BIDIRECTIONAL SEQUENTIAL DECODING

by

Kaiping Li

M. A. Sc., Northern Jiao-Tong University, Beijing, P. R. of China, 1986

B. A. Sc., Northern Jiao-Tong University, Beijing, P. R. of China, 1983

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

in

THE FACULTY OF GRADUATE STUDIES

DEPARTMENT OF ELECTRICAL ENGINEERING

We accept this thesis as conforming

to the required standard

The UNIVERSITY OF BRITISH COLUMBIA

June 1994

In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

(Signature)

Department of Electrical Engineering

The University of British Columbia
Vancouver, Canada

Date June 20, 1994

DE-6 (2/88)

Abstract

The main drawback of sequential decoding is the variability of its decoding effort which could cause decoding erasures. We propose and analyze in this dissertation efficient *bidirectional sequential decoding* (BSD) techniques to alleviate this drawback. In the proposed BSD, two decoders are used; one is called *forward decoder* (FD), and is used to search the tree from forward direction; while the other is called *backward decoder* (BD), and is used for the backward search of the tree. Forward decoding and backward decoding are performed simultaneously, and stop somewhere in the tree. In one class of BSD, to which we refer as BSD-merge, decoding stops whenever FD and BD merge at a common encoder state some where in the tree. In the other class of BSD, that is BSD-no-merge, no common encoder state is required, and decoding stops when FD meets BD. Different BSD algorithms based on the stack algorithm are constructed; namely Algorithm TAmeet which belongs to the class of BSD-no-merge, Algorithm TAmerge and Algorithm TTmerge which belong to BSD-merge, and finally Algorithm HTTmerge which is a hybrid version of BSD-merge and BSD-no-merge.

The relationships between backward coding and forward coding are examined in detail. Good convolutional codes, with memory m ranging from 2 to 25, suitable for bidirectional decoding, found through extensive computer search, are provided. These codes possess the same distance properties from both forward and backward directions.

It is found by analysis and computer simulations that the distribution of the total number of computations per decoded block of the proposed BSD is still Pareto, as that of *unidirectional sequential decoding* (USD). However, the advantage of the proposed

BSD appears as an increase in the Pareto exponent, and hence as a decrease in the computational variability and erasure probability. More specifically, we prove by using the random coding approach that the Pareto exponent of BSD using Algorithm TAmeet is asymptotically twice that of USD, and conjecture that this also applies to Algorithm TAmerge. On the other hand, it is found that the computational cutoff rate remains unchanged, but the use of our BSD reduces the average number of computations per decoded bit.

Using the random coding approach, we show that the error performance of BSD-merge is asymptotically the same as that of USD. Moreover, we show that the bit error probability of Algorithm TAmeet satisfies the random coding bound for block codes. Computer simulations are provided to confirm the analytical findings.

The use of BSD-merge substantially reduces the computational variability of conventional sequential decoding without compromising the error performance. However, BSD does not completely eliminate the erasure problem. We therefore combine our BSD idea in conjunction with the *multiple stack algorithm* (MSA), which is an erasure-free decoding algorithm. It is shown through analysis and computer simulations that the new *bidirectional multiple stack algorithm* (BMSA) offers substantial advantages over the MSA in terms of computational effort, memory requirements and error performance. The BMSA appears as an attractive alternative to the *Viterbi algorithm* (VA) where low error probabilities and high decoding speeds are required.

Contents

List of Tables

List of Figures

List of Symbols

$\triangleq$	Definition
$\lceil \tau \rceil$	Smallest integer not less than τ
$\lfloor \tau \rfloor$	Largest integer not greater than τ
α	Defined by $R = \hat{E}_o(\alpha)/\alpha$
Δ	Substack spacing
γ	Fano metric
λ	One branch metric drop
ρ_r	Defined by $R = E_o(\rho_r)/\rho_r$
$\phi(\cdot)$	Indicator function
$a_{d_{free}+i}$	Total number of paths of weight $d_{free} + i$.
C	$\triangleq 2 \max_i \left[\min\left(C_i^F, C_{L-i}^B\right)\right]$
C_b^{BMSA}	Total number of computations needed to decode one branch using the BMSA
C_b^{BSD}	Total number of computations needed to decode one branch using BSD
C_b^{MSA}	Total number of computations needed to decode one branch using the MSA
C_b^{USD}	Total number of computations needed to decode one branch using USD
C_{crit}	Critical number of computations
C_p	Channel capacity

$c_{d_{free}+i}$	Total number of nonzero information bits on all weight $d_{free}+i$ paths
C_L^{BSD}	Total number of computations required to decode a block of L branches by any BSD algorithm
C_L^F, C_L^B	Total number of computations required by forward and backward decoders to decode a whole block of L branches, respectively
C_x^F, C_{L-x}^B	Total number of computations needed by forward and backward decoders to decode x or $(L-x)$ branches without searching beyond the boundary at level x, respectively
$\hat{C}_x^F, \hat{C}_{L-x}^B$	Total number of computations required by forward and backward decoders to correctly decode x or $(L-x)$ branches, respectively
C_i	The i-th branch computations
C_L	Total number of computations needed to decode a block of L branches by using Algorithm TAmeet
$\hat{C}_L$	Total number of computations required to decode a block of L branches by using Algorithm TAmerge with the assumption of correct decoding
$C_L^{TAmerge}$	Total number of computations required to decode a block of L branches by using Algorithm TAmerge
C_L^{BMSA}	Total number of computations required to decode a block of L branches by using the BMSA
C_L^{MSA}	Total number of computations required to decode a block of L branches by using the MSA

C_{lim}	Computational limit
C_{Λ}^{USD}	Total number of computations required to correctly decode the first Λ branches using USD
D	Unit delay
$\boldsymbol{d}$	Distance profile
d_{free}	Free distance
$d_c^F[\cdot], d_c^B[\cdot]$	Column distance function of forward and backward codes, respectively
$\mathfrak{D}_i$	Incorrect path subset generated from level i
$E[\cdot]$	Expectation function
$E_o(\cdot)$	Gallager function
$\hat{E}_o(\cdot)$	Smallest concave function $\geq E_o(\cdot)$
$\mathbf{G}(D)$, $\mathbf{G_r}(D)$	Transfer function matrix of forward and backward codes, respectively
$H(\cdot)$	Entropy function
$\mathbf{I}$	Identity matrix
K	Code constraint length
k	Number of bits per unit time at the input of an encoder
L	Block length (including a known tail)
l	Meeting point
l_F, l_B	The farthest level reached by forward and backward decoders, respectively

$l_{F_{TOP}}, l_{B_{TOP}}$	The length of top path in FS and BS, respectively
l_o	Merging point
$M[\cdot]$	Path metric function
$\mathfrak{M}$	Total encode memory
m	Encode memory order
m_h	Number of matching information symbols in the merging test
N_b	Total number of bit errors given S_{mg} incorrect
n	Number of bits per unit time at the output of an encoder
$n_b(\tau)$	Expected number of bit errors caused by an incorrect path diverging at level τ
O_v	$\triangleq l_F + l_B - L$ at the end of decoding.
P_1	The first (primary) stack overflow probability
P_b	Bit error probability
P_e	Block error probability
P_{er}	Erasure probability
p	Crossover probability of BSC
R	Code rate (nats per channel symbol)
r	Code rate (bits per channel symbol)
R_{comp}	Computational cutoff rate
S_F, S_B	Current number of forward and backward stacks, respectively
S_i	Encoder state i

S_{mg}	Common encoder state at l_o
S_s	The first (primary) stack size saving ratio
$\mathbf{x}(i)$	Correct path from root node to level i
$\mathbf{x}'_i(t)$	Incorrect path which diverges from the correct path at level i and remerges at level $(i + t)$
$\tilde{\mathbf{x}}_i(t)$	Incorrect path which diverges from the correct path at level i and stretches out t branches from the correct node i
$\mathfrak{X}'(i)$	The set of all possible incorrect paths which diverge from the correct path at level i
$\tilde{\mathfrak{X}}(i;j)$	The set of all incorrect paths which diverge and remerge with the correct path at level i and $(L - j)$
$\left\|\tilde{\mathfrak{X}}(i;j)\right\|$	Total number of elements in $\tilde{\mathfrak{X}}(i;j)$
Z	$\triangleq Z_1^{MSA} = 2Z_1^{BMSA}$
Z_l	$\triangleq Z_i^{MSA} = 2Z_i^{BMSA}, i = 2, 3, \cdots$
Z_i^{BMSA}	The i-th forward or backward stack size in the BMSA
Z_i^{MSA}	The i-th stack size in the MSA

Acknowledgment

I would like to thank my wife, Libing, for her constant encouragement, support, and understanding during these years. I am also greatly indebted to my research supervisor, Dr. Samir Kallel, for his continual encouragement and highly constructive comments. I wish to express my sincere thanks for his guidance throughout this research.

I would like to thank Mr. Dimitrios P. Bouras, Mr. Siavash Massoumi, Mr. Longxiang Dai, and Mr. Weimin Sun for their helpful discussions.

I would also like to thank Dr. Samir Kallel, Mr. Siavash Massoumi, and Mr. Dimitrios P. Bouras for their help with transcript changes after I joined OKI Telecom. Finally, I would like to thank Mrs. Vicki Waldro-Snider and Mr. Peter Howard for their invaluable help in perfecting the use of the English language throughout this text.

Chapter 1
Introduction

Digital communication deals primarily with the process of transmitting digital information from one point to another through a channel which is subject to noise disturbances. This implies that the information received at the destination (prior to processing) is not always exactly the same as transmitted. Shannon [1] showed in his famous coding theorem that digital information can be transmitted with arbitrarily low error probability provided that the data rate is smaller than the channel capacity. Since then, numerous coding and decoding techniques have been proposed in the attempt to approach Shannon's promise of reliable communication near the channel capacity. The increasing demand for efficient and reliable digital communications over noisy channels has led to the use of sophisticated coding and decoding techniques for the purpose of error correction. These techniques, called *forward error correction* (FEC) are useful, especially with power-limited systems.

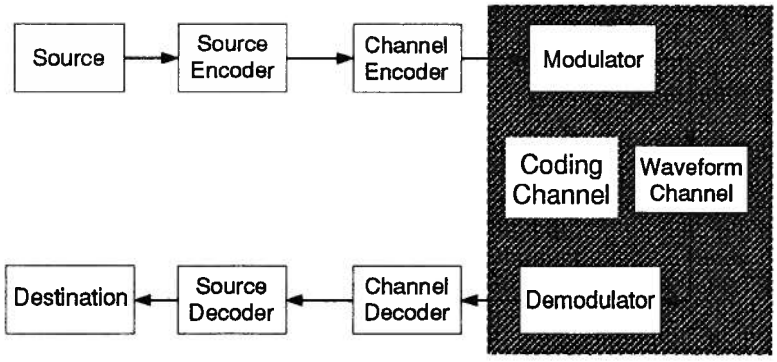


Figure 1.1 General digital communication system.

A general block diagram of a digital communication system is given in Figure 1.1. An arbitrary source generates a signal which is then transformed by a source encoder into a redundancy-free digital information sequence. If the source is analog, *analog-to-digital* (A/D) conversion is performed at the source encoder. Generally, source and source encoder are treated as a digital source which emits an equal likely binary sequence. The information sequence then enters the channel encoder which adds redundancy to combat the channel noise. The outputs from the channel encoder, referred to as codewords, are then mapped into a waveform signal by the modulator for transmission over the waveform channel. The waveform channel may represent a telephone line, a wireless radio link, a space communication link, etc. The demodulator processes the received waveform signal and produces an output that may be discrete (quantized) or continuous (unquantized). The output of the demodulator is called the received sequence. The channel decoder, based on the received sequence and the known structure of the channel encoder has to select the best possible transmitted sequence. The purpose of the channel encoder and decoder is to allow the digital information sequence to be efficiently and reliably reproduced at the output of the channel decoder. The decoded sequence then enters the source decoder, which reproduces a signal that is delivered to the user. If the source signal is continuous, this also involves *digital-to-analog* (D/A) conversion.

To focus attention on the channel encoder and channel decoder, the *modulator/demodulator* (modem) and the waveform channel are combined into a coding channel. In this dissertation, a *discrete memoryless channel* (DMC) is assumed, which constitutes the simplest class of coding channel models and is defined as follows. The input is a sequence of elements from a finite alphabet, and the output is a sequence of elements

from the same or different alphabet. Moreover, each element in the output sequence is statistically dependent only on the element in the corresponding position of the input sequence and is determined by a fixed conditional probability assignment. For example, the *binary symmetric channel* (BSC), which is shown in Figure 1.2, is a DMC with binary input and output alphabets, where each digit at the channel input is reproduced correctly at the channel output with some fixed probability ($1 - p$) and is altered by noise into the opposite digit with probability p.

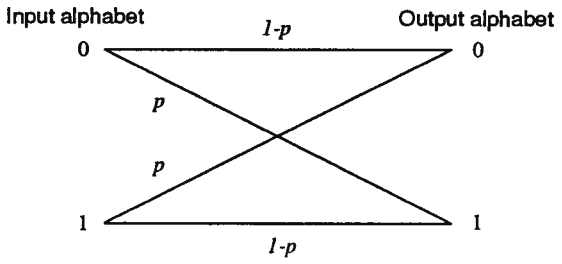


Figure 1.2 Binary symmetric channel.

A multitude of methods on coding and decoding techniques which permit low error probability in digital communications have been proposed and investigated. Since excellent introductory literature into coding/decoding theory is available [2–8], the following survey will therefore be limited to matters relevant to this thesis, that is convolutional encoding with probabilistic decoding. Supporting literature is found in the references.

A convolutional code is described by three integers, n, k, and m. At each unit time, the encoder accepts k-bit of the information sequence and produces a codeword (encoded sequence) of n-bits. The integer m is a parameter known as the code memory length. An important characteristic of an (n, k, m) convolutional code is that the encoder has memory

— the n-bits emitted by the convolutional encoding procedure is not only a function of the k-bit input, but is also a function of its previous m inputs.

Viterbi decoding [9] and sequential decoding[1] [11] are the two main probabilistic decoding methods for convolutional codes. The *Viterbi algorithm* (VA) is an optimum maximum likelihood decoding scheme, but its optimality is obtained at a large decoding effort. The number of computations performed per information digit is constant, but increases exponentially with the code memory length. In practice, the VA can thus be employed only with short memory codes.

Historically, sequential decoding was proposed in 1957 by Wozencraft and Reiffen [11] as the first practical decoding procedure for convolutional codes. In sequential decoding, the tree-like structure of the convolutional code is used in a step-by-step search of the most likely transmitted sequence. As long as the data rate does not exceed a quantity called computational cutoff rate R_{comp}, which is less than the channel capacity, the average number of computations required to decode one information digit is small and independent of the code memory length.

The error probability of sequential decoding decreases exponentially with the code memory length [12], and it is lower bounded by that of the VA. As $m \rightarrow \infty$, however, both decoding techniques yield the same error performance. In addition, the decoding complexity of sequential decoding is practically insensitive to the code memory. Consequently, convolutional encoding with sequential decoding is at present one of the best known methods to achieve an arbitrarily low error probability over a DMC [7, 8, 13, 14].

---

[1] In this dissertation, sequential decoding is used in a narrow sense. That is, it only includes the metric-first and the depth-first algorithms although the breadth-first algorithm can be categorized as sequential decoding in a wide sense [10].

Convolutional coding with sequential decoding has been used in various systems such as satellite communications, deep space communications, etc.

There are two principle sequential decoding algorithms: the Fano algorithm [13] and the stack algorithm proposed independently by Zigangirov [15] and Jelinek [16]. Regardless of the algorithm, a major problem with sequential decoding is the computational variability. As a consequence of this variability, the decoding effort for a given received sequence may occasionally exceed the physical limitations of the decoder, leading inevitably to buffer overflows and information erasures. Thus, while the undetected error probability of sequential decoding can be made arbitrarily small, the erasure probability or buffer overflow remains substantial [7, 8, 17]. Although the disadvantage of overflows or erasures of sequential decoding may be useful in systems with retransmission capabilities, it is desirable to make the erasure probability as small as possible without compromising the error performance in order to achieve higher throughput and reliability. In addition, there are many other cases where erasures are undesirable. For example, in real-time systems with long path delays, erasures can cause complete data loss. In this case, the main contribution of errors comes from the buffer overflow. Thus, it is very important to reduce the erasure probability. The prime objective of this research is tailored at alleviating the computational variability, and thus minimizing erasures in sequential decoding.

In the past, several modifications of sequential decoding to reduce the erasure probability have been reported and implemented. Excessively long tree searches can be terminated early by placing a backsearch limit on the Fano and the stack algorithms,

thereby lowering the erasure probability [14]. A similar effort is achieved with the stack algorithm by deleting the path with the lowest metric from the stack whenever the stack fills up [15, 16]. Although these methods accelerate decoding and reduce the erasure probability, the resulting increase in the error probability is found to be substantial [18].

Haccoun and Ferguson [19] proposed a class of generalized (or multiple path) stack algorithms which extends several paths simultaneously from the top of the stack. It was shown that the variability of the computational effort is reduced, but at a cost of a large average number of computations compared to the ordinary stack algorithm.

In order to eliminate erasures entirely, Chevillat and Costello [20, 21] introduced an interesting *multiple stack algorithm* (MSA). It is a modification of the stack algorithm and uses several stacks to accommodate the problem of stack overflow. However, it is found that the resulting increase in stack and buffer storage is substantial.

Forney [22] in 1967 suggested that sequential decoding can be started from the end of a block which is terminated by a known tail. He proposed a bidirectional decoding scheme which uses a forward decoder and a backward decoder to search a terminated tree from initial and final encoder states, respectively. Decoding with Forney's scheme stops whenever either the forward or the backward decoder reaches the end of its tree. Computer simulations revealed that this scheme does not offer much improvement in terms of computational performance.

The idea of bidirectional search has also been used for computing the free distance of convolutional codes [23, 24] and distance spectrum of trellis codes [25]. Recently, a bidirectional decoding search has been applied to the decoding of block codes [26].

Most recently, Haccoun and Belzile [27, 28] investigated bidirectional breadth-first M algorithm for the decoding of convolutional codes. Although erasures are avoided by the nature of the breadth-first M algorithm used, the loss of the correct path problem, which is inherent to the M algorithm, limits the performance of this scheme.

We propose and analyze in this dissertation efficient *bidirectional sequential decoding* (BSD) techniques [29]. It is shown that the proposed BSD techniques substantially alleviate the drawbacks of conventional or *unidirectional sequential decoding* (USD). In the proposed BSD, two decoders are used, like in Forney's scheme; one is called *forward decoder* (FD), and is used to search the tree from forward direction; the other is called *backward decoder* (BD), and is used for the backward search of the tree. Forward decoding and backward decoding are performed simultaneously. However, unlike Forney's scheme, decoding in the proposed BSD techniques stops somewhere in the tree. In one class of BSD, which we refer to as BSD-merge, decoding stops whenever FD and BD merge at a common encoder state somewhere in the tree. In the other class of BSD, that is BSD-no-merge, decoding stops when FD meets BD. Different BSD algorithms based on the stack algorithm are constructed; namely Algorithm TAmeet which belongs to the class of BSD-no-merge, Algorithm TAmerge and Algorithm TTmerge which belong to BSD-merge, and Algorithm HTTmerge which is a hybrid version of BSD-merge and BSD-no-merge. Finally, to eliminate erasures entirely, we combine Algorithm TTmerge in conjunction with the MSA of Chevillat and Costello [20, 18, 21], to obtain a more efficient and reliable erasure-free algorithm, which we call *bidirectional multiple stack algorithm* (BMSA).

1.1 Outline of the Dissertation

Chapter 2 is a brief summary of the basic elements of convolutional coding and sequential decoding. The idea and basic properties of backward coding/decoding are also presented in this chapter. Moreover, good convolutional codes suitable for bidirectional decoding, which are found through computer search, are listed in this chapter.

In Chapter 3, the proposed BSD algorithms are described and their fundamental properties are investigated. Furthermore, our proposed BSD algorithms are compared with the existing BSD algorithms which can be classified under the category of BSD-no-merge in a wide sense.

A theoretical analysis of the computational performance of the proposed BSD algorithms is given in Chapter 4, together with extensive computer simulation results. It is found, both by theoretical analysis as well as computer simulations, that the distribution of the total number of computations per decoded block by any of the proposed BSD algorithm is still Pareto, as that of USD. However, the advantage of the proposed BSD appears as an increase in the Pareto exponent, and hence as a decrease in the computational variability. More specifically, we prove by using the random coding approach that the Pareto exponent of BSD using Algorithm TAmeet is asymptotically twice that of USD, and conjecture that this also applies to Algorithm TAmerge. On the other hand, it is found that the computational cutoff rate remains unchanged, but the use of the proposed BSD reduces the average number of computations per decoded bit. This reduction in computational variability and average number of computations translates into a substantial decrease in the erasure probability (or buffer overflow problem), which is the main

drawback of sequential decoding.

The error performance of the proposed BSD techniques is explored in Chapter 5 by both theoretical analysis and computer simulations. It is found that BSD-merge has asymptotically the same error performance as USD, while the bit error probability of Algorithm TAmeet follows the random coding bound for block codes.

In Chapter 6, an erasure-free BSD technique based on the MSA [18, 21], which we call *bidirectional multiple stack algorithm* (BMSA), is proposed and analyzed. It is found that the BMSA performs better than the conventional MSA, in terms of memory requirements, computational effort and error performance.

Chapter 7 contains final conclusions and suggestions for future research.

Appendix A contains a rigorous proof of Theorem 4.4 which suggests that under the assumption of correct decoding the computational cutoff rate of BSD is the same as that of USD.

Appendix B lists acronyms used in this dissertation.

1.2 Claim of Contributions of this Dissertation

The major contributions of this research are summarized below.

The properties of backward coding/decoding are examined in great detail. More specifically, the relationships between a backward code and the corresponding forward code are given through Theorems 2.1 and 2.2, Corollaries 2.1 and 2.2 and Properties 2.1 and 2.2. Good convolutional codes suitable for bidirectional decoding found through computer search are provided.

Efficient BSD techniques that alleviate the drawbacks of conventional sequential decoding are proposed and examined by analysis and extensive computer simulations. The main theoretical contributions in the analysis of the proposed BSD techniques are as follow:

1. A computational upper bound for Algorithm TAmeet is found for a specific time-invariant convolutional code (Theorem 4.1).
2. A computational upper bound for Algorithm TAmeet is discovered for an ensemble of trellis codes (Theorem 4.2). It is conjectured that this bound also applies to Algorithm TAmerge.
3. A computational lower bound is obtained for any code and any type of BSD (Theorem 4.3).
4. The computational cutoff rate of BSD is found to be the same as that of USD (Theorem 4.4).
5. The error performance of BSD-merge is proved to be asymptotically the same as that of USD (Theorems 5.1 to 5.3, Corollaries 5.1 and 5.2).
6. The bit error probability of Algorithm TAmeet is found to satisfy the random coding bound for block codes.

An erasure-free algorithm, the BMSA, is proposed and analyzed. Through computer simulations and analysis, we demonstrate that the new BMSA performs better that the conventional MSA.

Chapter 2
Convolutional Coding and Sequential Decoding

2.1 Convolutional Coding

An (n, k, m) convolutional encoder can be implemented with k shift registers, n modulo-2 adders and a commutator that scans the output of the adders. The length of each of the k shift registers is less than or equal to the encoder memory order m [8].[2] The connections of the n adders to the k shift registers define the code. This is generally specified with a so-called $k \times n$ *transfer function matrix* $\mathbf{G}(D)$ [8], given by

$$\mathbf{G}(D) = \begin{bmatrix} g_1^{(1)}(D) & g_1^{(2)}(D) & \cdots & g_1^{(n)}(D) \\ g_2^{(1)}(D) & g_2^{(2)}(D) & \cdots & g_2^{(n)}(D) \\ \vdots & \vdots & g_i^{(j)}(D) & \vdots \\ g_k^{(1)}(D) & g_k^{(2)}(D) & \cdots & g_k^{(n)}(D) \end{bmatrix}, \tag{2.1}$$

where $g_i^{(j)}(D)$ is a polynomial of degree $\leq m$ that indicates the connection of the j-th adder to the i-th shift register. For example, Figure 2.1 represents a (3, 1, 2) encoder with $\mathbf{G}(D) = \left[1 + D, 1 + D^2, 1 + D + D^2\right]$.

Encoding is performed k-bit at a time. Each k-bit to be encoded is fed to the encoder from the left side, and the content of the k shift registers is shifted one position to the right. The modulo-2 adders are then sampled in sequence by the commutator, producing n output coded bits. For convenience, we refer to this as *forward encoding*.

Defining $\mathfrak{M} \triangleq \sum_{i=1}^{k} \max_{1 \leq j \leq n} \left[\deg g_i^{(j)}(D)\right]$ as the total encoder memory [8] and the state of the encoder as its k shift registers' contents, there are a total of $2^{\mathfrak{M}}$ different possible

[2] The memory order m is related with the constraint length K [7], which is defined as the maximum number of present and past encoder input symbols that influence any output symbol, i.e., $K = m + 1$.

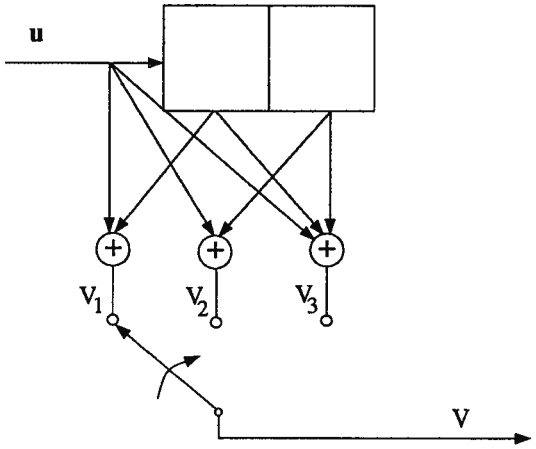


Figure 2.1 Convolutional encoder for the (3, 1, 2) code.

states. Then, the operations of the encoder can be described using a state diagram, where the n output bits are a function of the state and the k input bits [8]. Operations of a convolutional encoder can also be described by a trellis or a tree diagram [8]. Figures 2.2 and 2.3 represent six-level trellis and tree diagrams, respectively, for the (3, 1, 2) encoder of Figure 2.1, corresponding to information sequence of length *4* bits with a zero tail of length 2. The upper branch leaving each state in both the trellis and tree diagrams in Figures 2.2 and 2.3 represents input one, while the lower branch represents input zero. Starting from the initial state S_0, each path in the trellis or tree diagram corresponds to a possible encoded sequence as generated by the encoder. For example, the highlighted path on the forward code tree of Figure 2.3 corresponds to information sequence (110100) and code sequence (111, 010, 110, 100, 101, 011).

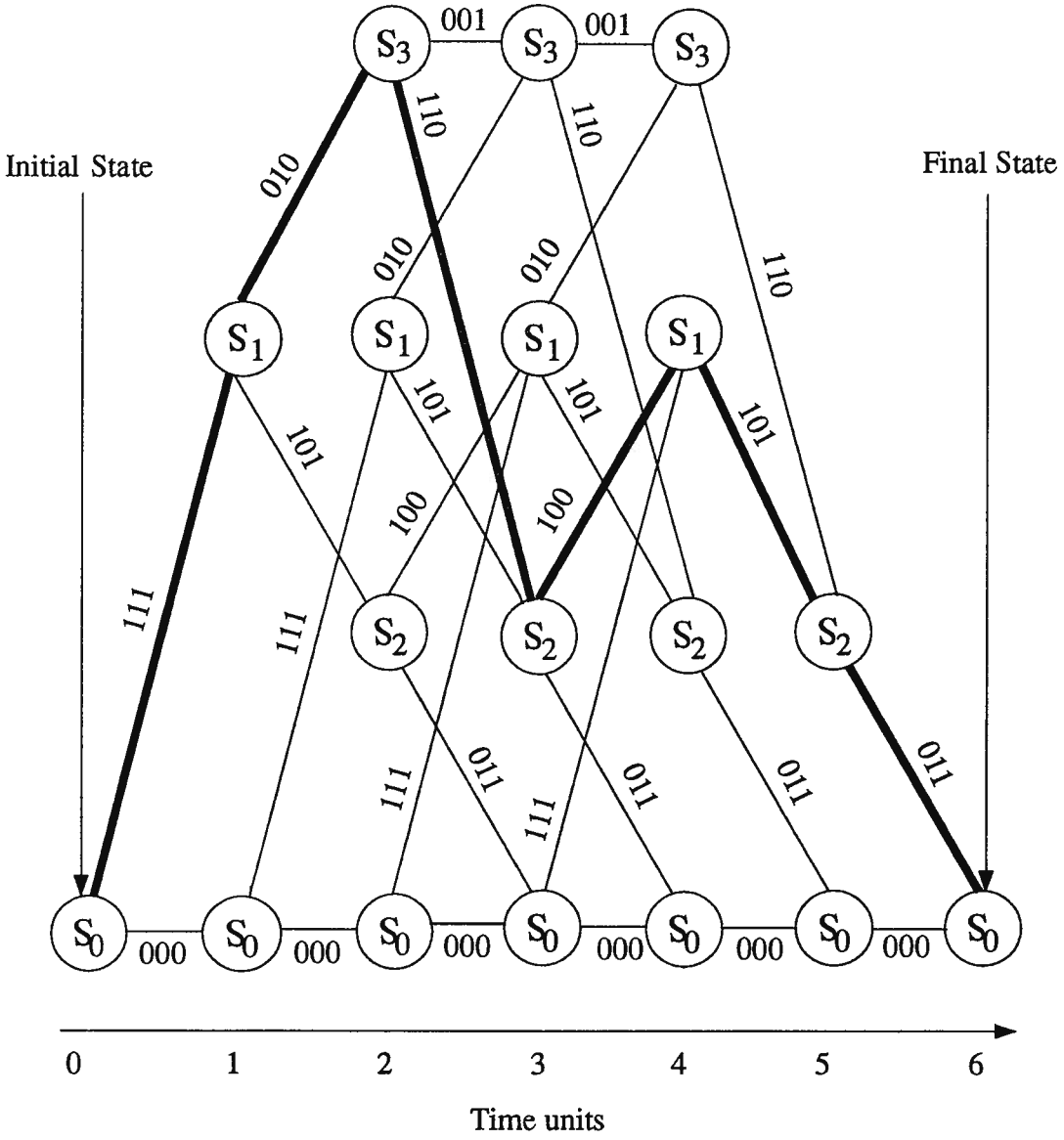


Figure 2.2 Forward code trellis diagram for the encoder of Figure 2.1.

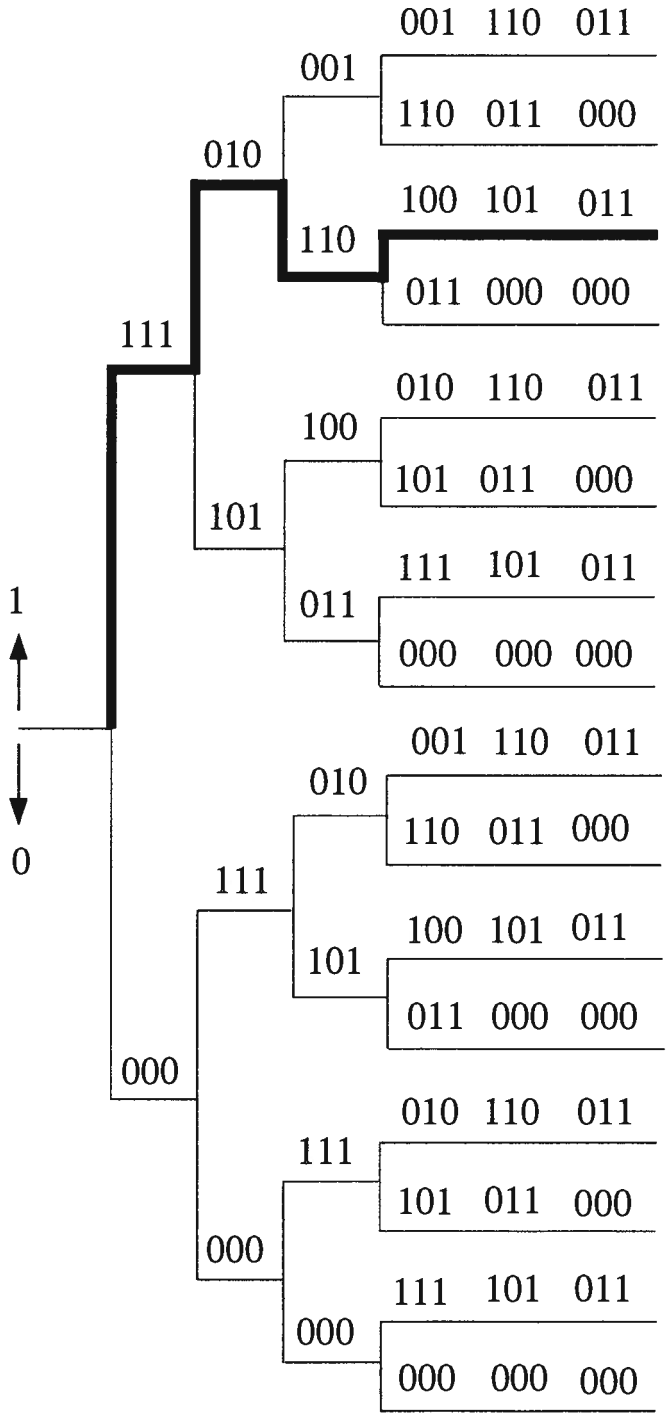


Figure 2.3 Forward code tree diagram of the encoder of Figure 2.1.

The code rate of an (n, k, m) convolutional code is $r = k/n$ information bits per coded bit or channel symbol. The code rate can also be expressed as

$$R \triangleq \ln(u)/n, \quad nats\ per\ channel\ symbol \tag{2.2}$$

where $u = 2^k$. We have thus $R = r \ln 2$.

The error correcting capabilities of a convolutional code are intimately related with the memory m and code rate r. The larger is m and/or the smaller is r, the more powerful is the code. Lowering the rate of the code involves a higher bandwidth expansion for transmission, which might be problematic in certain applications. It is therefore desirable to increase m as much as possible.

The two main probabilistic decoding techniques for convolutional codes are Viterbi decoding [9] and sequential decoding [11]. Given a received sequence of channel symbols and a code structure, a Viterbi decoder explores the whole code trellis in order to decide which is the most probable path or code sequence to have been transmitted. *Viterbi algorithm* (VA) is thus an optimum decoding technique in the maximum likelihood sense. However, due to the amount of computations needed for decoding, which grows exponentially with the code memory m, the VA is practically limited to short memory codes (typically $m \leq 7$) [7, 8]. A sequential decoder on the other hand attempts to make the best decision by only exploring a small portion of the code tree (or trellis). Sequential decoding is thus suboptimum, but can be used with long memory convolutional codes. Unfortunately, sequential decoding has serious drawbacks, and our efforts in this thesis are oriented towards the goal of alleviating some of these drawbacks.

2.2 Sequential Decoding

In a system using convolutional coding and sequential decoding, information is usually transmitted in blocks ($L - m$ branches), and each block is terminated by a known tail (usually m zero branches). Sequential decoding is a sub-optimum tree search procedure that only explores a fraction of the encoded tree in order to determine the most likely transmitted path for a given received sequence. The basic idea of sequential decoding is as follow. Assuming a DMC, starting from the root node, a sequential decoder explores the code tree, one branch at a time and uses the log-likelihood function or Fano metric [13] given by

$$\gamma_i = \ln \frac{P(y_i \mid x_i)}{P(y_i)} - R, \tag{2.3}$$

where x_i is the i-th channel input symbol ($1 \leq i \leq nL$), y_i is the corresponding received symbol, and $P(y_i) = \sum_j P(y_i|x_j)q(x_j)$ is the *a priori* probability for the received symbol y_i with $q(x_j)$ being the distribution of channel input symbol x_j. The total metric for a path of length l branches is then

$$\sum_{i=1}^{nl} \gamma_i, \tag{2.4}$$

and a sequential decoder, based on the stack algorithm, will always attempt to search and extend the path having the largest cumulative metric. At the end of the tree, the path with the highest metric is accepted as the decoded path. Thus, while this decoder is moving forward into the tree, it occasionally realizes that it is not following the best current path, and backs up to follow a more promising one. This backtracking ability [30], while essential for achieving good error performance, is also the main cause of the

serious problems in sequential decoding. Since here, decoding is performed along only the forward direction, it shall be referred to as *unidirectional sequential decoding* (USD).

Since the stack algorithm is much simpler to describe and analyze than the Fano algorithm, we will focus on the stack algorithm in this dissertation. In this algorithm, a stack is used to store the already searched paths in decreasing order of their metric values. The top of the stack is the path with the largest accumulated metric among the paths in the stack, and will be extended one level further into the tree. The stack is reordered after each extension, so that the top always contains the current best or most likely path. Decoding stops whenever the top path reaches the end of the tree. For ease of reference, we refer to this algorithm as *unidirectional stack algorithm* and list it in the following.

Unidirectional Stack Algorithm:

Step 0: Place the initial root node of the code tree in the stack.

Step 1: Compute the metrics of all successors of the top node in the stack, and enter the new nodes in the stack, ordered with respect to their cumulative metrics.

Step 2: Remove from the stack the node whose successors were just inserted.

Step 3: If the top node in the stack is the final node of the tree, stop. Otherwise, return to step 1.

The major problem of the above algorithm is the reordering of the stack after each extension (*Step 1*) of the top path. To overcome this difficulty, Jelinek [16] proposed a quantized version of the algorithm in which the paths are placed at random into substacks according to their metric values. All paths whose metric values are within a certain range

are stored in a substack. That is a path of metric Γ is inserted into substack H if

$$H\Delta \leq \Gamma < (H+1)\Delta, \tag{2.5}$$

where Δ is the substack spacing in metric value. In this quantized stack, the search for the top path reduces to the search for the highest non-empty substack. The path that is to be extended is taken at random or usually in a first-in last-out mode. In our computer simulation, the first-in last-out mode was used which is suggested by Johannesson [31] to be near-optimal.

The decoding effort of USD is usually measured by the distribution of the number of computations per either decoded bit or decoded block. In this thesis, the number of computations per block is used. A computation is defined as one hypothesis of a node to be on the correct path.[3] The lower bound given by Jacobs and Berlekamp [30] together with the upper bounds given by Savage [32], Falconer [33] and Jelinek [34] have shown that the computational distribution of USD is essentially Pareto. Specifically, define C_Λ^{USD} as the total number of computations required to decode the first Λ branches of the code tree using sequential decoding. With the assumption of correct decoding, Jacobs and Berlekamp [30] showed that for all codes, any DMC and any USD algorithm, C_Λ^{USD} satisfies

$$P\left(C_\Lambda^{USD} > N\right) \geq N^{-\alpha} \exp\left\{-O\left(\sqrt{\ln N}\right)\right\}, \tag{2.6}$$

where $O\left(\sqrt{\ln N}\right)$ is an asymptotically unimportant term,[4] and the Paretian exponent α is defined by the following parametric equation

$$R = \hat{E}_0(\alpha)/\alpha. \tag{2.7}$$

[3] A computation in the stack algorithm is an extension of the top node, while in the Fano algorithm, it is usually defined to be a forward move [5].

[4] $O(b_i)$ represents that the value can be upper-bounded by Ab_i, where A is a finite constant for all indexes i.

The function $\hat{E}_0(\alpha)$ in (2.7) is the smallest concave function greater than or equal to the Gallager function $E_0(\alpha)$ [5]. Using the random coding technique over an ensemble of trellis codes [35], Savage, Falconer and Jelinek showed that for a DMC, the first-branch computations C_1 of sequential decoding is upper-bounded as

$$\overline{P(C_1 \geq N)} \leq AN^{-\rho}, \tag{2.8}$$

where $\rho < \rho_r$, which is defined as $R = E_0(\rho_r)/\rho_r$,[5] and A is a finite constant independent of N [36]. Moreover, it was noticed that asymptotically, the lower bound gives an accurate description of the computational distribution for good codes and practical sequential decoding algorithms [31].

The computational performance of USD with a specific convolutional code depends on the decoding algorithm employed and the distance properties of the forward code. In particular, optimum forward sequential decoding performance requires rapid initial column distance growth of the forward code for fast decoding and minimum erasure probability, and a large free distance (d_{free}) for minimum error probability [37, 38]. Specifically, for a code with rate R and a BSC with a crossover probability p satisfying

$$R < \ln 2 + p \ln \frac{p}{1-p} - H(p), \tag{2.9}$$

where $H(p) = -p \ln p - (1-p)\ln(1-p)$ is the entropy function, the distribution $P(C_1 \geq N)$ for a specific time-invariant convolutional code is upper-bounded by [38]

$$P(C_1 \geq N) < \sigma \cdot n_{d_c^F} \cdot \exp\{-\mu d_c^F [\log_u N] + \phi \log_u N\}, \tag{2.10}$$

where σ, μ, ϕ and $n_{d_c^F}$ are parameters which are functions of p and R, but independent of N, and are as defined in [38].

[5] $\rho_r = \alpha$ if the channel is a BSC, but $\rho_r \leq \alpha$ in general.

The *distance profile* $\boldsymbol{d}$ of a forward code is defined as the *column distance function* (CDF) over only the first constraint length (i.e., $\boldsymbol{d} \triangleq [d_0, d_1, \cdots, d_m]$) [8]. Since the distance profile determines the initial column distance growth of the forward code, and is easier to compute than the entire CDF, it is usually used instead of the entire CDF as a criterion for selecting codes for use with sequential decoding. A code is said to have a $\boldsymbol{d}$ superior to a $\boldsymbol{d}'$ of another forward code of the same memory order m, when there is some $i \leq m$ such that

$$d_j \begin{cases} = d'_j, & j = 0, 1, \cdots, i-1 \\ > d'_j, & j = i. \end{cases} \tag{2.11}$$

A forward code is said to have an *optimum distance profile* (ODP) [8] if its $\boldsymbol{d}$ is superior to that of any other forward code of the same memory order.

2.3 Backward Coding/Decoding

Because data is transmitted with a known (e.g. zero) tail in a block mode, all paths depart from and terminate at the same state S_0. A path in the trellis or tree diagram can thus be viewed in reverse as starting from the final state S_0 and terminating at the initial state S_0. Every such path also corresponds to an encoded sequence. When viewed as starting from the final state, the distinct paths in the tree or trellis define a code which we refer to as the *backward code*. From the tree (trellis) of the original forward code, one can generate the backward tree (trellis) for the backward code. Figures 2.4 and 2.5 show, respectively, the backward trellis and tree for the backward code obtained from the original forward code (3, 1, 2) of Figure 2.1. Note that a backward code sequence is obtained by simply reversing the original forward code sequence. In addition, the

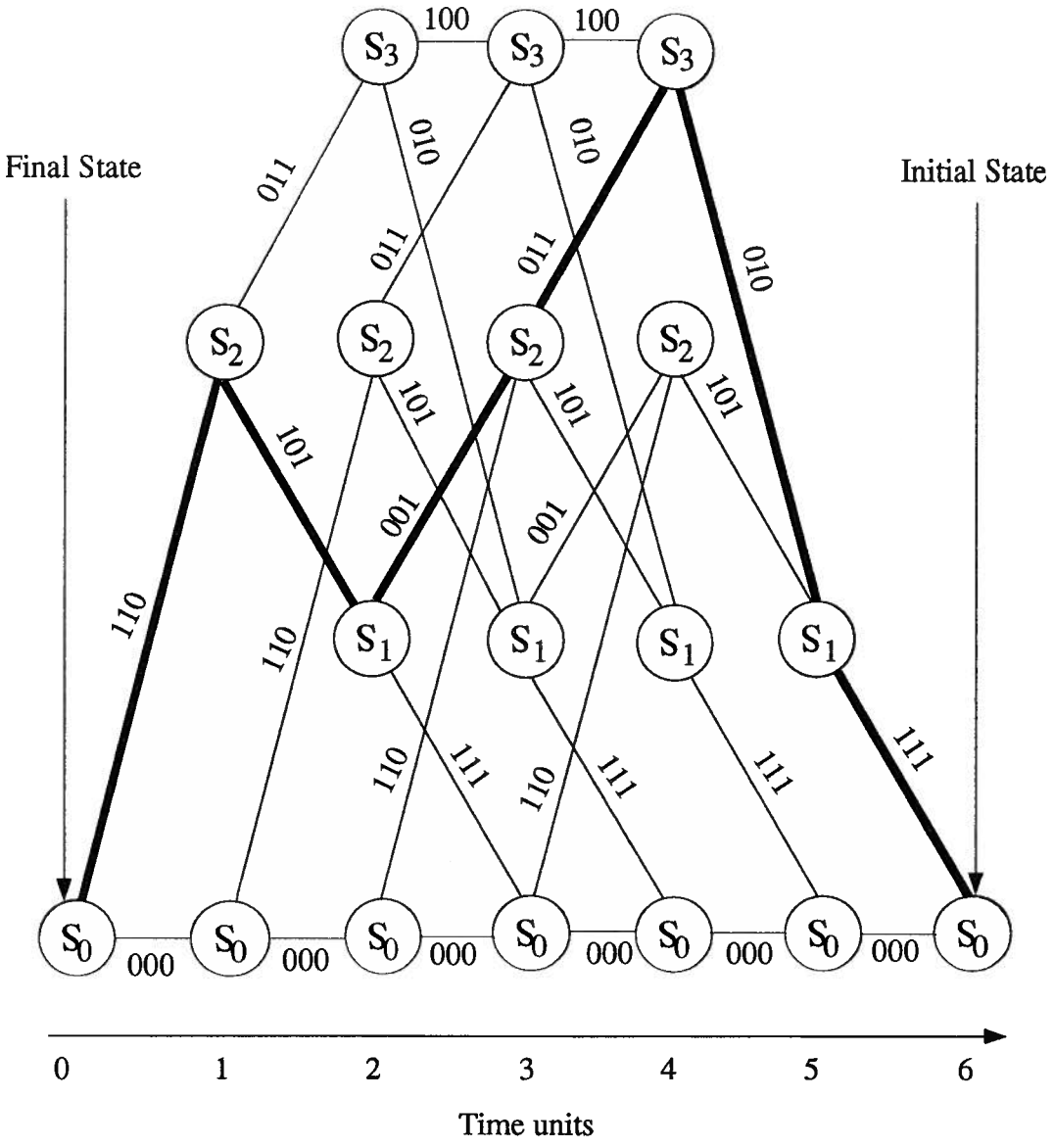


Figure 2.4 Backward code trellis diagram for the encoder of Figure 2.1.

corresponding information sequences are reversals of each other, as well. For example, the highlighted path on the

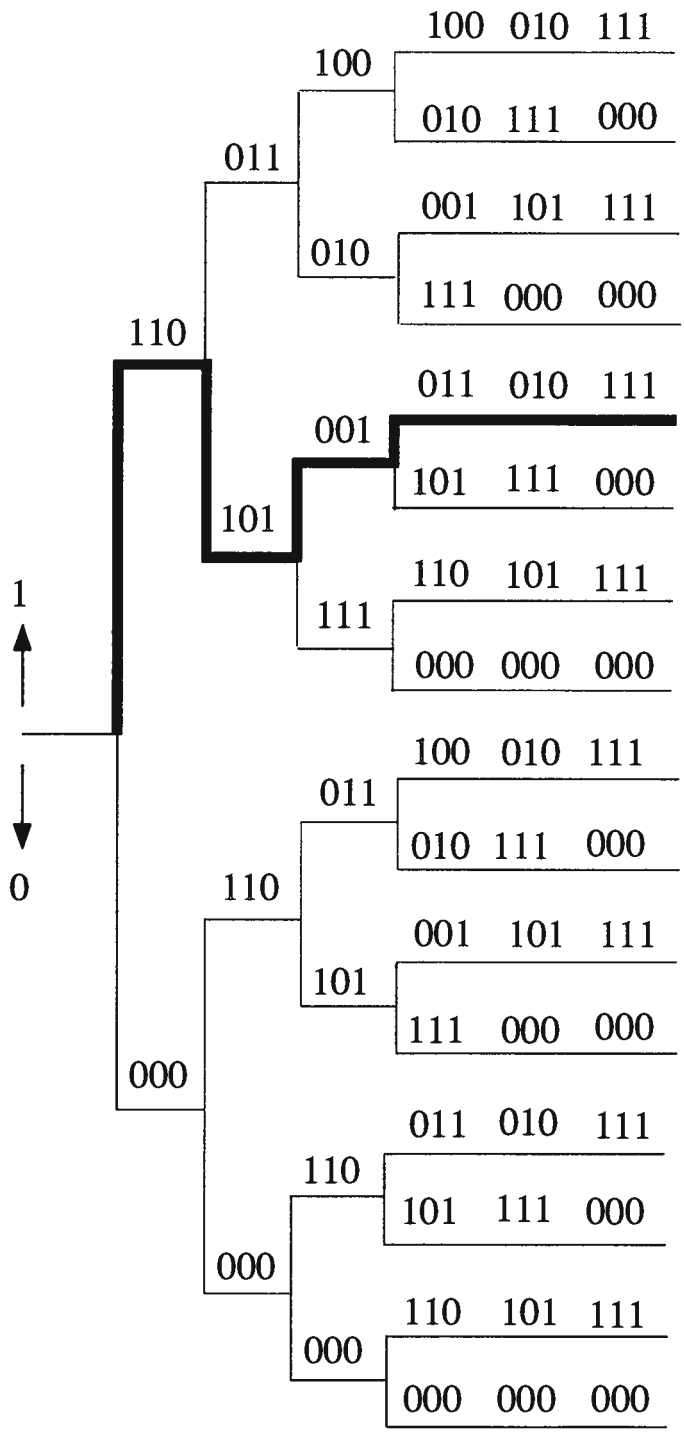


Figure 2.5 Backward code tree diagram of the encoder of Figure 2.1.

forward code tree of Figure 2.3 corresponds to the information sequence (1101) and the code sequence (111, 010, 110, 100, 101, 011). Their equivalent reversed information and code sequences are (1011) and (110, 101, 001, 011, 010, 111), respectively (see the highlighted path on the backward tree of Figure 2.5).

It can be readily shown that the transfer function matrix $\mathbf{G_r}(D)$ of an (n, k, m) backward convolutional code can be expressed in terms of the transfer function matrix of the forward code [39] by

$$\mathbf{G_r}(D) = D^m \begin{bmatrix} \mathbf{g}_1^{(n)}(D^{-1}) & \mathbf{g}_1^{(n-1)}(D^{-1}) & \cdots & \mathbf{g}_1^{(1)}(D^{-1}) \\ \mathbf{g}_2^{(n)}(D^{-1}) & \mathbf{g}_2^{(n-1)}(D^{-1}) & \cdots & \mathbf{g}_2^{(1)}(D^{-1}) \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{g}_k^{(n)}(D^{-1}) & \mathbf{g}_k^{(n-1)}(D^{-1}) & \cdots & \mathbf{g}_k^{(1)}(D^{-1}) \end{bmatrix}. \tag{2.12}$$

As an example, consider the encoder of Figure 2.1. The transfer function matrix of its corresponding backward encoder is $\mathbf{G_r}(D) = \left[1 + D + D^2, 1 + D^2, D + D^2\right]$.

It appears from the above discussion that, given a convolutional encoder, one can perform backward encoding by feeding the information bits to be encoded from the right side of the registers, and set the registers to shift left.

A forward (n, k, m) convolutional code is said to be *invertible* if there exists an $n \times k$ matrix $\mathbf{G}^{-1}(D)$ such that

$$\mathbf{G}(D)\mathbf{G}^{-1}(D) = \mathbf{I}D^{\delta} \tag{2.13}$$

for some $\delta \geq 0$, where $\mathbf{I}$ is the $k \times k$ identity matrix [8]. For an (n, 1, m) convolutional code, this means that there exists a feedforward inverse $\mathbf{G}^{-1}(D)$ of delay δ ($\delta \geq 0$) if and only if [40]

$$\mathbf{GCD}\left[\mathbf{g}^{(1)}(D), \mathbf{g}^{(2)}(D), \cdots, \mathbf{g}^{(n)}(D)\right] = D^{\delta}, \tag{2.14}$$

where **GCD** denotes the greatest common divisor. For an (*n, k, m*) convolutional code with $k > 1$, let $\Delta_i(D), i = 1, 2, \cdots, \binom{n}{k}$, be the determinants of the $\binom{n}{k}$ distinct $k \times k$ submatrixes of the transfer function matrix $\mathbf{G}(D)$. Then a feedforward inverse $\mathbf{G}^{-1}(D)$ of delay δ exists if and only if

$$\mathbf{GCD}\left[\Delta_i(D) : i = 1, 2, \cdots, \binom{n}{k}\right] = D^{\delta} \tag{2.15}$$

for some $\delta \geq 0$ [40]. Also, (2.14) or (2.15) is a necessary and sufficient condition for a code to be *noncatastrophic*. We now show that a backward convolutional code is invertible provided that the forward code is invertible.

Lemma 2.1: For an invertible (n, 1, m) convolutional code, the delay δ in (2.14) is always no greater than m, since $\deg\left[\mathbf{g}^{(i)}(D)\right] \leq m$ *for all i.*

Lemma 2.2: For an invertible (n, k, m) convolutional code with $k > 1$, the delay δ in (2.15) is always no greater than km, since $\deg[\mathbf{\Delta}_i(D)] \leq km$ *for all i.*

Theorem 2.1: A backward code corresponding to an invertible (n, 1, m) convolutional code with delay δ is invertible and its delay is equal to $(m - \delta)$.

Proof : From (2.14), we can write

$$\mathbf{GCD}\left[\mathbf{g}^{(n)}(D^{-1}), \mathbf{g}^{(n-1)}(D^{-1}), \cdots, \mathbf{g}^{(1)}(D^{-1})\right] = D^{-\delta}, \tag{2.16}$$

and

$$\mathbf{GCD}\left[D^m\mathbf{g}^{(n)}(D^{-1}), \cdots, D^m\mathbf{g}^{(1)}(D^{-1})\right] = D^{m-\delta}. \tag{2.17}$$

From (2.12), we can write (2.17) as

$$\mathbf{GCD}\left[\mathbf{g}_r^{(1)}(D), \mathbf{g}_r^{(2)}(D), \cdots, \mathbf{g}_r^{(n)}(D)\right] = D^{m-\delta}. \tag{2.18}$$

From Lemma 2.1, we know that $m - \delta \geq 0$. Q.E.D.

Theorem 2.2: A backward code corresponding to an invertible (n, k, m) convolutional code (k > 1) with delay δ is also invertible and its delay is equal to (km – δ).

Proof: According to (2.12), we can write

$$\mathbf{GCD}\left[\Delta_{\mathbf{r}_i}(D) : i = 1, 2, \cdots, \binom{n}{k}\right] = \mathbf{GCD}\left[D^{km}\Delta_i(D^{-1}) : i = 1, 2, \cdots, \binom{n}{k}\right]. \tag{2.19}$$

From (2.15), we can write

$$\mathbf{GCD}\left[\Delta_i(D^{-1}) : i = 1, 2, \cdots, \binom{n}{k}\right] = D^{-\delta}. \tag{2.20}$$

Therefore, from (2.19) and (2.20) we can write

$$\mathbf{GCD}\left[\Delta_{\mathbf{r}_i}(D) : i = 1, 2, \cdots, \binom{n}{k}\right] = D^{km-\delta}, \tag{2.21}$$

and from Lemma 2.2, we know $km - \delta \geq 0$. Q.E.D.

According to the above discussion we can write the following.

Corollary 2.1: A backward convolutional code is noncatastrophic if and only if its forward code is noncatastrophic.

Corollary 2.2: Decoding of a forward invertible convolutional code can be performed using the corresponding backward code. This will be called *backward decoding* as opposed to the usual forward decoding.

Property 2.1: The distance spectrum [8] of a backward convolutional code is identical to that of the corresponding forward code. This is because the Hamming weight of any path on the forward code tree or trellis is the same as the corresponding path on the backward code tree or trellis. As a consequence of this property, when maximum

likelihood decoding is used, the error performance of a backward code is the same as that of the corresponding forward code.

Property 2.2: The distance profile of a backward code is usually different from that of the corresponding forward code. This is rather obvious and can be seen from the forward and backward code trees of Figures 2.3 and 2.5. For this example, the first coefficient of the distance profile of the forward code is equal to 3; whereas for its corresponding backward code, it is equal to 2. As a consequence of this property, backward sequential decoding is in general not as efficient as its corresponding forward sequential decoding when a forward ODP code is used.

With our bidirectional sequential algorithms to be discussed later, decoding is performed in both forward and backward directions. It is thus desirable to use codes that possess the same good distance profile in both directions (see Theorem 4.1 in Chapter 4). A sufficient condition for a forward code to have the same distance profile as its corresponding backward code is symmetry. A code is said to be *symmetric* or *reversible* [41, 42] if the code generators of the forward code are the same as those of the corresponding backward code. For example, the code with generators $g^{(1)}(D) = 1+D+D^2$ and $g^{(2)}(D) = 1+D^2$ is symmetric. Moreover, since this code has an optimum distance profile [43] we call it a symmetric bidirectional optimum distance profile code. A code is said to have a *symmetric bidirectional optimum distance profile* (SBODP) if it is symmetric and its distance profile is optimum. Unfortunately, most known ODP codes do not possess the property of symmetry. However, one can search for new symmetric codes that have a good distance profile.

2.4 Search for Good Symmetric Codes

Using a computer search procedure [44, 45], we have found good symmetric (*2*, *1*, *m*) non-systematic codes with *m* ranging from *2* to *25*. In our computer search, we first attempt to find a symmetric code with the same distance profile as the corresponding known ODP code. If more than one code is found, we select the one with the highest d_{free} and the minimal number of paths of weight d_{free}. Of course, we considered only noncatastrophic codes. Results are given in Table 2.1. Also included in the table are the free distances of known ODP codes. Distance spectra of the found SBODP codes are listed in Table 2.2. In Table 2.2, $a_{d_{free}+i}$ denotes the number of paths of weight $d_{free}+i$, while $c_{d_{free}+i}$ denotes the total number of nonzero information bits on all these paths. From Table 2.1, one can immediately notice that SBODP codes do not exist for some values of *m*, which are marked by dash signs. Moreover, as it can be seen from Table 2.1, to find a SBODP code, usually d_{free} had to be sacrificed. In the following, some new codes which slightly compromise the distance profile for a better d_{free} are presented.

Table 2.1 Symmetric bidirectional optimum distance profile (SBODP) codes[6]

m	$G^{(1)}$	$G^{(2)}$	d_{free}	d_{free} of ODP
2	7	5	5	5
3	—	—	—	6
4	23	31	6	7
5	—	—	—	8
6	127	165	8	10
7	—	—	—	10
8	—	—	—	12
9	1157	1731	12	12
10	2251	3003	9	14
11	5247	7125	12	14
12	—	—	—	15
13	27537	37275	12	16
14	56777	77735	12	17
15	134277	176435	16	18
16	222511	304443	14	19
17	475417	741571	16	20
18	1346477	1762635	18	21
19	—	—	—	22
20	5734323	6261675	18	22
21	12022373	15744405	20	24
22	31065423	23340731	21	24
23	44407043	61070111	18	26
24	176153077	134442235	24	26
25	221446737	373544611	24	27

[6] The $m = 2$ SBODP code is the known ODP code found by Johannesson [43] and is listed here for completeness.

Table 2.2 Distance spectra of SBODP codes

m	$G^{(1)}$	$G^{(2)}$	d_{free}	$(a_{d_{free}+i}, i = 0, 1, 2, \cdots)$ $[c_{d_{free}+i}, i = 0, 1, 2, \cdots]$
4	23	31	6	(1,0,4,0,22,0,124,0,682,0) [1,0,10,0,96,0,778,0,5616,0]
6	127	165	8	(1,0,5,0,35,0,187,0,1074,0) [2,0,15,0,188,0,1275,0,9350,0]
9	1157	1731	12	(8,0,16,0,159,0,741,0,5027,0) [32,0,78,0,1150,0,6390,0,52470,0]
10	2251	3003	9	(1,0,1,1,,1,18,21,27,110,256) [1,0,5,4,9,88,123,186,796,2060]
11	5247	7125	12	(3,0,10,0,33,0,216,0,1216,0) [10,0,42,0,228,0,1702,0,11248,0]
13	27537	37275	12	(1,0,2,0,11,0,54,0376,0) [2,0,6,0,68,0,352,0,3278,0]
14	56777	77735	12	(2,0,0,0,10,0,16,0,208,0) [4,0,0,0,46,0118,01420,0]
15	134277	176435	16	(13,0,0,0,166,0,96,0,3651,0) [60,0,0,0,1330,0,734,0,37790,0]
16	222511	304443	14	(2,0,3,0,7,0,59,0,237,0) [5,0,13,0,36,0,455,0,1966,0]
17	475417	741571	16	(2,0,1,0,33,0,97,0,751,0) [6,0,7,0,220,0,753,0,7386,0]
18	1346477	1762635	18	(1,0,12,0,51,0,302,0,1801,0) [5,0,70,0,365,0,2720,0,18953,0]
20	5734323	6261675	18	(1,0,1,0,12,0,89,0,474,0) [5,0,6,0,92,0,782,0,5034,0]
21	12022373	15744405	20	(3,0,8,0,36,0,196,0,1246,0) [24,0,56,0,288,0,1782,0,14312,0]
22	31065423	23340731	21	(1,5,7,18,30,74,175,341,863,2097) [5,36,37,168,298,670,1721,3686,9883,25366]
23	44407043	61070111	18	(1,0,2,0,4,0,19,0,54,0,) [1,0,10,0,20,0,116,0,406,0]
24	176153077	134442235	24	(2,3,26,42,93,173) [16,13,234,350,884,1769]
25	221446737	373544611	24	(4,0,17,0,78,0,508,0,2780,0) [26,0,135,0,756,0,5540,0,34644,0]

In the event that no SBODP code can be found, we then slightly decrease the distance profile to find a good code (large d_{free} and good distance profile). The same method can be applied when the d_{free} of a SBODP code is inferior to that of the known ODP code. We shall refer to this symmetric non-ODP code as an *almost BODP* (SABODP) code. The computer search results are listed in Table 2.3. The dash sign in Table 2.3 indicates that no code with a better d_{free} is found after slightly decreasing the distance profile. Distance spectra of the found SABODP codes are listed in Table 2.4. The search for SABODP codes is conducted as follows. We first examine all codes obtained through possible modifications of the last four coefficients of the distance profile of the corresponding ODP code. Then, among these codes, we select the one with the highest d_{free} and the best distance profile. The last four coefficients of the distance profile of the obtained codes are given in the last column of Table 2.3. When the distance profile is slightly decreased, a code with a better d_{free} may be found. For example, d_{free} of the SBODP code of memory $m = 23$ is equal to 18. When only the last coefficient of the distance profile is decreased by one, a SABODP code with $d_{free} = 24$ is found.

Notice that for any $(2, 1, m)$ systematic code to be symmetric, the second adder of the encoder should be connected to only the last stage of the shift register. This would result in a very poor code in terms of both the free distance and the distance profile.

Table 2.3 Symmetric almost bidirectional optimum distance profile (SABODP) codes

m	$G^{(1)}$	$G^{(2)}$	$d_{m-3}, \ldots, d_m$	d_{free}	d_{free} of ODP
2	—	—	—	—	5
3	13	15	2,3,3,3	6	6
4	—	—	—	—	7
5	57	75	3,4,4,4	8	8
6	—	—	—	—	10
7	247	345	4,5,5,5	10	10
8	507	705	5,5,5,6	10	12
9	—	—	—	—	12
10	2617	3615	6,6,6,6	12	14
11	—	—	—	—	14
12	16767	12345	6,7,7,7	14	15
13	24433	33045	6,7,7,7	14	16
14	47263	63271	7,7,8,8	16	17
15	—	—	—	—	18
16	232357	367131	8,8,8,8	16	19
17	723705	507627	8,8,8,8	18	20
18	1167671	1712517	8,8,9,9	19	21
19	2714477	3744635	9,9,9,9	20	22
20	5267447	7117325	9,9,9,9	20	22
21	17242231	11444257	9,9,9,10	22	24
22	32427561	21675053	9,10,10,10	22	24
23	55231643	61346255	10,10,10,10	24	26
24	—	—	—	—	26
25	—	—	—	—	27

Table 2.4 Distance spectra of SABODP codes

m	$G^{(1)}$	$G^{(2)}$	d_{free}	$(a_{d_{free}+i}, i = 0, 1, 2, \cdots)$ $[c_{d_{free}+i}, i = 0, 1, 2, \cdots)]$
3	13	15	6	(2,0,10,0,49,0,241,0,1185,0) [4,0,38,0,277,0,1806,0,11063,0]
5	57	75	8	(3,0,12,0,70,0,397,0,2223,0) [8,0,46,0,400,0,2925,0,20446,0]
7	247	345	10	(4,0,19,0,95,0,539,0,3111,0) [10,0,86,0,649,0,4400,0,30287,0]
8	507	705	10	(2,0,10,0,51,0,287,0,1560,0) [4,0,36,0,289,0,2118,0,13984,0]
10	2617	3615	12	(2,0,15,0,71,0,383,0,2333,0) [6,0,65,0,494,0,3053,0,22486,0]
12	16767	12345	14	(2,0,20,0,93,0,612,0,3283,0) [6,0,111,0,701,0,5456,0,35447,0]
13	24433	33045	14	(3,0,10,0,47,0,280,0,1687,0) [11,0,52,0,323,0,2478,0,17675,0]
14	47263	63271	16	(6,0,27,0,140,0,808,0,4745,0) [34,0,151,0,1120,0,7520,0,52014,0]
16	232357	367131	16	(1,0,2,0,38,0,239,0,1131,0) [10,0,6,0,246,0,1917,0,11114,0]
17	723705	507627	18	(1,0,29,0,119,0,578,0,3433,0) [3,0,182,0,953,0,5816,0,39833,0]
18	1167671	1712517	19	(4,7,1,25,74,138,353,953,2090,5351) [20,58,3,182,612,1242,3369,10278,23840,66680]
19	2714477	3744635	20	(5,0,45,0,160,0,851,0,5184,0) [24,0,319,0,1440,8775,0,61394,0]
20	5267447	7117325	20	(1,0,19,0,101,0,423,0,2492,0) [4,0,143,0,884,0,4295,0,29122,0]
21	17242231	11444257	22	(9,0,58,0,275,0,1290,0,7381,0) [49,0,488,0,2695,14502,0,94513,0]
22	32427561	21675053	22	(1,0,44,0,128,0,793,0,4094,0) [11,0,368,0,1078,0,8648,0,50150,0]
23	55231643	61346255	24	(14,0,60,0,348,0,1959,0,10600,0) [108,0,592,0,3602,0,24395,0,145826,0]

Chapter 3
Bidirectional Sequential Decoding Algorithms

In this chapter, BSD algorithms in which the code tree is explored from both forward and backward directions are presented. Operations of all the proposed algorithms are essentially the same, except for their stopping rules. In our BSD algorithms, two separate stacks are used. One is used for the forward search of the tree and is called the *forward stack* (FS), while the other is used for the backward search and is called the *backward stack* (BS). Starting from the root nodes of the forward and backward trees (trellises), respectively, forward and backward search operations are performed alternatively according to the usual stack algorithm. Define the total number of computations needed to decode one block by any BSD algorithm as the sum of the computations performed by forward and backward decoders.

The proposed BSD algorithms can be classified into two categories: BSD-no-merge and BSD-merge. In BSD-merge, decoding stops whenever a path in one stack (FS or BS) merges with a path in the other stack. A path in one stack is said to *merge* (or agree) with a path in the other stack if the two paths share the same encoder state at a common tree level. In the class of BSD-no-merge, forward and backward portions of the decoded path do not have to agree with each other.

In the following, Algorithm TAmeet which belongs to the class of BSD-no-merge is first presented and analyzed. Next, two algorithms, Algorithm TAmerge and Algorithm TTmerge, which belong to the class of BSD-merge, are presented and analyzed. Finally,

Algorithm HTTmerge, which is a hybrid scheme of BSD-merge and BSD-no-merge, is presented. For the convenience of describing the following algorithms, let $l_{F_{TOP}}$ denote the level of the top node in the FS, and l_F the farthest level[7] reached by the forward decoder. Similarly, let $l_{B_{TOP}}$ denote the level of the top node in the BS, and l_B the farthest level reached by the backward decoder.

3.1 Algorithm TAmeet, a BSD-no-merge

Algorithm TAmeet stops decoding whenever a path in one of the two stacks meets a path in the opposite direction, and these two meeting paths do not have to agree with each other. The level of the forward tree at the meeting point is denoted by l. As a consequence, the portion of the tree explored by forward decoder does not overlap with the portion explored by backward decoder, as illustrated by Figure 3.1. Algorithm TAmeet is as follow.

Algorithm TAmeet:

Step 0: Place the root nodes of forward and backward trees in the FS and BS, respectively.

Step 1: Compute the metrics of all successors of the top node in the FS, and enter the new nodes in the FS, ordered with respect to their cumulative metrics.

Step 2: Remove from the FS the node whose successors were just inserted.

Step 3: Check if $l_F + l_{B_{TOP}} = L$. If yes, stop decoding and go to step 7. Otherwise, go to the next step.

---

[7] The farthest level is not necessarily equal to the level of the top node because the decoder allows backtracking.

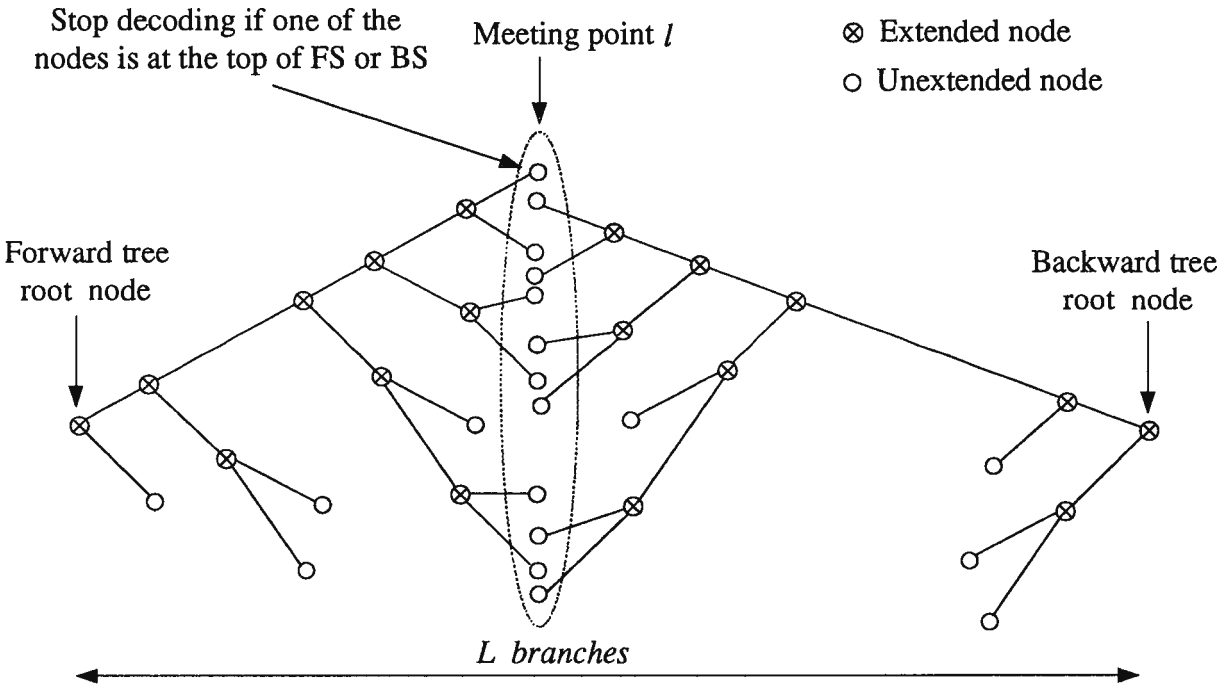


Figure 3.1 Illustration of the stopping rules of Algorithm TAmeet.

Step 4: Compute the metrics of all successors of the top node in the BS, and enter the new nodes in the BS, ordered with respect to their cumulative metrics.

Step 5: Remove from the BS the node whose successors were just inserted.

Step 6: Check if $l_{F_{TOP}} + l_B = L$. If yes, stop decoding and go to step 10. Otherwise, return to step 1.

Step 7: The top path[8] in the BS is accepted as the backward portion of the decoded path. The forward portion of the decoded path is selected from the FS, whose end node is at level l_F, (select the one with the highest metric value if there is more than one path at level l_F). If $\min(\lfloor m/2 \rfloor, l_{B_{TOP}}) = l_{B_{TOP}}$, retreat $m - l_{B_{TOP}}$ branches from the end node of the forward portion of the decoded path, then take the information

[8] Defined as the path with the top node as its end node.

symbols corresponding to the rest of the path as the decoded symbols, and go to step 13. Otherwise go to the next step.

Step 8: If $\min(\lceil m/2 \rceil, l_F) = l_F$, retreat $m - l_F$ branches from the end node of the backward portion of the decoded path, then take the information symbols corresponding to the rest of the path as the decoded symbols, and go to step 13. Otherwise go to the next step.

Step 9: Retreat the last $\lceil m/2 \rceil$ and $\lfloor m/2 \rfloor$ branches from the forward and backward portions of the decoded path, respectively. Take the information symbols corresponding to the rest of the two paths as the decoded symbols, and go to step 13.

Step 10: The top path in the FS is accepted as the forward portion of the decoded path. The backward portion of the decoded path is selected from the BS, whose end node is at level l_B (select the one with the highest metric value if there is more than one path at level l_B). If $\min(\lfloor m/2 \rfloor, l_{F_{TOP}}) = l_{F_{TOP}}$, retreat $m - l_{F_{TOP}}$ branches from the end node of the backward portion of the decoded path, then take the information symbols corresponding to the rest of the path as the decoded symbols, and go to step 13. Otherwise go to the next step.

Step 11: If $\min(\lceil m/2 \rceil, l_B) = l_B$, retreat $m - l_B$ branches from the end node of the forward portion of the decoded path, then take the information symbols corresponding to the rest of the path as the decoded symbols, and go to step 13. Otherwise go to the next step.

Step 12: Retreat the last $\lfloor m/2 \rfloor$ and $\lceil m/2 \rceil$ branches from the forward and backward portions of the decoded path, respectively. Take the information symbols corresponding

to the rest of the two paths as the decoded symbols.

Step 13: Output the decoded information symbols.

When the channel is not too noisy, the forward and backward portions of the decoded path may actually merge (agree) with each other since a sequential decoder always follows the most likely path. Should this not be the case, a valid path (not necessarily the correct one) can be assembled by retreating m branches from forward and backward portions of the decoded path, as described above.

Note that in the absence of a burst in a block, the meeting point l would most of the time fall near the middle of the tree. However, should there be a single burst error within a block, it is expected that the meeting point would occur near the middle of the burst. Thus by retreating about $m/2$ branches from forward and backward meeting paths, the burst error may be totally eliminated if its length is less than m.

The meeting point l at the end of decoding in Algorithm TAmeet is equal to $l_{F_{TOP}}$ if decoding is stopped at step 6, or $(L - l_{B_{TOP}})$ if decoding is stopped at step 3. Note that forward and backward decoders always meet somewhere in the tree, i.e. $1 \leq l \leq L-1$. Let x be an arbitrary level in the forward code tree, and imagine cutting the tree into two separate subtrees at this level. Let C_x^F denote the number of computations needed by the forward decoder to decode x branches without searching beyond level x of the forward tree, i.e., when the top node in the FS reaches depth x for the first time. This is equivalent to decoding an x branch-tree without a terminating tail. Similarly, let C_{L-x}^B denote the number of computations needed by the backward decoder to decode an $(L-x)$ branch-tree without searching beyond level $(L-x)$ of the backward tree. Clearly, for any

DMC and any fixed arbitrary level x, random variables C_x^F and C_{L-x}^B are independent because the forward and backward search areas are disjoint.

Let us assume that the amount of time to hypothesize (extend) one node is the same for both the forward and the backward decoders. Let C_L denote the total number of computations needed to decode one block by Algorithm TAmeet. By the nature of Algorithm TAmeet, we can write the following property.

Property 3.1: Let l be the meeting point, then

$$C_L = \begin{cases} 2C_l^F \text{ and } C_{L-l}^B \geq C_l^F, & \text{if decoding is stopped at step 6} \\ 2C_{L-l}^B + 1 \text{ and } C_l^F \geq C_{L-l}^B + 1, & \text{if decoding is stopped at step 3} \end{cases}. \quad (3.1)$$

Proof: According to Algorithm TAmeet, if decoding is stopped at step 6, the forward decoder must have extended the same number of nodes as the backward decoder. If decoding is stopped at step 3, however, the forward decoder must have extended one more node than the backward decoder. Let us first assume that decoding is stopped at step 6. By the definition of the meeting point l (see Figure 3.1), we know that $l_{F_{TOP}} = l$, which means $C_L = 2C_l^F$. Moreover, since $l_B = L - l \geq l_{B_{TOP}}$, then $C_{L-l}^B \geq C_l^F$. Now, suppose that decoding is stopped at step 3. In this case, $C_L = 2C_{L-l}^B + 1$ since $l_{B_{TOP}} = L-l$, where the additional computation is due to the last extension by the forward decoder. Moreover, since $l_F = l \geq l_{F_{TOP}}$, then $C_l^F \geq C_{L-l}^B + 1$. Q.E.D.

Corollary 3.1: Let l be the meeting point, then

$$C_L - 1 \leq 2\min\left(C_l^F, C_{L-l}^B\right), \quad (3.2)$$

since $C_L - 1 \leq 2C_l^F$ and $C_L - 1 \leq 2C_{L-l}^B$ according to Property 3.1.

Obviously, $C_L = 2C_l^F = 2C_{L-l}^B$ if and only if both top nodes in the FS and BS are at forward tree level l at the end of decoding, i.e., both top nodes are at the meeting point l. It is worth noticing that the minus one in (3.2) can actually be ignored and thus C_L is essentially equal to $2\min\left(C_l^F, C_{L-l}^B\right)$.

Now, we introduce an important property which is the foundation in our analysis of the computational performance of the proposed BSD in Chapter 4.

Property 3.2: Let integer $i \in [1, L-1]$, then

$$C_L - 1 \leq 2 \max_i \left[\min(C_i^F, C_{L-i}^B)\right]. \tag{3.3}$$

Proof: First of all, note that there must exist a meeting point in Algorithm TAmeet, i.e., $1 \leq l \leq L-1$. By Corollary 3.1, one can write

$$C_L - 1 \leq 2\min\left(C_l^F, C_{L-l}^B\right). \tag{3.4}$$

Moreover, we can write

$$\max_i \left[\min(C_i^F, C_{L-i}^B)\right] \geq \min(C_l^F, C_{L-l}^B), \tag{3.5}$$

since $1 \leq l \leq L-1$. Q.E.D.

In fact, it is easy to see that C_L is virtually equal to $2\max_i \left[\min\left(C_i^F, C_{L-i}^B\right)\right]$.[9] This approximation, however, is not necessary for the proof of the upper bounds on the computational distribution of Algorithm TAmeet in Chapter 4.

It is interesting to point out that Forney's scheme [22] belongs to BSD-no-merge since forward and backward decoders do not need to agree with each other, and its

[9] Note that $C_L \geq \min\left(C_i^F, C_{L-i}^B\right)$ for all possible i.

total number of computations is equal to $2\min(C_L^F, C_L^B)$, where C_L^F and C_L^B denote the number of computations needed by forward and backward decoders to decode the whole block, respectively.

Let us now compare Algorithm TAmeet with the bidirectional *M* algorithm [28]. It is obvious that they both belong to the same category of BSD-no-merge since merging between the forward and backward portions of the decoded path is not required in both algorithms. The main difference, however, is in the nature of the search algorithm employed in the forward and backward decoders. In the bidirectional *M* algorithm the number of computations per decoded branch is constant, whereas in Algorithm TAmeet that number is variable since backtracking is allowed. As a consequence, erasure free decoding is possible in the bidirectional *M* algorithm, while Algorithm TAmeet can not guarantee it. On the other hand, the bidirectional *M* algorithm suffers the correct path loss problem. Another difference is how they handle a burst. In Algorithm TAmeet, forward and backward decoders search into forward and backward trees alternatively no matter how many or where the bursts are located. At the end of decoding, both decoders in Algorithm TAmeet retreat about $m / 2$ branches. In the bidirectional *M* algorithm, however, in order for the first decoder encountering a burst to wait for the other decoder's help from the opposite direction, a burst detection method is required. Moreover, only the bad decoder retreats m branches at the end of decoding. The BER of the bidirectional *M* algorithm degrades with increasing block length L [28], while in Algorithm TAmeet, the BER improves as L increases (see Chapter 5 for analysis and simulation results).

3.2 Algorithm TAmerge, a BSD-merge

In BSD-merge, decoding does not stop whenever two opposing paths meet each other as in the algorithm described above. Instead, a merging test is applied to opposing paths and decoding stops only when the test is successful. We present in the following Algorithm TAmerge which belongs to the class of BSD-merge.

Algorithm TAmerge works in the same way as Algorithm TAmeet, except that at the meeting point l decoding is not stopped, unless a pair of forward and backward paths merge with each other. Instead, decoding is continued until a pair of merged forward and backward paths is found. Let l_0 denote the level of the forward tree when this happens. Obviously, here the portion of the tree searched by forward decoder does overlap with that portion searched by backward decoder, as illustrated by Figure 3.2. More specifically, Algorithm TAmerge is as follow.

Algorithm TAmerge:

Steps 0 to 2 are exactly the same as in Algorithm TAmeet.

Step 3: If $l_{B_{TOP}}+l_F \geq L$, check the top path in the BS with all paths with depths equal to $L - l_{B_{TOP}}$ in the FS. If one (or more) merging path is found, stop decoding. Among all merging paths, select the one with the highest cumulative metric as the decoded path. Otherwise, go to the next step.

Steps 4 and 5 are exactly the same as in Algorithm TAmeet.

Step 6: If $l_{F_{TOP}}+l_B \geq L$, check the top path in the FS with all paths with depths equal to $L - l_{F_{TOP}}$ in the BS. If one (or more) merging path is found, stop decoding. Among

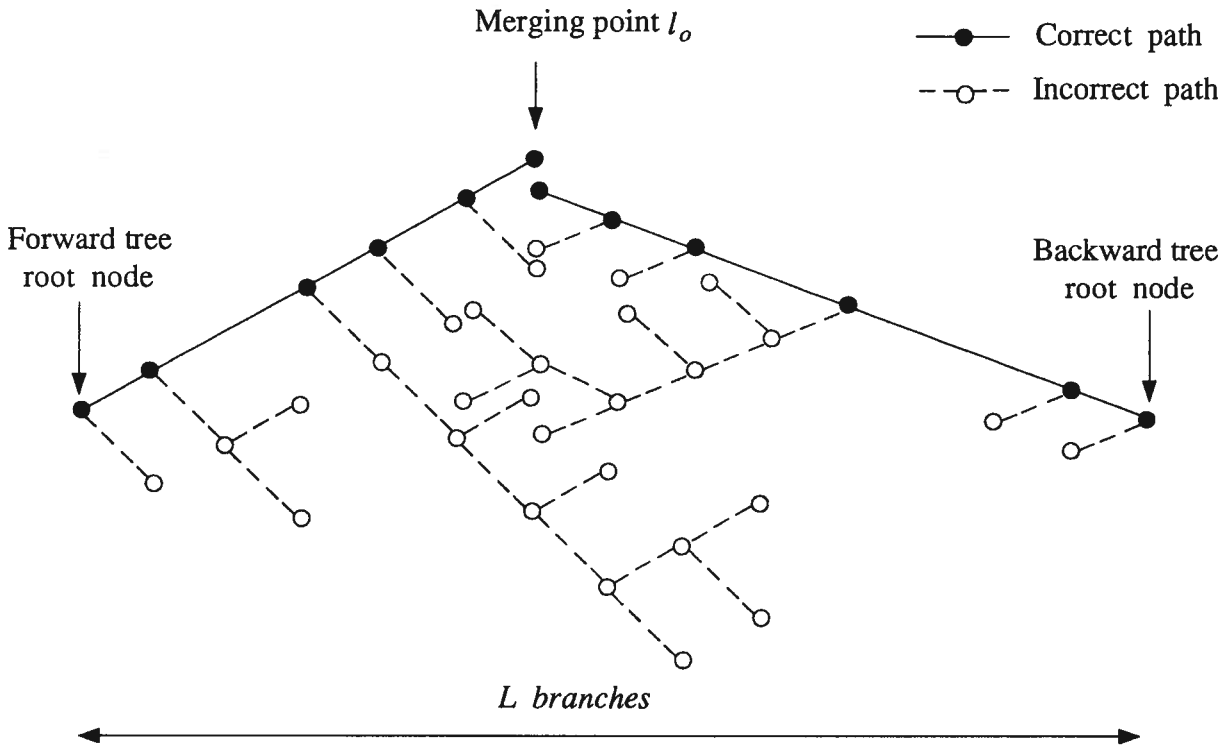


Figure 3.2 Illustration of the overlapping in BSD-merge.

all merging paths, select the one with the highest cumulative metric as the decoded path. Otherwise, go to step 1.

It is worth pointing out that all nodes in one stack (FS or BS) can be easily linked with respect to their depths. Hence, in the merging test (steps 3 and 6), it is easy to find all those nodes which should be tested with the top node in the opposite direction. During the merging test at steps 3 and 6, the top path in either the FS or BS is only tested with end nodes of paths in the opposite direction, which are at the same level with the tested top path. A natural question that may arise is whether two opposing paths may pass each other or not since intermediate nodes are not being examined during the merging test. For example, Figure 3.3 illustrates such a situation where two opposing

paths may have common encoder states at some of their intermediate nodes. We now show that the events shown in this figure can not take place in Algorithm TAmerge.

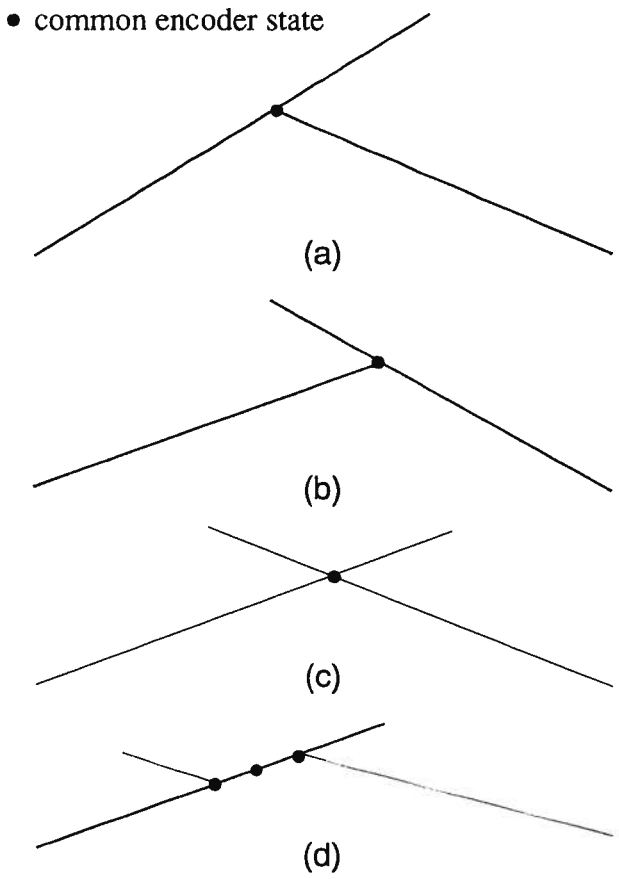


Figure 3.3 Example of opposing paths passing each other.

First of all, events like (c) and (d) in Figure 3.3 can not happen unless event (a) or (b) occurs in the first place. This is because both forward and backward decoders must reach their predecessors before any further extensions. We are going to show that events (a) and (b) in Figure 3.3 can not occur in algorithm TAmerge. Clearly, events (a) and (b) are the same type of event. Hence, only event (a), which is shown in Figure 3.4 for more detail, will be considered in the following.

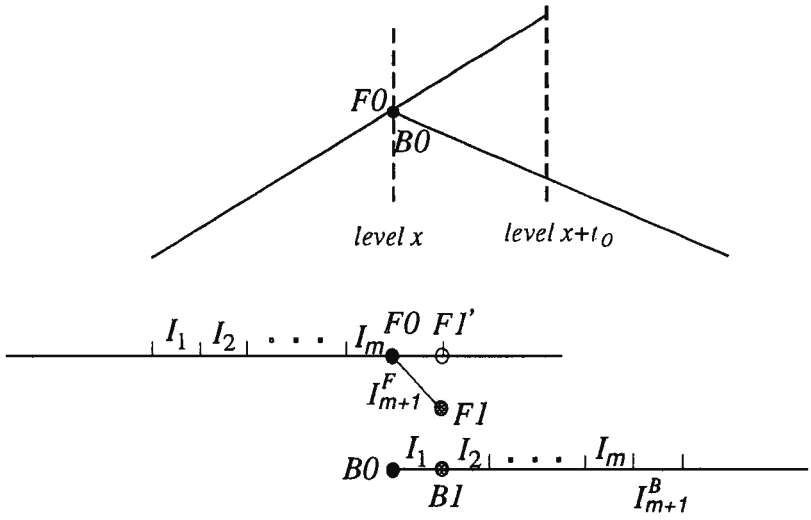
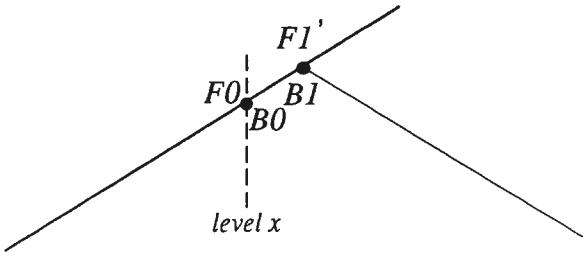


Figure 3.4 Typical impossible event in Algorithm TAmerge.

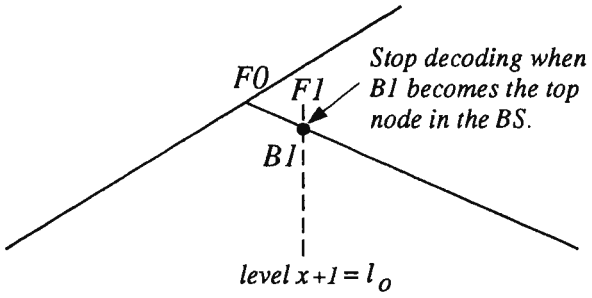
Figure 3.4 implies that the backward path with end node *B0* merges with the forward path with its intermediate node *F0*. Moreover, *F0* and *B0* are the only common nodes between the forward and backward paths. As Figure 3.4 indicates, $(I_1, I_2, \cdots, I_m)$ denotes the common encoder state. Moreover, the encoder state of node *B1* is either $(I_2, I_3, \cdots, I_m, 0)$ or $(I_2, I_3, \cdots, I_m, 1)$. Let I^B_{m+1} denote the last bit of the encoder state *B1*. Similarly, let I^F_{m+1} denote the last bit of the encoder state *F1*. Since *F0* is the intermediate state of the forward path, all successors of node *F0* must have been reached, and *F0* is consequently removed from the FS (steps 1 and 2 in Algorithm TAmerge). Let *F1* and *F1'* be the successors of *F0* as illustrated in Figure 3.4. Since *F0* and *B0* are the only common nodes between forward and backward paths, *F1* and *B1* are common nodes, i.e. $I^F_{m+1} = I^B_{m+1}$. Otherwise, nodes *F1'* and *B1* will share a common encoder state, as indicated in Figure 3.5.

Now, let us assume that *F1* is still in the FS, which means *F1* has not been extended



• common encoder state

Figure 3.5 Illustration of a typical impossible event in Figure 3.4.



• common encoder state

Figure 3.6 Illustration of the possible event.

yet before the backward path reached node *B0*. According to step 3 in Algorithm TAmerge, however, decoding must have been stopped at the time when *B1* is at the top of the BS. Thus, *B0* does not have any chance to be reached because *B1* must be at the top of the BS before *B0* can be reached. This is illustrated in Figure 3.6, which clearly shows that end node *F1* of the forward path merges with end node *B1* of the backward path.

Next, let us assume that *F1* is not in the FS, which means *F1* has already been extended. As shown in Figure 3.7, let *F2* and *F2'* denote the successors of *F1*, and assume that *F2* and *B2* have the same encoder state. Clearly, Figure 3.7 shows the same

type of event as that in Figure 3.3 (d) since there is more than one common encoder state. Based on our discussion above, this event can not occur before an event like that in Figure 3.3 (a) or (b) happens. Following this same line of reasoning, we get the following property.

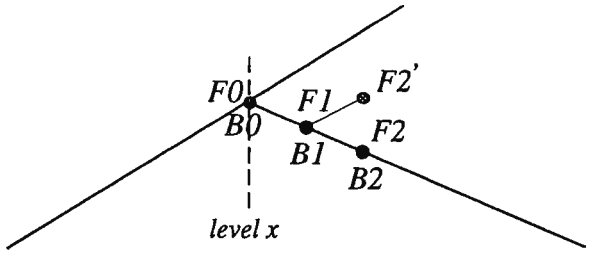


Figure 3.7 Illustration of another impossible event.

Property 3.3: In Algorithm TAmerge, forward and backward paths in the FS and BS can only merge at their end nodes. That is merging paths can not overlap each other.

Since the decoded path is a merged path, forward and backward portions of the decoded path will never overlap in Algorithm TAmerge. Furthermore, we can conclude that overlapped forward and backward paths in Algorithm TAmerge do not have the same encoder state at any of their common tree levels because otherwise Property 3.3 would be breached.

Two forward and backward paths are said to be totally *distinct* as long as they do not overlap or do not share the same encoder state at any common tree level in their overlapping portion. Figure 3.8 illustrates all possible types of paths in the FS and in the BS. In Figure 3.8 (a), forward and backward paths do not have a chance to meet.

In Figure 3.8 (b), merged forward and backward paths do not overlap, while in Figure 3.8 (c) forward and backward paths do not share any common encoder state in their overlapping area. Thus, we can write the following property.

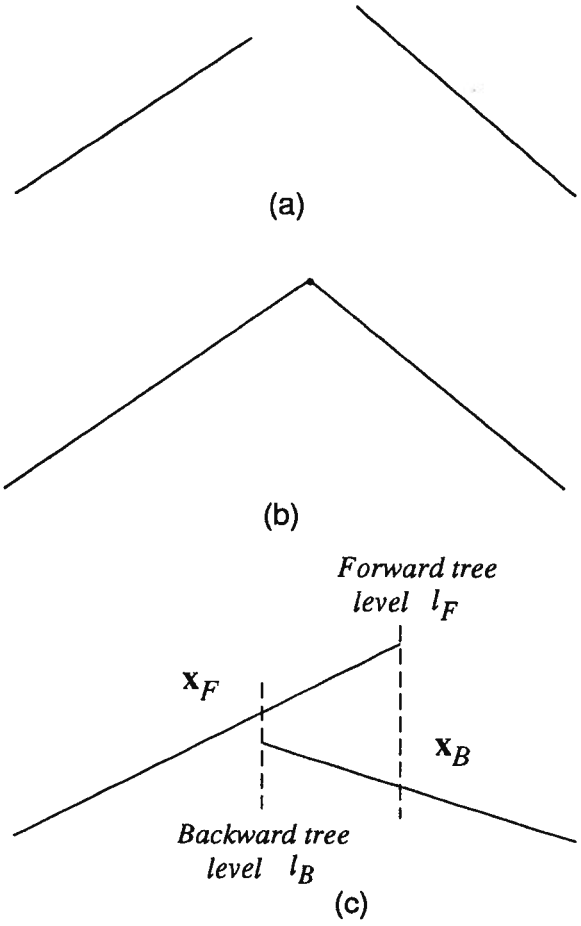


Figure 3.8 Distinct forward and backward paths in the FS and BS.

Property 3.4: In Algorithm TAmerge, all paths in the FS are totally distinct from all paths in the BS.

Like in Algorithm TAmeet, forward and backward decoders cannot reach their terminal nodes in Algorithm TAmerge because all successors of the root nodes of forward

and backward trees were reached at the beginning. Thus, forward and backward decoders must merge somewhere in the tree, i.e., $1 \leq l_o \leq L-1$. In the following analysis of Algorithm TAmerge, correct decoding is assumed, which is a good approximation, especially for long memory codes. For an arbitrary forward tree level x, let us define $\hat{C}_x^F$ as the number of computations required to correctly decode the first x branches by forward decoder. Similarly, define $\hat{C}_{L-x}^B$ as the number of computations required to correctly decode the last *(L-x)* branches by backward decoder. Unlike Algorithm TAmeet, forward decoder may extend incorrect paths beyond level x before it finally decodes all x branches. Backward decoder may also extend incorrect paths beyond level *(L-x)* before it finally decodes all *(L-x)* branches.

Now, let $\hat{C}_L$ denote the total number of computations needed by Algorithm TAmerge with the assumption of correct decoding. By analogy to Algorithm TAmeet, we can claim the following properties.

Property 3.5: Let l_o be the merging point of Algorithm TAmerge, then

$$\hat{C}_L = \begin{cases} 2\hat{C}_{l_o}^F \text{ and } \hat{C}_{L-l_o}^B \geq \hat{C}_{l_o}^F, & \text{if decoding is stopped at step 6} \\ 2\hat{C}_{L-l_o}^B + 1 \text{ and } \hat{C}_{l_o}^F \geq \hat{C}_{L-l_o}^B + 1, & \text{if decoding is stopped at step 3} \end{cases}. \quad (3.6)$$

Proof: Similar to Algorithm TAmeet, if decoding is stopped at step 6, forward decoder must have extended the same number of nodes as backward decoder. If decoding is stopped at step 3, however, forward decoder must have extended one more node than backward decoder. Without loss of generality, let us assume that decoding is stopped by step 6. Thus, the top node at the FS is at level l_o when decoding is stopped, and the forward decoded path up to this top node is correct. Hence, the forward

decoder must have performed exactly $\hat{C}_{l_o}^F$ computations. Furthermore, the algorithm does not require that the backward decoder totally decodes the last $(L - l_0)$ branches, i.e., $\hat{C}_{L-l_o}^B \geq \hat{C}_{l_o}^F$. Q.E.D.

Corollary 3.2: Let l_o be the merging point in Algorithm TAmerge, then

$$\hat{C}_L - 1 \leq 2\min\left(\hat{C}_{l_o}^F, \hat{C}_{L-l_o}^B\right), \tag{3.7}$$

according to Property 3.5. Similar to the discussion of Corollary 3.1, $\hat{C}_L$ is almost equal to $2\min\left(\hat{C}_{l_o}^F, \hat{C}_{L-l_o}^B\right)$.

Property 3.6: Let integer $i \in [1, L-1]$, then

$$\hat{C}_L - 1 \leq 2\max_i\left[\min(\hat{C}_i^F, \hat{C}_{L-i}^B)\right]. \tag{3.8}$$

Proof: Same as the proof of Property 3.2. Q.E.D.

Similar to Algorithm TAmeet, one can easily conclude that $\hat{C}_L$ is practically equal to $2\max_i\left[\min\left(\hat{C}_i^F, \hat{C}_{L-i}^B\right)\right]$. Strictly speaking, the number of computations without the assumption of correct decoding is less than or equal to that with the assumption.

We now examine the merging test (step 3 and step 6) of Algorithm TAmerge in more detail. In Algorithm TAmerge, decoding stops whenever a top path either in the FS or in the BS merges with any path from the opposite direction. Clearly, the number of paths being tested in the merging test is a random variable. This variable is small under good channel conditions and increases as the channel gets noisier. Suppose that the top path in the BS merges with a forward path *FP* which is not at the top of the FS (step 3). The end node of *FP* is at the merging point l_o according to Property 3.3. As a consequence of

this merging, forward decoder does not need to extend all forward paths whose metrics are higher than $M[FP]$ to decode the forward portion of the decoded path FP. In other words, the merging test at step 3 effectively reduces the number of computations needed by forward decoder to decode FP. Similarly, the merging test at step 6 may reduce the number of computations needed by backward decoder if the top path in the FS merges with an opposite path that is not at the top of the BS. Hence, the multiple examinations of the top path in one stack with all paths in the opposite stack results in a reduction in computational variability since the overall number of computations per decoded block is reduced. The parallel to this is the multiple path extensions in the generalized stack algorithms of Haccoun and Ferguson [19], which allows a reduction in computational variability at a cost of an increase in average number of computations per decoded block. Note that in Algorithm TAmerge, the average number of computations is not increased, but is even reduced, as it will be shown in Chapter 4.

3.3 Coarsening on Algorithm TAmerge

In order to efficiently perform the merging test in Algorithm TAmerge, nodes have to be linked together with respect to their depths. However, this introduces the drawback of extra processing time and memory needed for storage in forward and backward decoders. In the following, we provide two modified algorithms, quite similar to Algorithm TAmerge but without the aforementioned drawback. These two algorithms work best with quantized stacks.

The modification is done in the merging test of Algorithm TAmerge (steps 3 and 6). Here, the merging test is only performed among all nodes in the highest non-empty

substacks in the FS and BS instead of testing the top node in each stack with all possible merging nodes in the opposite direction. As a consequence, however, two opposing paths may pass each other because forward and backward portions of the merged path may not coexist in the highest non-empty substacks at the same time. In other words, forward and backward portions of the merged path may overlap, as illustrated in Figure 3.3. In order to reduce the possibility of two opposite paths passing each other, not only the endpoints of the two possible merging paths should be tested but also other possible merging points in the overlapping area as will be described later.

Coarsening 1 (Algorithm TTmerge):

Steps 0 to 2 are the same as in Algorithm TAmeet, except the top node is now any node in the non-empty highest substack.

Step 3: If $l_F + l_B \geq L$, check all paths in the non-empty highest substack in the BS with all paths in the non-empty highest substack in the FS. If there is an overlap between forward and backward paths that are being tested, check all encoder states in the overlapping area. If one or more merging paths is found, stop decoding. Among all merging paths, select the one with the highest cumulative metric as the decoded path. Otherwise, go to the next step.

Steps 4 and 5 are the same as in Algorithm TAmeet, except the top node is now any node in the non-empty highest substack.

Step 6: If $l_F + l_B \geq L$, check all paths in the non-empty highest substack in the FS with all paths in the non-empty highest substack in the BS. If there is an overlap between the forward and the backward paths being tested, check all encoder states in

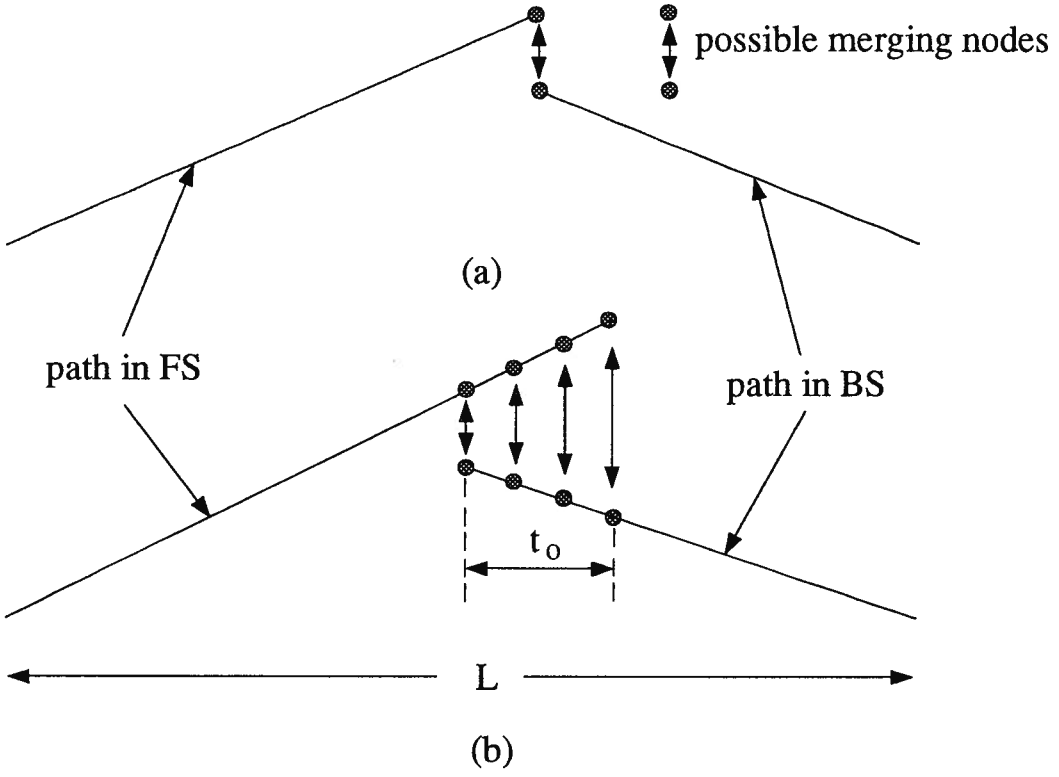


Figure 3.9 Merging test in Algorithm TTmerge.

the overlapping area. If one or more merging paths is found, stop decoding. Among all merging paths, select the one with the highest cumulative metric as the decoded path. Otherwise, go to the next step.

Step 7: Check if the node to be extended in the highest non-empty substack in the FS is a terminal node in the forward tree. Check if the node to be extended in the highest non-empty substack in the BS is a terminal node in the backward tree. If either one is a terminal node, stop decoding and that top path is accepted as the decoded path. In the event that both top paths in the FS and BS are terminal nodes in the forward and backward trees, respectively, the path with the highest cumulative metric is selected as the decoded path. Otherwise, go to step 1.

If two opposing paths pass each other and one of them reaches the end of its corresponding tree, clearly the decoding should be stopped. Therefore, step 7 is needed in Algorithm TTmerge. However, step 7 can be eliminated if we treat the forward and backward root nodes as connected with zero paths. For example, if the top node in the FS is at level L, it can be viewed as a merging node with the backward root node.

The merging test in Algorithm TTmerge is illustrated in Figure 3.8. In Figure 3.8 (a), the forward and backward paths do not overlap and thus only the end points are tested. However in Figure 3.8 (b), forward and backward paths do overlap over t_0 branches. In this case all intermediate nodes are examined. This involves t_0+1 comparisons which can be done by simply performing one exclusive-OR operation between the last $m+t_0$ information symbols of the tested paths, and checking if there exits m consecutive zeros. In case the depth of one of the tested paths is smaller than $(m + t_0)$, the shorter one can be viewed as one rooted by a zero path of any length.

It is interesting to notice that the number of nodes in the highest forward and backward substacks varies according to the channel conditions. When the channel is quiet, only the correct node is likely to float in the highest substack. As the channel condition degrades, the number of nodes in the highest substacks tends to increase. This means that the merging test involves a variable number of examinations, which depends on the channel conditions.

Coarsening 2 (Algorithm HTTmerge):

We now describe Algorithm HTTmerge which is a hybrid version of Algorithm TTmerge and Algorithm TAmeet. Algorithm HTTmerge is exactly the same as Algorithm

TTmerge except that the number of information symbols required to be the same in the merging test of Algorithm TTmerge (steps 3 and 6), denoted m_h, is less than m. The idea is that if two opposing paths agree on their last $m_h < m$ branches, it is anticipated that they would eventually agree on all m last branches if decoding is continued. Of course this may not happen all the time, especially if m_h is relatively small, and this premature stopping may introduce additional decoding errors. If $m_h=m$, Algorithm HTTmerge becomes equivalent to Algorithm TTmerge. In the other extreme, if $m_h=0$, it becomes equivalent to Algorithm TAmeet. Clearly, Algorithm HTTmerge reduces the amount of computations by compromising the error performance since the number of matching information symbols m_h is less than m. Hence, Algorithm HTTmerge can provide a trade-off between computational effort and error performance by controlling the number of matching information symbols in the merging test.

Chapter 4
Computational Performance of BSD

In this chapter, we analyze the computational performance of the proposed BSD algorithms. It is assumed that the channel is a DMC. Also, BSD using the stack algorithm is assumed. However, all results in this chapter can be applied to arbitrary BSD sequential decoders. Let C_L^{BSD} denote the total number of computations performed by any BSD algorithm for decoding one block of L branches (including a known tail), where one computation is defined as in Chapter 3, i.e., the extension of one node from either forward or backward direction. We are mainly concerned with the distribution of C_L^{BSD}, i.e., $P(C_L^{BSD} > N)$. In the following, we first quickly demonstrate the property of the computational distribution by a simple approach. Then, Algorithm TAmeet is analyzed in detail. More specifically, upper bounds on the computational distribution are given by Theorem 4.1 (for a specific time-invariant convolutional code) and by Theorem 4.2 (for an ensemble of trellis codes). Moreover, we conjecture that Theorem 4.2 can be applied to Algorithm TAmerge. A lower bound on the computational distribution for any code and any type of BSD is also derived. The average number of computations and the computational cutoff rate of BSD are then discussed. Finally, extensive computer simulation results are provided, confirming our theoretical analysis.

4.1 A Simple Approach to Computational Distribution

Before giving the general derivation, we first quickly show interesting results on the computational distribution of BSD when the code rate is below the computational

cutoff rate R_{comp}. Let us define a dip [36, 46] on the correct path as the largest metric difference between two consecutive non-adjacent breakout nodes (see Figure 4.1 (a)). Notice that the dip is almost the same whether decoding is performed from forward or backward direction because the metric of the backward correct path is the mirror image of its corresponding metric of the forward correct path (see Figure 4.1 (a) and (b)). Alternatively, the dip can be viewed as a "noise burst" [30]. It is admitted that the Pareto distribution $P\left(C_L^{USD} > N\right) \simeq N^{-\rho_r}$ is caused by the following two opposing effects: (i) the probability of the dip size (burst length) γ decreases exponentially with γ; (ii) the number of computations necessary to decode the correct path increases exponentially with γ [5, 15, 30, 32, 36, 46, 47]. All these suggest the inherent inability of USD to deal effectively with a large dip (long burst). These difficulties arise from the essential nature of USD algorithms, i.e., searching only from one direction, and not from any fundamental limitations on convolutional codes or from any basic difficulties arising from the dip (burst) [30]. In the following, we demonstrate that the proposed BSD relieves the inefficiency of USD by attacking the dip (burst) form both forward and backward directions.

Suppose that the code rate is below R_{comp}, experimental evidence [48] indicates that with a very high probability only one important dip exists in a block. Moreover, it is easy to see that C_x^F is an increasing function of x, while C_{L-x}^B is a decreasing function of x. Hence, according to Property 3.2, $C_l^F \approx C_{L-l}^B$ in Algorithm TAmeet, where l is the meeting point. Similarly, in Algorithm TAmerge, $\hat{C}_{l_o}^F \approx \hat{C}_{L-l_o}^B$ where l_o is the merging point. Thus, Algorithm TAmeet and Algorithm TAmerge optimally break every received block into two portions that require roughly the same number of computations

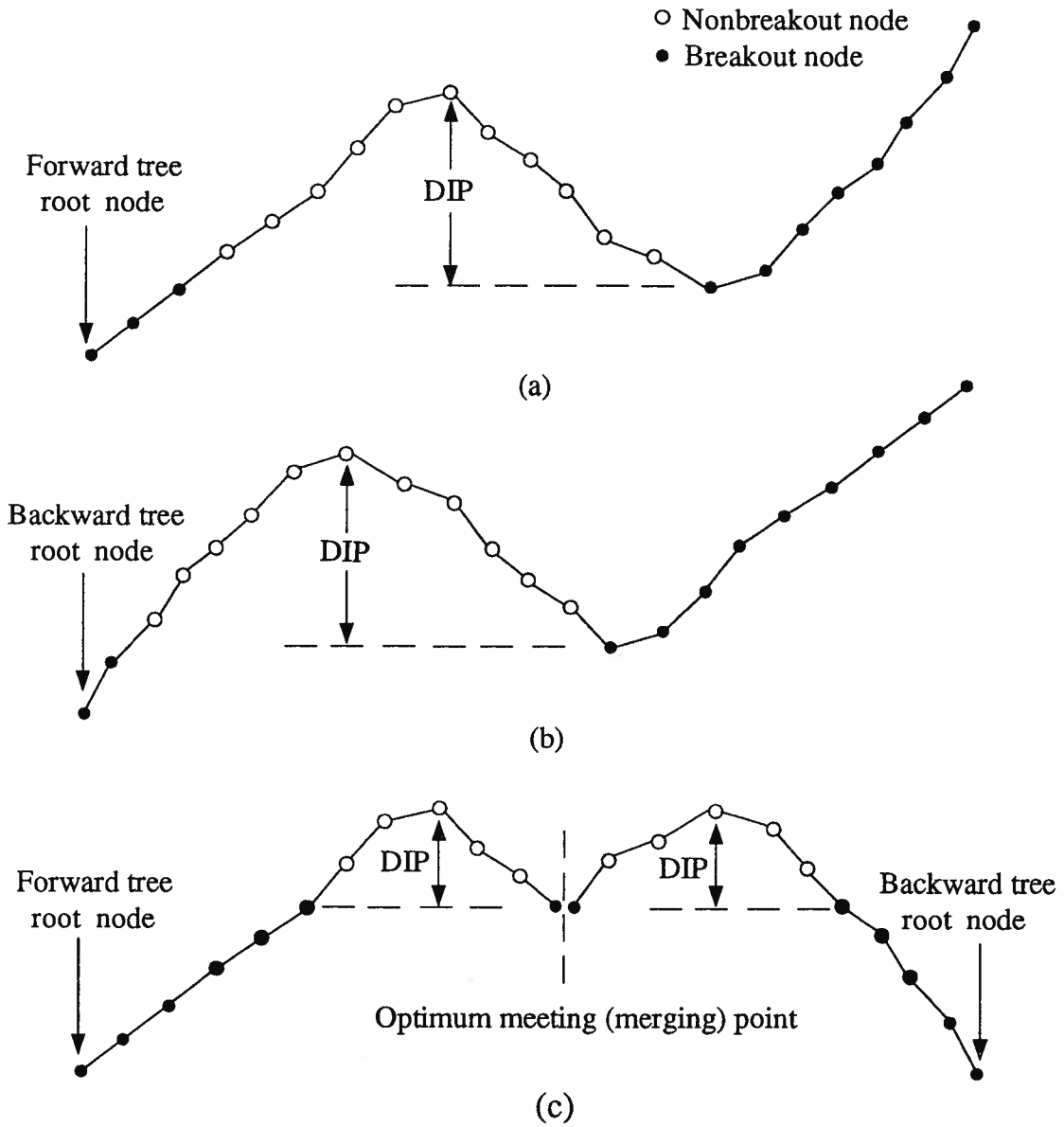


Figure 4.1 (a) Correct path metric with forward decoding. (b) Correct path metric with backward decoding. (c) Correct path metric with bidirectional decoding.

for decoding. Also, portions of the decoded forward and backward paths in the FS and BS will never overlap in Algorithm TAmeet and Algorithm TAmerge.

In summary, the proposed BSD algorithms attack the dip (burst) from both sides, and halt somewhere near the middle of the dip. Hence, with the assumption of correct

decoding, the meeting (or merging) event can be viewed as the breaking of the dip in an almost optimal way, i.e., into two almost equal parts (see Figure 4.1 (c)). It seems plausible that the required amount of computations by the proposed BSD is approximately the amount required by a USD to move through two dips (bursts) of half the original size. This is why our BSD needs substantially less computations. Therefore, if USD needs $N \propto \exp \gamma$ computations for any given correct path dip γ, our BSD asymptotically needs about twice of $\exp \frac{\gamma}{2} \propto \sqrt{N}$ computations for the same dip. If this characterization is valid, we can conclude that

$$P\left(C_L^{BSD} > N\right) \simeq P(C_L^{USD} > (\tfrac{N}{2})^2) \simeq 2^{2\rho_r} \cdot N^{-2\rho_r}. \tag{4.1}$$

Note the factor two in the exponent of the distribution (4.1).

4.2 Upper Bounds to Computational Distribution of Algorithm TAmeet

The computational effort to decode a given block depends on the strategy of the algorithm, the transmitted and the received sequences. For the convenience of analysis, define a random variable C as $C \triangleq 2 \max_i \left[\min\left(C_i^F, C_{L-i}^B\right)\right]$. We first derive upper bounds to the distribution function of the random variable C, then we will apply them to C_L of Algorithm TAmeet.

Lemma 4.1:

$$P(C \geq N) \leq \sum_x P\left(C_x^F \geq N/2\right) P\left(C_{L-x}^B \geq N/2\right), \tag{4.2}$$

where the summation is over all possible values of x, i.e., $1 \leq x \leq L-1$.

Proof: By the definition of the random variable C, one can immediately write

$$P(C \geq N) \triangleq P\Big(\max_x\{\min[C_x^F, C_{L-x}^B]\} \geq N/2\Big)$$
$$\leq \sum_x P\Big(\min[C_x^F, C_{L-x}^B] \geq N/2\Big), \tag{4.3}$$

where the summation is over all possible values of x, i.e., $1 \leq x \leq L-1$. The inequality in (4.3) is obvious since the summation includes the maximization point. Because random variables C_x^F and C_{L-x}^B are independent for any fixed arbitrary x, one can write (4.2) [49]. Q.E.D.

Before we continue the derivation of the upper bound, we need the following lemma.

Lemma 4.2: The distribution $P\big(C_\Lambda^{USD} \geq N\big)$ of USD to decode Λ branches in a finite Λ branch code tree (or infinite code tree) is upper bounded by

$$P\Big(C_\Lambda^{USD} \geq N\Big) \leq \Lambda P(C_1 \geq \frac{N}{\Lambda} - 1), \tag{4.4}$$

where C_1 denotes the number of computations performed by USD in order to decode the first branch in an infinite code tree.

Proof: For an infinite code tree, according to the union bound, we have [20]

$$P\Big(C_{\Lambda_I}^{USD} \geq N\Big) \leq P\left(\bigcup_{t=1}^{\Lambda} C_t \geq \frac{N-\Lambda}{\Lambda}\right)$$
$$\leq \sum_{t=1}^{\Lambda} P\left(C_t \geq \frac{N-\Lambda}{\Lambda}\right)$$
$$= \Lambda P\left(C_1 \geq \frac{N}{\Lambda} - 1\right), \tag{4.5}$$

where $C_{\Lambda_I}^{USD}$ denotes the number of computations required to decode the first Λ information branches in an infinite tree, and C_t is the number of computations by USD in

order to decode the t-th branch. For an infinite code tree, $C_t = C_1$ since the number of computations performed by USD to decode one branch is the same for all branches. For a finite code tree, we have

$$\begin{aligned} P\left(C_{\Lambda}^{USD} \geq N\right) &\leq P\left(C_{\Lambda_I}^{USD} \geq N\right) \\ &\leq \Lambda P\left(C_1 \geq \frac{N}{\Lambda} - 1\right) \end{aligned} \tag{4.6}$$

because $C_{\Lambda}^{USD} \leq C_{\Lambda_I}^{USD}$. Q.E.D.

Lemma 4.2 confirms that there exists no essential difference between $P\left(C_{\Lambda}^{USD} \geq N\right)$ and $P(C_1 \geq N)$ as Jacobs and Berlekamp suggested in [30]. From now on, we let C_1^F and C_1^B denote the number of computations required to decode the first branch in the infinite forward and backward code trees, respectively.

Theorem 4.1: For a specific time-invariant convolutional (i.e., linear trellis) code $\mathfrak{C}$ with rate R and a BSC with a crossover probability p satisfying (2.9), the computational distribution of Algorithm TAmeet is upper bounded by

$$\begin{aligned} P(C_L > N) < & A_1 \cdot n_{d_c^F} \cdot n_{d_c^B} \cdot \exp\{-\mu(d_c^F[\log_u N - \log_u (uL)] \\ & + d_c^B[\log_u N - \log_u (uL)]) + 2\phi \log_u N\} \end{aligned} \tag{4.7}$$

where A_I is a finite constant independent of N. Here μ, ϕ, $d_c^F[x]$, $d_c^B[x]$, $n_{d_c^F}$ and $n_{d_c^B}$ are the same as defined by Chevillat and Costello [38]. The superscripts F and B denote the corresponding forward and backward codes of $\mathfrak{C}$, respectively.

Proof: Using Chevillat and Costello's upper bound, which is given by (2.10) for a specific time-invariant convolutional code [38], we can express C_1^F as

$$P\left(C_1^F \geq \frac{N}{2x} - 1\right) < \sigma \cdot n_{d_c^F} \cdot \exp\{-\mu d_c^F \left[\log_u \left(\frac{N}{2x} - 1\right)\right] + \phi \log_u \left(\frac{N}{2x} - 1\right)\}, \tag{4.8}$$

and C_1^B as

$$P\left(C_1^B \geq \frac{N}{2(L-x)} - 1\right) < \sigma \cdot n_{d_c^B} \cdot \exp\{-\mu d_c^B \left[\log_u \left(\frac{N}{2(L-x)} - 1\right)\right] + \phi \log_u \left(\frac{N}{2(L-x)} - 1\right)\}. \quad (4.9)$$

Thus, according to Lemmas 4.1 and 4.2, we can write

$$P(C \geq N) < \sigma^2 \sum_x x \cdot (L-x) \cdot n_{d_c^F} \cdot n_{d_c^B} \cdot \exp\{-\mu(d_c^F \left[\log_u \left(\frac{N}{2x} - 1\right)\right] + d_c^B \left[\log_u \left(\frac{N}{2(L-x)} - 1\right)\right]) + \phi \log_u \left[\left(\frac{N}{2x} - 1\right)\left(\frac{N}{2(L-x)} - 1\right)\right]\}. \quad (4.10)$$

Notice that the CDF is a non-decreasing function and $\log_u (a-1) \geq \log_u (a) - 1$ when $a \geq 2$ and $u \geq 2$. We can thus write

$$\begin{aligned} P(C \geq N) &< \sigma^2 \sum_x x \cdot (L-x) \cdot n_{d_c^F} \cdot n_{d_c^B} \cdot \exp\{-\mu(d_c^F \left[\log_u \left(\frac{N}{2x}\right) - 1\right] \\ &\quad + d_c^B \left[\log_u \left(\frac{N}{2(L-x)}\right) - 1\right]) + \phi \log_u \left[\frac{1}{4}\left(\frac{N}{x}\right)\left(\frac{N}{L-x}\right)\right]\} \\ &< \sigma^2 \cdot 2^{-\frac{2\phi}{\ln u}} \cdot \sum_x [x \cdot (L-x)]^{1-\frac{\phi}{\ln u}} \cdot n_{d_c^F} \cdot n_{d_c^B} \cdot \exp\{-\mu(d_c^F [\log_u N \\ &\quad - \log_u (2uL)] + d_c^B [\log_u N - \log_u (2uL)]) + 2\phi \log_u N\} \\ &= A_1 \exp\{-\mu(d_c^F [\log_u N - \log_u (2uL)] \\ &\quad + d_c^B [\log_u N - \log_u (2uL)]) + 2\phi \log_u N\}. \end{aligned} \quad (4.11)$$

According to Property 3.2, we know that $C_L - 1 \leq C$. Hence, $P(C_L > N) \leq P(C \geq N)$. Q.E.D.

Theorem 4.1 shows that rapid column-distance growth of both forward and backward codes are essential to make BSD efficient. Using a computer search procedure, we have provided in Chapter 2 SBODP and SABODP codes which have the same rapid column-distance growth rate from both forward and backward directions.

However, if the convolutional code used is optimum from one direction but is very bad from the other direction, one can expect that the distribution of computations using BSD falls close to that using USD from the optimum direction, as the above bound indicates. A systematic ODP code is one typical example. Therefore, systematic codes are in general not suitable for BSD algorithms.

Corollary 4.1: For a symmetric convolutional code (e.g., a SBODP or SABODP code), Theorem 4.1 can be written as

$$P(C_L > N) < A_1 \cdot (n_{d_c})^2 \cdot \exp\{2(-\mu d_c[\log_u N - \log_u(2uL)] + \phi \log_u N)\}, \quad (4.12)$$

since $d_c^F[x] = d_c^B[x] = d_c[x]$ *and* $n_{d_c^F} = n_{d_c^B} = n_{d_c}$. Notice the factor two in the exponent of the distribution (4.12).

Furthermore, the above bound (Theorem 4.1) can also be shown to be related to random-coding results. For an ensemble of trellis codes, $d_c^F[x] = d_c^B[x] = n \cdot x/2$, $n_{d_c^F} < N/2$ and $n_{d_c^B} < N/2$ [38]. Therefore, for an ensemble of trellis codes, Theorem 4.1 can be expressed as

$$\begin{aligned}\overline{P(C_L > N)} &< \sigma^2 \cdot 2^{-\frac{2\phi}{\ln u}} \sum_x [x \cdot (L-x)]^{1-\frac{\phi}{\ln u}} \cdot \left(\frac{N}{2}\right)^2 \cdot \exp\{-2(\mu n/2 - \phi)\log_u N \\ &\quad + \mu n \log_u(2uL)\} \\ &= A_1 \cdot (2uL)^{\frac{\mu n}{\ln u}} \cdot \left(\frac{N}{2}\right)^2 \cdot \exp\{-2(\mu n/2 - \phi)\log_u N\} \\ &= A_2 N^{-2\psi}, \end{aligned} \quad (4.13)$$

where $\psi = (\mu \cdot n - 2\phi)/(2\ln u) - 1$, which is the same as that of USD [38]. Moreover, A_2 and ψ are functions of the channel and code rate but independent of N.

The above arguments are more clearly demonstrated by the following theorem. Notice that Lemmas 4.1 and 4.2 can be applied to an ensemble of codes.

Theorem 4.2: Let ρ_r be defined by $R = E_0(\rho_r)/\rho_r$. On any DMC and for large N, the computational distribution of Algorithm TAmeet for an ensemble of trellis codes, $\overline{P(C_L > N)}$, is upper bounded by

$$\overline{P(C_L > N)} \leq A_3 N^{-2\rho}, \tag{4.14}$$

where $\rho < \rho_r$ is the Pareto exponent of USD and A_3 is a finite constant independent of N.

Proof: According to Lemma 4.1, 4.2 and (2.8), one obtains

$$\begin{aligned}\overline{P(C \geq N)} &\leq A^2 \sum_x x(L-x)\left(\frac{N}{2x} - 1\right)^{-\rho}\left[\frac{N}{2(L-x)} - 1\right]^{-\rho} \\ &= 2^{2\rho} A^2 N^{-2\rho} \sum_x [x(L-x)]^{1+\rho}\left(1 - \frac{2x}{N}\right)^{-\rho}\left(1 - 2\frac{L-x}{N}\right)^{-\rho}. \end{aligned} \tag{4.15}$$

In (4.15) for N sufficiently large such that $N > 2L$, the terms in the last two brackets are very small and can be bounded by a constant. Therefore,

$$\overline{P(C \geq N)} \leq A_3 N^{-2\rho}, \tag{4.16}$$

where

$$\begin{aligned} A_3 &= 2^{2\rho} A^2 \sum_x [x(L-x)]^{1+\rho}\left(1 - \frac{2x}{N}\right)^{-\rho}\left[1 - \frac{2(L-x)}{N}\right]^{-\rho} \\ &< 2^{2\rho} A^2 \sum_x [x(L-x)]^{1+\rho}\left(1 + \frac{2\rho x}{N}\right)\left[1 + \frac{2\rho(L-x)}{N}\right] \\ &< 2^{2\rho} A^2 (1 + \rho + \rho^2/4) \sum_x [x(L-x)]^{1+\rho}. \end{aligned} \tag{4.17}$$

From Property 3.2, we know that $C_L - 1 \leq C$. Hence, $\overline{P(C_L > N)} \leq \overline{P(C \geq N)}$. Q.E.D.

From our computer simulation evidence (see section 4.6), we suggest that Theorem 4.2 can be applied to good specific time-invariant convolutional codes. The reason

is as follows. Experimental results [48] have demonstrated that inequality (2.7) can actually be used as an approximate expression for the computational distribution with good specific time-invariant convolutional codes. Moreover, Anderson [50, 51] showed that good specific time-invariant convolutional codes follow the random code properties quite closely. In other words, good specific time-invariant convolutional codes have pseudorandom characters although they are not random.

4.3 Extension to Other BSD Algorithms

In this section, we conjecture that the same asymptotic random coding upper bound (Theorem 4.2) also applies to Algorithm TAmerge. Algorithm TAmerge is similar to Algorithm TAmeet except for the stopping rules. More specifically, Property 3.2 of Algorithm TAmeet and Property 3.6 of Algorithm TAmerge are basically the same. By analogy to the analysis of Algorithm TAmeet, we can write

$$P\left(C_L^{TAmerge} > N\right) \leq P\left(\hat{C}_L > N\right) \leq \sum_x P\left(min\left[\hat{C}_x^F, \hat{C}_{L-x}^B\right] \geq N/2\right), \qquad (4.18)$$

where $C_L^{TAmerge}$ is the true number of computations of Algorithm TAmerge, i.e. without the assumption of correct decoding. Hence, the only question is that for any fixed arbitrary x whether $\hat{C}_x^F$ and $\hat{C}_{L-x}^B$ are independent or not.

As defined in Chapter 3, for an arbitrary forward tree level x, random variables $\hat{C}_x^F$ and $\hat{C}_{L-x}^B$ are the number of computations required to correctly decode x and *(L-x)* branches by forward and backward decoders, respectively. Furthermore, unlike Algorithm TAmeet, both forward and backward decoders may extend incorrect paths beyond level x before the end of decoding. Thus, some incorrect nodes searched by forward decoder may also be visited by backward decoder.

Let $\mathfrak{D}_i^F$ be the forward incorrect path subset generated from forward tree level i. Similarly, let $\mathfrak{D}_j^B$ denote the backward incorrect path subset generated from backward tree level j. According to the operation of sequential decoding, $\hat{C}_x^F$ is a function of the metrics of the correct path up to level x and metrics of incorrect paths in the subsets $\mathfrak{D}_i^F$, i =1, 2, ..., x, [31, 46, 52]. Thus, one can write

$$\hat{C}_x^F = f(M[\mathbf{x}_F(h)], M[\tilde{\mathbf{x}}_F]),\ where\ \ h \in [1, x]\ \ and\ \ \tilde{\mathbf{x}}_F \in \bigcup_{i=1}^{x} \mathfrak{D}_i^F. \tag{4.19}$$

Here, $\mathbf{x}_F(h)$ is the forward correct path up to forward tree level h, and $\tilde{\mathbf{x}}_F$ is a forward incorrect path. Similarly, one can write

$$\hat{C}_{L-x}^B = f(M[\mathbf{x}_B(h)], M[\tilde{\mathbf{x}}_B]),\ where\ \ h \in [1, L-x]\ \ and\ \ \tilde{\mathbf{x}}_B \in \bigcup_{i=1}^{L-x} \mathfrak{D}_i^B. \tag{4.20}$$

Here, $\mathbf{x}_B(h)$ is the backward correct path up to backward tree level h, and $\tilde{\mathbf{x}}_B$ is a backward incorrect path.

Strictly speaking, on a random code tree, $\hat{C}_x^F$ and $\hat{C}_{L-x}^B$ are not truly independent, because an incorrect forward path $\tilde{\mathbf{x}}_F$ may share some common nodes with an incorrect backward path $\tilde{\mathbf{x}}_B$. Fortunately, the probability of this event gets smaller as the code memory m increases.[10] Hence, by neglecting the above event, we can consider $\hat{C}_x^F$ and $\hat{C}_{L-x}^B$ essentially independent, and thus conjecture that Theorem 4.2 also applies to Algorithm TAmerge.

In Algorithm TTmerge, forward and backward merged paths may overlap each other, and hence the analysis of the computational distribution is even more difficult (if not impossible). However, as our computer simulation results (see Figure 4.9 in section 4.6)

[10] The probability of this event is in the same order of the probability of error, which is shown in Chapter 5 to decrease exponentially with m.

suggest, the overlapping area is insignificant, especially when code rate $R < R_{comp}$. Thus, it seems reasonable to suggest that the computational performance of Algorithm TTmerge can be approximated by that of Algorithm TAmeet if the code rate $R < R_{comp}$.

Finally, it is easy to see that the computational distribution of Algorithm HTTmerge is upper bounded by that of Algorithm TTmerge and lower bounded by that of Algorithm TAmeet. Clearly, Algorithm HTTmerge is closer to Algorithm TTmerge when the number of required matching information symbols m_h is close to the code memory m. Otherwise, it is more like Algorithm TAmeet. This phenomenon is clearly observed in the computer simulations (see Figure 4.2 in section 4.6).

4.4 Lower Bound to Computational Distribution

Following Jacobs and Berlekamp [30], we show that with the assumption of correct decoding no algorithm of BSD type can have a computational distribution better than $N^{-2\alpha}$.

Theorem 4.3: For all convolutional (or trellis) codes, any DMC, and with the assumption of correct decoding, the distribution $P\left(C_L^{BSD} > N\right)$ *of any BSD algorithm is lower bounded by*

$$P\left(C_L^{BSD} > N\right) \geq 2^{2\alpha} N^{-2\alpha} \exp\left\{-O\left(\sqrt{lnN}\right)\right\}, \qquad (4.21)$$

where α *is defined in parametric equation (2.7).*

Proof: Let us separate every received block into two almost equal parts, i.e. forward tree search operations will not go beyond forward tree level $\lfloor (L-m)/2 \rfloor$ and backward tree search will not go beyond backward tree level $\lceil (L-m)/2 \rceil$ by the help of a genie as

defined by Jacobs and Berlekamp [30]. Notice that there are m branches in the codeword tree separating the portions considered by the genie-aided forward and backward decoders, allowing them to choose codeword paths independently. Let $C^{FC}_{\lfloor (L-m)/2 \rfloor}$ and $C^{BC}_{\lceil (L-m)/2 \rceil}$ denote the total number of computations required before first correctly decoding the $\lfloor (L-m)/2 \rfloor$ branches by the idealized forward decoder and the $\lceil (L-m)/2 \rceil$ branches by the idealized backward decoder, respectively. Obviously under the assumption of correct decoding, $C^{BSD}_L \geq 2\min\left[C^{FC}_{\lfloor (L-m)/2 \rfloor}, C^{BC}_{\lceil (L-m)/2 \rceil}\right]$ for any BSD algorithm. Therefore, we can write

$$\begin{aligned} P\left(C^{BSD}_L > N\right) &\geq P\left(2\min\left[C^{FC}_{\lfloor (L-m)/2 \rfloor}, C^{BC}_{\lceil (L-m)/2 \rceil}\right] > N\right) \\ &= P\left(C^{FC}_{\lfloor (L-m)/2 \rfloor} > N/2, C^{BC}_{\lceil (L-m)/2 \rceil} > N/2\right). \end{aligned} \tag{4.22}$$

By the assistance of a genie, forward and backward search areas do not overlap, i.e. events $\left(C^{FC}_{\lfloor (L-m)/2 \rfloor} > N/2\right)$ and $\left(C^{BC}_{\lceil (L-m)/2 \rceil} > N/2\right)$ are independent for any DMC. Therefore, we finally get

$$\begin{aligned} P\left(C^{BSD}_L > N\right) &\geq P\left(C^{FC}_{\lfloor (L-m)/2 \rfloor} > N/2\right) \cdot P\left(C^{BC}_{\lceil (L-m)/2 \rceil} > N/2\right) \\ &\geq \left\{ \left(\tfrac{N}{2}\right)^{-\alpha} \exp\left[-O_1\left(\sqrt{\ln \tfrac{N}{2}}\right)\right]\right\}^2 \\ &= 2^{2\alpha} N^{-2\alpha} \exp\left[-O\left(\sqrt{\ln N}\right)\right]. \end{aligned} \qquad \text{Q.E.D.} \tag{4.23}$$

Combining the lower bound (Theorem 4.3) and the upper bound (Theorem 4.2), we demonstrated that the computational distribution of the proposed BSD is Pareto, and its Pareto exponent is between 2α and 2ρ, where $\rho < \rho_r$ is the Pareto exponent of USD. Since the Pareto exponent of USD is approximately α or ρ_r, we conclude that the Pareto exponent of BSD is approximately twice that of USD.

4.5 Average Number of Computations

In this section, we examine the average number of computations of BSD and show that it is smaller than that of USD. The computational cutoff rate of BSD is then discussed.

We define the average number of computations per decoded branch, denoted by C_b, as the total number of computations used to correctly decode one block divided by the block length L. Using Jacobs and Berlekamp's approximation [30] for the distribution of C_Λ^{USD}, which is given by

$$P\left(C_\Lambda^{USD} > N\right) \approx \Lambda N^{-\alpha}, \tag{4.24}$$

we can approximate the distribution of BSD as

$$P\left(C_L^{BSD} > N\right) \gtrapprox \left(2^{\alpha-1}L\right)^2 \cdot N^{-2\alpha}, \tag{4.25}$$

where we assume $L >> m$, and ignore the difference between $\lfloor (L-m)/2 \rfloor$ and $\lceil (L-m)/2 \rceil$ since both terms are almost equal to $L\,/\,2$ when L is large.

The total number of computations per decoded block of BSD can be any integer in the interval $[L, M\,]$ where M is the maximum number of computations in BSD. Thus,

$$\begin{aligned}
E\left[C_b^{BSD}\right] &= E\left[\frac{1}{L}C_L^{BSD}\right] \\
&= \sum_{N=L}^{M} \frac{1}{L} N P\left(C_L^{BSD} = N\right) \\
&= \frac{1}{L}\sum_{N=L}^{M} N\left[P\left(C_L^{BSD} \geq N\right) - P\left(C_L^{BSD} > N\right)\right] \\
&= \frac{1}{L}\sum_{N=L-1}^{M-1} (N+1)P\left(C_L^{BSD} > N\right) - \frac{1}{L}\sum_{N=L}^{M} NP\left(C_L^{BSD} > N\right)
\end{aligned}$$

$$= \frac{1}{L}\sum_{N=L}^{M-1} P\left(C_L^{BSD} > N\right) + P\left(C_L^{BSD} > L-1\right) - \frac{M}{L}P\left(C_L^{BSD} > M\right). \tag{4.26}$$

In (4.26), $P\left(C_L^{BSD} > L-1\right) = 1$ and $P\left(C_L^{BSD} > M\right) = 0$. Therefore, using (4.25), (4.26) becomes

$$E\left[C_b^{BSD}\right] = 1 + \frac{1}{L}\sum_{N=L}^{M-1} P\left(C_L^{BSD} > N\right) \gtrapprox 1 + 2^{2\alpha-2}\cdot L\sum_{N=L}^{M-1} N^{-2\alpha}. \tag{4.27}$$

Moreover, similar to [30], we can write

$$\sum_{N=L}^{M-1} N^{-2\alpha} \geq \int_L^M x^{-2\alpha}dx = \begin{cases} \frac{1}{1-2\alpha}\left(M^{1-2\alpha} - L^{1-2\alpha}\right), & \alpha \neq \frac{1}{2} \\ \ln M - \ln L, & \alpha = \frac{1}{2} \end{cases}. \tag{4.28}$$

Finally, using (4.28), we obtain

$$E\left[C_b^{BSD}\right] \gtrapprox \begin{cases} 1 + \frac{2^{2\alpha-2}}{(1-2\alpha)}\left[L\cdot M^{1-2\alpha} - L^{2(1-\alpha)}\right], & \alpha \neq \frac{1}{2} \\ 1 + 2^{2\alpha-2}\cdot L\cdot(\ln M - \ln L), & \alpha = \frac{1}{2} \end{cases}. \tag{4.29}$$

Similarly, one can get the average number of computations per decoded branch for USD as

$$E\left[C_b^{USD}\right] \gtrapprox \begin{cases} 1 + \frac{1}{1-\alpha}\left(M_1^{1-\alpha} - L^{1-\alpha}\right), & \alpha \neq 1 \\ 1 + \ln M_1 - \ln L, & \alpha = 1 \end{cases}, \tag{4.30}$$

where M_1 is the maximum number of computations in USD.

From (4.29) and (4.30), it can be seen that $E\left[C_b^{BSD}\right]$ is smaller than $E\left[C_b^{USD}\right]$ for any α. However, the difference becomes less significant when α is much larger than

one. This is because the average computation for both USD and the proposed BSD is essentially one computation per branch when $\alpha > 1$. However, it is expected that higher order moments of the computation will be significantly improved by the proposed BSD even for high α.

Let us now discuss the computational cutoff rate. The computational cutoff rate of sequential decoding is defined as the supremum of rates for which the average computation per branch is bounded [30]. It is well known that with the assumption of correct decoding the computational cutoff rate R_{comp} of USD is equal to $E_0(1)$ [30, 53]. R_{comp} is a truly critical channel parameter, and with the assumption of correct decoding the average number of computations per decoded branch of USD is unbounded if the code rate $R > R_{comp}$ [30, 53]. In the following theorem, it is shown that the computational cutoff rate for BSD with the assumption of correct decoding is the same as that of USD.

Theorem 4.4: Consider an infinite sequence of convolutional (or trellis) codes $\{\mathfrak{C}_i\}$ *and any DMC. For code* $\mathfrak{C}_i$, *let* R_i *denote its rate and* m_i *its memory. Suppose each code has n symbols per branch in its codeword tree, and* $R_i \geq R$ *for all i, for some* $R > R_{comp}$. *Code* $\mathfrak{C}_i$ *is used with block length* L_i *(including a known tail to terminate the encoder in a known state) for BSD. Equiprobable codewords and arbitrary sequential decoder are assumed. Under the assumption of correct decoding,*[11] *if* $L_i \to \infty$ *and* $L_i \geq \left(1 + \epsilon_0^{-1}\right) m_i$ *for any i and some* $0 < \epsilon_0 \leq 1$, *then the expected number of computations per branch satisfies*

$$E\left[C_b^{BSD}\right] \to \infty. \tag{4.31}$$

[11] This assumption is equivalent to defining the number of computations needed to be infinite if BSD makes decoding error.

Proof: See Appendix A. Q.E.D.

As a simple approach, from (4.29) one can actually show that $E[C_b^{BSD}] \rightarrow \infty$ when $L \rightarrow \infty$ and $\alpha < 1$. However, it is important to notice that the above computational cutoff rate is based on the assumption of correct decoding. As suggested by Anderson [51], this assumption is unrealistic and in fact there is no cutoff rate phenomenon for sequential decoding as long as $P_e > 0$. Nevertheless, the cutoff rate is a bound such that above this rate the average computation will increase exponentially. From (4.29) and (4.30), one can easily see that when $\alpha < 1$, i.e. above the cutoff rate, $E[C_b^{BSD}]$ is much smaller than $E[C_b^{USD}]$ although it is also much larger than one. Hence, the useful range of code rates with the proposed BSD is increased compared to that with USD according to Anderson's definition on a "useful decoder" [51].

4.6 Numerical and Simulation Results

Computer simulations have been performed in order to verify the above theoretical results. Non-systematic rate $r = 1/2$ (bits per channel symbol) convolutional codes were used. All algorithms were run over a BSC with a transition probability p under strictly identical conditions (i.e., using the same code and noise sequences), in order to make them comparable. In our computer simulations, metrics were scaled into integers. In Algorithm TAmeet, Algorithm TAmerge and Algorithm HTTmerge, the substack spacing Δ was chosen to be equal to one, i.e., using unquantized stacks. Different substack spacing values, namely $\Delta = 1$ and $\Delta = 7$ were chosen in Algorithm TTmerge. Notice here that there may exist more than one path in the highest non-empty substack even when $\Delta = 1$.

Figure 4.2 shows the distributions of the total number of computations per decoded block of both USD and BSD algorithms for the case $p = 0.0409$, corresponding to a Pareto exponent $\rho_r = 1.1$ (i.e., $R_{comp} = 0.52$ bits per channel symbol). A SBODP code of memory $m = 23$ was used, and 200,000 blocks each of length $L = 400$ bits (377 information bits) were simulated for each algorithm. The Pareto approximations for USD (stack algorithm $\Delta = 1$) and BSD (Algorithm TAmerge) are also indicated on the figure. As expected, the use of BSD results in a significant reduction of the computational variability of sequential decoding. More specifically, the slope of the straight line that approximates the tail of the distribution of Algorithm TAmerge is equal to 2.16, which is more than twice that of USD. This is in agreement with our derived upper bound on the computational distribution (Theorem 4.2). Moreover, it also agrees with the lower bound (Theorem 4.3), i.e., less than 2α. It can be noticed from Figure 4.2 that the slopes of the computational distributions of all BSD-merge algorithms are similar to that of Algorithm TAmeet, which suggests that the additional computations associated with the merging test are asymptotically unimportant as compared to the computations of Algorithm TAmeet when the code rate $R < R_{comp}$.

As Forney [36] suggested, Figure 4.2 shows that an unidirectional quantized stack algorithm (Δ=7) increases the computational effort compared to an unidirectional unquantized stack algorithm (Δ=1). However, Figure 4.2 shows that the tail of the computational distribution of Algorithm TTmerge improves when the quantized stack algorithm (Δ=7) is used. This is due to the fact that if the substack spacing Δ is increased, more paths will be tested in the merging test so that the chance of merging gets enlarged.

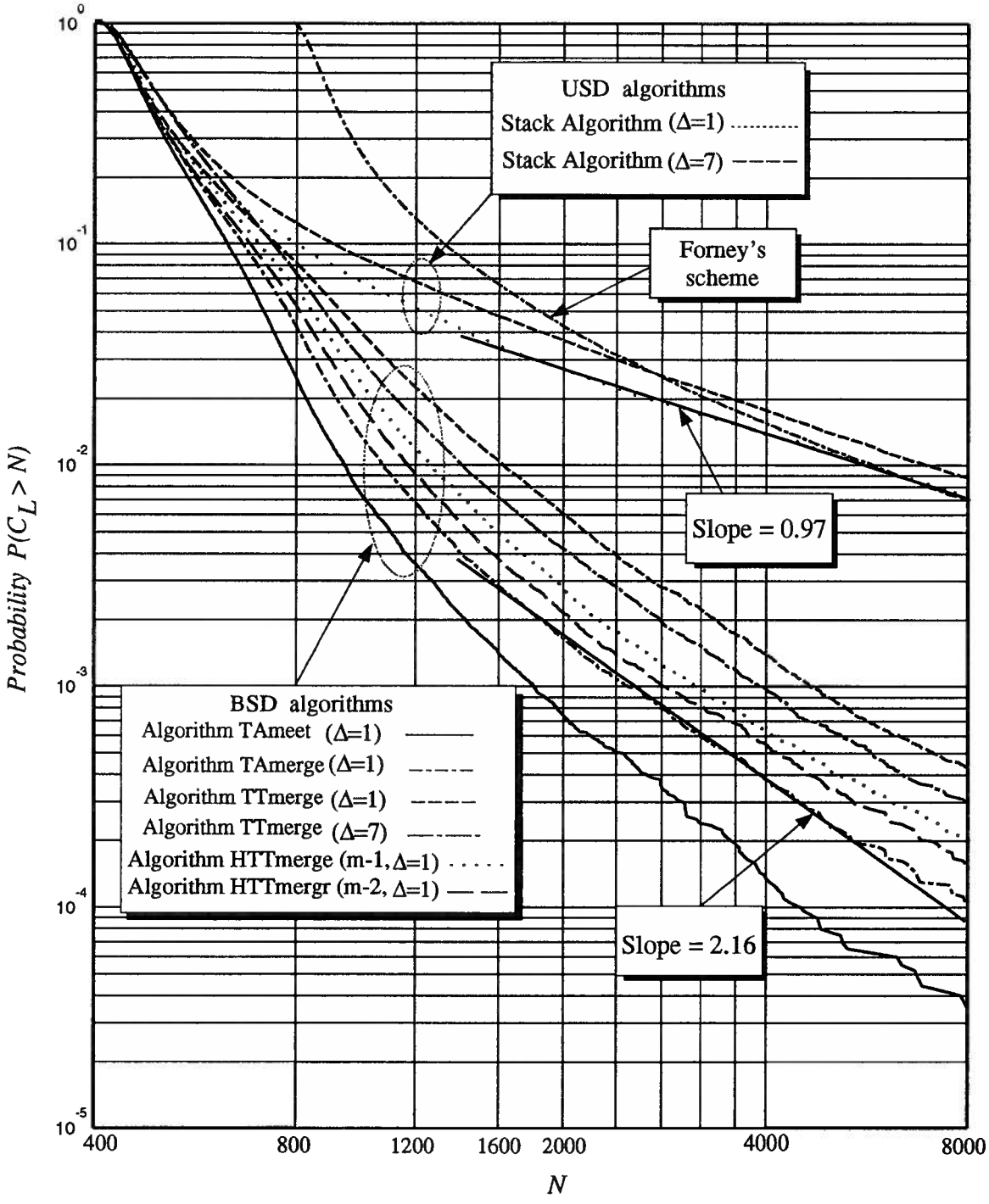


Figure 4.2 Distribution of the total number of computations per decoded block for the case $L = 400$ and $p = 0.0409$, i.e., $\rho_r = 1.1$.

For comparison purposes, the computational distribution of Forney's scheme [22] built on the stack algorithm is also depicted in Figure 4.2. As we pointed out in Chapter 3, the total number of computations of Forney's scheme is equal to $2\min\left(C_L^F, C_L^B\right)$. Hence, the minimum number of computations with this scheme is equal to twice that of USD. Clearly, as indicated in the figure, Forney's scheme does not provide any significant advantage over conventional USD. This is mainly because the dip size in the correct path metric is essentially the same whether decoding is performed from forward or backward directions (see Figure 4.1). Thus, the number of computations, which is mainly determined by the dip size of correct path metric, is basically the same for both forward and backward decoders.

Table 4.1 shows the average number of computations per decoded branch of different algorithms. Clearly, the average number of computations is reduced by using the proposed BSD. Table 4.1 also shows the number of erasure blocks (undecoded blocks) with a maximum allowed computation $C_{lim} = 8000$. These results suggest that our BSD can significantly alleviate the erasure or overflow problem, which is the main obstacle in the application of sequential decoding. Furthermore, Table 4.1 shows that with an increase in computations of about 3% with Algorithm TAmerge compared to Algorithm TAmeet, decoding errors (18.3% decoded blocks were in error after decoding by Algorithm TAmeet) were practically eliminated.[12] Thus, with the use of the merging test in BSD, decoding errors can be significantly reduced. For Algorithm HTTmerge, Figure 4.2 and Table 4.1 suggest that there exists a substantial computational gain by just letting $m_h = (m - 1)$ in the merging test. Thus, Algorithm HTTmerge can easily provide

[12] For a more detailed analysis and computer simulation results on the error performance of the proposed BSD, refer to Chapter 6.

Table 4.1 Comparison of average computations for $L = 400$, $C_{lim} = 8000$ and $p = 0.0409$.

SBODP Code $m = 23$ ρ_r=1.1, L=400, 2×10^5runs	Ave. comp. per branch	Erasure blocks	Error blocks
USD (unquantized stack algorithm $\Delta = 1$)	1.621	1324	0
USD (quantized stack algorithm $\Delta = 7$)	1.776	1743	0
Algorithm TAmeet ($\Delta = 1$)	1.264	7	36614
Algorithm TAmerge ($\Delta = 1$)	1.303	21	1
Algorithm TTmerge ($\Delta = 1$)	1.390	84	1
Algorithm TTmerge ($\Delta = 7$)	1.393	59	0
Algorithm HTTmerge ($m_h = m - 1$, $\Delta = 1$)	1.333	39	1
Algorithm HTTmerge ($m_h = m - 2$, $\Delta = 1$)	1.319	31	3

a trade-off between computational effort and error performance by adjusting the number of matching information symbols m_h.

Figure 4.3 compares the computational distributions of Algorithm TAmerge for ODP, SBODP and SABODP codes all of memory $m = 23$, using the same parameters as in Figure 4.2. It can be seen from this figure that the distribution of the number of computations per decoded block using a SBODP code is better than that with an ODP code. However, the distribution of the number of computations per decoded block using a SABODP code is almost identical to that with a SBODP code. With a SABODP code, no errors were found after decoding because its free distance (equals to 24) is much larger than the free distance of the SBODP code (equals to 18). These simulation results

are clearly in agreement with our theoretical observations.

Figure 4.4 shows the distributions of the total number of computations per decoded block of different algorithms using a systematic ODP code ($m = 23$). As expected, this figure shows that the computational distribution of BSD is almost the same as that of USD. This is due to the fact that the backward code of an ODP forward code is too poor and hence the backward decoder can hardly ever help in decoding. Figure 4.4 also demonstrates that the number of computations needed in Forney's scheme is essentially equal to twice that of USD.

Figure 4.5 shows the computational distributions of the total number of computations per decoded block of both USD and our BSD algorithms for the case $p = 0.0594$, corresponding to a Pareto exponent $\rho_r = 0.68$ (i.e., $R_{comp} = 0.44$ bits per channel symbol). The same code as in Figure 4.2 was used again, and 10,000 blocks each of length $L = 200$ bits were simulated for each algorithm. The Pareto approximations of these distributions are also indicated on the figure. In Figure 4.5, the computational distribution of Algorithm TAmeet (a BSD-no-merge) clearly violates the lower bound given by Theorem 4.3 because there were too many decoding errors (44.1% decoded blocks were in error as illustrated in Table 4.2).[13] However, no errors were noticed after decoding by Algorithm TTmerge, while only one block was in error after decoding by Algorithm TAmerge. The slope of the straight line that approximates the tail of the computational distribution of Algorithm TAmerge is equal to 0.89, which is more than twice that of USD. Again, it agrees with our upper and lower bounds. Moreover,

---

[13] Notice that the lower bound in Theorem 4.3 is based on the assumption of correct decoding.

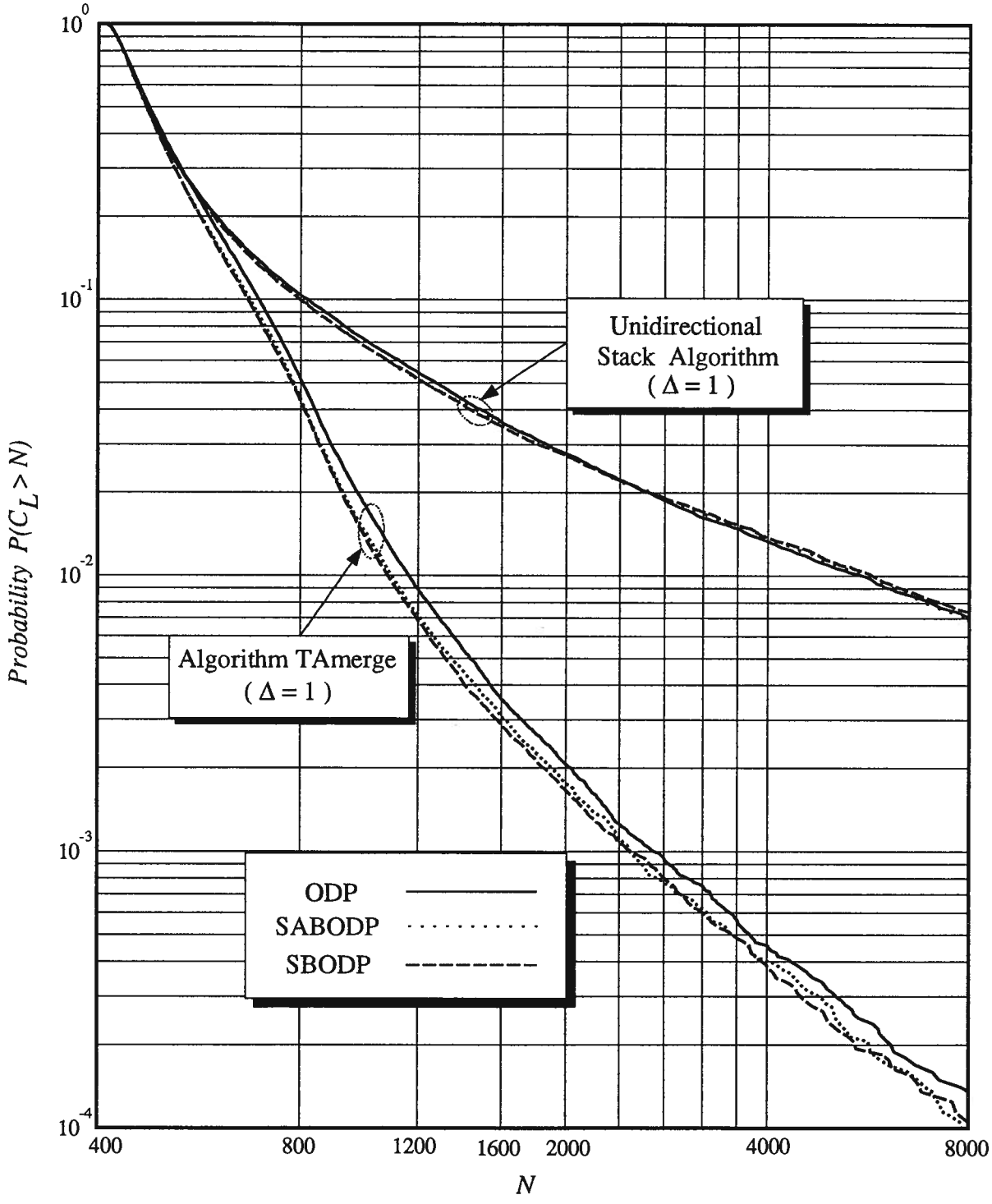


Figure 4.3 Distribution of the total number of computations per decoded block for different codes using the same parameters as in Figure 4.2.

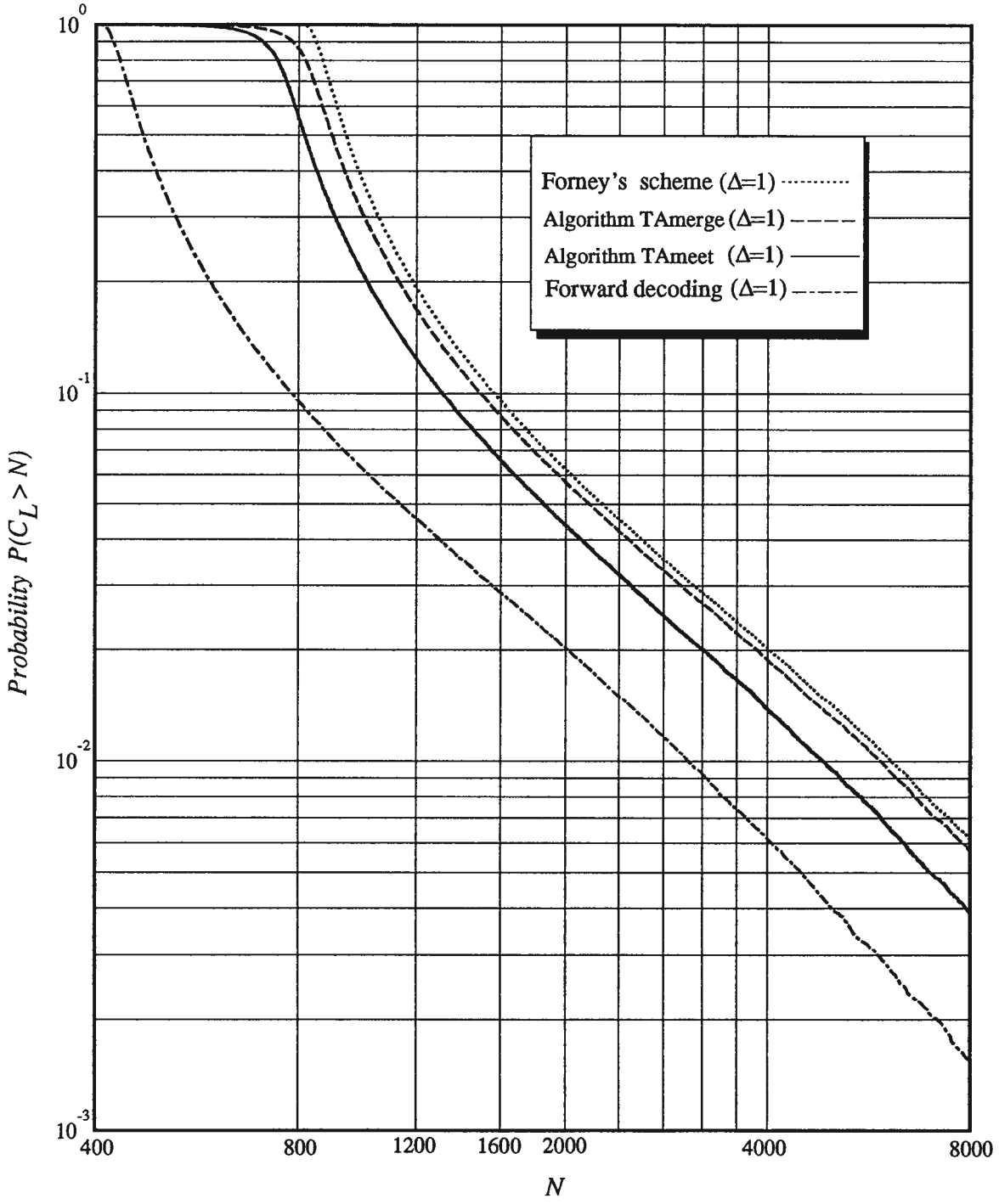


Figure 4.4 Distribution of the total number of computations per decoded block for a systematic ODP code (ρ_r = *1.1*).

the slopes that approximate the computational distributions of Algorithm TTmerge and Algorithm TAmerge are almost the same.

Table 4.2 compares the average number of computations for different algorithms for the case $p = 0.0594$. The number of erasure blocks and the number of error blocks are also shown in the table. It can be seen that the number of erasure blocks is substantially reduced by using the proposed BSD. Also, by using the merging test, almost all errors associated with Algorithm TAmeet were avoided. Furthermore, by comparing Table 4.1 and Table 4.2, one can notice that the improvements in the average number of computations of our BSD compared to that of USD are more significant when the Pareto exponent ρ_r is less than one. As anticipated, the useful range of code rates [51] with our BSD is increased as compared to the case with USD. These observations agree with our suggestions made in section 4.5.

Figure 4.6 shows the distributions of the total number of computations per decoded block of both USD and our BSD algorithms for the case $p = 0.0289$, corresponding to a Pareto exponent $\rho_r = 1.5$ (i.e., $R_{comp} = 0.58$ bits per channel symbol). The same code as in Figure 4.2 was used, and 5,000,000 blocks each of length $L = 400$ bits were run for each algorithm. Again, the slopes that approximate the computational distributions of the different BSD algorithms are quite similar. Once again, these simulation results are in agreement with our analytical results.

Table 4.3 compares the average number of computations for different algorithms for the case $p = 0.0289$. The average number of computations, the number of erasure blocks and the number of error blocks are also shown in Table 4.3. It can be noticed that BSD-merge eliminates all decoding errors associated with Algorithm TAmeet (4.7%

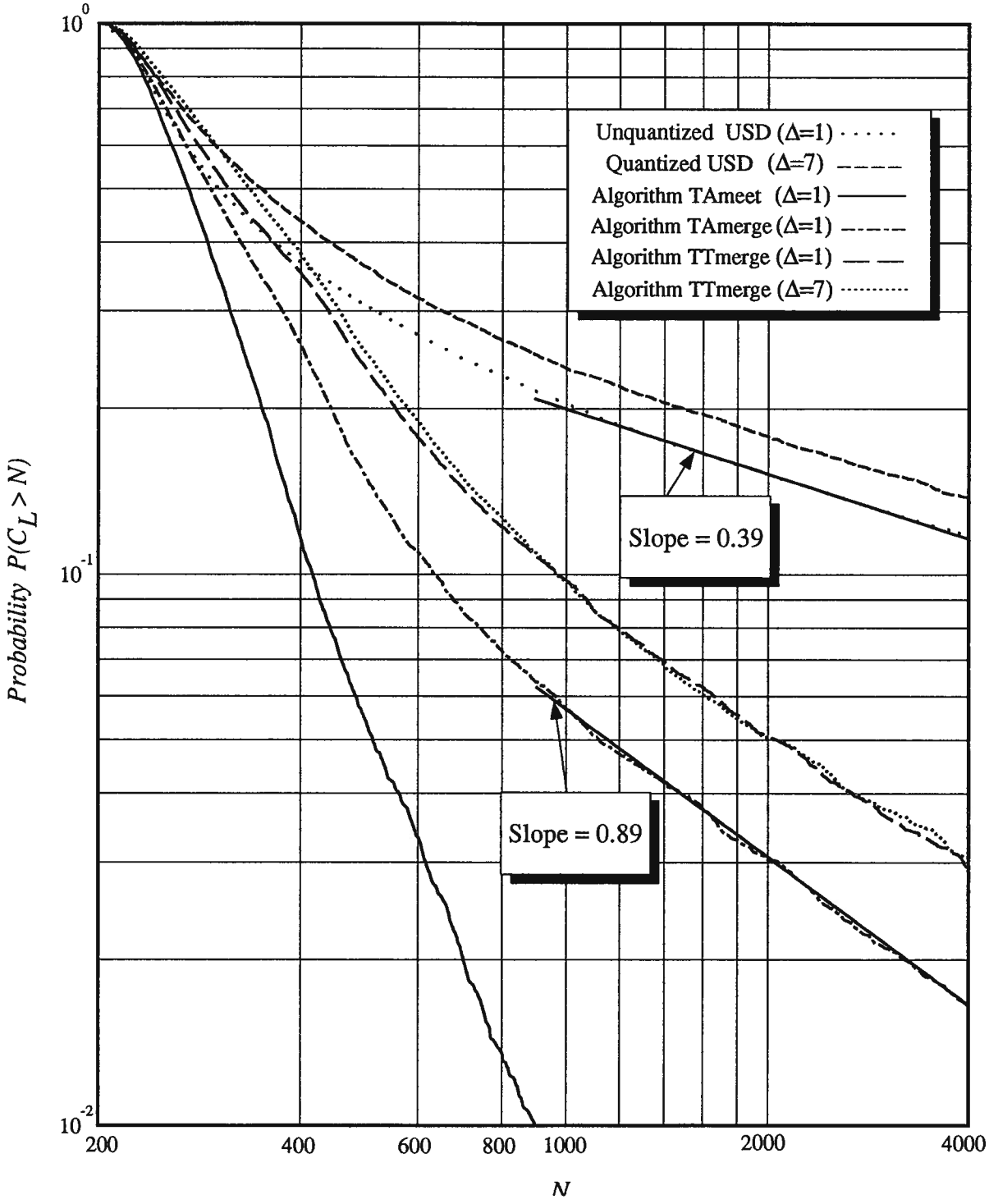


Figure 4.5 Distribution of the total number of computations per decoded block for the case $L = 200$ and $p = 0.0594$, i.e., $\rho_r = 0.68$.

Table 4.2 Comparison of average computations for $L = 200$, $C_{lim} = 4000$ and $p = 0.0594$.

SBODP Code $m = 23$ ρ_r=0.68, L=200, 10^4 runs	Ave. comp. per branch	Erasure blocks	Error blocks
USD (unquantized stack algorithm $\Delta = 1$)	4.561	1175	0
USD (quantized stack algorithm $\Delta = 7$)	5.106	1382	0
Algorithm TAmeet ($\Delta = 1$)	1.541	0	4406
Algorithm TAmerge ($\Delta = 1$)	2.284	166	1
Algorithm TTmerge ($\Delta = 1$)	2.799	293	0
Algorithm TTmerge ($\Delta = 7$)	2.894	306	0

blocks). Moreover, the additional number of computations due to the merging test is only 0.5% with Algorithm TAmerge compared to Algorithm TAmeet. All results show that the additional computations due to the merging test are worth the reward and those computations can be ignored when the code rate $R < R_{comp}$. As indicated in the table, the average number of computations of USD and our BSD are quite comparable here. However, due to the reduction in computational variability, as shown in Figure 4.6, it is expected that higher order moments will be smaller with the proposed BSD than with USD.

Figure 4.7 shows the empirical probability masses of merging and meeting points in BSD for the two cases $\rho_r = 1.1$ and $\rho_r = 1.5$. It can be seen that statistically, there is essentially no difference between the meeting and merging points. This implies that the overlapping portion between forward and backward search areas when the merging test

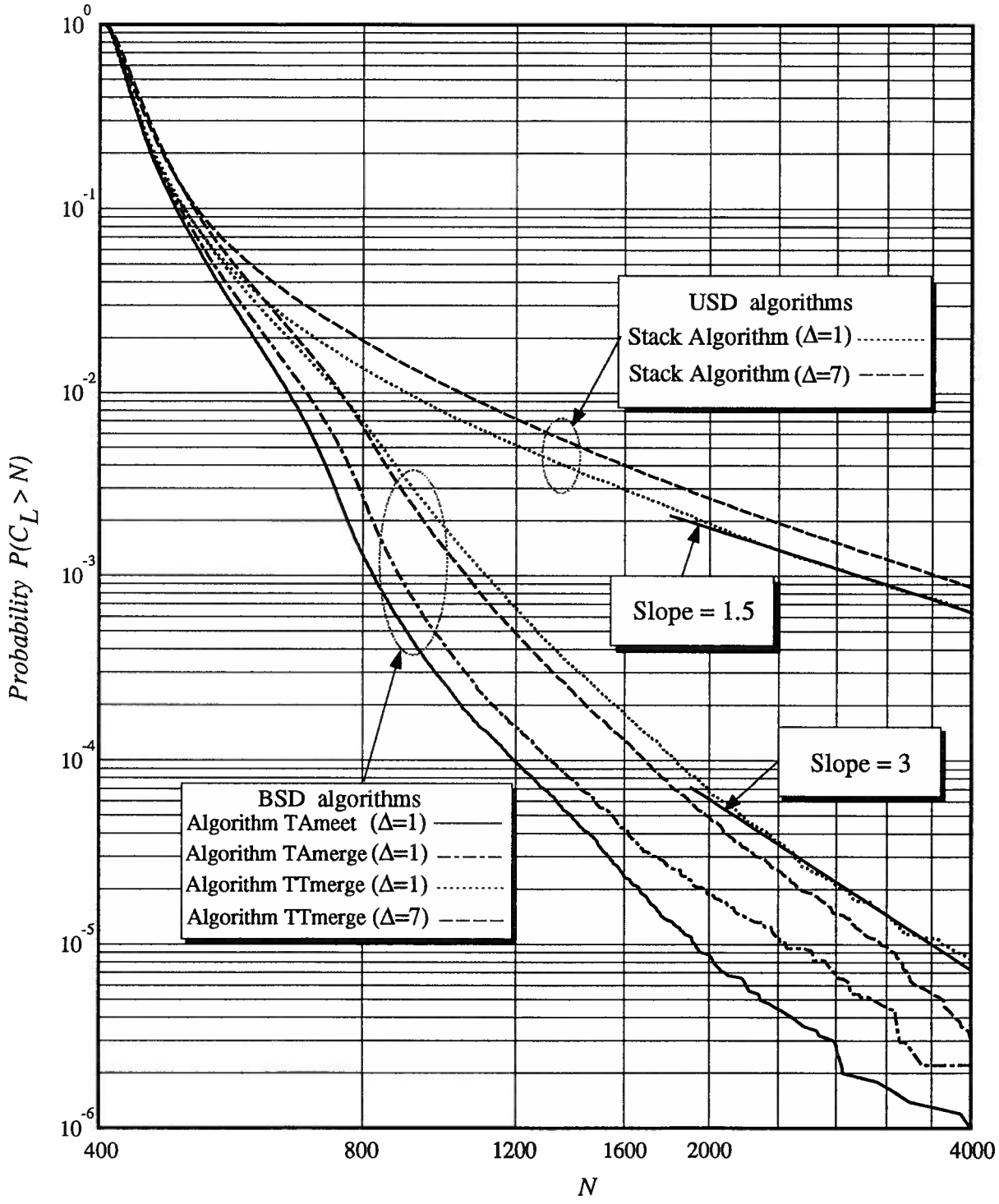


Figure 4.6 Distribution of the total number of computations per decoded block for the case $L = 400$ and $p = 0.0289$, i.e., $\rho_r = 1.5$.

Table 4.3 Comparison of average computations for $L = 400$, $C_{lim} = 4000$ and $p = 0.0289$.

SBODP Code $m = 23$ ρ_T=1.5, L=400, 5×10^6 runs	Ave. comp. per branch	Erasure blocks	Error blocks
USD (unquantized stack algorithm $\Delta = 1$)	1.139	575	0
USD (quantized stack algorithm $\Delta = 7$)	1.167	4336	0
Algorithm TAmeet ($\Delta = 1$)	1.107	5	233321
Algorithm TAmerge ($\Delta = 1$)	1.113	11	0
Algorithm TTmerge ($\Delta = 1$)	1.122	42	0
Algorithm TTmerge ($\Delta = 7$)	1.137	15	0

is used is negligible when the code rate $R < R_{comp}$. Moreover, Figure 4.7 also shows that the probability mass of the merging (meeting) point is concentrated towards the middle of the block. That is to suggest that when the channel is relatively quiet (i.e., $\rho_r > 1$), the merging (or meeting) point is more likely to be near $L / 2$, which means that the computational lower bound on the computational distribution (Theorem 4.3) is quite accurate in this case.

Figure 4.8 shows the empirical probability masses of the merging points for Algorithm TAmerge and Algorithm TTmerge and that of the meeting point of Algorithm TAmeet, for the case $\rho_r = 0.68$. As can be seen, the probability mass of the merging point with Algorithm TAmerge is quite similar to that with Algorithm TTmerge. However, unlike the above case (i.e., $R < R_{comp}$), here the probability mass of the meeting point starts to differ from that of the merging point. This suggests that the additional number of

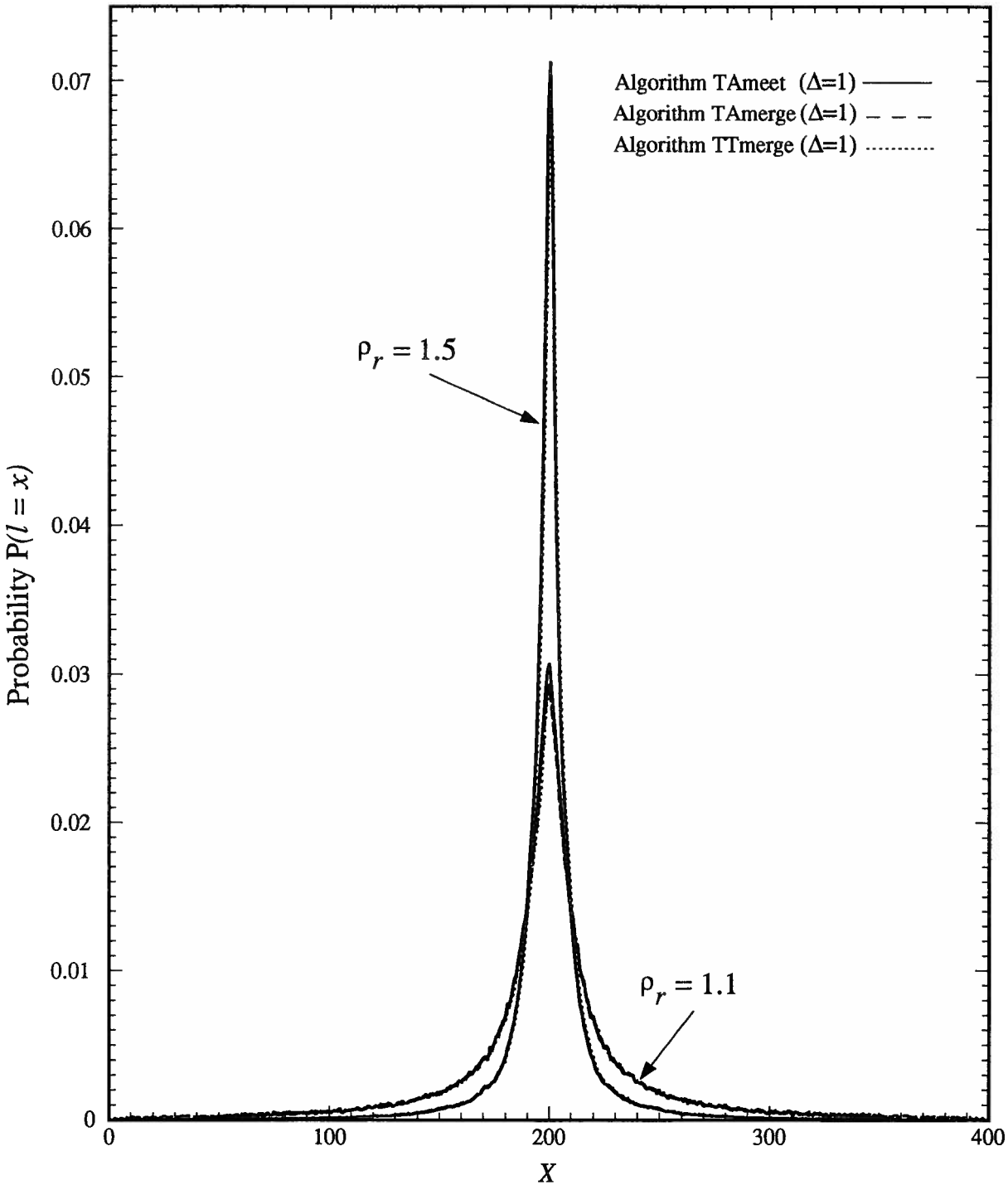


Figure 4.7 Probability mass of merging and meeting when code rate < R_{comp}.

computations needed for merging from the first meeting point increases as the code rate increases beyond R_{comp}.

Figure 4.9 shows the empirical distributions of overlapping lengths in Algorithm TAmerge and Algorithm TTmerge, which is defined as $O_v = l_F + l_B - L$ at the end of decoding. O_v represents the portion of the tree over which forward search and backward search overlap each other. In Figure 4.9, the block length *L=200* for the case $\rho_r = 0.68$, and *L=400* for $\rho_r = 1.1$ and $\rho_r = 1.5$. As expected, the distribution of the overlapping length decays faster as the channel becomes less noisy. Figure 4.9 suggests that the distribution of the overlapping length decays in an exponential fashion. This is because O_v is essentially the sum of the deepest depths reached by forward and backward decoders, and it is known that the distribution of the depth x of an incorrect path explored by a sequential decoder decreases exponentially with x [4, 38].

Finally, since in BSD two separate tree search processors can work in parallel, it seems more reasonable to define a computation in BSD as hypotheses (extension) of two nodes, one on the forward direction and the other one on the backward direction. Clearly, all analyses in this chapter still hold except that *"N"* should be replaced by *"2N"*. As a result of this definition, our computer simulation results should be shifted horizontally to the left by a factor of two.

Figure 4.10 compares the distributions of Algorithm TTmerge ($\Delta = 1$) for different SABODP codes ($m = 12, 16, 23$) using the same parameters as in Figure 4.2, but with the new definition of one computation with BSD. The number of blocks used for each simulation was 200,000 each of length $L = 400$. The maximum number of computations allowed to decode one block was set to 4000. The computational distributions of USD

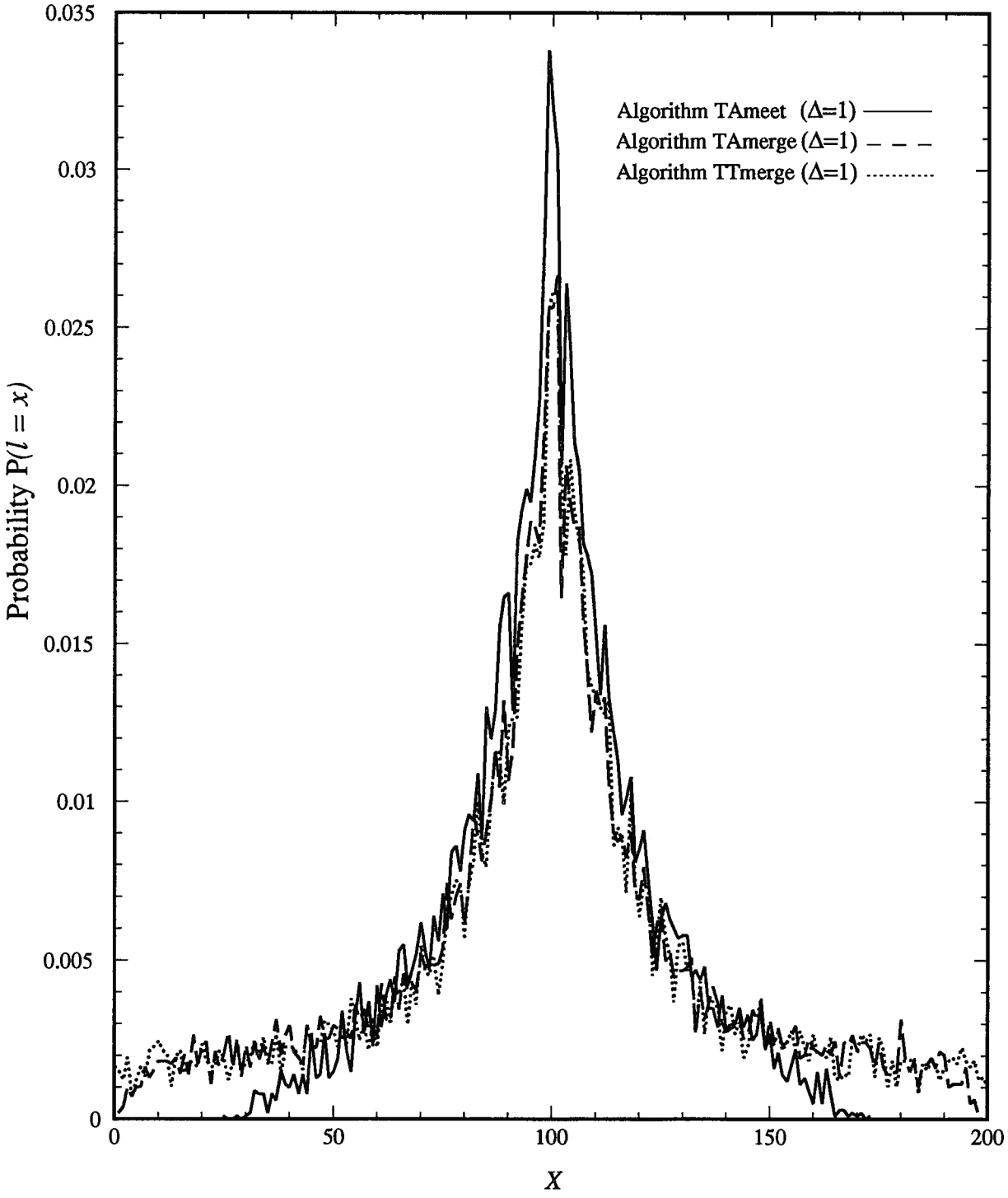


Figure 4.8 Probability mass of merging and meeting when code rate > R_{comp}.

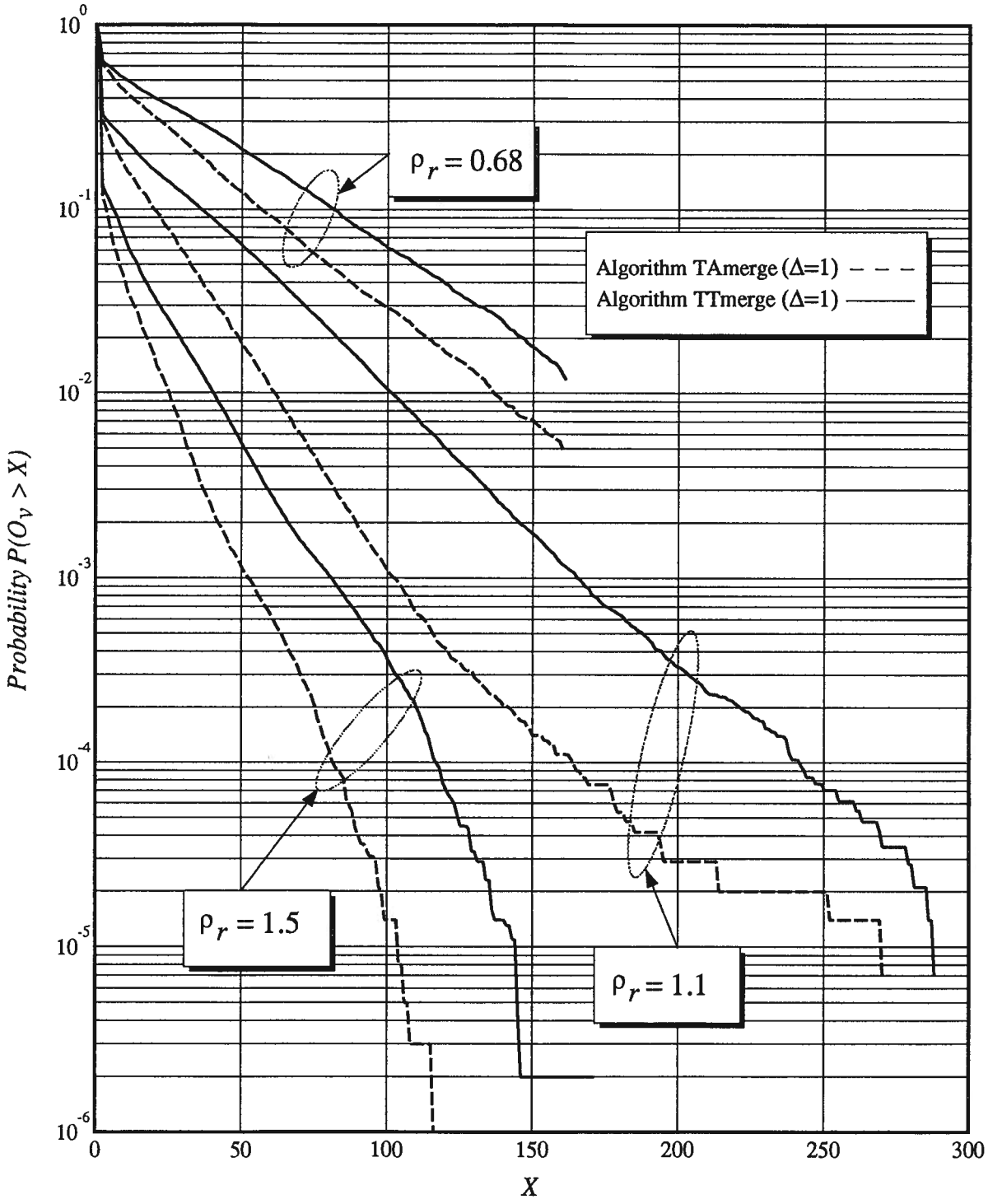


Figure 4.9 Distribution of the overlapping length.

Table 4.4 Comparison of computational and error performances of different m with the new definition of one computation.

SABODP Codes ρ_r=1.1, L=400, 2×10^5 runs	Ave. comp. per branch	Erasure blocks	Error blocks
USD (unquantized stack algorithm $\Delta = 1$, $m = 12$)	1.571	2761	100
USD (quantized stack algorithm $\Delta = 1$, $m = 16$)	1.564	2793	11
USD (quantized stack algorithm $\Delta = 1$, $m = 23$)	1.530	2595	0
Algorithm TTmerge ($\Delta = 1$, $m = 12$)	0.710	42	383
Algorithm TTmerge ($\Delta = 1$, $m = 16$)	0.707	73	27
Algorithm TTmerge ($\Delta = 1$, $m = 23$)	0.700	77	0

are also shown in Figure 4.10. Moreover, the number of erasure blocks, the number of error blocks, and the average number of computations, (using the new definition of one computation with BSD) are shown in Table 4.4. Similar to USD, the number of computations of BSD is essentially not related with m. However, the error performance is improved with increasing m.

In summary we have demonstrated that the use of the proposed BSD can reduce both the computational variability and average number of computations of sequential decoding. It should be pointed out at this stage that in our comparison of BSD-merge algorithms with USD, we did not take into account the additional processing time needed to perform the merging test in BSD-merge. However, one can use extra processors to perform this task that would not slow down the node extension operations of sequential decoding. Therefore, it seems that the additional complexity due to the merging test

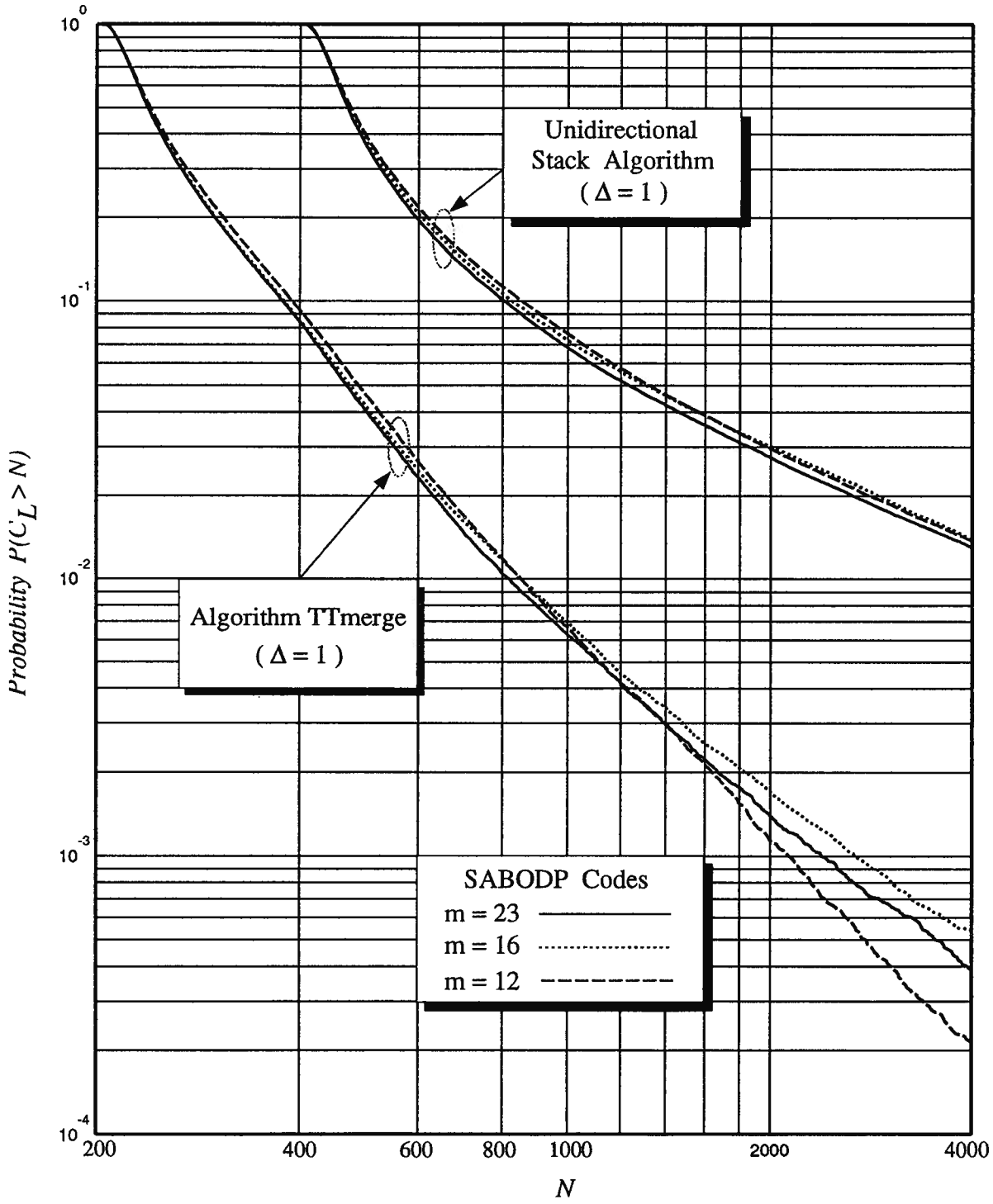


Figure 4.10 Distribution of the total number of computations for different m with the new definition of one computation.

is unimportant compared to the significant improvements in computational performance obtained by using the proposed BSD.

Chapter 5
Error Performance of BSD

In this chapter, the error performance of the proposed BSD algorithms is analyzed through the random coding argument. We assume that the channel is a DMC and BSD is based on the stack algorithm. However, all results obtained in this chapter can also be applied to arbitrary BSD sequential decoders. It is obvious that the error performance of any BSD is lower bounded by that of Viterbi decoding since sequential decoding is suboptimum. Thus, we only need to develop upper bounds on the error performance of the proposed BSD. First, the basic elements of the error performance of USD are summarized. Then, we deal with the probability of decoding error of BSD-merge (Algorithm TAmerge and Algorithm TTmerge). Finally, the error performance of BSD-no-merge (Algorithm TAmeet) is discussed. It is found that for an ensemble of linear trellis codes the error performance of BSD-merge is asymptotically the same as that of USD, while the bit error probability of Algorithm TAmeet satisfies the random coding bound for block codes.

5.1 Error Performance of USD

The basic results of the error performance of USD are summarized in the following. Let $\mathbf{x}(j)$ denote the correct path from root node to level j. Let $\mathbf{x}'_i(t)$ denote an incorrect path which diverges from the correct path $\mathbf{x}$ at level i and remerges at level $(i + t)$.

A *proto-error event* as defined by Forney [36] is an event for which some incorrect path $\mathbf{x}'_i(t)$ has a metric $M[\mathbf{x}'_i(t)]$ at the point of convergence with the correct path $\mathbf{x}$

such that

$$M\left[\mathbf{x}_i'(t)\right] \geq \min_{j \geq i} M[\mathbf{x}(j)]. \tag{5.1}$$

By definition, no error can occur without a proto-error event whereas a proto-error event may not necessarily lead to a decoding error for a unidirectional unquantized stack algorithm [36]. Furthermore, for an incorrect path to remerge with the correct path, t must be no less than the code constraint length K. For an ensemble of linear trellis codes, the block error probability $\overline{P_e^{USD}}$, and the bit error rate $\overline{P_b^{USD}}$ with a unidirectional unquantized stack algorithm can be bounded by [6, 12, 36]

$$\overline{P_e^{USD}} \leq \begin{cases} \frac{\left(e^{nR}-1\right)L}{\left[1-u^{-\epsilon E_o(1)/R}\right]^2} e^{-nKR}, & 0 \leq R \leq (1-\varepsilon)R_{comp} \\ \frac{\left(e^{nR}-1\right)L}{\left[1-u^{-\epsilon E_o(\delta)/R}\right]^2} e^{-nKE_o(\delta)}, & (1-\varepsilon)R_{comp} \leq R \leq (1-\varepsilon)C_p \end{cases}, \tag{5.2}$$

and

$$\overline{P_b^{USD}} \leq \begin{cases} \frac{e^{nR}-1}{\left[1-u^{-\epsilon E_o(1)/R}\right]^3} e^{-nKR}, & 0 \leq R \leq (1-\varepsilon)R_{comp} \\ \frac{e^{nR}-1}{\left[1-u^{-\epsilon E_o(\delta)/R}\right]^3} e^{-nKE_o(\delta)}, & (1-\varepsilon)R_{comp} \leq R \leq (1-\varepsilon)C_p \end{cases}, \tag{5.3}$$

where ϵ is a positive number, R is the code rate as defined in (2.2), $E_o(\delta)$ is the Gallager function [5], C_p is the channel capacity, and for $R_{comp} \leq R \leq C_p$, $\delta \in [0,1]$ is the solution of

$$R = \frac{(1-\varepsilon)E_o(\delta)}{\delta}, \tag{5.4}$$

and for $0 \leq R \leq R_{comp}$, $\delta = 1$.

Now, define a *proto-error event with a bias B* as the event for which some incorrect path $\mathbf{x}_i'(t)$ has a metric $M[\mathbf{x}_i'(t)]$ at the point of convergence with the correct path $\mathbf{x}$ such that

$$M\left[\mathbf{x}_i'(t)\right] \geq \min_{j \geq i} M[\mathbf{x}(j)] - B, \tag{5.5}$$

where B is a positive number. Thus, for the unidirectional quantized stack algorithm with a substack spacing Δ, no error can occur without a proto-error event with a bias $B = \Delta$. Hence, it can be shown that the error performance of the unidirectional quantized stack algorithm is asymptotically the same as that of an unquantized one but with an insignificant increase factor of $A_6 = e^{\Delta\delta/(1+\delta)}$ in (5.2) and (5.3) [6, 36].

5.2 Error Performance of BSD-merge

In this section, the error performance of Algorithm TAmerge using unquantized stacks is analyzed in detail. The results are then extended to other BSD algorithms. Although the following analysis assumes unquantized stacks, it can be easily generalized to the case of quantized stacks, in a similar way as for USD [6, 36].

Let S_{mg} denote the common encoder state of forward and backward portions of the merged path at the merging point l_o. Obviously, S_{mg} can be either correct or incorrect. Figures 5.1 and 5.2 show some typical situations in BSD where the common encoder state S_{mg} is correct and incorrect, respectively. As indicated in Figure 5.1, there is always a possibility that more than one forward (backward) path may merge with more than one path in the opposite direction at encoder state S_{mg}. Should this occurs, Algorithm TAmerge always choose the one (perhaps not the correct one) with the highest cumulative metric. In any case, the encoder state S_{mg} at the merging point l_o in Figure 5.1 is assumed to be the correct one. In Figure 5.2, however, S_{mg} is an incorrect encoder state. In the following, we divide the derivation of the block and bit error probabilities into three steps to prove that the error performance of Algorithm TAmerge is asymptotically the same as that of USD. In the first step, we show that if the encoder state S_{mg} is correct, decoding

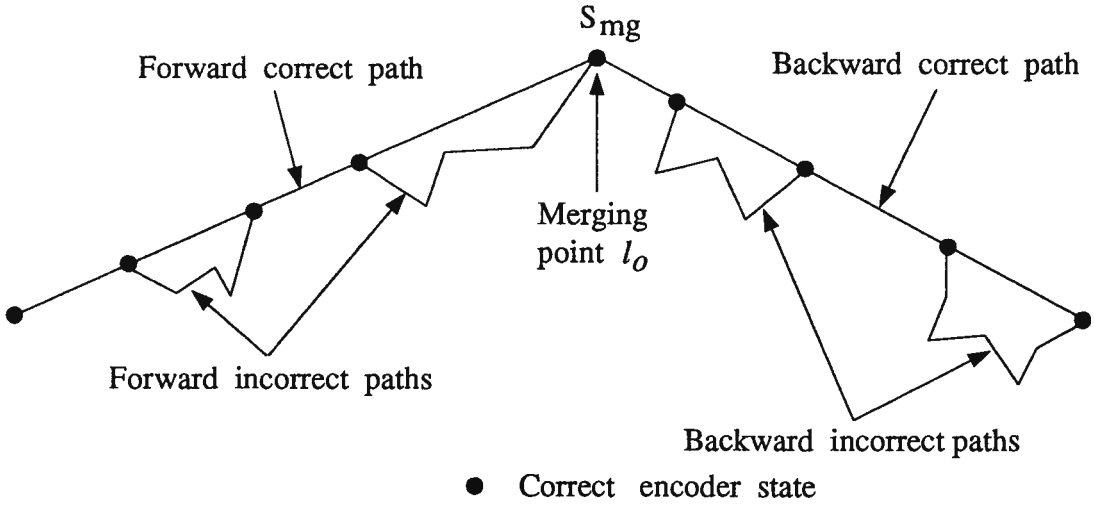


Figure 5.1 Example of error events with a correct merging encoder state.

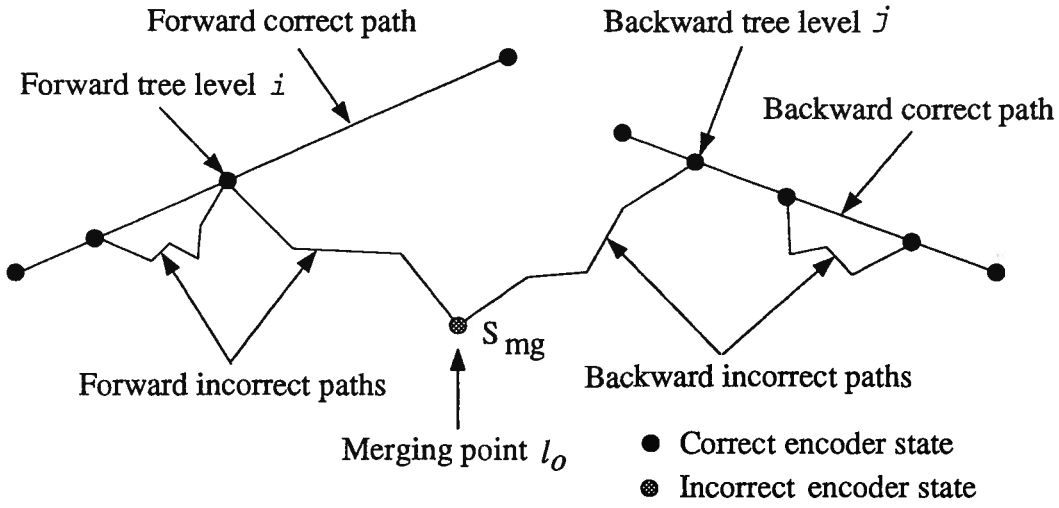


Figure 5.2 Example of error events with an incorrect merging encoder state.

errors can be upper-bounded by the proto-error event with a bias λ, where λ is the largest possible one branch metric drop. In step 2, we investigate the probability that the encoder state S_{mg} is incorrect. Finally, the overall error performance is derived in step 3.

Before investigating the above situations in detail, we need the following lemmas. Consider an incorrect path $\tilde{\mathbf{x}}_i(t)$ which diverges from the correct path $\mathbf{x}$ at level i and

stretches out t branches from the correct node i. First of all, an incorrect path will be extended by a sequential decoder using an unquantized stack algorithm only if [6, 36]

$$M[\tilde{\mathbf{x}}_i(t)] \geq \min_{j \geq i} M[\mathbf{x}(j)]. \tag{5.6}$$

Thus, one can immediately write the following lemma.

Lemma 5.1: An incorrect path $\tilde{\mathbf{x}}_i(t)$ (either in forward decoding or in backward decoding), which diverged from the correct path at level i and remained unmerged up to level (i + t), may be on the top of the stack only if

$$M[\tilde{\mathbf{x}}_i(t)] \geq \min_{i \leq j \leq i+h} M[\mathbf{x}(j)] \tag{5.7}$$

where (i + h) denotes the farthest level of the correct path reached by the unquantized stack algorithm.

Furthermore, notice that for any path present at any position in the stack, the parent node of that path must have been at the top of the stack at an earlier time. Thus, we can write the following lemma.

Lemma 5.2: An incorrect path $\tilde{\mathbf{x}}_i(t)$ (either in forward decoding or in backward decoding), which diverged from the correct path at level i and remained unmerged up to level (i + t), could be in the stack only if

$$M[\tilde{\mathbf{x}}_i(t)] \geq \min_{i \leq j \leq i+h} M[\mathbf{x}(j)] - \lambda \tag{5.8}$$

where (i + h) denotes the farthest level of the correct path reached by the unquantized stack algorithm.

We now proceed with the derivation of the error probability of Algorithm TAmerge.

Step 1: Analyze the error performance of Algorithm TAmerge given that the encoder state S_{mg} is correct. According to the operations of Algorithm TAmerge, merging can occur at either step 3 or step 6. Without loss of generality, we suppose that decoding is stopped at step 3. This means that the end node of the backward portion of the decoded path at level l_o is at the top of the BS, whereas the end node of the forward portion of the decoded path could be at any position in the FS. As indicated in Figure 5.1, decoding error in this case is clearly upper-bounded by proto-error events.

More specifically, let us suppose that there exists a forward incorrect path that merges with a backward incorrect path which is at the top of the BS. Clearly, decoding errors in the backward portion of the decoded path belong to proto-error events without a bias. Furthermore, according to Lemma 5.2, the metric of the forward incorrect path in the FS can not be lower than the minimum cumulative metric of the forward correct path minus λ. Hence, the decoding error event near the merging point l_o made by the forward decoder is bounded by the proto-error event with a bias $B = \lambda$. Clearly, any other decoding errors (if any) in the forward decoder can be upper-bounded by the proto-error event without a bias. In summary, all error events in Algorithm TAmerge when the encoder state S_{mg} is correct can be upper-bounded by the proto-error events with a bias $B = \lambda$. Thus, we can write the following lemma.

Lemma 5.3: If in the merging test the encoder state S_{mg} is correct, decoding error events in Algorithm TAmerge can be upper-bounded by the proto-error events with a bias λ, which is asymptotically the same as that of USD.

Step 2: Investigate the probability that the encoder state S_{mg} is incorrect. Since the

encoder state S_{mg} is incorrect, a decoding error must exist around the merging point l_o (see Figure 5.2). Let us define this decoding error event between the last diverging points of the correct forward and backward paths as the *merging error event* (see Figure 5.3). Clearly, only one such event can occur within each decoded block. Let P_e^{Merge} denote the probability of the merging error event. Obviously, outside the merging error event, everything is the same as in USD, as illustrated in Figure 5.2.

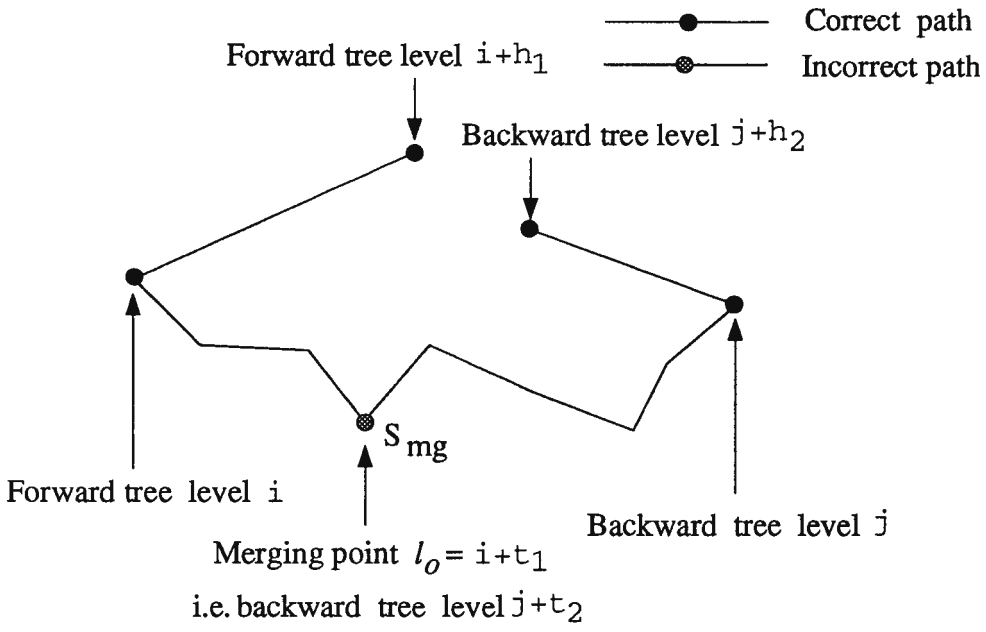


Figure 5.3 Illustration of the merging error event.

In Figure 5.3, we assume that forward and backward incorrect paths diverge from the correct path at forward tree level i and backward tree level j, respectively. As indicated in the figure, t_1 is the length of the forward portion of the decoded path from level i, and t_2 is the length of the backward portion of the decoded path from level j. Let the ensemble of all possible forward incorrect paths which diverge from the correct path at

forward tree level i be denoted by $\mathfrak{X}'_F(i)$. Similarly, let $\mathfrak{X}'_B(j)$ denote the ensemble of all possible backward incorrect paths which diverge from the correct path at backward tree level j. Let $\tilde{\mathbf{x}}_{F,i}(t_1)$ denote a forward incorrect path in $\mathfrak{X}'_F(i)$ whose end node is at forward tree level $(i + t_1)$. Analogously, let $\tilde{\mathbf{x}}_{B,j}(t_2)$ denote a backward incorrect path in $\mathfrak{X}'_B(j)$ whose end node is at backward tree level $(j + t_2)$. According to Property 3.3, $t_1 + t_2 = L - j - i$ and the merging point $l_o = i + t_1 = L - (j + t_2)$ since the forward and backward portions of the decoded path will never overlap. Let $(i + h_1)$ denote the farthest depth of the forward correct path reached by forward decoder. Similarly, let $(j + h_2)$ denote the farthest depth of the backward correct path reached by the backward decoder. Obviously, $h_1 + h_2 \leq t_1 + t_2$ according to Property 3.3.

Clearly, not all paths in $\mathfrak{X}'_F(i)$ and $\mathfrak{X}'_B(j)$ can pairwise merge. Let $\widetilde{\mathfrak{X}}(i;j)$ contain all paths in $\mathfrak{X}'_F(i)$ and $\mathfrak{X}'_B(j)$ which are pairwise merging paths. It is obvious that $\widetilde{\mathfrak{X}}(i;j)$ can be viewed as the set of all incorrect paths which diverge and remerge with the correct path at forward tree level i and $L-j$. Define $\mid \widetilde{\mathfrak{X}}(i;j) \mid$ as the total number of pairwise merging incorrect paths contained in the set $\widetilde{\mathfrak{X}}(i;j)$. Hence, $t_1 + t_2 \geq K$, and $\mid \widetilde{\mathfrak{X}}(i;j) \mid \leq (u-1)u^{L-j-i-K} = (u-1)u^{t_1+t_2-K}$ [6]. Now, let $P_e^{Merge}(i, t_1, h_1, t_2, h_2)$ be the probability of the merging error event as shown in Figure 5.3. Note that if parameters i, t_1, t_2 are given, parameter j is also determined since $j = L - (i + t_1 + t_2)$. We will demonstrate that the upper bound on $P_e^{Merge}(i, t_1, h_1, t_2, h_2)$ is only related with $(t_1 + t_2)$ and $(h_1 + h_2)$ after averaging over the ensemble of linear trellis codes.

Lemmas 5.1 and 5.2 are all that we need to determine the upper bound on the probability of the merging error event, and the method used in our derivation follows

closely that of Viterbi and Omura [6]. Also, it is important to notice that conditions in Lemmas 5.1 and 5.2 are only necessary but not sufficient.

Lemma 5.4: Given i, t_1, h_1, t_2, h_2, the probability of the merging error event in Algorithm TAmerge is upper-bounded by

$$P_e^{Merge}(i,t_1,h_1,t_2,h_2) \le e^{\sigma\delta\lambda} \sum_{\mathbf{y}} P(\mathbf{y}|\mathbf{x}) \sum_{\tau=i}^{i+h_1} e^{-\sigma\delta M[\mathbf{x}_F(\tau)]} \sum_{\tau=j}^{j+h_2} e^{-\sigma\delta M[\mathbf{x}_B(\tau)]}$$
$$\times \left\{ \sum_{(\tilde{\mathbf{x}}_{F,i}(t_1);\tilde{\mathbf{x}}_{B,j}(t_2))\in\widetilde{\mathfrak{X}}(i;j)} e^{\sigma M[\tilde{\mathbf{x}}_{F,i}(t_1)]} e^{\sigma M[\tilde{\mathbf{x}}_{B,j}(t_2)]} \right\}^{\delta}. \qquad (5.9)$$

where $j = L - (i + t_1 + t_2)$.

Proof: Without loss of generality, we assume that decoding is stopped at step 3 in Algorithm TAmerge. That is merging occurs between a top path $\tilde{\mathbf{x}}_{B,j}(t_2)$ in the BS and a path $\tilde{\mathbf{x}}_{F,i}(t_1)$ which could be at any position in the FS. According to Lemmas 5.1 and 5.2, we can write

$$M\left[\tilde{\mathbf{x}}_{F,i}(t_1)\right] - \min_{i\le\tau_1\le i+h_1} M[\mathbf{x}_F(\tau_1)] + \lambda \triangleq g_F(i,t_1,h_1) \ge 0 \qquad (5.10)$$

and

$$M\left[\tilde{\mathbf{x}}_{B,j}(t_2)\right] - \min_{j\le\tau_2\le j+h_2} M[\mathbf{x}_B(\tau_2)] \triangleq g_B(j,t_2,h_2) \ge 0. \qquad (5.11)$$

Moreover, $\tilde{\mathbf{x}}_{F,i}(t_1)$ and $\tilde{\mathbf{x}}_{B,j}(t_2)$ must share the same encoder state S_{mg} at level l_o, i.e., $\left(\tilde{\mathbf{x}}_{F,i}(t_1)\,;\tilde{\mathbf{x}}_{B,j}(t_2)\right) \in \widetilde{\mathfrak{X}}(i;j)$.

Similar to [6, 36], we can write

$$P_e^{Merge}(i,t_1,h_1,t_2,h_2) \le \sum_{\mathbf{y}} P(\mathbf{y}|\mathbf{x})\phi(\mathbf{y}), \qquad (5.12)$$

where the received code vector[14] runs over all symbols between forward tree level i and backward tree level j, and the indicator function $\phi(\mathbf{y})$ is defined as

$$\phi(\mathbf{y}) \triangleq \begin{cases} 1, & if\ g_F(i,t_1,h_1) \geq 0\ and\ g_B(j,t_1,h_2) \geq 0 \\ & \quad for\ some\ (\tilde{\mathbf{x}}_{F,i}(t_1);\tilde{\mathbf{x}}_{B,j}(t_2)) \in \widetilde{\mathfrak{X}}(i;j) \\ 0, & otherwise. \end{cases} \tag{5.13}$$

As in [6, 36], if for a given $\mathbf{y}$, $\phi(\mathbf{y}) = 1$, then for $\sigma \geq 0$

$$\exp\{\sigma \cdot g_F(i,t_1,h_1)\} \cdot \exp\{\sigma \cdot g_B(j,t_2,h_2)\} \geq 1, for\ some\ (\tilde{\mathbf{x}}_{F,i}(t_1);\tilde{\mathbf{x}}_{B,j}(t_2)) \in \widetilde{\mathfrak{X}}(i;j). \tag{5.14}$$

Hence for this $\mathbf{y}$

$$\Big[\sum_{(\tilde{\mathbf{x}}_{F,i}(t_1);\tilde{\mathbf{x}}_{B,j}(t_2)) \in \widetilde{\mathfrak{X}}(i;j)} \exp\{\sigma \cdot g_F(i,t_1,h_1)\} \cdot \exp\{\sigma \cdot g_B(j,t_2,h_2)\} \Big]^{\delta} \geq 1 = \phi_F(\mathbf{y}),\ \sigma,\delta \geq 0. \tag{5.15}$$

while if $\phi(\mathbf{y}) = 0$, (5.15) holds trivially. Also, we can write [36, 6]

$$\exp -\sigma\delta\{\min_{i \leq \tau_1 \leq i+h_1} M[\mathbf{x}_F(\tau_1)]\} \leq \sum_{\tau=i}^{i+h_1} \exp -\sigma\delta M[\mathbf{x}_F(\tau)], \tag{5.16}$$

and

$$\exp -\sigma\delta\{\min_{j \leq \tau_2 \leq j+h_2} M[\mathbf{x}_B(\tau_2)]\} \leq \sum_{\tau=j}^{j+h_2} \exp -\sigma\delta M[\mathbf{x}_B(\tau)]. \tag{5.17}$$

Substituting (5.16) and (5.17) into (5.15), and combining (5.12) and (5.15) yield (5.9). Q.E.D.

As in [54, 35, 6], define the exponent functions[15]

$$f_1(\gamma) = \ln \sum_y P(y) \sum_x q(x) \left[\frac{P(y|x)}{P(y)}\right]^{1-\gamma}, \tag{5.18}$$

---

[14] Notation is simplified if we do not specify the dimensions of vectors; they are either implicit or specifically designated after each equation.

[15] Note $f_1(\sigma\delta) = -E_C(\sigma,\delta) - \sigma\delta R$, $f_2(\sigma,\delta) = -E_{CI}(\sigma,\delta)$ and $f_3(\sigma,\delta) = -E_I(\sigma,\delta) + \sigma\delta R$, where $E_C(\sigma,\delta)$, $E_{CI}(\sigma,\delta)$ and $E_I(\sigma,\delta)$ are defined by Viterbi and Omura [6 pp. 358].

$$f_2(\sigma,\delta) = \ln \sum_y P(y) \left\{ \sum_x q(x) \left[\frac{P(y|x)}{P(y)} \right]^{\sigma} \right\}^{\delta} \sum_{x'} q(x') \left[\frac{P(y|x')}{P(y)} \right]^{1-\sigma\delta}, \tag{5.19}$$

and

$$f_3(\sigma,\delta) = \ln \sum_y P(y) \left\{ \sum_x q(x) \left[\frac{P(y|x)}{P(y)} \right]^{\sigma} \right\}^{\delta}. \tag{5.20}$$

By applying Holder inequality [4] to the above exponent functions, we can write

$$e^{f_1(\lambda)} \leq \exp -(1-\lambda)E_o[\lambda/(1-\lambda)] = \delta_C e^{-\lambda R}, \tag{5.21}$$

$$e^{f_2(\sigma,\delta)} \leq \exp -\{(1-\sigma\delta)E_o[\sigma\delta/(1-\sigma\delta)] + \sigma\delta E_o[(1-\sigma)/\sigma]\} = \delta_C \delta_I e^{-\delta R}, \tag{5.22}$$

and

$$e^{f_3(\sigma,\delta)} \leq \exp -\sigma\delta E_o[(1-\sigma)/\sigma] = \delta_I e^{(\sigma-1)\delta R}, \tag{5.23}$$

where δ_C and δ_I are as defined in [6 pp.359].

Now, we are ready to show that the upper bound on $\overline{P_e^{Merge}(i,t_1,h_1,t_2,h_2)}$ is only a function of $t_1 + t_2 = L - j - i \geq K$ and $h_1 + h_2 \leq t_1 + t_2$.

Lemma 5.5: Given parameters i, t_1, h_1, t_2, and h_2, the ensemble average probability of the merging error event in Algorithm TAmerge is upper-bounded by

$$\overline{P_e^{Merge}(i,t_1,h_1,t_2,h_2)} \leq e^{\sigma\delta\lambda} \left(e^{nR} - 1 \right)^{\delta} \delta_C^{n(h_1+h_2)} \delta_I^{n(t_1+t_2)} e^{-\delta n K R}, \tag{5.24}$$

where $\sigma \geq 0, \delta \in [0,1], t_1 + t_2 \geq K, h_1 \geq 1, h_2 \geq 1,$ *and* $h_1 + h_2 \leq t_1 + t_2$.

Proof: According to Lemma 5.4 and letting $\delta \in [0,1]$ as in [4, 6, 36, 54], we can write

$$\overline{P_e^{Merge}(i,t_1,h_1,t_2,h_2)} \leq e^{\sigma\delta\lambda} \sum_{\mathbf{y}} \overline{P(\mathbf{y}|\mathbf{x}) \sum_{\tau=i}^{i+h_1} e^{-\sigma\delta M[\mathbf{x}_F(\tau)]} \sum_{\tau=j}^{j+h_2} e^{-\sigma\delta M[\mathbf{x}_B(\tau)]}}$$

$$\times \left\{ \sum_{(\tilde{\mathbf{x}}_{F,i}(t_1);\tilde{\mathbf{x}}_{B,j}(t_2))\in\widetilde{\mathfrak{X}}(i;j)} \overline{e^{\sigma M[\tilde{\mathbf{x}}_{F,i}(t_1)]}} \times \overline{e^{\sigma M[\tilde{\mathbf{x}}_{B,j}(t_2)]}} \right\}^{\delta}$$

$$\leq e^{\sigma\delta\lambda} \mid \widetilde{\mathfrak{X}}(i;j) \mid \sum_{\mathbf{y}} \overline{P(\mathbf{y}|\mathbf{x}) \sum_{\tau=i}^{i+h_1} e^{-\sigma\delta M[\mathbf{x}_F(\tau)]} \sum_{\tau=j}^{j+h_2} e^{-\sigma\delta M[\mathbf{x}_B(\tau)]}}$$

$$\times \left\{ \overline{e^{\sigma M[\tilde{\mathbf{x}}_{F,i}(t_1)]}} \times \overline{e^{\sigma M[\tilde{\mathbf{x}}_{B,j}(t_2)]}} \right\}^{\delta}$$

$$\leq e^{\sigma\delta\lambda}(u-1)^{\delta}u^{(t_1+t_2-K)\delta}$$

$$\times \sum_{\mathbf{y}} \overline{P(\mathbf{y}|\mathbf{x}) \sum_{\tau=i}^{i+h_1} e^{-\sigma\delta M[\mathbf{x}_F(\tau)]} \sum_{\tau=j}^{j+h_2} e^{-\sigma\delta M[\mathbf{x}_B(\tau)]}}$$

$$\times \left\{ \overline{e^{\sigma M[\tilde{\mathbf{x}}_{F,i}(t_1)]}} \right\}^{\delta} \times \left\{ \overline{e^{\sigma M[\tilde{\mathbf{x}}_{B,j}(t_2)]}} \right\}^{\delta}. \tag{5.25}$$

Obviously, (5.25) is independent of indexes i and j due to the ensemble average. For the sake of simplicity, let $i = j = 0$ in above summations, and omit subscripts i and j in (5.25). Also, note that there is no difference in the ensemble average of the forward and backward codes. Thus

$$\begin{aligned}\overline{P_e^{Merge}}(i,t_1,h_1,t_2,h_2) \leq & e^{\sigma\delta\lambda}(u-1)^{\delta}u^{(t_1+t_2-K)\delta} \\ & \times \sum_{\mathbf{y}} \overline{P(\mathbf{y}|\mathbf{x}) \sum_{\tau=0}^{h_1} e^{-\sigma\delta M[\mathbf{x}_F(\tau)]} \sum_{\tau=0}^{h_2} e^{-\sigma\delta M[\mathbf{x}_B(\tau)]}} \\ & \times \left\{ \overline{e^{\sigma M[\tilde{\mathbf{x}}_F(t_1)]}} \right\}^{\delta} \left\{ \overline{e^{\sigma M[\tilde{\mathbf{x}}_B(t_2)]}} \right\}^{\delta} \\ = & e^{\sigma\delta\lambda}(u-1)^{\delta}u^{(t_1+t_2-K)\delta}T(h_1,t_1)T(h_2,t_2),\end{aligned} \tag{5.26}$$

where

$$T(h_i,t_i) \triangleq \sum_{\mathbf{y}} \sum_{\mathbf{x}(h_i)} P[\mathbf{y}|\mathbf{x}(h_i)]q[\mathbf{x}(h_i)]e^{-\sigma\delta M[\mathbf{x}(h_i)]}$$

$$\times \left\{ \sum_{\tilde{\mathbf{x}}(t_i)} q[\tilde{\mathbf{x}}(t_i)]e^{\sigma M[\tilde{\mathbf{x}}(t_i)]} \right\}^{\delta}, \tag{5.27}$$

and where $\mathbf{x}(h_i)$ and $\tilde{\mathbf{x}}(t_i)$ are codeword segments of h_i and t_i branches, respectively. Notice the similarity between equation (5.27) and equation (6.2.9) in Viterbi and Omura [6]. Hence, we can write $T(h_i, t_i)$ as

$$T(h_i, t_i) = \begin{cases} \exp n\{(h_i - t_i)[f_1(\sigma\delta) + \sigma\delta R] + t_i f_2(\sigma, \delta)\}, & t_i \leq h_i \\ \exp n\{(t_i - h_i)[f_3(\sigma, \delta) - \sigma\delta R] + h_i f_2(\sigma, \delta)\}, & t_i \geq h_i \end{cases}. \tag{5.28}$$

Substitute (5.21) through (5.23) into (5.28), we obtain

$$T(h_i, t_i) \leq \delta_C^{nh_i} \delta_I^{nt_i} e^{-nt_i\delta R}. \tag{5.29}$$

Finally, substitute (5.29) into (5.26), and note $u = e^{nR}$ by the definition of the code rate in (2.2), to obtain (5.24). Q.E.D.

In the following, we show that for an ensemble of linear trellis codes $\overline{P_e^{Merge}}$ and $\overline{P_b^{Merge}}$ have the same exponential exponent as $\overline{P_e^{USD}}$ and $\overline{P_b^{USD}}$ but with an insignificant increase factor of $e^{\lambda\delta/(1+\delta)}$ at most.

Theorem 5.1: For an ensemble of linear trellis codes, the probability of the merging error event is upper-bounded by

$$\overline{P_e^{Merge}} \leq \begin{cases} A_7 e^{-nKR}, & 0 \leq R \leq R_{comp} \\ A_7 e^{-nKE_o(\delta)}, & R_{comp} \leq R \leq C_p \end{cases}, \tag{5.30}$$

where

$$A_7 = e^{\frac{\delta}{1+\delta}\lambda}\left(e^{nR} - 1\right) \tag{5.31}$$

is a finite constant, and for $R_{comp} \leq R \leq C_p$, $\delta \in [0, 1]$ *is the solution of*

$$R = \frac{E_o(\delta)}{\delta} \tag{5.32}$$

while for $0 \leq R \leq R_{comp}$, $\delta = 1$.

Proof: The probability of the merging error event without any condition is given by

$$\overline{P_e^{Merge}} = \sum_{\forall i,t_1,h_1,t_2,h_2} \overline{P_e^{Merge}(i,t_1,h_1,t_2,h_2)} P(i,t_1,h_1,t_2,h_2). \tag{5.33}$$

According to Lemma 5.5, given parameters i, t_1, h_1, t_2, and h_2, the ensemble average probability of the merging error event is upper-bounded by

$$\overline{P_e^{Merge}(i,t_1,h_1,t_2,h_2)} \leq e^{\sigma\delta\lambda}\left(e^{nR}-1\right)^{\delta} \delta_C^{n(h_1+h_2)} \delta_I^{n(t_1+t_2)} e^{-\delta nKR}, \tag{5.34}$$

where $\sigma \geq 0, \delta \in [0,1], t_1+t_2 \geq K, h_1 \geq 1, h_2 \geq 1,$ *and* $h_1+h_2 \leq t_1+t_2$. Moreover, one can write [6]

$$\delta_C \leq 1, \delta_I \leq 1, \qquad if\ \sigma = \frac{1}{1+\delta}\ and\ R \leq \frac{E_o(\delta)}{\delta}. \tag{5.35}$$

Thus letting $R = E_o(\delta)/\delta$ *where* $\delta \in [0,1]$, $h_1 = h_2 = 1$ and $t = K$, we can bound (5.34) by

$$\begin{aligned}\overline{P_e^{Merge}(i,t_1,h_1,t_2,h_2)} &\leq e^{\frac{\delta}{1+\delta}\lambda} \delta_C^{2n} \delta_I^{nK} \left(e^{nR}-1\right)^{\delta} e^{-\delta nKR} \\ &\leq e^{\frac{\delta}{1+\delta}\lambda}\left(e^{nR}-1\right) e^{-\delta nKR}. \end{aligned} \tag{5.36}$$

As in [6], choose $\delta = 1$ if $R \leq E_o(1) = R_{comp}$. Substituting (5.36) into (5.33), proves the above theorem. Q.E.D.

Note that in (5.30) $A_7 = A_6\left(e^{nR}-1\right)$ if $\Delta = \lambda$. Thus, A_7 has the similar effect to that of quantizing the stack in USD with $\Delta = \lambda$. Therefore, by comparing (5.30) with (5.2) of USD, one can conclude that the additional error if any due to the merging error event has the same exponent as USD.

Step 3: Now, we are ready to show that the overall error performance of Algorithm TAmerge is asymptotically the same as that of USD.

Theorem 5.2: For an ensemble of linear trellis codes, the probability of error with Algorithm TAmerge is upper-bounded by

$$\overline{P_e^{TAmerge}} < A_6 \overline{P_e^{USD}} + \overline{P_e^{Merge}}. \tag{5.37}$$

Proof: Let A, B and B' be some events where B' is the complement of event B. Thus, we can write

$$\begin{aligned} P(A) &= P(A,B) + P(A,B') \\ &= P(A|B)P(B) + P(A|B')P(B'). \end{aligned} \tag{5.38}$$

Now let A be the decoding error event and B the event that the encoder state S_{mg} is correct. Then $P(A|B') = 1$ and we can write

$$\begin{aligned} P(A) &= P(A|B)P(B) + P(B') \\ &< P(A|B) + P(B'). \end{aligned} \tag{5.39}$$

Combining (5.39) and Lemma 5.3, we complete the proof. Q.E.D.

Corollary 5.1: For an ensemble of linear trellis codes, the probability of error with Algorithm TAmerge is upper-bounded by

$$\overline{P_e^{TAmerge}} < 2A_6 \overline{P_e^{USD}}. \tag{5.40}$$

Corollary 5.1 suggests that the error probability of Algorithm TAmerge is at most twice that of a quantized USD with $\Delta = \lambda$.

Theorem 5.3: Given that the encoder state S_{mg} is incorrect, the bit error probability of Algorithm TAmerge follows the same upper-bound as that of a unidirectional quantized stack algorithm with $\Delta = \lambda$.

Proof: According to the definition in [6], the bit error probability is the expected number of bit errors in a given sequence of received bits normalized by the total number of bits in the sequence. Thus for a trellis code with block length L and $\log_2 u$ information bits per branch,

$$\overline{P_b^{TAmerge}(S_{mg}\ is\ incorrect)} = \frac{\overline{E[N_b]}}{(L-m)\log_2 u}, \tag{5.41}$$

where N_b is the total number of bit errors in the L-branch code sequence given that the encoder state S_{mg} is incorrect.

Now, let $n_b(\tau)$ denote the expected number of bit errors caused by an incorrect path diverging at forward tree level τ. As indicated in Figure 5.2, N_b can be divided into three parts i.e.,

$$E[N_b] \leq \sum_{\tau=0}^{i-m} E[n_b(\tau)] + E[n_b(i,t_1,h_1,t_2,h_2)] + \sum_{\tau=L-j}^{L-m} E[n_b(\tau)], \tag{5.42}$$

where the first and third terms correspond to the total number of bit errors outside the merging error event, and the second term represents the number of bit errors due to the merging error event with parameters i, t_1, h_1, t_2, h_2. The inequality in (5.42) follows from the fact that bit error sequences outside the merging error event may overlap [6].

As in [6], we can write

$$E[n_b(i,t_1,h_1,t_2,h_2)] \leq (t_1+t_2-K+1)\log_2(u) P_e^{Merge}(i,t_1,h_1,t_2,h_2), \tag{5.43}$$

since $(t_1 + t_2 - K + 1)\log_2(u)$ is the number of bit errors due to the merging error event. According to Lemma 5.5, $\overline{E[n_b(i, t_1, h_1, t_2, h_2)]}$ is only related with parameters $t \triangleq t_1 + t_2$ and $h \triangleq h_1 + h_2$, where $t \geq K$ and $0 < h \leq t$. Moreover, since $\overline{P_e^{Merge}(t, h)}$ follows the same upper bound as that of the proto-error event with a bias λ, we can write

$$\overline{E[N_b]} \leq e^{\sigma\delta\lambda} \sum_{\tau=0}^{L-m} \overline{E[n_b(\tau)]} \tag{5.44}$$

where $\sigma \geq 0 \quad and \quad \delta \in [0, 1]$.

Substitute (5.44) into (5.41), we can write

$$\overline{P_b^{TAmerge}(S_{mg}\ is\ incorrect)} \leq \frac{e^{\sigma\delta\lambda}}{(L-m)\log_2 u} \sum_{\tau=0}^{L-m} \overline{E[n_b(\tau)]}. \tag{5.45}$$

Notice that aside from the factor $e^{\sigma\delta\lambda}$, the right side of (5.45) is the upper bound of the bit error probability of a unidirectional unquantized stack algorithm [6 pp. 312]. As discussed earlier, $e^{\sigma\delta\lambda}$ has the same effect as using a unidirectional quantized stack algorithm with $\Delta = \lambda$. Q.E.D.

Similar to the proof of Theorem 5.2, one can write

$$\begin{aligned} P(A) &= P(A|B)P(B) + P(A|B')P(B') \\ &= P(A|B) + P(B')\left[P(A|B') - P(A|B)\right], \end{aligned} \tag{5.46}$$

where B still denotes the event that the encoder state S_{mg} is correct. Now, let *P(A)* be the bit error probability with Algorithm TAmerge. Then, $P(A|B)$ is the bit error rate of Algorithm TAmerge given that the encoder state S_{mg} is correct, and $P(A|B')$ the bit error rate given that S_{mg} is incorrect.

In (5.46), $P(B')$ is the probability that S_{mg} is incorrect, which is upper-bounded by Theorem 5.1. Furthermore, According to Lemma 5.3, $P(A|B)$ follows the same upper

bound as for a unidirectional quantized stack algorithm with $\Delta = \lambda$. Similarly, according to Theorem 5.3, $P(A|B')$ also follows the same upper bound as for a unidirectional quantized stack algorithm with $\Delta = \lambda$. Although $P(A|B) \neq P(A|B')$ in general, we can still conclude that the second term in (5.46) compared to the first term is asymptotically unimportant. Thus, the following corollary follows.

Corollary 5.2: For an ensemble of linear trellis codes, the bit error probability of Algorithm TAmerge is asymptotically upper-bounded by

$$\overline{P_b^{TAmerge}} \lessapprox A_6 \overline{P_b^{USD}}. \tag{5.47}$$

According to Corollaries 5.1 and 5.2, one can conclude that the error performance of Algorithm TAmerge is asymptotically the same as that of USD. More accurately, the error performance of Algorithm TAmerge using unquantized stacks is similar to that of a unidirectional quantized stack algorithm with $\Delta = \lambda$.

The error performance of Algorithm TTmerge using unquantized stacks is better than that of Algorithm TAmerge since the metric of its merged path is higher than or equal to the merged path metric in Algorithm TAmerge. This is because the merging test in Algorithm TTmerge is only performed among paths in the highest forward and backward substacks. The error performance of Algorithm HTTmerge is close to that of Algorithm TTmerge when the number of required matching information symbols m_h is close to the code memory length m. On the other hand, when that number is zero (or close to zero), Algorithm HTTmerge behaves like Algorithm TAmeet. Thus, one can say that the error performance of Algorithm HTTmerge, is upper-bounded by that of Algorithm

TAmeet and lower bounded by that of Algorithm TTmerge. Furthermore, the number of computations needed by Algorithm HTTmerge decreases as m_h decreases. Hence, Algorithm HTTmerge can easily provide a trade-off between the error performance and computational effort by controlling the number of matching information symbols m_h during the merging test.

5.3 Error Performance of Algorithm TAmeet

According to Property 2.1 in Chapter 2, the distance spectra of the forward and backward codes of a specific time-invariant convolutional code are exactly the same. Suppose we let forward and backward decoders decode to the ends of their trees independently. Then, the error performance of the backward decoder is expected to be almost the same as that of the forward decoder. In Algorithm TAmeet, however, decoding stops whenever the search areas of forward and backward decoders meet. In other words, the searching areas of the forward and backward decoders will never overlap. The benefits are that the computational effort is reduced to its minimum among our proposed BSD algorithms and the extra merging test which might be time consuming is not required. However, the disadvantage is that the decoded forward and backward paths may not merge (agree) with each other. Thus, Algorithm TAmeet may make additional decoding errors near its meeting point. Hence, the error performance of Algorithm TAmeet is obviously worse than that of any other BSD algorithms with a merging test.

In Algorithm TAmeet, both decoded forward and backward paths retreat about $m / 2$ branches from their end nodes.[16] Thus, the length of the retreated branches, $m / 2$, can

[16] Here, we ignore the Diophantine constraint, i.e., treat $m / 2$ as a integer.

be viewed as the length of the backsearch limit for the last decoded information symbols by forward and backward decoders. For the rest of the decoded information symbols, the backsearch limit is clearly greater than m / 2. Hence, the decoded information symbols near the beginning and end of each block are more reliable than those near the meeting point. Thus, we can upper-bound the bit error probability of Algorithm TAmeet by that of USD with a backsearch limit equal to $\lfloor m/2 \rfloor$. Clearly, the above upper bound indicates that the BER of Algorithm TAmeet is essentially not related with the block length L. Furthermore, the above argument suggests that a long block length is preferred in order to reduce the actual BER with Algorithm TAmeet.

For an ensemble of linear trellis codes, Zigangirov [55] provided an upper bound on the bit error probability of USD with backsearch limit τ that is greater than the constraint length K. When the backsearch limit τ is equal to or less than the constraint length K, it is known [12, 56, 57] that the probability of error satisfies the random coding bound for block codes. Therefore, we can conclude that the random block coding exponent applies to the bit error probability of Algorithm TAmeet, and thus we can write

$$\overline{P_b^{TAmeet}} \leq A_8 e^{-\lfloor m/2 \rfloor n[E_o(\rho_r)-\rho_r R]}. \tag{5.48}$$

From the above discussion, one can see that the only reason the error performance of Algorithm TAmeet is worse than that of USD is because it does not require the decoded forward and backward paths to merge with each other.

5.4 Numerical and Simulation Results

In order to verify the analysis results, lengthy computer simulations were conducted.

Since decoding errors rarely occur with a long memory code, a short memory ($m = 10$) SABODP code was used. Similar to Chapter 4, the code rate was chosen to be equal to 0.5, a BSC channel was used, and metrics were scaled into integers in the simulations. Moreover, all algorithms were run under strictly identical conditions (i.e., using the same code and noise sequences) in order to make the results comparable.

Results are shown in Tables 5.1 and 5.2 for $\rho_r = 1.1$ and $\rho_r = 0.68$, respectively. In these simulations, the one branch metric drop $\lambda = 16$ and 14 for $\rho_r = 1.1$ and $\rho_r = 0.68$, respectively. 50,000 blocks of length $L = 400$ were simulated for each algorithm in Table 5.1, while 5,000 blocks of length $L = 200$ were simulated for each algorithm in Table 5.2. In addition, the computational limit for each algorithm was selected large enough to make the erasure probability negligible.

As expected, Tables 5.1 and 5.2 show that the error performance (both bit and block error rate) of Algorithm TAmerge is similar to that of USD. More precisely, the BER of Algorithm TAmerge with unquantized stacks is very close to that of a unidirectional quantized stack algorithm with a substack spacing $\Delta = \lambda$. Moreover, as Corollary 5.1 suggested, the block error probability of Algorithm TAmerge with unquantized stacks is upper-bounded by twice that of unidirectional quantized stack algorithm with $\Delta = \lambda$. Tables 5.1 and 5.2 suggest that the error performance of Algorithm TTmerge with $\Delta = 1$ is very close to that of a unidirectional unquantized stack algorithm. Table 5.1 also shows that the error performance of Algorithm TTmerge with a small Δ, i.e., $\Delta < \lambda$, is slightly better than that of Algorithm TAmerge using unquantized stacks. One can also notice from Tables 5.1 and 5.2 that the error performance of Algorithm TTmerge with $\Delta = \lambda$ is close to that of Algorithm TAmerge with $\Delta = 1$. Moreover, when $m_h = (m - 1)$

Table 5.1 Error performance comparison of different algorithms for the case $\rho_r = 1.1$, i.e., $R < R_{comp}$

SABODP Code m=10 ρ_r=1.1,L=400,5×10^4 runs	BER	Error blocks	Erasure blocks	Max. comp.	Ave. comp./br.
USD (unquantized stack algorithm Δ = 1)	1.86e-4	261	0	163045	1.824
USD (quantized stack algorithm Δ = 16)	5.73e-4	649	4	2e5	3.383
Algorithm TAmeet (Δ = 1)	8.09e-3	14552	0	9278	1.274
Algorithm TAmerge (Δ = 1)	6.08e-4	962	0	9364	1.310
Algorithm TTmerge (Δ = 1)	1.39e-4	340	0	17567	1.397
Algorithm TTmerge (Δ = 7)	2.79e-4	496	0	16877	1.403
Algorithm TTmerge (Δ = 16)	8.07e-4	1188	0	23383	1.876
Algorithm HTTmerge (m_h = m – 1, Δ = 1)	3.52e-4	672	0	9562	1.336
Algorithm HTTmerge (m_h = m – 2, Δ = 1)	6.57e-4	1204	0	9562	1.322

and $\Delta = 1$, the error performance of Algorithm HTTmerge is similar to that of Algorithm TTmerge. The trade-off between the error performance and computational effort that can be provided by Algorithm HTTmerge is clearly apparent in Tables 5.1 and 5.2.

The average and maximum number of computations, and the number of blocks remained undecoded (erasure blocks) with different algorithms are listed in Tables 5.1 and 5.2. As it can be seen, the maximum number of computations and the average number of computations necessary to decode a block using the proposed BSD are much less than those needed by USD. Moreover, the simulation results suggest that a large

Table 5.2 Error performance comparison of different algorithms for the case $\rho_r = 0.68$, i.e., $R > R_{comp}$

SABODP Code m=10 ρ_r=0.68,L=200,5×10^3 runs	BER	Error blocks	Erasure blocks	Max. comp.	Avg. comp./br.
USD (unquantized stack algorithm Δ=1)	1.99e-2	413	4	2e5	18.434
USD (quantized stack algorithm with Δ=14)	3.88e-2	770	6	2e5	32.514
Algorithm TAmeet (Δ = 1)	4.88e-2	3047	0	2545	1.559
Algorithm TAmerge (Δ = 1)	1.58e-2	591	0	3095	1.898
Algorithm TTmerge (Δ=1)	9.27e-3	300	3	2e4	3.026
Algorithm TTmerge (Δ=14)	2.91e-2	871	0	19107	4.144
Algorithm HTTmerge ($m_h = m$ - 1, Δ=1)	1.39e-2	515	0	6758	2.108
Algorithm HTTmerge ($m_h = m$ - 2, Δ=1)	1.70e-2	716	0	4258	1.938

Δ in Algorithm TTmerge does not benefit both the error performance and average computations. Thus, a small value of Δ seems to be a better choice for the overall performance of Algorithm TTmerge as well as Algorithm HTTmerge.

Finally, we examine the effect of block length L on the error performance of the proposed BSD algorithms. Simulation results of the error performance of Algorithm TAmeet and Algorithm TAmerge are shown in Table 5.3 and Table 5.4, respectively. It can be noticed from Table 5.3 that the BER of Algorithm TAmeet improves slightly as L increases, which confirms the findings in section 5.3. The BER of all algorithms with a merging test is not sensitive to L. However, as expected, the block error probability

Table 5.3 Error performance comparison of Algorithm TAmeet (Δ = *1*) for different block length L

SABODP Code m=10 ρ_r=1.1, 5×10^4 runs	L=100	L=200	L=300	L=400	L=500
Bit error rate (BER)	1.03e-2	9.84e-3	8.90e-3	8.09e-3	7.56e-3
# of error blocks	8869	11944	13482	14552	15348
# of erasure blocks	0	0	0	0	0
Max. # of comp.	583	2805	5021	9278	11200
Ave. comp. / branch	1.142	1.202	1.244	1.274	1.302

Table 5.4 Error performance comparison of Algorithm TAmerge (Δ = *1*) for different block length L

SABODP Code m=10 ρ_r=1.1, 5×10^4 runs	L=100	L=200	L=300	L=400	L=500
Bit error rate (BER)	3.07e-4	4.62e-4	5.50e-4	6.08e-4	6.52e-4
# of error blocks	151	409	664	962	1256
# of erasure blocks	0	0	0	0	0
Max. # of comp.	1133	3074	6465	9364	11362
Ave. comp. / branch	1.189	1.247	1.283	1.310	1.335

increases with increasing L, for all the algorithms under study.

Tables 5.3 and 5.4 indicate that Algorithm TAmerge and Algorithm TAmeet have basically the same maximum and average number of computations when the code rate is lower than R_{comp}. This argument is especially true when L is not too small. Thus, the additional computations introduced by the merging test are asymptotically insignificant, and are worth the improvements in error performance. Moreover, the maximum and average computations is much smaller than those of USD for all values of L. In other words, the number of erasure blocks with the proposed BSD is much smaller than those of USD under the same maximum number of computations.

Chapter 6
Bidirectional Multiple Stack Algorithm

The main disadvantage of sequential decoding algorithms is their inherent overflow problem. The number of computations in sequential decoding is a random variable with a Pareto distribution. Hence, there is always a small fraction of blocks which require an infeasible number of computations and thus cannot be completely decoded. This ratio of undecodable blocks is often called the erasure probability P_{er}. Several known techniques have been proposed to alleviate the overflow problem of sequential decoding [14–16, 18, 19, 58].

In Chapter 3, we proposed BSD algorithms which were shown to significantly reduce the erasure probability. The proposed BSD algorithms improve the computational performance by approximately doubling the Pareto exponent of conventional sequential decoding techniques. However, the decoding effort is still a random variable with a Pareto distribution, and the overflow or erasure problem is still not eliminated.

Chevillat and Costello [18] proposed an interesting decoding algorithm, called *multiple stack algorithm* (MSA), that allows erasure-free decoding. The basic idea is to make use of higher order stacks for blocks that require an excessive number of computations to be decoded. However, to achieve a good error performance with the MSA, a large amount of memory is needed.

In this chapter we propose to modify the MSA to accommodate bidirectional tree search [29], as described in Chapter 3. We refer to this decoding algorithm as *bidirectional multiple stack algorithm* (BMSA). It is found by analysis as well as computer simulations

that the use of bidirectional decoding with the MSA improves the performance in terms of computational effort, memory requirements and decoding errors.

First, the basic elements of the conventional MSA are reviewed briefly. The BMSA is then discussed in detail and its performance is evaluated by analysis and computer simulations. Finally, different decoding algorithms, including the MSA, the new BMSA and the Viterbi algorithm, are compared on the basis of their memory requirements, bit error performances and computational efforts.

6.1 The Multiple Stack Algorithm (MSA)

The multiple stack algorithm (MSA) eliminates the overflow problem entirely at the expense of a substantial increase in stack and buffer storage requirements. It is a modification of the stack algorithm which employs several stacks to alleviate the problem of stack overflow. The basic functions of the MSA are illustrated in Figure 6.1. Initially, the MSA decoder operates as a conventional stack algorithm. If a terminal node in the tree is reached before the first (primary) stack with size Z_1^{MSA} is filled, decoding is complete, and the decoded sequence is identical to that of a conventional stack algorithm. However, if the first stack is filled and an extensive search is needed, the top T nodes of the first stack are transferred to the second stack and decoding resumes using *only* the T transferred nodes. If the end of the tree is reached before this stack is full, the terminal node is accepted as a *tentative* decision and is stored in a tentative register. The decoder clears the second stack, returns to the first stack, and attempts to find another tentative decision. The new tentative decision is compared to the preceding one, and the better one is retained and stored in the register. The process of transferring T nodes to another stack

is not limited to the second stack. Additional stacks are formed as needed. If a stack fills up before the end of decoding, T nodes are transferred to a higher order (secondary) stack until a tentative decision is made. For simplicity, we assume that all secondary stacks have the same stack size, i.e., $Z_i^{MSA} = Z^{MSA}, i = 2, 3, \cdots$. The MSA terminates decoding when it reaches the end of the tree in the first stack, or if a predetermined computational limit C_{lim} is reached while the search is in progress. In both cases, the best tentative decision is accepted as the final decision.

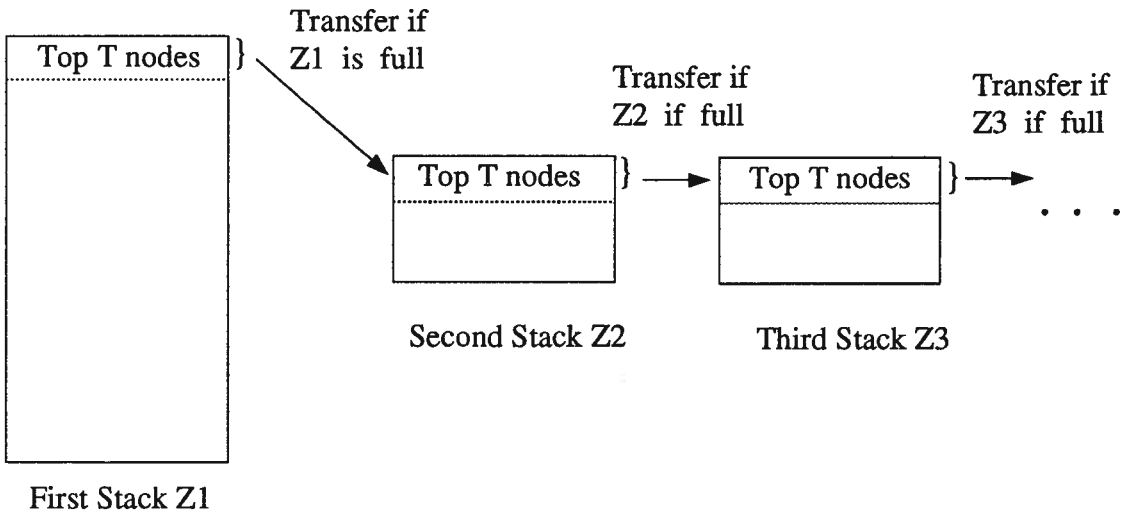


Figure 6.1 Illustration of the multiple stack algorithm.

The major difference between the single stack algorithm and the multi-stack algorithm is well described by their names. The single stack algorithm stops the decoding process when the stack is full and erases the undecoded block. The MSA continues decoding in secondary stacks to reach a decision. When a tentative decision is made in one of the secondary stacks, and the computational limit C_{lim} has not been reached, the MSA

returns to previous stacks to refine the tentative decision. Decoding stops either after satisfactorily achieving first-stack decoding or by reaching the computational limit.

In [18], the block error probability of the MSA, P_e^{MSA}, is upper-bounded by

$$P_e^{MSA} \leq P_e^{USD} + P_1^{USD} \tag{6.1}$$

where P_e^{USD} is the block error probability of the single stack algorithm with an infinite stack size, and P_1^{USD} is the probability of the first stack overflow. It is shown in [18] that

$$P_1^{USD} \approx A_9 \left(Z_1^{MSA} - 1 \right)^{-\rho_r}, \tag{6.2}$$

where A_9 is a constant and ρ_r is the Pareto exponent of USD as defined in Chapter 2. Similarly, the bit error probability of the MSA can be upper-bounded by

$$P_b^{MSA} \leq P_b^{USD} + P_1^{USD}. \tag{6.3}$$

As pointed out by Viterbi and Omura [6], the block error probability is not a reasonable performance measure, and ultimately the most useful measure is the bit error probability. Moreover, as suggested in Chapter 5, the bit error probability of sequential decoding, including our BSD, is independent of the block length L. Hence, in this chapter, the bit error probability is the criteria used to measure the error performance of various algorithms.

According to (6.3), in order to achieve a low bit error probability with the MSA, both P_b^{USD} and P_1^{USD} must be very small. It is trivial to make P_b^{USD} very small by employing a long memory convolutional code. The objective is then to make P_1^{USD} as small as

the desirable error performance. However, to make P_1^{USD} very small, say comparable to P_b^{USD} with a long memory code, the size of the first stack must be chosen very large. As indicated in (6.2), P_1^{USD} only decreases in a Pareto function with an increase in the first stack size Z_1^{MSA}. This means that a substantial first stack size Z_1^{MSA} is needed to make P_1^{USD} relatively small. As it will shown later, the use of the proposed BSD is an alternative to reduce the first stack overflow probability for a given first stack size.

Chevillat and Costello [18] investigated the distribution of block computations for the first tentative decision of the MSA. They showed that it is Pareto before the first stack is filled, decreasing exponentially thereafter. They also analyzed the computational distribution for the final decision and gave the following upper bound:

$$P\left(C_L^{MSA} > N\right) \leq \begin{cases} A_9 N^{-\rho}, & N \leq Z_1^{MSA} - 1 \\ P_1^{USD}, & Z_1^{MSA} \leq N \leq C_{lim} \\ 0, & N > C_{lim}, \end{cases} \tag{6.4}$$

where C_L^{MSA} is the total number of computations required for the final decision and $\rho < \rho_r$. However, no experimental results on the final computational distribution are reported in the literature. Since the number of computations for the final decision represents the overall computational performance of the decoder, in the rest of this dissertation, the computational distribution for the final decision will be the focal point.

In all computer simulations reported in this chapter, antipodal signaling over an *additive white Gaussian noise* (AWGN) channel with hard decision at the output of a coherent demodulator is assumed. Figure 6.2 shows empirical results of the computational distribution for the final decision of the MSA with parameters $L = 128$, $T = 3$, $Z^{MSA} = 11$, $\Delta = 7$, $C_{lim} = 2000$, $E_b / N_o = 4.58$ *dB*, corresponding to $\rho_r = 1$, and $Z_1^{MSA} = 4L, 5L, 6L$. The code used is an $m = 7$ SABODP code of rate $r = 1 / 2$ (see Table 2.3).

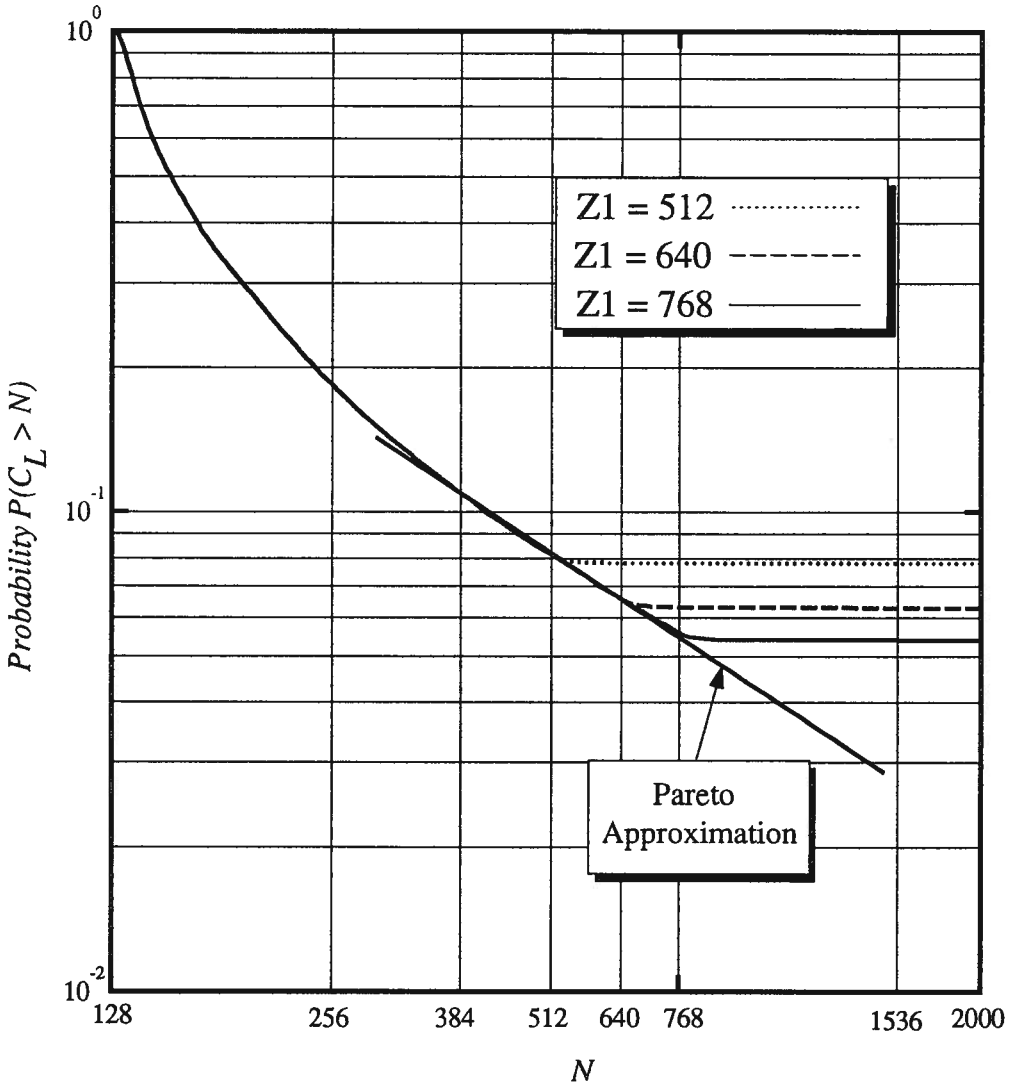


Figure 6.2 Empirical computational distribution for the final decision of the MSA.

As indicated by (6.4), it can be seen from the figure that the computational distribution for the final decision can be divided into three different segments. Before the first stack becomes full (i.e., $N \leq Z_1^{MSA}$), the computational effort follows a Pareto distribution as in the case of a conventional infinite stack size algorithm. After the first stack fills up (i.e., $Z_1^{MSA} \leq N \leq C_{lim}$), the distribution quickly reaches a computational floor.

Finally, after the computational limit (i.e. $N > C_{lim}$) is reached, it becomes zero.

The behavior of the first and last segments of the final computational distribution are obvious. The behavior of the second segment (i.e., $Z_1^{MSA} \leq N \leq C_{lim}$) can be explained as follows. Whenever the first stack fills up, the algorithm transfers T nodes to the second stack. The transferred nodes may either include the correct one, or they can all be incorrect nodes. In the first case, the correct node is amongst the transferred nodes in the secondary stacks. As described before, the decoding process is stopped only if a tentative decision is reached in the first stack, or if the computational limit is reached. Since the current decoding path is only accepted as a tentative decision, the MSA will not stop decoding even after the correct final node is located in a secondary stack. Let's assume that the correct final node is in the i-th $(i>1)$ stack. The MSA empties the i-th stack and returns to the $(i-1)$-th stack. Since the correct node is removed from the i-th stack, there is no correct node in any of the stacks.[17] It is therefore unlikely to return to the first stack and reach a tentative decision. In other words, with a high probability, decoding will not stop until the computational limit is reached.

Next, assume that all the transferred T nodes are incorrect. Since there is no correct node in the second stack, it will take many computations to find one tentative decision and it is therefore very likely to require higher order stacks. Assume that the tentative decision is made in the i-th $(i>1)$ stack. The MSA empties the i-th stack and returns to the $(i-1)$-th stack and tries to find another decision. After many computations, if the computational limit is not yet reached, the MSA will finally return to the first stack. However, the MSA can only extend T nodes before the first stack fills up again. This

[17] Here, we ignore the fact that an incorrect path may merge with the correct path since this only occurs with a small probability.

implies that there is a very small chance that a tentative decision can be reached within the T extensions in the first stack. If the first stack fills up again, another T nodes will be transferred to the second stack. Hence, if the correct node is not transferred into the second stack, during the first time, it may be transferred into the second stack during the second, third, ···, etc. times. In summary, whenever the first stack fills up, the MSA will not, most of the time, stop the decoding process until the computational limit is reached.

6.2 Bidirectional Multiple Stack Algorithm (BMSA)

A *bidirectional multiple stack algorithm* (BMSA) is constructed in this section, which combines the ideas of BSD and the use of multiple stacks. Any of the BSD algorithms described in Chapter 3 can be used. However, Algorithm TTmerge is chosen due to its ease of implementation and good error and computational performances.

The BMSA based on Algorithm TTmerge consists of a number of forward and backward finite-size memory stacks. We assume that the pairs of forward and backward stacks are of the same size, denoted by Z_i^{BMSA}. Initially, decoding with the BMSA is exactly the same as with Algorithm TTmerge. If decoding is terminated before the two stacks fill up, the decoded sequence is the same as that with Algorithm TTmerge. However, should one of the stacks (FS or BS) fill up, like in the conventional MSA, the T top nodes in that stack are transferred to a secondary stack, and decoding resumes using only the T transferred nodes. Once a merged path is found,[18] that path is taken as a tentative decision, and is stored in a register. The secondary stack is then cleared, and decoding continues on the primary stacks. If, however, one of the stacks fills up

[18] For simplicity and ease of implementation, the merging test is only conducted between nodes of top non-empty substacks of the current pair of forward and backward stacks.

before a decision is reached, the T top nodes in that stack are transferred to yet another secondary stack, and decoding resumes once again using only the T transferred nodes. This process is continued, that is secondary stacks are used as needed. Once a tentative decision is found in a given pair of forward and backward stacks, these two stacks are cleared, and decoding resumes on the predecessor stacks in the attempt to refine that tentative decision. Decoding is stopped whenever a decision is reached in the pair of primary stacks or when a predetermined computation limit C_{lim} is reached. In any event, the best tentative decision is accepted as the decoded path. A detailed description of the BMSA is given below. Let S_F denote the current number of the forward stack, and S_B the current number of the backward stack. The steps are as follow:

Step 0: Place the root nodes of forward and backward trees in the 1st FS and the 1st BS, respectively. Set $S_F = S_B = 1$.

Step 1: Compute the metrics of all successors of any node from the highest non-empty substack in the S_Fth FS, remove this node from the S_Fth FS, and enter the new nodes in their proper substack in the S_Fth FS.

Step 2: If $l_F + l_B \geq L$, check all paths in the highest non-empty substack in the S_Bth BS with all paths in the highest non-empty substack in the S_Fth FS. If one or more merging paths is found, select the one, including the one already in the tentative register if any, with the highest cumulative metric as a tentative decision and go to step 13. Otherwise, go to the next step.

Step 3: If the S_Fth FS is full, transfer T nodes from the highest non-empty substack to the $(S_F + 1)$th FS,[19] and set $S_F = S_F + 1$. Go to the next step.

[19] If the highest non-empty substack contains less than T nodes, pick the rest from the next highest non-empty substack.

Step 4: Check if the node to be extended in the highest non-empty substack in the S_Bth BS is a terminal node of the backward tree. If yes, compare that path with the one in the tentative register, if any, select the better one, and go to the next step. Otherwise, go to step 6.

Step 5: If $S_B = 1$, stop decoding and take the path in the tentative register as the decoded path. Otherwise, go to step 15.

Step 6: If the number of computations is equal to C_{lim}, stop decoding and take the path in the tentative register as the decoded path. Otherwise, go the next step.

Step 7: Compute the metrics of all successors of any node from the highest non-empty substack in the S_Bth BS, remove this node from the stack and enter the new nodes in their proper substack in the S_Bth BS.

Step 8: If $l_F + l_B \geq L$, check all paths in the highest non-empty substack in the S_Fth FS with all paths in the highest non-empty substack in the S_Bth BS. If one or more merging paths is found, select the one, including the one already in the tentative register, if any, with the highest cumulative metric as a tentative decision and go to step 16. Otherwise, go to the next step.

Step 9: If the S_Bth BS is full, transfer T nodes from the highest non-empty substack to the $(S_B + 1)$th BS, and set $S_B = S_B + 1$. Go to the next step.

Step 10: Check if the node to be extended in the highest non-empty substack in the S_Fth FS is a terminal node of the forward tree. If yes, compare that path with the one in the tentative register, if any, select the better one, and go to the next step. Otherwise, go to step 12.

Step 11: If $S_F = 1$, stop decoding and take the path in the tentative register as the decoded path. Otherwise, go to step 18.

Step 12: If the number of computations is equal to C_{lim}, stop decoding and take the path in the tentative register as the decoded path. Otherwise, go to step 1.

Step 13: If $S_F = 1$ and $S_B = 1$, stop decoding and take the path in the tentative register as the decoded path. Otherwise, go to the next step.

Step 14: If $S_F > 1$, empty the S_Fth FS, set $S_F = S_F - 1$, and go to the next step. Otherwise, remove the merged node from the 1st FS, and go to the next step.

Step 15: If $S_B > 1$, empty the S_Bth BS, set $S_B = S_B - 1$, and go to step 6. Otherwise, remove the merged node from the 1st BS, and go to step 6.

Step 16: If $S_F = 1$ and $S_B = 1$, stop decoding and take the path in the tentative register as the decoded path. Otherwise, go to the next step.

Step 17: If $S_B > 1$, empty the S_Bth BS, set $S_B = S_B - 1$, and go to the next step. Otherwise, remove the merged node from the 1st BS, and go to the next step.

Step 18: If $S_F > 1$, empty the S_Fth FS, set $S_F = S_F - 1$, and go to step 12. Otherwise, remove the merged node from the 1st FS, and go to step 12.

6.3 Computational Properties of the BMSA

Define the total number of computations C_L^{BMSA} needed to decode one block by the BMSA as the sum of the computations performed by forward and backward decoders. Let $Z_1 = 2Z_1^{BMSA} = Z_1^{MSA}$, and $Z = 2Z_i^{BMSA} = Z_i^{MSA}$ for i=2,3,$\cdots$. As in the MSA, the size of the 1st stack Z_1^{BMSA} should be made large enough so that only the very noisy blocks, which are potential erasures in conventional BSD, use higher order

forward and backward stacks. On the other hand, the size of the secondary stacks Z_i^{BMSA} should be made substantially small for the BMSA to quickly advance through forward and backward trees, and find a reasonably good tentative decision.

For $T=1$, at most L pairs of stacks are formed before reaching the first tentative decision. Once a tentative decision is made, the decoder returns to previous stacks to refine the tentative decision. The BMSA is therefore erasure-free if one or more tentative decisions are made before reaching the computational limit C_{lim}. In the worst case scenario, where forward and backward decoders do not merge before the final forward or backward node is reached, the number of computations needed by the BMSA to guarantee erasure-free decoding would be twice the critical number of computations for the MSA, C_{crit}, which is given by [18]

$$C_{crit} = \sum_{i=1}^{L-m-1} \left(\frac{Z_i}{2} - 1 \right) + 2m. \tag{6.5}$$

Therefore, to ensure erasure-free decoding using the BMSA with $T=1$, we must have

$$C_{lim} \geq 2 \times C_{crit}. \tag{6.6}$$

Although the above criteria for selecting C_{lim} is extremely conservative, it offers an easy and safe design rule.

Clearly, the computational properties of the first tentative decision of the BMSA are similar to those of the MSA. Thus, following [18] and using the approximation on the computational distribution of Algorithm TTmerge (see Chapter 4), one can write

Property 6.1: The first stack overflow probability for the BMSA, P_1^{BSD}, is given by the erasure probability of Algorithm TTmerge with primary stack size Z_1. That is,

$$P_1^{BSD} = P\left(C_L^{BSD} > Z_1 - 1\right) \approx A_{10}(Z_1 - 1)^{-2\rho_r}, \tag{6.7}$$

where A_{10} is a constant independent of Z_1, and ρ_r is the Pareto exponent.

From (6.7) and (6.2), it can be noticed that P_1^{BSD} is much smaller than P_1^{USD} for a given Z_1. In other words, to achieve a given overflow probability, the required Z_1 with the BMSA is much smaller than that with the MSA. To quantify this reduction in stack size, let $Z_1^{MSA}(P_1, E_b/N_o)$, and $Z_1^{BMSA}(P_1, E_b/N_o)$ denote respectively, the stack sizes needed with the MSA and the BMSA to achieve a certain first stack overflow probability P_1, at a given signal to noise ratio E_b / N_o. The first (primary) stack size saving ratio[20] can be expressed by

$$S_s \triangleq \frac{Z_1^{MSA}(P_1, E_b/N_o)}{Z_1^{BMSA}(P_1, E_b/N_o)}, \tag{6.8}$$

which can be simplified to

$$\begin{aligned} S_s &\approx \frac{(A_9/P_1)^{1/\rho_r}}{(A_{10}/P_1)^{1/2\rho_r}} \\ &= \left(\frac{A_9}{\sqrt{A_{10}}}\right)^{\frac{1}{\rho_r}} P_1^{-1/2\rho_r} \\ &= A_{11} P_1^{-1/2\rho_r}, \end{aligned} \tag{6.9}$$

where A_{11} is a constant that depends on the channel condition and the decoder used, which can be determined by simulation. For comparison purpose, from now on, we let $A_{11} = 1$.

Figure 6.3 shows a plot of S_s as a function of P_1 for different values of Pareto exponent ρ_r. As shown in this figure, the smaller the required first stack overflow probability P_1 is, the greater is the saving in Z_1. Moreover, the saving in the primary stack size increases as the Pareto exponent (or E_b/N_o) decreases. For instance, when ρ_r

[20] The first stack size saving ratio represents approximately the saving in the total memory size since the first stack is much larger than the secondary stacks.

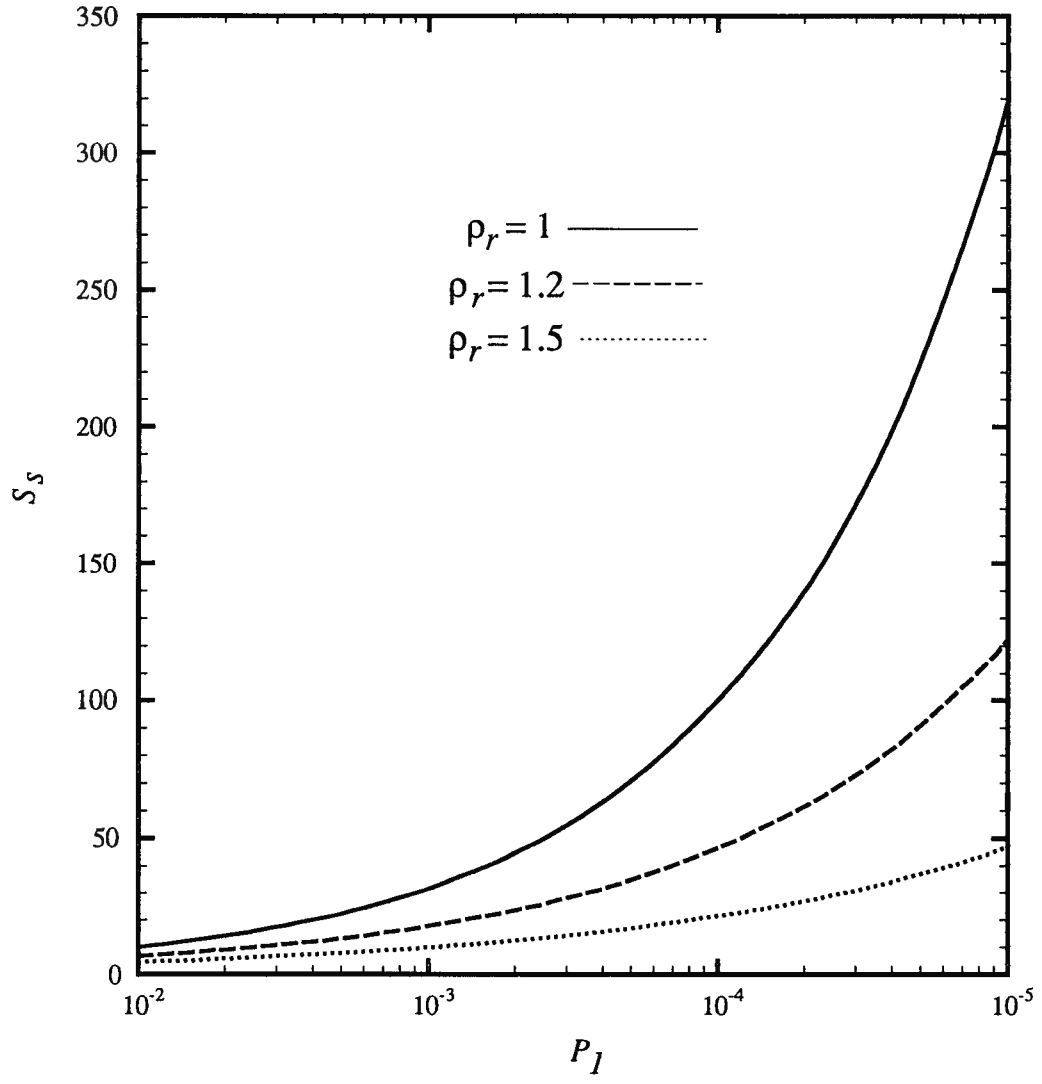


Figure 6.3 The first stack saving as a function of P_1.

$= 1$ (E_b/N_0=4.58 dB), and $P_1 = 10^{-3}$, the primary stack size saving is about 32 times. For the same P_1, $S_s = 18$ for $\rho_r = 1.2$ (E_b/N_0=5.0 dB).

As for the total number of computations for the final decision of the BMSA, following [18], we can write the following property.

Property 6.2: The final computational distribution of the BMSA is upper-bounded by

$$P\left(C_L^{BMSA} > N\right) \leq \begin{cases} A_{10}N^{-2\rho}, & N \leq Z_1 - 1 \\ P_1^{BSD}, & Z_1 \leq N \leq C_{lim} \\ 0, & elsewhere, \end{cases} \qquad (6.10)$$

where the Pareto exponent $\rho < \rho_r$, *and* $Z_1 = 2Z_1^{BMSA}$.

The computational distribution of the final decision for the BMSA reaches a floor beyond a certain threshold like that of the conventional MSA. However, since $P_1^{BSD} << P_1^{USD}$ for a given primary stack size Z_1, when compared to the MSA, the computational floor is substantially reduced with the BMSA.

In order to verify the above properties, computer simulations were performed using 50,000 blocks each of length $L = 128$. The signal to noise ratio E_b / N_o was fixed to *4.58 dB*, corresponding to a Pareto exponent of $\rho_r = 1$. The code used is a SABODP, non-systematic rate $r = 1 / 2$ memory $m = 7$ code (see Table 2.3). The stacks in both the MSA and the BMSA were quantized with $\Delta = 7$.

Figure 6.4 shows the results of the final computational distributions of the MSA and the BMSA for different values of the first stack size Z_1 with parameters $T = 3$, $Z = 11$, and $C_{lim} = 2000$. The Pareto approximations are plotted as solid straight lines on the figure. As it can be seen from this figure, the simulation results are in total agreement with the analytical properties discussed above.

Figure 6.5 shows the results of the final computational distributions of the MSA and the BMSA for different values of T. The different parameters used are: $Z_1 = 6L$, $Z = 11$ and $C_{lim}=2000$. This figure suggests that the first stack overflow probability and the final number of computations are essentially independent of parameter T, as it was predicted by analysis (see (6.7) and (6.10)).

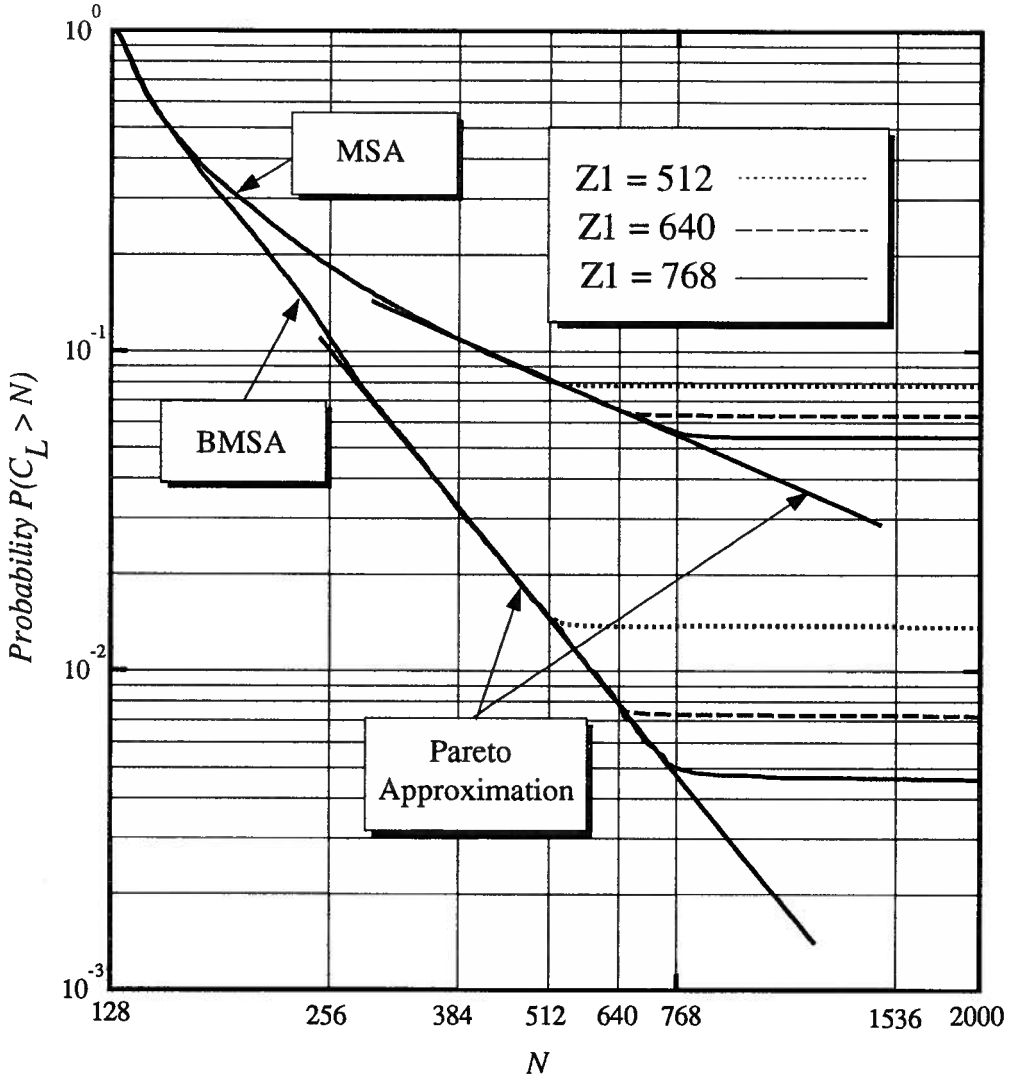


Figure 6.4 Empirical computational distribution
for the final decision using Z_1 as a parameter.

Figure 6.6 shows the results of the final computational distributions of the MSA and the BMSA for various values of Z. The different parameters used are: $C_{lim} = 2000, 3300$ and 4500 for $Z = 11, 21$ and 31, respectively, $T = 3$, and $Z_1 = 6L$. C_{lim} is increased with Z in order to guarantee erasure-free decoding. This figure shows that parameter Z does not affect either the first stack overflow probability or the final number of computations,

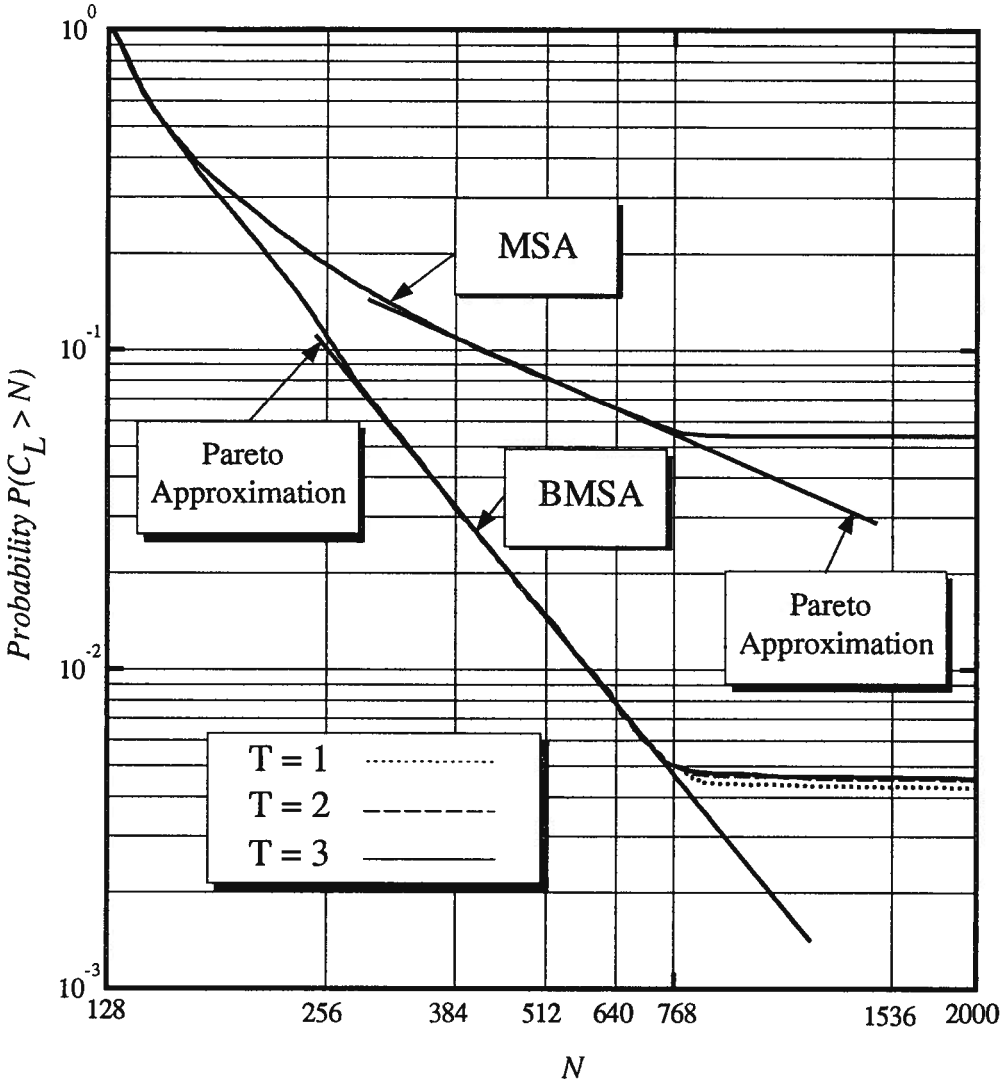


Figure 6.5 Empirical computational distribution for the final decision using T as a parameter.

as it was predicted by analysis (see (6.7) and (6.10)).

According to the above analysis and computer simulation results, the primary stack size Z_1 is the dominating factor in the final computational effort for both the MSA and the BMSA, while C_{lim} is mainly determined by C_{crit} to guarantee erasure-free decoding.

The average number of computations of the proposed BMSA is now examined.

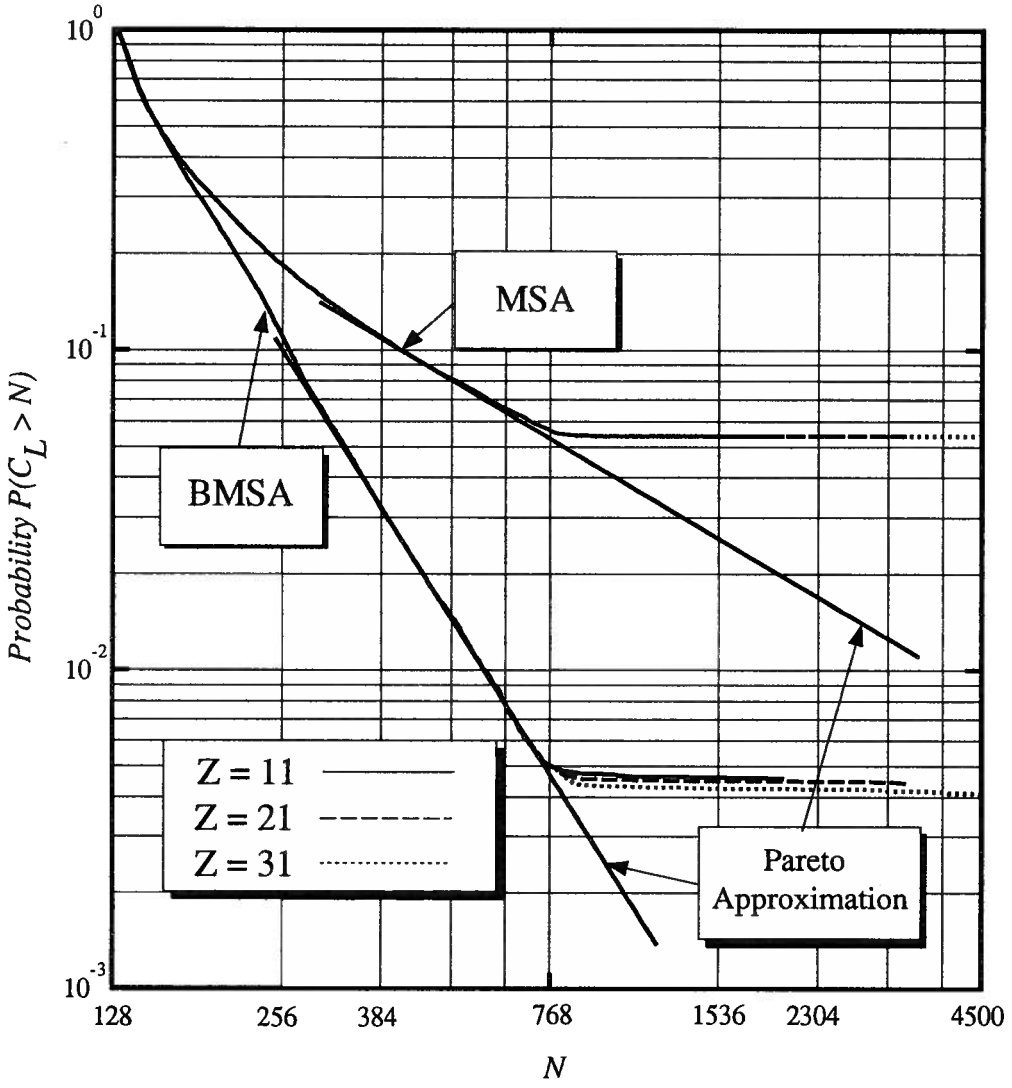


Figure 6.6 Empirical computational distributions
for the final decision using Z as a parameter.

Following the derivation of the average number of computations of the proposed BSD in Chapter 4, we get the following.

Property 6.3: The average number of computations per branch of the BMSA is upper-

bounded by

$$E\left[C_b^{BMSA}\right] \leq A_{12} + \begin{cases} \frac{A_{10}}{(2\rho-1)L}\left[(L-1)^{1-2\rho} - (Z_1-1)^{1-2\rho}\right], & \rho \neq \frac{1}{2} \\ \frac{A_{10}}{L}[\ln(Z_1-1) - \ln(L-1)], & \rho = \frac{1}{2} \end{cases}, \tag{6.11}$$

where the Pareto exponent $\rho < \rho_r$, and A_{12} is a constant given by

$$A_{12} \triangleq 1 + \frac{P_1^{BSD}(C_{lim} - Z_1)}{L}. \tag{6.12}$$

Proof: Similar to the analysis for the proposed BSD in Chapter 4, the average number of computations per branch can be expressed by

$$E\left[C_b^{BMSA}\right] = 1 + \frac{1}{L}\sum_{N=L}^{C_{lim}-1} P\left(C_L^{BMSA} > N\right), \tag{6.13}$$

where $P(C_L^{BMSA} > N)$ is upper-bounded by (6.10). Substituting (6.10) into (6.13), yields

$$\begin{aligned} E\left[C_b^{BMSA}\right] &\leq \frac{1}{L}\sum_{N=L}^{Z_1-1} A_{10}N^{-2\rho} + \frac{P_1^{BSD}(C_{lim} - Z_1)}{L} + 1 \\ &= \frac{A_{10}}{L}\sum_{N=L}^{Z_1-1} N^{-2\rho} + A_{12}, \end{aligned} \tag{6.14}$$

where $A_{12} = 1 + P_1^{BSD}(C_{lim} - Z_1)/L$. Moreover, since

$$\begin{aligned} \sum_{N=L}^{Z_1-1} N^{-2\rho} &\leq \int_{L-1}^{Z_1-1} x^{-2\rho}dx \\ &= \begin{cases} \frac{1}{2\alpha-1}\left[(L-1)^{1-2\rho} - (Z_1-1)^{1-2\rho}\right], & \rho \neq \frac{1}{2} \\ \ln(Z_1-1) - \ln(L-1), & \rho = \frac{1}{2} \end{cases}, \end{aligned} \tag{6.15}$$

and substituting (6.15) into (6.14) the above property is proved. Q.E.D.

Similarly, for the MSA, the average number of computations per branch can be bounded by

$$E\left[C_b^{MSA}\right] \leq A_{13} + \begin{cases} \frac{A_9}{(\rho-1)L}\left[(L-1)^{1-\rho} - (Z_1-1)^{1-\rho}\right], & \rho \neq 1 \\ \frac{A_9}{L}[\ln(Z_1-1) - \ln(L-1)], & \rho = 1 \end{cases}, \tag{6.16}$$

where A_{13} is a constant and is defined as below

$$A_{13} \triangleq 1 + \frac{P_1^{USD}(C_{lim} - Z_1)}{L}. \tag{6.17}$$

By comparing (6.11) and (6.16), one can notice that under the same parameters, the average number of computations with the BMSA is smaller than that with the MSA. Under the same C_{lim}, in order to reduce the average number of computations, the primary stack size Z_1 should be increased. This indicates a clear trade-off between the required memory and the computational performance for both the MSA and the BMSA. However, the BMSA offers a substantial improvement in terms of computational performance and memory requirements compared to the MSA, as it was shown above. Moreover, one can notice from (6.11) and (6.16) that the average number of computations is less sensitive to C_{lim} for the BMSA as compared to the MSA.

Table 6.1 shows some empirical results of the average number of computations for different values of Z_1. As predicted, in all cases, the use of the BMSA results in a significant reduction of the average number of computations.

Table 6.1 Average number of computations as a function of Z_1

SABODP (m = 7), L =128, 5×10^4 runs E_b/N_o = 4.58 dB, T = 3, Z = 11, C_{lim} = 2000	Z_1 = 512	Z_1 = 640	Z_1 = 768
MSA	2.512	2.347	2.256
BMSA	1.576	1.504	1.478

Table 6.2 shows some empirical results of the average number of computations for different values of T. These results indicate that parameter T is not related with the average number of computations. This can also be seen from the analytical expressions given by (6.11) and (6.16).

Table 6.2 Average number of computations as a function of T

SABODP (m = 7), L = 128, 5×10^4 runs E_b/N_o = 4.58 dB, Z_1 = 768, Z = 11, C_{lim} = 2000	T = 1	T = 2	T = 3
MSA	2.256	2.256	2.256
BMSA	1.475	1.477	1.478

Table 6.3 shows some empirical results on the average number of computations as a function of C_{lim}. The results confirm that the average number of computations is less sensitive to C_{lim} in the case of the BMSA compared to the MSA.

Table 6.3 Average number of computations as a function of C_{lim}

SABODP (m = 7), L = 128, 5×10^4 runs E_b/N_o = 4.58 dB, Z_1 = 768, T = 3	C_{lim} = 2000 (Z = 11)	C_{lim} = 3300 (Z = 21)	C_{lim} = 4500 (Z = 31)
MSA	2.256	2.803	3.310
BMSA	1.478	1.522	1.555

6.4 Error Performance of the BMSA

The error performance of the BMSA can be derived in the same way as for the conventional MSA [18]. One can hence write the following property.

Property 6.4: The bit error probability of the BMSA is upper-bounded by

$$P_b^{BMSA} \leq P_b^{BSD} + P_1^{BSD}, \tag{6.18}$$

where P_b^{BSD} is the bit error probability of BSD using a single pair of infinite size FS and BS.

The effect of parameters Z_l, T and Z on the bit error probability of the BMSA is now provided.

6.4.1 Effect of Parameter Z_l

Based on the results in Chapter 5, P_b^{BSD} of BSD-merge is asymptotically the same as that of USD. As it was found in the previous section, for the same primary stack size Z_l, the first stack overflow probability P_1^{BSD} of the BMSA is much smaller than that of the conventional MSA. For example, as shown in Figure 6.4, for $Z_l = 768$, P_1^{BSD} is around 5×10^{-3}, whereas P_1^{USD} is only about 5×10^{-2}. Hence, by employing the BMSA, the error performance can be substantially improved without increasing the primary stack size Z_l.

Figure 6.7 shows the bit error probability curves of the MSA and the BMSA as a function of Z_l for the two values $E_b/N_o = 4.58\ dB$ ($\rho_r = 1$) and $E_b/N_o = 5.55\ dB$ ($\rho_r = 1.5$), which are drawn in dotted and dashed lines, respectively. The parameters used in the decoding are the same as those used for Figure 6.4. As it can be seen from the figure, the error performance of both the MSA and the BMSA improves as Z_l increases, up to a certain floor which is determined by the error performance limit of the employed code, that is P_b^{BSD} of BSD-merge or P_b^{USD} of USD. For reference, we obtained P_b^{BSD} through computer simulations using Algorithm TTmerge with *unlimited* sizes of FS and BS. Results are plotted on Figure 6.7 as straight lines. The error floor at $E_b/N_o = 4.58\ dB$ is around 3×10^{-3}, and at $E_b/N_o = 5.55\ dB$ it is around 10^{-4}.

With the BMSA, the error floor is practically reached with $Z_l = 640$ and $Z_l = 512$ for $E_b/N_o = 4.58\ dB$ and $E_b/N_o = 5.55\ dB$, respectively. For $E_b/N_o = 4.58\ dB$ and $Z_l = 640$, we can get an approximation of P_1^{BSD} which is about 7×10^{-3} from Figure 6.5. To achieve this same value with the MSA, using the Pareto approximation on Figure 6.5, the required Z_l is about 5356, which is 8 times more than that of the BMSA.

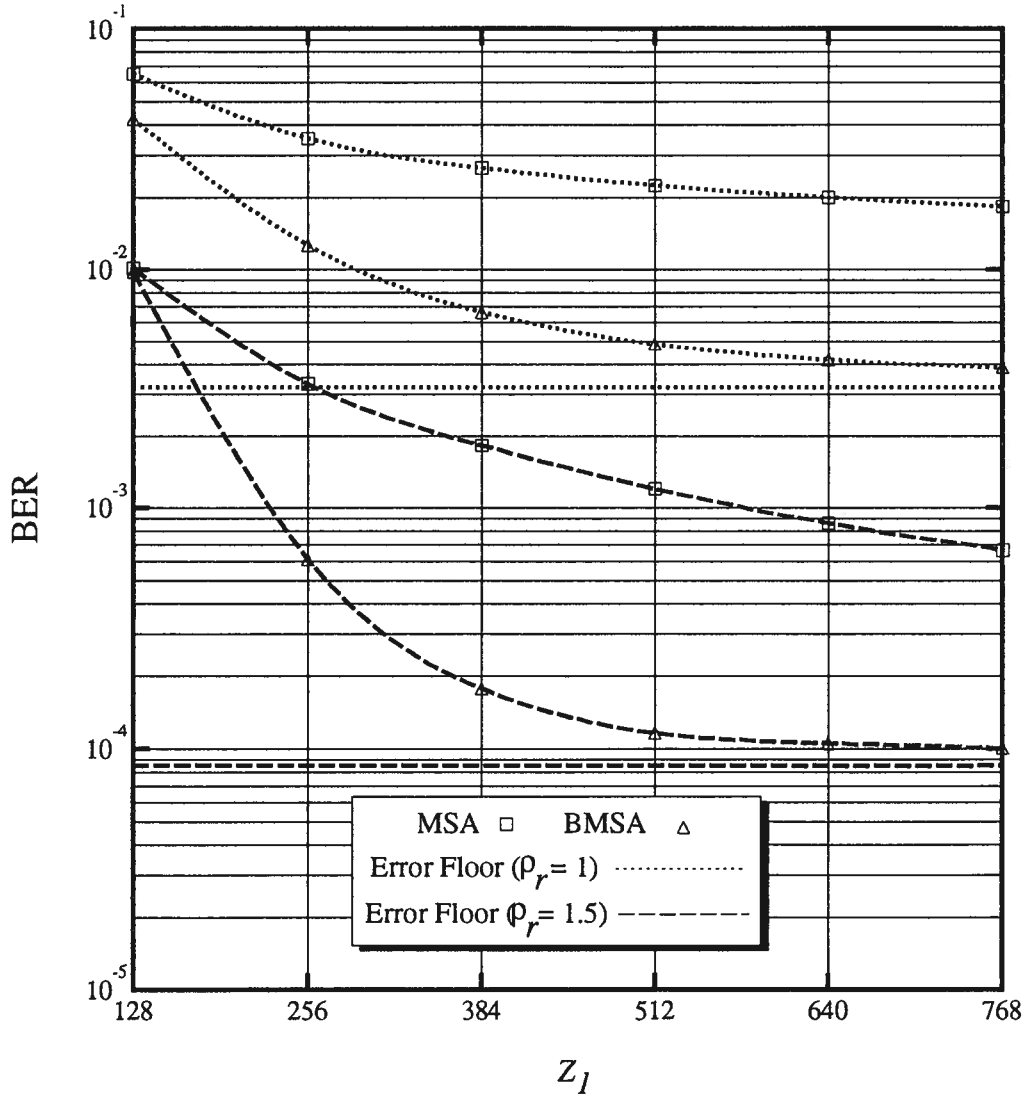


Figure 6.7 Bit error probability of the BMSA and MSA as a function of Z_l.

6.4.2 Effects of Parameters T and Z

As suggested by Chevillat and Costello [18], T and Z have no significant effects on the error performance of the MSA. Similar arguments also hold for the BMSA. Tables 6.4 and 6.5 show the error performance of both the BMSA and the MSA as a function of T and Z with the same parameters used for Figures 6.5 and 6.6, respectively. As expected, the error performance is independent of parameters T and Z. Thus, T and Z can be used to improve memory requirements and computational performance.

Table 6.4 Error performance as a function of parameter T.

SABODP code $m = 7, L = 128$ 5×10^4 runs E_b/N_o=4.58 dB	MSA			BMSA		
	$T = 1$	$T = 2$	$T = 3$	$T = 1$	$T = 2$	$T = 3$
BER	1.88e-2	1.86e-2	1.84e-2	3.96e-3	3.89e-3	3.88e-3
# of error blocks	3503	3480	3464	1809	1803	1802

Table 6.5 Error performance as a function of parameter Z.

SABODP code $m = 7, L = 128$ 5×10^4 runs E_b/N_o=4.58 dB	MSA			BMSA		
	$Z = 11$	$Z = 21$	$Z = 31$	$Z = 11$	$Z = 21$	$Z = 31$
BER	1.84e-2	1.81e-2	1.80e-2	3.88e-3	3.81e-3	3.81e-3
# of error blocks	3464	3457	3454	1802	1796	1795

6.5 Comparison with Viterbi Decoding

We now compare the proposed BMSA with the optimum *Viterbi algorithm* (VA). Figure 6.8 shows the BER of the various algorithms found through computer simulations. The codes and parameters used are as follow:

- A rate *1/2* memory $m = 7$ optimum free distance code [7] was used with VA.
- A rate *1/2* memory $m = 7$ SABODP code was used with both the MSA and the BMSA. The different parameters used with both the MSA and the BMSA are: $L = 128$, $T = 3$, $Z = 11$, $Z_1 = 640$, and $C_{lim} = 2000$.
- Another rate *1/2* memory $m = 10$ SABODP code was used with the BMSA. The different parameters used are: $L = 128$, $T = 3$, $Z = 11$, $Z_1 = 1024$, and $C_{lim} = 2300$.

As expected, the BER performance of the BMSA is better than that of the conventional MSA. For example, for the same code and the range of E_b/N_o shown in Figure 6.8, the BMSA is about 0.5 dB better than the MSA. The VA using the optimum rate *1/2* code performs better than the BMSA for the same code memory m. However, as the memory m of the code used increases, the error performance of the BMSA improves. For example, with the $m = 10$ code, the error performance of the BMSA is about *0.4* dB better than that of the VA at a BER = 10^{-5}. This improvement in BER can be even made larger by using a longer memory code. For the BMSA, the average number of computations per decoded bit is typically less than 2 (see Table 6.1). This is much smaller than that of the VA, which requires 2^m computations per bit. However, as pointed out by Ma [21], a node extension is executed much faster with the VA than with a sequential decoder (typically 10 times faster). Nevertheless, the overall decoding speed with the BMSA remains faster than with the VA. More importantly, the decoding complexity with the BMSA is virtually insensitive to the code memory m.

Moreover, it was found by Ma [21] that as the code memory m increases, the MSA tends to require less memory for decoding than with the VA. Furthermore, we have shown that the proposed BMSA not only performs better than the MSA, but also requires less

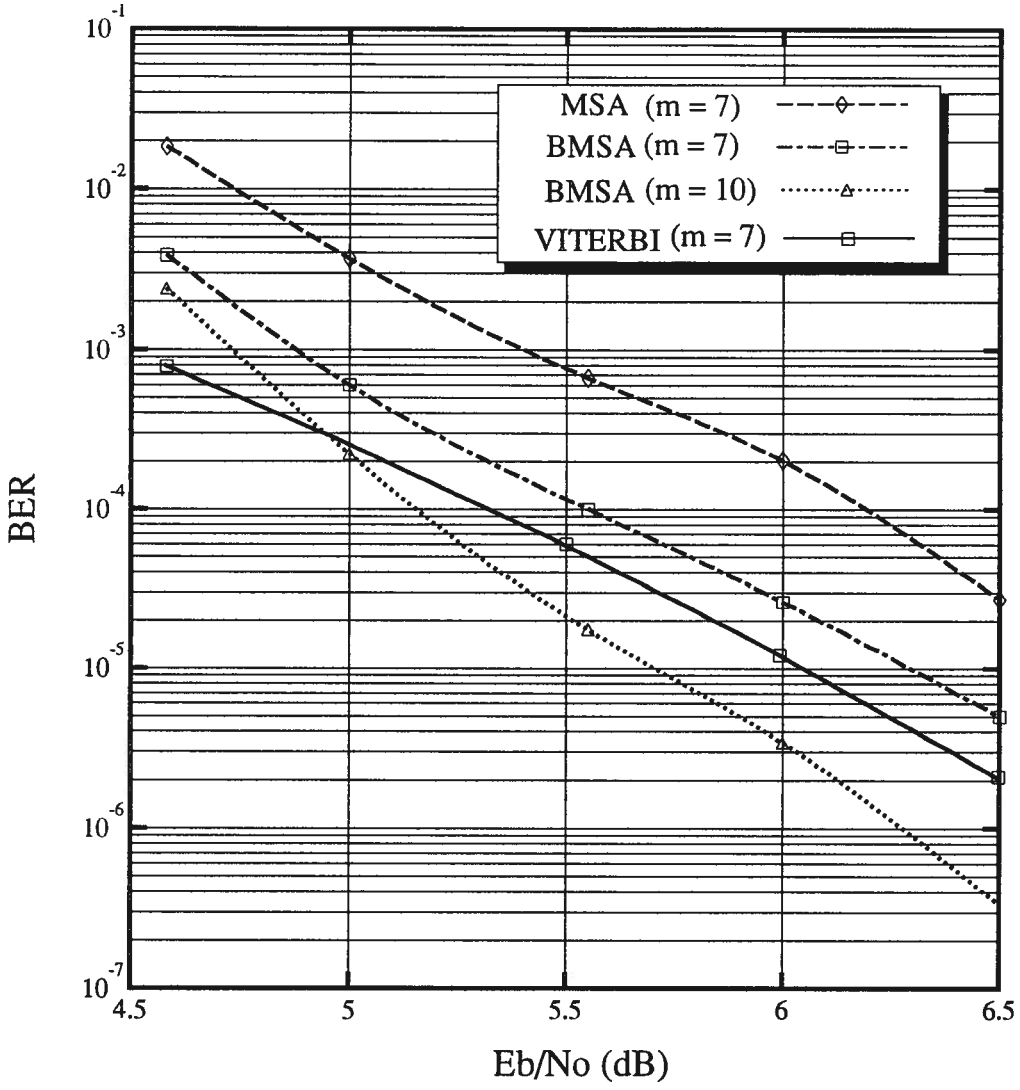


Figure 6.8 Comparison on error performance for different algorithms.

memory for decoding. Thus, the BMSA appears to be an attractive alternative to the VA in applications where low error probabilities together with high decoding speeds are required.

Chapter 7

Conclusions and Suggestions for Further Research

In a system using convolutional coding and sequential decoding, data is transmitted in blocks, and each block is terminated by a known tail. A conventional sequential decoder searches the tree from the forward direction, starting from initial encoder state (initial root of the tree), up to the final encoder state (end root of the tree). The main drawback of sequential decoding is the variability of its decoding effort which could cause decoding erasures. In order to alleviate this drawback, efficient *bidirectional sequential decoding* (BSD) techniques, in which the tree is simultaneously searched from forward and backward directions, were proposed and analyzed in this dissertation.

The relationships between backward coding and forward coding have been examined in detail. Good convolutional codes, with memory m ranging from 2 to 25, suitable for bidirectional decoding have been found through computer search. These codes possess the same distance properties from both forward and backward directions.

In the proposed BSD, two decoders are used; one is called *forward decoder* (FD), and is used to search the tree from forward direction; while the other is called *backward decoder* (BD), and is used for the backward search of the tree. Forward decoding and backward decoding are performed simultaneously, and stop somewhere in the tree. In one class of BSD, which are referred to as BSD-merge, decoding stops whenever FD and BD reach a common encoder state somewhere in the tree. In the other class of BSD, that is BSD-no-merge, no common encoder state is required, and decoding stops when

FD meets BD. Different BSD algorithms were constructed based on the stack algorithm; Algorithm TAmeet which belongs to the class of BSD-no-merge, Algorithm TAmerge and Algorithm TTmerge which belong to BSD-merge, and finally Algorithm HTTmerge which is a hybrid version of BSD-merge and BSD-no-merge.

The computational performance of the proposed BSD algorithms has been analyzed in detail. It was found that the distribution of the total number of computations per decoded block is still Pareto, as that of *unidirectional sequential decoding* (USD). However, the advantage of the proposed BSD appears as an increase in the Pareto exponent, and hence as a decrease in the computational variability. This results a decrease in the erasure probability. More specifically, it is proved by using the random coding approach that the Pareto exponent of BSD using Algorithm TAmeet is asymptotically twice that of USD. It is conjectured that this also applies to Algorithm TAmerge. On the other hand, it was found that the computational cutoff rate of sequential decoding remains unchanged, but the use of BSD reduces the average number of computations per decoded block. Extensive computer simulations were performed which confirmed our theoretical analysis.

Based on the random coding approach, the error performance of BSD-merge is shown to be asymptotically the same as that of USD. Moreover, the bit error probability of Algorithm TAmeet is found to satisfy the random coding bound for block codes. Computer simulations were conducted and found to agree with the analytical findings.

Among all the proposed BSD algorithms, Algorithm TAmeet is the simplest one in terms of implementation. However, its error performance is relatively poor compared to that of USD. Both Algorithm TAmerge and Algorithm TTmerge offer good computational

and error performances. The merging test is more demanding in Algorithm TAmerge than in Algorithm TTmerge, and therefore, Algorithm TTmerge may be preferred in practice. Algorithm HTTmerge is very useful as it can provide a tradeoff between computational effort and error performance.

None of the above four schemes can completely eliminate the erasure problem. Therefore, the idea of BSD was combined with the MSA, and an efficient erasure-free *bidirectional MSA* (BMSA) was constructed. Through analysis and computer simulations, it is shown that the new BMSA offers substantial advantages over the MSA in terms of computational effort, memory requirements and error performance. The BMSA appears as an attractive alternative to the VA where low error probabilities and high decoding speeds are required.

Some interesting problems for future research are:

1. The implementation aspects of the proposed BSD algorithms need to be investigated. One interesting systolic search was recently proposed for the stack algorithm [59]. It would be nice to see how and if this idea can be adapted to the BSD algorithms.
2. BSD can be used for the decoding of tree-like block codes. Of course one needs to examine the distance properties of these tree-like block codes from the backward direction. Perhaps good block codes suitable for BSD need to be designed.
3. Recently sequential decoding was successfully applied to trellis codes [58], and good long memory trellis codes were found. Similarly, the proposed BSD algorithms can be applied to trellis codes. Our analysis is general and applies to both convolutional and trellis codes. However, one needs to find good symmetric trellis codes that

possess good distance properties from both forward and backward directions. Note that the rate 1/2 SBODP or SABODP codes are suitable for Gray mapped QPSK modulation since in this case the Euclidean and Hamming distances are equivalent.

4. It would be of interest to see how well the proposed BSD algorithms perform for resolving intersymbol interference (ISI) channels, compared to conventional sequential decoding [60].
5. Finally, the BSD idea can be combined in conjunction with the *buffer looking algorithm* which is another erasure-free decoding method proposed recently by Wang [58]. A study of this approach is also of interest.

References

[1] C. E. Shannon, "A Mathematical Theory of Communication," *Bell Syst. Techn. J.*, vol. 27, July and Oct. 1948.

[2] J. M. Wozencraft and I. M. Jacobs, *Principles of Communication Engineering*. New York: Wiley, 1965.

[3] W. W. Peterson and E. J. Weldon, *Error-Correcting Codes*. Cambridge, Mass.: MIT Press, 2nd ed., 1972.

[4] F. Jelinek, *Probabilistic Information Theory*. McGraw-Hill Book Company, 1968.

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

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

[7] V. K. Bhargava, D. Haccoun, R. Matyas, and P. Nuspl, *Digital Communication by Satellite*. New York: Wiley, 1981.

[8] S. Lin and D. J. Costello, Jr., *Error Control Coding: Fundamentals and Applications*. NJ: Prentice-Hall: Englewood Cliffs, 1983.

[9] A. J. Viterbi, "Error Bounds for Convolutional Codes and an Asymptotically Optimum Decoding Algorithm," *IEEE Trans. on Inform. Theory*, vol. IT-13, pp. 260–269, April 1967.

[10] J. B. Anderson and S. Mohan, "Sequential Coding Algorithms: A Survey and Cost Analysis," *IEEE Trans. on Comm.*, vol. COM-32, pp. 169–176, Feb. 1984.

[11] J. M. Wozencraft and B. Reiffen, *Sequential Decoding*. Cambridge, Mass.: MIT Press, 1961.

[12] H. L. Yudkin, "Channel State Testing in Information Decoding," *Sc.D. Diss., Dept. Elec. Eng., M.I.T., Cambridge*, 1964.

[13] R. M. Fano, "A Heuristic Discussion of Probabilistic Decoding," *IEEE Trans. on Inform. Theory*, vol. IT-9, pp. 64–73, April 1963.

[14] G. D. Forney, Jr. and E. K. Bower, "A High-Speed Sequential Decoder: Prototype Design and Test," *IEEE Trans. Commun. Technol.*, vol. COM-19, pp. 821–835, Oct. 1971.

[15] K. S. Zigangirov, "Some Sequential Decoding Procedures," *Probl. Peredach. Inform.*, vol. 2, no. 4, pp. 13–15, 1966.

[16] F. Jelinek, "Fast Sequential Decoding Using a Stack," *IBM J. Res. Develop.*, vol. 13, pp. 675–685, Nov. 1969.

[17] J. W. Layland, and W. A. Lushbaugh, "A Flexible High-Speed Sequential Decoder for Deep Space Channels," *IEEE Trans. Commun. Technol.*, vol. COM-19, pp. 813–820, Oct. 1971.

[18] P. R. Chevillat and D. J. Costello, Jr., "A Multiple Stack Algorithm for Erasurefree Decoding of Convolutional Codes," *IEEE Trans. on Comm.*, vol. COM-25, pp. 1460–1470, Dec. 1977.

[19] D. Haccoun and M. J. Ferguson, "Generalized Stack Algorithms for Decoding Convolutional Codes," *IEEE Trans. on Inform. Theory*, vol. IT-21, pp. 638–651, Nov. 1975.

[20] P. R. Chevillat, "Fast Sequential Decoding and a New Complete Decoding Algorithm," *Ph.D. Diss., Dept. Elec. Eng., Illinois Institute of Technology, Chicago, IL*, May 1976.

[21] H. H. Ma, "The Multiple Stack Algorithm Implemented on a Zilog Z-80 Microcomputer," *IEEE Trans. on Comm.*, vol. COM-28, pp. 1876–1882, Nov. 1980.

[22] G. D. Forney, Jr., "Final Report on a Coding System Design for Advanced Solar Missions," *Rep. NAS2-3637, Contract NASA CR73167, NASA Ames Res. Center, Moffett Field, CA*, 1967.

[23] L. R. Bahl, C. D. Cullum, W. Frazer, and F. Jelinek, "An Efficient Algorithm for Computing Free Distance," *IEEE Trans. on Inform. Theory*, vol. IT-18, pp. 437–439, May 1972.

[24] K. J. Larsen, "Comments on 'An Efficient Algorithm for Computing Free Distance'," *IEEE Trans. on Inform. Theory*, vol. IT-19, pp. 577–579, July 1973.

[25] M. Rouanne and D. J. Costello, Jr., "An Algorithm for Computing the Distance Spectrum of Trellis Codes," *IEEE J. on Selected Areas in Commun.*, vol. 7, pp. 929–940, Aug. 1989.

[26] F. Hemmati, "Bidirectional Trellis Decoding," *1990 IEEE Book of Abstracts of Information Theory Symposium, San Diego*, p. 107, Jan. 1990.

[27] D. Haccoun and J. Belzile, "Bidirectional Algorithms for the Decoding of Convolutional Codes," *1990 IEEE Book of Abstracts of Information Theory Symposium, San Diego*, p. 177, Jan. 1990.

[28] J. Belzile and D. Haccoun, "Bidirectional Breadth-First Algorithms for the Decoding of Convolutional Codes," *IEEE Trans. on Comm.*, vol. 41, pp. 370–380, Feb. 1993.

[29] K. Li and S. Kallel, "Bidirectional Sequential Decoding Algorithms," *IEEE Symposium on Information Theory, in San Antonio, TX, USA*, Jan. 1993.

[30] I. M. Jacobs and E. R. Berlekamp, "A Lower Bound to the Distribution of Computation for Sequential Decoding," *IEEE Trans. on Inform. Theory*, vol. IT-13, pp. 167–174, April 1967.

[31] R. Johannesson, "On the Distribution of Computation for Sequential Decoding Using the Stack Algorithm," *IEEE Trans. on Inform. Theory*, vol. IT-25, pp. 323–331, May 1979.

[32] J. E. Savage, "Sequential Decoding – the Computation Problem," *Bell Syst. Techn. J.*, vol. 45, no. 1, pp. 149–175, 1966.

[33] D. D. Falconer, "An Upper Bound on the Distribution of Computation for Sequential Decoding with Rate Above Rcom," *M.I.T.-RLE Quart. Progress Rept.*, pp. 174–179, April 1966.

[34] F. Jelinek, "An Upper Bound on Moments of Sequential Decoding Effort," *IEEE Trans. Inform. Theory*, vol. IT-15, pp. 140–149, January 1969.

[35] G. D. Forney, Jr., "Convolutional Codes. II. Maximum-Likelihood Decoding," *Inform. Contr.*, vol. 25, pp. 222–266, 1974.

[36] G. D. Forney, Jr., "Convolutional Codes. III. Sequential Decoding," *Inform. Contr.*, vol. 25, pp. 267–297, 1974.

[37] P. R. Chevillat and D. J. Costello, Jr., "Distance and Computation in Sequential Decoding," *IEEE Trans. on Commun.*, vol. COM-24, pp. 440–447, April 1976.

[38] P. R. Chevillat and D. J. Costello, Jr., "An Analysis of Sequential Decoding for Specific Time-Invariant Convolutional Codes," *IEEE Trans. on Inform. Theory*, vol. IT-24, pp. 443–451, July 1978.

[39] K. Li and S. Kallel, "Bidirectional Sequential Decoding for Convolutional Codes," *1991 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing in Victoria, B.C., Canada*, May 1991.

[40] J. L. Massey and M. K. Sain, "Inverses of Linear Sequential Circuits," *IEEE Trans. on Comp.*, vol. C-17, pp. 330–337, April 1968.

[41] J. L. Massey, "Reversible Codes," *Inform. and Control*, vol. 7, pp. 369–380, 1964.

[42] J. P. Robinson, "Reversible Convolutional Codes," *IEEE Trans. on Inform. Theory*, pp. 609–610, July 1968.

[43] R. Johannesson, "Robustly-Optimal Rate One-Half Binary Covolutional Codes," *IEEE Trans. on Inform. Theory*, vol. IT-21, pp. 464–468, July 1975.

[44] R. Johannesson and E. Paaske, "Further Results on Binary Covolutional Codes with an Optimum Distance Profile," *IEEE Trans. on Inform. Theory*, vol. IT-24, pp. 264–268, March 1978.

[45] M. Cedervall and R. Johannesson, "A Fast Algorithm for Computing Distance Spectrum of Convolutional Codes," *IEEE Trans. on Inform. Theory*, vol. 35, pp. 1146–1159, Nov. 1989.

[46] D. Haccoun, "A Markov Chain Analysis of the Sequential Decoding Metric," *IEEE Trans. on Inform. Theory*, vol. IT-26, pp. 109–113, January 1980.

[47] J. B. Anderson and S. Mohan, *Source and Channel Coding: an Algorithmic Approach.* Kluwer Academic Publishers, 1991.

[48] J. K. L. Jordan, "The Performance of Sequential Decoding in Conjunction with Efficient Modulation," *IEEE Trans. Commun. Technol.*, vol. COM-14, pp. 283–297, June 1966.

[49] A. Papoulis, *Probability, Random Variables, and Stochastic Processes.* McGraw-Hill Book Company, 1965.

[50] J. B. Anderson, "Limited Search Trellis Decoding of Convolutional Codes," *IEEE Trans. on Inform. Theory*, vol. IT-35, pp. 944–955, Sept. 1989.

[51] J. B. Anderson, "Sequential Decoding Based on an Error Criterion," *IEEE Trans. on Inform. Theory*, vol. IT-38, pp. 987–1001, May 1992.

[52] D. Haccoun, "A Branching Process Analysis of the Average Number of Computations of Stack Algorithm," *IEEE Trans. on Inform. Theory*, vol. IT-30, pp. 497–508, May 1984.

[53] E. Arikan, "An Upper Bound on the Cutoff Rate of Sequential Decoding," *IEEE Trans. on Inform. Theory*, vol. IT-34, pp. 55–63, January 1988.

[54] F. Jelinek, "Upper Bounds on Sequential Decoding Performance Parameters," *IEEE Trans. Inform. Theory*, vol. IT-20, pp. 227–239, March 1974.

[55] K. S. Zigangirov, "On the Error Probability of Sequential Decoding on the BSC," *IEEE Trans. Inform. Theory*, vol. IT-18, pp. 199–202, January 1972.

[56] K. S. Zigangirov, "Algorithm of Sequential Decoding in which the Error Probability Increases in Agreement with the Random Coding Bound," *Probl. Peredach. Inform.*, vol. 4, pp. 83–85, 1968.

[57] K. S. Zigangirov, "Procedures of Sequential Decoding," *Coding and Complexity - CISM Courses and Lectures No. 216, Springer-Verlag, Wien, New York*, pp. 109–130, 1975.

[58] F. Q. Wang, "Efficient Sequential Decoding of Trellis Codes," *Ph.D. Diss., Dept. Elec. Eng., University of Notre Dame, Notre Dame, Indiana*, Dec. 1992.

[59] C. Y. Chang and K. Yao, "Systolic array architecture for the sequential stack decoding algorithm," *Advanced algorithms and architectures for signal processing*, vol. 696, pp. 196–203, 1986.

[60] F. Xiong, A. Zerik and E. Shwedyk, "Sequential Sequence Estimation for Channels with Intersymbol Interference of Finite or Infinite Length," *IEEE Trans. Commun.*, vol. 38, pp. 795–804, June 1990.

[61] B. Hajek, "Hitting-time and Occupation-time Bounds Implied by Drift Analysis with Applications," *Advances in Applied Probability*, vol. 14, pp. 502–525, 1982.

Appendix A
Proof of Theorem 4.4

In the proof of Theorem 4.4, we need a lemma (see Lemma 2.2 of [61] for a more general result).

Lemma A.1: Suppose a real-valued random variable V satisfies $E[V] \geq \overline{\mu} > 0$ *and* $|V| \leq M$ *almost surely, for some constants* $\overline{\mu}$ and M. *Then for any* $\eta > 0$,

$$E\left[e^{-\eta V}\right] \leq -\eta(\overline{\mu} + M) + e^{\eta M}. \qquad (A.1)$$

In particular, for $\eta = (1/M)\ln(1 + \overline{\mu}/M)$,

$$E\left[e^{-\eta V}\right] \leq (1 + \overline{\mu}/M)[1 - \ln(1 + \overline{\mu}/M)] < 1. \qquad (A.2)$$

Proof: The proof follows from noting that $e^{-\eta V} = \sum_{k=0}^{\infty} (-\eta V)^k/k! \leq 1 - \eta V + [e^{\eta M} - (1 + \eta M)]$. *Q.E.D.*

Proof of Theorem 4.4: Similar to the proof of the computational lower bound in Theorem 4.3, separate every received block into two almost equal parts, i.e., levels $[1, \lfloor (L-m)/2 \rfloor]$ for forward tree search operations and $[\lfloor (L-m)/2 \rfloor + m, L]$ for backward tree search. Here, L and m denote any L_i and m_i for any code $\mathfrak{C}_i$. Also with the assumption of correct decoding, $C_L^{BSD} \geq \min\left[C^{FC}_{\lfloor (L-m)/2 \rfloor}, C^{BC}_{\lceil (L-m)/2 \rceil}\right]$ where $C^{FC}_{\lfloor (L-m)/2 \rfloor}$ and $C^{BC}_{\lceil (L-m)/2 \rceil}$, defined in the proof of Theorem 4.3, are independent random variables.

Now, let l_n be an arbitrary integer of multiple n and less than $\lfloor (L-m)/2 \rfloor \cdot n$, and define $U \triangleq \lfloor (L-m)n/2l_n \rfloor$. For $1 \leq j \leq U$, define N_j^F to be the set of nodes in the

forward tree at segment jl_n that are descendants of the correct end node of segment $(j-1)l_n$. Given that the end node of the segment $(j-1)l_n$ is correctly decoded, let c_j^F be the number of distinct nodes in N_j^F that are hypothesized (extended) by the idealized forward decoder before correctly decoding the segment jl_n. Then $C^{FC}_{\lfloor (L-m)/2 \rfloor} \geq \sum_{j=1}^{U} c_j^F$. Define a similar quantity, n_j^F, to be the number of nodes in N_j^F that are at least as likely to be the correct nodes at segment jl_n. Then, because the codewords are *a priori* equiprobable and correct decoding is assumed, one can use Lemma 3.1 in [53][21] to show that

$$E\left[c_j^F\right] \geq \frac{1}{2} E\left[n_j^F\right]. \tag{A.3}$$

Lemma 3.1 of [53] assumes that the stack algorithm is used. However, its proof remains valid for any sequential decoder for which one can determine which of two nodes at the same level is hypothesized (extended) first, by considering only the received message up to that level.

Now let $\mathbb{K}$ denote the DMC channel. For every $Q > 0$ and $0 < \epsilon_0 \leq 1$, one can always choose a finite $l_n(\mathbb{K}, R, \varepsilon_0, Q)$ sufficiently large to satisfy the condition of Lemma 6.1 in [53], and thus

$$\frac{1}{l_n}[\exp\left(l_n(R-R_1)/8\right) - 2] > 32Q\left(1+\varepsilon_0^{-1}\right)n^{-1}. \tag{A.4}$$

Therefore, using Lemma 6.1 in [53] into (A.3), we can write

$$\begin{aligned} E\left[c_j^F\right] &\geq \frac{1}{2} E\left[n_j^F\right] \\ &\geq \frac{1}{2}\exp\left[l_n(R-R_1)/8\right] \\ &> 16Ql_n\left(1+\varepsilon_0^{-1}\right)n^{-1} + 1, \qquad 1 \leq j \leq U. \end{aligned} \tag{A.5}$$

[21] The quantities defined by Arikan in [53] do not count the correct node itself (and thus equal one less than the quantities defined here), but clearly the result still holds.

Let $\mu_0 = 16Ql_n(1+\varepsilon_0^{-1})n^{-1}$, $\overline{\mu} = 1$, and $M = |X|^{l_n}$ where $|X|$ denote the size of the channel input alphabet. Then, $E\left[c_j^F - \mu_0\right] > 1$ and $|c_j^F - \mu_0| \leq M$ since $0 < c_j^F \leq \exp(Rl_n)$, $0 < \mu_0 < E\left[c_j^F\right] \leq \exp(Rl_n)$, and for distinct codewords, $\exp(Rl_n) \leq |X|^{l_n}$. Let $\eta = (1/M)\ln(1+1/M)$. By Lemma A.1,

$$E\left[\exp\left(-\eta\left(c_j^F - \mu_0\right)\right)\right] \leq \theta, \tag{A.6}$$

where $\theta = (1+1/M)[1-\ln(1+1/M)]$, and $\theta < 1$.

Since c_j^F, $1 \leq j \leq U$ are identically distributed, independent random valuables, one can use Chernoff bound and get

$$\begin{aligned} P\left\{\sum_{j=1}^{U}\left(c_j^F - \mu_0\right) \leq 0\right\} &\leq E\left[\exp\left(-\eta\sum_{j=1}^{U}\left(c_j^F - \mu_0\right)\right)\right] \\ &= \left\{E\left[\exp\left(-\eta\left(c_j^F - \mu_0\right)\right)\right]\right\}^U \\ &\leq \theta^U. \end{aligned} \tag{A.7}$$

Now choose $L_0(\mathbb{K}, R, \varepsilon_0, Q)$ so that:

$$\begin{aligned} &(1)\ L_0 > 4\left(1+\varepsilon_0^{-1}\right)l_n/n \ \textit{and} \\ &(2)\ L_0 > 2l_n(1+\varepsilon_0)(1-\ln 2/\ln\theta)/n. \end{aligned} \tag{A.8}$$

Suppose$L_i \geq L_0$ and $L_i \geq \left(1+\varepsilon_0^{-1}\right)m_i$. Then

$$\begin{aligned} P\left(C^{FC}_{\lfloor (L_i - m_i)/2 \rfloor} > U\mu_0\right) &\geq P\left(\sum_{j=1}^{U} c_j^F > U\mu_0\right) \\ &= 1 - P\left(\sum_{j=1}^{U} c_j^F \leq U\mu_0\right) \\ &\geq 1 - \theta^U \\ &> 1/2. \end{aligned} \tag{A.9}$$

In the same way, one can show

$$P\left(C^{BC}_{\lceil (L_i-m_i)/2 \rceil} > U\mu_0\right) > 1/2. \tag{A.10}$$

Since $C^{FC}_{\lfloor (L_i-m_i)/2 \rfloor}$ and $C^{BC}_{\lceil (L_i-m_i)/2 \rceil}$ are independent, we have

$$\begin{aligned} E\left[C_b^{BSD}\right] &= E\left[C_{L_i}^{BSD}\right]/L_i \\ &\geq E\left[\min\left(C^{FC}_{\lfloor (L_i-m_i)/2 \rfloor}, C^{BC}_{\lceil (L_i-m_i)/2 \rceil}\right)\right]/L_i \\ &> (1/2)(1/2)U\mu_0/L_i \\ &> Q. \end{aligned} \qquad \text{Q.E.D. (A.11)}$$

Appendix B
List of Acronyms

A/D	Analogy-to-Digital
AWGN	Additive White Gaussian Noise
BD	Backward Decoder
BMSA	Bidirectional Multiple Stack Algorithm
BODP	Bidirectional Optimum Distance Profile
BS	Backward Stack
BSC	Binary Symmetric Channel
BSD	Bidirectional Sequential Decoding
BSD-merge	Bidirectional Sequential Decoding With a Merging Test
BSD-no-merge	Bidirectional Sequential Decoding Without a Merging Test
CDF	Column Distance Function
D/A	Digital-to-Analogy
DMC	Discrete Memoryless Channel
E_b/N_o	Energy Per Bit to Noise Ratio
FD	Forward Decoder
FEC	Forward Error Correction
FS	Forward Stack
GCD	Greatest Common Divisor
ISI	Intersymbol Interference

modem	Modulator/Demodulator
MSA	Multiple Stack Algorithm
ODP	Optimum Distance Profile
QPSK	Quaternary Phase Shift Key
SABODP	Symmetric Almost Bidirectional Optimum Distance Profile
SBODP	Symmetric Bidirectional Optimum Distance Profile
USD	Unidirectional Sequential Decoding
VA	Viterbi Algorithm