CHAPTER 14

Trellis-Based Soft-Decision Decoding Algorithms

Trellis representation of codes allows us to devise soft-decision decoding algorithms to achieve either optimal or suboptimal error performance with a significant reduction in decoding complexity. The best known and most commonly used trellis-based decoding algorithm is the Viterbi algorithm presented in Chapter 12. Convolutional codes with the Viterbi decoding algorithm have been widely used for error control in digital communications for many years and will continue to be used for many years to come.

This chapter presents several trellis-based decoding algorithms for linear block codes—some optimal, and some suboptimal. The first algorithm to be presented is the Viterbi algorithm. Decoding a linear block code with the Viterbi algorithm is exactly the same as decoding a terminated convolutional code; however, the decoding computational complexity depends on the sectionalization of a code trellis. The second algorithm is a recursive MLD algorithm based on a divide-and-conquer technique. A code is first divided into short sections, which are processed in parallel to form metric tables, one for each section. These metric tables are then combined recursively to form metric tables for longer sections until a metric table for the entire code is formed. This final table then contains only the most likely codeword and its metric. Metric tables for two adjacent code sections are combined based on a special trellis for the two sections. This decoding algorithm allows parallel/pipeline processing of received sequences to speed up the decoding process. The third decoding algorithm to be presented is a suboptimum decoding algorithm based on a low-weight subtrellis for a code. This decoding algorithm provides a good trade-off between error performance and decoding complexity. The next four decoding algorithms to be presented are the MAP algorithm and its variations. These decoding algorithms are devised to minimize the bit-error probability and to provide reliability information on decoded bits. Also included in this chapter are parallel and bidirectional MAP decodings.

14.1 THE VITERBI DECODING ALGORITHM

Decoding linear block codes with the Viterbi algorithm is quite straightforward. If an n-section bit-level trellis T for an (n, k) linear code is used for decoding, the decoding process is exactly the same as that for decoding a terminated convolutional code. The decoder processes the trellis from the initial state to the final state serially, section by section. The survivors at each level of the code trellis are extended to the next level through the connecting branches between the two levels. The paths that

enter a state at the next level are compared, and the most probable path (or the path with the largest correlation metric) is chosen as the survivor. This process continues until the end of the trellis is reached. At the end of the trellis, there is only one state, the final state s_f, and there is only one survivor, which is the most likely codeword. Decoding is then completed and the decoder is ready to process the next incoming received sequence.

The number of computations required to decode a received sequence can be enumerated easily. In a bit-level trellis, each branch represents one code bit, and hence a branch metric is simply a bit metric. The total number of additions required to extend the survivors from level to level is simply equal to the total number of branches in the trellis, which is given by (9.46). Let N_a denote the total number of additions required to process the trellis T. Then,

$$N_a = \sum_{i=0}^{n-1} 2^{\rho_i} \cdot I_i(a^*) \tag{14.1}$$

where 2^{ρ_i} is number of states at the ith level of the code trellis T, a^* is the current input information bit at time-i, and $I_i(a^*)$ is defined by (9.45), which is either 1 or 2. If there is an oldest information bit a^0 to be shifted out from the encoder memory at time-i, then there are two branches entering each state s_{i+1} of the trellis at time-$(i+1)$. This says that there are two paths entering each state s_{i+1} at the $(i+1)$th level of the bit-level trellis T. Otherwise, there is only one branch entering each state s_{i+1} at time-i. Define

$$J_i(a^0) = \begin{cases} 0, & \text{if } a^0 \notin A_i^s, \\ 1, & \text{if } a^0 \in A_i^s, \end{cases} \tag{14.2}$$

where A_i^s is the state-defining information set at time-i. Let N_c denote the total number of comparisons required to determine the survivors in the decoding process. Then,

$$N_c = \sum_{i=0}^{n-1} 2^{\rho_{i+1}} \cdot J_i(a^0). \tag{14.3}$$

Therefore, the total number of computations (additions and comparisons) required to decode a received sequence based on the bit-level trellis T using the Viterbi decoding algorithm is $N_a + N_c$.

EXAMPLE 14.1

Consider the (8, 4) RM code whose 8-section bit-level trellis is shown in Figure 9.6. From Table 9.2, we find that

$$I_0(a^*) = I_1(a^*) = I_2(a^*) = I_4(a^*) = 2,$$
$$I_3(a^*) = I_5(a^*) = I_6(a^*) = I_7(a^*) = 1,$$
$$J_0(a^0) = J_1(a^0) = J_2(a^0) = J_4(a^0) = 0,$$
$$J_3(a^0) = J_5(a^0) = J_6(a^0) = J_7(a^0) = 1.$$

The state-space dimension profile of the trellis is (0, 1, 2, 3, 2, 3, 2, 1, 0). Then, from (14.1) and (14.3), we find that

$$\begin{aligned} N_a &= 2^0 \cdot 2 + 2^1 \cdot 2 + 2^2 \cdot 2 + 2^3 \cdot 1 + 2^2 \cdot 2 + 2^3 \cdot 1 + 2^2 \cdot 1 + 2^1 \cdot 1 \\ &= 44, \\ N_c &= 2^1 \cdot 0 + 2^2 \cdot 0 + 2^3 \cdot 0 + 2^2 \cdot 1 + 2^3 \cdot 0 + 2^2 \cdot 1 + 2^1 \cdot 1 + 2^0 \cdot 1 \\ &= 11. \end{aligned}$$

Therefore, Viterbi decoding of the (8, 4) RM code requires 44 additions and 11 comparisons.

Suppose the bit-level trellis T is sectionalized into ν sections with section boundary locations in $\Lambda = \{t_0, t_1, \cdots, t_\nu\}$, where $0 = t_0 < t_1 < \cdots t_\nu = n$. This sectionalization results in a ν-section trellis $T(\Lambda)$. At boundary location t_i, the state space is $\Sigma_{t_i}(C)$. The ith section of $T(\Lambda)$ consists of the state space $\Sigma_{t_{i-1}}(C)$, the state space $\Sigma_{t_i}(C)$, and the branches that connect the states in $\Sigma_{t_{i-1}}(C)$ to the states in $\Sigma_{t_i}(C)$. A branch in this section represents $t_i - t_{i-1}$ code bits. Two adjacent states may be connected by multiple branches, called *parallel branches*. For convenience, we say that these parallel branches form a *composite branch*.

Viterbi decoding based on the ν-section trellis $T(\Lambda)$ is carried out as follows. Suppose the decoder has processed i trellis sections up to time-t_i. There are $|\Sigma_{t_i}(C)|$ survivors, one for each state in $\Sigma_{t_i}(C)$. These survivors together with their path metrics are stored in the decoder memory. To process the $(i + 1)$th section, the decoder executes the following steps:

1. Each survivor is extended through the composite branches diverging from it to the next level at time-t_{i+1}.
2. For each composite branch entering a state in $\Sigma_{t_{i+1}}(C)$, the single branch with the largest (correlation) metric is found. This metric is the branch metric.
3. Each composite branch is replaced by the branch with the largest metric.
4. The metric of a branch is added to the metric of the survivor from which the branch diverges. For each state $s \in \Sigma_{t_{i+1}}(C)$, the metrics of paths entering it are compared, and the path with the largest metric is selected as the survivor terminating at state s.

The decoder executes these steps repeatedly until it reaches the final state s_f. The sole survivor is the decoded codeword.

The total number of computations (additions and comparisons) required to process a sectionalized trellis $T(\Lambda)$ depends on the choice of the section boundary location set $\Lambda = \{t_0, t_1, \cdots, t_\nu\}$. A sectionalization of a code trellis that gives the smallest total number of computations (addition and comparison are considered to have the same complexity) is called an *optimal sectionalization* for the code. An algorithm for finding an optimal sectionalization has been devised by Lafourcade and Vardy [3]. This algorithm is based on a simple structure: for any two integers x and y with $0 \leq x < y \leq n$, the section from time-x to time-y in any sectionalized trellis $T(\Lambda)$ with $x, y \in \Lambda$ and $x + 1, x + 2, \cdots, y - 1 \notin \Lambda$ is identical. Let $\varphi(x, y)$

denote the number of computations required in steps 1 to 4 of the Viterbi decoding algorithm to process the trellis section from time-x to time-y. This number $\varphi(x, y)$ is determined solely by the choices of x and y. Let $\varphi_{\min}(x, y)$ denote the smallest number of computations required in decoding steps 1 to 4 to process the trellis section(s) from time-x to time-y in any sectionalized trellis $T(\Lambda)$ with $x, y \in \Lambda$. The value $\varphi_{\min}(0, n)$ gives the total number of computations of the Viterbi algorithm for processing the code trellis with an optimum sectionalization. It follows from the definitions of $\varphi(x, y)$ and $\varphi_{min}(x, y)$ that

$$\varphi_{min}(0, y) = \begin{cases} \min\{\varphi(0, y), \min_{0<x<y}\{\varphi_{min}(0, x) + \varphi(x, y)\}\}, & \text{for } 1 < y \leq n \\ \varphi(0, 1), & \text{for } y = 1. \end{cases} \tag{14.4}$$

For every $y \in \{1, 2, \cdots, n\}$, $\varphi_{min}(0, y)$ can be computed as follows. The values of $\varphi(x, y)$ for $0 \leq x < y \leq n$ are computed using the structure of the trellis section from time-x to time-y. First, $\varphi_{min}(0, 1)$ is computed. The value $\varphi_{min}(0, y)$ can be computed from $\varphi_{min}(0, x)$ and $\varphi(x, y)$ with $0 < x < y$ using (14.4). By storing the information when the minimum value occurs in the right-hand side of (14.4), we find an optimum sectionalization from the computation of $\varphi_{min}(0, n)$.

In general, optimum sectionalization of a code trellis results in a computational complexity less than that of the bit-level trellis. For example, consider the second-order RM code of length 64 that is a (64, 22) code with minimum Hamming distance 16. Using the Lafourcade–Vardy algorithm, we find that the boundary location set $\Lambda = \{0, 8, 16, 32, 48, 56, 61, 63, 64\}$ results in an optimum sectionalization. The resultant trellis is an eight-section trellis. With this optimum sectionalization, $\varphi_{\min}(0, 64)$ is 101,786. If the 64-section bit-level trellis is used for decoding, the total number of computations required is 425,209. Therefore, optimum sectionalization results in a significant reduction in computational complexity.

In general, optimum sectionalization results in a nonuniformly sectionalized trellis in which the section lengths vary from one section to another. This nonuniformity may not be desirable in IC hardware implementation of a Viterbi decoder [6, 8, 9]. Suppose the bit-level trellis T for the (64, 22) RM code is uniformly sectionalized into an 8-section trellis with the section boundary location set $\Lambda = \{0, 8, 16, 24, 32, 40, 48, 56, 64\}$. Each section consists of 8 code bits. Viterbi decoding based on this uniform 8-section trellis requires a total of 119,935 computations, which is larger than that based on the foregoing optimum sectionalization but is still much smaller than that based on the bit-level trellis.

If a code trellis consists of parallel and structurally identical subtrellises without cross connections between them, then it is possible to devise identical, less complex Viterbi decoders to process these subtrellises in parallel. At the end, there is one survivor from each subtrellis. The survivors from these subtrellises are then compared, and the one with the largest metric is chosen as the final survivor. This parallel processing of subtrellises not only simplifies the decoding complexity but also speeds up the decoding process. For example, the 4-section trellis for the (8, 4) RM code shown in Figure 9.17 has two 2-state parallel and structurally identical subtrellises; hence, two identical 2-state Viterbi decoders can be devised to process these two subtrellises in parallel. For a large code trellis, parallel decomposition, presented in Section 9.7, can be carried out to decompose the trellis into a desired number of parallel and structurally identical subtrellises.

Because the trellis for a block code terminates, decoding can be bidirectional. To facilitate the decoding, a codeword $\mathbf{v} = (v_0, v_1, \cdots, v_{n-1})$ is first permuted into the sequence $(v_0, v_{n-1}, v_2, v_{n-2}, \cdots)$ before its transmission. The corresponding received sequence is $(r_0, r_{n-1}, r_1, r_{n-2}, \cdots)$. From this received sequence, two received sequences are formed, $(r_0, r_1, \cdots, r_{n-1})$ and $(r_{n-1}, r_{n-2}, \cdots, r_0)$. These two received sequences are then fed into the code trellis from both ends for decoding. Suppose two decoders are used for decoding in both directions. When two decoders meet at the middle of the trellis, a decoding decision is made. The path with the largest metric is chosen as the decoded codeword. If the code trellis has mirror-image symmetry, the two decoders are identical. This bidirectional decoding further reduces the decoding delay and speeds up the decoding process. A high-speed Viterbi decoder for a (64, 40) RM subcode has been implemented based on both parallel and mirror-image structures of the code [9, 10].

14.2 A RECURSIVE MAXIMUM LIKELIHOOD DECODING ALGORITHM

The decoding algorithm to be presented in this section is based on a divide-and-conquer approach. An (n, k) linear block code C is first divided into short sections. The distinct vectors in each section form a linear code. These vectors are processed based on the corresponding section of the received sequence. For each section, we eliminate the less probable vectors and save the surviving vectors and their metrics in a table. Then, we combine the metric tables of two neighbor sections to form a metric table for a longer section of the code. In the combining process, we eliminate the less probable vectors. The resultant table contains the surviving vectors and their metrics for the combined sections. Continue the combining process until the full length the code is reached. The final table obtained contains only one survivor and its metric. This final survivor is the most likely codeword with respect to the received vector and hence the decoded codeword. Because the table combining process is carried out recursively, such a MLD algorithm is called a *recursive MLD* (RMLD) algorithm. The first such decoding algorithm was devised in [14, 15].

14.2.1 Metric Tables for Trellis Sections

Let T be the minimal n-section trellis diagram for an (n, k) binary linear block code C. Consider the trellis section from time-x to time-y with $0 \leq x < y \leq n$, denoted by $T_{x,y}$. For two states s_x and s_y with $s_x \in \Sigma_x(C)$ and $s_y \in \Sigma_y(C)$, if s_x and s_y are connected, the set of paths connecting state s_x to state s_y, denoted by $L(s_x, s_y)$, is a coset in the partition $p_{x,y}(C)/C^{tr}_{x,y}$. For convenience, $L(s_x, s_y)$ is regarded as a single path, called a *composite* path. Because each composite path $L(s_x, s_y)$ in $T_{x,y}$ is a coset in $p_{x,y}(C)/C^{tr}_{x,y}$, the number of distinct composite paths in $T_{x,y}$ is

$$2^{k(p_{x,y}(C))-k(C_{x,y})}. \tag{14.5}$$

Based on the trellis structures developed in Section 9.4, it is possible to show that each coset in $p_{x,y}(C)/C^{tr}_{x,y}$ appears as a composite path in $T_{x,y}$

$$2^{k-k(C_{0,x})-k(C_{y,n})-k(p_{x,y}(C))} \tag{14.6}$$

times [8, 11].

For each distinct composite path $L(s_x, s_y)$ in $T_{x,y}$, we find the path with the largest metric with respect to the corresponding section of the received sequence, $(r_x, r_{x+1}, \cdots, r_{y-1})$. Let $l(L(s_x, s_y))$ and $m(L(s_x, s_y))$ denote this path and its path metric, respectively. For convenience, we call $l(L(s_x, s_y))$ and $m(L(s_x, s_y))$ the *label* and *metric* of $L(s_x, s_y)$, respectively. We form a table that stores the label and metric for each distinct composite path $L(s_x, s_y)$ in $T_{x,y}$ (or a coset in $p_{x,y}(C)/C_{x,y}^{tr}$). This table is called the *composite path metric table*, denoted by $\text{CPMT}_{x,y}$, for the trellis section $T_{x,y}$ (or for the partition $p_{x,y}(C)/C_{x,y}^{tr}$). Because the partition $p_{0,n}(C)/C_{0,n} = C/C$ consists of C only, $\text{CPMT}_{0,n}$ contains only the codeword in C that has the largest metric. This is the most likely codeword with respect to the received sequence $\mathbb{r} = (r_0, r_1, \cdots, r_{n-1})$. When $\text{CPMT}_{0,n}$ is constructed, the decoding is completed, and $\text{CPMT}_{0,n}$ contains the decoded codeword.

A straightforward method for constructing $\text{CPMT}_{x,y}$ is to compute the metrics of all vectors in the punctured code $p_{x,y}(C)$ and then to find the vector with the largest metric for every coset in $p_{x,y}(C)/C_{x,y}^{tr}$ by comparing the metrics of vectors in the coset. This direct construction method is efficient only when $y - x$ is small and should be used at the beginning of the recursion process. When $y - x$ is large, $\text{CPMT}_{x,y}$ is constructed from $\text{CPMT}_{x,z}$ and $\text{CPMT}_{z,y}$ for a properly chosen integer z with $x < z < y$.

Table combining is based on the trellis connectivity structure developed in Section 9.4 and shown in Figure 9.11. Consider two states s_x and s_y in the code trellis T with $s_x \in \Sigma_x(C)$ and $s_y \in \Sigma_y(C)$ that are connected by the composite path $L(s_x, s_y)$. Let

$$\Sigma_z(s_x, s_y) \triangleq \{s_z^{(1)}, s_z^{(2)}, \cdots, s_z^{(\mu)}\} \tag{14.7}$$

denote the subset of states in $\Sigma_z(C)$ through which the paths in $L(s_x, s_y)$ connect state s_x to state s_y, as shown in Figure 14.1. Then, it follows from (9.28) and (9.29)

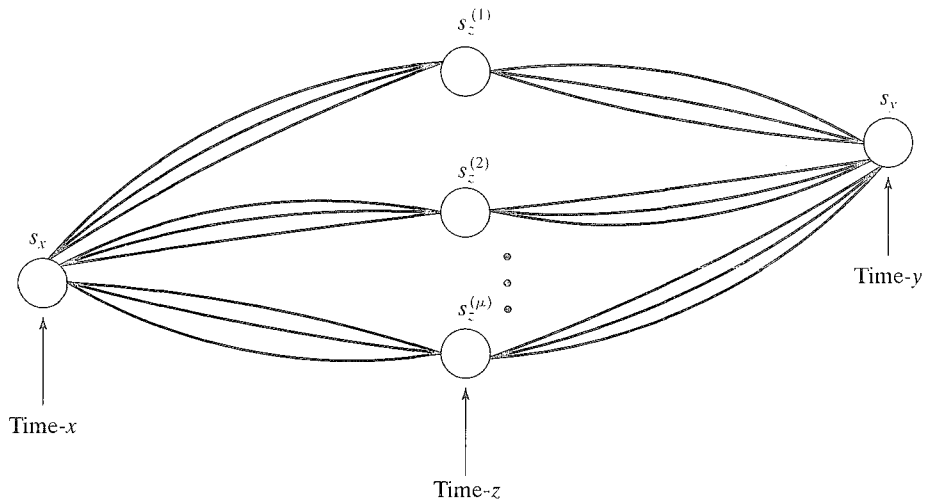

FIGURE 14.1: Connection between two states.

that

$$L(s_x, s_y) = \bigcup_{s_z^{(i)} \in \Sigma_z(s_x, s_y)} L(s_x, s_z^{(i)}) \circ L(s_z^{(i)}, s_y), \tag{14.8}$$

where $L(s_x, s_z^{(i)})$ is the composite path connecting state s_x to state $s_z^{(i)}$, and $L(s_z^{(i)}, s_y)$ is the composite path connecting state $s_z^{(i)}$ to state s_y. $L(s_x, s_z^{(i)})$ and $L(s_z^{(i)}, s_y)$ are cosets in partitions $p_{x,z}(C)/C_{x,z}^{tr}$ and $p_{z,y}(C)/C_{z,y}^{tr}$, respectively. It follows from (14.8) and the definitions of metric and label of a composite path that

$$m(L(s_x, s_y)) = \max_{s_z^{(i)} \in \Sigma_z(s_x, s_y)} \{m(L(s_x, s_z^{(i)})) + m(L(s_z^{(i)}, s_y))\}, \tag{14.9}$$

and

$$l(L(s_x, s_y)) = l(L(s_x, s_z^{(i_{\max})})) \circ l(L(s_z^{(i_{\max})}, s_y)), \tag{14.10}$$

where i_{max} denotes the index i for which the sum in (14.9) takes its maximum. Because the metrics $m(L(s_x, s_z^{(i)}))$ and $m(L(s_z^{(i)}, s_y))$ and labels $l(L(s_x, s_z^{(i_{max})}))$ and $l(L(s_z^{(i_{max})}, s_y))$ are stored in $\text{CPMT}_{x,z}$ and $\text{CPMT}_{z,y}$, respectively, (14.9) and (14.10) show that the $\text{CPMT}_{x,y}$ for the trellis section $T_{x,y}$ can be constructed from $\text{CPMT}_{x,z}$ and $\text{CPMT}_{z,y}$ for the trellis sections $T_{x,z}$ and $T_{z,y}$.

Based on the structural properties of trellises for block codes developed in Section 9.4, we can readily show that the size of state set $\Sigma_z(s_x, s_y)$ is given by

$$\mu = |\Sigma_z(s_x, s_y)| = 2^{k(C_{x,y}) - k(C_{x,z}) - k(C_{z,y})}. \tag{14.11}$$

Therefore, computation of the metric $m(L(s_x, s_y))$ from (14.9) using $\text{CPMT}_{x,z}$ and $\text{CPMT}_{z,y}$, requires μ additions and $\mu - 1$ comparisons. It follows from (14.5) that the total number of computations required to form $\text{CPMT}_{x,y}$ from $\text{CPMT}_{x,z}$ and $\text{CPMT}_{z,y}$ is

$$(2\mu - 1) \cdot 2^{k(p_{x,y}(C)) - k(C_{x,y})}; \tag{14.12}$$

however, if the metric $m(L(s_x, s_y))$ is computed directly from the paths in $L(s_x, s_y)$, we need to compute $|C_{x,y}^{tr}| = 2^{k(C_{x,y})}$ path metrics and perform $|C_{x,y}^{tr}| - 1$ comparisons. Because computation of each path metric requires $y - x - 1$ additions, the total number of real operations required to compute $m(L(s_x, s_y))$ is

$$(y - x - 1)2^{k(C_{x,y})} + 2^{k(C_{x,y})} - 1 = (y - x)2^{k(C_{x,y})} - 1, \tag{14.13}$$

which is much larger than $2\mu - 1$ (see (14.11)) for large $y - x$. Hence, constructing the metric table $\text{CPMT}_{x,y}$ from $\text{CPMT}_{x,z}$ and $\text{CPMT}_{z,y}$ requires many fewer computations than direct construction of $\text{CPMT}_{x,y}$ from vectors in $p_{x,y}(C)$ and cosets in $p_{x,y}(C)/C_{x,y}^{tr}$.

In principle, we can construct the $\text{CPMT}_{x,y}$ using the trellis section $T_{x,y}$ as follows:

1. For each coset $D \in p_{x,y}(C)/C_{x,y}^{tr}$, identify a state pair (s_x, s_y) such that $L(s_x, s_y) = D$.

2. Determine the state set $\Sigma_z(s_x, s_y)$.
3. Compute the metric $m(L(s_x, s_y))$ and label $l(L(s_x, s_y))$ from (14.9) and (14.10), respectively.

This process requires constructing the entire code trellis and examining the section from time-x to time-y. For a long code, it is difficult to construct a large trellis T and examine the section $T_{x,y}$. Furthermore, the total number of composite paths in $T_{x,y}$ can be very large, with only a small fraction of distinct composite paths. Examining $T_{x,y}$ to execute steps 1 to 3 can be very time-consuming and inefficient. Consequently, decoding implementation will be complex and costly.

To overcome the complexity problem and to facilitate the computation of (14.9), we construct a much simpler special two-section trellis for the punctured code $p_{x,y}(C)$ with section boundary locations at x, z, and y that has multiple "final" states at time-y, one for each coset in $p_{x,y}(C)/C_{x,y}^{tr}$. This special two-section trellis contains only the needed information for constructing the metric table $\text{CPMT}_{x,y}$ from $\text{CPMT}_{x,z}$ and $\text{CPMT}_{z,y}$. Let $T(\{x, z, y\})$ denote this special trellis. Let Σ_z and Σ_y denote the state spaces of $T(\{x, z, y\})$ at time-z and time-y, respectively. We require that $T(\{x, z, y\})$ have the following structural properties:

1. It has an initial state, $s_{x,0}$, at time-x.
2. There is a one-to-one correspondence between the states in $\sum_z$ and the cosets in $p_{x,z}(C)/C_{x,z}^{tr}$. Let D_z denote a coset in $p_{x,z}(C)/C_{x,z}^{tr}$, and $s(D_z)$ denote its corresponding state at time-z. Then, the composite path connecting $s_{x,0}$ to $s(D_z)$ is $L(s_{x,0}, s(D_z)) = D_z$.
3. There is a one-to-one correspondence between the states in the state space Σ_y and cosets in $p_{x,y}(C)/C_{x,y}^{tr}$. Let D_y denote a coset in $p_{x,y}(C)/C_{x,y}^{tr}$, and $s(D_y)$ denote its corresponding state at time-y. Then, the composite path connecting the initial state $s_{x,0}$ to state $s(D_y)$ is $L(s_{x,0}, s(D_y)) = D_y$. If a state $s(D_z)$ at time-z and a state $s(D_y)$ at time-y are connected, then the composite path $L(s(D_z), s(D_y))$ is a coset in $p_{z,y}(C)/C_{z,y}^{tr}$. Every coset in $p_{z,y}(C)/C_{z,y}^{tr}$ appears as a composite path between a state in Σ_z and a state in Σ_y.
4. For every state $s(D_y)$ at time-y, there is set of μ states in Σ_z that connect the initial state $s_{x,0}$ to state $s(D_y)$. Let

$$\Sigma_z(s_{x,0}, s(D_y)) = \{s(D_z^{(1)}), s(D_z^{(2)}), \cdots, s(D_z^{(\mu)})\} \tag{14.14}$$

denote this set of states. Then,

$$\begin{aligned} D_y &= L(s_{x,0}, s(D_y)) \\ &= \bigcup_{s(D_z^{(i)}) \in \sum_z (s_{x,0}, s(D_y))} L(s_{x,0}, s(D_z^{(i)})) \circ L(s(D_z^{(i)}), s(D_y)). \end{aligned} \tag{14.15}$$

From the structural properties of the special two-section trellis for the punctured code $p_{x,y}(C)$ we see that (1) for every state $s(D_z) \in \Sigma_z$, its state metric (or composite path metric) $m(L(s_{x,0}, s(D_z)))$ and label $l(L(s_{x,0}, s(D_z)))$ are given in metric table $\text{CPMT}_{x,z}$; and (2) for each composite path between a state $s(D_z)$ at time-z and an adjacent state $s(D_y)$ at time-y, its metric $m(L(s(D_z), s(D_y)))$ and

label $l(L(s(D_z), s(D_y)))$ are given in $\mathrm{CPMT}_{z,y}$. It follows from (14.15) and the structural properties of the special two-section trellis $T(\{x, z, y\})$ that for each coset $D_y \in p_{x,y}(C)/C^{tr}_{x,y}$, the metric $m(L(s_{x,0}, s(D_y)))$ is given by

$$m(L(s_{x,0}, s(D_y))) = \max_{s(D_z^{(i)}) \in \sum_z(s_{x,0}, s(D_y))} \{m(L(s_{x,0}, s(D_z^{(i)}))) + m(L(s(D_z^{(i)}), s(D_y)))\}. \tag{14.16}$$

Equation (14.16) is simply equivalent to (14.9). Therefore, the metric table $\mathrm{CPMT}_{x,y}$ can be constructed from $\mathrm{CPMT}_{x,z}$ and $\mathrm{CPMT}_{z,y}$ by using the special two-section trellis $T(\{x, z, y\})$ for the punctured code $p_{x,y}(C)$. For each state $s(D_y) \in \Sigma_y$, the state set $\Sigma_z(s_{x,0}, s(D_y))$ can be easily identified from the trellis $T(\{x, z, y\})$. In general, this special trellis $T(\{x, z, y\})$ is much simpler than the section $T_{x,y}$ of the entire code trellis T from time-x to time-y. As a result, the construction of $\mathrm{CPMT}_{x,y}$ is much simpler.

The special two-section trellis $T(\{x, z, y\})$ for the punctured code $p_{x,y}(C)$ is constructed as follows [8, 15]. We choose a basis $\{\mathbb{g}_1, \mathbb{g}_2, \cdots, \mathbb{g}_{k(p_{x,y}(C))}\}$ of $p_{x,y}(C)$ such that the first $k(C^{tr}_{x,y}) = k(C_{x,y})$ vectors form a basis of $C^{tr}_{x,y}$. Let

$$n_{x,y} \triangleq y - x + k(p_{x,y}(C)) - k(C_{x,y}). \tag{14.17}$$

We form the following $k(p_{x,y}(C)) \times n_{x,y}$ matrix:

$$\mathbb{G}(x, y) = \begin{bmatrix} \mathbb{g}_1 & \vdots & \\ \vdots & \vdots & \mathbb{0} \\ \mathbb{g}_{k(C_{x,y})} & \vdots & \\ \cdots & \cdots & \cdots \\ \mathbb{g}_{k(C_{x,y})+1} & \vdots & \\ \vdots & \vdots & I \\ \mathbb{g}_{k(p_{x,y}(C))} & \vdots & \end{bmatrix}, \tag{14.18}$$

where $\mathbb{0}$ denotes the $k(C_{x,y}) \times (k(p_{x,y}(C)) - k(C_{x,y}))$ all-zero matrix, and I denotes the identity matrix of dimension $k(p_{x,y}(C)) - k(C_{x,y})$. Let $C(x, y)$ be the binary linear code of length $n_{x,y}$ generated by $\mathbb{G}(x, y)$. We construct a three-section trellis $T(\{x, z, y, x + n_{x,y}\})$ for $C(x, y)$ with section boundary locations at x, z, y, and $x + n_{x,y}$, as shown in Figure 14.2. Then, the first two sections of $T(\{x, z, y, x + n_{x,y}\})$ give the desired two-section trellis $T(\{x, z, y\})$ for $p_{x,y}(C)$. In construction, we do not need to complete the trellis $T(\{x, z, y, x + n_{x,y}\})$. We need only construct the first two sections from time-x to time-y.

In fact, from (14.16) and the structural properties of $T(\{x, z, y\})$, we need only the second section of $T(\{x, z, y\})$ to carry out the computations of (14.16) and to form the metric table $\mathrm{CPMT}_{x,y}$. For convenience, we denote this one-section trellis $T_x(z, y)$. We assign each state $s(D_z)$ of $T_x(z, y)$ at time-z a metric $m(L(s_{x,0}, s(D_z)))$ and a label $l(L(s_{x,0}, s(D_z)))$ based on $\mathrm{CPMT}_{x,z}$. Table $\mathrm{CPMT}_{z,y}$ gives the composite

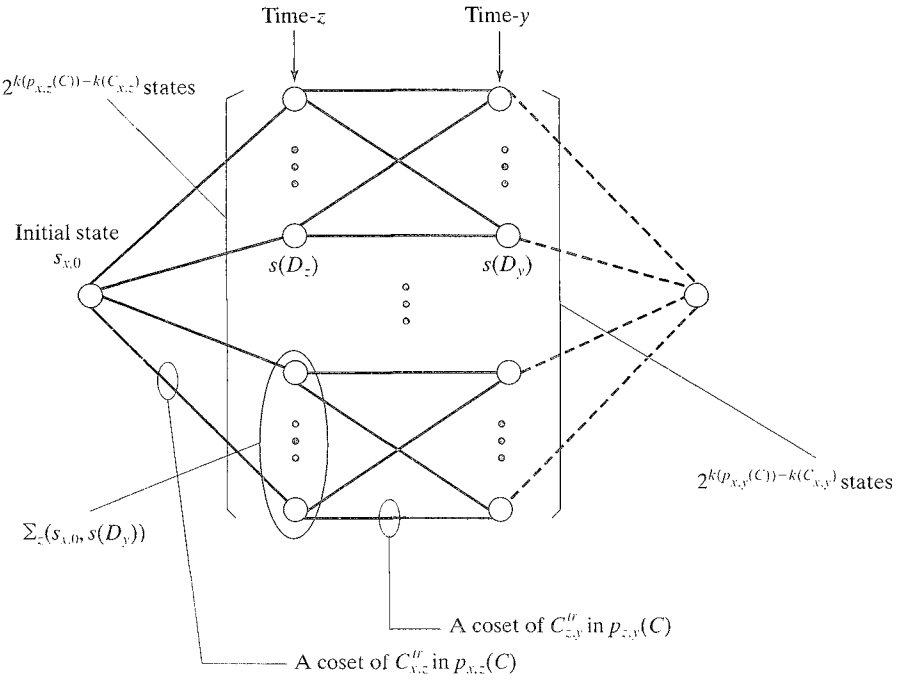

FIGURE 14.2: Structure of the trellis diagram $T(\{x, z, y, x + n_{x,y}\})$.

path metrics of $T_x(z, y)$ between time-z and time-y. With all this information, we can easily construct $\text{CPMT}_{x,y}$ by carrying out the computations of (14.16).

EXAMPLE 14.2

Let C be the $RM(2, 4)$ code, a (16, 11) code, whose trellis structure we studied in Example 9.10. Set $x = 4$, $y = 12$, and $z = 8$. Then, $p_{4,12}(C) = RM(2, 3)$, $C_{4,12}^{tr} = RM(1, 3)$, $C_{4,8}^{tr} = C_{8,12}^{tr} = RM(0, 2)$, and $n_{4,12} = 12$. $C(4, 12)$ is the (11, 9) linear block code generated by

$$\mathbb{G}(4, 12) = \left[\begin{array}{cccccccc|ccc} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ - & - & - & - & - & - & - & - & - & - & - \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \end{array}\right].$$

By simple row operations, we can put $\mathbb{G}(4, 12)$ in trellis-oriented form. The one-section trellis $T_4(8, 12)$ consists of two 4-state parallel and structurally identical subtrellises, called *parallel components*. One of the components is depicted in Figure 14.3. The other can be obtained by adding (0001) to each branch label. From

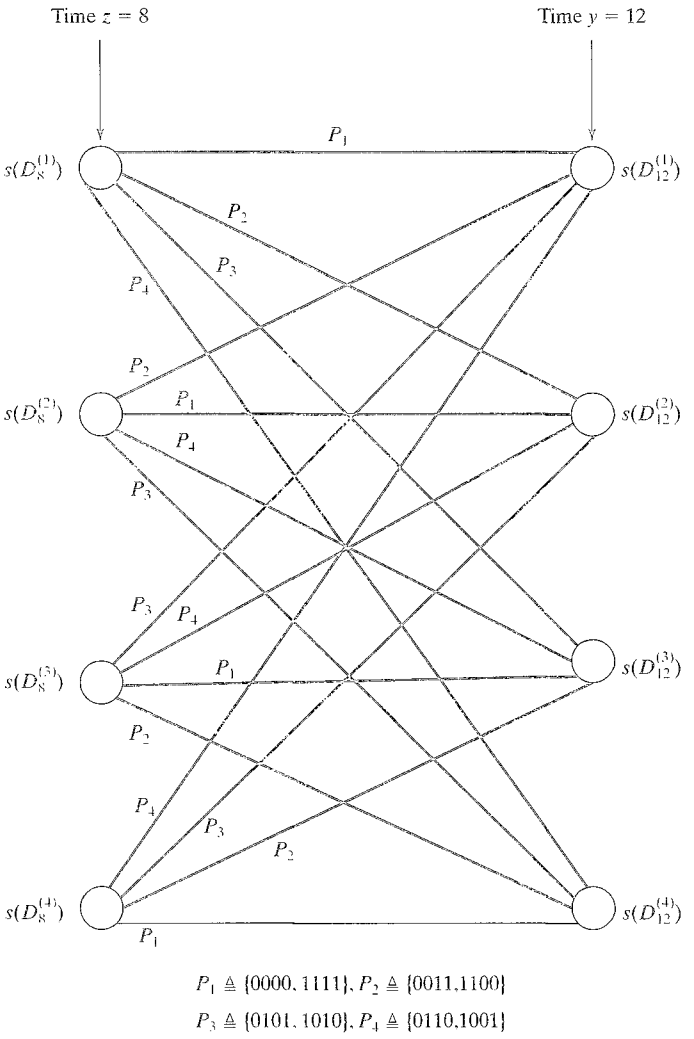

FIGURE 14.3: A parallel component of $T_4(8, 12)$ for the $RM(2, 4)$ code.

Figure 14.3 we can readily identify the state set $\sum_8(s_{4,0}, s(D_{12}))$ for each state $s(D_{12})$ at time-12. Each set consists of 4 states at time-8.

14.2.2 An RMLD Algorithm

Based on the foregoing method for constructing composite path metric tables, a RMLD algorithm can be formulated [15]. Let RMLD(x, y) denote the recursive algorithm for constructing the composite path metric table $\text{CPMT}_{x,y}$ for $p_{x,y}(C)/C_{x,y}^{ir}$. This algorithm uses two procedures, denoted by $MakeCPMT(x, y)$ and $CombCPMT(x, y; z)$, to construct $\text{CPMT}_{x,y}$. The MakeCPMT(x, y) procedure constructs $\text{CPMT}_{x,y}$ directly from vectors in $p_{x,y}(C)$. The CombCPMT$(x, y; z)$

procedure constructs $\text{CPMT}_{x,y}$ by combining tables $\text{CPMT}_{x,z}$ and $\text{CPMT}_{z,y}$ using (14.9) (or (14.16)) and (14.10), where $x < z < y$. Because tables $\text{CPMT}_{x,z}$ and $\text{CPMT}_{z,y}$ are constructed by executing RMLD(x, z) and RMLD(z, y), we can express the procedure CombCPMT$(x, y; z)$ in recursive form:

$$\text{CombCPMT(RMLD}(x, z)\text{,RMLD}(z, y)\text{)}.$$

Algorithm RMLD(x, y) is used as follows:
Construct $\text{CPMT}_{x,y}$ using the less complex one of the following two options:

1. Execute MakeCPMT(x, y).
2. Execute CombCPMT(RMLD(x, z),RMLD(z, y)), where z with $x < z < y$ is selected to minimize computational complexity.

Decoding is accomplished by executing RMLD$(0, n)$.

To start the RMLD algorithm to decode a code, we first divide the code into short sections. The metric table for each of these short sections is constructed by executing the MakeCPMT procedure. This forms the *bottom* (or the *beginning*) of the recursion process. Then, we execute the RMLD(x, y) algorithm repeatedly to form metric tables for longer sections of the code until the table $\text{CPMT}_{0,n}$ is obtained. Figure 14.4 depicts such a recursion process. We see that the RMLD algorithm allows *parallel/pipeline* processing of received words. This speeds decoding process.

As pointed out earlier, the simplest and most straightforward way to construct the $\text{CPMT}_{x,y}$ directly is to compute the metrics of all the vectors in the punctured code $p_{x,y}(C)$ independently and then find the vector with the largest metric for each coset in $p_{x,y}(C)/C^{tr}_{x,y}$ by comparing the metrics of vectors in the coset. Each surviving vector and its metric are stored in the $\text{CPMT}_{x,y}$. Let MakeCPMT-I(x, y) denote this make-table procedure. The computational complexity of this procedure can be easily evaluated. There are $2^{k(p_{x,y}(C))-k(C_{x,y})}$ cosets. The number of computations required to determine the metric of each coset is given by (14.13). Then, the total number of computations required by the MakeCPMT-I(x, y) procedure is

$$\Psi_M^{(I)}(x, y) = 2^{k(p_{x,y}(C))-k(C_{x,y})} \cdot [(y - x)2^{k(C_{x,y})} - 1]. \tag{14.19}$$

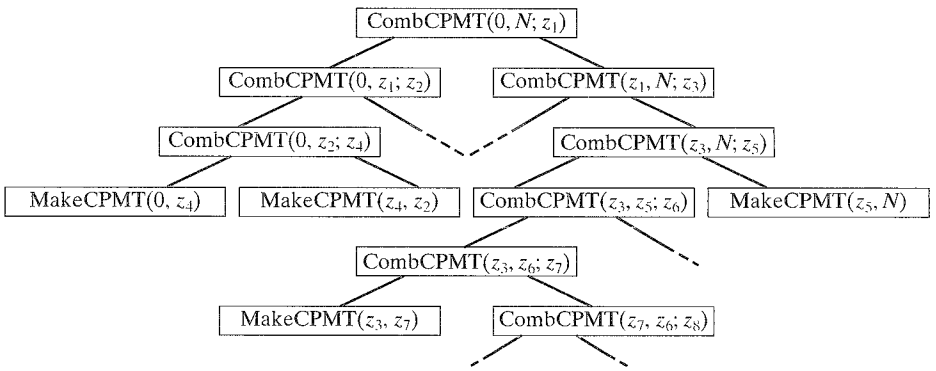

FIGURE 14.4: Illustration of the recursion process of the RMLD algorithm.

The CombCPMT$(x, y; z)$ procedure simply performs the computation of (14.9) (or (14.16)) and finds the label of (14.10) based on the one-section trellis $T_x(z, y)$. A straightforward procedure to achieve this is to apply the conventional add-compare-select (ACS) procedure that is used in the Viterbi algorithm. For each coset D_y in $p_{x,y}(C)/C'_{x,y}$, the metric sum $m(L(s_{x,0}, s(D_z^{(i)}))) + m(L(s(D_z^{(i)}), s(D_y)))$ is computed for every state $s(D_z^{(i)}) \in \sum_z (s_{x,0}, s(D_y))$, and $m(D_y) = m(L(s_{x,0}, s(D_y)))$ is found by comparing all the computed metric sums. This procedure is called the *CombCPMT-V*$(x, y; z)$ procedure. The total number of computations required by this procedure to construct $\text{CPMT}_{x,y}$ is given by (14.12):

$$\Psi_c^{(V)}(x, y; z) = 2^{k(p_{x,y}(C))-k(C_{x,y})} \cdot [2^{k(C_{x,y})-k(C_{x,z})-k(C_{z,y})+1} - 1]. \tag{14.20}$$

The RMLD algorithm that uses MakeCPMT-I(x, y) and CombCPMT-V$(x, y; z)$ procedures to construct metric tables is called the *RMLD-(I,V)* algorithm. Because both MakeCPMT-I(x, y) and CombCPMT-V$(x, y; z)$ are very simple, the RMLD-(I,V) algorithm is very easy to implement in either software or hardware. A computationwise more efficient CombCPMT$(x, y; z)$ procedure can be devised if detail structure of the one-section trellis $T_x(z, y)$ is used. Such a procedure has been proposed in [15].

From (14.19) and (14.20) we can readily see that (1) if $y - x > 2$, then for any choice of z with $x < z < y$, the CombCPMT-V$(x, y; z)$ procedure requires fewer computations to form the $\text{CPMT}_{x,y}$ than the MakeCPMT-I(x, y) procedure; and (2) if $y - x = 2$, the computational complexity of CombCPMT-V$(x, y; x + 1)$ and that of MakeCPMT-I(x, y) are the same. This simply says that the MakeCPMT-I(x, y) is efficient only when $y - x$ is small and only when used at the beginning to form metric tables to start the recursion process. When $y - x$ is large, the CombCPMT-V$(x, y; z)$ procedure should be used to construct metric tables. At the beginning of the recursion process, $y - x$ is small and few computations are done by the MakeCPMT-I(x, y) procedure during the entire decoding process. Therefore, the major computation is carried out by the CombCPMT-V$(x, y; z)$ procedure.

14.2.3 Optimum Sectionalization

The overall computational complexity of the RMLD algorithm depends on the sectionalization of a code trellis (or the choices of z's). A sectionalization that results in the smallest overall computational complexity is called an *optimum sectionalization* [15].

Let $\Psi_{\min}(x, y)$ denote the smallest number of computations (additions and comparisons) required to construct the $\text{CPMT}_{x,y}$. An addition operation and a comparison operation are assumed to have equal weight (same complexity). It follows from the RMLD(x, y) algorithm given in the previous section that

$$\Psi_{min}(x, y) \triangleq \begin{cases} \Psi_M^{(I)}(x, y), & \text{if } y = x + 1, \\ \min\{\Psi_M^{(I)}(x, y), \min_{x<z<y}\{\Psi_{R,min}(x, y; z)\}\}, & \text{otherwise,} \end{cases} \tag{14.21}$$

where

$$\Psi_{R,min}(x, y; z) \triangleq \Psi_{min}(x, z) + \Psi_{min}(z, y) + \Psi_c^{(V)}(x, y; z). \tag{14.22}$$

TABLE 14.1: Computational complexities of the RMLD(I,V) algorithm and the Viterbi algorithm for decoding some RM codes.

Code	Viterbi based on 64-section trellis	RMLD-(I,V)	Viterbi based on Lafourcade–Vardy algorithm
$RM(2, 6)$ $(64, 22)$	425,209	78,209	101,786
$RM(3, 6)$ $(64, 42)$	773,881	326,017	538,799
$RM(4, 6)$ $(64, 57)$	7,529	5,281	6,507

Then, the total number of computations required to decode a received word is given by $\Psi_{\min}(0, n)$.

Using (14.19) through (14.22) we can compute $\Psi_{\min}(x, y)$ for every (x, y) with $0 \leq x < y \leq n$ as follows: We compute the values of $\Psi_{\min}(x, x+1)$ for $0 \leq x < n$ using (14.19). For an integer i with $0 \leq x < x+i \leq n$, we can compute $\Psi_{\min}(x, x+i)$ from $\Psi_{\min}(x', y')$ with $y' - x' < i$, (14.19), and (14.20). By keeping track of the values of z thus selected we can find an optimum sectionalization.

The RMLD-(I,V) algorithm with optimum sectionalization is more efficient than the Viterbi algorithm based on the Lafourcade–Vardy's optimum sectionalization [3]. A few examples are given in Table 14.1.

From Table 14.1 we see that both the RMLD-(I,V) and Viterbi–Lafourcade–Vardy algorithms are more efficient than the Viterbi algorithm based on the bit-level trellis. The RMLD-(I,V) algorithm not only is more efficient than the Viterbi–Lafourcade–Vardy algorithm but it also allows parallel/pipeline processing to speed up decoding, which is important in high-speed communications.

14.3 A SUBOPTIMUM ITERATIVE DECODING ALGORITHM BASED ON A LOW-WEIGHT SUBTRELLIS

Trellises for long block codes have very large state and branch complexities. These complexities grow exponentially with code length. Consequently, maximum likelihood decoding of long codes based on full-code trellises would be very difficult if not impossible to implement. For this reason it is very desirable to devise trellis-based suboptimum decoding algorithms to provide an efficient trade-off between error performance and decoding complexity.

This section presents a suboptimum iterative decoding algorithm based on low-weight subtrellis searches that was proposed in [17]. This algorithm consists of three simple steps. First, a simple method is used to generate a sequence of candidate codewords, one at a time, based on the reliability information of the received symbols. When a candidate codeword is generated, it is tested based on an optimality condition. If it satisfies the optimality condition, then it is the most likely codeword, and decoding stops. If the optimality test fails, then the smallest region centered on the tested candidate codeword that contains the most likely codeword is determined. If this region is small, a search is conducted through a low-weight

subtrellis for the given code using a trellis-based decoding algorithm such as Viterbi or RMLD. If the search region is too large, a new candidate codeword is generated, and the optimality test and search are renewed. This process continues until either the most likely codeword is found or candidate codewords are exhausted (or a stopping criterion is met), and the decoding process is then terminated.

14.3.1 Generation of Candidate Codewords

Let C be a binary (n, k) linear code with minimum distance d_{min} to be decoded. Let $\mathbb{r} = (r_0, r_1, \cdots, r_{n-1})$ be the soft-decision received sequence and $\mathbb{z} = (z_0, z_1, \cdots, z_{n-1})$ be its corresponding hard-decision binary vector. A simple method for generating candidate codewords is to use a set of test error patterns to modify the hard-decision received vector $\mathbb{z}$ and then decode each modified vector with an algebraic decoder. The choice of the test error patterns determines the performance and effectiveness of the decoding algorithm. Obviously, only the most probable error patterns should be used as the test error patterns, and they should be used in likelihood order to generate the candidate codewords for test. Let p be a positive integer no greater than n, and Q_p denote the set of the p least reliable positions of the soft-decision received sequence $\mathbb{r}$ as discussed in Section 10.2. Let E denote the set of 2^p binary error patterns of length n with errors confined to the positions in Q_p. The error patterns in this set are more likely to occur than the other error patterns. In Chase decoding algorithm-2 [18], as presented in 10.4.2, p is chosen to be $\lfloor d_{min}/2 \rfloor$, and E consists of $2^{\lfloor d_{min}/2 \rfloor}$ test error patterns. In the following section, Chase algorithm-2 is used to generate candidate codewords for the optimality test.

14.3.2 Optimality Test and Search Region

The optimality conditions presented in Section 10.3 are used to test the candidate codewords. Let w_k be the kth nonzero weight in the weight profile $W = \{\omega_0 = 0, \omega_1, \ldots, \omega_m\}$ of code C. Let $\mathbb{v}$ be a candidate codeword for test. We define

$$\rho_k \triangleq \omega_k - |D_1(\mathbb{v})|, \tag{14.23}$$

$$G(\mathbb{v}, \omega_k) \triangleq \sum_{i \in D_0^{(\rho_k)}(\mathbb{v})} |r_i|, \tag{14.24}$$

and

$$R(\mathbb{v}, \omega_k) \triangleq \{\mathbb{v}' \in C : d(\mathbb{v}', \mathbb{v}) < \omega_k\}, \tag{14.25}$$

where sets $D_1(\mathbb{v})$ and $D_0^{(\rho_k)}(\mathbb{v})$ are defined in Section 10.3 ((10.23) and (10.29)), and $d(\mathbb{v}', \mathbb{v})$ denotes the Hamming distance between $\mathbb{v}'$ and $\mathbb{v}$. Then, we can restate the condition given by (10.33) (Theorem 10.1) as follows: for a candidate codeword $\mathbb{v} \in C$ and a nonzero weight $\omega_k \in W$, if the correlation discrepancy $\lambda(\mathbb{r}, \mathbb{v})$ between $\mathbb{v}$ and the received sequence $\mathbb{r}$ satisfies the condition

$$\lambda(\mathbb{r}, \mathbb{v}) \leq G(\mathbb{v}; \omega_k), \tag{14.26}$$

then the most likely codeword $\mathbb{v}_{ML}$ is in the region $R(\mathbb{v}, \omega_k)$. The condition given by (14.26) simply defines a region in which the most likely codeword $\mathbb{v}_{ML}$ can be

found. It says that $\mathbf{v}_{ML}$ is among those codewords in C that are at distance ω_{k-1} or less from the candidate codeword $\mathbf{v}$; that is,

$$d(\mathbf{v}, \mathbf{v}_{ML}) \leq \omega_{k-1}. \tag{14.27}$$

If ω_{k-1} is small, we can search in the region $R(\mathbf{v}, \omega_k)$ to find $\mathbf{v}_{ML}$. If ω_{k-1} is too large, then it is better to generate another candidate codeword $\mathbf{v}'$ for testing in the hope that the search region $R(\mathbf{v}', \omega_k)$ is small.

From (14.26) we readily see that

1. If $k = 1$, the candidate codeword $\mathbf{v}$ is the most likely codeword $\mathbf{v}_{ML}$.
2. If $k = 2$, the most likely codeword $\mathbf{v}_{ML}$ is either the candidate codeword $\mathbf{v}$ or a nearest neighbor of $\mathbf{v}$.

Therefore, we have the following two optimality conditions:

1. If $\lambda(\mathbf{r}, \mathbf{v}) \leq G(\mathbf{v}, \omega_1)$, then $\mathbf{v} = \mathbf{v}_{ML}$.
2. If $G(\mathbf{v}, \omega_1) < \lambda(\mathbf{r}, \mathbf{v}) \leq G(\mathbf{v}, \omega_2)$, then $\mathbf{v}_{ML}$ is at a distance no greater than the minimum distance $d_{\min} = \omega_1$ from $\mathbf{v}$.

The first condition is a sufficient condition for optimality of a candidate codeword. The second condition is a sufficient condition for $\mathbf{v}_{ML}$ to be a nearest neighbor of a tested candidate codeword. We call $G(\mathbf{v}, \omega_1)$ and $G(\mathbf{v}, \omega_2)$ the optimality and nearest-neighbor test thresholds, respectively. They are used for testing a candidate codeword to determine whether it is the most likely codeword or a search for the most likely codeword among its nearest neighbors is needed.

14.3.3 An Iterative Decoding Algorithm Based on a Minimum-Weight Trellis Search

Suppose a candidate codeword $\mathbf{v}$ is tested, and the most likely codeword $\mathbf{v}_{ML}$ is found in the region $R(\mathbf{v}, \omega_2)$. Then, $\mathbf{v}_{ML}$ can be found by searching through the minimum-weight subtrellis $T_{\min}(\mathbf{v})$ centered on $\mathbf{v}$ using either the Viterbi or the RMLD algorithm [17].

The decoding algorithm is iterative in nature and consists of the following key steps:

1. A candidate codeword $\mathbf{v}$ is generated by using a test error pattern $\mathbf{e}$ in the set E.
2. The optimality test or the nearest-neighbor test is performed for each generated candidate codeword $\mathbf{v}$.
3. If the optimality test succeeds, decoding stops. If the optimality fails, but the nearest-neighbor test succeeds, a search in the region $R(\mathbf{v}, \omega_2)$ is initiated to find the most likely codeword $\mathbf{v}_{ML}$.
4. If both tests fail, a new candidate codeword is generated.

Suppose the most likelihood $\mathbf{v}_{ML}$ has not been found at the end of the $(j-1)$th decoding iteration. Then, the jth decoding iteration is initiated. Let $\mathbf{v}_{best}$ and $\lambda(\mathbf{r}, \mathbf{v}_{best})$ denote the best codeword and its correlation discrepancy that have been found so far and are stored in the decoder memory. The jth decoding iteration consists of the following steps:

Step 1. Fetch $\mathbf{e}_j$ from E and decode $\mathbf{e}_j + \mathbf{z}$ into a codeword $\mathbf{v} \in C$. If the decoding succeeds, go to step 2; otherwise, go to step 1.

Step 2. If $\lambda(\mathbf{r}, \mathbf{v}) \leq G(\mathbf{v}, \omega_1)$, $\mathbf{v}_{ML} = \mathbf{v}$; stop the decoding process. Otherwise, go to step 3.

Step 3. If $\lambda(\mathbf{r}, \mathbf{v}) \leq G(\mathbf{v}, \omega_2)$, search $T_{min}(\mathbf{v})$ to find $\mathbf{v}_{ML}$ and stop the decoding process; otherwise, go to step 4.

Step 4. If $\lambda(\mathbf{r}, \mathbf{v}) < \lambda(\mathbf{r}, \mathbf{v}_{best})$, replace $\mathbf{v}_{best}$ with $\mathbf{v}$ and $\lambda(\mathbf{r}, \mathbf{v}_{best})$ by $\lambda(\mathbf{r}, \mathbf{v})$; otherwise, go to step 5.

Step 5. If $j < |E|$, go to step 1; otherwise, search $T_{min}(\mathbf{v}_{best})$ and output the codeword with the least correlation discrepancy. Stop.

The decoding process is depicted by the flowchart shown in Figure 14.5. It requires at most one minimum-weight trellis search. The only case for which the decoded codeword is not the most likely codeword $\mathbf{v}_{ML}$ is the output from the search of the minimum-weight trellis centered at $\mathbf{v}_{best}$. For this decoding algorithm, the main cause of a decoding error is that the number of errors in the $n - \lfloor d_{min}/2 \rfloor$ most reliable positions of $\mathbf{z}$ is larger than $\lfloor (d_{min} - 1)/2 \rfloor$.

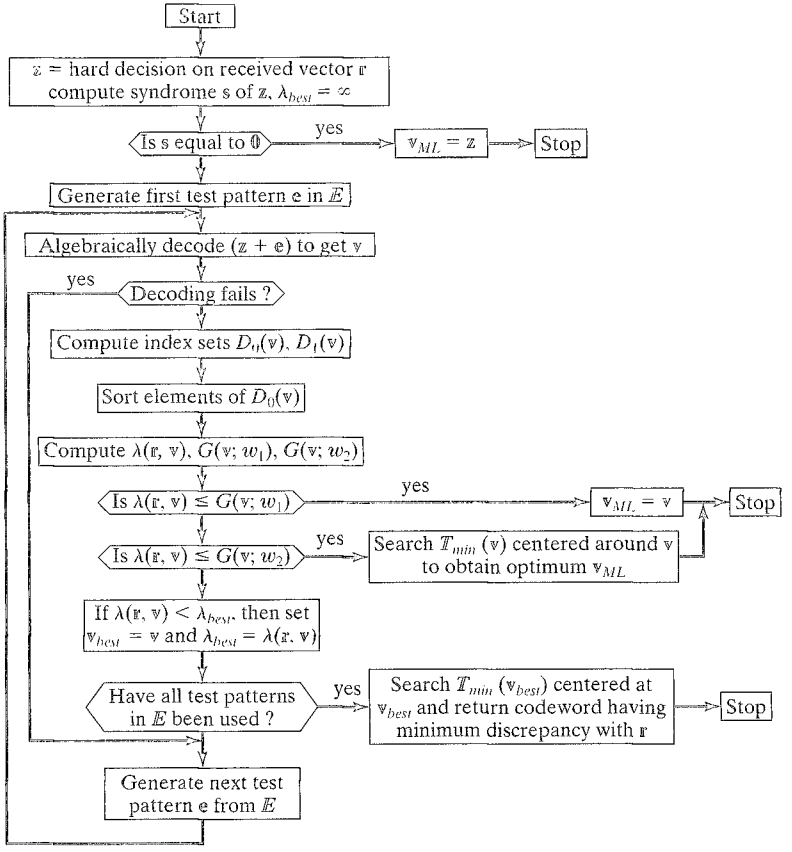

FIGURE 14.5: Flowchart for the iterative decoding algorithm.

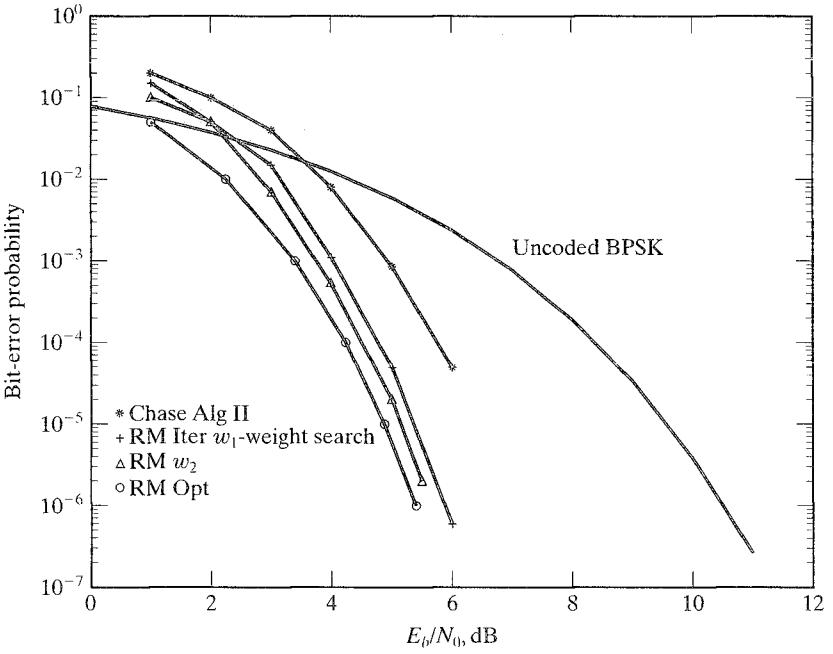

FIGURE 14.6: Error performances of the iterative decoding (w_1-, w_2-weight trellis search) for the (64, 42, 8) RM code.

Because this iterative decoding algorithm uses the Chase algorithm-2 to generate candidate codewords for the optimality test and minimum-weight trellis search, it may be regarded as an improved Chase algorithm-2 and hence achieves asymptotically optimal (MLD) error performance with a faster rate.

Consider the (64, 42) RM code with a minimum distance of 8. This code can be decoded with majority-logic decoding. Therefore, with the iterative decoding described here, the algebraic decoder is very simple. The error performances of this code with various decoding algorithms are shown in Figure 14.6. We see that it has a 0.5-dB loss in coding gain at BER of 10^{-6} compared with the optimum MLD, but it outperforms the Chase algorithm-2 by more than 1 dB from 10^{-3} BER down.

14.3.4 Computational Complexity

Assume that the computational complexity of the algebraic decoding to generate the candidate codewords is small compared with the computational complexity required to process the minimum-weight trellis. The computational complexity of the decoding algorithm can be analyzed easily from the flowchart shown in Figure 14.5.

Let N_s denote the fixed number of computations required in sorting the components of the received sequence $\mathbf{r}$ in increasing order of reliability, and let N_{loop} denote the number of computations required in

1. computing the index sets $D_1(\mathbf{v})$ and $D_0(\mathbf{v})$;

2. computing the correlation discrepancy $\lambda(\mathbb{r}, \mathbb{v})$, the optimality test threshold $G(\mathbb{v}, \omega_1)$, and the nearest-neighbor test threshold $G(\mathbb{v}, \omega_2)$; and
3. comparing $\lambda(\mathbb{r}, \mathbb{v})$ with $G(\mathbb{v}, \omega_1)$, $G(\mathbb{v}, \omega_2)$, and $\lambda(\mathbb{r}, \mathbb{v}_{best})$.

Let $N(T_{min})$ denote the number of computations required to process the minimum-weight trellis $T_{min}(\mathbb{v})$ (or $T_{min}(\mathbb{v}_{best})$). The number of decoding iterations required to decode a received sequence depends on SNR. The maximum number of decoding iterations is limited by $2^{\lfloor d_{min}/2 \rfloor}$. Therefore, the *worst-case maximum number* of computations required to decode a received sequence with the minimum-weight trellis-based iterative algorithm is

$$N_{max} = N_s + 2^{\lfloor d_{min}/2 \rfloor} \times N_{loop} + N(T_{min}). \tag{14.28}$$

Again, consider the (64, 42) RM code with $d_{min} = 8$. The uniform eight-section full trellis for this code has state complexity profile (1, 128, 1024, 8192, 1024, 8192, 1024, 128, 1). The state complexity profile for the eight-section minimum-weight trellis is (1, 45, 157, 717, 157, 717, 157, 45, 1). To process this eight-section minimum-weight trellis requires 8111 computations. The maximum number of iterations required to decode a received sequence of 64 symbols with the iterative decoding algorithm is 16. From (14.28) we find that $N_{max} = 16,495$. In fact, the average number of computations required can be much smaller than this worst-case computational complexity, as shown in Figure 14.7. For example, at SNR = 5 dB, the average number of computations needed is about 400, and the average number of

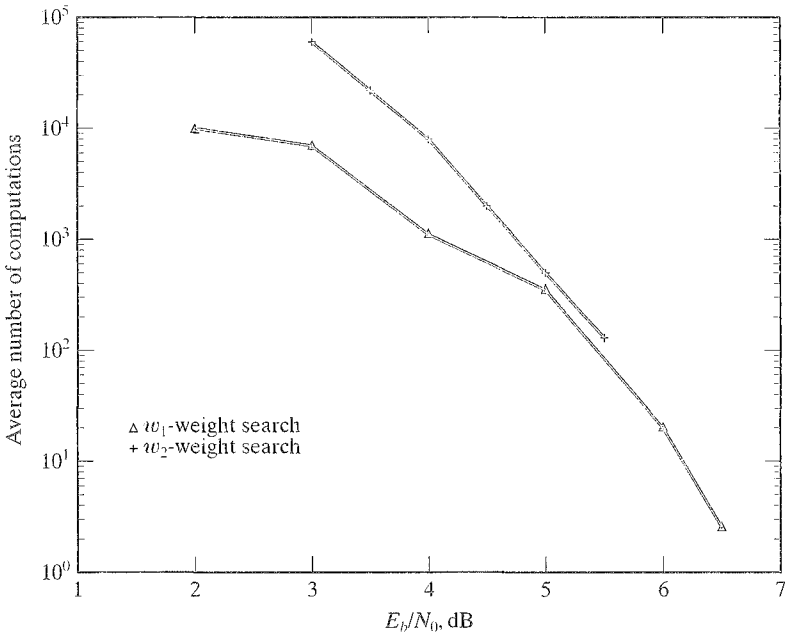

FIGURE 14.7: Average number of computations for the iterative decoding (w_1-, w_2-weight trellis search) of the (64, 42, 8) RM code.

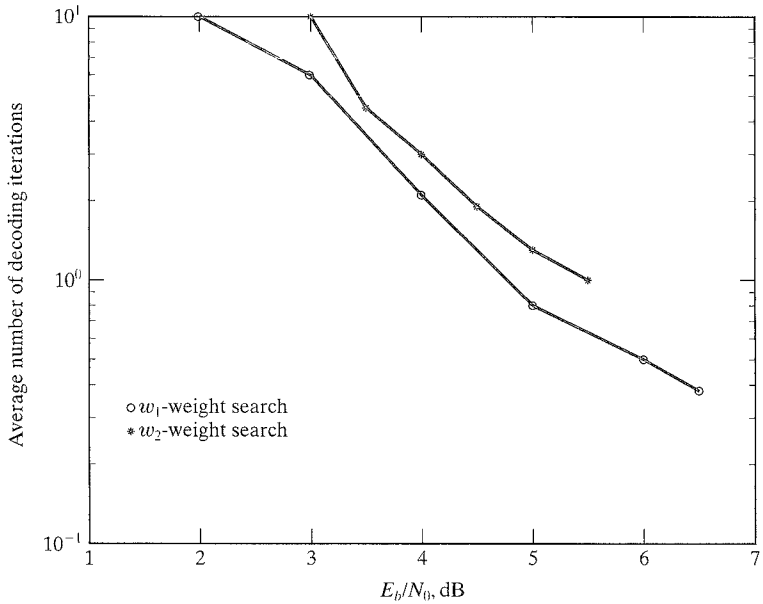

FIGURE 14.8: Average number of decoding iterations for the iterative decoding (w_1-, w_2-weight trellis search) of the (64, 42, 8) RM code.

iterations required to complete the decoding is 1, as shown in Figure 14.8; however, to decode this code with the Viterbi algorithm based on the uniform eight-section full trellis requires 554, 640 computations. With optimum sectionalization of the full-code trellis the number of computations required is 538, 799. We see that the iterative decoding algorithm results in a significant reduction in computational complexity, and the performance degradation is only 0.5 dB at BER of 10^{-6} compared with the optimum MLD.

14.3.5 Improvements

The error performance of the iterative decoding algorithm can be improved by using a larger search region. For example, we may use the region $R(\mathbf{v}, \omega_3)$, which consists of all the codewords at distances ω_1 (minimum distance) and ω_2 (next to the minimum distance) from the current tested candidate codeword for searching for the most likely codeword $\mathbf{v}_{ML}$. In the case that

$$G(\mathbf{v}, \omega_2) < \lambda(\mathbf{r}, \mathbf{v}) \leq G(\mathbf{v}, \omega_3),$$

the decoder searches the purged trellis $T_{\omega_2}(\mathbf{v})$ centered on $\mathbf{v}$ for finding $\mathbf{v}_{ML}$, where $T_{\omega_2}(\mathbf{v})$ consists of $\mathbf{v}$ and all the paths of the overall trellis of the code that are at distances ω_1 and ω_2 from $\mathbf{v}$. We call this search a *ω_2-weight trellis search*. It is clear that the improvement in error performance with a larger search region is achieved at the expense of additional computational complexity.

The error performance of the (64, 42) RM code and the computational complexity with a w_2-weight trellis search are shown in Figures 14.6, 14.7, and 14.8,

respectively. The w_2-weight trellis search achieves a 0.25-dB coding gain over the w_1-weight trellis search with some additional computations.

In the foregoing iterative decoding algorithm an algebraic decoder is used to generate candidate codewords. The algebraic decoder may fail to decode the modified received sequence and hence results in a decoding failure. This decoding failure can be prevented by using the first-order decoding based on the ordered statistics presented in Section 10.8.3 to generate candidate codewords. This decoding is very simple, and it never fails to decode. Furthermore, this decoding results in better candidate codewords (i.e., smaller correlation discrepancies) than the algebraic decoding in many cases. As a result, the decoding iteration process converges at a faster rate. With first-order statistic decoding, the maximum number of candidate codewords to be generated is k for an (n, k) binary linear code.

Another possible improvement is to use multiple candidate codewords, the current one and several previously tested ones, to determine the optimality and the search region. One such improvement can be found in [19], which provides significant improvement in error performance for long codes. Of course, this improvement is achieved at the expense of additional computational complexity.

14.4 THE MAP DECODING ALGORITHM

Maximum likelihood decoding minimizes the block (or sequence) error probability; however, it does not necessarily minimize the bit (or symbol) error probability. MLD delivers only hard-decoded bits (called *hard outputs*) without providing their reliability values (or measures). In many error-control coding schemes it is desirable to provide both decoded bits and their reliability values (called *soft outputs*) for further decoding processing to improve the system's error performance and to reduce the information loss. For example, in iterative decoding (see Chapters 16 and 17), the soft outputs of one decoder can be used as the soft inputs to another decoder, and the two decoders process the received sequence iteratively until the performance limit of the code is achieved.

A decoding algorithm that processes soft-decision inputs and produces soft-decision outputs is called a soft-in/soft-out (SISO) decoding algorithm. The most well known SISO decoding algorithm is the MAP (maximum a posteriori probability) decoding algorithm that was devised by Bahl, Cocke, Jelinek, and Raviv in 1974 [28]. This algorithm, called the BCJR algorithm, is devised to minimize the bit-error probability and to provide reliability values of the decoded bits. The MAP algorithm (or its suboptimum version) is the heart of turbo or iterative decoding [29, 30].

14.4.1 The MAP Algorithm Based on a Bit-Level Trellis

Again, assume BPSK transmission. Let $\mathbf{v} = (v_0, v_1, \cdots, v_{n-1})$ be a codeword and $\mathbf{c} = (c_0, c_1, \cdots, c_{n-1})$ be its corresponding bipolar signal sequence, where for $0 \le i < n$, $c_i = 2v_i - 1 = \pm 1$. Let $\mathbf{r} = (r_0, r_1, \cdots, r_{n-1})$ be the soft-decision received sequence. In decoding, the MAP algorithm estimates each transmitted code bit v_i by computing its log-likelihood ratio (LLR), which is defined as

$$L(v_i) \triangleq \log \frac{p(v_i = 1|\mathbf{r})}{p(v_i = 0|\mathbf{r})}, \tag{14.29}$$

where $p(v_i|\mathbb{r})$ is the a posteriori probability of v_i given the received sequence $\mathbb{r}$. The estimated code bit v_i is then given by the sign of its LLR as follows:

$$v_i = \begin{cases} 1, & \text{if } L(v_i) > 0, \\ 0, & \text{if } L(v_i) \leq 0. \end{cases} \tag{14.30}$$

From (14.29) we see that $|L(v_i)|$ is a measure of the reliability of the estimated code bit v_i: the larger the $|L(v_i)|$, the more reliable the hard decision of (14.30). Therefore, $L(v_i)$ represents the soft information associated with the decision on v_i.

Computation of $L(v_i)$ is facilitated by using the trellis for the code C to be decoded. In the n-section bit-level trellis T for C, let $B_i(C)$ denote the set of all branches (s_i, s_{i+1}) that connect the states in state space $\Sigma_i(C)$ at time i and the states in state space $\Sigma_{i+1}(C)$ at time-$(i+1)$. Each branch $(s_i, s_{i+1}) \in B_i(C)$ represents a single code bit v_i. Let $B_i^0(C)$ and $B_i^1(C)$ denote the two disjoint subsets of $B_i(C)$ that correspond to code bits $v_i = 0$ and $v_i = 1$, respectively. Clearly,

$$B_i(C) = B_i^0(C) \cup B_i^1(C)$$

for $0 \leq i < n$. Also, based on the structure of linear codes, $|B_i^0(C)| = |B_i^1(C)|$. For $(s', s) \in B_i(C)$, we define the joint probabilities

$$\lambda_i(s', s) \triangleq p(s_i = s', s_{i+1} = s, \mathbb{r}) \tag{14.31}$$

for $0 \leq i < n$. Then, the joint probabilities $p(v_i, \mathbb{r})$ for $v_i = 0$ and $v_i = 1$ are given by

$$p(v_i = 0, \mathbb{r}) = \sum_{(s',s) \in B_i^0(C)} \lambda_i(s', s), \tag{14.32}$$

$$p(v_i = 1, \mathbb{r}) = \sum_{(s',s) \in B_i^1(C)} \lambda_i(s', s). \tag{14.33}$$

The MAP algorithm computes the probabilities $\lambda_i(s', s)$'s, which are then used to evaluate $p(v_i = 0|\mathbb{r})$ and $p(v_i = 1|\mathbb{r})$ in (14.29) from (14.32) and (14.33). In fact, the LLR of v_i can be computed directly from the joint probabilities $p(v_i = 0, \mathbb{r})$ and $p(v_i = 1, \mathbb{r})$, as follows:

$$L(v_i) = \log \frac{p(v_i = 1, \mathbb{r})}{p(v_i = 0, \mathbb{r})}. \tag{14.34}$$

For $0 \leq j \leq l \leq n$, let $\mathbb{r}_{j,l}$ denote the following section of the received sequence $\mathbb{r}$:

$$\mathbb{r}_{j,l} \triangleq (r_j, r_{j+1}, \cdots, r_{l-1})$$

where for $l = j$, $\mathbb{r}_{j,j}$ denotes the null sequence. For any state $s \in \Sigma_i(C)$, we define the probabilities

$$\alpha_i(s) \triangleq p(s_i = s, \mathbb{r}_{0,i}), \tag{14.35}$$

$$\beta_i(s) \triangleq p(\mathbb{r}_{i,n}|s_i = s). \tag{14.36}$$

It follows from the definitions of $\alpha_i(s)$ and $\beta_i(s)$ that

$$\alpha_0(s_0) = \beta_n(s_f) = 1. \tag{14.37}$$

For any two adjacent states s' and s with $s' \in \Sigma_i(C)$ and $s \in \Sigma_{i+1}(C)$, we define the probability

$$\begin{aligned} \gamma_i(s', s) &\triangleq p(s_{i+1} = s, r_i | s_i = s') \\ &= p(s_{i+1} = s | s_i = s') p(r_i | (s_i, s_{i+1}) = (s', s)). \end{aligned} \tag{14.38}$$

For a memoryless channel, it follows from the definitions of $\lambda_i(s', s)$, $\alpha_i(s)$, $\beta_i(s)$, and $\gamma_i(s', s)$ that for $0 \leq i < n$,

$$\lambda_i(s', s) = \alpha_i(s') \gamma_i(s', s) \beta_{i+1}(s). \tag{14.39}$$

Then, it follows from (14.32), (14.33), and (14.39) that we can express (14.34) as follows:

$$L(v_i) = \log \frac{\sum_{(s', s) \in B_i^1(C)} \alpha_i(s') \gamma_i(s', s) \beta_{i+1}(s)}{\sum_{(s', s) \in B_i^0(C)} \alpha_i(s') \gamma_i(s', s) \beta_{i+1}(s)}. \tag{14.40}$$

For a state $s \in \Sigma_i(C)$, let $\Omega_{i-1}^{(c)}(s)$ and $\Omega_{i+1}^{(d)}(s)$ denote the sets of states in $\Sigma_{i-1}(C)$ and in $\Sigma_{i+1}(C)$, respectively, that are adjacent to s, as shown in Figure 14.9. Then $\alpha_i(s)$ and $\beta_i(s)$ can be expressed as follows:

1. For $0 \leq i \leq n$,

$$\begin{aligned} \alpha_i(s) &= \sum_{s' \in \Omega_{i-1}^{(c)}(s)} p(s_{i-1} = s', s_i = s, r_{i-1}, \mathbb{r}_{0,i-1}) \\ &= \sum_{s' \in \Omega_{i-1}^{(c)}(s)} \alpha_{i-1}(s') \gamma_{i-1}(s', s); \end{aligned} \tag{14.41}$$

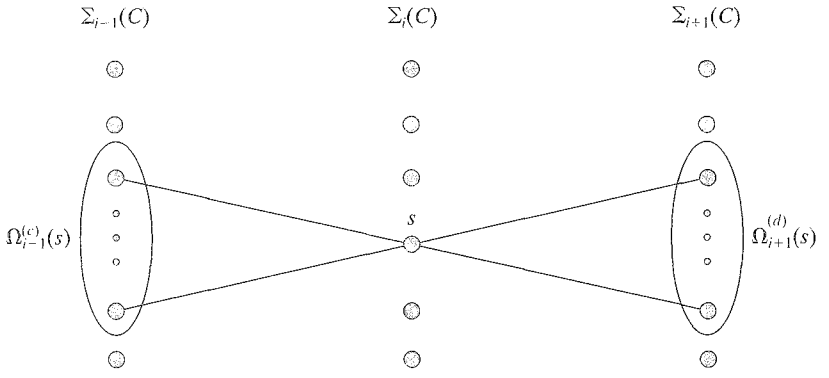

FIGURE 14.9: States connected to state $s \in \Sigma_i(C)$.

2. For $0 \le i \le n$,

$$\begin{aligned}\beta_i(s) &= \sum_{s' \in \Omega_{i+1}^{(d)}(s)} p(r_i, \mathbb{r}_{i+1,n}, s_{i+1} = s' | s_i = s) \\ &= \sum_{s' \in \Omega_{i+1}^{(d)}(s)} \gamma_i(s, s') \beta_{i+1}(s').\end{aligned} \tag{14.42}$$

From (14.41) we see that probabilities $\alpha_i(s)$'s, $0 < i \le n$, can be computed recursively from the initial state s_0 to the final state s_f of the n-section bit-level trellis T with initial condition $\alpha_0(s_0) = 1$ once the state transition (or branch) probabilities $\gamma_i(s', s)$'s are computed. This computation process is called the *forward recursion*. The probabilities $\beta_i(s)$'s, $0 \le i < n$, can be computed from (14.42) recursively from the final state s_f to the initial state s_0 of T with initial condition $\beta_n(s) = 1$. This computation process is called the *backward recursion*.

The state transition probability $\gamma_i(s', s)$ depends on the probability distribution of the information bits and the channel. Assume that each information bit is equally likely to be 0 or 1. Then, all the states in $\Sigma_i(C)$ are equiprobable, and the transition probability

$$p(s_{i+1} = s | s_i = s') = \frac{1}{|\Omega_{i+1}^{(d)}(s')|}.$$

For an AWGN channel with zero mean and two-sided power spectral density $N_0/2$, the conditional probability $p(r_i | (s_i, s_{i+1}) = (s', s))$ is given by

$$p(r_i | (s_i, s_{i+1}) = (s', s)) = \frac{1}{\sqrt{\pi N_0}} \exp\{-(r_i - c_i)^2 / N_0\}, \tag{14.43}$$

where $c_i = 2v_i - 1$, and v_i is the code bit on the branch (s', s). Because in (14.29) (or (14.34)) we are interested only in the ratio of $p(v_i = 1 | \mathbb{r})$ to $p(v_i = 0 | \mathbb{r})$ (or $p(v_i = 1, \mathbb{r})$ to $p(v_i = 0, \mathbb{r})$), it follows from (14.32) and (14.33) that we can scale $\lambda_i(s', s)$ by any factor without changing the LLR of each estimated code bit. Consequently, we can use

$$w_i(s', s) \triangleq \exp\{-(r_i - c_i)^2 / N_0\} \tag{14.44}$$

to replace $\gamma_i(s', s)$ in computing $\alpha_i(s)$, $\beta_i(s)$, $\lambda_i(s', s)$, $p(v_i = 1, \mathbb{r})$, and $p(v_i = 0, \mathbb{r})$. We call $w_i(s', s)$ the *weight* of the branch (s', s).

To carry out the forward recursion set $\alpha_0(s_0) = 1$ to initiate the recursion process.

1. Assume that the probabilities $\alpha_i(s')$'s have been computed for all states s' in $\Sigma_i(C)$.
2. In the trellis section from time-i to time-$(i + 1)$, assign the weight $w_i(s', s)$ to each branch $(s', s) \in B_i(C)$.
3. For each state $s \in \Sigma_{i+1}(C)$ compute and store

$$\alpha_{i+1}(s) = \sum_{s' \in \Omega_i^{(c)}(s)} \alpha_i(s') w_i(s', s). \tag{14.45}$$

4. Repeat the process until $\alpha_n(s_f)$ is computed.

To carry out the backward recursion from the state s_f to the initial state s_0 as follows, set $\beta_n(s_f) = 1$ to initiate the recursion.

1. Assume that the probabilities $\beta_{i+1}(s')$ for all states $s' \in \Sigma_{i+1}(C)$ have been computed.
2. In the trellis section from time-i to time-$(i+1)$, assign the weight $w_i(s, s')$ to each branch $(s, s') \in B_i(C)$.
3. For each state $s \in \Sigma_i(C)$ compute and store

$$\beta_i(s) = \sum_{s' \in \Omega_{i+1}^{(d)}(s)} w_i(s, s')\beta_{i+1}(s'). \tag{14.46}$$

4. Repeat the process until $\beta_0(s_0)$ is computed.

Then, to carry out the MAP decoding algorithm, use the following three steps:

1. Perform the forward recursion process to compute the forward state probabilities, $\alpha_i(s)$'s, for $0 \le i \le n$.
2. Perform the backward recursion process to compute the backward state probabilities, $\beta_i(s)$'s, for $0 \le i \le n$.
3. Decoding is also performed during the backward recursion. As soon as the probabilities $\beta_{i+1}(s)$'s for all states $s \in \Sigma_{i+1}(C)$ have been computed, evaluate the probabilities $\lambda_i(s', s)$'s for all branches $(s', s) \in B_i(C)$. From (14.32) and (14.33) compute the joint probabilities $p(v_i = 0, \mathbb{r})$ and $p(v_i = 1, \mathbb{r})$. Then, compute the LLR of v_i from (14.34) and decode v_i based on the decision rule given by (14.30).

14.4.2 Bidirectional and Parallel MAP Decoding

Because the forward and backward recursions are independent of each other, they can be executed simultaneously from both directions of the code trellis. This bidirectional process reduces the decoding delay. To achieve bidirectional decoding, we permute the codeword $\mathbb{v} = (v_0, v_1, \cdots, v_{n-1})$ to $(v_0, v_{n-1}, v_1, v_{n-2}, \cdots)$ before its transmission. Let $\mathbb{r} = (r_0, r_{n-1}, r_1, r_{n-2}, \cdots)$ be its corresponding received sequence. Before decoding, we permute $\mathbb{r}$ to obtain two received sequences, $\mathbb{r}^{(1)} = (r_0, r_1, \cdots, r_{n-1})$ and $\mathbb{r}^{(2)} = (r_{n-1}, r_{n-2}, \cdots, r_0)$, which are called *forward* and *backward received sequences*, respectively. Then, we shift $\mathbb{r}^{(1)}$ and $\mathbb{r}^{(2)}$ into the code trellis from the initial state s_0 and the final state s_f, respectively, to perform forward and backward recursions simultaneously for computing α and β probabilities in opposite directions [33, 34]. When the recursions meet at the middle of the trellis, the decoding begins. As soon as both $\alpha_i(s')$ and $\beta_{i+1}(s)$, with $s' \in \Sigma_i(C)$ and $s \in \Sigma_{i+1}(C)$, have been computed, the LLR of the estimated code bit v_i is evaluated from (14.39), (14.32), (14.33), and (14.34).

Even though the same number of computations is required, the bidirectional process roughly reduces the decoding time by half. If the code trellis has mirror-image symmetry, we can devise two identical circuits to compute α's and β's.

If the code trellis T for a code is too large for practical IC implementation, we can decompose the trellis into parallel and structurally identical subtrellises without

exceeding the maximum state complexity of T. Each of these subtrellises has much smaller state and branch complexities than the full-code trellis T. This parallel decomposition allows us to devise identical smaller MAP decoders to process the subtrellises in parallel independently without communications between them, which simplifies the IC implementation and speeds up the decoding process.

In Chapter 9 it was shown that it is possible to decompose a trellis T into parallel and structurally identical subtrellises without exceeding the maximum state complexity of T. Suppose T can be decomposed into μ subtrellises, denoted by $T^{(1)}, T^{(2)}, \cdots, T^{(\mu)}$. Based on each of these subtrellises $T^{(j)}$, we compute

$$Q^{(j)}(1) = \sum_{(s',s)\in B_i^1(C)_j} \alpha_i^{(j)}(s')\gamma_i^{(j)}(s', s)\beta_{i+1}^{(j)}(s),$$

and

$$Q^{(j)}(0) = \sum_{(s',s)\in B_i^0(C)_j} \alpha_i^{(j)}(s')\gamma_i^{(j)}(s', s)\beta_{i+1}^{(j)}(s),$$

where $B_i^0(C)_j$ and $B_i^1(C)_j$ denote the sets of branches labeled with 0 and 1, respectively, in subtrellis $T^{(j)}$. Then, the LLR $L(v_i)$ of v_i is given by [33, 34]

$$L(v_i) = \log \frac{\sum_{j=1}^{\mu} Q^{(j)}(1)}{\sum_{j=1}^{\mu} Q^{(j)}(0)} \tag{14.47}$$

for $0 \le i < n$.

From the standpoint of decoding speed, the effective computational complexity of decoding a received sequence is defined as the computational complexity of a single parallel subtrellis plus the number of computations required to compute the LLR of each estimated code bit from (14.47). The time required to compute $L(v_i)$ from (14.47) is relatively small compared with the time required for processing a subtrellis. Because all the subtrellises are processed in parallel, the speed of decoding is therefore limited only by the time required to process one subtrellis.

14.4.3 Computational Complexity

The computational complexity of the MAP algorithm can be analyzed by enumerating the numbers of real operations required to compute $\gamma_i, \alpha_i, \beta_i, \lambda_i, p(v_i, \mathbb{r})$, and $L(v_i)$ for each trellis section. We assume that $\exp(\cdot)$ and $\log(\cdot)$ are computed by using a read-only memory (ROM) (i.e., table lookup).

Because $w_i(s', s)$ is used to replace $\gamma_i(s', s)$ in all computations, this value can be read out from ROM for each received symbol r_i. Consider the trellis section T_i between time-i and time-$(i+1)$ for $0 \le i < n$. For any state $s \in \Sigma_{i+1}(C)$, $|\Omega_i^{(c)}(s)|$ is simply the number of branches entering s (or incoming degree), which is the same for all states in $\Sigma_{i+1}(C)$. The term $|\Omega_i^{(c)}(s)|$ is either 1 or 2 depending on whether there is an oldest information bit a^0 to be shifted out from the encoder memory. Then, the total number of branches in T_i is

$$|B_i(C)| = 2^{\rho_{i+1}}|\Omega_i^{(c)}(s)|, \tag{14.48}$$

where ρ_{i+1} is the dimension of the state space $\Sigma_{i+1}(C)$ at time-$(i+1)$. For any state $s \in \Sigma_i(C)$, $|\Omega_{i+1}^{(d)}(s)|$ is simply the number of branches leaving s (outgoing degree), which is the same for all states in $\Sigma_i(C)$. The term $|\Omega_{i+1}^{(d)}(s)|$ is either 1 or 2 depending on whether there is a current input information bit a^* at time-i. It is clear that the total number of branches in T_i is

$$|B_i(C)| = 2^{\rho_i}|\Omega_{i+1}^{(d)}|, \tag{14.49}$$

where ρ_i is the dimension of the state space $\Sigma_i(C)$ at time-i.

It follows from (14.45) and (14.48) that to compute $\alpha_{i+1}(s)$ for all states in $\Sigma_{i+1}(C)$ requires $|B_i(C)|$ multiplications and $|B_i(C)| - 2^{\rho_{i+1}}$ additions. Similarly, it follows from (14.46) and (14.49) that to compute $\beta_i(s)$ for all states in $\Sigma_i(C)$ requires $|B_i(C)|$ multiplications and $|B_i(C)| - 2^{\rho_i}$ additions. It follows from (14.32), (14.33), and (14.39) that computation of $p(v_i = 0, \mathbb{r})$ and $p(v_i = 1, \mathbb{r})$ requires $2|B_i(C)|$ multiplications and $|B_i(C)| - 2$ additions. Taking the ratio $p(v_i = 1, \mathbb{r})/p(v_i = 0, \mathbb{r})$ requires one division. A division operation is equivalent to a multiplication operation. Based on the preceding analysis, we find that MAP decoding of a received sequence $\mathbb{r}$ requires

$$N_m = 4\mathcal{E} + n \tag{14.50}$$

multiplications and

$$N_a = 3\mathcal{E} - 2V - 2(n-1) \tag{14.51}$$

additions, where

$$\mathcal{E} = \sum_{i=0}^{n-1} |B_i(C)|, \tag{14.52}$$

and

$$V = \sum_{i=0}^{n} 2^{\rho_i}. \tag{14.53}$$

Note that $\mathcal{E}$ and V are simply the total number of branches and states in the trellis T, respectively. From (14.50) and (14.51) we see that the major factor in the computational complexity of the MAP algorithm is the number of multiplications.

Because for each trellis section, $\gamma_i(s', s)$ for the distinct branch, $\alpha_i(s)$ or $\beta_i(s)$ for all $(s', s) \in B_i$, and LLRs for all code bits must be stored, the MAP decoder requires a memory to store

$$M = 2n + (V-1) + n = 3n + V - 1$$

real numbers.

As an example, consider the (32, 16) RM code with a minimum distance of 8. The bit-error performances of this code based on MAP and Viterbi decodings are shown in Figure 14.10. We see that the MAP algorithm performs just slightly better than the Viterbi algorithm. The 32-section bit-level trellis for this code has a total of 4798 states and a total of 6396 edges. MAP decoding of this code requires 25,616 multiplications, 9530 additions and 4893 storage units.

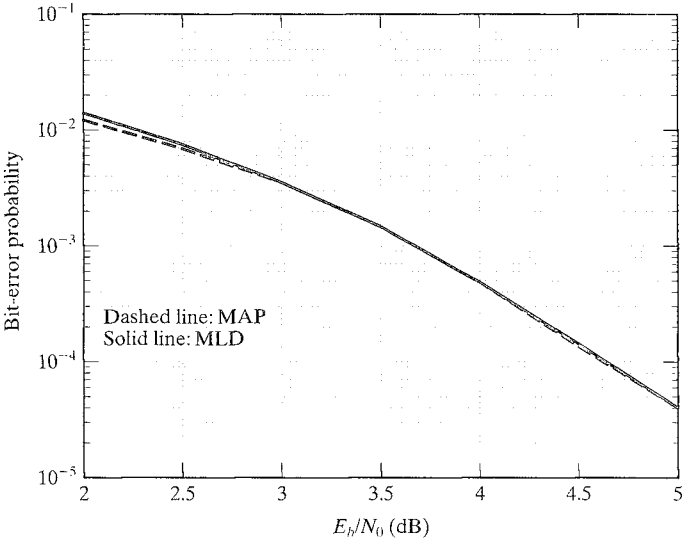

FIGURE 14.10: Bit-error performances of the (32, 16) RM code with MAP and Viterbi decoding algorithms.

14.5 MAP DECODING BASED ON A SECTIONALIZED TRELLIS

The MAP decoding algorithm can be carried out based on a sectionalized trellis [33, 34]. Proper sectionalization of a code trellis may result in a significant reduction in computational complexity and storage requirements.

14.5.1 The Algorithm

Let $T(\Lambda)$ denote a v-section trellis for an (n, k) linear code C with section boundary locations in $\Lambda = \{t_0, t_1, \cdots, t_v\}$, where $0 = t_0 < t_1 < \cdots < t_v = n$. For two adjacent states $s' \in \Sigma_{t_i}(C)$ and $s \in \Sigma_{t_{i+1}}(C)$, let $L(s', s)$ denote the set of parallel branches connecting state s' to state s, called a composite branch. Each branch $\mathbb{b}(s', s) \in L(s', s)$ consists of $t_{i+1} - t_i$ code bits,

$$(v_{t_i}, v_{t_i+1}, \cdots, v_{t_{i+1}-1}).$$

For each branch $\mathbb{b}(s', s)$ in $L(s', s)$, we define the following probability:

$$\begin{aligned}\gamma_{t_i}(\mathbb{b}(s', s)) &\triangleq p(s_{t_{i+1}} = s, \mathbb{b}(s', s), \mathbb{r}_{t_i, t_{i+1}} | s_{t_i} = s') \\ &= p(s_{t_{i+1}} = s, \mathbb{b}(s', s) | s_{t_i} = s') \cdot p(\mathbb{r}_{t_i, t_{i+1}} | (s_{t_i}, s_{t_{i+1}}) = (s', s), \mathbb{b}(s', s)),\end{aligned} \tag{14.54}$$

which is called a *branch probability* connecting state s' to state s. Then, the probability of the composite branch $L(s', s)$ connecting state s' to state s is given by

$$\gamma_{t_i}(L(s', s)) \triangleq \sum_{\mathbb{b}(s', s) \in L(s', s)} \gamma_{t_i}(\mathbb{b}(s', s)). \tag{14.55}$$

For the MAP algorithm based on the sectionalized trellis $T(\Lambda)$, the forward and backward recursions are to compute the following probabilities:

$$\alpha_{t_i}(s) = \sum_{s' \in \Omega^{(c)}_{t_{i-1}}(s)} \alpha_{t_{i-1}}(s')\gamma_{t_{i-1}}(L(s', s)), \tag{14.56}$$

$$\beta_{t_i}(s) = \sum_{s' \in \Omega^{(d)}_{t_{i+1}}(s)} \gamma_{t_i}(L(s, s'))\beta_{t_{i+1}}(s'), \tag{14.57}$$

where s is a state in $\Sigma_{t_i}(C)$, $\Omega^{(c)}_{t_{i-1}}(s)$ denotes the set of states in $\Sigma_{t_{i-1}}(C)$ that are adjacent to s, and $\Omega^{(d)}_{t_{i+1}}(s)$ denotes the set of states in $\Sigma_{t_{i+1}}(C)$ that are adjacent to s. The initial conditions for the recursions are $\alpha_0(s_0) = 1$ and $\beta_n(s_f) = 1$.

For computing the LLR of an estimated code bit v_l with $t_i \leq l < t_{i+1}$, we define

$$\gamma_{t_i}(L^{(0)}_{v_l}(s', s)) \triangleq \sum_{\substack{\mathbb{b}(s', s) \in L(s', s) \\ v_l = 0}} \gamma_{t_i}(\mathbb{b}(s', s)), \tag{14.58}$$

$$\gamma_{t_i}(L^{(1)}_{v_l}(s', s)) \triangleq \sum_{\substack{\mathbb{b}(s', s) \in L(s', s) \\ v_l = 1}} \gamma_{t_i}(\mathbb{b}(s', s)). \tag{14.59}$$

It follows from (14.55), (14.58), and (14.59) that

$$\gamma_{t_i}(L(s', s)) = \gamma_{t_i}(L^{(0)}_{v_l}(s', s)) + \gamma_{t_i}(L^{(1)}_{v_l}(s', s)) \tag{14.60}$$

for every code bit v_l with $t_i \leq l < t_{i+1}$. Then, the LLR of v_l is given by

$$L(v_l) = \log \frac{\sum_{\substack{(s', s) \\ v_l = 1}} \alpha_{t_i}(s')\gamma_{t_i}(L^{(1)}_{v_l}(s', s))\beta_{t_{i+1}}(s)}{\sum_{\substack{(s', s) \\ v_l = 0}} \alpha_{t_i}(s')\gamma_{t_i}(L^{(0)}_{v_l}(s', s))\beta_{t_{i+1}}(s)}, \tag{14.61}$$

where the summations are over all the adjacent state pairs (s', s) with $s' \in \Sigma_{t_i}(C)$ and $s \in \Sigma_{t_{i+1}}(C)$.

To carry out the decoding process we must first compute the composite branch probabilities, $\gamma_{t_i}(L(s', s))$'s. In a sectionalized trellis a section may consist of many composite branches; however, the number of distinct composite branches is relatively small. In computing $\alpha_{t_i}(s)$, $\beta_{t_i}(s)$, and $L(v_l)$, we need the probabilities of only the distinct composite branches. Thus, we may perform a preprocessing step to compute $\gamma_{t_i}(L(s', s))$, $\gamma_{t_i}(L^{(0)}_{v_l}(s', s))$, and $\gamma_{t_i}(L^{(1)}_{v_l}(s', s))$ for each distinct composite branch $L(s', s)$ and each code bit v_l and store them in a table called the γ_{t_i}-table. From (14.56) we see that $\alpha_{t_i}(s)$'s can be computed along with $\gamma_{t_i}(L(s', s))$'s.

The MAP algorithm based on a sectionalized trellis $T(\Lambda)$ can be carried out in two stages. At the first stage, the parallel branches of each distinct composite branch are preprocessed to obtain the probabilities $\gamma_{t_i}(L(s', s))$, $\gamma_{t_i}(L^{(0)}_{v_l}(s', s))$, and

$\gamma_{t_i}(L_{v_l}^{(1)}(s', s))$. Then, the MAP decoding is performed with parallel branches viewed as a single branch.

Bidirectional and parallel MAP decoding based on a sectionalized trellis can be performed in exactly the same manner as that based on the bit-level trellis T. In bidirectional decoding, $\alpha_{t_i}(s)$'s and $\beta_{t_i}(s)$'s can be computed simultaneously along with $\gamma_{t_i}(L(s', s))$'s.

Consider the computation of a branch probability $\gamma_{t_i}(\mathbb{b}(s', s))$ given by (14.54). If all the states at any section boundary location are equiprobable, then

$$p(s_{t_{i+1}} = s, \mathbb{b}(s', s) | s_{t_i} = s') = \frac{1}{\sum_{s'' \in \Omega_{t_{i+1}}^{(d)}(s')} |L(s', s'')|}.$$

It has been proved that $|\Omega_{t_{i+1}}^{(d)}(s')|$ and $|L(s', s'')|$ remain the same for all states $s' \in \Sigma_{t_i}(C)$. Therefore, $p(s_{t_{i+1}} = s, \mathbb{b}(s', s) | s_{t_i} = s')$ is constant in the trellis section T_i from time-t_i to time-t_{i+1}. Because we are interested only in the ratio given by (14.61), we can scale the branch probability $\gamma_{t_i}(\mathbb{b}(s', s))$ by any factor without changing the LLR of an estimated code bit. Therefore, in computing α, β, and $L(v_l)$, we can use

$$p(\mathbb{r}_{t_i, t_{i+1}} | (s_{t_i}, s_{t_{i+1}}) = (s', s), \mathbb{b}(s', s)) \tag{14.62}$$

to replace $\gamma_{t_i}(\mathbb{b}(s', s))$ to simplify computations. For an AWGN channel with zero mean and variance $N_0/2$,

$$\begin{aligned} p(\mathbb{r}_{t_i, t_{i+1}} | (s_{t_i}, s_{t_{i+1}}) &= (s', s), \mathbb{b}(s', s)) \\ &= \frac{1}{(\pi N_0)^{m_i/2}} \exp\left\{ - \sum_{j=0}^{m_i - 1} (r_{t_i + j} - (2v_{t_i + j} - 1))^2 / N_0 \right\}, \end{aligned} \tag{14.63}$$

where $m_i = t_{i+1} - t_i$. Because $1/(\pi N_o)^{m_i/2}$ is constant in T_i, we can use

$$w_{t_i}(\mathbb{b}(s', s)) = \exp\left\{ \frac{2}{N_0} \sum_{j=0}^{m_i - 1} r_{t_i + j} \cdot v_{t_i + j} \right\} \tag{14.64}$$

to replace $\gamma_{t_i}(\mathbb{b}(s', s))$ in computing $L(v_l)$ of (14.61).

To construct the γ_{t_i}-table for the trellis section T_i from time-t_i to time-t_{i+1} with $0 \leq i < \nu$, we can use the following procedure-γ to save computations:

1. Compute $w_{t_i}(\mathbb{b}(s', s))$ using (14.64) for all distinct branches in T_i.
2. Compute $\gamma_{t_i}(L_{v_l}^{(0)}(s', s))$ and $\gamma_{t_i}(L_{v_l}^{(1)}(s', s))$ from (14.58) and (14.59) with $\gamma_{t_i}(\mathbb{b}(s', s))$ replaced by $w_{t_i}(\mathbb{b}(s', s))$ for each code bit v_l with $t_i \leq l < t_{i+1}$.
3. Compute $\gamma_{t_i}(L(s', s))$ for all distinct composite branches from (14.60).

From (14.56) we see that $\alpha_{t_i}(s)$'s can be computed along with the construction of γ-tables from the initial state s_0 to the final state s_f of the code trellis in the

forward direction. If bidirectional decoding is performed, it follows from (14.57) that $\beta_{t_i}(s)$'s can be computed along with the construction of γ-tables from the final state s_f to the initial state s_0 of the code trellis in the backward direction. As soon as $\alpha_{t_i}(s)$'s and $\beta_{t_{i+1}}(s)$'s at the boundaries of the trellis section T_i have been computed, the LLRs of the code bits v_l with $t_i \leq l < t_{i+1}$ are evaluated from (14.61).

14.5.2 Computational Complexity and Storage Requirement

To analyze the computational complexity of the MAP algorithm based on a sectionalized code trellis $T(\Lambda)$, we need to use some trellis structural properties developed in Chapter 9. Consider the section T_i of $T(\Lambda)$ from time-t_i to time-t_{i+1}. Every composite branch $L(s', s)$ in this section is a coset in the partition $p_{t_i,t_{i+1}}(C)/C^{tr}_{t_i,t_{i+1}}$ and consists of

$$B_i^p = 2^{k(C_{t_i,t_{i+1}})} \tag{14.65}$$

parallel branches. There are

$$B_i^d = 2^{k(p_{t_i,t_{i+1}}(C))-k(C_{t_i,t_{i+1}})} \tag{14.66}$$

distinct composite branches. The total number of composite branches in T_i is

$$B_i^c = 2^{k(C)-k(C_{0,t_i})-k(C_{t_{i+1},n})-k(C_{t_i,t_{i+1}})}. \tag{14.67}$$

Therefore, the total number of branches in T_i is

$$B_i = B_i^c \cdot B_i^p. \tag{14.68}$$

The number of composite branches entering a state $s \in \Sigma_{t_i}(C)$ (called the incoming degree of s) is given by

$$\deg(s)_{in} = 2^{k(C_{0,t_i})-k(C_{0,t_{i-1}})-k(C_{t_{i-1},t_i})}. \tag{14.69}$$

Then, it follows from the definitions of $\Omega^{(c)}_{t_{i-1}}(s)$ and $\deg(s)_{in}$ that

$$|\Omega^{(c)}_{t_{i-1}}(s)| = \deg(s)_{in}. \tag{14.70}$$

The number of composite branches leaving a state $s \in \Sigma_{t_i}(C)$ (called the outgoing degree of s) is given by

$$\deg(s)_{out} = 2^{k(C_{t_i,n})-k(C_{t_{i+1},n})-k(C_{t_i,t_{i+1}})}. \tag{14.71}$$

Then, it follows from the definitions of $\Omega^{(d)}_{t_{i+1}}(s)$ and $\deg(s)_{out}$ that

$$|\Omega^{(d)}_{t_{i+1}}(s)| = \deg(s)_{out}. \tag{14.72}$$

Again, we assume that $\exp(\cdot)$ and $\log(\cdot)$ are computed by using table lookup. To form the γ_{t_i}-table for the ith trellis section T_i based on the procedure-γ presented in the previous subsection, the computations required at each step are listed:

1. Step 1 requires $B_i^d \cdot B_i^p \cdot (m_i - 1)$ additions and m_i multiplications.
2. Steps 2 and 3 require $B_i^d \cdot (B_i^p - 2) \cdot m_i$ and B_i^d additions, respectively.

For the case of $B_i^p = 1$, since there is only one branch in each composite branch, we need to compute only $w_{t_i}(\mathbb{b}(s', s))$ for all distinct branches. Therefore, construction of the γ_{t_i}-table for trellis section T_i requires a total of

$$N_a^i(\gamma) = \begin{cases} B_i^d \cdot B_i^p \cdot (m_i - 1) + B_i^d \cdot (B_i^p - 2) \cdot m_i + B_i^d, & \text{for } B_i^p > 1, \\ B_i^d \cdot (m_i - 1), & \text{for } B_i^p = 1, \end{cases} \tag{14.73}$$

additions and a total of

$$N_m^i(\gamma) = m_i \tag{14.74}$$

multiplications.

Computation of α's and β's from (14.56) and (14.57) requires a total of $N_a^i(\alpha) + N_a^i(\beta)$ additions and $N_m^i(\alpha) + N_m^i(\beta)$ multiplications, where

$$N_a^i(\alpha) = (\deg(s_{t_i})_{in} - 1) \cdot 2^{\rho_{t_i}} = B_{i-1}^c - 2^{\rho_{t_i}}, \tag{14.75}$$

$$N_a^i(\beta) = (\deg(s_{t_i})_{out} - 1) \cdot 2^{\rho_{t_i}} = B_i^c - 2^{\rho_{t_i}}, \tag{14.76}$$

$$N_m^i(\alpha) = B_{i-1}^c, \tag{14.77}$$

$$N_m^i(\beta) = B_i^c. \tag{14.78}$$

The last step of MAP decoding is to compute the LLRs of the estimated code bits v_l for $t_i \le l < t_{i+1}$ with $0 \le i < \nu$. We define

$$S^{(i)} \triangleq \sum_{(s',s)} \alpha_{t_i}(s')\gamma_{t_i}(L(s', s))\beta_{t_{i+1}}(s), \tag{14.79}$$

$$S_0^{(i)}(v_l) \triangleq \sum_{\substack{(s',s) \\ v_l=0}} \alpha_{t_i}(s')\gamma_{t_i}(L_{v_l}^{(0)}(s', s))\beta_{t_{i+1}}(s), \tag{14.80}$$

$$S_1^{(i)}(v_l) \triangleq \sum_{\substack{(s',s) \\ v_l=1}} \alpha_{t_i}(s')\gamma_{t_i}(L_{v_l}^{(1)}(s', s))\beta_{t_{i+1}}(s). \tag{14.81}$$

It follows from (14.60) and (14.79) through (14.81) that

$$S^{(i)} = S_0^{(i)}(v_l) + S_1^{(i)}(v_l) \tag{14.82}$$

for any l, with $t_i \le l < t_{i+1}$. To compute the LLR $L(v_l)$'s from (14.61), we need to compute $S_0^{(i)}(v_l)$'s and $S_1^{(i)}(v_l)$'s. The $S_0^{(i)}(v_l)$'s and $S_1^{(i)}(v_l)$'s can be computed efficiently by using the following procedure-S:

1. Compute $S^{(i)}$ from (14.79), which requires $B_i^c - 1$ additions and $2B_i^c$ multiplications.
2. Compute $S_1^{(i)}(v_l)$'s from (14.81) for $t_i \le l < t_{i+1}$, which requires $(B_i^c - 1) \cdot m_i$ additions and $B_i^c \cdot m_i$ multiplications (using partial results from step 1).
3. Compute $S_0^{(i)}(v_l)$'s from (14.82) by taking the differences $S^{(i)} - S_1^{(i)}(v_l)$, for $t_i \le l < t_{i+1}$. This step requires m_i subtractions (equivalent to additions).

Once $S_0^{(i)}(v_l)$'s and $S_1^{(i)}(v_l)$'s have been computed, the LLRs of estimated code bits v_l for $t_i \le l < t_{i+1}$ can be evaluated, which requires m_i divisions (equivalent to multiplications). Therefore, computation of the LLRs of estimated code bits in the ith trellis section T_i requires a total of

$$N_a^i(L) = B_i^c - 1 + B_i^c \cdot m_i \tag{14.83}$$

additions and a total of

$$N_m^i(L) = 2B_i^c + (B_i^c + 1) \cdot m_i \tag{14.84}$$

multiplications (including m_i divisions).

In summary, execution of the MAP algorithm based on the sectionalized trellis $T(\Lambda)$ requires a total of

$$N_a(\Lambda) = \sum_{i=0}^{\nu-1} \{N_a^i(\gamma) + N_a^i(L) + N_a^i(\alpha) + N_a^i(\beta)\} \tag{14.85}$$

additions and a total of

$$N_m(\Lambda) = \sum_{i=0}^{\nu-1} \{N_m^i(\gamma) + N_m^i(L) + N_m^i(\alpha) + N_m^i(\beta)\} \tag{14.86}$$

multiplications. The numbers $N_a(\Lambda)$ and $N_m(\Lambda)$ together give a measure of computational complexity of the MAP algorithm based on the sectionalized trellis $T(\Lambda)$.

During the decoding process the γ-tables must be stored for the computation of α's, β's, and LLRs of the estimated code bits, which require

$$M(\gamma) = \sum_{i=0}^{\nu-1} B_i^d (2m_i + 1) \tag{14.87}$$

storage locations (or units). The α's and β's for the computation of LLRs of the estimated code bits also need to be stored. Thus, for bidirectional decoding,

$$M(\alpha, \beta) \triangleq \frac{M(\alpha) + M(\beta)}{2} \tag{14.88}$$

storage locations are required, where

$$M(\alpha) = M(\beta) = \sum_{i=0}^{\nu-1} |\Sigma_{t_i}(C)| = V - 1. \tag{14.89}$$

If the LLRs of estimated code bits are to be used for further decoding process, they must also be stored, which requires another

$$M(L) = n \tag{14.90}$$

storage locations. Therefore, the total storage requirement for MAP decoding based on the sectionalized trellis $T(\Lambda)$ is

$$M(\Lambda) = M(\gamma) + M(\alpha, \beta) + M(L). \tag{14.91}$$

The computational complexity and storage requirement of MAP decoding of a linear block code very much depend on the sectionalization of the code trellis. A sectionalization that minimizes both is desirable; however, in general, such a sectionalization does not exist. If there is no severe constraint on the size of memory storage, we may choose a sectionalization that minimizes the computational complexity. Based on the foregoing analysis, decoding computation of the MAP algorithm involves two kinds of real-number operations, additions and multiplications, in every decoding step. A multiplication operation is more complex than an addition operation, and the operations cannot be treated the same (have the same weight) in the minimization of computational complexity. This makes it difficult (if not impossible) to find a sectionalization that minimizes both the number of additions and the number of multiplications. Because the number of multiplications required in decoding is much larger than the number of additions required, and a multiplication operation is much more complex than an addition operation, we may just find a trellis sectionalization to minimize the total number of multiplications with a constraint that the number of additions may not exceed a times the number of additions based on the bit-level trellis; or we can weight an addition operation as a fraction of a multiplication operation, say, $1/b$ of a multiplication. Then, we find a trellis sectionalization to minimize the total number of multiplications and weighted additions. The Lafourcade–Vardy algorithm [3] presented in Section 14.1 can be used to find such an optimum trellis sectionalization to minimize the total number of multiplications and weighted additions. Optimum trellis sectionalizations (in terms of minimizing the number of multiplication operations with constraint $a = 1.2$) of some RM codes and the (24, 12) Golay code for MAP decoding are given in Table 14.2. For comparison, the computational complexity based on the bit-level trellis for each code is also included. From this table we see that a bit-level trellis requires much greater computational complexity than an optimum sectionalized trellis. We also see that the optimum trellis sectionalization in terms of minimizing the number of multiplication operations may not reduce the number of addition operations. Thus, there is a trade-off between the number of multiplications and the number of additions. Optimum trellis sectionalizations (in terms of minimizing the total number of multiplications and weighted additions with 1/5 ($b = 5$)) of some block codes for MAP decoding are given in Table 14.3.

TABLE 14.2: Optimum trellis sectionalizations of some block codes for MAP decoding with constraint $a = 1.2$.

	Bit-level trellis			Optimum sectionalization			
Code	Multiplication	Addition	Memory	Boundary location	Multiplication	Addition	Memory
$RM(8, 4)$	192	50	57	{0, 1, 2, 4, 6, 7, 8}	128	58	41
$RM(16, 5)$	720	186	197	{0, 1, 2, 4, 7, 8, 10, 12, 14, 15, 16}	400	218	119
$RM(16, 11)$	1,040	426	197	{0, 2, 4, 6, 7, 8, 10, 12, 13, 14, 15, 16}	808	504	139
$RM(32, 6)$	2,800	714	733	{0, 2, 4, 8, 16, 20, 24, 26, 28, 30, 31, 32}	872	856	251
$RM(32, 16)$	25,648	9,530	4,893	{0, 4, 8, 9, 12, 16, 17, 20, 23, 24, 27, 28, 29, 30, 31, 32}	11,128	11,404	1,295
$RM(32, 26)$	4,784	2,202	733	{0, 4, 6, 7, 8, 10, 12, 13, 14, 16, 18, 19, 20, 21, 22, 24, 25, 26, 28, 29, 30, 31, 32}	4,088	2,639	591
$RM(64, 7)$	11,056	2,794	2,829	{0, 2, 4, 8, 12, 16, 24, 32, 44, 48, 52, 58, 60, 62, 63, 64}	2,216	3,344	627
$RM(64, 22)$	1,500,272	475,258	325,053	{0, 8, 16, 24, 28, 31, 32, 34, 40, 46, 47, 48, 54, 55, 56, 59, 60, 61, 62, 63, 64}	348,536	569,812	37,795
$RM(64, 42)$	2,197,616	998,266	325,053	{0, 1, 8, 12, 14, 16, 17, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54 56, 58, 59, 60, 61, 62, 63, 64}	1,454,720	1,196,690	139,453
$RM(64, 57)$	20,464	9,850	2,829	{0, 4, 6, 7, 8, 9, 11, 12, 13, 15, 16, 18, 19, 20, 21, 23, 24, 25, 26, 27, 28, 29, 30, 32, 34, 36, 37, 38, 39, 40, 42, 43, 44, 45, 46, 48, 49, 50, 51, 52, 53, 56, 57, 58, 60, 61, 62, 63, 64}	19,128	11,811	2,303
Golay(24, 12)	14,368	5,322	2,757	{0, 3, 8, 9, 12, 15, 16, 19, 20, 21, 22, 23, 24}	6,742	6,372	813

TABLE 14.3: Optimum trellis sectionalizations of some block codes for MAP decoding with $b = 5$.

Code	Bit-level trellis operations	Optimum sectionalization	
		Boundary location	Operations
$RM(8, 4)$	202	{0, 8}	74
$RM(16, 5)$	757	{0, 16}	247
$RM(16, 11)$	1,125	{0, 4, 6, 8, 10, 12, 16}	796
$RM(32, 6)$	2,942	{0, 8, 16, 24, 32}	697
$RM(32, 16)$	27,554	{0, 3, 8, 12, 16, 20, 24, 29, 32}	11,970
$RM(32, 26)$	5,224	{0, 4, 6, 7, 8, 10, 11, 12, 13, 14, 16, 18, 19 20, 21, 22, 24, 25, 26, 28, 32}	4,532
$RM(64, 7)$	11,614	{0, 8, 16, 32, 48, 56, 64}	1,881
$RM(64, 22)$	1,595,323	{0, 3, 8, 16, 24, 32, 40, 56, 61, 64}	412,598
$RM(64, 42)$	2,397,269	{0, 1, 4, 8, 12, 14, 16, 20, 22, 24, 26, 28, 32, 36, 38, 40, 42, 44, 48, 50, 52, 56, 60, 63, 64}	1,652,378
$RM(64, 57)$	22,434	{0, 4, 6, 7, 8, 10, 11, 12, 13, 14, 15, 16, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 32 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46 48, 49, 50, 51, 52, 53, 54, 56, 57, 58, 60, 64}	21,013
Golay(24, 12)	15,432	{0, 3, 8, 12, 16, 21, 24}	6,742

14.6 MAX-LOG-MAP DECODING ALGORITHM

To achieve optimum bit-error performance, the MAP decoding algorithm requires a very large number of real computations, especially multiplications. For long codes with large trellises, this large computational complexity makes implementation of this decoding algorithm very difficult and impractical. If we do not insist on attaining optimum error performance, a suboptimum MAP decoding algorithm can be devised to provide a very efficient trade-off between error performance and decoding computational complexity. This suboptimum algorithm is known as the Max-log-MAP algorithm and is based on a very simple approximation for the logarithm of a sum of real numbers [30]–[32].

14.6.1 Max-log-MAP Decoding Based on a Bit-Level Trellis

For a finite set of real numbers $\{\delta_1, \delta_2, \cdots, \delta_q\}$ the following approximation holds:

$$\log\left(\sum_{i=1}^{q} e^{\delta_i}\right) \simeq (\max_{1 \le i \le q} \{\delta_i\}). \tag{14.92}$$

This approximation is called the *maximum logarithm* (*max-log*) *approximation*.

With the max-log approximation, we can approximate the LLR of an estimated code bit given by (14.40) by

$$\tilde{L}(v_i) = \max_{(s',s)\in B_i^1(C)} \{\log \alpha_i(s') + \log \gamma_i(s', s) + \log \beta_{i+1}(s)\} - \max_{(s',s)\in B_i^0(C)} \{\log \alpha_i(s') + \log \gamma_i(s', s) + \log \beta_{i+1}(s)\}. \tag{14.93}$$

From (14.41), (14.42), and (14.43), we have

$$\log \alpha_i(s) = \max_{s'\in \Omega_{i-1}^{(c)}(s)} \{\log \alpha_{i-1}(s') + \log \gamma_{i-1}(s', s)\}, \tag{14.94}$$

$$\log \beta_i(s) = \max_{s'\in \Omega_{i+1}^{(d)}(s)} \{\log \gamma_i(s, s') + \log \beta_{i+1}(s')\}, \tag{14.95}$$

$$\log \gamma_i(s', s) = -(r_i - c_i)^2. \tag{14.96}$$

where $c_i = 2v_i - 1$. (Note that since N_0 and $1/\sqrt{\pi N_0}$ are constants, we simply use $-(r_i - c_i)^2$ as the branch metric.) The metrics $\log \alpha_i(s)$ and $\log \beta_i(s)$ are simply the forward and backward metrics of state s, respectively, and they can be computed recursively with initial conditions $\log \alpha_0(s_0) = 0$ and $\log \beta_n(s_f) = 0$. The metric $\log \gamma_i(s', s)$ is simply the branch metric. The sum $\log \alpha_i(s') + \log \gamma_i(s', s) + \log \beta_{i+1}(s)$ is simply the path metric corresponding to the code bit v_i on the branch (s', s).

The Max-log-MAP decoding algorithm is to compute the LLR $\tilde{L}(v_i)$ for each code bit based on (14.93). First, the forward and backward recursions are carried out to compute the metrics $\log \alpha_i(s)$ and $\log \beta_i(s)$ for all the states in the code trellis. The $\tilde{L}(v_i)$ is then computed by executing the *add-compare-select-subtract* (ACSS) process (similar to the Viterbi algorithm). Bidirectional decoding can be applied to reduce the decoding time.

Again, assume that $\log(\cdot)$ is done by table lookup. Then, the Max-log-MAP decoding requires mainly additions (including subtraction) and comparisons. Because a comparison operation is as complex as an addition operation, it is regarded as an addition-equivalent operation. Computational complexity analysis of the Max-log-MAP is quite straightforward, based simply on (14.93) through (14.96).

For each trellis section, $\log \gamma_i(s', s)$ has only two values, corresponding to $c_i = \pm 1$ (or $v_i = 1$ or 0). Computing $\log \gamma_i(s', s)$'s in the entire decoding process requires only $2n$ additions and $2n$ multiplications. Computing α's and β's, requires

$$N_a(\alpha) = N_a(\beta) = \mathcal{E} \tag{14.97}$$

additions each and

$$N_c(\alpha) = N_c(\beta) = \mathcal{E} - V + 1 \tag{14.98}$$

comparisons. To compute $\tilde{L}(v_i)$ for $0 \leq i < n$ requires a total of

$$N_a(\tilde{L}) = 2\mathcal{E} \tag{14.99}$$

additions, a total of

$$N_c(\tilde{L}) = \mathcal{E} - 2n \tag{14.100}$$

comparisons, and

$$N_s(\tilde{L}) = n \tag{14.101}$$

subtractions. Therefore, the Max-log-MAP decoding requires a total of

$$N_a = 7\mathcal{E} - 2V + n + 2 \tag{14.102}$$

addition-equivalent operations and $4n$ multiplications. Because $\mathcal{E} \gg n$, the number of multiplications required in the decoding is very small compared with the number of addition-equivalent operations and hence can be ignored in measuring complexity.

To store γ's, α's, β's, and $\tilde{L}$'s requires $2V + 3n - 2$ storage locations.

The error performance of Max-log-MAP decoding is very close to that of MAP decoding while reducing the computational complexity significantly. Figure 14.11 depicts the error performances of the (32, 16) RM code using both MAP and Max-log-MAP decodings. We see that Max-log-MAP decoding results in only a very small performance degradation, less than 0.1 dB. The Max-log-MAP decoding algorithm is actually equivalent to the soft-output Viterbi decoding algorithm (SOVA) [35, 36].

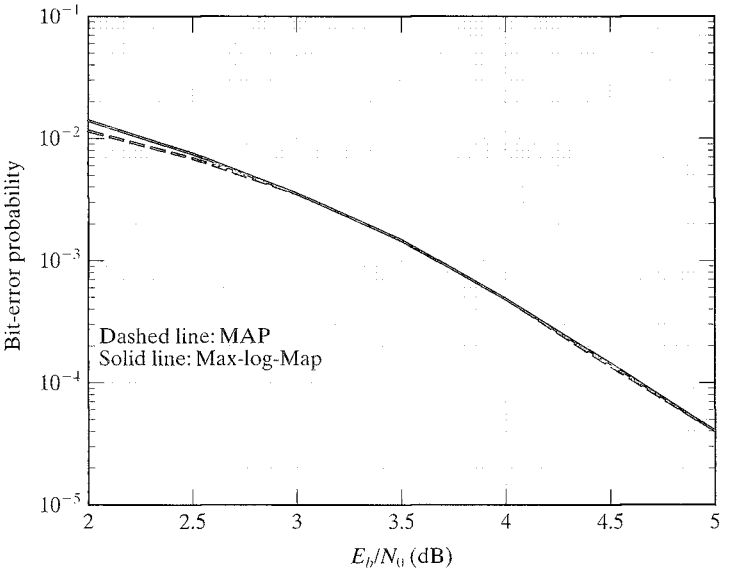

FIGURE 14.11: The bit-error performances of the (32, 16) RM code using MAP and Max-log-MAP decodings.

14.6.2 Max-log-MAP Decoding Based on a Sectionalized Trellis

Using a max-log approximation, we can approximate the LLR of an estimated code bit given by (14.61) based on a sectionalized trellis $T(\Lambda)$ by [34]:

$$\tilde{L}(v_l) = \max_{\substack{(s',s) \\ v_l=1}} \{\log \alpha_{t_i}(s') + \log \gamma_{t_i}(L_{v_l}^{(1)}(s', s)) + \log \beta_{t_{i+1}}(s)\}$$

$$- \max_{\substack{(s',s) \\ v_l=0}} \{\log \alpha_{t_i}(s') + \log \gamma_{t_i}(L_{v_l}^{(0)}(s', s)) + \log \beta_{t_{i+1}}(s)\}, \quad (14.103)$$

for $t_i \leq l < t_{i+1}$. It follows from (14.55) through (14.59) that the state and composite branch metrics are given by

$$\log \alpha_i(s) = \max_{s' \in \Omega_{t_{i-1}}^{(c)}(s)} \{\log \alpha_{t_{i-1}}(s') + \log \gamma_{t_{i-1}}(L(s', s))\}, \quad (14.104)$$

$$\log \beta_i(s) = \max_{s' \in \Omega_{t_{i+1}}^{(d)}(s)} \{\log \gamma_{t_i}(L(s, s')) + \log \beta_{t_{i+1}}(s')\}, \quad (14.105)$$

$$\log \gamma_{t_i}(L(s', s)) = \max_{\mathbb{b}(s',s) \in L(s',s)} \{\log \gamma_{t_i}(\mathbb{b}(s', s))\}, \quad (14.106)$$

$$\log \gamma_{t_i}(L_{v_l}^{(0)}(s', s)) = \max_{\substack{\mathbb{b}(s',s) \in L(s',s) \\ v_l=0}} \{\log \gamma_{t_i}(\mathbb{b}(s', s))\}, \quad (14.107)$$

$$\log \gamma_{t_i}(L_{v_l}^{(1)}(s', s)) = \max_{\substack{\mathbb{b}(s',s) \in L(s',s) \\ v_l=1}} \{\log \gamma_{t_i}(\mathbb{b}(s', s))\}, \quad (14.108)$$

for $t_i \leq l < t_{i+1}$.

To carry out the decoding based on $T(\Lambda)$, we preprocess the parallel branches of each distinct composite branch $L(s', s)$ to obtain the composite branch metric $\log \gamma_{t_i}(L(s', s))$ and the composite branch metrics $\log \gamma_{t_i}(L_{v_l}^{(0)}(s', s))$ and $\log \gamma_{t_i}(L_{v_l}^{(1)}(s', s))$ corresponding to code bit v_l for $t_i \leq l < t_{i+1}$. These metrics are stored in a branch metric table. The branch metric tables are then used to compute state metrics $\log \alpha_{t_i}(s)$ and $\log \beta_{t_i}(s)$, and LLR $\tilde{L}(v_l)$, for all the states in $T(\Lambda)$ and all the code bits. In the decoding process each composite branch $L(s', s)$ is regarded as a single branch, which is assigned a branch metric, $\gamma_{t_i}(L(s', s))$, and m_i pairs of metrics, $(\gamma_{t_i}(L_{v_l}^{(0)}(s', s)), (\gamma_{t_i}(L_{v_l}^{(1)}(s, s'))$, corresponding to m_i code bits on the branch where $m_i = t_{i+1} - t_i$. Computations of state metrics are based on (14.104) and (14.105). Computations of the LLRs of code bits are based on (14.103) using the ACSS process. If bidirectional decoding is performed, the forward and backward state metrics are computed along with the branch metrics in both directions. When the forward and backward state metrics at the two boundary locations of a trellis section have been computed, computations of the LLRs of the code bits corresponding to the trellis section begin.

For AWGN and BPSK signaling, we can use (14.63) to compute $\log \gamma_{t_i}(\mathbb{b}(s', s))$. Because N_0 and $\left(\frac{1}{\pi N_0}\right)^{m_i/2}$ are constant in the trellis section T_i, we can use

$$\log \gamma_{t_i}(\mathbb{b}(s', s)) \triangleq \sum_{l=t_i}^{t_{i+1}-1} r_l \cdot (2v_l - 1) \quad (14.109)$$

as the metric of branch $\mathbb{b}(s', s)$ in computing $\log \gamma_{t_i}(L(s', s))$'s, $\log \gamma_{t_i}(L_{v_l}^{(0)}(s', s))$'s, and $\log \gamma_{t_i}(L_{v_l}^{(1)}(s', s))$'s.

Let $\mathbb{b}^*(s', s) \triangleq (v_{t_i}^*, v_{t_i+1}^*, \cdots, v_{t_{i+1}-1}^*)$ be the branch that has the largest branch metric among the parallel branches in $L(s', s)$. From (14.106) we find that

$$\log \gamma_{t_i}(L(s', s)) = \log \gamma_{t_i}(\mathbb{b}^*(s', s)). \tag{14.110}$$

Then, it follows from (14.106) through (14.108), and (14.110), that

$$\begin{aligned} \log \gamma_{t_i}(L(s', s)) &= \max\{\log \gamma_{t_i}(L_{v_l}^{(0)}(s', s)), \log \gamma_{t_i}(L_{v_l}^{(1)}(s', s))\} \\ &= \begin{cases} \log \gamma_{t_i}(L_{v_l}^{(0)}(s', s)), & \text{if } v_l^* = 0, \\ \log \gamma_{t_i}(L_{v_l}^{(1)}(s', s)), & \text{if } v_l^* = 1, \end{cases} \end{aligned} \tag{14.111}$$

for $t_i \le l < t_{i+1}$. Based on (14.111), we can construct the branch metric table for the trellis section T_i using the following procedure-B:

1. We compute $\log \gamma_{t_i}(L(s', s))$ from (14.106) for each distinct composite branch in T_i.
2. For each code bit v_l in T_i, based on $\mathbb{b}^*(s', s)$ and from (14.111), we first check whether $v_l^* = 0$ or 1 (a logic operation) to determine which of $\log \gamma_{t_i}(L_{v_l}^{(0)}(s', s))$ and $\log \gamma_{t_i}(L_{v_l}^{(1)}(s', s))$ is equal to $\log \gamma_{t_i}(L(s', s))$. Then, we need to compute only the one between $\log \gamma_{t_i}(L_{v_l}^{(0)}(s', s))$ and $\log \gamma_{t_i}(L_{v_l}^{(1)}(s', s))$ that is not equal to $\log \gamma_{t_i}(L(s', s))$.

At the second stage of the Max-log-MAP decoding, the state metrics $\log \alpha_{t_i}(s)$'s and $\log \beta_{t_i}(s)$'s are computed from (14.104) and (14.105) recursively with initial conditions $\log \alpha_{t_0}(s_0) = 0$, and $\log \beta_{t_v}(s_f) = 0$. For bidirectional decoding, $\log \alpha_{t_i}(s)$'s and $\log \beta_{t_i}(s)$'s are computed simultaneously from both directions of the trellis $T(\Lambda)$ while the branch metric tables are being constructed, section by section. Once the state metrics $\log \alpha_{t_i}(s')$'s and $\log \beta_{t_{i+1}}(s)$'s at the section boundary locations t_i and t_{i+1} have been computed and the branch metric table for trellis section T_i has been constructed, we can compute the LLRs of the estimated code bits v_l for $t_i \le l < t_{i+1}$ from (14.103) by executing the ACSS process.

To compute the LLR $\tilde{L}(v_l)$ efficiently, we define

$$R^{(i)} \triangleq \max_{(s', s)} \{\log \alpha_{t_i}(s') + \log \gamma_{t_i}(L(s', s)) + \log \beta_{t_{i+1}}(s)\}, \tag{14.112}$$

$$R_0^{(i)}(v_l) \triangleq \max_{\substack{(s', s) \\ v_l = 0}} \{\log \alpha_{t_i}(s') + \log \gamma_{t_i}(L_{v_l}^{(0)}(s', s)) + \log \beta_{t_{i+1}}(s)\}, \tag{14.113}$$

$$R_1^{(i)}(v_l) \triangleq \max_{\substack{(s', s) \\ v_l = 1}} \{\log \alpha_{t_i}(s') + \log \gamma_{t_i}(L_{v_l}^{(1)}(s', s)) + \log \beta_{t_{i+1}}(s)\}, \tag{14.114}$$

for $0 \le i < v$. From (14.60), (14.112) through (14.114), and by using the max-log approximation, we readily see that $t_i \le l < t_{i+1}$,

$$R^{(i)} = \max\{R_0^{(i)}(v_l), R_1^{(i)}(v_l)\}. \tag{14.115}$$

It follows from (14.111) through (14.114) that

$$R^{(i)} = \begin{cases} R_0^{(i)}(v_l), & \text{if } v_l^* = 0, \\ R_1^{(i)}(v_l), & \text{if } v_l^* = 1. \end{cases} \tag{14.116}$$

Then, an efficient procedure for computing $R_0^{(i)}(v_l)$, $R_1^{(i)}(v_l)$, and $\tilde{L}(v_l)$, called procedure-R, follows:

1. We compute $R^{(i)}$ based on (14.112).
2. For each code bit v_l in T_i, based on $\mathbb{b}^*(s', s)$ and from (14.116), we first check whether $v_l^* = 0$ or 1 to determine which of $R_0^{(i)}(v_l)$ and $R_1^{(i)}(v_l)$ is equal to $R^{(i)}$. Then, we compute the one that is not equal to $R^{(i)}$.
3. We compute $\tilde{L}(v_l)$ by taking the difference $R_1^{(i)}(v_l) - R_0^{(i)}(v_l)$, for $t_i \le l < t_{i+1}$.

The computational complexity for constructing the branch metric tables for the trellis sections can be analyzed by procedure-B. We readily find that construction of the branch metric table for the trellis section T_i requires a total of

$$N_c^i(\gamma) = B_i^d \cdot \{B_i^p - 1 + (B_i^p/2 - 1) \cdot m_i\} \tag{14.117}$$

comparisons and a total of

$$N_a^i(\gamma) = B_i^d \cdot B_i^p \cdot (m_i - 1) \tag{14.118}$$

additions.

From (14.104) and (14.105) we find that the computation of metrics of states at the boundaries of T_i requires a total of $N_a^i(\alpha) + N_a^i(\beta)$ additions and $N_c^i(\alpha) + N_c^i(\beta)$ comparisons, where

$$N_a^i(\alpha) = N_a^i(\beta) = B_i^c, \tag{14.119}$$

$$N_c^i(\alpha) = B_{i-1}^c - 2^{\rho_{t_i}}, \tag{14.120}$$

$$N_c^i(\beta) = B_i^c - 2^{\rho_{t_i}}. \tag{14.121}$$

To analyze the complexity of computing the LLRs given by (14.103), we follow procedure-R and find that a total of

$$N_c^i(\tilde{L}) = (B_i^c - 1) \cdot (m_i + 1) \tag{14.122}$$

comparisons and a total of

$$N_a^i(\tilde{L}) = (m_i + 2) \cdot B_i^c + m_i \tag{14.123}$$

additions (including subtractions) are required to compute the LLRs of the estimated code bits in T_i.

Because a comparison operation has the same complexity as an addition, it is regarded as an addition-equivalent operation. Therefore, to decode a received

sequence, the Max-log-MAP algorithm based on the sectionalized trellis $T(\Lambda)$ requires a total of

$$N_{ae}(\Lambda) = \sum_{i=0}^{\nu-1} \{N_a^i(\gamma) + N_c^i(\gamma) + N_a^i(\tilde{L}) + N_c^i(\tilde{L}) + N_a^i(\alpha) + N_c^i(\alpha) + N_a^i(\beta) + N_c^i(\beta)\} \tag{14.124}$$

addition-equivalent operations. $N_{ae}(\Lambda)$ is used as a measure of the computational complexity of the Max-log-MAP decoding algorithm based on a sectionalized trellis.

The storage requirement for the Max-log-MAP algorithm is the same as that for the MAP algorithm.

The computational complexity $N_{ae}(\Lambda)$ depends on the sectionalization Λ of the code trellis T. The sectionalization that minimized $N_{ae}(\Lambda)$ is called an *optimum sectionalization*. The Lafourcade–Vardy algorithm presented in Section 14.1 can be used to find such a sectionalization. Table 14.4 gives optimum sectionalizations of trellises for some RM codes with the Max-log-MAP decoding. For comparison, the computational complexities of these codes based on bit-level trellises are also included. We see that proper sectionalization reduces computational complexity significantly.

TABLE 14.4: Optimum trellis sectionalizations of some RM codes for Max-log-MAP decoding.

	Bit-level trellis		Optimum sectionalization		
$RM(r, m)$	Operations	Memory	Boundary location	Operations	Memory
$RM(1, 3)$	230	55	{0, , 4, 8}	156	81
$RM(1, 4)$	886	195	{0, 2, 4, 8, 12, 14, 16}	486	77
$RM(2, 4)$	1,446	195	{0, 2, 4, 6, 8, 10, 12, 14, 16}	1,222	109
$RM(1, 5)$	3,478	731	{0, 2, 4, 8, 16, 24, 28, 30, 32}	1,510	165
$RM(2, 5)$	35,142	4,891	{0, 1, 3, 5, 8, 12, 16, 20, 24, 27, 29, 31, 32}	22,078	919
$RM(3, 5)$	6,950	731	{0, 2, 4, 6, 7, 8, 10, 11, 12, 13, 14, 16, 18, 19, 20, 21, 22, 24, 25, 26, 28, 30, 32}	6,446	549
$RM(1, 6)$	13,782	2,827	{0, 2, 4, 8, 16, 24, 32, 40, 48, 56, 60, 62, 64}	5,094	469
$RM(2, 6)$	1,975,462	325,051	{0, 1, 3, 5, 8, 10, 16, 18, 24, 30, 32, 34, 40, 46, 48, 54, 56, 59, 61, 63, 64}	905,974	37,839
$RM(3, 6)$	3,195,814	325,051	{0, 2, 4, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62, 64}	2,646,566	141,925
$RM(4, 6)$	30,246	2,827	{0, 2, 4, 6, 7, 8, 10, 11, 12, 13, 14, 15, 16, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 32, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 48, 49, 50, 51, 52, 53, 54, 56, 57, 58, 60, 62, 64}	29,174	2,453

14.6.3 log-MAP Algorithm

The Max-log-MAP algorithm is a suboptimum realization of the MAP algorithm. Even though it gives an error performance very close to that of the MAP algorithm, as shown in Figure 14.11, it produces soft-output values inferior to those of the MAP algorithm, owing to the approximation of (14.92). Hence, when we use the Max-log-MAP algorithm in turbo decoding (see Chapter 16), the inferior reliability value (the soft output of the Max-log-MAP decoder) results in performance degradation compared with the optimum MAP decoder.

To overcome this problem, we can use the Jacobian logarithm

$$\log(e^{\delta_1} + e^{\delta_2}) = \max\{\delta_1, \delta_2\} + \log(1 + e^{-|\delta_2 - \delta_1|}) = \max\{\delta_1, \delta_2\} + f_c(|\delta_2 - \delta_1|), \tag{14.125}$$

where $f_c(\cdot)$ is a correction function. Then, for a finite set of real numbers $\{\delta_1, \cdots, \delta_q\}$, we can compute $\log(e^{\delta_1} + \cdots + e^{\delta_q})$ recursively. The recursion is initialized with the two terms given by (14.125). Suppose that $\log(e^{\delta_1} + \cdots + e^{\delta_{i-1}})$ with $1 < i \leq q$ is

TABLE 14.5: Optimum trellis sectionalizations of some RM codes for log-MAP decoding.

	Bit-level trellis		Optimum sectionalization		
$RM(r, m)$	Operations	Memory	Boundary location	Operations	Memory
$RM(1, 3)$	330	55	{0, 4, 8}	244	81
$RM(1, 4)$	1,258	195	{0, 1, 2, 4, 8, 12, 14, 15, 16}	866	83
$RM(2, 4)$	2,298	195	{0, 1, 2, 3, 4, 6, 8, 10, 12, 13, 14, 15, 16}	2,242	115
$RM(1, 5)$	4,906	731	{0, 1, 2, 4, 8, 16, 24, 28, 30, 31, 32}	2,946	171
$RM(2, 5)$	54,202	4,891	{0, 1, 2, 3, 5, 8, 9, 12, 15, 16, 17, 20, 23, 24, 27, 29, 30, 31, 32}	43,906	1,415
$RM(3, 5)$	11,354	731	{0, 1, 2, 3, 4, 6, 7, 8, 10, 11, 12, 13, 14, 16, 18, 19, 20, 21, 22, 24, 25, 26, 28, 29, 30, 31, 32}	11,210	555
$RM(1, 6)$	19,370	2,827	{0, 1, 2, 4, 8, 16, 24, 32, 40, 48, 56, 60, 62, 63, 64}	10,690	475
$RM(2, 6)$	2,925,978	325,051	{0, 1, 2, 3, 5, 8, 9, 12, 16, 17, 20, 24, 28, 31, 32, 33, 36, 40, 44, 47, 48, 52, 55, 56, 59, 61, 62, 63, 64}	1,933,682	63,819
$RM(3, 6)$	5,192,346	325,051	{0, 1, 2, 3, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 61, 62, 63, 64}	5,009,346	141,931
$RM(4, 6)$	49,946	2,827	{0, 1, 2, 3, 4, 6, 7, 8, 10, 11, 12, 13, 14, 15, 16, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 32, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 48, 49, 50, 51, 52, 53, 54, 56, 57, 58, 60, 61, 62, 63, 64}	49,618	2,459

known. Hence,

$$\begin{aligned}\log(e^{\delta_1} + \cdots + e^{\delta_i}) &= \log(\Delta + e^{\delta_i}) \\ &= \max\{\log \Delta, \delta_i\} + f_c(|\log \Delta - \delta_i|),\end{aligned} \tag{14.126}$$

with $\Delta = e^{\delta_1} + \cdots + e^{\delta_{i-1}}$. Based on this recursion, we modify the Max-log-MAP algorithm by using simple correction functions. This algorithm, called the *log-MAP algorithm* [31], gives the same error performance as the MAP algorithm but is easier to implement. Each correction term needs an additional one-dimensional lookup table and two additions based on (14.125). Consequently, the log-MAP algorithm requires only additions and comparisons to compute the LLRs.

The storage requirement for the log-MAP algorithm is the same as those for the MAP and Max-log-MAP algorithms, assuming the storage of the lookup tables is negligible.

Consider the computational complexity of the log-MAP algorithm. Because two extra additions are required per comparison to calculate $f_c(\cdot)$ in (14.125), a total of $N_a^i(\gamma) + 3N_c^i(\gamma)$, $N_a^i(\alpha) + 3N_c^i(\alpha)$, $N_a^i(\beta) + 3N_a^i(\beta)$, and $N_a^i(\tilde{L}) + 3N_c^i(\tilde{L})$ addition-equivalent operations are required to compute $\log \gamma$'s, $\log \alpha$'s, $\log \beta$'s, and LLRs in T_i, respectively, where $N_a^i()$'s and $N_c^i()$'s are the numbers of additions and comparisons evaluated for the Max-log-MAP algorithm.

Table 14.5 gives optimum sectionalizations (in terms of minimizing the number of addition-equivalent operations) of trellises for some RM codes with the log-MAP decoding. For comparison, the computational complexities and storage requirements of these codes based on bit-level trellises are also included. We also see that proper sectionalization reduces computational complexity and storage requirements for the log-MAP algorithm.

PROBLEMS

14.1 Suppose the (8, 4) RM code is decoded with the Viterbi algorithm. Determine the number of real operations (additions and comparisons) required for the following trellises:
- a. The eight-section bit-level trellis.
- b. The uniform four-section (two-bits per section) trellis shown in Figure 9.17.
- c. Optimum sectionalization based on the Lafourcade–Vardy algorithm.

14.2 Suppose the (8, 4) RM code is decoded with the differential Viterbi decoding algorithm based on the uniform 4-section trellis of the code. Determine the number of real operations required to decode the code.

14.3 The first-order RM code of length 16 is a (16, 5) linear code with a minimum distance of 8. Decode this code with the Viterbi algorithm. Determine the number of real operations required for the decoding based on the following trellis sectionalizations:
- a. The 16-section bit-level trellis.
- b. The uniform eight-section trellis.
- c. The uniform four-section trellis.
- d. Optimum sectionalization based on the Lafourcade–Vardy algorithm.

14.4 Decode the (16, 5) first-order RM code with the differential Viterbi decoding algorithm based on the uniform four-section trellis. For each section, determine the parallel components, the set of branches leaving a state at the left end of a parallel component, and the set of branches entering a state at the right end of

a component. Decompose each component into 2-state butterflies with doubly complementary structure. Determine the total number of real operations required to decode the code.

14.5 Decode the (8, 4) RM code with the trellis-based recursive MLD algorithm. At the beginning (or bottom) of the recursion, the code is divided into four sections, and each section consists of 2 bits. The composite path metric table for each of these basic sections is constructed directly. Devise a recursion procedure to combine these metric tables to form metric tables for longer sections until the full length of the code is reached (i.e., a procedure for combining metric tables). For each combination of two tables using the CombCPMT(x, y; z) procedure, construct the two-section trellis $T(\{x, y; z\})$ for the punctured code $p_{x,y}(C)$. Determine the number of real operations required to decode the code with the RMLD-(I,V) algorithm.

14.6 Decode the (16, 5) RM code with the RMLD-(I,V) algorithm using uniform sectionalization. At the beginning, the code is divided into eight sections, of 2 bits each. Devise a recursion procedure to combine composite path metric tables. For each combination of two adjacent metric tables, construct the special two-section trellis for the corresponding punctured code. Determine the total number of real operations required to decode the code.

14.7 Repeat Problem 14.6 by dividing the code into four sections, 4 bits per section, at the beginning of the recursion. Compare the computation complexity of this recursion with that of the recursion devised in Problem 14.6.

14.8 Devise an iterative decoding algorithm based on a minimum-weight trellis search using the ordered statistic decoding with order-1 reprocessing (presented in Section 10.8.30) to generate candidate codewords for optimality tests. Analyze the computational complexity of your algorithm. To reduce decoding computational complexity, the order i should be small, say $i = 0, 1$, or 2. The advantage of ordered statistic decoding over the Chase-II decoding is that it never fails to generate candidate codewords.

14.9 Simulate the error performance of the iterative decoding algorithm devised in Problem 14.8 for the (32, 16) RM code using order-1 reprocessing to generate 17 candidate codewords for testing and search of the ML codeword. Determine the average numbers of real operations and decoding iterations required for various SNR.

14.10 Decode the (32, 16) RM code with MAP and Max-log-Map decoding algorithms based on a uniform four-section trellis. Simulate and compare the error performances for two algorithms, and compare their computational complexities.

14.11 The (32, 16) RM code can be decomposed into eight parallel and structurally identical four-section subtrellises. Decode this code with the parallel Max-log-MAP algorithm. Compute the number of real operations required to process a single subtrellis and the total number of real operations required to decode the code. Also determine the size of the storage required to store the branch metrics, state metrics, and the likelihood ratios.

BIBLIOGRAPHY

1. A. J. Viterbi, "Error Bounds for Convolutional Codes and an Asymptotically Optimum Decoding Algorithm," *IEEE Trans. Inform. Theory*, 13: 260–69, April 1967.

2. G. D. Forney, Jr., "The Viterbi Algorithm," *Proc. IEEE*, 61: 268–78, March 1973.

3. A. Lafourcade and A. Vardy, "Optimum Sectionalization of a Trellis," *IEEE Trans. Inform. Theory*, 42: 689–703, May 1996.

4. R. J. McEliece, "On the BCJR Trellis for Linear Block Codes," *IEEE Trans. Inform. Theory*, 42: 1072–92, July 1996.

5. R. J. McEliece, "The Viterbi Decoding Complexity of Linear Block Codes," *Proc. IEEE Intl. Symp. Inform. Theory*, p. 341, Trondheim, Norway, June 1994.

6. H. Thirumoorthy, "Efficient Near-Optimum Decoding Algorithms and Trellis Structure for Linear Block Codes," Ph.D. dissertation, Dept. of Electrical Engineering, University of Hawaii at Manoa, November 1996.

7. B. Honary and G. Markarian, *Trellis Decoding of Block Codes*, Kluwer Academic, Boston, Mass., 1997.

8. S. Lin, T. Kasami, T. Fujiwara, and M. P. C. Fossorier, *Trellises and Trellis-Based Decoding Algorithms for Linear Block Codes*, Kluwer Academic, Boston, Mass., 1998.

9. H. T. Moorthy, S. Lin, and G. Uehara, "Good Trellises for IC Implementation of Viterbi Decoders for Linear block Codes," *IEEE Trans. Commun.*, 45: 52–63, January 1997.

10. E. Nakamura, G. Uehara, C. Chu, and S. Lin, "A 755 Mb/s Viterbi Decoder for $RM(64, 35, 8)$," *Proc. IEEE Intl. Solid-State Circuit Conf.*, San Francisco, Calif., February 1999.

11. T. Kasami, T. Takata, T. Fujiwara, and S. Lin, "On Structural Complexity of the L-Section Minimum Trellis Diagrams for Binary Linear Block Codes," *IEICE Trans. Fundamentals*, E76-A (no. 9): 1411–21, September 1993.

12. N. Deo, *Graph Theory with Applications to Engineering and Computer Science*, Prentice Hall, Englewood Cliffs, N.J., 1974.

13. M. P. C. Fossorier, S. Lin, and D. J. Rhee, "Differential Trellis Decoding of Convolutional Codes," *IEEE Trans. Inform. Theory*, 46: 1046–53, May 2000.

14. T. Fujiwara, H. Yamamoto, T. Kasami, and S. Lin, "A Recursive Maximum Likelihood Decoding Procedure for a Linear Block Code Using an Optimum Sectionalized Trellis Diagram," *Proc. Thirty-Third Annual Allerton Conf. on Commun. Control and Computing*, pp. 700–709, October 4–6, 1995.

15. T. Fujiwara, H. Yamamoto, T. Kasami, and S. Lin, "A Trellis-Based Recursive Maximum-Likelihood Decoding Algorithm for Binary Linear Block Codes," *IEEE Trans. Inform. Theory*, 44: 714–29, March 1998.

16. T. Kasami, H. Tokushige, T. Fujiwara, H. Yamamoto, and S. Lin, "A Recursive Maximum Likelihood Decoding Algorithm for Some Transitive Invariant Binary Block Codes," *IEICE Trans. Fundamentals*, E81-A (no. 9): 1916–24, September 1998.

17. H. Moorthy, S. Lin, and T. Kasami, "Soft-Decision Decoding of Binary Linear Block Codes Based on an Iterative Search Algorithm," *IEEE Trans. Inform. Theory*, 43: 1030–40, May 1997.

18. D. Chase, "A Class of Algorithms for Decoding of Block Codes with Channel Measurement Information," *IEEE Trans. Inform. Theory*, 18: 170–81, January 1972.

19. T. Koumoto, T. Takata, T. Kasami, and S. Lin, "A Low-Weight Trellis Based Soft-Decision Decoding Algorithm for Binary Linear Block Codes," *IEEE Trans. Inform. Theory*, 45: 731–41, March 1999.

20. A. R. Calderbank, "Multilevel Codes and Multistage Decoding," *IEEE Trans. Commun.*, 37: 222–29, March 1989.

21. T. Takata, S. Ujita, T. Kasami, and S. Lin, "Multistage Decoding of Multilevel Block M-PSK Modulation Codes," *IEEE Trans. Inform. Theory*, 39: 1024–18, July 1993.

22. J. Wu, S. Lin, T. Kasami, T. Fujiwara, and T. Takata, "An Upper Bound on the Effective Error Coefficient of Two-Stage Decoding and Good Two-Level Decomposable Codes," *IEEE Trans. Commun.*, 42: 813–18, February/March/April, 1994.

23. T. Takata, Y. Yamashita, T. Fujiwara, T. Kasami, and S. Lin, "Suboptimum Decoding of Decomposable Codes," *IEEE Trans. Inform. Theory*, 40: 1392–1405, September 1994.

24. G. Schnabl and M. Bossert, "Soft-Decision Decoding of Reed–Muller Codes as Generalized Concatenated Codes," *IEEE Trans. Inform. Theory*, 41: 304–8, January 1995.

25. D. Stojanovic, M. P. C. Fossorier, and S. Lin, "Iterative Multistage Maximum Likelihood Decoding of Multilevel Codes," *Proc. Coding and Cryptograph*, pp. 91–101, Paris, France, January 11–14, 1999.

26. U. Wachsmann, R. F. H. Fischer, and J. B. Huber, "Multilevel Codes: Theoretical Concepts and Practical Design Rules," *IEEE Trans. Inform. Theory*, 45: 1361–91, July 1999.

27. N. Seshadri and C. W. Sundberg, "List Viterbi Decoding Algorithms with Applications," *IEEE Trans. Commun.*, 42: 313–22, February/March/April, 1994.

28. L. R. Bahl, J. Cocke, F. Jelinek, and J. Raviv, "Optimum Decoding of Linear Codes for Minimizing Symbol Error Rate," *IEEE Trans. Inform. Theory*, 20: 284–87, March 1974.

29. C. Berrou, A. Glavieux, and P. Thitimajshima, "Near Shannon Limit Error-Correcting Coding and Decoding: Turbo Codes," *Proc. IEEE Intl. Conf. Commun.*, Geneva, Switzerland, pp. 1064–70, May 1993.

30. J. Hagenauer, E. Offer, and L. Papke, "Iterative Decoding of Binary Block and Convolutional Codes," *IEEE Trans. Inform. Theory*, 42: 429–45, March 1996.

31. P. Robinson, E. Villebrun, and P. Hoeher, "A Comparison of Optimal Suboptimal MAP Decoding Algorithms Operating in Log Domain," *Proc. Intl Conf. Commun.*, Seattle, Wash., pp. 1009–13, 1995.

32. A. J. Viterbi, "An Intuitive Justification and a Simplified Implementation of the MAP Decoder for Convolutional Codes," *IEEE J. Selected Areas Commun.*, 16: 260–64, February 1998.

33. Y. Liu, M. P. C. Fossorier, and S. Lin, "MAP Algorithms for Decoding Linear Block Codes Based on Sectionalized Trellis Diagrams," *Proc. IEEE GlobeCom. Conf.*, Sydney, Australia, pp. 562–66, November 1998.

34. Y. Liu, M. P. C. Fossorier, and S. Lin, "MAP Algorithms for Decoding Linear Block Codes Based on Sectionalized Trellis Diagrams," *IEEE Trans. Commun.*, 48: 577–587, April 2000.

35. J. Hagenauer and P. Hoeher, "A Viterbi Algorithm with Soft-Decision Outputs and Its Applications," *Proc. IEEE GlobeCom. Conf.*, Dallas, Tex., pp. 1680–86, November 1989.

36. M. P. C. Fossorier, F. Burkert, S. Lin, and J. Hagenauer, "On the Equivalence between SOVA and Max-log-MAP Decoding," *IEEE Commun. Lett.*, 2: 137–49, May 1998.