{n-1} + A_n + A_n^2/2)) \ldots)))) \quad (4.6)$$

Minimizing the above expression with respect to A_n we get

$$A_n = (2A_{n-1})^{1/2} \quad (4.7)$$

Note that if n = 2, the GMA reduces to the 2-list MA and that the results from section 4.3.1 agree with (4.7). Substituting the expression for optimal A_n from (4.7) into (4.6) we get

$$\simeq 1/A_2 (S + A_2 + A_2/A_3 (A_2 + A_3 + A_3/A_4 (A_3 + A_4 + A_4/A_5 (\ldots + A_{n-2}/A_{n-1} (A_{n-2} + A_{n-1} + \sqrt{2}\, A_{n-1}^{3/2}) \ldots)))) \quad (4.8)$$

Again, minimizing the above expression with respect to A_{n-1} we get

$$A_{n-1} = 2^{1/3} A_{n-2}^{2/3} \quad (4.9)$$

Similarly,

$$
\begin{aligned}
A_{n-2} &= 2^{1/4}\ A_{n-3}^{3/4} \\
A_{n-3} &= 2^{1/5}\ A_{n-4}^{4/5} \\
&\vdots \\
A_{n-i} &= 2^{1/i+2}\ A_{n-i-1}^{i+1/i+2} \\
&\vdots \\
A_3 &= 2^{1/n-1}\ A_2^{n-2/n-1} \\
A_2 &= 2^{1/n}\ S^{n-1/n}
\end{aligned}
\qquad (4.10)
$$

Using the O-notation and expressing the auxiliary list sizes in terms of S, we get

$$
\begin{aligned}
A_2 &= O(S^{n-1/n}) \\
A_3 &= O(S^{n-2/n}) \\
&\vdots \\
A_i &= O(S^{n-i+1/n}) \\
&\vdots \\
A_n &= O(S^{1/n})
\end{aligned}
\qquad (4.11)
$$

Substituting the optimal values for A_2, A_3, ..., A_n from (4.11) into (4.6) and adding (n-1) to the resulting expressions to account for the number of comparisons done in selecting the best path for extension, we get

$$O(S^{1/n}) + 2\ (n-1) \qquad (4.12)$$

for an upper bound to the number of comparisons done per branch viewed. Since the smallest list, list n, must be at least of size b, the number of branches out of a node in the code tree, we have

$$A_n = O(S^{1/n}) \geq b$$

or
$$n \leq O(\log_b S) \tag{4.13}$$

Substituting the bound on n into (4.12), we obtain

$$O(S^{1/n}) + O(\log_b S) \text{ comparisons/branch viewed} \tag{4.14}$$

Introducing the term for ambiguity check, the above equation becomes

$$[O(S^{1/n}) + O(\log_b S)]\ E[C_{SA}] + S + \sum_{i=2}^{n} A_i \tag{4.15}$$

comparisons per branch viewed.

Table 4.1 gives the list sizes for a main list size of 1000 and for n = 2, 3, ..., 5. It is seen that for fixed S the number of side lists chosen depends on the value of $E[C_{SA}]$. For example, if $E[C_{SA}]$ lies between 5 and 12, n = 3 minimizes the number of comparisons per branch viewed while for all values of $E[C_{SA}]$ between 12 and 347, n = 4, and for $E[C_{SA}]$ above 347, n = 5 are the optimum values, respectively. For a given range of $E[C_{SA}]$ there exists an n that minimizes the number of comparisons per branch released. The equations (4.11) and (4.15) therefore present a useful set of design criteria for arriving at an optimum n if S and $E[C_{SA}]$ are given.

Table 4.1 Estimated Comparisons-Based Cost Measure and List Size of GMA for S = 1000; A_i's are Calculated from (4.7), (4.9) and (4.10)

n	S	A_2	A_3	A_4	A_5	Comparisons/ branch released (from (4.5))
2	1000	44				46 $E[C_{SA}]$ + 1044
3	1000	125	15			27 $E[C_{SA}]$ + 1140
4	1000	211	44	9		17 $E[C_{SA}]$ + 1264
5	1000	288	83	23	7	16 $E[C_{SA}]$ + 1401

Product Cost of the GMA

Merging list (i-1) with list i requires A_{i-1} accesses to a storage of size LA_{i-1}, A_i accesses to a storage of size LA_i, and $(A_{i-1} + A_i)$ accesses to a storage of size $L(A_{i-1} + A_i)$. If i = n, in addition to the above mentioned accesses, $A_n^2/2$ accesses to a storage of size LA_n are required in filling the list. The total product cost in an OMC is then calculated as follows.

Product cost in an OMC

$$
\begin{aligned}
= L(&S^2 + A_2^2 + (S + A_2)^2 + \lfloor A_2/A_3 \rfloor (A_2^2 + A_3^2 + (A_2 + A_3)^2 \\
&+ \lfloor A_3/A_4 \rfloor (A_3^2 + A_4^2 + (A_3 + A_4)^2 \\
&+ \ldots \\
&+ \lfloor A_{n-2}/A_{n-1} \rfloor (A_{n-2}^2 + A_{n-1}^2 + (A_{n-2} + A_{n-1})^2 \\
&+ \lfloor A_{n-1}/A_n \rfloor (A_{n-1}^2 + A_n^2 + (A_{n-1} + A_n)^2 + A_n^3/2)) \ldots))).
\end{aligned}
\tag{4.16}
$$

On dividing (4.16) by A_2, removing the floor signs to obtain an upperbound, and minimizing successively with respect to A_n, A_{n-1}, ..., A_3, A_2, one gets

$$\begin{aligned} A_n &= O(A_{n-1}^{2/3}) \\ A_{n-1} &= O(A_{n-2}^{6/7}) \\ A_{n-2} &= O(A_{n-3}^{14/15}) \\ &\vdots \\ A_{n-i} &= O(A_{n-i-1}^{(2^{i+2}-2)/(2^{i+2}-1)}) \\ &\vdots \\ A_2 &= O(S^{(2^n-2)/(2^n-1)}) \end{aligned} \tag{4.17}$$

The constants within the O's of (4.17) for calculating up to four auxiliary list sizes are given below. For higher values of n the constant is close to 0.5.

$$\begin{aligned} A_n &= 1.26\ A_{n-1}^{2/3} \\ A_{n-1} &= 0.82\ A_{n-2}^{6/7} \\ A_{n-2} &= 0.65\ A_{n-3}^{14/15} \\ A_{n-3} &= 0.58\ A_{n-4}^{30/31} \end{aligned} \tag{4.17a}$$

On expressing list sizes in terms of S, we have

$$
\begin{aligned}
A_2 &= O(S^{(2^n-2)/(2^n-1)}) \\
A_3 &= O(S^{2(2^{n-1}-2)/(2^n-1)}) \\
A_4 &= O(S^{2(2^{n-2}-2)/(2^n-1)}) \\
&\vdots \\
A_i &= O(S^{2^{(i-2)}\ (2^{n-i+2}-2)/(2^n-1)}) \\
&\vdots \\
A_n &= O(S^{2^{n-1}/(2^n-1)})
\end{aligned}
\qquad (4.18)
$$

From the above set of equations

$$\lim_n A_2 = O(S)$$

$$\lim_n A_n = O(S^{1/2}).$$

All the other lists are of order between $\sqrt{S}$ and S.

Asymptotically, the product cost is

$$L[\{O(S^{2^n/(2^n-1)}) + (n-1)S\}\ E[C_{SA}] + S^2 + \sum_{i=2}^{n} A_i^2] \qquad (4.19)$$

where $L[S^2 + \Sigma_{i=2}^{n} A_i^2]$ is the cost of ambiguity checks. It is no longer clear if this cost can be neglected except when the side lists are few. The factor LS(n-1) in (4.19) accounts for the cost of selecting the best path.

Consider the example in Table 4.2. As in the comparisons-based example of Table 4.1, the optimum value of n depends on the value of $E[C_{SA}]$. Given n and S, the auxiliary list sizes are greater for the product cost

Table 4.2 Estimated Product Cost Measure and List Sizes of GMA for S = 1000; Auxiliary List Sizes are Calculated Using (4.17a)

n	S	A_2	A_3	A_4	A_5	Estimated Product Cost per Branch Released (from (4.17a) and (4.16))
2	1000	126				$L\ [27063\ E[C_{SA}] + 10^6]$
3	1000	305	57			$L\ [16411\ E[C_{SA}] + 1.09 \times 10^6]$
4	1000	536	179	32		$L\ [11506\ E[C_{SA}] + 1.18 \times 10^6]$
5	1000	680	374	131	26	$L\ [11751\ E[C_{SA}] + 1.26 \times 10^6]$

measure than for the comparisons-based cost measure. For larger n, the term corresponding to ambiguity checks in (4.19) increases enormously.

Sum Cost of the GMA

The space plus time cost in an OMC is

$$[LS + LA_2 + A_1 + A_2 + \lfloor A_2/A_3 \rfloor (LA_2 + LA_3 + A_2 + A_3 + \lfloor A_3/A_4 \rfloor (LA_3 + LA_4 + A_3 + A_4 + \ldots + \lfloor A_{n-1}/A_n \rfloor (LA_{n-1} + LA_n + A_{n-1} + A_n + A_n^2/2)\ldots))]. \quad (4.20)$$

On dividing (4.20) by A_2, removing the floor signs to obtain an upperbound, minimizing the resulting equation successively with respect to A_n, A_{n-1}, A_{n-2}, ..., A_2, one gets the following set of equations for optimal list sizes.

$$A_2 = O((L+1)^{1/n} S^{(n-1)/n})$$

$$A_3 = O((L+1)^{2/n} S^{(n-2)/n})$$

$$\vdots$$

$$A_i = O((L+1)^{(i-1)/n} S^{(n-i+1)/n})$$

$$\vdots$$

$$A_n = O((L+1)^{(n-1)/n} S^{1/n}). \qquad (4.21a)$$

It can easily be shown that

$$A_i = 2^{(i-1)/n} (L+1)^{(i-1)/n} S^{(n-i+1)/n} \qquad (4.21b)$$

Asymptotically, the sum cost is

$$O[(L+1)^{(n-1)/n} S^{1/n}] E[C_{SA}] + (n-1)(L+2) E[C_{SA}] + S + \sum_{i=2}^{n} A_i \qquad (4.22)$$

where $(n-1) E[C_{SA}]$ is the cost of selecting the best path and $S + \Sigma_{i=2}^{n} A_i$ is the cost of performing ambiguity checks. Again, as for the product cost, the cost of ambiguity check may not be negligible compared to the other two terms in (4.22) except for small values of n.

Table 4.3 gives an example to illustrate the significance of the equation for the sum cost measure of the GMA given in (4.22). Conclusions cited before regarding the optimum value of n for a given range of values of $E[C_{SA}]$ hold good here also.

Table 4.3 Estimated Sum Cost and List Sizes of GMA for S = 1000 and L = 25; A_i's are Calculated from (4.21b)

n	S	A_2	A_3	A_4	Estimated sum cost per branch released (from (4.20))
2	1000	224			255 $E[C_{SA}]$ + 1224
3	1000	373	139		220 $E[C_{SA}]$ + 1512
4	1000	477	228	108	221 $E[C_{SA}]$ + 1813

4.6 Summary

Expressions for the different cost measures were derived for the MA and GMA. It was shown that the MA reduces the cost of the stack algorithm. Generalizing the merge algorithm yields further cost advantages. Table 4.4 summarizes the results.

The GMA is a generalized metric-first encoding algorithm. A scheme utilizing multiple side lists proposed in the literature is the multiple stack algorihm due to Chevillat and Costello [29]. Their algorithm is not a strictly metric-first search and does not use merging. Another algorithm due to Haccoun and Ferguson [28] maintains a list of paths and extends at each time the top M best paths. This algorithm is again not strictly metric-first, but a combination of metric-first and breadth-first techniques; it also does not use merging. Merging to minimize the resource cost was first proposed by Mohan and

Anderson in [54].

Table 4.4 Resource Costs of the Stack, 2-List Merge and Generalized Merge Algorithms

Algorithm	Resource Cost/branch released: Comparisons-Based	Product	Sum
Stack	$O(S\ E[C_{SA}])$	$O(LS^2\ E[C_{SA}])$	$O(LS)+O(SE[C_{SA}])$
2-List Merge	$O(\sqrt{S}\ E[C_{SA}])$	$LO(S^{4/3}\ E[C_{SA}]+S^2)$	$S+O\{(LS)^{1/2}\ (1+E[C_{SA}])\}$
Generalized	$[O(S^{1/n}) +O(\log_b S)]\ E[C_{SA}] + S + \Sigma_{i=2}^{n} A_i$	$L[\{O(S^{2^n/(2^n-1)}) +(n-1)S\}E[C_{SA}] + S^2 + \Sigma_{i=2}^{n} A_i^2]$	$O[(L+1)^{(n-1)/n}\ S^{1/n}]\ E[C_{SA}] + S + \Sigma_{i=2}^{n} A_i$

CHAPTER 5

A CODE TREE SEARCH ALGORITHM USING A BALANCED TREE DATA STRUCTURE

5.1 Introduction

This chapter explores avenues for further reduction in the resource cost of an algorithm through the use of balanced trees to store code tree paths. We now have two types of trees, the code tree used in encoding the source or decoding the received symbols and the data tree to store the identities or path maps of paths encoded or decoded by the algorithm. A class of balanced trees known as height-balanced trees (also known as AVL trees) are introduced. The use of such trees to store paths reduces the search, deletion, and insertion times of paths to O(log S) and results in an alternative method to the GMA, achieving reduced costs.

5.2 Binary Trees - Preliminaries

As a prelude to the topic of balanced trees, a few definitions are in order. The definitions here closely follow those in [57, Sec. 2.3] and [58, Sec. 2.3].

A binary tree is one in which every node has at most

two sons. It is defined recursively as follows. A binary tree T is either empty or consists of a distinguished node called the root node whose left and right subtrees, T_ℓ and T_r, are in turn binary trees.

A binary tree is said to be in symmetric order if all names in the left subtree of any node x_i precede x_i in natural order and all names in the right subtree of x_i follow x_i in natural order. Lexicographic ordering is synonymous with natural ordering.

Figure 5.1 shows four different binary trees over the names A, B, C, and D with A < B < C < D as their natural order. The trees are in symmetric order.

An extended binary tree (see Fig. 5.2(b)) is formed by adding external nodes to the binary tree of Fig. 5.2(a). The external nodes are also called leaves. All other nodes are called internal nodes. A binary tree with n internal nodes has (n+1) external nodes.

The height $h_b(T)$ of a tree T is defined to be the length of the longest path from the root node to an external node.

A binary search tree over names $x_1, x_2, \ldots, x_n$ is an extended binary tree in which the names occur in symmetric order. If $y_o, y_1, \ldots, y_n$ represent the (n+1) external names, then node y_o corresponds to all names that precede x_1, node y_i (i = 1, ..., n-1) to all names following x_i but

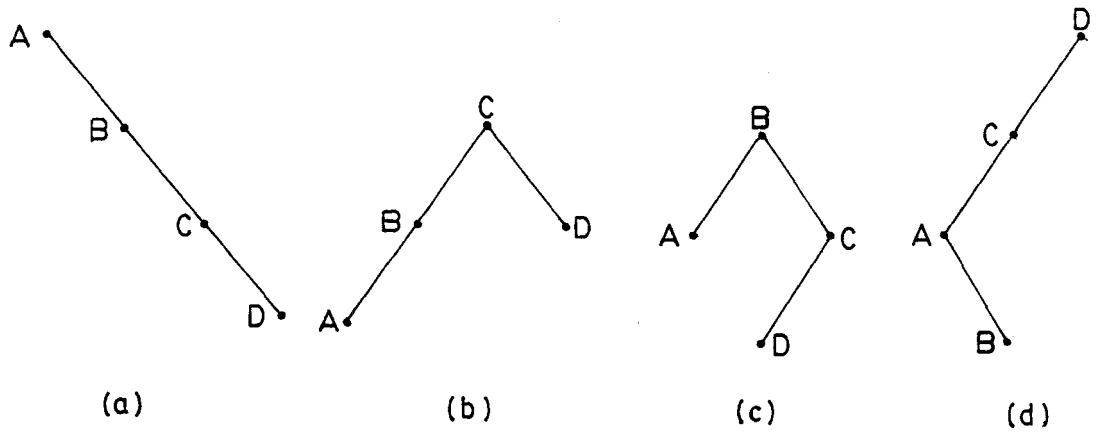


Figure 5.1 Four Binary Trees Over Names A, B, C, and D in Symmetric Order

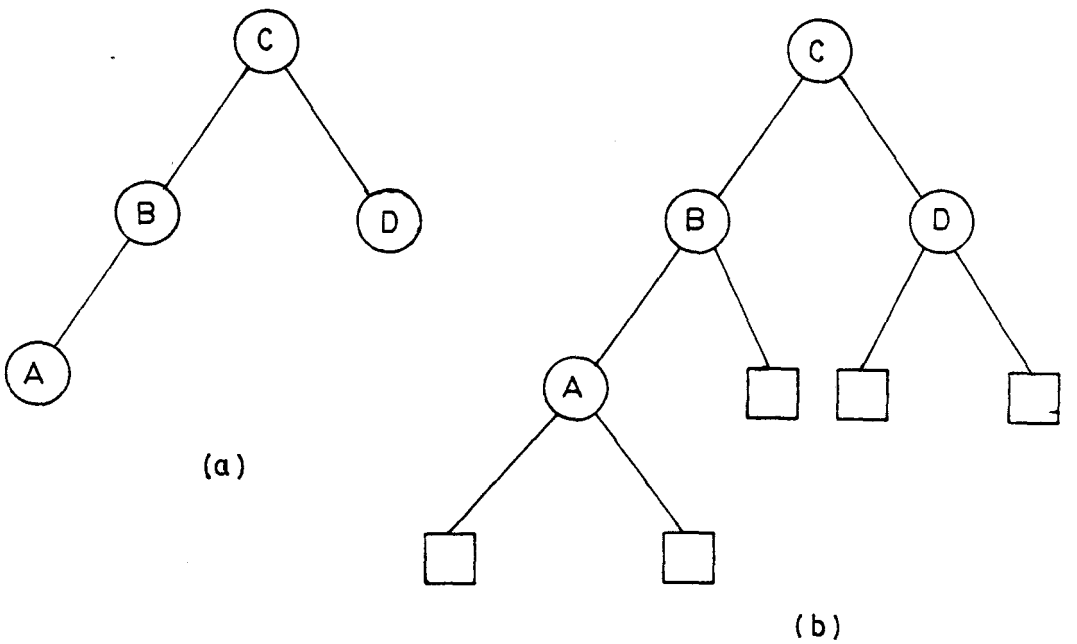


Figure 5.2 A Binary Tree and Its Extension

preceding x_{i+1}, and y_n to all names that follow x_n. The external nodes correspond to 'holes' or 'gaps' in the binary search tree.

Algorithm to Search for a Name in a Binary Search Tree

A binary search tree is searched for a name z as follows.

1) If the tree is empty (i.e. no root node exists), z does not exist in the tree. The search terminates unsuccessfully.
2) If z precedes the name of the root node, the left subtree is searched.
3) If z follows the name of the root node, the right subtree is searched.
4) If z equals the name of the root node, the search terminates successfully.

The search terminates unsucessfully whenever an external node is reached during the search.

Algorithm to Insert a Name into a Binary Search Tree

A node is considered a triple (LLINK, NAME, RLINK), where the NAME field contains the name associated with a node, LLINK a pointer to the left son of that node, and RLINK a pointer to the right son. A new name is inserted into a binary search tree by the following recursive

algorithm.

1) If the tree is empty, NAME of root node ← z and LLINK (root node) ← RLINK (root node) ← Λ, where Λ is an empty pointer. The algorithm terminates.
2) If z equals the name of the root node, z already exists in the tree. The algorithm terminates.
3) If the name z precedes the name of the root node, follow this algorithm to insert z into the left subtree.
4) If the name z follows the name of the root node, follow this algorithm to insert z into the right subtree.

Note that the new node so inserted becomes an end node of the tree.

Algorithm to Delete a Name from a Binary Search Tree

Figure 5.3 illustrates the deletion algorithm. If the name to be deleted z has no son, the link of z's father that points to z is replaced by Λ, the null pointer (Fig. 5.3(a)). If z has one son, the link field of the father of z that points to z is replaced by the link field of z that points to z's son (Fig. 5.3(b)). If z has two sons, then the procedure is slightly complicated. Note that the predecessor of z by the natural order is the right-most node in the left subtree (node y in Fig. 5.3(c)) and that it has

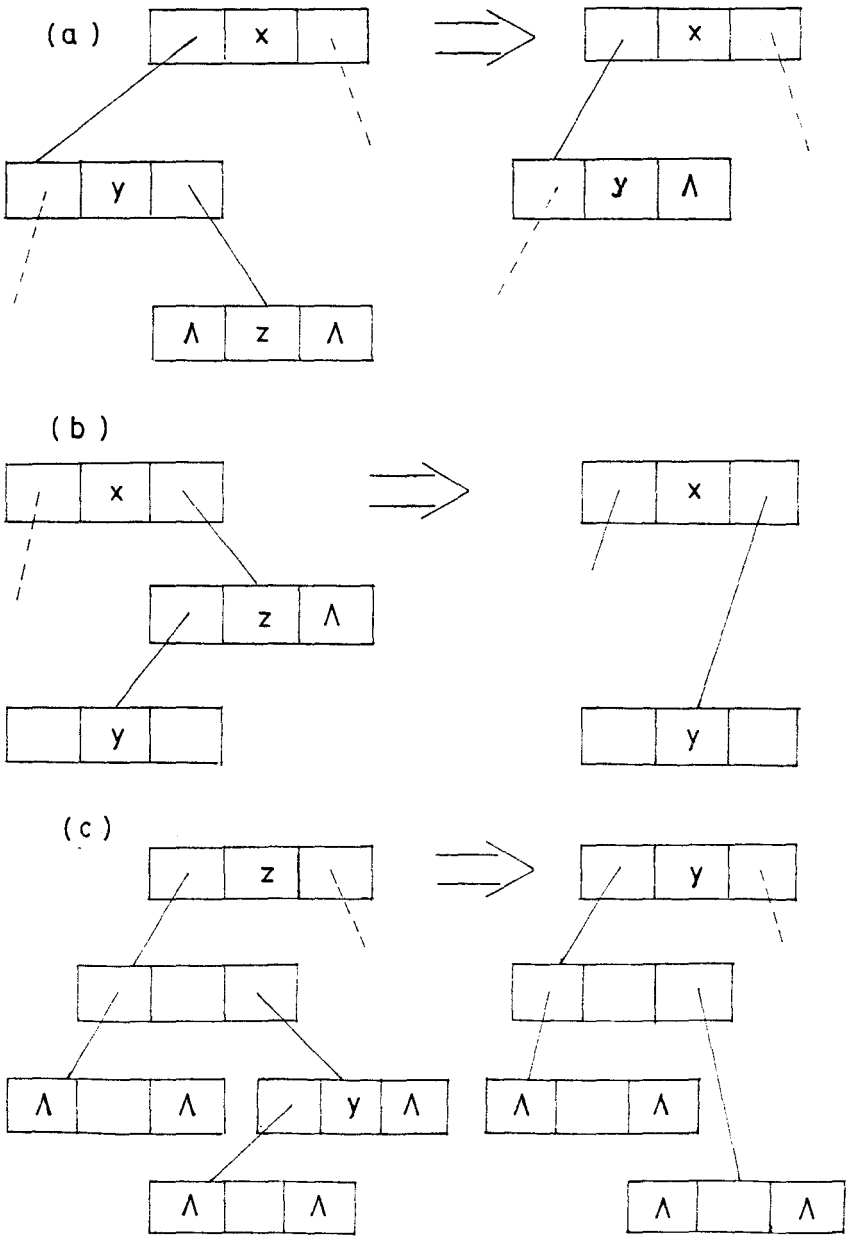


Figure 5.3 Deletion Algorithm; a) z has no Son, b) z has One Son, c) z has Two Sons

at most one son. Now replace name z by name y and delete the node that contained y (z may equally well be replaced by the name that follows it in natural order).

Some more definitions follow.

Define a complete binary tree as one having all external nodes on levels q and q+1 for some q. For a complete binary tree with n internal nodes

$$q = \log_2 (n+1). \tag{5.1}$$

The external path length E of a tree is defined to be the sum - taken over all external nodes - of the lengths of the paths from the root to each external node.

The internal path length I is the sum - taken over all internal nodes - of the lengths of the paths from the root to each internal node.

For a complete binary tree (see [57], [58])

$$E = I + 2n. \tag{5.2}$$

Equation (5.1) implies that a name in a complete binary tree can be searched for in O(log n) time. A much more complex problem is the optimal organization of names with a given set of frequencies. Note that Huffman's procedure [57, Sec. 2.3.4.5] for constructing trees with minimum weighted path length will not, in general, produce a tree arranged in symmetric order. This problem of constructing optimal binary search trees is studied in detail in references [55], [58], [49].

While n random names can be inserted into or deleted from a binary search tree in O(nlogn) time, the worst-case time is $O(n^2)$. The quadratic time behaviour is due to the tendency of the tree to get skewed when names are non-random. In the degenerate case the tree may just be a sequential list requiring linear rather than logarithmic time for a single insertion or deletion operation. In order to prevent the tree from getting skewed, it needs to be rebalanced so that it deviates from a completely balanced tree as little as possible. Consider the example in Fig. 5.4, where a new node is added to the tree on the left. The tree is restructured as shown at right. The insertion of the new node has affected all the nodes, requiring work proportional to n. The height-balanced trees achieve such insertions and deletions by carrying out a succession of local changes along a single path from the root node to a leaf in only O(logn) time. This we study next.

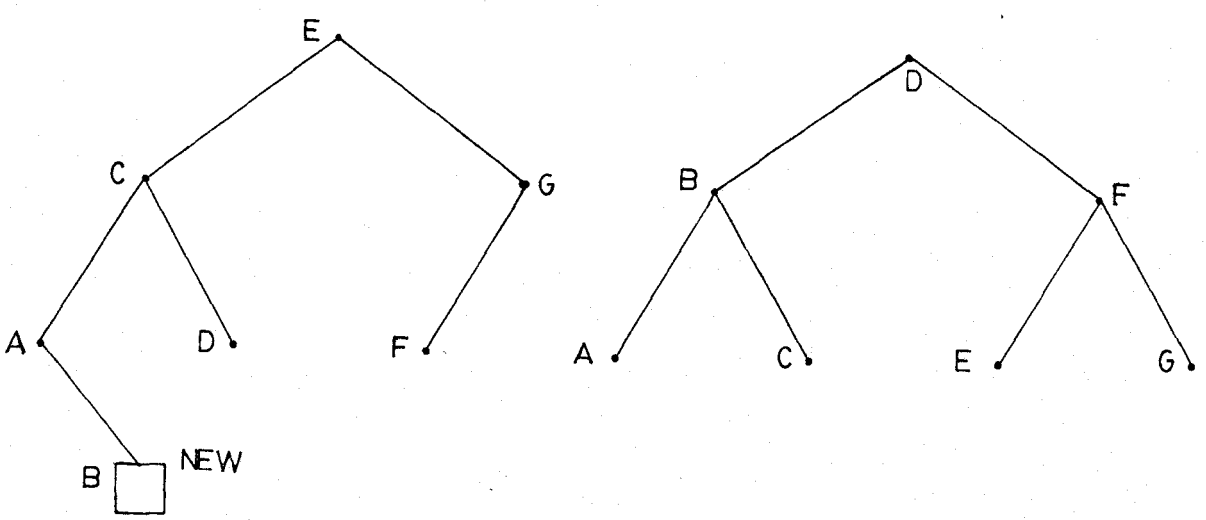


Fig. 5.4 Rebalancing Requires Work Proportional to n

5.3 Height-Balanced Binary Trees

The subject of height-balanced trees for data or file organization has been amply treated in the literature (see [58, Sec. 6.4], [49, Sec. 4.9], and [55, Sec. 6.2.3]). They first appeared in Adelson-Velskii and Landis [59] and hence height-balanced trees are also known as AVL trees. A survey of different strategies for file searching and organization is provided in [60]. The definitions and algorithms in this section closely follow [58].

Definition: A binary tree T is height-balanced if and only if the two subtrees of the root, T_ℓ and T_r, satisfy

1) $|h_b(T_\ell) - h_b(T_r)| \leq 1$, where $h_b(\cdot)$ denotes height.

2) T_ℓ and T_r are height-balanced.

Figure 5.5 shows some examples of height-balanced binary trees and one that is not. The height constraint forces the height-balanced binary trees to differ as little as possible from completely balanced binary trees. The balanced trees in Fig. 5.5 are in fact the most-skewed height-balanced binary trees of height 1, 2, and 3, respectively.

Empirical evidence [55, pp. 440] suggests that the number of comparisons needed to insert the Nth item into a balanced tree is approximately $\log_2 N + 0.25$ for large N. This differs from the average search or insertion time for a

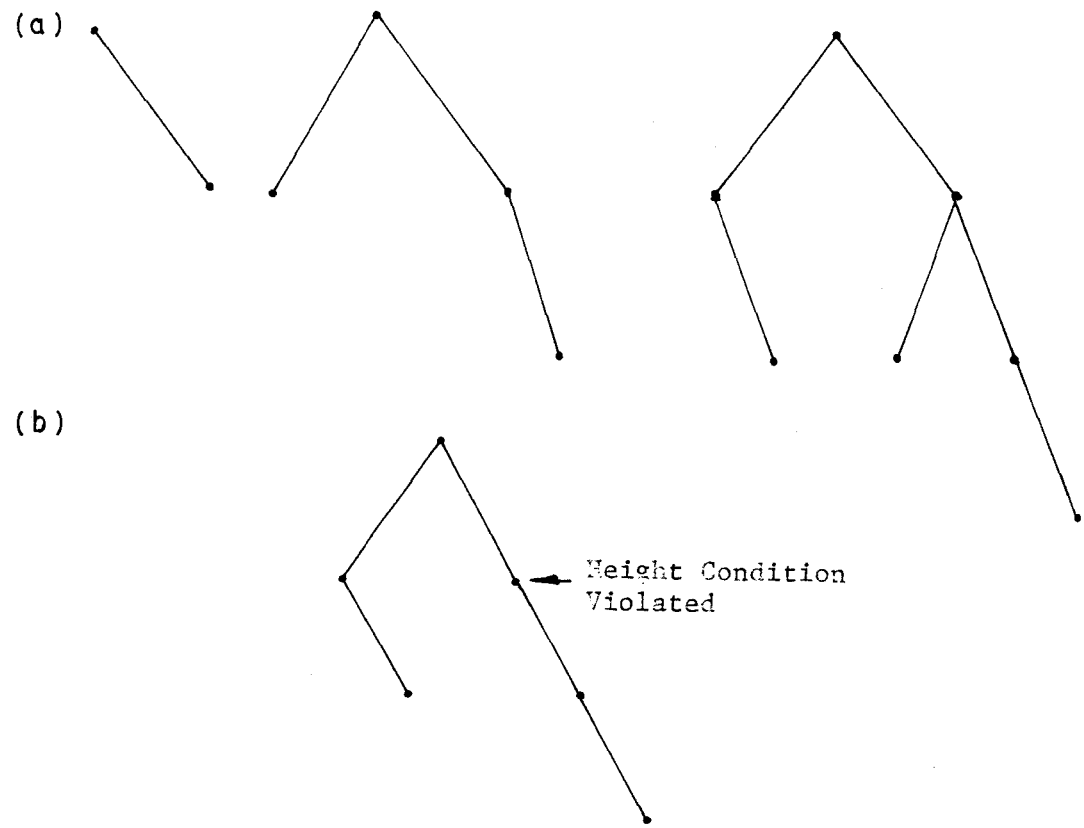


Figure 5.5 Examples of a) Height-Balanced and b) Non-Height-Balanced Trees

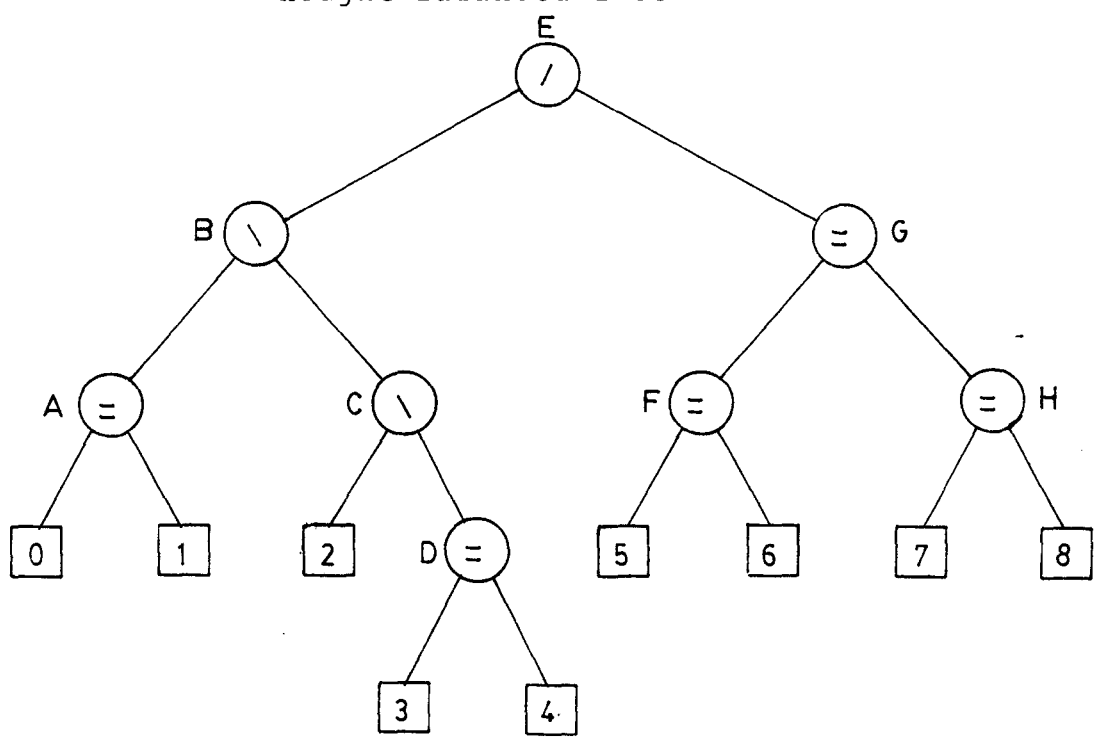


Figure 5.6 A Height-Balanced Tree Showing Balance Factors of Nodes

completely balanced binary tree only by a constant. In fact the following theorem ([59], [55, pp. 453]) shows that the average search time need never be more than 45% above the optimum.

Theorem: The height of a balanced tree with N internal nodes always lies between $\log_2(N+1)$ and $1.44 \log_2(N+2) - 0.328$.

Proof: Ref. [55].

Figure 5.6 shows a height balanced tree in which balance factors are associated with nodes to indicate their height conditions. If $h_b(T_\ell)$ and $h_b(T_r)$ denote the heights of left and right subtrees of a node, respectively, then the balance factor associated with that node is \, =, or / according as $h_b(T_r) - h_b(T_\ell) = 1$, 0, or -1, respectively.

Two tree transformations called rotation and double rotation are used to restore the balance of a tree after an insertion or deletion of a node. In Fig. 5.7(a) a new node NEW added to the right subtree of node B changes the balance factor of B and makes the right subtree 'too heavy' (Subtrees are indicated by large triangles). After the transformation, as shown on the right side of Fig. 5.7(a), the tree is rebalanced and balance factors are changed as indicated. This transformation is known as rotation.

Figure 5.7(b) shows a situation, where the new node added makes the left subtree of C and right subtree of A

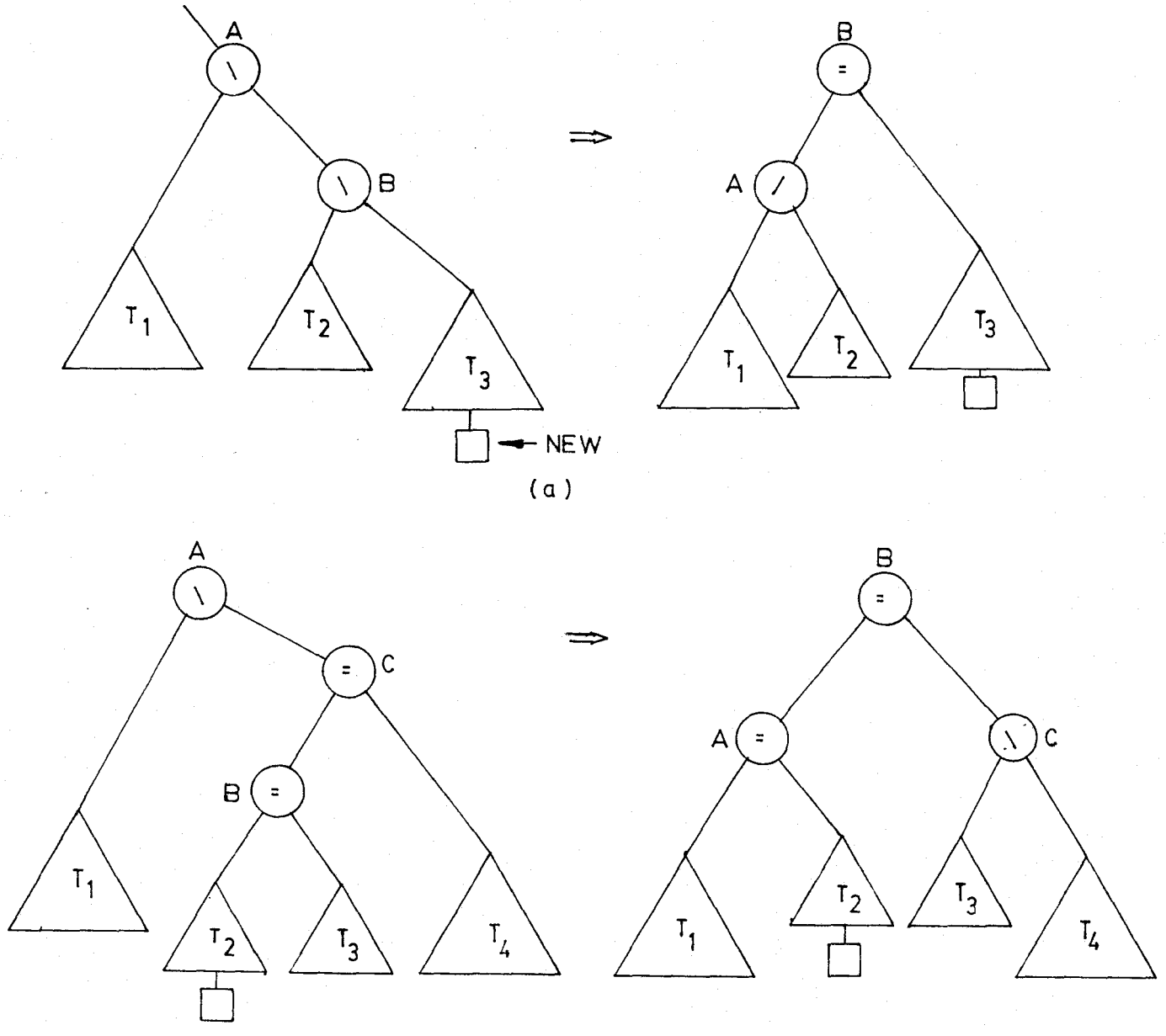


Figure 5.7 Examples to Illustrate the Tree Transformations; a) Rotation, and b) Double Rotation

'too heavy'. A transformation known as double rotation (so called because of two rotations performed, one around A and another around C) restores the balance of the tree. This affects the balance factors as shown at right. Note that both the transformations, rotation and double rotation, preserve the symmetric order of the tree.

Algorithm for Inserting a New Node into a Balanced Tree

Using first the algorithm for insertion of a new node into a binary tree given in Sec. 5.2, the new node is inserted into the balanced tree. Note that this algorithm makes the new node a terminal node or a leaf. Consequently we need only consider the effect of adding a leaf to the balanced tree. The path from the newly inserted leaf to the root node is traced upwards incorporating changes to the tree depending at most on the two immediately preceding transformations on the upward path. The algorithm works as follows.

1) If the current node's balance factor is =, it is changed to \ if the last step originated from the right son and to / if it originated from the left son. If the current node is the root node, the algorithm terminates; otherwise it continues upwards.
2) If the current node's balance factor is / or \, it is changed to = if the last step originated from the

shorter of the two subtrees of the current node. The algorithm then terminates.

3) If the current node's balance factor is / or \, and the last step originated from the taller of the two subtrees of the current node, then
   a) if the last two steps originated in the same direction, i.e., both from right sons or both from left sons, then perform an appropriate rotation and terminate the algorithm. Or,
   b) if the last two steps originated in the opposite directions, perform an appropriate double rotation and terminate the algorithm.

Algorithm for Deleting a Node from a Balanced Tree

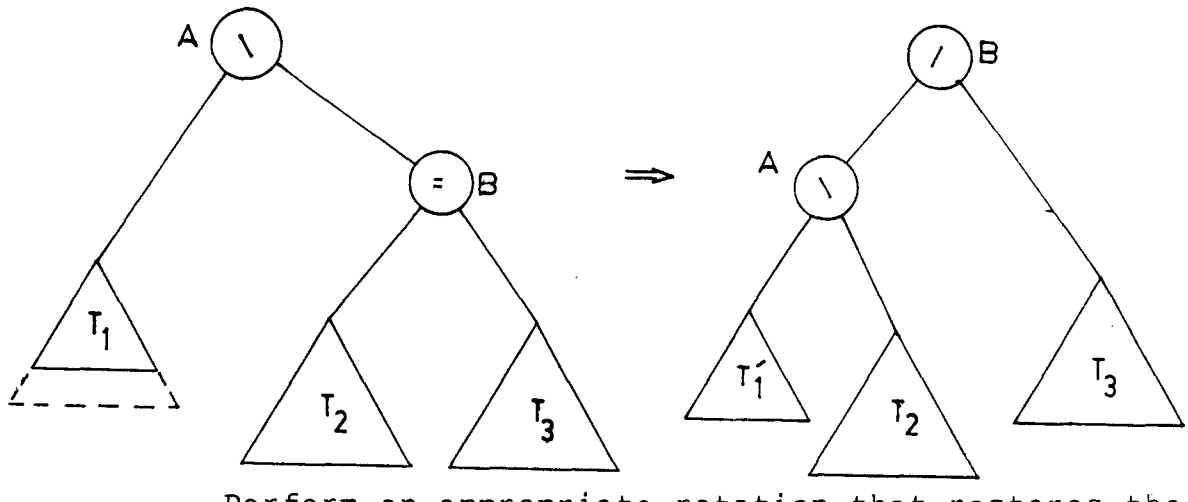
First, using the deletion algorithm of Sec. 5.2 delete the given node from the balanced tree. As was already mentioned in Sec. 5.2, the effect of this algorithm was to delete a leaf. Consequently, we need only consider the effect of deletion of a leaf on the balanced tree. The algorithm traverses the tree upwards from the deleted leaf to the root node passing along the message that the subtree rooted at the current node has been shortened. In the following figures, dotted lines are appended to the subtree that has been shortened. The deletion algorithm works as follows.

1) If the current node's balance factor is =, change it to / if the right subtree was shortened and to \ if the left subtree was shortened. Terminate the algorithm.

2) If the current node's balance factor is / or \, change it to = if the taller of the two subtrees of the current node was shortened; continue upwards passing along the message that the subtree rooted at the current node has been shortened.

3) If the current node's balance factor is / or \ and the shorter of the two subtrees was shortened, then 'too heavy' condition exists at the current node. The following subcases exist depending on the balance factor of the current node's son. Mirror images of the following cases are similarly dealt with.

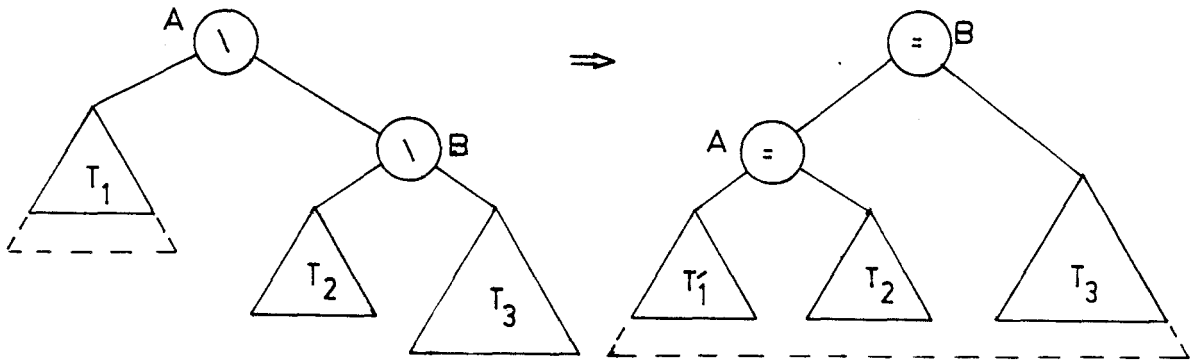
a)



Perform an appropriate rotation that restores the balance without altering the height of the

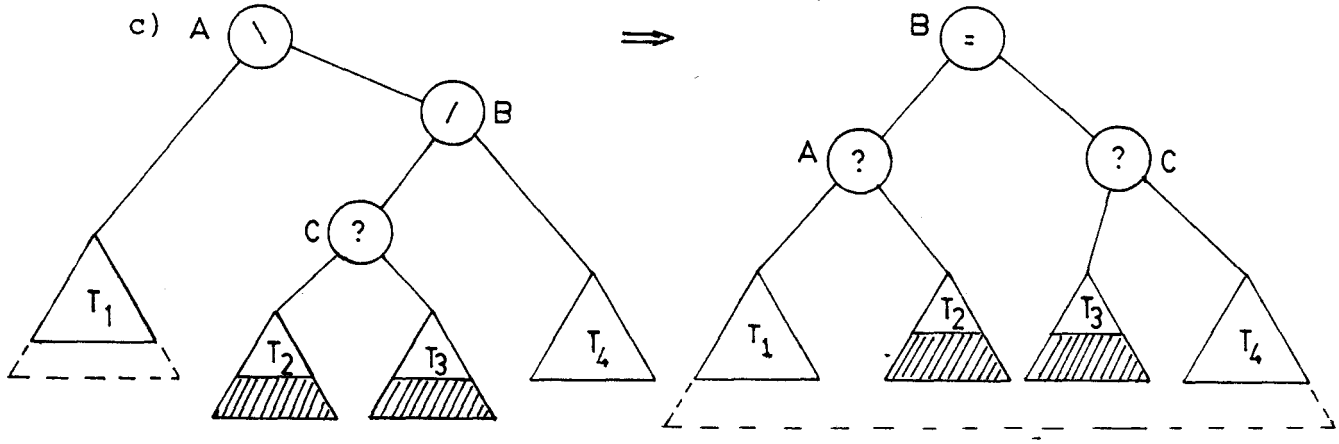
subtree rooted at the current node and terminate the algorithm.

b)



Perform an appropriate rotation that shortens the subtree rooted at the current node and continue upwards.

c)



Perform an appropriate double rotation that reduces the height of the subtree rooted at the current node and continue the algorithm upwards. The balance factors of A and C after the transformation depend on the balance factor of B prior to the transformation.

Whereas the insertion algorithm requires at most one transformation, the deletion algorithm requires as many as [h/2] transformations, where h is the height of the tree. However, in most cases the algorithm requires a constant number of transformations that is independent of the height of the tree.

An astute reader would have by now recognized our motivations in describing in detail the concepts of height-balanced binary trees. If we maintain a height-balanced tree data structure whose nodes are code tree paths, with tree symmetrically ordered by path metric, then we can carry out deletion or insertion of a path in O(logS) time. Though the algorithms for insertion and deletion seems complicated at first glance, with a suitable data structure, they can easily be translated into a language of list manipulation. We describe next a metric-first code tree search algorithm that uses height-balanced tree data structure to store paths.

5.4 A Code Tree Search Algorithm with Balanced Tree Data Structure

Let a cell consist of the following: path map, S(•), length of the path, L(•), metric of the path, μ(•), a pointer to the left son, LLINK(•), and a pointer to the right son, RLINK (•). Figure 5.8 shows the cell structure.

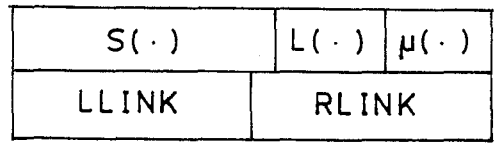


Figure 5.8 Cell Structure; S(.)-Path Map, L(.)-Length, μ(.)-Metric, LLINK(.)- Pointer to Left Son, RLINK(.)- Pointer to Right Son

All the available cells are initially linked together and maintained in a pool called the available space. Cells are drawn from it whenever a path is extended. Deleted cells are returned to the available space.

Initially, the data tree consists of just the root node of the code tree. The root node is then extended and the b newly extended paths are inserted into the data tree using the insertion algorithm of Sec. 5.4. The old best path (initially the root node of the code tree) is deleted using the deletion algorithm of Sec. 5.4. At any time instant, the best path corresponding to the right most node of the data tree is extended and this node is deleted. The b new paths are inserted into the data tree by using cells drawn from the available space. If the available space becomes empty, the worst path corresponding to the left most node of the tree is deleted if the metric of the new path is greater than that of the worst path and the deleted cell used for the new path. If the available space is empty and

the new path is 'poorer' than the worst path, it is dropped from contention. This provision assures that only the S best paths are retained after a path extension. Also, extending the right most node of the data tree makes the algorithm strictly metric-first.

Let B point to the node containing the best path and W to that containing the worst. In the algorithm that follows, by DT we mean the data tree and by CT the code tree.

```
Procedure <metric-first-balanced-tree>
   begin
      {Initialize data tree, i.e., create its root node}
      S (root node of DT) ← path map corresponding to the
                            root node of CT;
      RLINK (root node of DT) ← LLINK (root node of DT) ← Λ
      μ (root node of DT) ← 0;
      B ← W ← address of root node of DT;
      While not all source samples are encoded do
         begin
            If L(B) < the allowed maximum length of paths,
                      L,
               then
                  begin
                     <extend> the path in B;
                     <delete> the cell B;
                     <insert> new paths into DT;
```

```
                update B and W;
            end
        else
            begin
                <traverse> the tree in symmetric order;
                perform <ambiguity check>;
                <release output symbol>;
            end
    end if
  end
end while
end
end <metric-first-balanced-tree>
```

The resource cost of the above algorithm is

space cost: $LS + S \log L + pS + 2 S \log_b S$

path storage | length storage | metric storage | links storage

$= S (L + \log L + p + 2 \log_b S)$

b-ary symbols (5.3)

time cost: $O(\log S)$ accesses/branch viewed, for deletion and insertion of paths, plus

$O(S)$ access/branch released, for ambiguity check (5.4)

The total space-time product cost is

$O(L S \log S \, E[C_{SA}]) + O(LS^2)$ access-symbols/branch released (5.5)

In (5.5) we have deleted the $\log_b S$ term from (5.3) on the assumption that L dominates it. Comparing (5.5) with the $O(S^2 E[C_{SA}])$ product cost of the stack algorithm clearly points to the cost reductions obtainable by the balanced-tree code tree searching scheme. This yields cost advantages over the 2-list merge algorithm as well.

One improvement to the algorithm proposed here is to assign a set of paths instead of a single path to a node of the data tree. All the paths assigned to a node may have metrics $\mu(\cdot)$ lying within $M_i \leq \mu(\cdot) \leq M_i + \Delta M$, where M_i is some suitable number within the allowed metric values for paths and ΔM a suitable metric increment. Such groups of paths are referred to as <u>buckets</u>. This strategy moves the algorithm away from being metric-first and forms the subject of the next chapter.

CHAPTER 6

A DYNAMIC BUCKET ALGORITHM

6.1 Introduction

It was shown in Chapter 4 that the dependence on S for the product cost of the stack algorithm, S^2, could be reduced to $S^{4/3}$ for the 2-list merge algorithm. In Chapter 5, the use of balanced binary trees to store paths was shown to further reduce the dependence on S to SLogS. It is of interest to explore other strategies to see if the dependence on S can be further reduced. It is shown here that relaxing the stringent requirement of always extending the best path accomplishes such a reduction in cost. Still unanswered is whether we must relax the requirement in order to get this cost.

The algorithms described here make use of the concept of equivalence classes known as buckets. A bucket contains a set of paths whose metrics fall within a metric range assigned to that bucket. Paths are not sorted within a bucket, but there is an implicit ordering of buckets. Paths chosen from the best bucket are extended and, when storage is limited, paths are continually deleted from the worst bucket. Since a path from the best bucket, considered for extension, need not have the best metric, algorithms using

buckets are not strictly metric-first; but they become roughly so when buckets are assigned successively finer metric ranges.

The concept of buckets, well known to computer scientists in the field of sorting and searching [55, Sec. 6.4], was first proposed by Jelinek [27] in the context of sequential decoding. Dick [53] used this concept for encoding Gaussian sources. Data structures for the implementation of the bucket algorithm are proposed in [56]. Motivated primarily by the concepts of hashing, dynamic hasing, and trie searching [61], [55, Secs. 6.3 and 6.4], we propose here a new bucket-type code tree search algorithm and data structures for its implementation. Before describing these, we briefly outline Jelinek's scheme.

6.2 Jelinek's Bucket Algorithm

In Jelinek's bucket algorithm [27], two paths belong to the same bucket if they satisfy a certain equivalence relation defined on their metrics such as, for example, the requirement that the integral parts of their metrics be equal. Two sets of storage locations are assigned; one is the available space and the other the header list* referred

---

* Jelinek refers to them as the stack and the auxiliary stack respectively. However, we have used the terms available, space and headers drawn from list processing languages terminology.

to respectively by indices $g \in \{1, 2, \ldots, S\}$ and $\ell \in \{-K, -K+1, \ldots, -1, 0, 1, \ldots, J\}$. The range -K to J of the header list index ℓ is so chosen that the path metrics lie between -K and J with a sufficiently high probabiliy. A location ℓ of the header list is a pointer $G(\ell)$ to the bucket ℓ containing paths, the integral parts of whose metrics are equal to ℓ. A cell g in a bucket consists of three types of information, the path information or identiy $S(g)$, the metric of the path $\mu(g)$, and a pointer $P(g)$ to the next entry in the same bucket. The last entry in the bucket, or an entry that is the only item in a bucket, will have its $P(g)$ value set to 0. The storage information is depicted in Fig. 6.1.

Initially, all the cells in the available space are linked together through their pointer fields using the procedure <initialize available space>. New cells are drawn from it using the <get new cell> procedure. NEWCELL is a pointer to the first free cell in the available space. All the cells within a bucket are linked together using their $P(\cdot)$ fields. There are pointers B and W pointing to the best and worst buckets, respectively. A deleted cell is returned to the available space and appropriately linked to the rest of the cells.

We now redefine several of the operations such as extend, add path, delete path, etc., in terms of the data

Header List		Available Space			
Location ℓ	Pointer to an entry in the available space $G(\ell)$	Location g	Path Info. S(g)	Metric $\mu(g)$	Pointer to next entry P(g)
		1	000	-1 1/4	2
		2	1	-1 3/4	0
		3	0100	-2	4
-3	5	4	001	-1 1/4	1
-2	3	5	011	-2 1/4	0
-1	0	6	0110	0	0
0	6				
1					
2					

Fig. 6.1 The Storage Information at an Intermediate Stage During Searching of a Code Tree, $h(\mu) = \lfloor \mu \rfloor$.

structures defined in the two preceding paragraphs.

```
Procedure <extend> {a path from the best bucket B}
   begin
      g ← G(B)
      Compute the cumulative metrics of the b new paths by
      extending the path at location g to its b nearest
      neighbours;
      <delete> the path at location g;
      for p = 1 to b do
         begin
            <get new cell>;
            <add path> p at location pointed to by NEWCELL;
```

```
            end
        end for
    end
end <extend>

Procedure <add path> {at location p}
    begin
        {let ℓ be the integral part of the metric μ(.) of the
        new path}
        {link the path to the top of bucket ℓ}
        P(p) ← G(ℓ);
        G(ℓ) ← p;
        {update pointers to best and worst buckets}
        If B < ℓ then B ← ℓ;
        If W > ℓ then W ← ℓ;
    end
end <add path>

Procedure <delete> {path at location p}
    begin
        ℓ ← integral part of μ(p);
        if G(ℓ) = p
            then {cell to be deleted is the first one in bucket
                  ℓ}
                G(ℓ) ← P(p)
            else {let cell q be the predecessor of p;
                  link the successor of p to predecessor of p}
                P(q) ← P(p)
```

```
        end if

        {link cell p to available space}

        P(p) ← NEWCELL

        NEWCELL ← p

        {If the bucket ℓ happens to be the best bucket B, B
        may have become empty if the deleted path were the
        only one in the bucket; update B}

        while G(B) = 0 do B ← B-1;

        {similarly update W}

        while G(W) = 0 do W ← W-1;

    end

end <delete>


Procedure <get newcell>

    begin

        If available space is not empty

            then return the address of the top cell from the
                 available space in NEWCELL

            else <delete> top path from the worst bucket W and
                 return this address in NEWCELL;
                 Make the worst bucket W the available space
        end if

        Link the available space header to next-to-top cell in
        the available space

    end

end <get new cell>


Procedure <ambiguity check>

    begin

        examine each path;
```

```
      <delete> ambiguous paths;
   end
end <ambiguity check>
```

Using the above procedures the bucket algorithm is described below.

```
Procedure <bucket>
   begin
      <initialize available space>;
      <get new cell>;
      Assign root node to it; metric of root node is 0;
      W ← 0; B ← 0;
      While not all source samples are encoded do
         begin
            If length of top path in B < L
               then
                  begin
                     <extend> top path in B;
                     {delete this path and add new paths
                      into buckets}
                  end
               else
                  begin
                     perform <ambiguity check>;
                     <release output symbol>;
                  end
```

```
            end if
          end
        end while
      end
end <bucket>
```

The cost of the bucket algorithm is

time cost: K' accesses/branch viewed and

K" S accesses/branch released, for ambiguity check

space cost: $S\ (L + \log L + p + \log_b S)$ b-ary symbols (6.1)

Here K' and K" are constants being determined by the precise sequence of reads and writes in procedures <extend>, <add path>, <delete>, etc. The $\log_b S$ term accounts for the storage taken up by the link field P(•). Asymptotically, the product cost is

$$\approx LS\{E[C_{SA}] + S\} + H \text{ access symbols/branch released} \quad (6.2)$$

It is assumed here that the node comptuation is close to that of the stack algorithm. The factor H accounts for the cost of mapping the metric of a path onto one of the buckets. The term $\log_b S$ is omitted under the assumption that L dominates it.

The bucket procedure described above, while retaining the essential features of Jelinek's algorithm, is slightly simplified. Further it is presented in the algorithmic

language that we use throughout this thesis. Jelinek's original scheme permitted paths to be extended as long as storage was available, i.e., there was no explicit constraint L on the path length like we have here.

6.3 Later Bucket Algorithm Developments

Anderson and Mohan propose data structures for the bucket algorithm in [56]. They propose that the bucket procedure be viewed as hashed search. In this scheme paths are hashed onto one of the Q buckets by a function $h(\mu)$, called hashing function, that operates on the metrics of paths. There is no separate header list as in Jelinek's scheme; instead, the first Q locations in the memory form headers of buckets. Initially bucket 1 is the best bucket and bucket Q the worst. The available storage is chained to Q. New paths are stored in cells drawn from the worst bucket, i.e., the available space. At any intermediate point in time, bucket ℓ is the best bucket if ℓ is not empty and buckets 1, 2, ..., ℓ-1 are empty. Similarly m is the best bucket if m is not empty and buckets m+1, m+2, ..., Q are all empty. Buckets are constructed as shown in Fig. 6.2.

Let B refer to the best bucket and w to the worst. The primitives <extend path>, <add path>, etc., as defined in [56] are as follows.

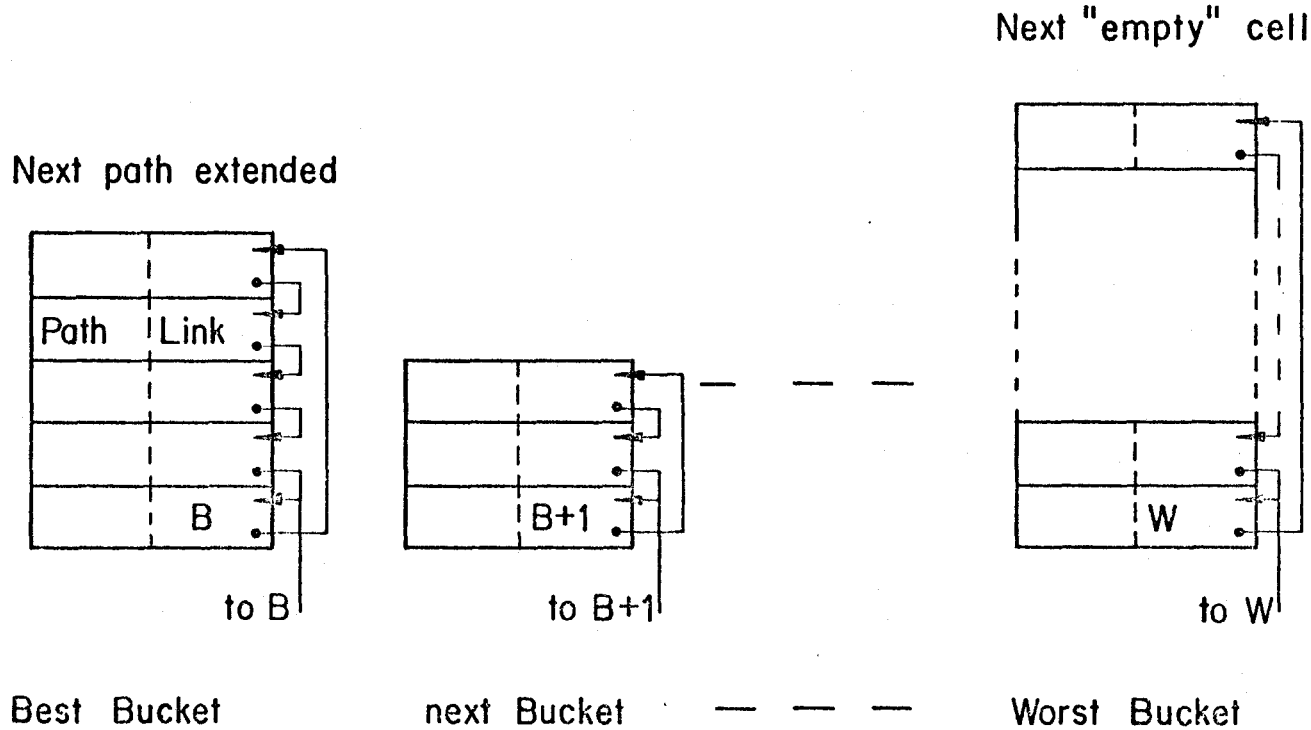


Figure 6.2 Chained Storage of Buckets for Bucket Algorithm. W Points to Worst Bucket and B to Best Bucket

```
Procedure <extend path>
   begin
      Access bucket B, the best bucket;
      Remove top path, extend it to its nearest neighbours
      and <add paths> to buckets;
      Link cell B to next-to-top cell;
      Return the removed cell containing the path just
      extended to the available space, i.e., to bucket W;
      If B is empty update B;
         {while B is empty do B ← B+1 endwhile}
   end
end <extend path>

Procedure <add path>
   begin
      Compute h(μ); if h(μ) > W then W ← h end if
      if h(μ) < B then B ← h end if
      <get new cell>;
      Link h(μ) to point to NEWCELL;
      Place new path in NEWCELL;
      Link NEWCELL to old top cell;
   end
end <add path>

Procedure <get new cell>
   begin
      Access bucket W;
```

```
        While W is empty do W ← W-1 end while;
        Return topcell address pointed to by W in NEWCELL;
        Link W to next-to-top cell;
    end
end <get new cell>
```

With the primitives defined as given above, the bucket procedure described in [56] takes on the same form as in section 6.2. The resource cost formula also takes on essentially the same form as in equation (6.2), where H is now the hashing function cost.

In both these schemes, buckets may grow arbitrarily large giving rise to imbalancing. Imbalancing is the result of a poor choice of hashing function. This gives rise to high insertion and deletion times for paths. As both these schemes do not have a provision for modifying the hashing function while the algorithm is in progress, such an imbalancing may occur with a high probabiliy. It is not clear what is the choice of optimal hashing function $h(\cdot)$. It is unlikely that it is uniform especially as S grows very large. It is desirable to redefine h from time to time [56], but it is not clear how this is to be done. In order to overcome these difficulties, we propose next an algorithm that uses buckets that split and merge dynamically according to certain criteria. Appropriately, we have named this algorithm the dynamic bucket algorithm. Since, during

bucket splits and merges, the metric ranges assigned to buckets change dynamically, we can view the hashing function h(•) as a self-modifying hashing function. This results in a more balanced data structure. Larson [61] has proposed a dynamic hashing concept that is applicable here.

6.4 A Dynamic Bucket Algorithm

Here, there are initially Q root nodes that are pointers to Q empty buckets. These root nodes will form the root nodes of the data structure to be described and should be differentiated from the root node of the code tree. Unlike in Jelinek's scheme, here buckets are of fixed storage capacity K.

Initially, root node 1 is the pointer to the best bucket and root node Q to the worst. Paths are hashed onto one of the Q root nodes by a function h(•) acting on their metrics. If a path is hashed onto a root node pointing to a partially filled bucket, the path is assigned to a free location in the bucket. If a bucket is not yet assigned to the root node, a new bucket is drawn from the available space of buckets and the path is assigned to an empty location in the bucket.

If the bucket is full, the following actions take place. Two new nodes are drawn from the available space of nodes and they form the left and right sons of the node onto

which the path was hashed. Each of these two newly created nodes will now point to a bucket; the left node will point to an empty bucket drawn from the available space of buckets and the right node to the bucket that is full. These two buckets in the same level in the data structure are called brother buckets. Approximately half the number of paths from the full bucket are then transferred to the newly created brother bucket. The new path is assigned to one of these buckets depending on its metric. This situation of a bucket being full and a new bucket being created is illustrated in Fig. 6.3. In graph-theoretic terminology the data structure of Fig. 6.3 corresponds to a forest of binary trees.

The reverse situation to bucket splits, i.e., bucket merging, takes place when deleting a path; if two brother buckets together have less than K paths these two buckets are merged to form a single bucket. The father of the two nodes that were pointing to the two buckets before merger now points to the bucket that contains the merged paths. Two nodes and a bucket are returned to the space of available nodes and buckets respectively. For example, if in Fig. 6.3(c) the buckets pointed to by nodes 100 and 101 together have less than or equal to K paths, these two buckets are merged, and the data structure shown in Fig. 6.3(b) will emerge. The nodes 100 and 101 and the bucket

(a)

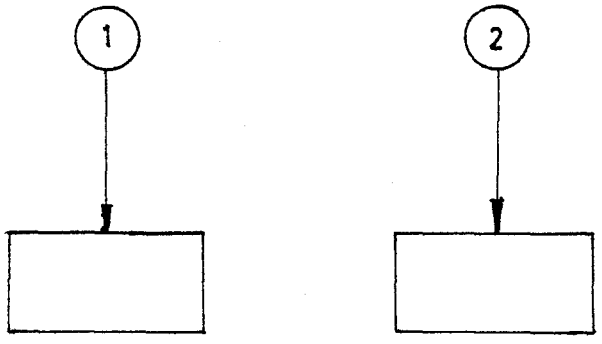


Figure 6.3(a) Bucket 1 is Full; a New Path is to be Entered into Bucket 1; Q = 2

(b)

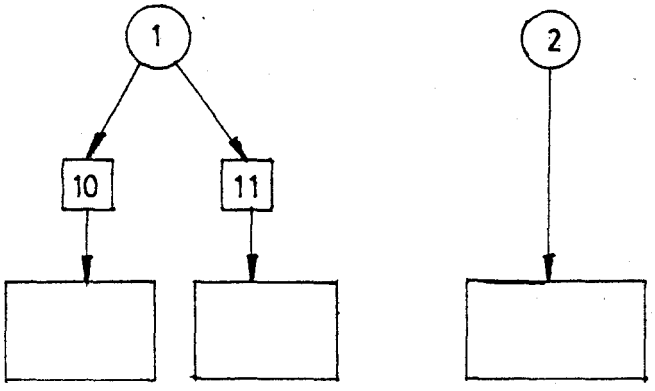


Figure 6.3(b) Two New Nodes 10 and 11, Successors to node 1, are Created; These Point to Two Buckets

(c)

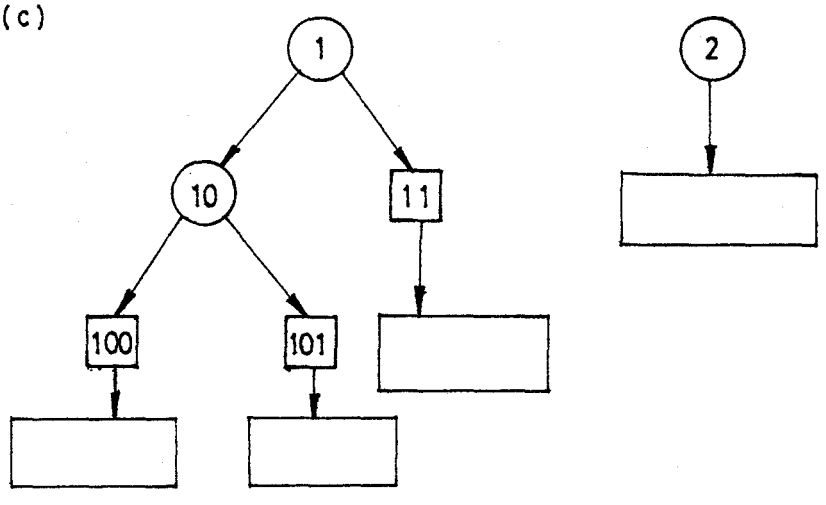


Figure 6.3(c) Further Splits May Occur to Modify the Data Structure

pointed to by 100 are returned to the available space.

Different strategies could be adopted during a bucket split. One is simply to transfer half the number of paths (say the top K/2 paths) to the new bucket. Another strategy is to transfer only the best K/2 paths to the new brother bucket. The later strategy is in conformity with the rule of keeping the leftmost bucket as the best bucket; as one traverses from left to right the buckets get progressively worse, with the rightmost bucket the worst. The later strategy is preferable in an actual implementation as it provides a compensation for a poor initial choice of the hashing function h(•) by dynamically providing for finer refinements of the path metrics assigned to buckets. However, it does not guarantee that a binary tree starting at a root node will not grow unduly large. A binary tree, however, lends itself to rebalancing as has already been seen in the previous chapter. A combination scheme incorporating rebalancing may thus profitably be used.

During a bucket split, the strategy of transferring the best K/2 paths to a new bucket requires that the paths within the bucket to be split be sorted on their metrics. We can avoid sorting by using the following procedure. Assume that the metric distribution within a bucket is roughly uniform, a reasonable assumption if finer metric ranges are assigned to buckets. When a bucket with a metric

range $[\mu_i, \mu_j)$ is to be split, paths with metric falling within $[\mu_i, \mu_i + (\mu_j-\mu_i)/2]$ are assigned to a new bucket; the rest of the paths remain in the old bucket. If it so happens that all the K+1 paths (K paths in the bucket and one new path) go either to the left or to the right bucket, further splits occur until the buckets have fewer than K paths.

Consider the example in Fig. 6.4. When a path is hashed onto node i with metric range [1,5), bucket i is found to be full. The root node is then assigned two sons i0 and i1. Bucket i is split and assigned to two buckets i0 and i1. i1 is empty and so i0 is split again into i00 and i01. Again, as i00 is empty, i01 is split into buckets i010 and i011. As none of the buckets are now empty, further splitting stops and the new path is assigned to one of the buckets i010 or i011.

Assuming that paths go either to the left or to the right with equal probability,

$$P\{\text{a bucket is full}\} = (1/2)^K \tag{6.3}$$

$$P\{\gamma \text{ bucket splits}\} = (1/2)^{(\gamma-1)K}\{1 - (1/2)^K\} \tag{6.4}$$

$$E\{\text{no. of bucket splits}\} = \Sigma\, \gamma\, (1/2)^{(\gamma-1)K}\{1 - (1/2)^K\}$$

$$= 1/(1-(1/2)^K) \tag{6.5}$$

Multiple bucket splits are very rare. If K = 5, multiple splits occur once every 32 splits, if K = 10 once every 1000 splits, and if K = 20 once every 10^6 splits.

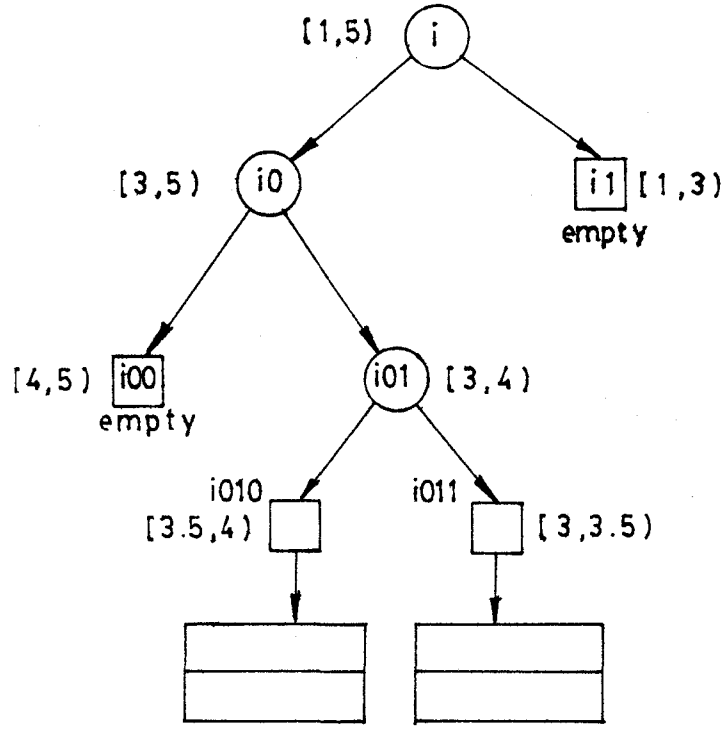


Figure 6.4 Bucket Splitting Strategy

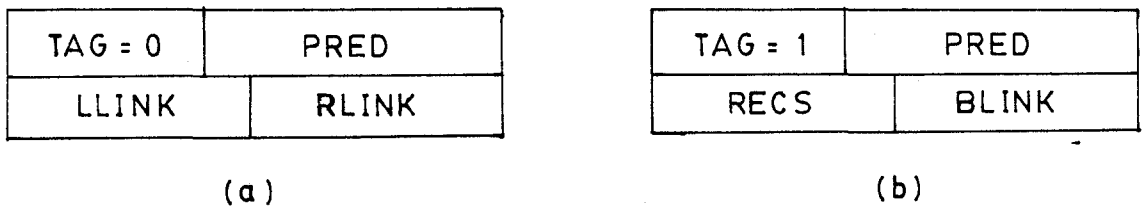


Figure 6.5 Node Structure. a) Internal Node and b) External Node

Though the assumption that paths go either to the left or to the right bucket with equal probability and the independence assumption are not valid due to the nature of the code tree structure, the above analysis gives an indication of the nature of multiple splits. Due to the rarity of multiple splits, we can expect the data structure to be reasonably balanced.

Data structures required to implement the dynamic bucket algorithm will now be described. Two types of available spaces are maintained, the available space of nodes and the available space of buckets. A node has the structure illustrated in Fig. 6.5. A node can be either an internal node (circular nodes in Fig. 6.3) or an external node (square nodes in Fig. 6.3). A link field of an internal node may either point to another internal node or to an external node while that of an external node points to a bucket if one is assigned to it or, if no bucket is assigned to it, contains 0. The tag field of a node is either 0 or 1. 0 indicates an internal node and 1 an external node. The PRED field of a node points to its predecessor. If there is no predecessor, as in the case of a root node, PRED field contains 0. LLINK and RLINK of an internal node point to the left and right sons respectively. An external node has instead BLINK and RECS in their places. BLINK links to a bucket and RECS indicates the number of

records in that bucket. A bucket may be implemented either as a linked list as in Fig. 6.2, or it may simply be K contiguous storage locations. In the later case, each location has a one bit field set aside to indicate whether or not a path resides in that location. When a bucket becomes empty, the whole block of K locations is returned to the available space of buckets. Again we redefine basic operations such as add path and delete path using the data structures described here.

```
Procedure <add path>
   begin
      {Let m be the metric of the path to be added}
      R ← h(m) {Hashing function h(•) hashes the path onto
                one of the Q root nodes R}
      {Let [μi, μj) be the metric range assigned to the root
         node R. Once R is given, this range is easily
         computable from a predetermined formula}
      i ← R;
      while TAG(i) = 0 do
         if m ε [μi, μi + (μj-μi)/2)
            then begin i ← RLINK(i); μj ← μi + (μj-μi)/2 end
            else begin i ← LLINK(i); μi ← μi + (μj-μi)/2 end
         end if
      end while
      if RECS(i) < K
            then begin if BLINK(i) = 0 then
                          BLINK(i) ← NEWBUCKET end if
```

```
                assign path to a free location in
                bucket BLINK(i)

        end

    else {Bucket BLINK(i) must be split as the new
          path cannot be entered into it}

        begin

            j ← NEWBUCKET {a new bucket is fetched
                           from the avaiable space of
                           buckets}

            while RECS(i) = K do

                begin

                    LRECS ← 0; RRECS ← 0;

                    for p = 1 to K do

                        if metric of path p in bucket
                        BLINK(i) ε [μ_i + (μ_j-μ_i)/2, μ_j)

                            then begin remove path p from
                                       bucket BLINK(i) and
                                       assign it to a free
                                       location in bucket
                                       j; LRECS ← LRECS+1;
                                          RRECS ← RRECS-1;

                                 end

                        end if

                    end for

                    {Fetch new nodes from the avaiable
                    space of nodes and link them to node
                    i}

                    LNODE ← NEWNODE; RNODE ← NEWNODE;

                    TAG(LNODE) ← TAG(RNODE) ← 1;

                    RECS(LNODE) ← LRECS;

                    RECS(RNODE) ← RRECS;
```

```
PRED(LNODE) ← PRED(RNODE) ← i;
If LRECS = K {all paths into the
              left bucket}
   then begin
        BLINK(LNODE) ← j;
        BLINK(RNODE) ← 0;
        j ← BLINK(i); i ← LNODE;
        μi ← μi + (μj-μi)/2;
        end
   else if RRECS = K {all paths into
                      the right
                      bucket}
        then begin
             BLINK(RNODE) ←
                BLINK(i);
             BLINK(LNODE) ← 0;
             i ← RNODE;
             μj ← μi + (μj-μi)/2;
             end
        end if
   end if
   LLINK(PRED(LNODE)) ← LNODE;
   RLINK(PRED(RNODE)) ← RNODE;
end
end while
If m ε [μi, μi + (μj-μi)/2)
   then add new path to BLINK (RNODE)
   else add new path to BLINK (LNODE)
```

```
                    end if

                end

            end if

        end

end <add path>

Procedure <delete path>

    begin

        {Let i be the node pointing to the bucket in which
        resides the path to be deleted.  Let its brother node
        be j.  After removing the path from the bucket pointed
        to by i, it is merged with its brother bucket if the
        two have together ≤ K paths.  This is carried onto
        higher levels if necessary.}

        Remove the path from bucket BLINK(i);

        RECS(i) ← RECS(i)-1;

        While RECS(i) + RECS(j) ≤ K do

            begin

                Merge the buckets i and j into i; link bucket i
                to the predecessor of nodes i and j;

                if the new merged bucket has a brother bucket
                    then j ← brother bucket's address

                end if

            end

        end while

    end

end procedure
```

Using these two procedures, the dynamic bucket algorithm will take on essentially the same form as the procedure

<bucket> in Sec. 6.2.

6.5 Analysis Using Tries

Turning attention now to the analysis of the dynamic bucket algorithm, we first note that the data structure, forest of binary trees, generated by the algorithm can be considered as Q binary tries (see Knuth [55, Sec. 6.3] for a discussion on tries). The name trie from the middle letters of the word retrieval was first proposed in [62]. Knuth defines a trie as "essentially an M-ary tree, whose nodes are M-place vectors with components corresponding to digits or characters. Each node on level ℓ represents the set of all keys that begin with a certain sequence of ℓ characters; the node specifies an M-ary branch, depending on the $(\ell+1)$st character".

The following example from [58] illustrates trie searching. The trie shown in Fig. 6.6 stores decimal digits. Each node has associated with it a ten-place vector or index. In order to locate the name 2718, branch 2 of the root node is taken. This branch leads to another node whose pointer at location 7 leads to the name 18. Adding the prefix 27, the path map of the path that leads to the node containing name 18 from the root node, to 18 yields the name 2718. If the name to be searched is 573, following the above procedure, we find the address in location 5 of the

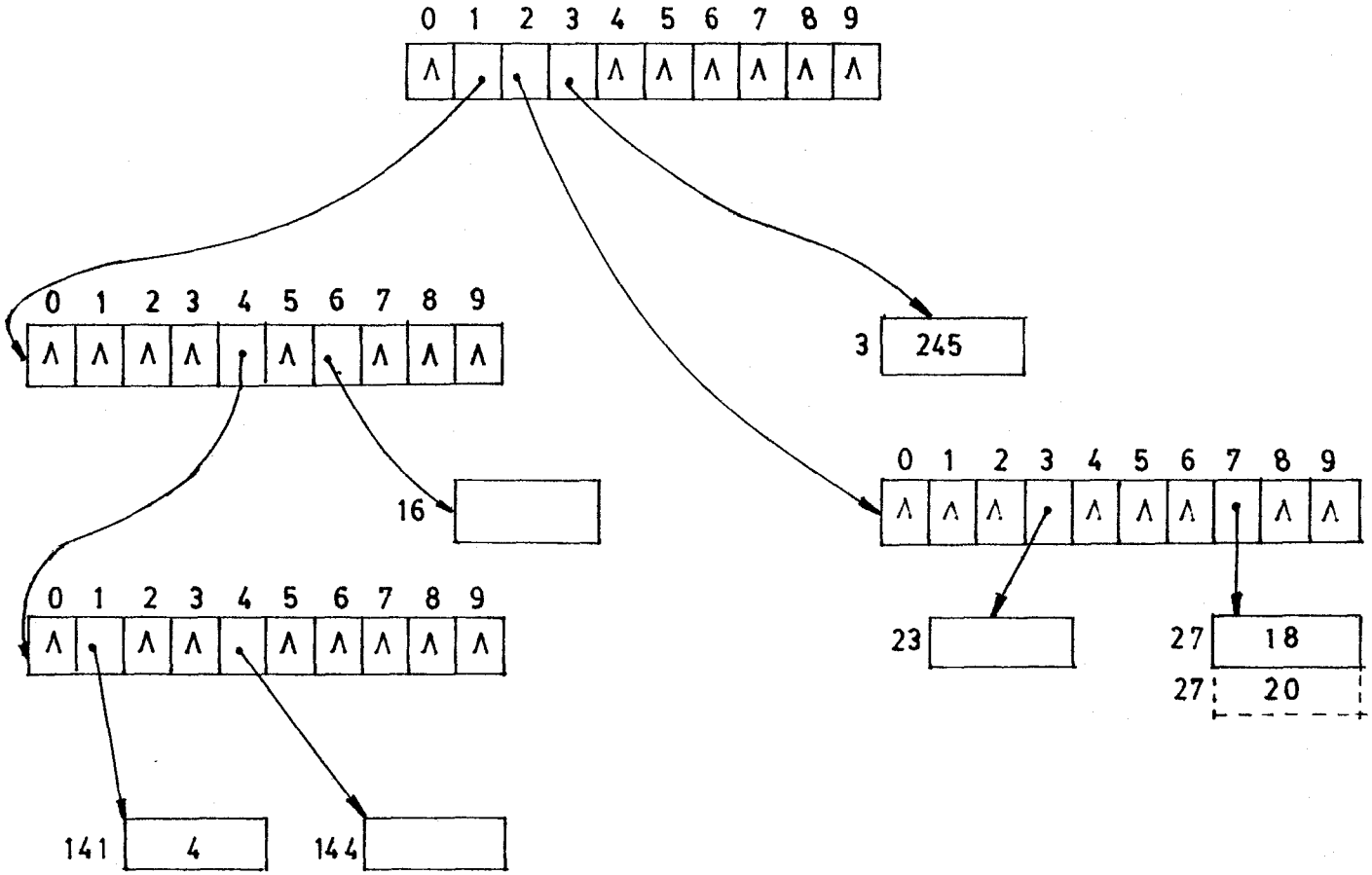


Figure 6.6 An Example to Illustrate Trie Searching and Organization

root node. Since this is Λ, the null pointer, name 573 does not exist in the trie. More than one name may be assigned to an index of a node. A new name 2720 is assigned as shown by the dotted slot in Fig. 6.6 if the trie permits assigning two names to an index of a node.

In the dynamic bucket algorithm, each node represents a range of metric $[\mu_i, \mu_j)$. Depending on whether the metric of a path falls in the upper or the lower half of the metric range, the node specifies a two-way branch, the left branch or the right branch. This being true at each and every internal node of the Q binary trees, these are essentially binary tries.

From Knuth [55, Sec. 6.3], the number of nodes needed to store N random keys in an M-ary trie, with the trie branching terminated for subfiles of $\leq$ s keys, is approximately

$$N/(s \ln M) + N\, g(N) + O(1) \qquad (6.6)$$

In (6.6), g(N) is a complicated function whose value is always less than 10^{-6}; hence it can be neglected. Equation (6.6) then reduces to

$$N/(s \ln M) \qquad (6.7)$$

In our case s = K, the bucket size, and M = 2, the number of branches out of an internal node. Let n_i, i = 1, 2, ..., Q be the number of paths stored in the ith of the Q binary

tries. Since there are a total of S paths stored in the Q binary tries, we have

$$\sum_{i=1}^{Q} n_i = S \tag{6.8}$$

Applying (6.7) to each of the Q binary tries and summing up, we get

$$E\{\text{number of internal nodes}\} \simeq \sum_{i=1}^{Q} \frac{n_i}{K \ln 2} = \frac{S}{K \ln 2} \tag{6.9}$$

In a binary tree, there are n+1 external nodes and n internal nodes. So,

$$E\{\text{no. external nodes}\} \simeq \sum_{i=1}^{Q} \frac{n_i + 1}{K \ln 2} = \frac{S + Q}{K \ln 2} \tag{6.10}$$

Adding (6.9) and (6.10), we have

$$E\{\text{total number of nodes}\} \simeq \frac{1}{K \ln 2}[2S + Q] \simeq \frac{2S}{K \ln 2} \tag{6.11}$$

Since there are three link fields associated with each node, total link storage is approximately

$$\frac{6 S \log_b S}{K \ln 2} \text{ b-ary digits} \tag{6.12}$$

For a related algorithm, Larson [61] shows that about 96% of the random records (in our case 96% of the paths) are located on levels 2 and 3 of the trie. This further confirms the conclusion that multiple splits are very rare, as was already argued. Also, to insert paths into buckets one need not traverse arbitrarily long chain of buckets.

Thus the number of accesses to storage required in order to insert or delete paths is bounded by a constant most of the time. The cost of the dynamic bucket algorithm is

Time Cost: C_1 accesses/branch viewed

C_2 S accesses/branch released, for ambiguity check

$$\text{Space Cost: } S\ (C_3\ L + \frac{6\ \log_b S}{K\ \ell n\ 2} + \log L + p). \qquad (6.13)$$

Since the buckets may only be partially filled most of the time, in order to store S paths, a storage greater than S is necessary. The constant C_3 in (6.13) accounts for this extra storage. C_3 is likely to be about 1.4 [61]. The total product cost is

$$O(LSE[C_{SA}]) + O(LS^2) + H_d \text{ access-symbols/branch released} \qquad (6.14)$$

where H_d is the cost of hashing, splitting, and merging. It is assumed that $K \geq 6/\ell n2$ and that L dominates $\log_b S$.

The sum cost of the algorithm is

$$O(LS) + O(S) + O(E[C_{SA}]) + H_d. \qquad (6.15)$$

The precise number of comparisons is difficult to estimate but it is of $O(S) + O(E[C_{SA}])$.

It is likely that H_d in (6.14) is greater than the H factor in (6.2) for the bucket algorithm. But, due to its dynamic nature, the dynamic bucket algorithm comes closer to being metric-first than does the bucket algorithm. It is well known that of all the known algorithms the metric-first

ones have the least node computation [30]. We can thus expect the E[C] for the dynamic bucket algorithm to be less than the E[C] for the bucket algorithm. This may more than compensate for the slight increase in H_d over H. The dynamic bucket algorithm again establishes an order of dependence on S for the product cost which is just S, the lowest for the metric-first algorithms so far considered. A summary of the results of Chapters 4, 5, and 6 appeared in Mohan and Anderson [63].

CHAPTER 7

BRANCHING PROCESS METHODS FOR THE SINGLE STACK ENCODING ALGORITHM

7.1 Introduction

Initially, interest in rate-distortion theory centered around developing rate-distortion functions for different sources. Only ten years after the birth of the theory was attention given to one of the two facets of practical source coding, the design of codes for sources, by Jelinek [11], who proved the existence of tree codes that achieve the rate-distortion bound. Jelinek's proof was valid only for symmetric sources, and it remained for Davis and Hellman [50], making use of the theory of branching processes in random environments (BPRE) [64], to prove that for any i.i.d. source, tree codes exist whose performance is as close to the R(D) curve as desired. Tan [65], again using the BPRE theory [68], showed that such tree codes exist for stationary block-ergodic sources. Viterbi and Omura [21] have shown the existence of time-varying trellis codes that achieve R(D). A recent paper by Johannesson [66] uses the theory of multitype Galton-Watson branching processes to generate the computational distribution for

sequential decoding using the stack algorithm. Since the paper by Jelinek [11] appeared in the literature, attention has also centered around the second facet of source coding, the design and analysis of source coding algorithms (see Anderson [17], Anderson and Jelinek [16], and Gallager [18]). While these papers consider only symmetric sources, our interest here is in applying the BPRE theory to the single stack encoding algorithm applied to asymmetric sources, with the aim of deriving an expression for the number of tree branches visited. This expression is shown to be the stochastic analog of an expression given by Gallager for the case of symmetric sources. Some intriguing simulation results for the algorithm and its variants, which cast light on the possible range of solutions to our equations, are presented.

7.2 Preliminaries

Rate-Distortion Theory

Let a discrete memoryless source (i.i.d. source) have a probability distribution $P(x)$ defined on elements of a source alphabet X. Let an additive single letter distortion measure $d(x,y)$ be defined on elements of source and reproduction alphabets, X and Y. Let $Q(y|x)$ be a conditional probability assignment and $I(X;Y)$ the mutual information between X and Y.

Definition: The rate-distortion function R(D) is the minimum rate necessary to encode the information source with additive single letter distortion measure so that the average distortion does not exceed some distortion D; it is given by

$$R(D) \triangleq \inf_{Q \in Q_D} I(X;Y) \tag{7.1}$$

where

$$Q = \{Q_D(y|x): \sum_{x,y} P(x)Q(y|x)d(x,y) \leq D\} , \tag{7.2a}$$

and

$$I(X;Y) = \sum_{x,y} P(x)\, Q(y|x)\, \ell n \frac{P(x)\, Q(y|x)}{P(x)\, Q(y)} . \tag{7.2b}$$

Haskell [67] has shown that

$$R(D) = \max_{\rho \geq 0} \min_{Q(y)} - \sum_{x} P(x)\, \ell n \{\sum_{y} Q(y)\, \exp\, [-\rho(d(x,y)-D)]\} \tag{7.3}$$

where Q(y) is a probability density defined on the reproduction alphabet given by

$$Q(y) = \sum_{x} P(x)\, Q(y|x) . \tag{7.4}$$

Let ρ_o and $Q_o(y)$ optimize the expression in (7.3). Then we have the following equations [10, pp. 34-37].

$$Q_o(y|x) = Q_o(y)\, \exp\, [-\rho_o\, d(x,y)]\, \lambda(x) \tag{7.5}$$

$$\lambda(x) = \{\sum_{y} Q_o(y)\, \exp\, [-\rho_o\, d(x,y)]\}^{-1} \tag{7.6}$$

$$Q_o(y) = \sum_{x} P(x)\, Q_o(y|x) \tag{7.7}$$

$$R(D) = -\rho_o D + \sum_x P(x) \ln \lambda(x) \quad . \qquad (7.8)$$

Branching Processes with Random Environments

We will find in Sec. 7.3 that branching processes with random environments (BPRE), formulated by Smith and Wilkinson [64] and extended by Athreya and Karlin [68], are an appropriate tool for analyzing the single stack encoding algorithm. So we briefly describe the working of a BPRE. Harris [69] and Athreya and Ney [70] are excellent sources on the theory of branching processes. The book by Mode [71] is especially devoted to multitype branching processes and that by Jagers [72] to biological applications of branching processes. Refer to Feller [73, pp. 293-301] for introductory material on simple Galton-Walson branching processes.

Suppose we have Z_o particles at time $n = 0$ (or generation 0). Each of these Z_o particles creates further particles so that the population size at the first generation is

$$Z_1 = \sum_{i=1}^{Z_o} X_{1i} \qquad (7.9)$$

where X_{1i}, $i = 1, 2, \ldots, Z_o$ are independent and identically distributed random variables with probability generating function (p.g.f.) $\phi_{\zeta o}(\underset{\sim}{s})$. Here

$$\phi_{\zeta o}(\underset{\sim}{s}) = \sum_{j=0}^{\infty} p(j|\zeta_o)\ s^j \qquad (7.10)$$

where $p(j|\zeta_o)$ is the probability that a zeroth generation particle gives rise to j first generation particles, given environment ζ_o. The Z_1 first generation particles then give rise to second generation particles according to p.g.f. $\phi_{\zeta 1}(\underset{\sim}{s})$. Continuing in this way, the (n+1)th generation population is the cumulative sons or progenies of the Z_n nth generation particles, each reproducing according to p.g.f. $\phi_{\zeta n}(\underset{\sim}{s})$. $\{\zeta_n,\ n = 0, 1, 2, \ldots\}$ is called the <u>environmental process</u>. In the case of an i.i.d. environmental process, $\{\phi_{\zeta n}(\underset{\sim}{s})\}$ are all identical, $\{\zeta_n\}$ are i.i.d., and Z_n, n = 0, 1, 2, ... is a branching process developing in an i.i.d. random environment. We can visualize the Z_n process as one developing in a stochastically changing environment that affects the reproductive behaviour of the process.

Allowing for generalization, we now stipulate that a particle may be any one of a number of <u>types</u>, say m types. Thus, starting from a zeroth generation particle of type i, we have at the first generation γ_1 particles of type 1, γ_2 of type 2, ..., γ_m of type m produced according to p.g.f.

$$\phi_{\zeta_o}^{(i)}(\underset{\sim}{s}) = \phi(\underset{\sim}{s}|\zeta_o,\ z_o = i) \qquad (7.11)$$

$$= \sum_{\gamma_1,\gamma_2,\ldots\gamma_m} p_i\ (\gamma_1,\gamma_2,\ldots\gamma_m|\zeta_o)\ s_1^{\gamma_1}\ s_2^{\gamma_2}\ \ldots\ s_m^{\gamma_m}$$

where $\underset{\sim}{s} = (s_1, s_2, \ldots, s_m)$, $0 \leq s_i \leq 1$, $i = 1, \ldots, m$. Again a type j particle at the first generation reproduces according to p.g.f. $\phi_{\zeta 1}^{(j)}(\underset{\sim}{s})$, and so on. Thus we have a vector process defined by $\underset{\sim}{Z}_0 = \underset{\sim}{e}_i = (0, 0, \ldots, 1, \ldots, 0)$, a vector with 1 in the ith place and 0's elsewhere, and $\underset{\sim}{Z}_1 = (Z_1(i,1), Z_1(i,2), \ldots, Z_1(i,m))$ etc, where $Z_1(i,\gamma)$ is the number of first generation particles of type γ from a zeroth generation particle of type i. In general, if $\underset{\sim}{Z}_n = (Z_n(i,1), Z_n(i,2), \ldots, Z_n(i,m))$, then the (n+1)th generation population vector Z_{n+1} is the sum of $Z_n(i,1) + Z_n(i,2) + \ldots + Z_n(i,m)$ independent random vectors, where each of the $Z_n(i,k)$, $k = 1, 2, \ldots, m$, independent random vectors is produced according to the probability assignment $p_K(\gamma|\zeta_n)$ where γ is an m vector. Again, when $\{\zeta_n\}$ is i.i.d., $\underset{\sim}{Z}_n$, $n = 0, 1, ..,$ is a multitype (m-type) branching process in random environment.

The above process is said to be extinguished if $\underset{\sim}{Z}_n = \underset{\sim}{0}$ for some n, where $\underset{\sim}{0} = (0, 0, \ldots, 0)$ is an m vector. If the extinction probability q_i of the process, starting from an initial particle of type i, is defined as

$$q_i \triangleq P[\underset{\sim}{Z}_n = 0 \text{ for some } n \mid \underset{\sim}{Z}_o = e_i], \tag{7.12}$$

then we can associate a probability vector $\underset{\sim}{q}$ with the process, given by $\underset{\sim}{q} = (q_1, q_2, \ldots, q_m)$. If $\underset{\sim}{\zeta} = (\zeta 0, \zeta 1, \ldots)$, then from [5] $\underset{\sim}{q}(\underset{\sim}{\zeta}) = (\underset{\sim}{q}_1(\underset{\sim}{\zeta}), \underset{\sim}{q}_2(\underset{\sim}{\zeta}) \ldots \underset{\sim}{q}_m(\underset{\sim}{\zeta}))$, the

extinction probability vector conditioned on the environment $\underset{\sim}{\zeta}$, satisfies the functional equation

$$\underset{\sim}{q}(\underset{\sim}{\zeta}) = \lim_{n\to\infty} \underset{\sim}{\phi}_{\zeta 0}(\underset{\sim}{\phi}_{\zeta 1}(\cdots \underset{\sim}{\phi}_{\zeta n}(\underset{\sim}{s}))\ldots)) \tag{7.13}$$

$$= \underset{\sim}{\phi}_{\zeta 0}(\underset{\sim}{q}(T\underset{\sim}{\zeta})) \tag{7.14}$$

where T is the shift operator and $T\underset{\sim}{\zeta} = (\zeta 1, \zeta 2, \ldots)$ and the vector $\underset{\sim}{\phi}$ is given by $\underset{\sim}{\phi}_{\zeta\gamma}(\underset{\sim}{s}) = (\phi^{(1)}_{\zeta\gamma}(\underset{\sim}{s}), \ldots, \phi^{(m)}_{\zeta\gamma}(\underset{\sim}{s}))$. $\underset{\sim}{q}$, the unconditional probability of extinction, is then $\underset{\sim}{q} = E[\underset{\sim}{q}(\underset{\sim}{\zeta})]$.

While the papers [64] and [68] deal with conditions for certain and noncertain extinction and with proving certain limit theorems, our interest here is to model the single stack encoding algorithm as a multi-type BPRE and to derive an expression for the number of branches visited.

Random Tree Codes

Consider random tree codes with b branches out of each node and β symbols on each branch. The symbols are chosen from reproduction alphabet Y according to the distribution $Q_o(y)$ given in (7.7). The rate R of such a tree code is given by

$$R \triangleq \frac{\log_2 b}{\beta} \text{ bits/symbol.} \tag{7.15}$$

If D_o is the desired average distortion at the end of encoding, then R is chosen to satisfy $R \geq R(D_o)$, where $R(\cdot)$

is defined in (7.8). Given any $\varepsilon > 0$, the object of encoding a given source is to find a path through the code tree that has an average distortion $D \leq D_o + \varepsilon$. Associated with a path $\underset{\sim}{y}^{\ell}$ of ℓ/β branches is a quantity called the metric of the path defined by

$$\mu(\underset{\sim}{y}^{\ell}) \triangleq D(\underset{\sim}{x}^{\ell}) - d(\underset{\sim}{x}^{\ell}, \underset{\sim}{y}^{\ell}) \tag{7.16}$$

where $D(\underset{\sim}{x}^{\ell})$ is defined as

$$D(\underset{\sim}{x}^{\ell}) \triangleq \sum_{\underset{\sim}{y}^{\ell}} d(\underset{\sim}{x}^{\ell}, \underset{\sim}{y}^{\ell})\, Q_o(\underset{\sim}{y}^{\ell} | \underset{\sim}{x}^{\ell}). \tag{7.17}$$

Here $\underset{\sim}{x}^{\ell}$ and $\underset{\sim}{y}^{\ell}$ refer to length ℓ source and reproduction symbols respectively. Also, if

$$\underset{\sim}{x}^{\ell} = (x_{11}, x_{12}, \ldots, x_{1\beta}, x_{21}, \ldots, x_{2\beta}, \ldots, x_{\ell/\beta,1}, \ldots, x_{\ell/\beta,\beta})$$

$$= (\underset{\sim}{x}_1^{\beta}, \underset{\sim}{x}_2^{\beta}, \ldots, \underset{\sim}{x}_{\ell/\beta}^{\beta})$$

and $\underset{\sim}{y}^{\ell}$ is similarly defined, then

$$D(\underset{\sim}{x}^{\ell}) = \sum_{i=1}^{\ell/\beta} D(\underset{\sim}{x}_i^{\beta}) \tag{7.18}$$

and

$$d(\underset{\sim}{x}^{\ell}, \underset{\sim}{y}^{\ell}) = \sum_{i=1}^{\ell/\beta} d(\underset{\sim}{x}_i^{\beta}, \underset{\sim}{y}_i^{\beta}) \tag{7.19}$$

Thus

$$\mu(\underset{\sim}{y}^{\ell}) = \sum_{i=1}^{\ell/\beta} \mu(\underset{\sim}{y}_i^{\beta}) \quad . \tag{7.20}$$

Here the $d(\underset{\sim}{x}_i^{\beta}, \underset{\sim}{y}_i^{\beta})$'s and $\mu(\underset{\sim}{y}_i^{\beta})$'s are branch distortion values and branch metric increments, respectively.

In defining the metric of a code tree path, we have used the $D(\underset{\sim}{x}^{\ell})$ criterion instead of the traditional one using D_o. This criterion was first suggested by Berger [10, p. 220]; Dick et al. [53] used it to encode Gaussian sources, while Davis and Hellman [50] proved it sufficient for asymmetric sources and distortion measures.

Next, in order to model the algorithm as a multitype BPRE, we quantize the metric values of paths. Where no confusion will arise, we will still call the quantized metric as metric only and denote it by the same symbol μ. Thus, redefining the metric, we have

$$\mu(\underset{\sim}{y}^{\ell}) = \frac{1}{\xi}\left\lceil D(\underset{\sim}{x}^{\ell}) - d(\underset{\sim}{x}^{\ell}, \underset{\sim}{y}^{\ell})\right\rceil \qquad (7.21)$$

where $\lceil a \rceil$ is the smallest integer N such that $a \leq N$ and ξ is an arbitrary small positive number. Still,

$$\mu(\underset{\sim}{y}^{\ell}) = \sum_{i=1}^{\ell/\beta} \mu(\underset{\sim}{y}_i^{\beta}).$$

7.3 Analysis of the Single Stack Encoding Algorithm

The single stack algorithm (see Chapter 2) is essentially a depth-first search procedure, trying to explore along the depth of the code tree a path that satisfies a discard criterion. When the path falls below a lower barrier B, the algorithm backtracks along the path to

another node from where the search again proceeds in a depth-first fashion. For the purposes of analysis we introduce an upper barrier A and consider only paths that lie within the barriers B and A. Because of the definition in (7.21), μ is now allowed to take only integer values lying between B and A. There are (A-B+1) such metric values. We denote the set of allowed metric values as N(B,A). Introduction of upper barrier A will not appreciably alter the behaviour of the algorithm if A is considered to be very large. We assume further that the algorithm works successively on source blocks of ℓ symbols.

Now we turn to a BPRE model of the algorithm. At any stage during encoding we can consider code tree paths as particles and their metrics as the particle types. Thus we have an m = (A-B+1) type process. For the symmetric source case, all source blocks of length ℓ correspond to the same environment. In the case of asymmetric sources this is no longer true. If a block of length ℓ source sequence is an atypical one (i.e. a hostile environment) it may be particularly difficult to find a code tree path of length ℓ/β branches that lies within the barriers. In such a case the algorithm may explore a large number of code tree branches. If one were to consider paths of length ℓ/β branches whose metrics belong to the set N(B,A) as offspring of the root node, then it is clear that the distribution of

offspring varies from generation to generation (i.e. from one ℓ-block to another ℓ-block). Hence the behaviour of the algorithm within a block is dependent on the source sequence $\underset{\sim}{x}^{\ell}$ corresponding to that block. The "environment" within a block can be identified with $\underset{\sim}{x}^{\ell}$, and successive environments vary in an i.i.d. manner, since $\underset{\sim}{x}^{\ell}$'s are i.i.d. Thus we have an m = (A-B+1) type BPRE.

Equivalently, note that, for asymmetric sources, the metrics of successors of a single node at a given level in the code tree are dependent on the source letter present at that level. If the vertical axis represents metrics of code tree paths and the horizontal axis time or code tree levels, a node extension maps its successors on the vertical axis depending on their metrics. Such a process is called a branching random walk (BRW) for the symmetric case and branching random walk in random environments (BRWRE) for the asymmetric case. We can identify the type with the metric of nodes and convert the BRWRE into a multitype BPRE. Results based on the ℓ-block environment approach appeared in Mohan and Anderson [74]. Here, we use the BRWRE approach, where environment is identified with β source letters, i.e., $\zeta_t = \underset{\sim}{x}_t^{\beta}$. Both approaches yield similar results.

The single-type BPRE formulation of Jelink's tree coding process was first given by Davis and Hellman [50],

and we have analysed a new coding process, the single stack algorithm, in a similar manner, but as a multi-type BPRE. Our asymmetric case analysis is similar in form to the one employed by Gallager [18] for the symmetric case, and we have used several of his results.

Equation for the Probability Generating Function of a BRWRE

Let $P(j-i|\zeta) \triangleq P$ {an immediate descendent of a node with metric i has metric j, given environment ζ}

Let $z(i,j) \triangleq$ the number of the b descendents of a node with metric i that have metric j.

Let $X_\ell(i,j) = 1$, if the ℓth descendent of a node with metric i has metric j

$= 0$, otherwise.

From (7.11), the probability generating function,

$$\phi_\zeta^{(i)}(\underset{\sim}{s}) = E\left[\prod_{j=1}^{m} s_j^{z(i,j)} | \zeta\right] \tag{7.22}$$

$$= E\left[\prod_{j=1}^{m} s_j^{\sum_{\ell=1}^{m} X_\ell(i,j)} | \zeta\right] \tag{7.23}$$

$$= E\left[\prod_{j=1}^{m}\prod_{\ell=1}^{b} s_j^{X_\ell(i,j)} | \zeta\right] \tag{7.24}$$

$$= \prod_{\ell=1}^{b} E\left[\prod_{j=1}^{m} s_j^{X_\ell(i,j)} | \zeta\right] \tag{7.25}$$

In (7.22), we have assumed, without loss of generality, that there are m types, i.e., $N(B,A) = \{1, 2, \ldots, m\}$. In (7.25), we have made use of the fact that, when conditioned on the environment, different descendents move independently of each other.

When conditioned on the environment, $X_\ell(i,j) = 1$ with probability $P(j-i|\zeta)$. The ℓth descendent is absorbed by the barriers with probability

$$1 - \sum_{j=1}^{m} P(j-i|\zeta)$$

i.e., $X_\ell(i,j) = 0$ with probability

$$1 - \sum_{j=1}^{m} P(j-i|\zeta),$$

when conditiond on the environment. If $X_\ell(i,j) = 1$, the product

$$\prod_{j=1}^{m} s_j^{X_\ell(i,j)}$$

inside the expectation in (7.25) is s_j, and if $X_\ell(i,j) = 0$, it is 1. Hence,

$$E\left[\prod_{j=1}^{m} s_j^{X_\ell(i,j)} |\zeta\right] = 1 - \sum_{j=1}^{m} P(j-i|\zeta) + \sum_{j=1}^{m} P(j-i|\zeta)\, s_j \qquad (7.26)$$

Combining (7.25) and (7.26), we have

$$\phi_{\zeta}^{(i)}(\underset{\sim}{s}) = [1 - \sum_{j=1}^{m} P(j-i|\zeta)\ (1-s_j)]^b \tag{7.27}$$

Equation (7.27) will form the basis of the branching process in random environments generated by the algorithm.

The Moment Generating Function of the Metric of a Branch

Define the moment generating function of the metric of a branch conditioned on environment ζ as follows:

$$g_{\zeta}(r) = \sum_{j=1}^{m} P(j|\zeta)\ \exp(rj) \tag{7.28}$$

$$= E\ [\exp(rj)|\zeta] \tag{7.29}$$

Using the definition of metric of a branch from (7.21), we have

$$g_{\zeta}(r) \geq E\ [\exp\{\frac{r}{\xi}\ (D(\underset{\sim}{x}^{\beta}) - d(\underset{\sim}{x}^{\beta}, \underset{\sim}{y}^{\beta})\}] \tag{7.30}$$

$$= \sum_{\underset{\sim}{y}^{\beta}} Q_o\ (\underset{\sim}{y}^{\beta})\ \exp\{\frac{r}{\xi}\ (D(\underset{\sim}{x}^{\beta}) - d(\underset{\sim}{x}^{\beta}, \underset{\sim}{y}^{\beta})\} \tag{7.31}$$

On taking logarithms on both sides of (7.31) and finding the expectation of log $g_{\zeta}(r)$ over all possible enviroments $\underset{\sim}{x}^{\beta}$, we get

$$E[\log g_{\zeta}(r)] \geq \sum_{\underset{\sim}{x}^{\beta}} P(\underset{\sim}{x}^{\beta})\ \ell n \sum_{\underset{\sim}{y}^{\beta}} Q_o(\underset{\sim}{y}^{\beta})$$

$$\cdot\ \exp\{\frac{r}{\xi}\ (D(\underset{\sim}{x}^{\beta}) - d(\underset{\sim}{x}^{\beta}, \underset{\sim}{y}^{\beta})\} \tag{7.32}$$

$$= \frac{r}{\xi} \sum_{\underset{\sim}{x}^{\beta}} P(\underset{\sim}{x}^{\beta})\ D(\underset{\sim}{x}^{\beta}) + \sum_{\underset{\sim}{x}^{\beta}} P(\underset{\sim}{x}^{\beta})\ \ell n \sum_{\underset{\sim}{y}^{\beta}} Q_o(\underset{\sim}{y}^{\beta})$$

$$\cdot \exp\left(-\frac{r}{\xi}\, d(\underset{\sim}{x}^{\beta}, \underset{\sim}{y}^{\beta})\right) \qquad (7.33)$$

Substituting βD_o for

$$\sum_{\underset{\sim}{x}^{\beta}} P(\underset{\sim}{x}^{\beta})\ D(\underset{\sim}{x}^{\beta})$$

and ρ_o for r/ξ in (7.33) and making use of (7.6) and (7.8), we get

$$E[\ell n\ g_{\zeta}(r)] = -\beta\ R(D_o) \qquad (7.34)$$

$$= -\beta\ [R_n - \gamma] \qquad (7.35)$$

where $R_n = (\ell n\ b)/\beta = R\ \ell n\ 2$ and $\gamma = R_n - R(D_o) > 0$.
In (7.34), we have made use of the fact, proven by Gallager [18], that $\rho_o = r/\xi$ optimizes (7.3). Rearranging (7.35), one gets

$$E[\ell n\ bg_{\zeta}(r)] \geq \beta\ \gamma > 0 \qquad (7.36)$$

Gallager [18] has shown that $bg(r) > 1$ for the symmetric case. He further shows that the condition $b\ g(r) > 1$ is a necessary condition for the branching process generated by the algorithm to have a probability of extinction strictly less than 1. In (7.36) we have derived an analogous condition for the asymmetric case. While we have not yet proved so , we can hope that the condition $E[\ell n\ bg_{\zeta}(r)] > 0$ is necessary for the BPRE generated by the algorithm to have an extinction probability strictly less than 1. With $q < 1$ it can be shown that the algorithm

achieves R(D).

Equation for Node Computation by the Single Stack Algorithm

A node is said to be visited by the algorithm when it is pushed down onto the stack during forward motion in the code tree. Define the node computation as the total number of nodes visited. A branch may be traversed either in the forward direction when a node is pushed onto the stack, or in the reverse direction when a node is popped up from the stack. Any branch may at most be traversed twice. Since there are b branches out of a node, the total number of branches visited is upperbounded by 2b times the node computation.

Theorem: Let $C_i(\underset{\sim}{\zeta}_n)$ be the node computation forward of a node with metric i needed to encode $(n+1)\beta$ source letters, given environment $\underset{\sim}{\zeta}_n = (\zeta_0, \zeta_1, \ldots, \zeta_n) = (\underset{\sim}{x}_0^{\beta}, \underset{\sim}{x}_1^{\beta}, \ldots, \underset{\sim}{x}_n^{\beta})$. Then, for $n \geq 2$,

$$C_i(\underset{\sim}{\zeta}_n) = 1 + U_i(\underset{\sim}{\zeta}_n) + U_i(\underset{\sim}{\zeta}_n)\ \rho_i(\underset{\sim}{\zeta}_n)$$

$$+ U_i(\underset{\sim}{\zeta}_n)\ \{\rho_i(\underset{\sim}{\zeta}_n)\}^2 + \ldots + U_i(\underset{\sim}{\zeta}_n)\ \{\rho_i(\underset{\sim}{\zeta}_n)\}^{b-1} \qquad (7.37)$$

where

$$U_i(\underset{\sim}{\zeta}_n) = \sum_{j=1}^{m} P(j-i|\zeta_o)\ C_j(T\underset{\sim}{\zeta}_n), \qquad (7.38)$$

$$\rho_i(\underset{\sim}{\zeta}_n) = 1 - \sum_{j=1}^{m} P(j-i|\zeta_o)\ \{1 - q_j(T\underset{\sim}{\zeta}_n)\}, \qquad (7.39)$$

and T is the shift operator given by

$$T\underset{\sim}{\zeta}_n = (\zeta_1, \zeta_2, \ldots, \zeta_n) = (\underset{\sim}{x}_1^{\beta}, \underset{\sim}{x}_2^{\beta}, \ldots, \underset{\sim}{x}_n^{\beta}).$$

Proof: The first descendent of the root node, conditioned on the environment $\zeta_o = \underset{\sim}{x}_o^{\beta}$, has metric j with probability $P(j-i|\zeta_o)$, where i is the metric of the root node. Since the environment forward of the first descendent is $T\underset{\sim}{\zeta}_n = (\zeta_1, \zeta_2, \ldots, \zeta_n) = (\underset{\sim}{x}_1^{\beta}, \underset{\sim}{x}_2^{\beta}, \ldots, \underset{\sim}{x}_n^{\beta})$, the node computation forward of it is $C_j(T\underset{\sim}{\zeta}_n)$. Since the first descendent can have any one of m allowed metric values, the unconditional node computation forward of it is

$$U_i(\underset{\sim}{\zeta}_n) = \sum_{j=1}^{m} P(j-i|\underset{\sim}{\zeta}_n)\ C_j(T\underset{\sim}{\zeta}_n),$$

the second term in (7.37), where we have summed out the conditioning on its metric. Assuming, for the time being, that n is constant, the algorithm searches forward of the second descendent of the root node only if the BPRE forward of the first descendent extinguishes itself before level (n+1) forward of the root node at level 0. The probability of this happening, given that the first descendent has metric j, is $q_j(T\underset{\sim}{\zeta}_n)$. Removing the conditioning on the metric of the first descendent, the unconditioned probability of extinction is

$$\sum_{j=1}^{m} P(j-i|\zeta_o)\ q_j(T\xi_n).$$

The BPRE forward of the first descendent is never started if the first descendent is absorbed by the barriers. This happens with probability

$$1 - \sum_{j=1}^{m} P(j-i|\zeta_o).$$

Thus the probability that the algorithm fails to find a path of length n forward of the first descendent is

$$\rho_i(\xi_n) = 1 - \sum_{j=1}^{m} P(j-i|\zeta_o)\ \{1 - q_j(T\ \xi_n)\},$$

which is also the probability that the algorithm searches forward of the second descendent.

Given that the algorithm has failed to find a path of length n forward of the first descendent, the node computation forward of the second descendent is $U_i(\xi_n)$ by similar arguments. Unconditionally the node computation forward of the second node is $U_i(\xi_n)\ \rho_i(\xi_n)$, the third term in (7.37). Since the third descendent is searched if both the BPRE's forward of the first and second descendents fail and this happens with probability $\{\rho_i(\xi_n)\}^2$, we see that the fourth term in (7.37) is the unconditional node computation forward of the third descendent. The last term in (7.37) is the node computation forward of the bth descendent of the

root node. Accounting for the root node (1 in (7.37)), the theorem follows.

From [68] $q_i(\underset{\sim}{\zeta}) \geq q_i(\underset{\sim}{\zeta}_n)$ for $n = 1, 2, \ldots$.

Using this in (7.39), we have

$$\rho_i(\underset{\sim}{\zeta}_n) \leq 1 - \sum_{j=1}^{m} P(j-i|\zeta_o) \{1 - q_j(T\underset{\sim}{\zeta})\}$$

$$\triangleq \nu_i(\underset{\sim}{\zeta}) \tag{7.40}$$

Using (7.40) in (7.37), we get

$$C_i(\underset{\sim}{\zeta}_n) \leq 1 + U_i(\underset{\sim}{\zeta}_n) \sum_{k=0}^{b-1} \{\nu_i(\underset{\sim}{\zeta})\}^k \tag{7.41}$$

$$= 1 + U_i(\underset{\sim}{\zeta}_n) \frac{[1 - \{\nu_i(\underset{\sim}{\zeta})\}^b}{[1 - \nu_i(\underset{\sim}{\zeta})]} \tag{7.42}$$

But

$$\nu_i(\underset{\sim}{\zeta}) = 1 - \sum_{j=1}^{m} P(j-i|\zeta_o) \{1 - q_j(T\underset{\sim}{\zeta})\}$$

$$= \phi_{\zeta_o}^{(i)} (\underset{\sim}{q}(T\underset{\sim}{\zeta})) \tag{7.43}$$

$$= q_i(\underset{\sim}{\zeta}) . \tag{7.44}$$

Equation (7.43) follows from (7.27) and (7.44) from (7.14). Combining equations (7.38), (7.42), and (7.44), we get

$$C_i(\underset{\sim}{\zeta}_n) \leq \sum_{j=1}^{m} \frac{P(j-i|\zeta_o)\, C_i(T\underset{\sim}{\zeta}_n)\, (1 - q_i(\underset{\sim}{\zeta}))}{\sum_{k=1}^{m} P(k-i|\zeta_o) \{1 - q_k(T\underset{\sim}{\zeta})\}} \tag{7.45}$$

Let $B(i,j)$ = (i,j)th element of matrix B

$$= \frac{P(j-i|\zeta_o)\ (1 - q_i(\underset{\sim}{\zeta}))}{\sum_{k=1}^{m} P(k-i|\zeta_o)\ \{1 - q_k(T\underset{\sim}{\zeta})\}} \tag{7.46}$$

Let

$$\underset{\sim}{C}(\underset{\sim}{\zeta}_n) = (C_1(\underset{\sim}{\zeta}_n),\ C_2(\underset{\sim}{\zeta}_n),\ \ldots,\ C_n(\underset{\sim}{\zeta}_n)) \tag{7.47}$$

Then, from (7.45)-(7.47), we have

$$\underset{\sim}{C}(\underset{\sim}{\zeta}_n) \leq \underset{\sim}{1} + [B(\underset{\sim}{\zeta})]\ \underset{\sim}{C}(T\underset{\sim}{\zeta}_n), \tag{7.48}$$

where $[B(\underset{\sim}{\zeta})]$ is the matrix defined in (7.46) and $\underset{\sim}{1}$ = (1, 1, ..., 1) is an all 1 m vector.

Gallager derives the equation $\underset{\sim}{C}_n \leq \underset{\sim}{1} + [B]\ \underset{\sim}{C}_{n-1}$ for the symmetric case. Here we have derived the stochastic analog of this equation for the asymmetric case. Heretofore only symmetric sources have been considered in the literature. Though a closed form solution to (7.48) is not yet known, we strongly believe that the behaviour of code tree search algorithms with asymmetric sources can be analyzed using the BPRE methods presented here. Next we present some simulation results in which the single stack algorithm uses the $D(\underset{\sim}{x}^{\ell})$ discard criterion.

7.4 Asymmetric Source Simulations

For the purpose of simulation, we have used the binary i.i.d. source and Hamming distortion measure. The algorithm worked on the ensemble of rate-1/2 tree codes (b =

$\beta = 2$) with code words chosen i.i.d. according to the output probability distribution [10, p. 37]

$$Q_o(0) = (P(0) - D_o)/(1 - 2D_o)$$

$$Q_o(1) = (P(1) - D_o)/(1 - 2D_o)$$

where D_o is given by the distortion-rate function $\Delta(R)$. The algorithm was parameterized by the lower barrier B and the depth limit or block length L. In all our simulation examples, the algorithm encoded a few thousand source samples and used a depth limit of 200 to 1000 tree branches.

Table 7.1 lists simulation results which cast some light on the possible range of solutions to our equations for node computation derived in Sec. 7.3. Result obtained using the binary symmetric source is also given for comparison. The computation in the asymmetric case is much smaller than that for the symmetric case. However, the algorithm attained a high distortion of 39% above $\Delta(R)$ while encoding the source $P(0) = 0.2$ and $P(1) = 0.8$. Conversely, for this source, the algorithm will require a large node computation in order to achieve distortions close to $\Delta(R)$.

While the algorithm seems to have a complicated dependence on various parameters, the following question arises. Asymptotically, does the algorithm behave quite differently when used with asymmetric sources? While we have not yet proved so, we can hope that the algorithm's

behaviour is not much different when used with asymmetric sources and that the node computation E[C], as is the case with many different algorithms used with symmetric sources [17], [16] and [18], is of the form $E[C] = \exp[c(D-D_o)^{-\alpha}]$, where c and α are constants, $D_o = \Delta(R)$, and D is the expected distortion per output symbol for the encoded path.

Table 7.1 Simulation Results for the Single Stack Encoding Algorithm. Binary i.i.d. Source with Hamming Distortion, R = 1/2, L = 200 - 1000.

P(0)	P(1)	B	Branches Viewed	% above $\Delta(R)$
0.5	0.5	-5.5	1500	15.0
		-1.5	355	8.1
0.3	0.7	-3.5	368	4.5
		-7.5	176	9.2
0.2	0.8	-2.5	707	39.0

CHAPTER 8

ENCODING THE BINARY IID SOURCE WITH HAMMING DISTORTION USING THE SINGLE STACK ALGORITHM

8.1 Introduction

Simulation results obtained by encoding the binary i.i.d. source with Hamming distortion using the single stack algorithm are reported here. We study the behaviour of the bias factor or target distortion, D*, stack length, L, level of the lower barrier, B, and number of branches searched per source symbol encoded, $E[C_{SS}]$, on the final distortion attained by the encoder, D_F. Defining stack configuration as the triple (D*, B, L), we find by simulation the optimum stack configuration that minimizes $E[C_{SS}]$ for a given final distortion attained. Two variants of the algorithm, incorporating dynamic raising and lowering of the absorbing lower barrier, are proposed. The effects of D*, L, and $E[C_{SS}]$ on the distortion performance of the variants are studied.

Denote by SSA0, the basic single stack algorithm with the lower barrier fixed at level B, and by SSA1 and SSA2, the algorithm with modifications 1 and 2, respectively, as given below. Let B be the initial level of the lower

barrier and ℓ the relative level of the top most node of the code tree path that resides in the stack.

Modification 1: Whenever, during encoding, the metric of the end node of the path in the stack exceeds the previous maximum attained by a portion of the path residing in the stack (i.e., $\mu(\hat{\underset{\sim}{x}}^{\ell\beta}) > \mu(\hat{\underset{\sim}{x}}^{i\beta})$ for $i = 0, 1, \ldots, \ell-1$), the lower barrier is raised to $\mu(\hat{\underset{\sim}{x}}^{\ell\beta}) - B$; also, when backtracking to the first node of the code tree path in the stack with an unsearched branch, after the path had fallen below the lower barrier, the barrier is lowered to $\mu_m - B$, where $\mu_m = \max \mu(\hat{\underset{\sim}{x}}^{j\beta})$, where $j < \ell$. This modification dynamically raises the lower barrier during forward motion in the code tree and lowers it while backtracking along a code tree path, in a simple version of Fano's algorithm.

Modification 2: Here the barrier may only be raised, i.e., whenever $\mu(\hat{\underset{\sim}{x}}^{\ell\beta}) > \mu(\hat{\underset{\sim}{x}}^{i\beta})$ for $i = 0, 1, \ldots, \ell-1$, the lower barrier is raised to $\mu(\hat{\underset{\sim}{x}}^{\ell\beta}) - B$. Modifications 1 and 2 were reported in [75].

The barrier movements, with and without the modifications to the algorithm, are shown in Fig. 8.1. In Fig. 8.1(a), the numbers on the branches indicate the order in which the algorithm traverses the code tree branches. In Figures 8.1(b) and (c), the barrier levels, generated at different time instants by modifications 1 and 2, respectively, are shown. The effects of such barrier

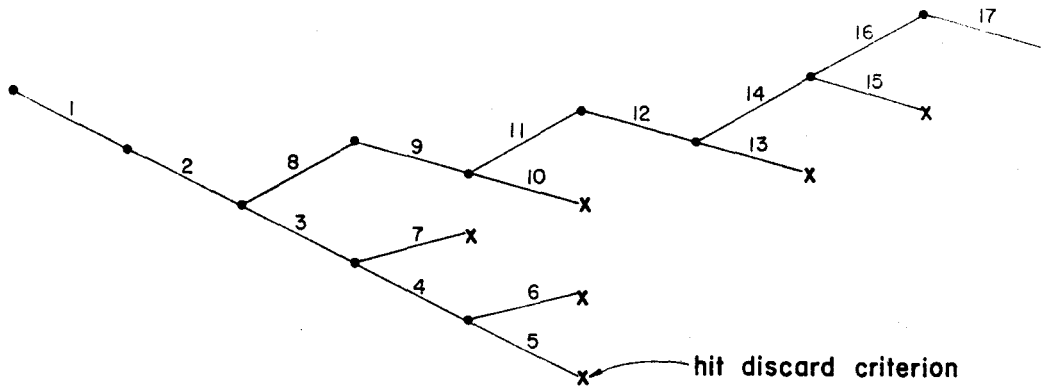


Figure 8.1(a) Push-Down Stack Search, b = 2

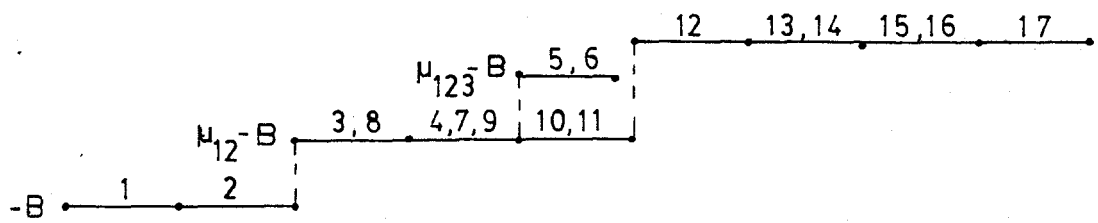


Figure 8.1(b) Barrier Movement with Modification 1

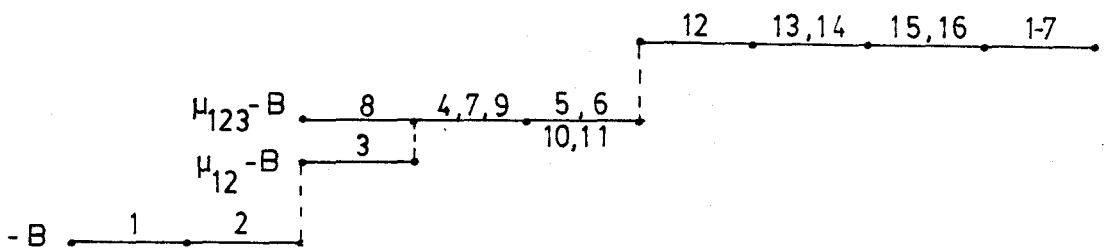


Figure 8.1(c) Barrier Movement with Modification 2

movements on the distortion performance of the algorithm are also investigated.

8.2 Simulation Results - Effects of Length Limit, B Barrier, and Target Distortion

The source chosen for encoding here is an equiprobable binary source. The Hamming distance is used as the distortion criterion, i.e., $d(x,\hat{x}) = \delta_{x,\hat{x}}$, where x is the source bit, $\hat{x}$ the reproducer bit, δ the Kronecker delta, and $d(.,.)$ the distortion between the source and reproducer bits. $R(D) = 1 - H(D)$ for this source, where $H(.)$ is the binary entropy function. Such a source belongs to the class of symmetric sources and the branching process generated by the algorithm is a simple Galton-Watson branching process. Branching process concepts introduced in Chapter 7 are useful in explaining the simulation results.

The code tree branches were populated by a random number generator of the "linear congruential" type that produced 0's and 1's with equal probability. In what follows, D^* and D_F will denote the target distortion and the final distortion actually attained by the algorithm, respectively. Simulated codes have rate 1/2, at which $\Delta(1/2) = 0.110$ is the value of the inverse rate-distortion function.

The remainder of this section shows the effect of the

three basic parameters L, B, and D* on the total distortion performance of the basic single stack algorithm.

Effect of Lower Barrier B on the Distortion Performance of SSAO

The branching process generated by the single stack algorithm (BPSSA) has a greater chance of survival when the barrier is lowered and moved away from the zero level than when it is closer. Consequently, the algorithm scrutinizes more code tree branches as the barrier is lowered and, hence, achieves a better distortion performance. These conclusions are verified to be true from Figs. 8.2(a)-(e), which show the distortion performance of SSAO versus $E[C_{SS}]$ for a fixed D* and for different L. B decreases from -0.5 to -7.5 along each of these curves.

Lowering B below a critical value may not improve the distortion performance. This is due to the fact that the probability of survival of the BPSSA increases and the algorithm is content more often with a poorer path. This is apparent from the curves.

Effect of L on the Distortion Performance of SSAO

The longest code tree path searched by the algorithm and lying above the lower barrier is limited by two parameters: 1) the barrier at level B and 2) the length of

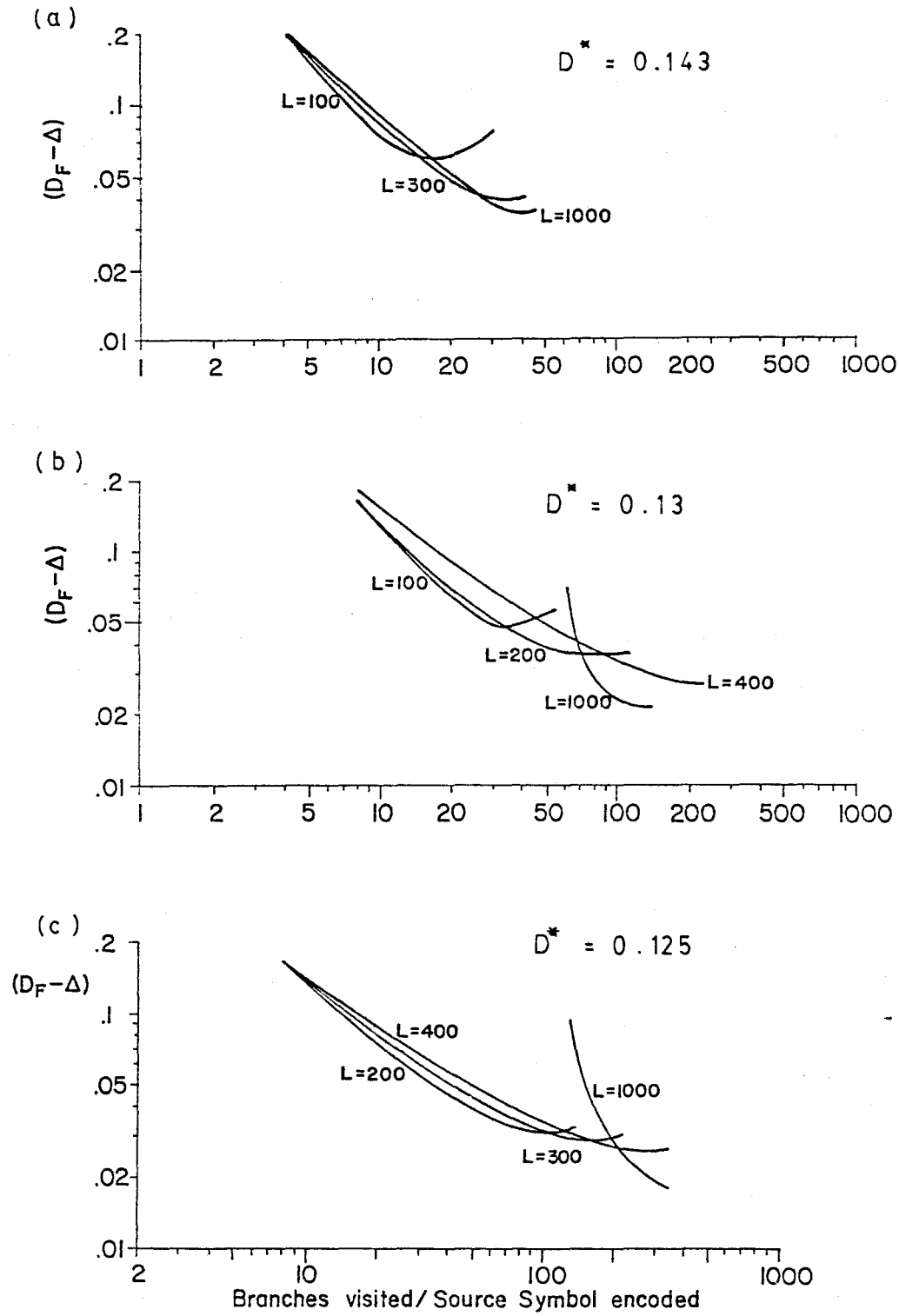
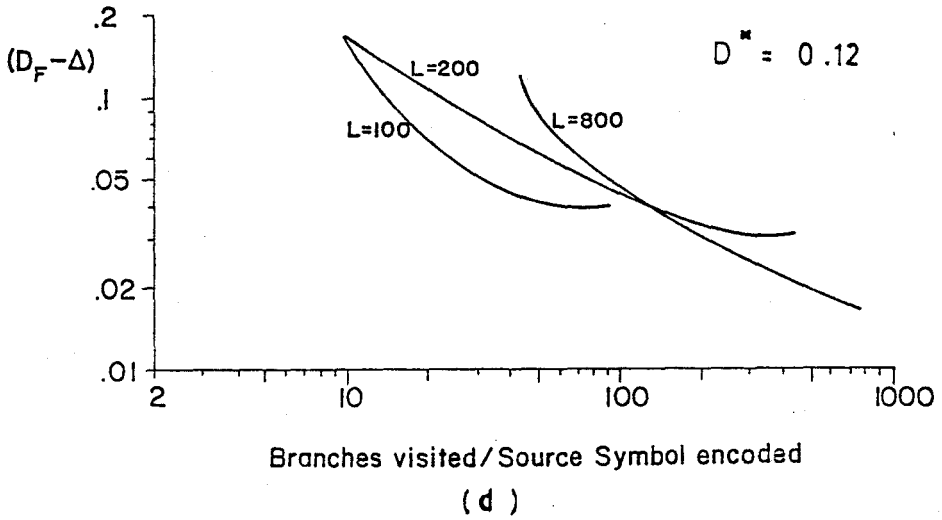


Figure 8.2 Distortion Performance Curves of SSAU ($(D_F-\Delta)$ Versus Branches Visited per Source Symbol Encoded) with Length Limit for Different D*



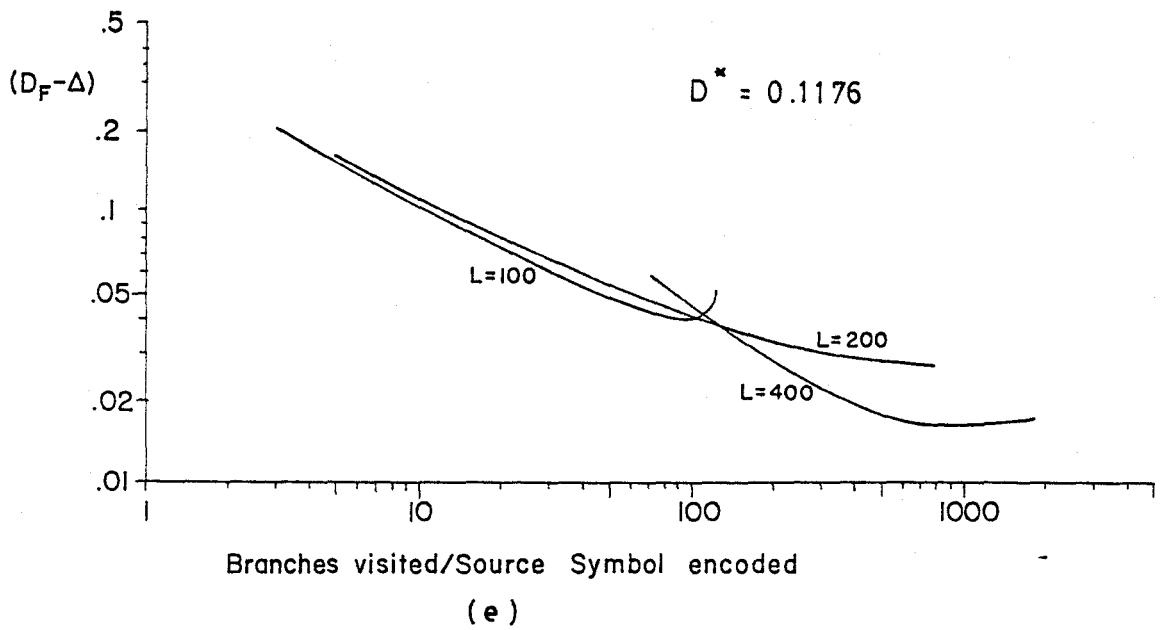


Figure 8.2 Distortion Performance Curves of SSAU ($(D_F-\Delta)$ Versus Branches Visited per Source Symbol Encoded) with Length Limit for Different D*

the stack, L. If B is close to the zero level, the BPSSA may get extinguished before a path of length L is found. For a lower B, L dominates the distortion performance of the algorithm. Assuming that a path above the lower barrier and equal to L in length has been found, we have

$$(\hat{\underset{\sim}{x}}^{L\beta}) = L\beta D^* - d(\underset{\sim}{x}^{L\beta}, \hat{\underset{\sim}{x}}^{L\beta}) > B \qquad (8.1)$$

or

$$d(\underset{\sim}{x}^{L\beta}, \hat{\underset{\sim}{x}}^{L\beta}) < -B + L\beta D^* \qquad (8.2)$$

Dividing both sides of (8.2) by $L\beta$, we get

$$\text{Distortion/source symbol encoded} < D^* + \frac{|B|}{L\beta} \qquad (8.3)$$

Equation (8.2) shows that, providing the branching process survives, as L increases the average distortion per source symbol encoded decreases and tends towards D*. This behaviour is clear in Fig. 8.2(a), which plots $(D_F-\Delta)$ versus $E[C_{SS}]$ for several L. For example, when L = 100, the minimum D_F achieved is 18% above D*, whereas for L = 1000, it is only 3.5% above D*.

It can also be seen from the figure that, for a fixed D*, a given distortion performance can be attained by different (B,L) stack configurations, but only one of them attains it with the least number of computations. For example, for a D_F of 18%, 12%, and 3.5% above D*, L = 100, 300, and 1000 are the optimum stack lengths, respectively,

minimizing the computation $E[C_{SS}]$. For a fixed D*, the envelope of the curves in Fig. 8.2(a) represents the optimum (B,L) stack configurations that minimize the number of branches searched per source symbol encoded. The above conclusions are apparent in Figs. 8.2(b)-(e) as well.

Effect of Target Distortion, D*, on the Distortion Performance of SSAO

Figures 8.2(a)-(e) show the effect of varying D* on the distortion performance of SSAO. The envelopes of curves in Figs. 8.2(a)-(e) are shown in Fig. 8.3. Each curve of Fig. 8.3 specifies the optimum (B,L) stack configurations for a given D*. The envelope of curves in Fig. 8.3 will then represent the optimum (D*, B, L) stack configurations that minimize $E[C_{SS}]$, given D_F. Since D* is the target distortion the smaller the value of D*, the more stringent are the requirements on the distortion performance of the algorithm. Consequently, as D* decreases, the total number of branches generated by the BPSSA and, hence, $E[C_{SS}]$ increase and the algorithm attains distortions closer to Δ. These conclusions are verified from Fig. 8.3. Table 8.1 gives a set of optimum stack configurations obtained from Fig. 8.3.

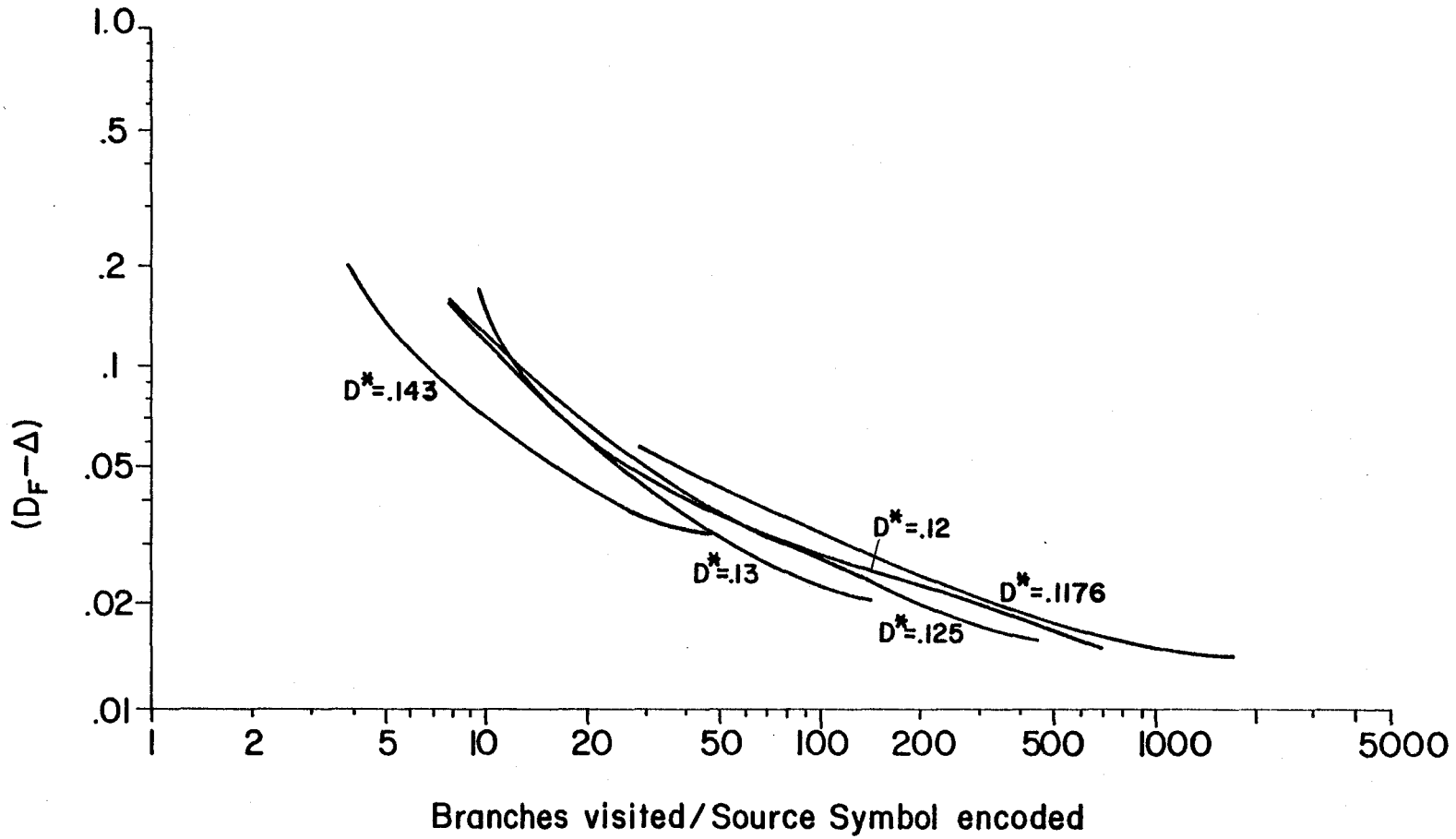


Figure 8.3 Envelopes of Distortion Performance Curves of SSA0 with Length Limit for Different D*; L Increases Along Each Curve

Table 8.1 Optimum (D*, B, L) Stack Configurations from Fig. 8.7 Minimizing $E[C_{SS}]$ (B Vlaues, not shown in the figure, are from simulation results).

$(D_F-\Delta)$	D*	B	L	Minimized $E[C_{SS}]$
0.04	0.143	-4.5	300	27
0.022	0.13	-6.5	1000	150
0.018	0.125	-9.5	1000	350
0.016	0.12	-4.5	800	700

8.3 Effect of B and D* on the Distortion Performances of SSA1 and SSA2

Figures 8.4 and 8.5 show the distortion performances versus branches visited for SSA1 and SSA2, respectively, where B decreases along each curve. We have dispensed with L as a free parameter, since its effect is relatively slight; L is set to 1000.

As B is lowered, $E[C_{SS}]$ increases and better distortion performance is achieved, but B may not be lowered below a critical level. Figures 8.4 and 8.5 also reveal the existence of optimum (D*, B) stack configurations of SSA1 and SSA2, respectively, that minimize $E[C_{SS}]$. The envelopes of Figs. 8.4 and 8.5 represent these configurations. Most of the conclusions regarding the effect of the parameters D*, B, and L on the distortion performance of SSA0 carry over to SSA1 and SSA2 as well.

The envelopes of curves in Figs. 8.3, 8.4, and 8.5

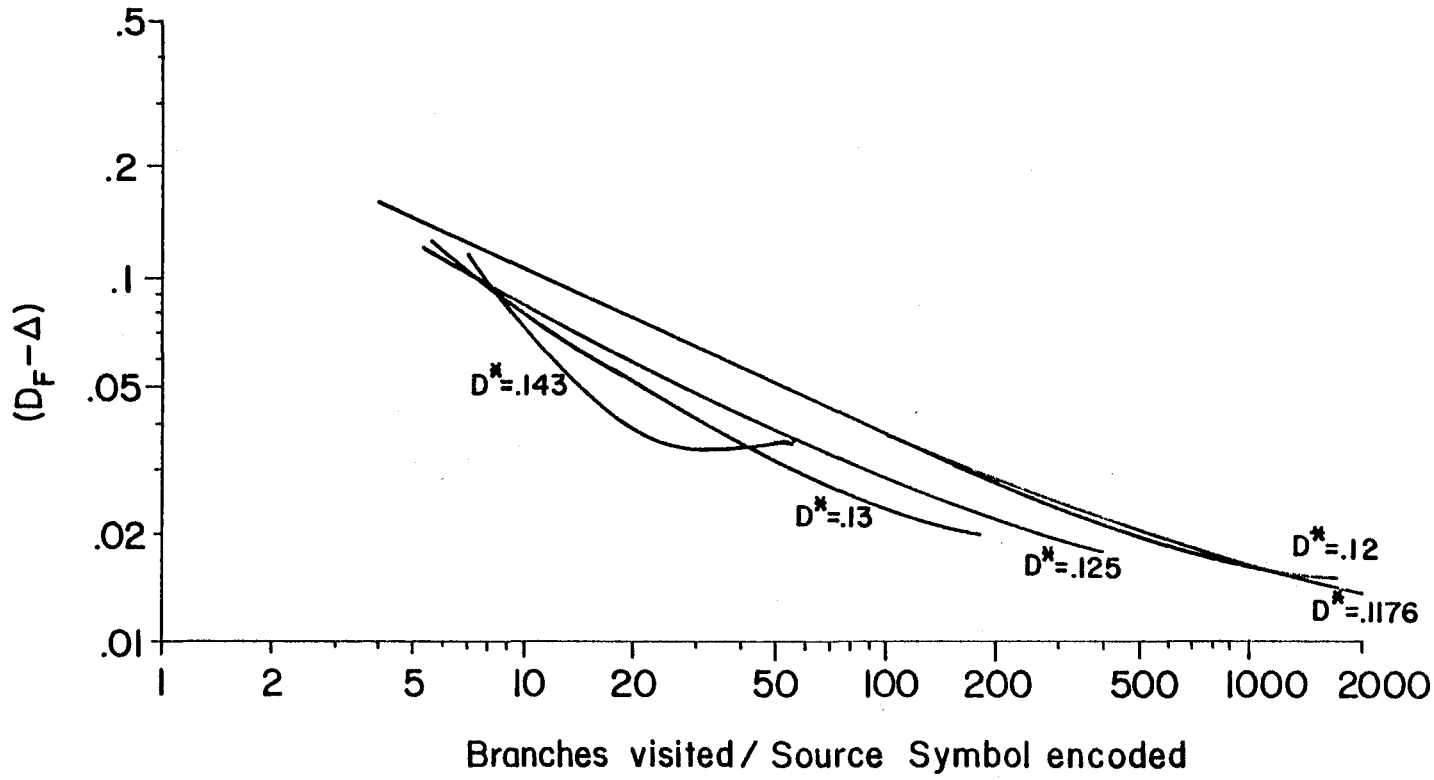


Figure 8.4 Distortion Performance Curves of SSA1 for Different D* with L = 1000

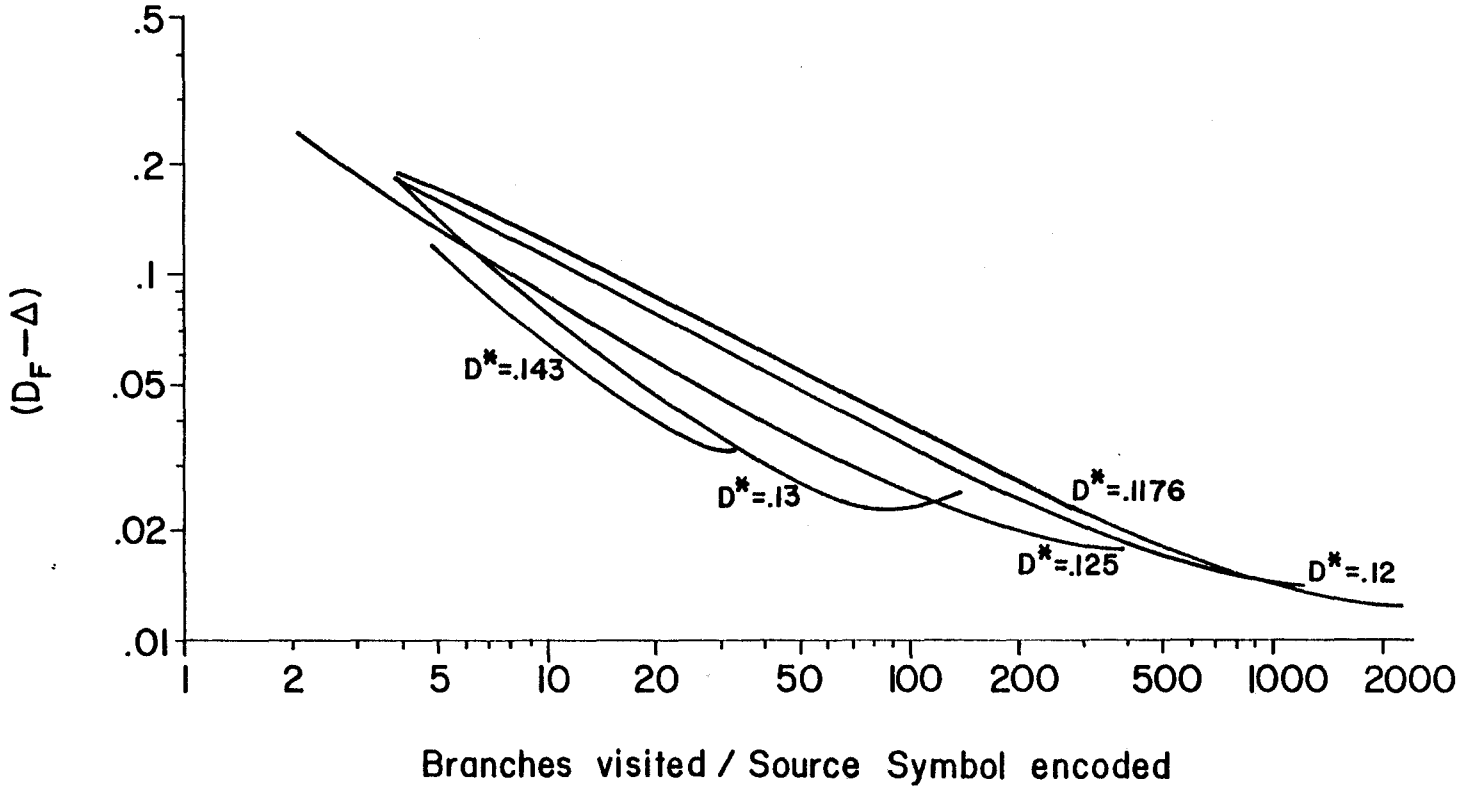


Figure 8.5 Distortion Performance Curves of SSA2 for Different D* with L = 1000

are shown in Fig. 8.6. SSA1 achieves a better distortion performance than SSA0 for values of $E[C_{SS}]$ in the range 5 to 150, while for larger $E[C_{SS}]$ there is not much difference in the distortion performances. For low $E[C_{SS}]$ values (between 5 and 30) SSA1 performs better than SSA2, while for larger $E[C_{SS}]$ values, SSA2 performs better than both SSA0 and SSA1. With $E[C_{SS}]$ about 2000 and $D^* = 0.1176$, SSA1 and SSA2 achieved final distortion performances of 13% and 10%, respectively, above Δ, compared to 15% for SSA0. This shows that some sort of adaptation of B is desirable.

8.4 Effects of Limiting Computations on the Distortion Performance of SSA1 and SSA2

Since SSA1 and SSA2 perform better than SSA0 and any algorithm must be limited dynamically, the effects of limiting the total number of computations (C_T) are investigated for SSA1 and SSA2.

Figures 8.7(a)-(c) show the effect of limiting C_T on the distortion performance of SSA1. The number of source digits encoded was allowed to vary. Thus, on any of the curves of Figs. 8.7(a)-(c), the number of source digits encoded times the number of code tree branches visited per source symbol encoded is a constant equal to C_T. B decreases along each of these curves and L is fixed at 1000.

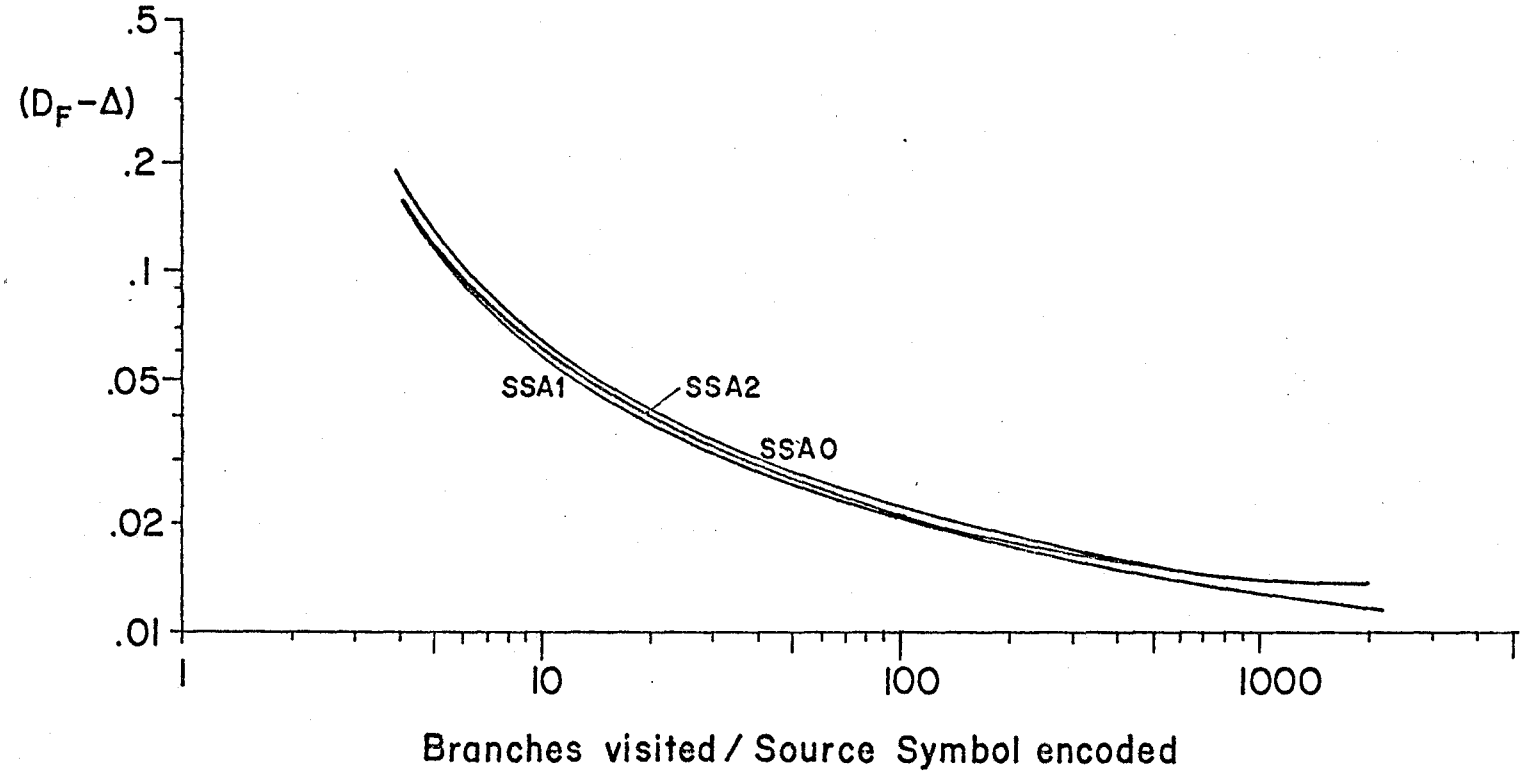


Figure 8.6 Envelopes of Performance Curves of SSA0, SSA1, and SSA2 with Length Limit; D* Decreases Along Each Curve

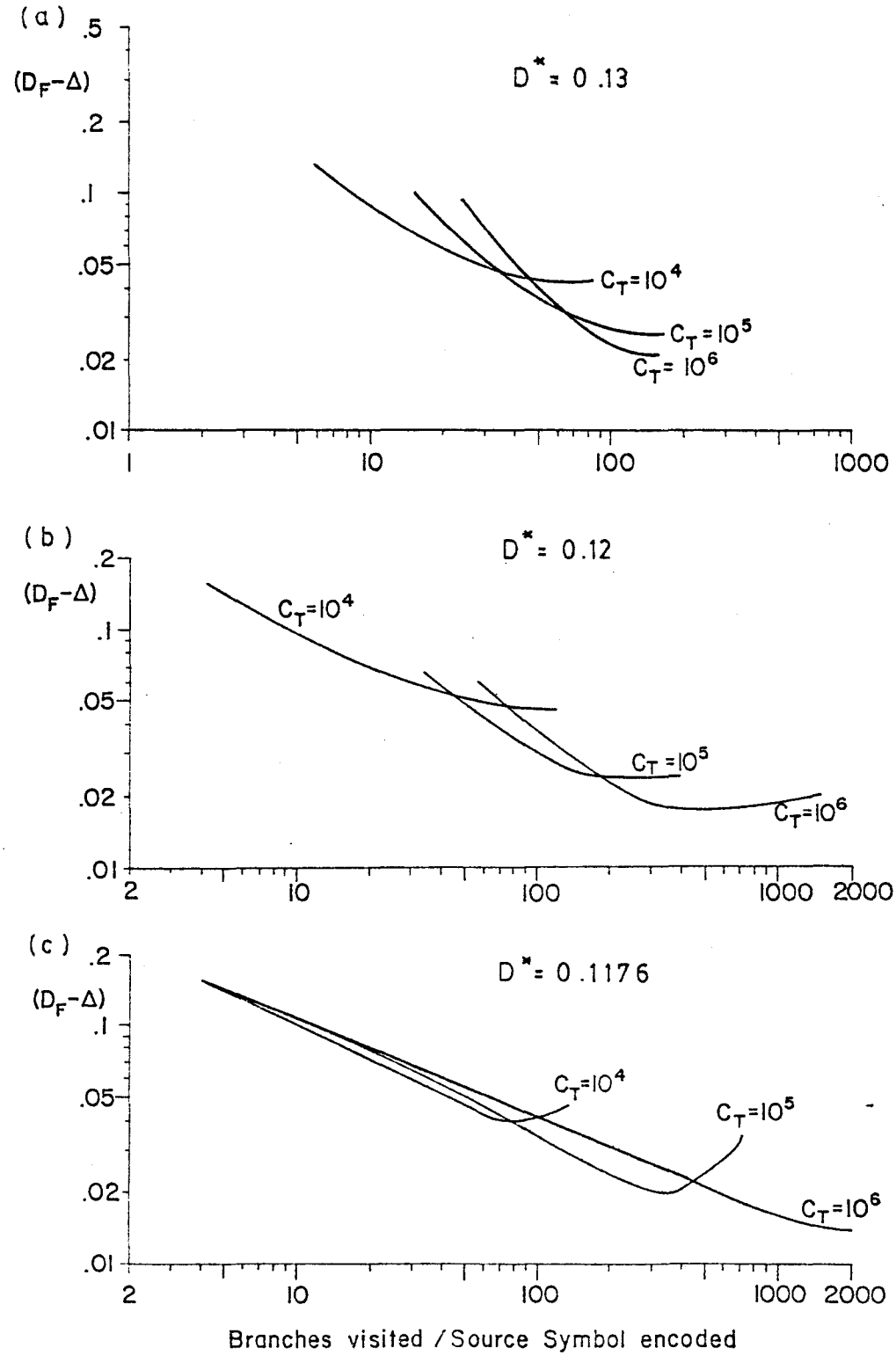


Figure 8.7 Distortion Performance Curves of SSA1 with Computational Limit for Different D* with L=1000

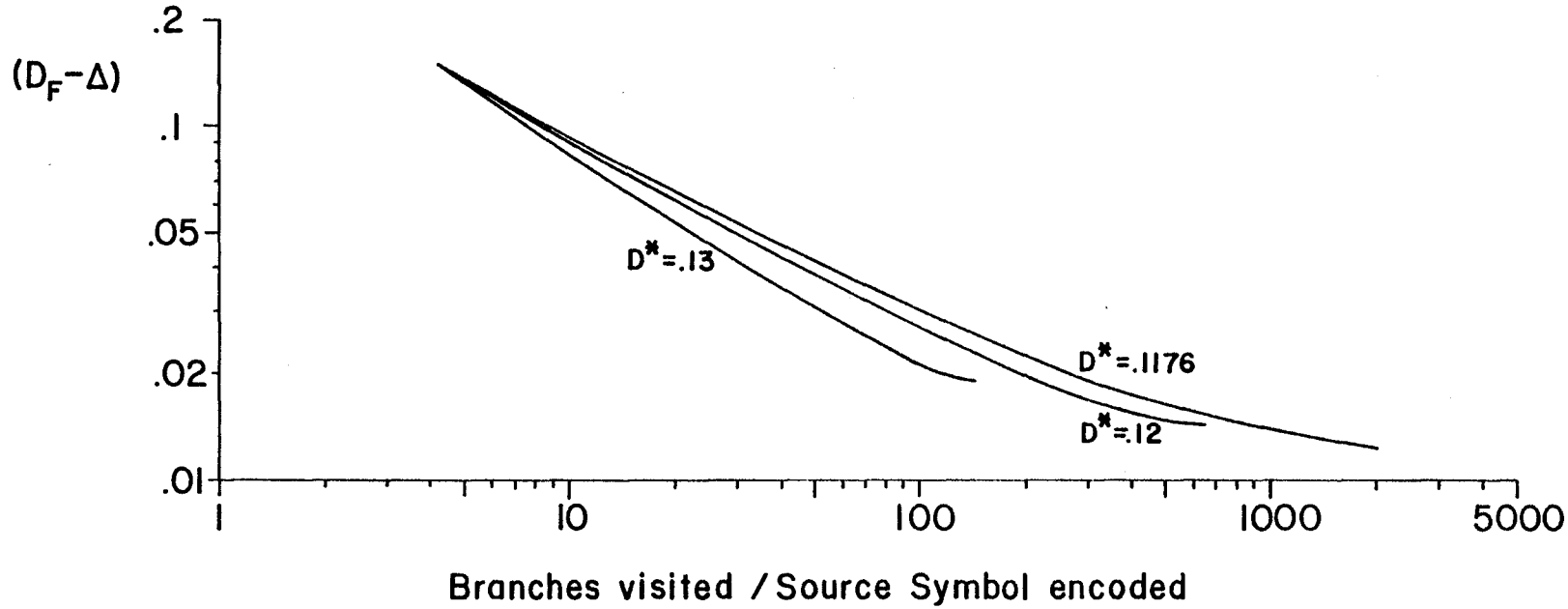


Figure 8.8 Envelopes of Distortion Performance Curves of SSA1 with Computational Limit; C_T Increases Along Each Curve; L=1000

The parameters B and D* can be seen to affect the performance of SSA1 similarly to the case without the computational limit. The envelopes of the curves of Figs. 8.7(a)-(c), representing the optimal (C_T, B) stack configurations, are shown in Fig. 8.8. The envelope of curves in Fig. 8.8, in turn, represents the optimum (D*, C_T, B) stack configurations.

The effect of computational limit C_T on the distortion performance of SSA2 are shown in Figs. 8.9(a)-(c) and their envelopes in Fig. 8.10. Unlike in the previous cases, the envelope of the set of curves in Fig. 8.9(a) is identical to the outermost curve corresponding to the largest C_T limit. This is true of Figs. 8.9(b) and (c) as well. Recall that SSA2 only raises the barrier during forward motion and does not lower it while backtracking. Assume that, with a limit on computation equal to C_T', the algorithm has found a path of length $L_1 < L$ with an average distortion equal to μ_1. Let the barrier stand at B_1. Let the computational limit be raised to $C_T'' > C_T'$. Let the algorithm SSA2 now search forward of the code tree path of length L, already in storage and find a path of legnth $L_2 \geq L_1$. Let the barrier now stand at B_2. Since the barrier cannot be lowered, barrier B_2 must now be at least as high as B_1, i.e., $B_2 \geq B_1$. Consequently, the average distortion μ_2 cannot be worse than μ_1. This implies that the

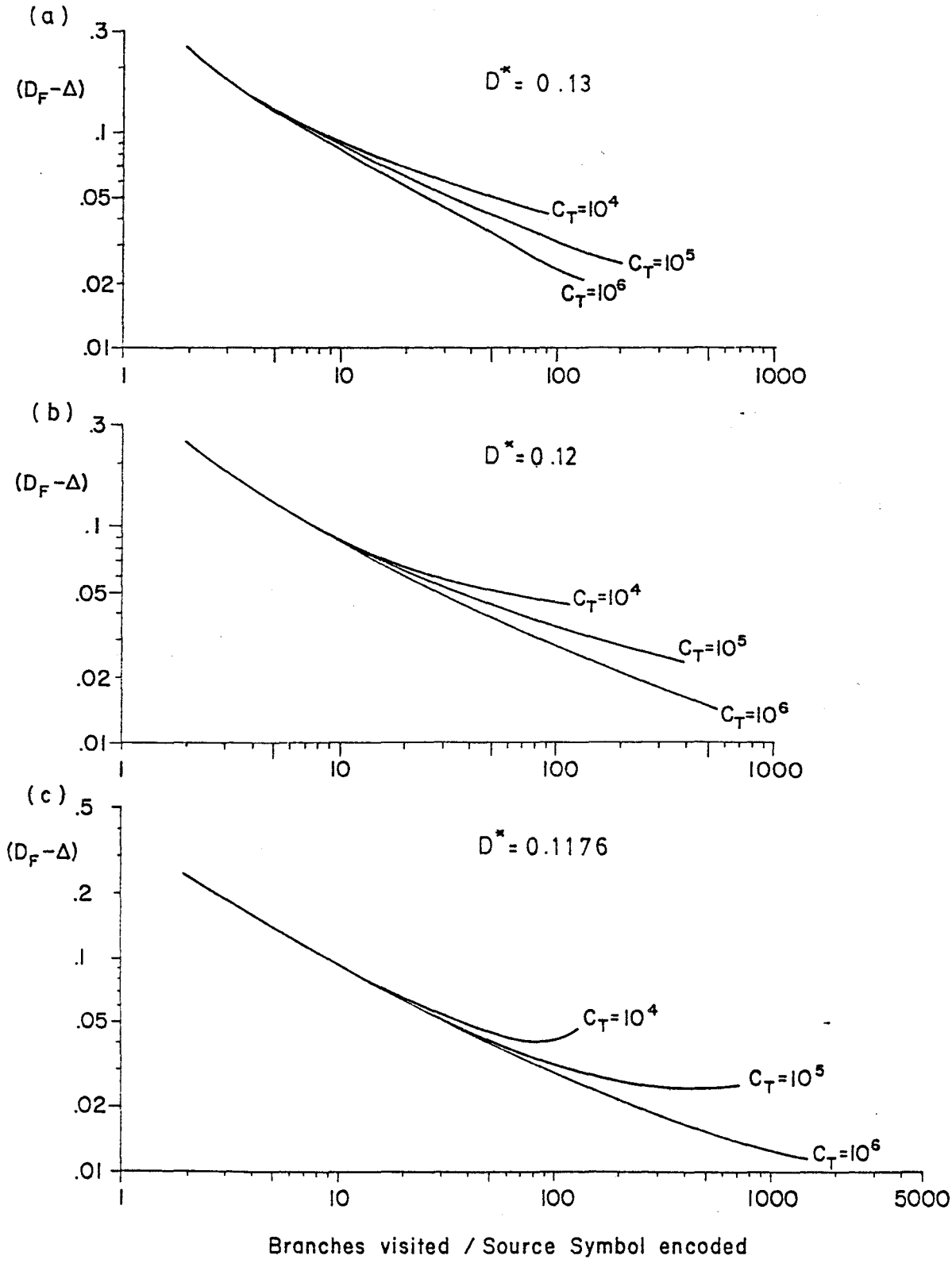


Figure 8.9 Distortion Performance Curves of SSA1 with Computational Limit for Different D* with L=1000

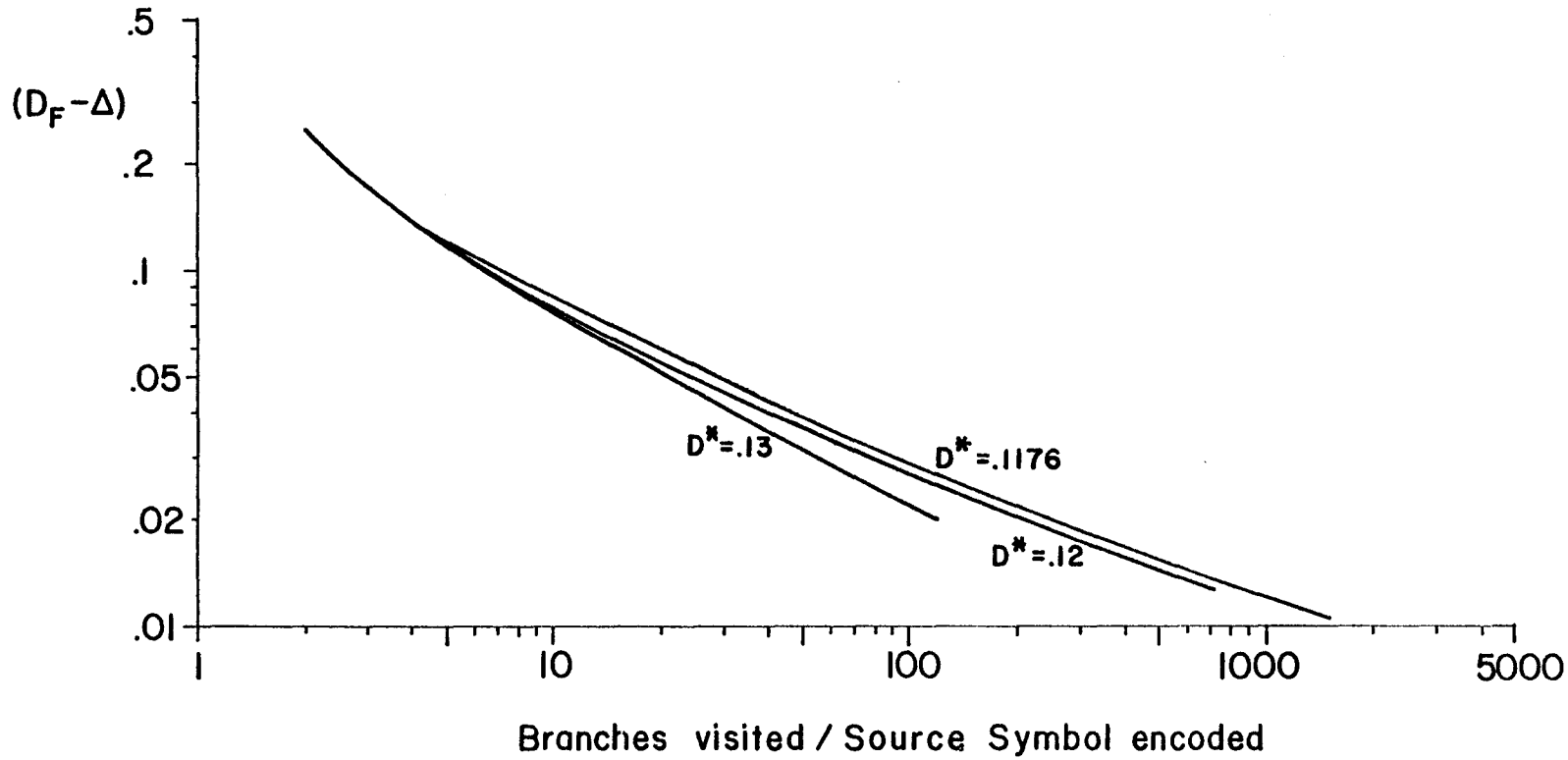


Figure 8.10 Envelopes of Distortion Performance Curves of SSA2; C_T Increases Along Each Curve; L=1000

distortion performance curve corresponding to C_T" must lie below that of C_T', just as the simulation results indicate.

The envelopes of curves in Fig. 8.8 and Fig. 8.10, giving the optimal (D*, C_T, B) stack configurations, are shown in Fig. 8.11. For a given $E[C_{SS}]$, SSA2 achieves a lower distortion per source symbol than SSA1.

8.5 Summary

We have suggested improvements to the single stack algorithm that modify the barrier dynamically during encoding. These are alternatives to the $D(\underset{\sim}{x}^{\ell})$ discard criterion suggested elsewhere [10, p. 220], [53], [50]. The dynamic discard criteria used in our simulations have not yet been mathematically analyzed. Simulation results favour the discard criterion that raises the barrier during forward motion in the code tree together with a computational limit.

The simulations exhibit optimal stack configurations that minimize the average number of code tree branches searched by the algorithm to encode each source symbol. Anderson's [17] theoretical work has shown the existence of such optimal curves for the stack algorithm. We show next that such curves exist even for "real" sources like speech.

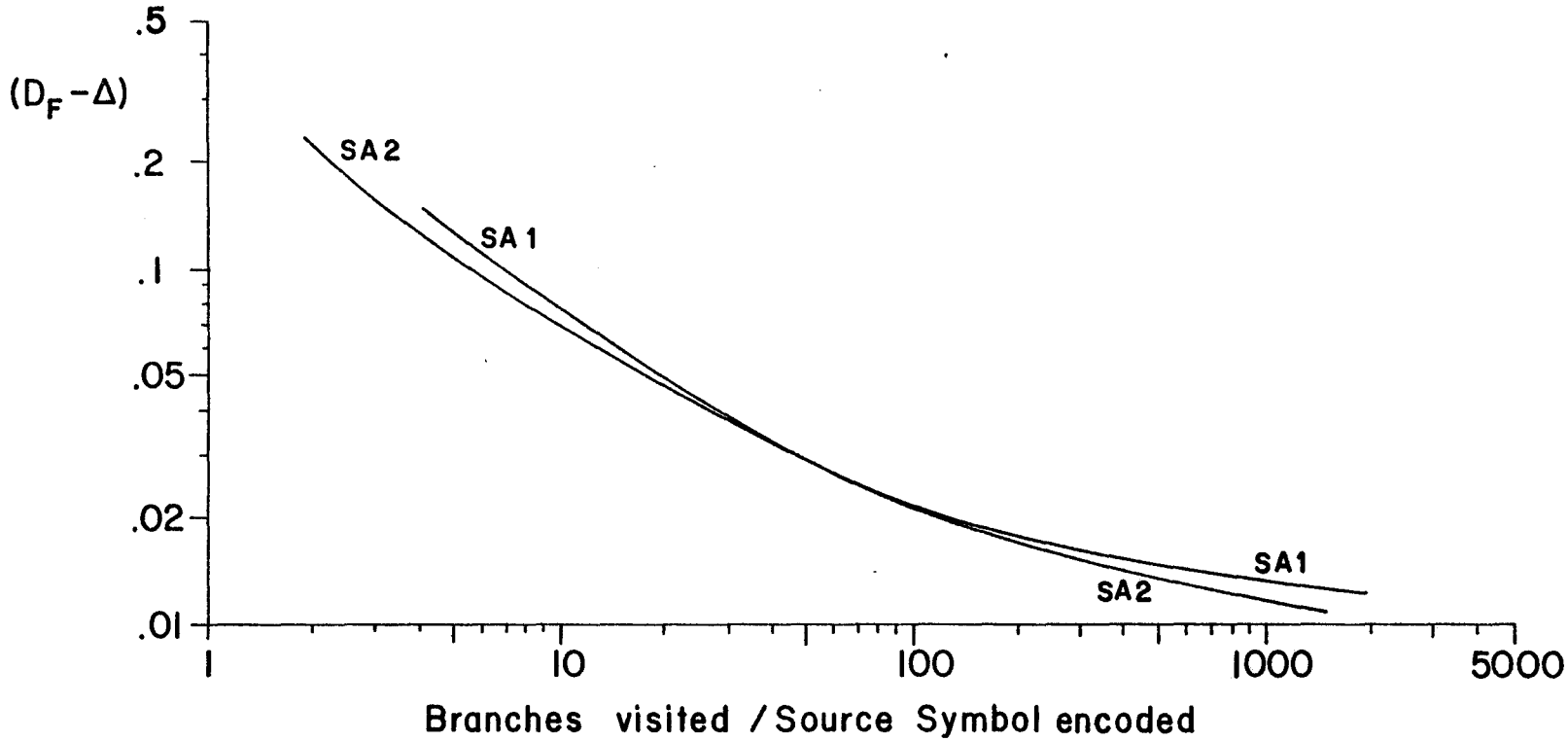


Figure 8.11 Envelopes of Distortion Performance Curves of SSA1 and SSA2 with Limit on Computation; D* Decreases Along Each Curve; L=1000

CHAPTER 9

SPEECH ENCODING BY THE STACK ALGORITHM

9.1 Introduction

We report here experiments in which the stack algorithm is used to encode a voiced speech sound. We show how the algorithm can be optimized with respect to its free parameters, the number of paths stored S and length of paths stored L, and the target distortion D* and with respect to derived quantities such as the expected node computation E[C] (the number of tree nodes visited per symbol released as output), the total storage that the algorithm uses, and the execution time T needed to carry out the processing. Each of this latter group is optimized by a different parameter combination.

Our results for speech are compared with earlier theoretical results obtained by Anderson [17] for the stack algorithm encoded binary i.i.d. source with Hamming distortion measure and similarities between the two results are pointed out.

Several authors have used the breadth-first (M,L) algorithm to encode speech [13]-[15], [33], and they have reported significant gains over single path search methods.

Anderson and Bodie [13] and Jayant and Christensen [14] have considered the effects of finite M (the number of paths stored), and finite L (the length of paths) on the performance of the (M,L) algorithm. However, in the case of metric-first algorithms, such as the stack algorithm, the effects of these two as well as a new parameter, the target distortion, have yet to be considered. Moreover, metric-first algorithms, being powerful procedures, are expected to yield significant SNR gains over other methods. Our simulation results confirm this, but it is still not clear which type of algorithm is cheaper to use for a fixed, moderate SNR. We invoke theoretical cost functions proposed in Chapter 3 in order to compare the stack algorithm performance with that of the (M,L) algorithm. Reference [76] reports the results presented in this chapter. A summary of the results appeared in [54].

9.2 Example and Instrumentation of the Stack Algorithm

Instead of using the stack algorithm (Chapter 2), we have used the merge algorithm, a metric-first algorithm that exactly mimics the stack algorithm. This enables the use of a computationally more efficient sort and merge procedure. The merge algorithm was described in detail in Chapter 4.

The speech tree code of Anderson and Law [77], used in our simulations, is given in Fig. 9.1. This code is

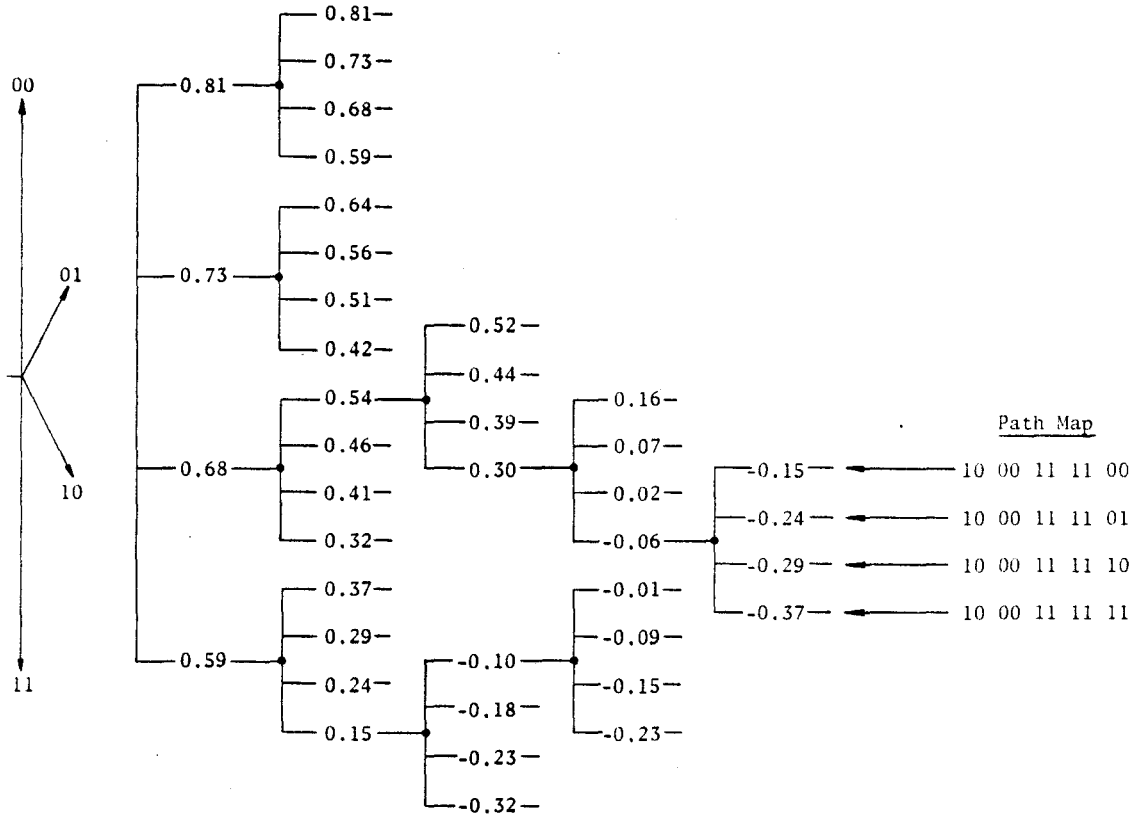


Figure 9.1 Rate 2 Speech Tree Code Generated by the Constraint 4 Real-Number Convolutional Code Generator with Coefficients C_0 = 0.7704, C_1 = 1.5154, C_2 = 1.6332, C_3 = 1.2054, and C_4 = 0.4962

defined by the relation $\hat{x}_t = \sum_i C_i\, q_{t-i}$, where the q_i are quantizer outputs and $\hat{x}_t$ are reproducer letters [13], [77]. The rate 2 tree code shown has four branches out of each node (b=4) and one symbol per branch (β=1). Here the numbers on branches correspond to normalized amplitude levels. This code is generated by the transversal filter of Fig. 9.2. The filter, of constraint length ν, in fact generates a convolutional code over real numbers, but we view it as a tree code for encoding purposes. Instead of calculating the sum $\Sigma_i\ C_i\ q_{t-i}$ every time a node is extended, a more efficient table look-up scheme is used: All the possible $2^{R(\nu+1)}$ code words are stored in a table, and when a node is to be extended, ν past path map digits plus the current one are used as an address to retrieve the code word from the table.

Figure 9.3 illustrates the working of the stack algorithm. Associated with each branch are two numbers. The first one is the branch metric. Squared error distortion criterion (i.e., $d(x,\hat{x}) = (x-\hat{x})^2$) and D* = 0.01 are used in computing the metrics. The second number corresponds to the cumulative metric of the path leading up to the end node of the branch from the root node. Nodes are numbered $N_{i,j}$, where i corresponds to the level number of the node and j correspopnds to the number of that node in level i. Node $N_{0,1}$ corresponds to the root node at level 0.

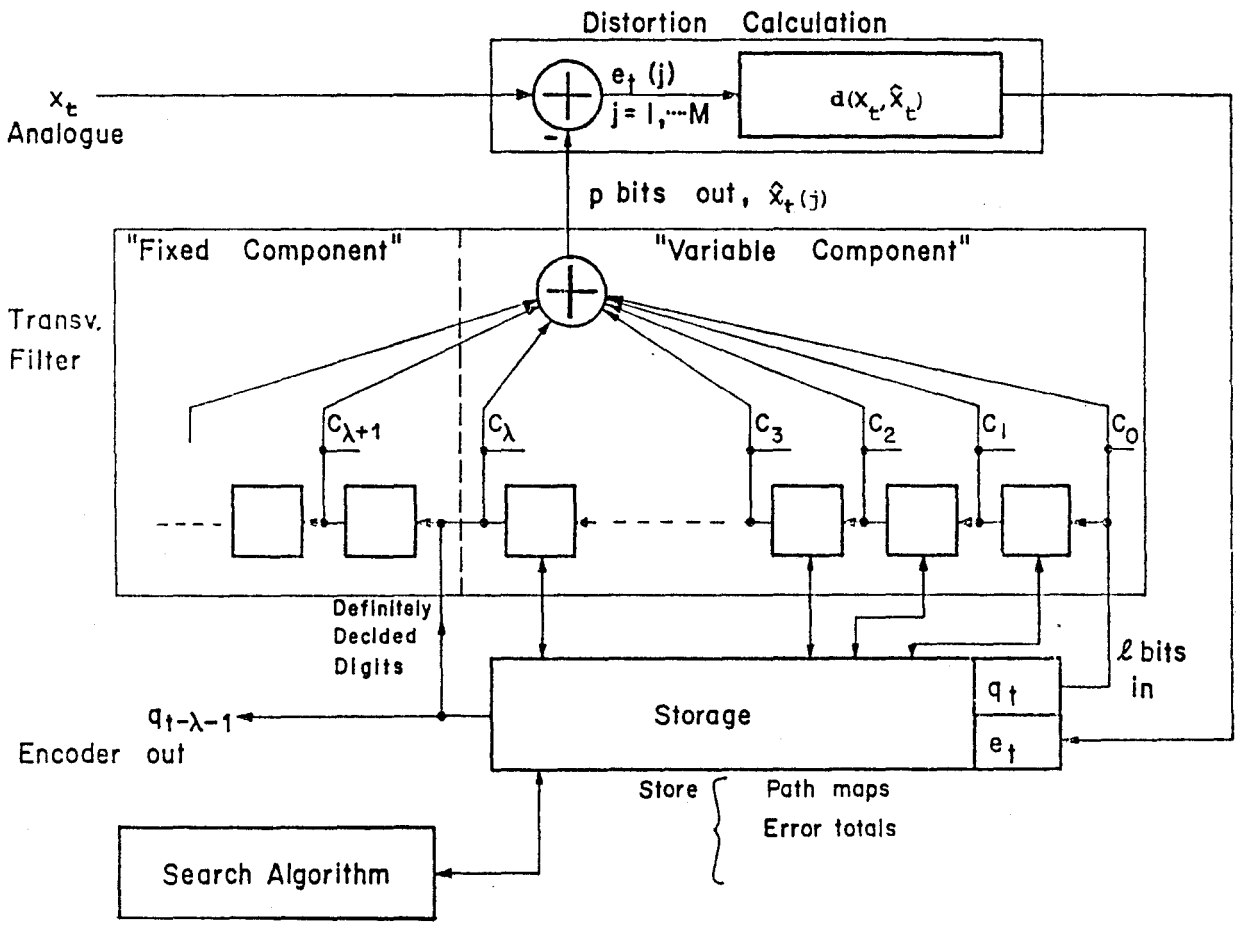


Figure 9.2 Transversal Filter Realization of the Tree Code Generator of Fig. 9.1 (From [13])

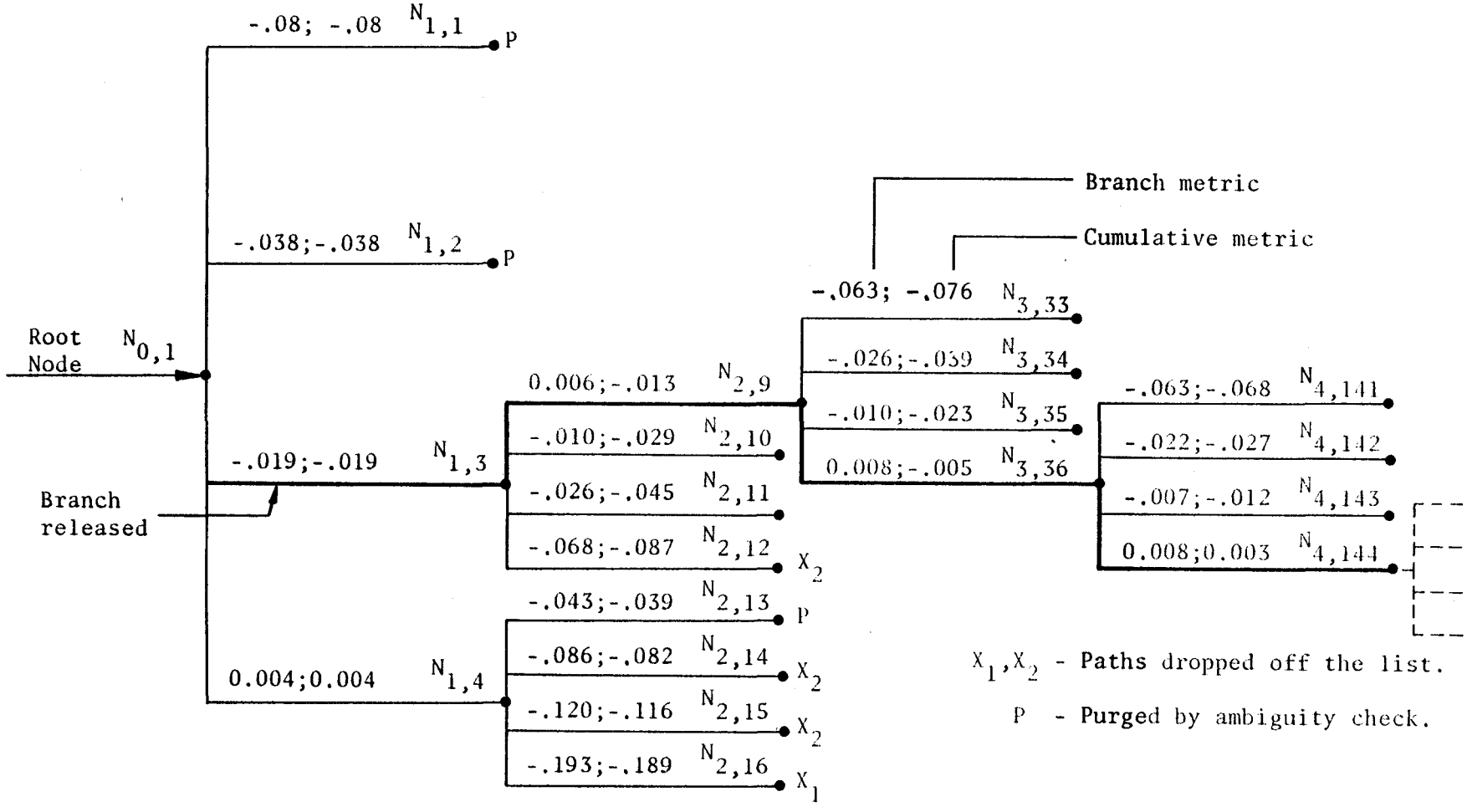


Figure 9.3 Working of the Stack Algorithm with D* = 0.01, List Size S = 12, and List Width L = 4; Source Sequence $\underset{\sim}{x}^4$ = 0.51, 0.60, 0.25, -0.11

It is assumed in this example that the list size S=12, and the list width L=4.

Node $N_{0,1}$ is extended to the four nearest nodes. As $N_{1,4}$ is the best in the list so far, it is extended next. Next $N_{1,3}$ is extended, as the cumulative metrics of all the newly extended paths from $N_{1,4}$ fall below that of $N_{1,3}$. As $N_{2,9}$ becomes the best node, it is extended next. There are now a total of 13 paths and as the list can accommodate only 12 paths, the worst one, leading to $N_{2,16}$, is deleted (indicated in Fig. 9.3 by X_1). Next $N_{3,36}$ is extended and three more paths $N_{2,12}$, $N_{2,14}$, and $N_{2,15}$ are deleted (indicated by X_2 in Fig. 9.3). As the length of the best path in the list $N_{4,144}$ equals 4, the width of the list, the encoder must now release the earliest symbol of $N_{4,144}$ corresponding to the branch $N_{0,1}$, $N_{1,3}$. After performing ambiguity check, the encoder purges paths $N_{1,1}$, $N_{1,2}$, and $N_{2,13}$. The search then proceeds forward of $N_{4,144}$.

9.3 Results of Tests - Effects of Free Parameters

The stack algorithm operated on the rate 2 speech tree code of Fig. 9.1. A speech record of the word "speed", sampled at 8 KHz (a bit rate of 16 Kbits/sec) was used for encoding purposes. List size varied from 3 to 48 path map entries. Width of the list varied from 8 bits to 48 bits corresponding to 4 to 24 source samples or tree branches.

Definition: Define signal to noise ratio (SNR) in decibels of the encoded speech as

$$\text{SNR} = 10 \log_{10} \frac{\sum_i x_i^2}{\sum_i (x_i - \hat{x}_i)^2} \quad \text{dB}$$

where x_i is the source signal amplitude,

$\hat{x}_i$, the corresponding reproducer letter, and

$x_i - \hat{x}_i$ is the error due to encoding.

Effect of Bias Factor D* on Node Computation E[C]

Referring to the definition of metric of a path, it is clear that a path with metric close to zero will have a per letter distortion close to D*. However, D* is really a free parameter, not necessarily related to the end distortion, and its main function is to control the search pattern. A large D* rewards forward motion in the code tree search, reduces node computation, and thus causes the search to be satisfied with a poorer path. Conversely, a small D* does not reward forward motion, causing intense searching among code tree branches and consequently better performance. Referring to nodes visited per branch released as output E[C] versus D* curves (Fig. 9.4), we see the above observations are indeed true. As D* decreases the search activity increases, but for large enough D*, E[C] is close to 1. The algorithm then behaves almost like a single-path

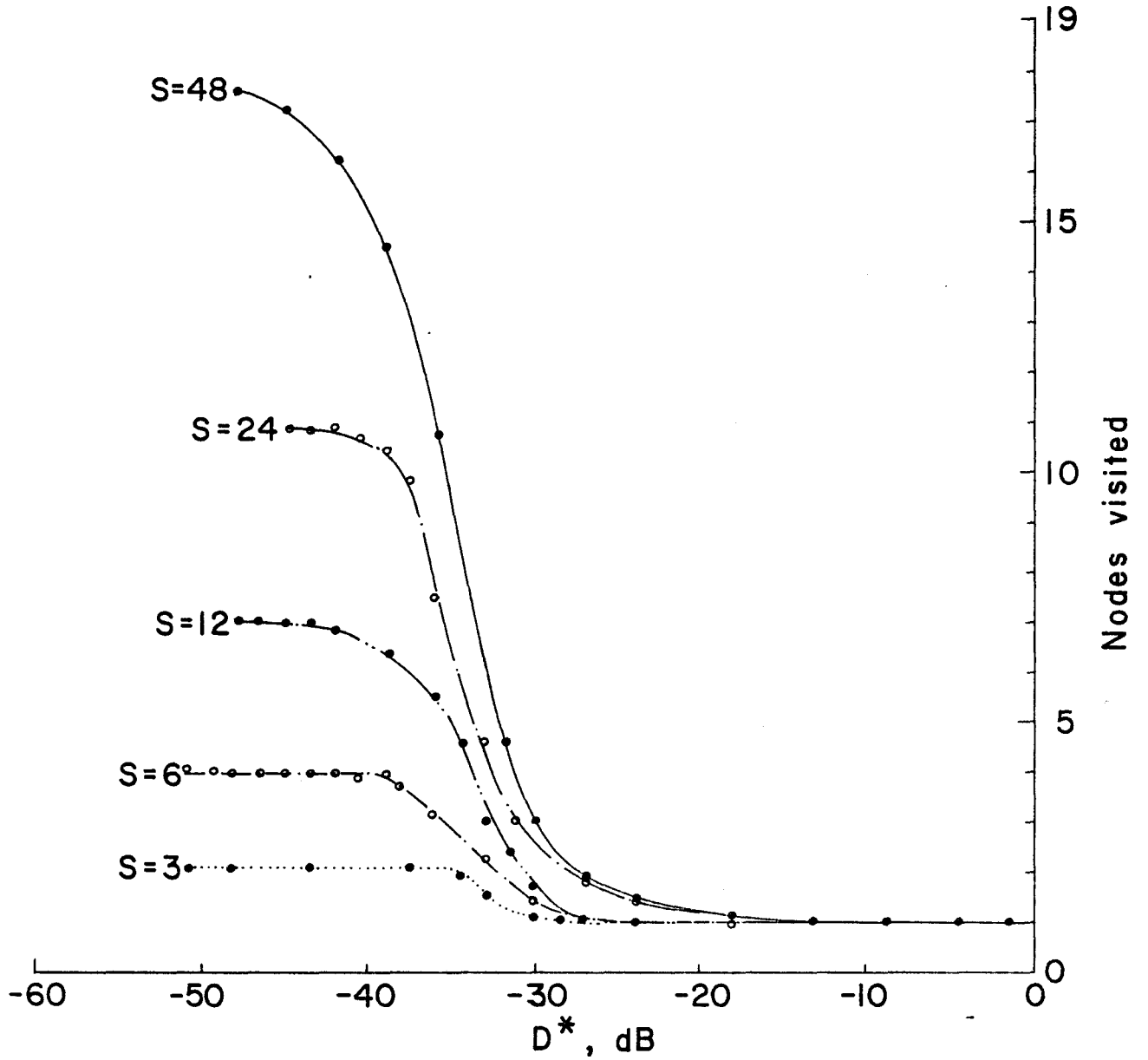


Figure 9.4 Effect of Bias Factor D* on Nodes Visited Per Branch Released as Output E[C]; 10 $\log_{10}$ D* is the value of D* in decibels

encoder.

Effect of Bias Factor D* on SNR

Figure 9.5 shows the effect of bias factor D* on SNR. SNR versus D* curves exhibit three different regions. In the first region, corresponding to large D*, search activity, as we have already noted in the preceding paragraph, remains near 1 and consequently SNR remains almost constant. In the second region, D* has decreased to the point where it begins to intensify the search, resulting in a corresponding increase in SNR. The finiteness of the list imposes a limit on the intensity of searching, and consequently decreasing D* below a critical value D_c^* does not result in any further increase in E[C]. The SNR then remains constant or tends to decrease slightly in this third "saturation" region. Consequently, this region should never be used and only biases in the second region are of interest.

The larger the list size, the smaller is the critical value D_c^* associated with that list. Hence larger lists achieve a better SNR performance. It is also clear from the curves of Fig. 9.5 that, as the list size increases, the performance curves tend to converge to a limiting curve. There is little to be gained by using very large lists, as E[C] increases enormously for little SNR gain.

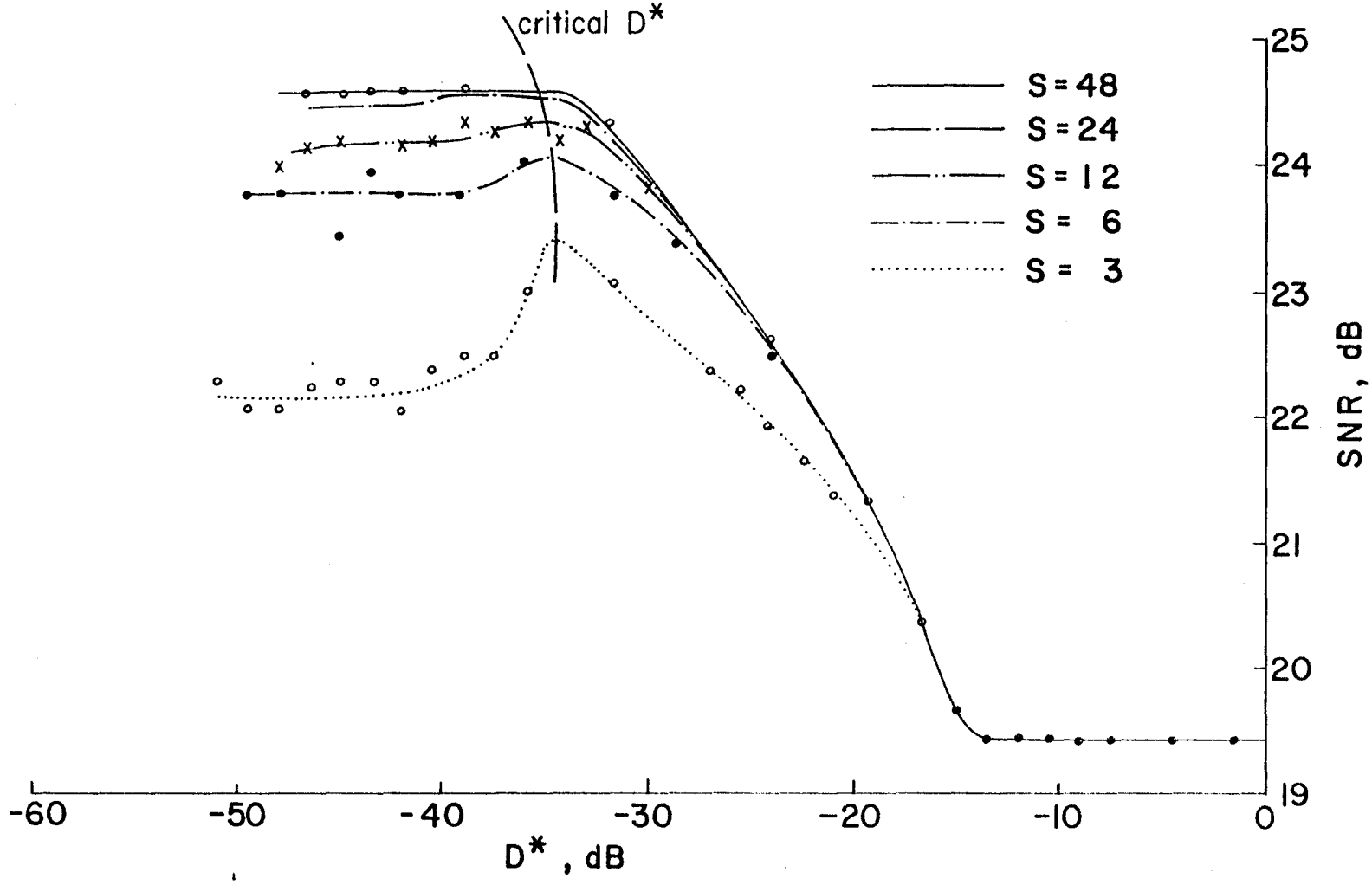


Figure 9.5 Effect of D* on SNR

Effect of Bias Factor D* on Execution Time T

The execution time must increase if E[C] increases. Thus D* has a similar effect on T as it has on E[C] (see Fig. 9.6). However, at low D* values, a larger sized list consumes more time than a smaller one. Since at low D* values the algorithm behaves like a single path encoder, it is inefficient to employ large lists; the T versus D* curves indicate this.

Effect of List Width L on SNR

Figure 9.7 shows the effect of list width L on SNR. The stacks operated near the critical bias. The SNR versus L curves exhibit two different regions. In the first region, there is a significant increase in SNR as L increases. For example, for a size 12 list, the increase in SNR is as much as 1.1 db as L increases from 8 bits to 16 bits. There is little to be gained by employing an L greater than the knee apparent in the envelope of Fig. 9.7; for the rate 2 encoder of Fig. 9.7, saturation occurs at 32 bits.

9.4 Results of Tests - Optimization of SNR

Now, we separately minimize E[C], execution time, and storage, using SNR performance as an objective function. The minimization is over the three free parameters in the

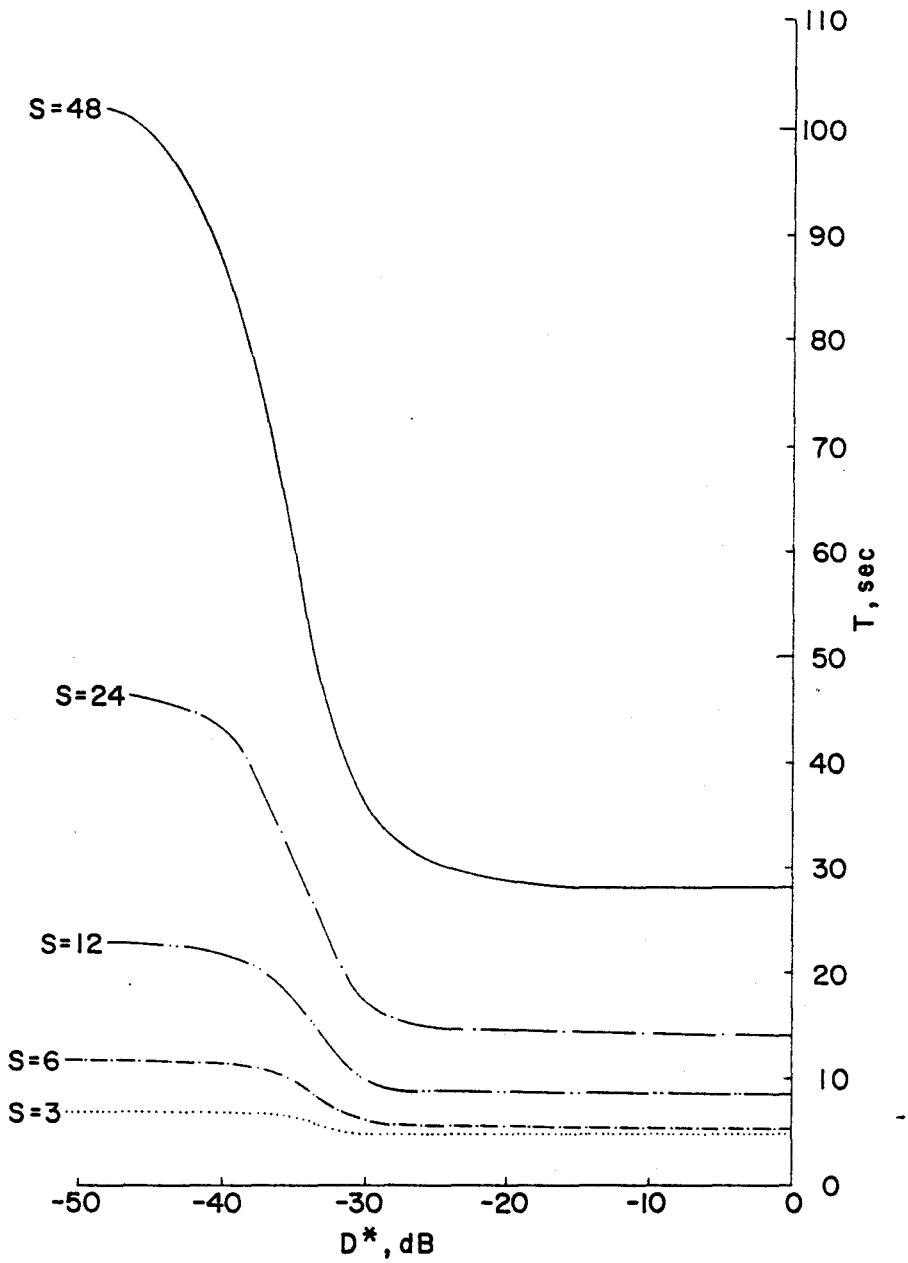


Figure 9.6 Effect of D* on Execution Time T

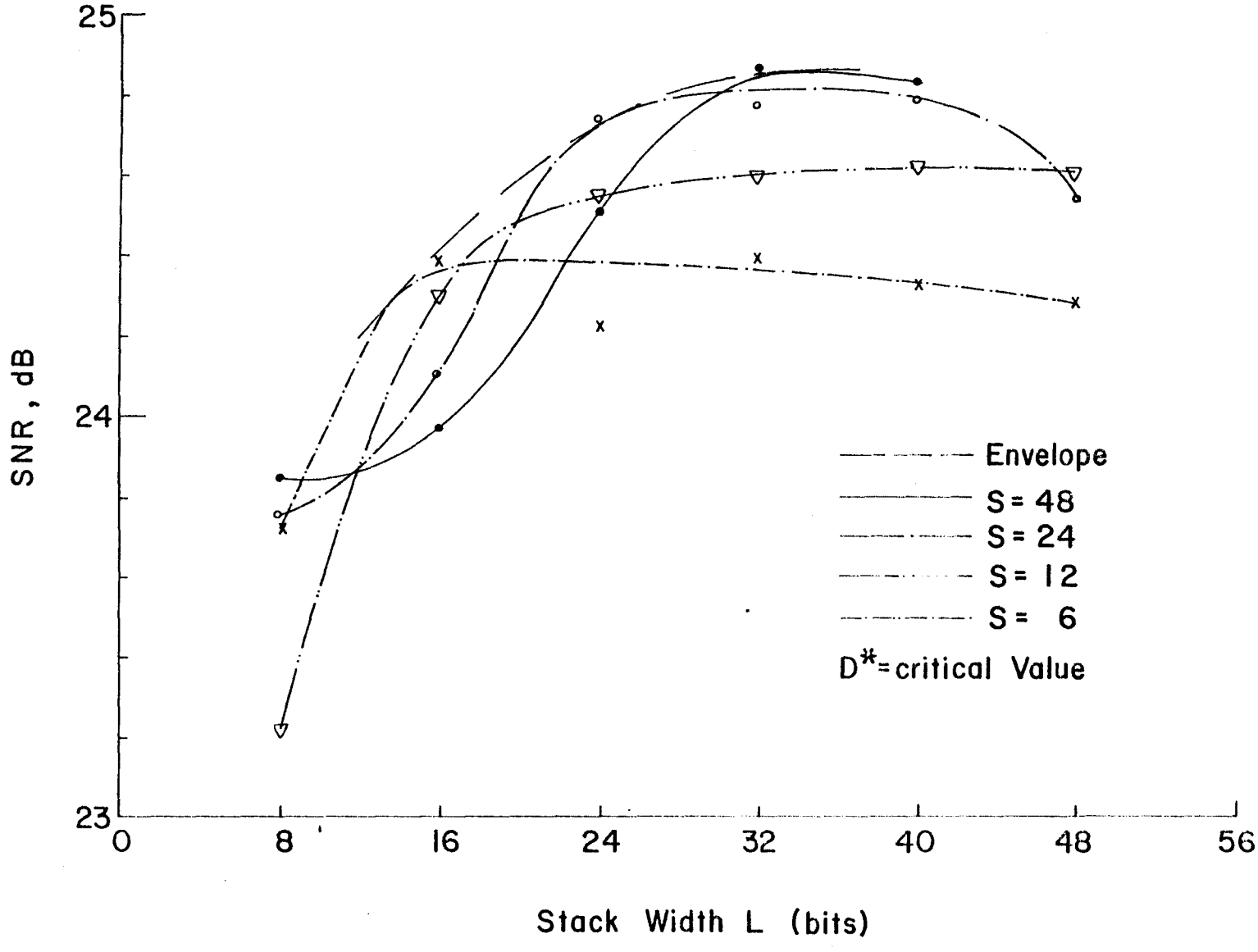


Figure 9.7 Effect of List Width L on SNR; D* Fixed at Critical Value

stack algorithm design, list size, list width, and D*.

Optimal SNR Performance Curves with respect to E[C]

Figure 9.8 plots SNR versus E[C] for different list sizes; D* decreases along each curve and we have fixed the L at 48. These curves overlap and intersect in a complicated way. It can be inferred from the curves, that a given SNR performance can be achieved by different list configurations, and, sometimes, by the same list for different bias factors. But in each of these cases E[C] is different. The best E[C] at a given SNR lies near the envelope of the curves; the best list size and D* are indicated by which curve is closest to the envelope. For example, an SNR of 23.3 dB can be achieved by list sizes of 3,6, and 24 with E[C] equal to 1.5, 1.3, and 1.85, respectively. If minimum node computation is the criterion in choosing the list size and bias factor, then for a performance of 23.3 dB a size 6 list should be used. For a given SNR performance, there exists a unique best list and bias combination that minimizes E[C]. The dashed line in Fig. 9.8 represents this.

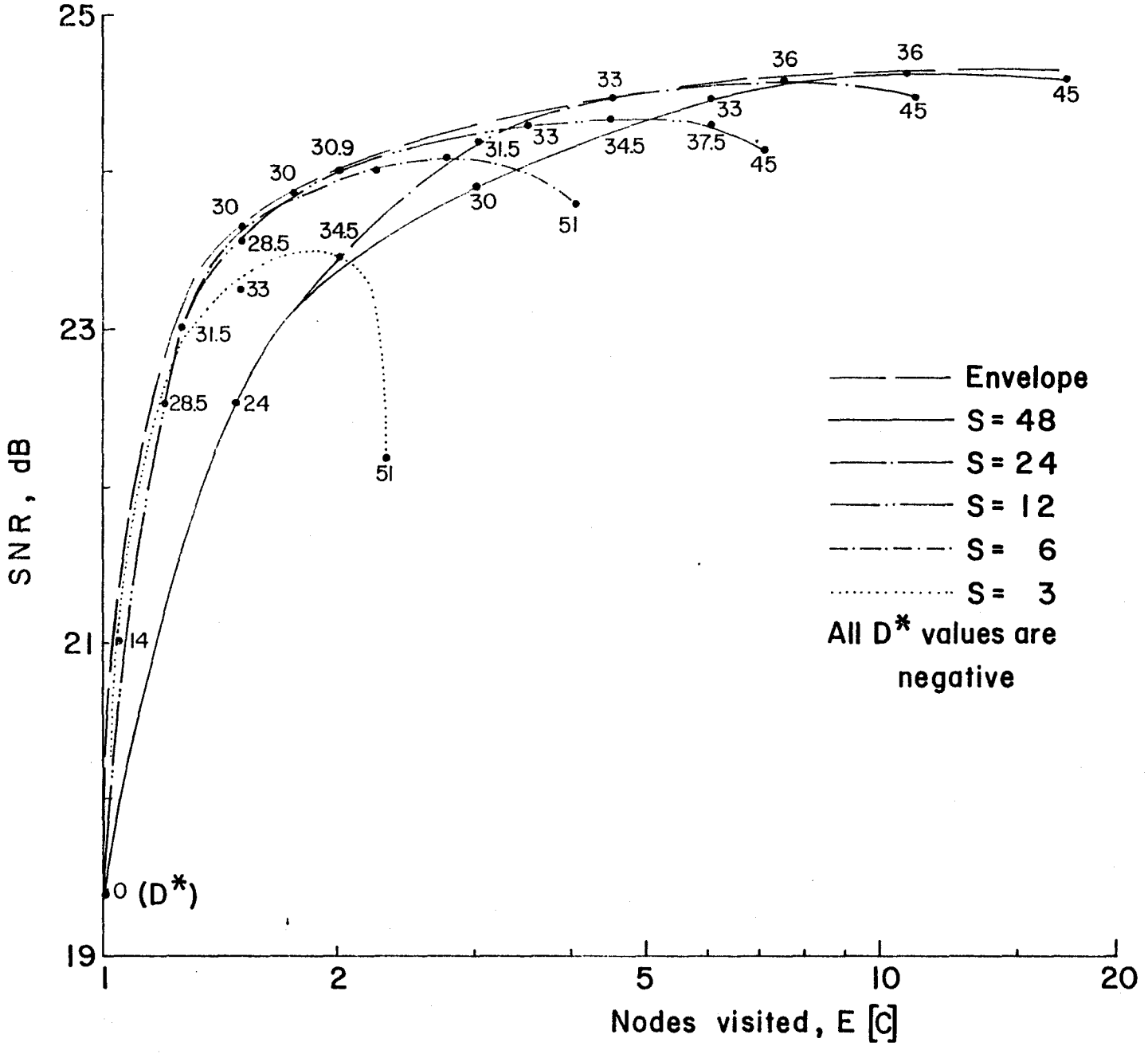


Figure 9.8 Optimal SNR Performance Curves with Respect to E[C]; D* Decreases Along Each Curve From 0 dB to -51 dB; Only Magnitudes Shown; L=48

Optimal SNR Performance with respect to Execution Time, and Storage Capacity

Figure 9.9 shows SNR performance with respect to execution time T, and Fig. 9.10 shows the same with respect to storage. Dashed lines in these figures represent the optimal E[C] and T at each SNR.

Comparing Figs. 9.8, 9.9 and 9.10, it is seen that a list configuration and D* that achieve the optimal E[C] do not do so for execution time or storage. Table 9.1 gives the list size and D* combinations minimizing E[C], T, and storage for different SNRs. It is seen that no two can be simultaneously optimized.

Anderson's [17] theoretical investigation, for the stack algorithm-encoded binary i.i.d. source with Hamming distortion measure revealed the same conflict in optimal performance curves with respect to storage and node computation. Here we have shown that such curves exist even for speech. That the stack algorithm should behave similarly with such widely different sources as the binary i.i.d. source and speech is indeed surprising. We have noticed as well that another algorithm, the single stack algorithm, exhibited similar characteristics for the binary source (see Chapter 8).

Selecting the free parameters D*, L, and S thus depends on which quantity E[C], T, or storage, one wishes to

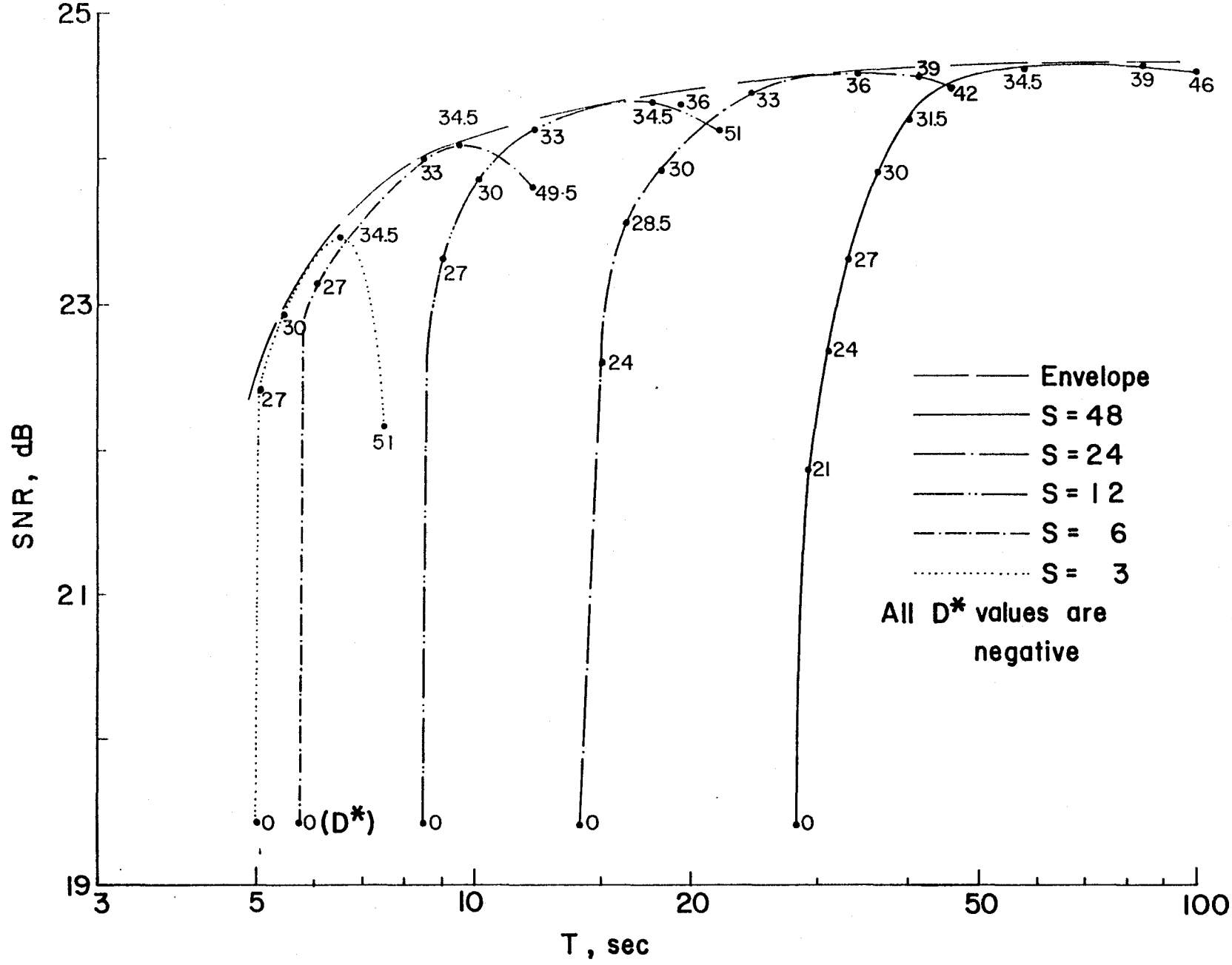


Figure 9.9 Optimal SNR Performance Curves with Respect to Execution Time T: D* Decreases Along Each Curve; L=48

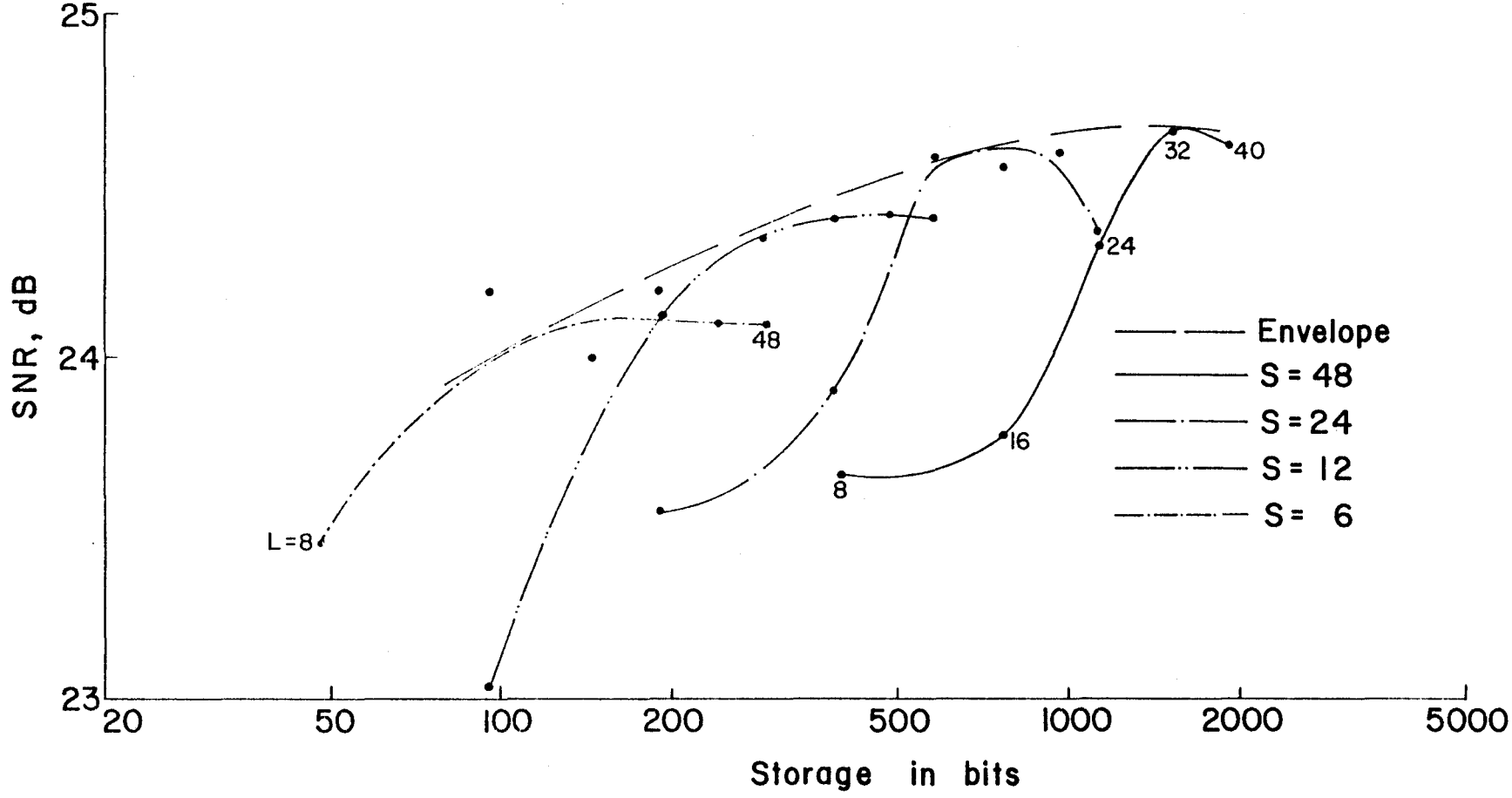


Figure 9.10 Optimal SNR Performance Curves with Respect to Storage Capacity; L Increases Along Each Curve from 8 bits to 48 bits; D* Fixed at Critical Value

Table 9.1: Optimum List Configurations with respect to E[C], T, and Storage. L fixed at 48 for Node Computation and Execution Time Minimizations. D* fixed at Critical Value for Storage Minimization.

SNR dB	D* and list size minimizing E[C]		D* and list size minimizing T		List width and size minimizing storage	
	D* dB	S	D* dB	S	L	S
23.4	-28.5	6	-34.5	3	8	6
24.0	-30.9	12	-33.0	6	16	6
24.4	-33.0	24	-36.0	12	32	12

optimize. There is some debate over which quantity to choose. Traditionally, it has been the expected node computation that was used to measure sequential coding efficiency. But the discussions in the preceding paragraphs point to its inadequacy. We have proposed several cost functions suited to hardware and software implementations of sequential coding algorithms. From algorithmic considerations, the asymptotic time costs for the stack and M-algorithms are as given below. (The time cost of an algorithm is the number of basic operations it performs.)

$$\text{Stack algorithm time cost} = C_1 SE[C],$$

$$\text{M-algorithm time cost} = C_2 M \log M,$$

where C_1 and C_2 are proportionality constants $\simeq 1$.

Using these time cost formulas, we can compare the stack and M-algorithms. Considering first the stack algorithm, the time-optimizing combination D* = -30 dB and S = 3 achieved an SNR performance of about 23 db. The time cost for this combination is 4.5 C_1. Another combination D* = -26 dB and S = 6 yielded the same performance with a time cost of 7.8 C_1. Experimentally measured T curves in Fig. 9.9 show that the second combination indeed consumed more time. The second however has a lower node computation, 1.3 instead of 1.5. The M-algorithm with M = 4 achieves 23 db with cost equal to 8 C_2. Thus with the right list size and bias combination, the stack algorithm is only one-half to

two-thirds as costly as the M-algorithm. Whether or not the cost formulas are useful except asymptotically is open to question. However we have the following results concerning program running times. While the M-algorithm with M=8 required 16.4 seconds of central processor time, the stack algorithm required only 12 seconds to achieve the same performance.

9.5 Conclusions

We have shown, using a software encoder for speech, how the algorithm's performance can be optimized with respect to the expected node computation, execution time, and storage. Since no two of these can be simultaneously minimized, execution time in a software environment was chosen for minimization. The stack algorithm is found to be less time consuming than the M-algorithm for speech. Several conflicting optimal curves, previously shown to exist for theoretical sources, are shown here to exist for speech as well.

Successful application of the Viterbi algorithm to speech encoding depends on whether short enough convolutional codes are available for speech. Good speech codes have been shown to have between 256 to 1,024 code states [77]. While experimental evidence shows that only 4 to 8 code states need to be searched [14], [77], the Viterbi

algorithm must search all the code states. Thus, the Viterbi algorithm may prove to be quite expensive. Metric-first algorithms lack the synchronism of the M-, or the Viterbi algorithm, but as a compensation seem to be much cheaper. In applications where synchronism is not a crucial factor, such as in stored voice answer-back, metric-first algorithms provide an attractive alternative.

CHAPTER 10

CONCLUSIONS

This thesis is motivated by the increasing applications of code search algorithms and the need to devise efficient methods. We started in Chapter 2 by classifying the algorithms into three main classes, breadth-first, metric-first, and depth-first techniques. The metric-first algorithms, like the stack algorithm, have the least node computation of all the schemes, but they require large storage space. On the other hand, the depth-first procedures, like the single stack and the Fano algorithms, require the least amount of storage, but they search a large number of code tree branches. Breadth-first algorithms, like the M-algorithm, follow a middle course between these two extremes and require moderate storage and computation.

In order to compare these different schemes, we proposed in Chapter 3 several cost functions based on the size of and number of accesses to storage and the number of comparisons done by the algorithms. Resource costs of many existing algorithms were derived.

Since the metric-first algorithms are efficient in terms of node computation, we inquired if it was possible to

design efficient algorithms having reduced cost compared to the stack algorithm. The merge and the generalized merge algorithms of Chapter 4, the AVL tree-based algorithm of Chapter 5, and the dynamic bucket algorithm of Chapter 6 provided the answer. While the generalized merge and AVL-based algorithms reduced the search times to O(log S) from O(S) for the stack algorithm, the "roughly" metric-first dynamic bucket algorithm provided a constant search time. These are as optimum as they can possibly be. In the process of devising these algorithms, we also discovered efficient data structures. The merge and the AVL-based schemes used efficient in-core (internal storage) data structures, while the dynamic bucket algorithm would be efficient with large external storages in which the buckets could be stored and retrieved in blocks. Consequently, the merge, AVL-based, and the dynamic bucket algorithms are likely to be optimal for small, medium, and large storage sizes, respectively. Thus the dynamic bucket algorithm is not a universal panacea even though it has the least order of dependence on S of any "roughly" metric-first schemes.

One of the factors that affects the resource costs of an algorithm is the expected node computation E[C], and we have dealt at some length with this. The asymptotic behaviour of E[C] is known for different algorithms used with symmetric sources. However, there exists little

knowledge of its behaviour with regard to asymmetric sources. Chapter 7 analyzed asymmetric source encoding by the single stack algorithm and derived an equation for node computation that is the stochastic analog of the equation for the symmetric case. The branching process method used here opens up further avenues for research, for example, in encoding sources with memory.

To choose the best algorithm for a given situation, one must determine the combination of L, S, and E[C] that optimizes some objective function, perhaps the cost function for the desired encoding distortion or an error probability. This is a most difficult task, for in addition to constructing and testing the algorithms, one must optimize over many lesser parameters such as the discard criterion, threshold increments, list width, and the like. However, in Chapters 8 and 9, we presented simulation results for the single stack algorithm-encoded binary i.i.d. source and stack algorithm-encoded speech, respectively, and used them to optimize the performance of the algorithms over a number of parameters. The configurations that optimize storage, time, or E[C] were all different and no two of these could be simultaneously minimized. Similar studies should be made for other code searching schemes.

From studies that have been made here or elsewhere, some conclusions may be drawn. First consider sequential

source encoding of binary i.i.d. sources with Hamming distortion measure. Table 10.1 (from [56]) reproduces simulations for the stack, M-, 2-Cycle, and single stack algorithms used with random tree codes. The encoder rate is 1/2 output bit/source bit, and all the algorithms achieve an average distortion per bit of about 0.125, 15% above the distortion-rate function. The cost formulas from Chapter 3 are used to evaluate cost, and because of the asymptotic nature of these and uncertainties with simulations only orders of magnitude are significant in the results. Still, it is clear the two non-sorting algorithms have greatly reduced cost compared to the stack or M-algorithms under either the space-time or space-plus-time cost evaluation. Even the merge and the bucket algorithm (borrowing the same L, S, and E[C] as the stack algorithm) fall well short, and in fact perform only as well as the M-algorithm.

However, speech encoding using the M- and the stack algorithms points to the superiority of metric-first procedures. Our simulation results in Chapter 9 have shown that the stack algorithm is only one-half to two-thirds as costly as the M-algorithm. Typical time cost per branch for the stack algorithm (to achieve SNR = 23 dB using S = 3 and $E[C_{SA}] = 1.5$) turns out to be 4.5 K, where K is a constant ≈ 1. For the single stack algorithm, however, the time cost $\approx E[C_{SS}]$; the single stack algorithm is yet to be used to

Table 10.1: Evaluation of Cost for Certain Algorithms, taken from Experimental Data. Binary i.i.d. Source with Hamming Distortion, R = 1/2, Encoded Distortion 0.125 (Shannon Limit = 0.110).

	Branches Viewed E[C]	Paths Stored S	Space.Time Cost	Space+Time Cost
Stack	200	>500	10^{10}**	200K
M-	500	250	7×10^{7}	51K
2-Cycle	1000	1*	200K	8500
Single Stack	1500	1	300K	1700

L = 200-300, all cases (200 used for cost).

* About 150 paths of average length 50 were kept in the save stack.

** 2×10^{8} with Merge Alg.; 7×10^{7}+H with Bucket Alg.

encode speech, but judging from its behaviour with other sources, it is expected to have an $E[C_{SS}]$ much larger than that of the stack algorithm. Thus the stack algorithm may be superior for "real" sources like speech. For longer list sizes and higher SNR performances, the more efficient merge algorithm would replace the stack algorithm.

The stack algorithm's space-time and space-plus-time costs are 54 and 16.5, respectively, under similar conditions as in the preceding paragraph. The corresponding costs for the M-algorithm (using M = 4 and L = 4) are 128 and 24, respectively. Thus the stack algorithm outperforms the M-algorithm by a factor of 1.5 to 2 under any cost measure. For the single stack algorithm these costs are $LE[C_{SS}]$ and $L+E[C_{SS}]$. Assuming a moderate L of 50 symbols we see that the stack algorithm outperforms the single stack algorithm as well. In view of the superiority of the metric-first schemes over others, the new metric-first algorithms proposed in this thesis take on added significance.

In many sequential decoding algorithms, erasure of output data by noise-induced computational overload can occur, and thwarting this problem is a factor in their design. The different cost measures make no explicit mention of erasures, but to the extent that erasures stem from cost overflow, that is, from exhaustion of resources,

the estimates of cost presented here indicate susceptibility to erasures.

Research into the applications of search procedures to the intersymbol interference problem has only just begun. There should be interesting results here, since the Viterbi algorithm estimator would seem to have limited applicability to severely band limited channels.

The practice and theory of both source coding and channel coding are assuming greater significance. Transmission of digital voice signals and teleconferencing using compressed pictures are becoming increasingly common. Great improvements in information transmission can result by the use of efficient code search procedures for data compression and for channel transmission. We hope that this thesis has helped the understanding of this many faceted problem of code searching, by presenting the right blend of art, engineering, and theory.

REFERENCES

[1] C.E. Shannon, "A mathematical theory of communication", Bell Syst. Tech. J., Vol. 27, July 1948, pp. 379-423.

[2] W.W. Peterson and E.J. Weldon, Error-Correcting Codes, 2nd ed., The MIT Press, Cambridge, MA, 1972.

[3] J.M. Wozencraft and I.M. Jacobs, Principles of Communication Engineering, John Wiley, New York, 1965.

[4] F. Jelinek, Probabilistic Information Theory, McGraw-Hill, New York, 1968.

[5] R.G. Gallager, Information Theory and Reliable Communication, John Wiley, New York, 1968.

[6] A.J. Viterbi and J.K. Omura, Principles of Digital Communication and Coding, McGraw-Hill, New York, 1979.

[7] D.A. Huffman, "A method for the construction of minimum redundancy codes", Proc. IRE, Vol. 40, No. 10, Sept. 1952, p. 1098.

[8] F. Jelinek, "Buffer overflow in variable length coding of fixed rate sources", IEEE Trans. Inform. Theory, Vol. IT-14, No. 3, May 1968, pp. 490-501.

[9] C.E. Shannon, "Coding theorems for a discrete source with a fidelity criterion", IRE Nat. Conv. Rec., part 4, 1959, pp. 142-163.

[10] T. Berger, Rate Distortion Theory, Prentice-Hall, Inc., Englewood Cliffs, N.J., 1971.

[11] F. Jelinek, "Tree encoding of memoryless time-discrete sources with a fidelity criterion", IEEE Trans. Inform. Theory, Vol. IT-15, Sept. 1969, pp. 584-590.

[12] F. Jelinek and J.B. Anderson, "Instrumentable tree encoding of information sources", IEEE Trans. Inform. Theory, Vol. IT-17, June 1971, pp. 118-119.

[13] J.B. Anderson and J.B. Bodie, "Tree encoding of speech", IEEE Trans. Inform. Theory, Vol. IT-21, July 1975, pp. 379-387.

[14] N.S. Jayant and S.A. Christensen, "Tree encoding of speech using the (M,L)-algorithm and adaptive quantization", IEEE Trans. Communications, Vol. COM-26, Sept. 1978, pp. 1376-1379.

[15] S.G. Wilson and S. Hussain, "Adaptive tree encoding of speech at 8000 bits/s with a frequency-weighted fidelity criterion", IEEE Trans. Communications, Vol. COM-27, Jan. 1979, pp. 165-170.

[16] J.B. Anderson and F. Jelinek, "A 2-cycle algorithm for source coding with a fidelity criterion", IEEE Trans. Inform. Theory, Vol. IT-19, Jan. 1973, pp. 77-92.

[17] J.B. Anderson, "A stack algorithm for source coding with a fidelity criterion", IEEE Trans. Inform. Theory, Vol. IT-20, March 1974, pp. 211-226.

[18] R.G. Gallager, "Tree encoding for symmetric sources with a distortion measure", IEEE Trans. Inform. Theory, Vol. IT-20, Jan. 1974, pp. 65-76.

[19] A.J. Viterbi, "Error bounds for convolutional codes and an asymptotically optimal decoding algorithm", IEEE Trans. Inform. Theory, Vol. IT-13, April 1967, pp. 260-269.

[20] G.D. Forney, "The Viterbi algorithm", Proc. IEEE, Vol. 61, No. 3, March 1973, pp. 268-278.

[21] A.J. Viterbi and J.K. Omura, "Trellis encoding of memoryless discrete-time sources with a fidelity criterion", IEEE Trans. Inform. Theory, Vol. IT-20, 1974, pp. 325-331.

[22] J.M. Wozencraft, "Sequential decoding for reliable communication", Nat. IRE Conv. Rec., Vol. 5, part 2, 1957, pp. 11-25.

[23] J.M. Wozencraft and B. Rieffen, Sequential Decoding, The MIT Press, Cambridge, MA, 1961.

[24] R.M. Fano, "A heuristic discussion of probabilistic decoding", IEEE Trans. Inform. Theory, Vol. IT-9, April 1963, pp. 64-74.

[25] J.L. Massey, "Variable-length codes and the Fano metric", IEEE Trans. Inform. Theory, Vol. IT-18, No. 1, Jan. 1972, pp. 196-198.

[26] K.Sh. Zigangirov, "Some sequential decoding procedures", Probl. Peredach. Inform., Vol. 2, No. 4, 1966, pp. 13-25.

[27] F. Jelinek, "A fast sequential decoding algorithm using a stack", IBM J. Res. Develop., Vol. 13, Nov. 1969, pp. 675-685.

[28] D. Haccoun and M.J. Ferguson, "Generalized stack algorithms for decoding convolutional codes", IEEE Trans. Inform. Theory, Vol. IT-21, Nov. 1975, pp. 638-651.

[29] P.R. Chevillat and D.J. Costello, "A multiple stack algorithm for erasure free decoding of convolutional codes", IEEE Trans. Communications, Vol. COM-25, Dec. 1977, pp. 1460-1470.

[30] J.B. Anderson, "Asymptotic computation for certain sequential algorithms for source coding with a fidelity criterion", IEEE Trans. Inform. Theory, Vol. IT-22, Jan. 1976, pp. 82-83.

[31] J.E. Savage, "Sequential decoding - The computation problem", Bell Syst. Tech. J., Vol. 45, Jan. 1966, pp. 149-175.

[32] G.D. Forney, Jr., "Maximum-likelihood sequence estimation in the presence of intersymbol interference", IEEE Trans. Inform. Theory, Vol. IT-18, May 1972, pp. 363-378.

[33] J. Uddenfeldt, "A performance bound for tree search coding of speech with minimum phase codes", Conf. Record, 1978 International Conf. on Communications, Toronto, Canada, June 1978, pp. 34.2.1-34.2.5.

[34] J.D. Gibson and A.C. Goris, "Incremental and variable-length tree coding of speech", Conf. Record, 1979 International Conf. on Communications, Boston, MA, June 10-14, 1979, pp. 8.5.1-8.5.5.

[35] G.P. Ashkar and J.W. Modestino, "The Contour extraction problem with biomedical applications", Computer Graphics and Image Processing, Vol. 7, 1978, pp. 331-355.

[36] J.W Modestino, V. Bhaskaran, and J.B. Anderson, "Tree encoding of images in the presence of channel errors", in submission, Aug. 1979.

[37] S.G. Wilson and J.R. Troxel, "Facsimile coding: distortion measures, code generation, and tree encoding", Conf. Record, 1979 International Conf. on Communications, Boston, MA, June 10-14, 1979, pp. 8.4.1-8.4.5.

[38] J.A. Stuller and B. Kurz, "Intraframe sequential picture coding", IEEE Trans. on Communications, Vol. COM-25, No. 5, May 1977, pp. 485-495.

[39] J.A. Heller and I.M. Jacobs, "Viterbi decoding for satellite and space communication", IEEE Trans. Commun. Technol., Vol. COM-19, Oct. 1971, pp. 835-847.

[40] I.M. Jacobs, "Sequential decoding for efficient communication from deep space", IEEE Trans. Commun. Technol., Vol. COM-15, Aug. 1967, pp. 492-501.

[41] J.L. Massey and D.J. Costello, "Nonsystematic convolutional codes for sequential decoding in space applications", IEEE Trans. Commun. Technol., Vol. COM-19, Oct. 1971, pp. 806-813.

[42] I.M. Jacobs, "Practical applications of coding", IEEE Trans. Inform. Theory, Vol. IT-20, No. 3, May 1974, pp. 305-310.

[43] S.L. Bernstein, D.A. McNeill, and J. Richer, "A signalling scheme and experimental receiver for extremely low frequency communications", IEEE Trans. on Communications, Vol. COM-22, April 1974, pp. 508-528.

[44] G.D. Forney, Jr., "Use of a sequential decoder to analyze convolutional code structure", IEEE Trans. Inform. Theory, Vol. IT-16, Nov. 1970, pp. 793-795.

[45] P.R. Chevillat, "Fast sequential decoding and a new complete decoding algorithm", Tech. Rep. EE7606-32-01, Dept. Elec. Eng., Illinois Inst. Tech., Chicago, Illinois, June 1976.

[46] G.P. Ashkar and J.W. Modestino, "A mathematical programming approach to sequential decoding strategies", unpublished.

[47] F.P. Preparata and S.R. Ray, "An approach to artifical non-symbolic cognition", Inform. Sci., Vol. 4, Jan. 1972, pp. 65-86.

[48] J. Uddenfeldt and L.H. Zetterberg, "Algorithms for delayed encoding in delta modulation with speech-like signals", IEEE Trans. on Communications, Vol. COM-24, June 1976, pp. 652-658; see also "Adaptive delta modulation with delayed decision", IEEE Trans. Commun., Vol. COM-22, Sept. 1974, pp. 1195-1198 (same authors).

[49] A.V. Aho, J.E. Hopcroft, and J.D. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley, Reading, MA,, 1974.

[50] C.R. Davis and M.E. Hellman, "On tree coding with a fidelity criterion", IEEE Trans. Inform. Theory, Vol. IT-21, July 1973, pp. 373-378.

[51] J.B. Anderson and C.-W.P. Ho, "Architecture and construction of a hardware sequential encoder for speech", IEEE Trans. Commun., Vol. COM-25, July 1977, pp. 703-707.

[52] W.H. Ng and R.M.F. Goodman, "An efficient minimum-distance decoding algorithm for convolutional error-correcting codes", Proc. IEE, Vol. 125, No. 2, Feb. 1978.

[53] R.J. Dick, "Tree encoding for Gaussian sources", Ph.D. dissertation, Sch. Elec. Eng., Cornell Univ., Ithaca, N.Y., May 1973; see also R.J. Dick, T. Berger, and F. Jelinek, "Tree encoding of Gaussian sources", IEEE Trans. Inform. Theory, Vol. IT-20, May 1974.

[54] S. Mohan and J.B. Anderson, "Stack algorithm speech encoding", 1977 IEEE International Symposium on Information Theory, Ithaca, NY, 10-14 October 1977.

[55] D.E. Knuth, The Art of Computer Programming, Vol. III: Sorting and Searching, Addison-Wesley, Reading, MA, 1973.

[56] J.B. Anderson and S. Mohan, "A systematic analysis of cost for sequential coding algorithm", Communications Res. Lab., McMaster Univ., Hamilton, Ont., CRL Report #56, July 1978.

[57] D.E. Knuth, The Art of Computer Programming: Vol. I: Fundamental Algorithms, Addison-Wesley, Reading, MA, 1968.

[58] E.M. Reingold, J. Nivergelt, and N. Deo, Combinatorial Algorithms: Theory and Practice, Prentice-Hall, Englewood Cliffs, NJ, 1977.

[59] G.M. Adelson-Velskii and YE.M. Landis, "An algorithm for the organisation of information", Soviet Math. Dokl., Vol. 3, 1962, pp. 1259-1262.

[60] J. Nivergelt, "Binary search trees and file organization", Computing Surveys, Vol. 6, No. 3, Sept. 1974, pp. 195-207.

[61] P. Larson, "Dynamic hashing", BIT, Vol. 18, 1978, pp. 184-201.

[62] E. Fredkin, "Trie memory", Comm. ACM, Vol. 3, 1960, pp. 490-500.

[63] S. Mohan and J.B. Anderson, "Data structures and complexity measures for new source coding algorithms", 1979 IEEE International Symposium on Information Theory, Grignano, Italy, June 25-29, 1979.

[64] W. Smith and W. Wilkinson, "On branching processes in random environments", Ann. Math. Statist., Vol. 40, 1969, pp. 814-827.

[65] H.H. Tan, "Tree coding of discrete time abstract alphabet stationary block-ergodic sources with a fidelity criterion", IEEE Trans. Inform. Theory, Vol. IT-22, No. 6, Nov. 1976, pp. 671-681.

[66] R. Johannesson, "On the distribution of computation for sequential decoding using the stack algorithm", IEEE Trans. Inform. Theory, Vol. IT-25, No. 3, May 1979, pp. 323-331.

[67] B. Haskell, "The computation and bounding of rate-distortion functions", IEEE Trans. Inform. Theory, Vol. IT-15, Sept. 1969, pp. 525-531.

[68] K.B. Athreya and S. Karlin, "On branching processes with random environments: I - Extinction probabilities", Ann. Math. Statist., Vol. 42, 1971, pp. 1499-1520.

[69] T.E. Harris, The Theory of Branching Processes, Springer-Verlag, NY, 1972.

[70] K.B. Athreya and P.E. Ney, Branching Processes, Springer-Verlag, NY, 1972.

[71] C.J. Mode, Multitype Branching Processes: Theory and Applications, American Elsevier, NY, 1971.

[72] P. Jagers, Branching Processes with Biological Applications, Wiley, NY, 1975.

[73] W. Feller, An Introduction to Probability Theory and Its Applications, Vol. 1, 3rd ed., John Wiley, NY, 1968.

[74] S. Mohan and J.B. Anderson, "Branching process methods for the single stack encoding algorithm", Proceedings, Sixteenth Annual Allerton Conference on Communications, Control, and Computing, Monticello, Illinois, Oct. 4-6, 1978, pp. 961-970.

[75] J.B. Anderson and S. Mohan, "A push-down stack measure of encoding algorithm complexity", Conf. Rec., International Conference on Communications, Toronto, Canada, June 4-7, 1978.

[76] S. Mohan and J.B. Anderson, "Speech encoding by the stack algorithm", Commun. Res. Lab., McMaster Univ., CRL Rep. #64, May 1979; also, to appear, IEEE Trans. Commun.

[77] J.B. Anderson and C.-W. Law, "Real-number convolutional codes for speech-like quasi-stationary sources", IEEE Trans. Inform. Theory, Vol. IT-23, Nov. 1977, pp. 778-782.

[78] J.B. Anderson and S. Mohan, "A cost function for sequential coding", Proceedings, Sixteenth Annual Allerton Conference on Communications, Control, and Computing, Monticello, Illinois, Oct. 4-6, 1978, pp. 725-734.