Implementing the Viterbi Algorithm

Fundamentals and real-time issues for processor designers

HUI-LING LOU

The Viterbi algorithm, an application of dynamic programming, is widely used for estimation and detection problems in digital communications and signal processing. It is used to detect signals in communication channels with memory, and to decode sequential error-control codes that are used to enhance the performance of digital communication systems. The Viterbi algorithm is also used in speech and character recognition tasks where the speech signals or characters are modeled by hidden Markov models. This article explains the basics of the Viterbi algorithm as applied to systems in digital communication systems, and speech and character recognition. It also focuses on the operations and the practical memory requirements to implement the Viterbi algorithm in real-time.

Overview

The Viterbi Algorithm (VA) finds the most-likely state transition sequence in a state diagram, given a sequence of symbols. In digital communication systems, the VA is widely used to detect sequential error-control codes and to detect symbols in channels with memory by finding the most-likely noiseless sequence, given a sequence of symbols that are corrupted by noise such as additive white gaussian noise [1-6]. With each symbol in the noisy sequence, the VA recursively finds the most-likely transition coming into each state. The VA can also be applied to speech and character recognition tasks where the speech symbols or characters are modeled by hidden Markov models (HMMs). Given an observation sequence representing a word or a character, the VA can be used to find the most-likely state sequence and the likelihood score of this sequence in a given HMM [7-10].

In summary, the Viterbi algorithm is used to find the most likely noiseless finite-state sequence, given a sequence of finite-state signals that are corrupted by noise. The finite-state signals are signals generated from a finite-state diagram such as the one shown in Fig. 1a. That is, given a sequence of input symbols and an initial state, one can derive a sequence of output symbols based on the transitions and their input/output relations in a given finite-state diagram. For example, in the Figure, if the initial state is 1, an input of -1 will cause a transition from state 1 to state -1, and the output symbol of this transition is 0.

To aid in visualizing the transitions from state to state, one often represents a finite-state diagram by a time-indexed equivalent called a trellis [2]. Figure 1b shows the trellis equivalent of the two-state state diagram in Fig. 1a. That is, at state 1 in Fig. 1b, an input symbol of -1 will cause a transition from state 1 to state -1, and the output symbol of this transition is 0. With this time-indexed representation of a state diagram, we can see that if there are four symbols in an input sequence, four *stages* of a trellis diagram is required to show the transitions (Fig 2).

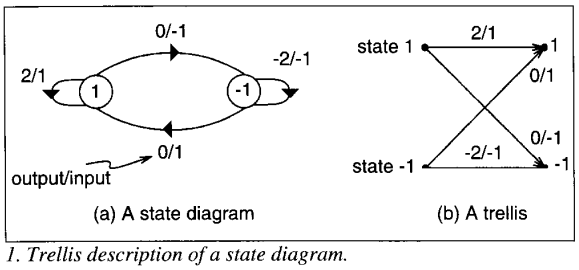
If a sequence of symbols, generated from a trellis, is corrupted by additive white gaussian noise (AWGN), the VA is the optimum method of finding the most-likely original noiseless sequence, given this sequence of noisy symbols [2-3]. The trellises can be a description of an error-control code (such as a convolutional code or a trellis code described later), or a communication channel (such as a partial-response channel).

Viterbi Detection

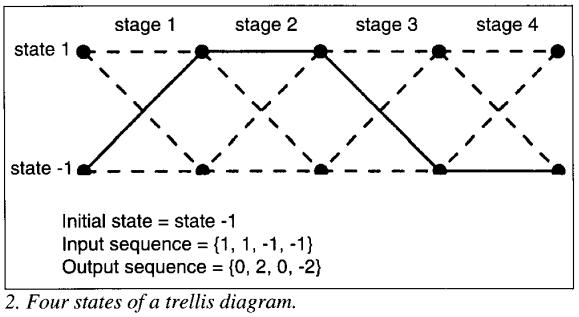
As described previously, given a received sequence of symbols corrupted by AWGN, the VA finds the sequence of symbols in the given trellis that is closest in distance to the received sequence of noisy symbols. This sequence computed is the global *most likely sequence*. When Euclidean distance is used as a distance measure, the VA is the optimal maximum-likelihood detection method in AWGN [2]. In practice, however, the Hamming distance (defined later) is often used, even though the performance of the VA is sub-optimal. Regardless of the distance measure (Euclidean or Hamming), the procedure to search for the most-likely sequence is the same. We will use Euclidean distance to illustrate the VA.

To compute the global most-likely sequence, the VA first recursively computes the *survivor path* entering each state. The survivor path for a given state is the sequence of symbols, entering the given state, that is closest in distance to the

1053-5888/95/$4.00©1995IEEE



1. Trellis description of a state diagram.



2. Four states of a trellis diagram.

received sequence of noisy symbols. The distance between the survivor path and the sequence of noisy symbols is called the *path metric* for that state. After the survivor paths entering all states are computed, the survivor path that has the minimum path metric is selected to be the global most-likely path.

Let us assume that the received sequence of noisy symbols is $\bar{y} = (y_1, y_2, y_3, \ldots y_k)$ At each recursion, that corresponds to one stage of a trellis, the VA computes the most likely transition coming into each state, and then updates the survivor path and the path metric for that state (Fig. 3). That is, at recursion n, the VA computes the most-likely transition coming into state j by computing the metrics of all the possible paths coming into state j, and then selecting the path with the minimum metric as the survivor path coming into state j, and its metric is the updated path metric for state j. If Euclidean distance is used, the metric of a path is the accumulated squared distances between the received sequence of noisy symbols and the ideal sequence of symbols in that path. That is, the metric of the path coming into state j from state i, at recursion n, is the sum of the branch metric of the transition from state i to state j and the path metric of state i computed at recursion $(n - 1)$. The branch metric is the squared distance between the received noisy symbol, y_n, and the ideal noiseless output symbol of that transition. (The branch metric for Hamming distance is described later). That is, the branch metric for the transition from state i to state j at recursion n is

$$B_{i,j,n} \equiv (y_n - C_{i,j})^2 \tag{1}$$

where $C_{i,j}$ is the output symbol of the transition from state i to state j. If we also define $M_{j,n}$ as the path metric for state j at recursion n, and $\{i\}$ as the set of states that have transitions to state j, then the most likely path coming into state j at recursion n is the path that has the minimum metric,

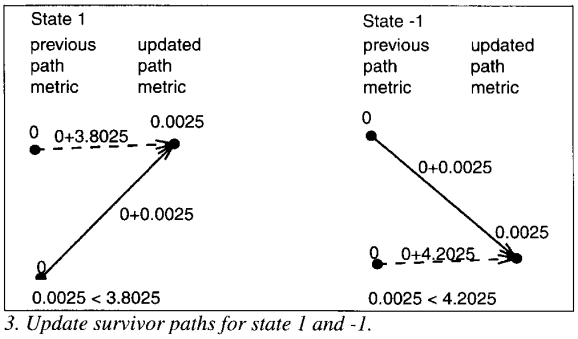
$$M_{j,n} = \min_{\{i\}} [M_{i,n-1} + B_{i,j,n}] \tag{2}$$

After the most likely transition to state j at recursion n is computed, the path metric for state j, $M_{j,n}$, is updated and this most likely transition, say from state m to state j, is appended to the survivor path of state m at recursion $(n - 1)$ in order to form the updated survivor path coming into state j at the current recursion, n.

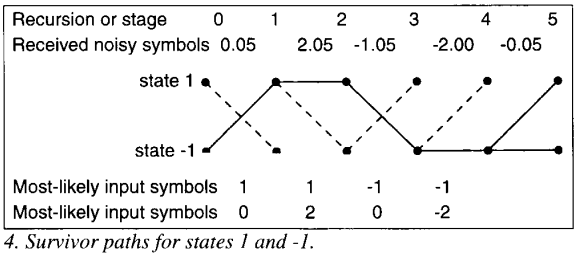
The path metrics and the survivor paths are updated for all states at each recursion. At the end of the recursions, recursion k in this case, the survivor path with the minimum path metric is selected to be the most likely path. The most likely path can be traced from the state that has the minimum path metric backwards.

In practice, the received noisy symbols arrives continuously. That is, $k \rightarrow \infty$ in the above case. From the description given above, one has to wait for an infinite amount of time before a most likely sequence can be determined. Note, however, that if the survivor paths for all states converge to one state, say state j, and at recursion n, all states will append the survivor path coming into state j at recursion n to their survivor paths. Thus, all states will have the identical survivor path sequence prior to stage n. Therefore, we can trace back from any state and obtain the most likely sequence up to stage n (Fig. 4).

For a given trellis and the performance of the Viterbi decoder required, one can determine the number of recursions required for the survivor paths to converge with very high probability. We call this number the *survivor path length*, L. Therefore, after an initial delay of L recursions, one can assume that the survivor paths for all states have converged at L stages ago. Thus, one can start from any state and



3. Update survivor paths for state 1 and -1.



4. Survivor paths for states 1 and -1.

trace its survivor path L stages backwards in order to find the most likely symbol at (L -1) stages ago. In practice, L is determined by computer simulation.

Example—Viterbi Detection

Given the trellis in Fig. 1 and the received noisy sequence $\bar{y}$ = (0.05, 2.05, -1.05, -2.00, -0.05) at recursion 1, we first calculate the branch metrics for all the possible transitions (Eq. 1):

$$B_{1,1,1} = (y_1 - C_{1,1})^2 = (0.05 - 2)^3 = 3.8025$$

$$B_{-1,-1,1} = (y_1 - C_{-1,-1})^2 = (0.05 + 2)^2 = 4.2025$$

$$B_{1,-1,1} = B_{-1,1,1} = (0.05)^2 = 0.0025$$

The next step is to update the survivor path coming into each state. There are two possible transitions coming into both state 1 and state -1. That is, in Eq. 2, {i} = {1, -1} for both states. Thus, we first compute the sums of the branch metrics, B, and their corresponding previous path metrics, M, for the two paths entering each state. After that, we select the path with the smaller sum as the survivor path coming into that state (Eq. 2). In Fig. 3, we assume that at the beginning of the recursions, all the path metrics are zero ($M_{1,0} = M_{-1,0} = 0$). Thus, for state 1 at recursion 1,

$$M_{1,1} = \min_{i=1,-1} [M_{i,0} + B_{i,1}]$$

and since

$$([M_{1,0} + B_{1,1}] = 0 + 3.8025) > ([M_{-1,0} + B_{-1,1}] = 0 + 0.0025)$$

Then, $M_{1,1} = 0.0025$. That is, the survivor path for state 1 is the transition from state -1 to state 1, and 0.0025 is the updated path metric for state 1. Similarly, we can calculate the survivor path for state -1. The survivor paths are shown in solid black lines in Fig. 3

If we recursively update the survivor paths for 5 stages, given the noisy sequence $\bar{y}$ we will have a survivor for each state as shown in Fig 4. To trace the survivor path for state 1, one can trace from state 1 backwards to state -1 (at stage 4) to state -1 (at stage 3), and so on. Note that since the two survivor paths for both states merge at state -1 at stage 4, we can trace back from either state and obtain the identical most likely sequence from stage 0 to stage 4. Further, if we know the most likely transitions, we can find the corresponding sequence of output and input symbols from the given trellis.

In summary, at each recursion or stage, there are three major steps in Viterbi detection:

1. Branch metric generation: The branch metrics for all possible transitions between all pairs of states in the given trellis are generated. The branch metric of a given transition is the squared distance between the noisy received symbol and the ideal output symbol associatedwith the transition (Eq. 1).

2. Survivor path and path metric update for all states: For all of the incoming transitions to each state, add their branch metrics to their corresponding previous path metrics. The path with the minimum sum is the survivor path for that state. A survivor path is selected and its path metric updated for each state at each recursion (Eq. 2).

3. Most likely path trace back: The survivor path of a given state is traced L stages back in order to determine the most likely transition at that stage. That is, the most likely symbols in the sequence are determined at a latency of L stages.

Real-time Implementation

Given the basics of the VA in the previous section, we now describe how the VA can be implemented in real time. Since the implementation can vary significantly according to the given trellis description and the application, we focus on the general ways of implementing each step in the VA described previously. We will use convolutional codes and trellis codes as examples to illustrate how to estimate the amount of computation and memory required to implement the VA for a given trellis and application.

Error-control codes are introduced to preserve the integrity of communication systems. Trellis codes and convolutional codes are examples of one generalized and useful type of coding called sequential coding. In particular, trellis codes, which were introduced about ten years ago, can be very powerful, bringing system performance to near theoretical limits in AWGN channels [32-33].

The input/output relations of these sequential codes are specified by finite-state diagrams or trellises. That is, given a trellis, an input sequence of symbols to be encoded, and an initial state, one can derive an output sequence of coded symbols (called *codewords*) based on the input/output relations of the transitions in the trellis. This sequence of codewords can be corrupted by AWGN in a communication channel. To decode the sequences of codewords that are corrupted by AWGN, the VA, first proposed in 1967 as a method of decoding convolutional codes [1], is the optimum detection method for these sequential codes on AWGN channels [1-6].

Step 1: Branch Metric Generation

For optimal VA performance in AWGN, the branch metric of a given transition is defined as the squared distance between the noisy symbol and the output symbol of the transition. That is, from Eq. 1,

$$B_{i,j,n} = (y_n - C_{i,j})^2 = y_n^2 - 2y_n C_{i,j} + (C_{i,j})^2 \qquad (3)$$

In the VA, comparison between the path metrics are required to determine the survivor path. Thus, one is only concerned with the differences between the path metrics. That is, one can arbitrarily add a constant to all the path metrics without changing the comparison result. Since the path metrics are made up of the accumulated branch metrics (Eq. 2), we can focus only on the branch metrics for our discussion. Since all the branch metrics have the term y_n^2 (Eq. 3), we can subtract

y_n^2 from all the branch metrics without changing the differences among them. That is, one can represent the branch metrics by:

$$B_{i,j,n} = -2y_n C_{i,j} + (C_{i,j})^2 \tag{4}$$

without changing the comparison result. C_{ij} is defined in the given trellis, and thus $C_{i,j}^2$ can be precomputed. Therefore, one needs addition and multiplication operations and/or table-lookup operations to compute the branch metrics.

Note that Eq. 4 may be simplified further,depending on the nature of the transition output symbols $C_{i,j}$. For example, consider the following two cases:

Case I: The transition output symbols are binary and are represented by -a and a. That is, $C_{i,j} \in \{-a, a\}$. In this case, we can further simplify the branch metric. Since

$$B_{i,j,n} - B_{k,j,n} = \begin{cases} 0 & for\ C_{i,j} = C_{k,j} \\ -2y_n C_{i,j} + 2y_n C_{k,j} & for\ C_{i,j} \neq C_{k,j} \end{cases}$$

dividing all branch metrics by 2 will not change the comparison results, we can represent the branch metrics as

$$B_{i,j,n} = -y_n C_{i,j} = \begin{cases} -y_n & for\ C_{i,j} = a \\ y_n & for\ C_{i,j} = -a \end{cases} \tag{5}$$

In this case, only negation operations are required to compute the branch metrics. This result can be applied to convolutional codes with anti-podal signaling and using soft-decision decoding discussed in the next section.

Case II: $C_{i,j} \in \{b, b+k\}$, where $k > 0$. In other words, the transition output symbols are either b or $b + k$, where k is the distance between the two symbols. Similar to case I, we can simplify the branch metric expression. The difference of the branch metrics is

$$B_{i,j,n} - B_{k,j,n} = \begin{cases} 0 & for\ C_{i,j} = C_{k,j} \\ 2y_n k - 2bk - k^2 & for\ C_{i,j}=b\ and\ C_{k,j}=(b+k) \\ -2y_n k + 2bk + k^2 & for\ C_{i,j} = (b+k)\ and\ C_{k,j}=b \end{cases}$$

Even though dividing all branch metrics by k, a positive constant, will change the differences of the branch metrics, it will not change the comparison results. Thus, we can denote the new difference, $B'_{i,j,n} - B'_{k,j,n}$.

$$B'_{i,j,n} - B'_{k,j,n} = \begin{cases} 0 & for\ C_{i,j} = C_{k,j} \\ 2y_n - 2b - k & for\ C_{i,j} = b\ and\ C_{k,j} = (b+k) \\ -2y_n + 2b + k & for\ C_{i,j} = b+k\ and\ C_{k,j} = b \end{cases}$$

Equivalently,

$$B'_{i,j,n} - B'_{k,j,n} = \begin{cases} 0 & for\ C_{i,j} = C_{k,j} \\ (y_n - b) - ((b+k) - y_n) & for\ C_{i,j} = b\ and\ C_{k,j} = (b+k) \\ ((b+k) - y_n) - y_n - b) & for\ C_{i,j} = (b+k)\ and\ C_{k,j} = b \end{cases}$$

Note that, in this case, the branch metric can be represented as the distance between the noisy symbol and the output symbol of the given transition. Thus,

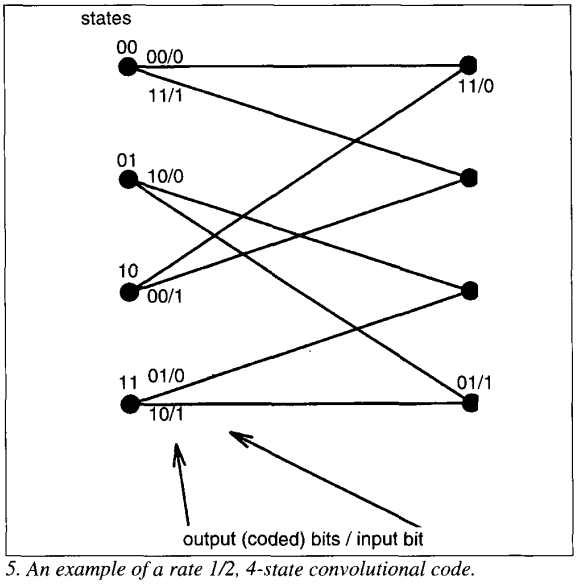
$$B_{i,j,n} = \begin{cases} y_n - b & for\ C_{i,j} = b \\ (b+k) - y_n & for\ C_{i,j} = b+k \end{cases} \tag{6}$$

That is, only addition operations or table-lookup operations are required to compute the branch metrics. This property can be applied to computing the branch metrics of some trellis codes when their branch metrics can be computed one dimension at a time and the nearest codewords in each given dimension are either b or $(b + k)$ (the value of b can be different in different dimensions).

Computing the branch metrics can vary significantly, according to the given trellis and the application. The following section explains how the branch metrics can be computed for convolutional codes and trellis codes.

Convolutional Codes

Convolutional codes are described by trellises. For the most commonly used convolutional codes, the inputs and outputs are defined in binary digits (or bits). For a *rate k/n, N-state convolutional code*, there are N-states in the trellis and 2^k transitions exiting and entering each state. For each transition, there are k input bits and n output (coded) bits (Fig. 5). If the n coded bit sequence is sent into an AWGN channel, each transmitted bit can be corrupted by the noise in the channel. The received waveform representing each transmitted bit at the receiver is quantized into an m-bit symbol, where



5. An example of a rate 1/2, 4-state convolutional code.

m is the quantization precision at the receiver. These quantized noisy symbols are then sent intothe Viterbi decoder.In the Viterbi decoder, one can use either Hamming distance (*hard decision decoding*) or Euclidean distance (*soft decision decoding*) for branch metric computation. Hard and soft decision decoding are described in the following sections.

Hard Decision Decoding

For hard decision decoding, each noisy symbol in the received sequence of noisy symbols is quantized to one bit before it is compared to the corresponding coded bit in the coded bit sequence of the given transition. The number of bits in which the two sequences differ iscalled the Hamming distance, and it is used as the branch metric for that transition. For example, if the received sequence of noisy symbols {0.05, 0.4, 0.8} is quantized to the bit sequence {0, 0, 1}, and if the coded bit sequence of the given transitionis {0, 1, 1}, then the Hamming distance is 1. The Hamming distance can easily be calculated using exclusive-or type of operations. If the processor has a precision more than the number of bits in each coded bit sequence, one exclusive-or operation is required to generate the Hamming distance for each transition in the trellis. Thus, if there are m distinct coded bit sequences in the trellis, m exclusive-or operations at most are required to generate the branch metrics of all the transitions and m memory locations are required to store these metrics.

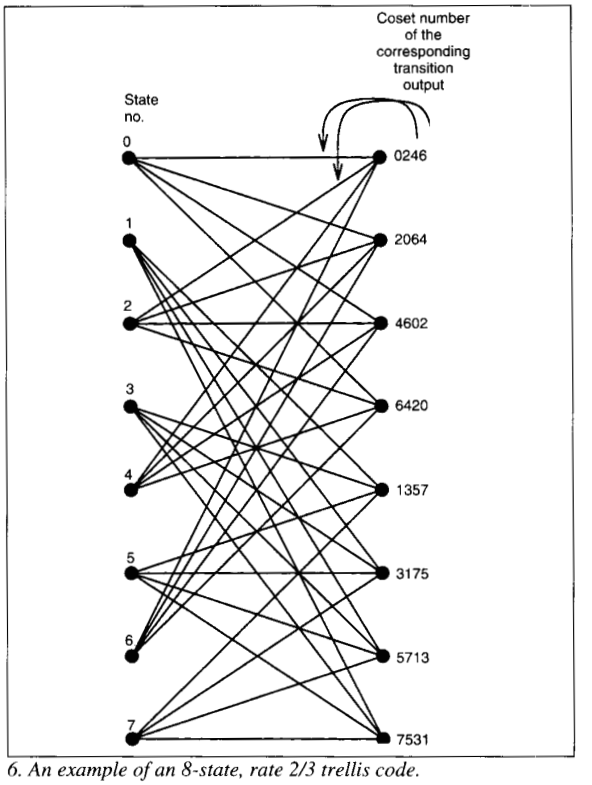
Soft Decision Decoding

In hard decision decoding, the noisy symbols are quantized to bits before the Hamming distances are computed. This may result in some loss in performance in the Viterbi decoder. To keep the information contained in the received noisy symbols, soft decision decoding computes the branch metrics by using the noisy symbols directly instead of quantizing them into bits. For each noisy symbol in the sequence, it calculates the squared distance between each noisy symbol and the corresponding coded bit in the coded bit sequence of the given transition, and then adds these distances together to form the branch metric of that transition. That is, if the given noisy symbol is x and the coded bits are represented by a and b, then the distance to coded bits are $(x - a)^2$ and $(x - b)^2$, respectively. For a commonly used rate k/n convolutional code with binary output symbols represented by, say -a and a, one can use negation operations to compute the metric for each bit as discussed in Case I above. Since there are n bits in the output (or coded) bit sequence of each transition,n noisy symbols are received at the input of theViterbi decoder at each recursion. Thus, n negation operations are required to find the metric of all the received symbols and (n - 1) additions are required to compute the branch metric for a given coded bit sequence. Therefore, if m is the number of distinct coded bit sequences in the given trellis, at most M(n - 1) additions and n negations are required to compute the branch metrics for all the transitions and m memory locationsare required to store these branch metrics.

Trellis Codes

For an n-dimensional trellis code, each code word has n-dimensions. That is, each codeword has n one-dimensional symbols. In trellis codes, the output for each transition in the trellis is a number denoting a subset (or called a *coset*) (Fig. 6, [5, 34-35]. Each coset usually consists of more than one codeword. To compute the branch metric of a given transition, one should first find the codeword in the coset associated with the given transition that is closest in distance to the received noisy symbol. The squared distance between this codeword and the received noisy symbol is the branch metric of the transition. Once the branch metrics are computed for all the transitions, the same procedures described above can be used to update the survivor path entering each state.

The way to compute the nearest codeword in a given coset is depends on the configuration of the codewords in the coset. For most practical applications, one can compute the branch metric for a higher dimensional symbol by computing the squared distances one or two dimensions at a time, and then add these distances to form the branch metrics of a higher dimensional symbol. For example, if the received noisy symbol has four dimensions, it can be represented by four one-dimensional symbols, (w, x, y, z). In many applications, one can take w, x, y and z independently and compute the squared distances to their respective nearest one-dimensional codeword in the given coset. After that, these squared distances are added together to form the branch metrics of the four-dimensional symbol. Note that depending on the arrangement of the codewords in a given coset, one might be able to use



6. An example of an 8-state, rate 2/3 trellis code.

absolute linear distances, rather than squared distances, as branch metrics (see Case II above).

In some trellis codes, the codewords in a higher dimensional coset are made up of unions and cross products of codewords in lower dimensional cosets. Thus, to find the nearest codeword of a higher dimensional noisy symbol, one can first find the closest codeword in each coset by finding the closest codewords in the respective lower dimensional cosets. After that, comparisons can be made to find the codeword, among the closest codewords, that is closest in distance to the received noisy symbol. That is, one may require the add-compare-select operations, described in the next section, to compute the branch metrics.

The operations required to compute the branch metrics of a trellis code vary significantly, depending on the structure of the cosets in the code. In general, one may use squaring operations or table-lookup operations to find the squared distances. Additions and add-compare-select operations may also be required.

Step 2: Survivor Path Update

To find the survivor path entering each state, the branch metric of a given transition is added to its corresponding previous path metric. This sum is compared to all the other sums corresponding to all the other transitions entering that state. The transition that has the minimum sum is chosen to be the survivor path (Eq. 2). That is, if we denote *mem*[*k*] as location *k* of memory *mem, branch* as the memory segment that stores the branch metrics, and *path* as the memory segment that stores the path metrics, we can perform the following operations to find the survivor path for a given state:

To initialize the metric:

min = branch[i] + path [j];

To find the path with the minimum path metric:

if ((branch[k] + path[l]) min)

min = branch[k] + path[l];

if ((branch[m] + path[n])

min = branch[m] + path[n];

.

.

.

We can define a new type of operation called an *add-compare-select* operation with the instruction:

acsmin(a, b, min), index

to perform the following operations:

if ((a + b) min){

min = a + b;

index → predetermined memory location;}

where *index* is used as an index to update the path with the minimum metric (survivor path). The implementation of the index is discussed later.

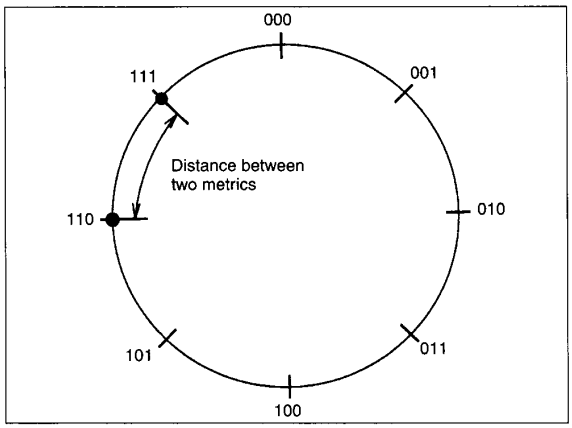
Given a trellis with a total of *M* transitionsand *N* states, a maximum of (*M* - *N*) *acsmin* operations are required to perform the comparison and *N* sums are required to initialize the metric for each state. The number of memory accesses required to store the indexes for the survivor paths is dependent on the way the survivor path memory is implemented. The survivor path memory is discussed in the following section.

The other real-time implementation issue for the survivor path update is the possibility of overflows in the registers that hold the path metrics, since the registers are of finite lengths. In order to ensure that the path metric registers will not overflow, the conventional approach is to renormalize the path metrics by subtracting a constant from all the path metrics, to be compared from time to time. This renormalization will not affect the comparison results, since the differences of the path metrics are not affected by subtracting a constant from all the path metrics to be compared. One can use the path metric with the minimum value, compared to all the path metrics at the current recursion, as the constant to be subtracted. With this approach, additional comparison and subtraction operations are required to renormalize the path metrics.

We can avoid the extra computation required to renormalize the path metrics by designing the registers to store the path metrics to be of length greater than $2D_{max}$, where D_{max} is the maximum possible difference between path metrics [36]. In this case, we can use the comparison rule described below and determine which path metric is smaller or larger, without worrying about overflows in the path metric registers. We can use the modulo arithmetic concept to explain the effects of overflows in a register that stores a path metric. We can imagine that due to the finite precision of the register, a path metric is basically running on a circle in the clockwise direction every time a positive number is added to it (Fig. 7 is an example where the register length is 3 bits). Thus, if we can make sure that this circle is large enough so that the distance between the two path metrics to be compared on the circle is always less than half the length of the circumference of the circle, then we know that the metric that is farther along in the clockwise direction has a larger value.

The way to determine which metric is larger or smaller is described as follows:

Let $\overline{m}_1 = (m_{1n},...m_{10})$ and $\overline{m}_2 = (m_{2n},...m_{20})$ be the two metrics to be compared, and $\overline{d} = (d_n,...d_0) = (\overline{m}_1 - \overline{m}_2)$ using



7. Finite precision effects of path metrics.

2's complement arithmetic. If $Z((\overline{m}_1,\overline{m}_2)$ denotes the logical result of comparing the path metrics $\overline{m}_1$ and $\overline{m}_2$, then

$$Z(\overline{m}_1,\overline{m}_2)=\begin{cases}1, & \text{if } \overline{m}_1<\overline{m}_2\\ 0, & \text{otherwise}\end{cases}$$

and $Z((\overline{m}_1,\overline{m}_2)=d_n$

Step 3: Optimum Path Trace Back

At each recursion or each decoding stage, the survivor path for each state is updated. In practice, the survivor path length, L, is the delay required for the survivor paths for all states to converge with very high probability. Thus, after L recursions, one can trace back from any arbitrary state and find the most likely transition at $(L - 1)$ previous stages.

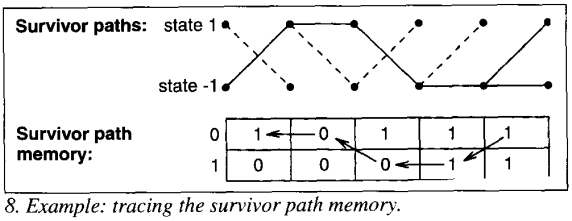
There are many ways to store and trace the survivor paths. For example, [39-41] provides ways to implement the survivor path memories for high-speed Viterbi decoding using multiple processors. This section focuses on two general and commonly used methods to store the survivor paths.

Method I stores the symbols associated with a survivor path directly.That is, at each recursion, the symbols associated with the complete survivor path for each state are updated. For the Viterbi detection example discussed earlier, the survivor path memory for state 1 at recursion 4, is [1, -1, 1, 1], where the left most symbol is the most recently stored data. At recursion 5, the survivor path for state 1 becomes [1, -1, -1, 1, 1].

The disadvantage of this scheme is that the complete survivor path for each state has to be updated at each recursion. Thus, the number of memory accesses required to update the survivor paths may be large. However, for convolutional codes that use bits as input symbols, one may pack many bits in the survivor path together to form a word and thus the number of memory accesses required to update the survivor paths may be reduced significantly. The advantage of this method is that no trace back is required to find the most likely symbol, because it can be read directly from the survivor path memory of a chosen state.

Method II is a more commonly used method that uses pointers to keep track of the survivor paths. A straightforward implementation of the survivor path memory can be implemented by dividing the memory into several blocks (Fig. 8). The number of blocks is dependent on how long one wants to keep the survivors. That is, the number is dependent on the survivor path length, L. Each block in the survivor path memory corresponds to one stage of a trellis or one recursion. Each location in each block stores the survivor path pointer for each state. The survivor path pointers are implemented as offsets from location 0 of each block. The offset gives the location that contains the pointer to the survivor path at the previous stage.

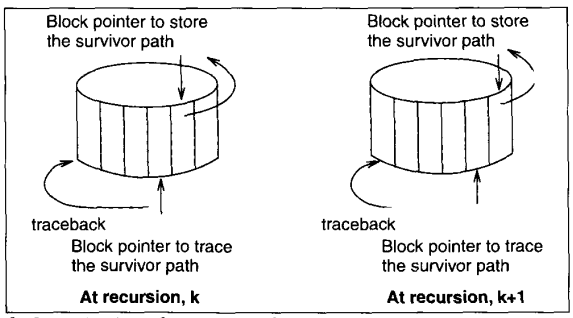
To trace the survivor path memory, a block counter is used to keep track of the current block of data to be read. To trace one stage of the survivor path, the block counter is decremented and concatenated with the offset read from memory to form the address of the memory location that contains the offset, pointing to the survivor path at the previous stage. Figure 8 illustrates the organization of the survivor path memory. The survivor path memory in this example contains two locations in each block to store the survivor paths for state 1 and state -1. Location 0 of each block stores the survivor path pointer for state 1, and location 1 of each block stores the survivor path pointer for state -1. Therefore, if we want to trace the survivor path from state 1 of the most recently stored block, we can set the pointer to read location 0 of this block. Since the content in location 0 has a value of 1, after decrementing the block counter and concatenating with the offset value of 1, we read location 1 of the previous block. The content in location 1 is 0 and thus the content in location 0 of the previous block will be read. In this way, one can trace the most-likely path.



8. Example: tracing the survivor path memory.

After pointing to the last block in the survivor path memory, the block counter overflows and the overflow bit is ignored to ensure automatic wraparound. Thus, we can imagine the survivor path memory to be a circular block of memory, as shown in Fig. 9. The block counter resets to location 0 of the survivor path memory when the limit of the survivor path memory is reached. If the survivor path length is L, there should be at least $(L + 1)$ blocks in the survivor path memory. The survivor path updates can be implemented in a first-in first-out manner. The oldest block of data is overwritten continuously in the survivor path memory. The block counter to trace the survivor path, on the other hand, starts tracing the survivor path from this same block of most recently stored data at the next period, but tracing back in the opposite direction from the block pointer that stores the updated survivor path pointers.

The advantage of using this scheme is that one needs to update only the survivor path pointers of the current recursion, as compared to Method I, where the symbols associated with the complete survivor paths of all states are to be updated. Using this implementation method, the survivor path memory requires $(L + 1)N$ locations and $\log_2(N)$, bits for



9. Organization of survivor path memory.

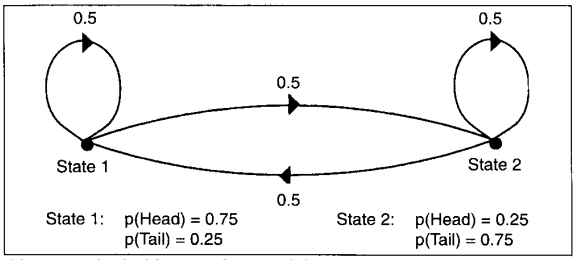
each location, where n is the total number of states in the given trellis. Further, L memory-indexing operations are requiredto trace the survivor path memory and n memory accessesare required to update the survivor path pointers for all states. For a rate $1/n$ convolutional code, L is about $5(\log_2(N)$ [4] or longer, depending on the application..

Viterbi Algorithm Applied to HMMs

In speech and character recognition, hidden Markov models (HMMs) can be used to model spoken words or written characters [7-9]. HMMs are similar to the finite state diagrams or trellises of convolutional codes discussed previously. However, the states in an HMM are "hidden" because each state is associated with a set of discrete symbols, with an *observation probability* assigned to each symbol, or is associated with a set of continuous observation with a continuous observation probability density. Each transition in the state diagram of an HMM also has a *transition probability* associatedwith it. Figure 10 is an example of an HMM where the transition probabilities for all transitions are 0.5 and the observation symbols are Head and Tail. For state 1, the observation probabilities for Head and Tail are 0.75, and 0.25, respectively.

Given an observation sequence representing a word or a character, the VA can be used to find the most likely state sequence and the accumulated probability (called the *likelihood score*) associated with this sequence in a given HMM. During training, when an HMM is being designed for the given observation sequence, the probability measures the likelihood an HMM matches the observation sequence. After the HMMs representing, for example, a set of words or characters are defined, the VA can also be used during the recognition process to determine the HMM within the set of HMMs that best matches with a given observation sequence. That is, the VA, given an observation sequence, computes the likelihood score of the most likely state sequence in each HMM and selects the HMM with the highest likelihood score to be the HMM (corresponds to a given work or character) that best matches with this observation sequence.

This section addresses the basics of the VA applied, to find the state sequence that best matches with the given observation sequence in a given one-dimensional HMM used in speech and character recognition. VA applied to two-dimensional HMMs, for example, and off-line (not real-time) character recognition [12] can readily be generalized once the basics of the one-dimensional case is mastered.



10. Example, hidden Markov model (HMM).

Applying the VA to HMMs

Given an observation sequence, the VA finds the most likely state sequence in a given HMM and the likelihood associated with this most likely sequence. At the beginning of each search, each state in a given HMM has some pre-defined initial likelihood score. To find the most-likely transition and update the state likelihood score for each state at the next time instant, or the next stage of the trellis, one must find the most likely transition coming into a given state. This is done by adding the transition probabilities of all the transitions coming into this given state to their corresponding previous state likelihood scores, and selecting the transition with the maximum sum to be the the most likely transition coming into this state. This sum is then added to the observation probability assigned by the given state, to the current observation symbol in order to form the updated state likelihood score for this given state. The most likely transitions and the state likelihood scores are updated for all states at each recursion. This process is performed recursively until all symbols in the given observation sequence is processed. At the end of the recursion, the path associated with the state that has the highest likelihood score is selected as the most likely state sequence for the given observation sequence. That is, if the observation sequence is = $(y_1, y_2,..., y_k)$, then for state j at recursion n, one wants to compute

$$M_{j,n} = \max_{\{i\}} [M_{i,n-1} + T_{i,j}] + O_{j,n} \tag{7}$$

where $O_{j,n}$ is the observation probability for y_n assigned by state j at recursion n, $T_{i,j}$ is the transition probability of the transition from state i to state j, $M_{j,n}$ is the state likelihood score for state j at recursion n, and $\{i\}$ is the set of states that have transitions to state j.

In recognition tasks, one is often only concerned with the likelihood score of the most likely state sequence rather than the state sequence itself. (In this case, the forward algorithm [8] is also often used.) For some applications, such as during training when the HMM is being defined, one would also be interested to know the most likely state sequence itself. Thus, the most likely transition for each state should be registered at each recursion so that the most likely state sequence can be traced at the end of the recursion.

A VA Example

Given the HMM in Fig. 10, the observation sequence = (Head, Tail), and the initial state likelihood scores for state 1 and 2 to be 0.5 and 0 respectively, one can compute the most likely state sequence in the HMM recursively. At recursion 1, one can first calculate the observation probabilities for state 1 ($O_{1,1}$) = p(Head) = 0.25) and for state 2 ($O_{2,1}$ = p(Head) = 0.25). After that, one can compute the updated state likelihood scores for state 1 and 2. For state 1

$$M_{1,1} = \max_{\{i\}} [M_{1,0} + T_{i,1}] + O_{1,1}$$

Since

$$([M_{1,0} + T_{1,1}] = 0.5 + 0.5) > ([M_{2,0} + T_{2,1}] = 0 + 0.5)$$
$$M_{1,1} = [M_{1,0} + T_{1,1}] + O_{1,1} = 1.75$$

Similarly, one can compute $M_{2,1} = 1.25$, $M_{1,2} = 2.50$, and $M_{2,2} = 3.0$.

In summary, at each stage in the recursion for each symbol in the observation sequence, there are two steps:

Step 1: Computation of observation probabilities: Compute the observation probability of the current symbol for each state by finding the probability assigned to the observation symbol for that state as defined by the given HMM.

Step 2: Update state likelihood scores: Update the most-likely path entering each state and update its state likelihood score. That is, for a given state, compare the sums of the transition probabilities of all transitions entering this state to their corresponding previous state likelihood scores, and select the transition with the maximum accumulated likelihood score to be the most likely transition. This maximum value is added to the observation probability computed in step 1 for this given state in order to form the updated state likelihood score for this state. If one has to find the most likely state sequence itself, the most likely transition to this state should also be registered .

At the end of the recursion, the path that has the highest likelihood score is selected to be the most likely path or state sequence. In some applications, only the likelihood score is of importance. However, for some other applications, the trace back is also necessary to find the most likely state sequence.

Real-time Implementation

Real-time implementation of VA applied to HMMs is similar to that of Viterbi detection discussed previously. The differences between the trellises of codes/channels used in Viterbi detection and the HMMs are that the symbols in the trellis of a code/channel are associated with the transitions, and the symbols in an HMM are associated with the states.

In Viterbi detection, the received symbols are noisy and the VA finds the sequence of transition output symbols that is closest in Euclidean distance to the received sequence of noisy symbols. In HMMs, each state has a set of symbols associated with it and each symbol in the set is assigned an observation probability. The transition probability for each transition is pre-defined in a given HMM. Given a sequence of observation symbols, the VA finds the observation probability assigned to each symbol by each state, and then adds these probabilities to the corresponding transition probabilities in order to compute the most likely state sequence and its likelihood score in the given HMM. Thus, Euclidean distances are used in Viterbi detection, and likelihood scores are used in HMMs. Further, in Viterbi detection, the path with the *minimum* accumulated metric, measured in distances, is computed and in HMM, the path with the *maximum* metric, measured in probability, is computed.

Computing Observation Probabilities

In HMMs,each state in the model is associated with a set of discrete symbols, with an observation probability assigned to each symbol, or a set of continuous observations with a continuous observation probability density. Given a discrete observation symbol, table-lookups can be used to find the observation probability for that symbol for each state. For continuous probability densities, computations are required to find the observation probabilities. For an HMM with N states, a total of N table-lookups or computations are required to compute the observation probabilities for the N states. In Viterbi detection, the states are not hidden, and there are no sets of symbols associated with the states. Thus, no computation is required to compute the observation probabilities.

However, each transition in the trellis of a code/channel has one or more symbols associated with it. The branch metric for that transition is the squared Euclidean distance between the received noisy symbol and the nearest symbol in the set of possible symbols. On the other hand, HMMs have no symbols associated with each transition, and the transition probability for each transition is pre-defined in a given HMM. Therefore, the VA applied to HMMs requires computation of observation probabilities for all states, and Viterbi detection requires computing the branch metrics for all transitions.

Updating State Likelihood Scores

Similar to Viterbi detection, add-compare-select operations can be used to add the transition probabilities to their corresponding previous state likelihood scores and select the path with the maximum accumulated probability to be the most likely path coming into this state. This accumulated probability is added to the observation probability of the given state in order to form its updated state likelihood score. Contrary to the add-compare-select operations used in Viterbi detection, where the minimum sum is selected, the transition with the maximum sum is selected. Thus, we define a new add-compare-select operation that finds the maximum value:

acsmax(a, b, max), index

to perform the following operations:

if ((a + b) > max {
max = a + b;
index → predetermined memory location; }

where *index* is used as an index to update the transition with the maximum likelihood score.

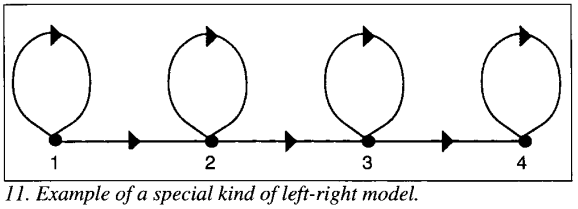
In addition to the *addmax* operations, another add operation is required to add the accumulated probability to the observation probability of that state. Thus, VA used for HMM requires an additional add operation at the end of the search to update the state likelihood score for the given state. Therefore, using our definition for *acsmax*, if there are m transitions coming into a given state, $(m - 1)$ *acsmax* operations and 2 add operations (one to initialize the maximum, and one to add the accumulated probability to the observation probability) are required to update the most likely path and the state likelihood score for each state.

Tracing of the Most Likely State Sequence

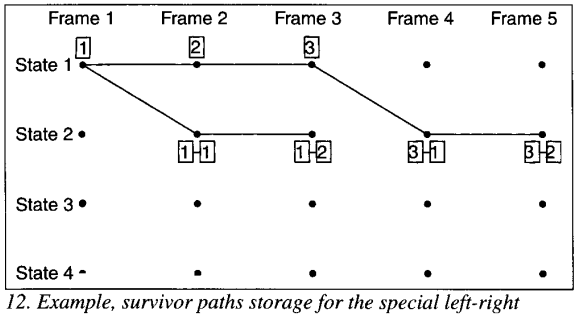
In Viterbi detection, the most likely path is to be computed, and the trace back can be performed continuously after an initial delay or latency. For the VA applied to HMMs, the best state sequence may not be of interest. In many cases, only the likelihood score associated the most likely state sequence is required. However, if the most likely state sequence is of interest, the trace back is performed at the end of the recursion. The trace back of the most likely state sequence can be implemented using previous state pointers similar to Viterbi detection. However, a large memory space may be required to store the survivor paths for all states if the observation sequence is long. For example, if there are 128 states in an HMM and 1000 symbols in the observation sequence, 1000 stages in the recursion are required and 128,000 memory locations are required to store the survivor path pointers for all states. Fortunately, in many applications, one can explore the structure of the HMMs and reduce the memory required to store the survivor paths for all states.

For example, in speech recognition and on-line character recognition, a special type of model known as *left-right models* [7, 9] is often used.The state transitions in these models are connected either to the state it originated from, or to the next state (Fig. 11). With this special structure, one can keep track ofthe survivor path for each state by counting the number of times the path stayed in the same state, or if the survivor path comes from the previous state, inheriting the path counts from that previous state.

An example of the implementation of this scheme is given in Fig 12. In the figure, the numbers given in the boxes indicate the number of times the survivor paths stayed in the same state. For example, at frame 4, the survivor path for state 2 came from state 1. Thus, state 2 inherits the survivor path count of state 1. At frame 5, the survivor path for state 2 came from the same state, state 2. Thus, the survivor path count for state 2 is incremented by 1.



11. Example of a special kind of left-right model.



12. Example, survivor paths storage for the special left-right model in Fig. 11.

Summary

We have described the basics of the VA that is widely used in digital communication systems to detect sequential error-control codes and signals in channels with memory. Given a sequence of noisy symbols, the VA computes the most likely sequence in the given trellis recursively. At each recursion, there are three major steps in Viterbi detection: generation of branch metrics, updating the survivor paths and path metrics, and tracing the most likely path. The general methods of implementing these steps in real-time were also described, and the implementation of the VA for decoding convolutional codes and trellis codes were explained. In general, one may need addition, multiplication and/or table look-up operations to generate the branch metrics. Addition and add-compare-select operations are required to find the survivor paths, and memory referencing operations are required to trace the most likely path. Depending on how the survivor path memory is implemented, one might also need indexing operations to access the memory.

We have also described the way the VA can be applied to hidden Markov models used in speech and character recognition. Given an observation sequence representing a word or a character, the VA can be used to find the most likely state sequence and the likelihood score of this sequence in a given HMM. This operation is performed recursively, and at each recursion, there are two major steps: computing the observation probabilities, and updating the state likelihood scores for all states. If one wants to know the most likely state sequence, in addition to its likelihood score, one also needs to trace this most likely sequence at the end of the recursion. Real-time implementation of the VA applied to HMMs, and the differences between the VA used in digital communication systems and the VA used for HMMs, were also discussed. In general, table-lookup operations can be used to compute the observation probabilities if the observation symbols are discrete. Addition and add-compare-select operations are used to update the survivor paths and the state likelihood scores for all states. Memory referencing operations are required to trace the most likely state sequence.

Hui-Ling Lou is a Member of Technical Staff with the Signal Processing Research Department at AT&T Bell Laboratories, Murray Hill, NJ.

References

1. A. J. Viterbi. "Error Bounds for Convolutional Codes and an Asymptotically Optimum Decoding Algorithm," *IEEE Trans. on Information Theory*, IT-13, April 1967.

2. G. D. Forney Jr. "Maximum-Likelihood Sequence Detection in the Presence of Intersymbol Interference," *IEEE Trans. on Information Theory*, IT-18(3):363-378, May 1972.

3. G. D. Forney Jr. "The Viterbi Algorithm," *IEEE Proceedings*, IT-61(3):268-278, March 1973.

4. A. J. Viterbi and J. K. Omura. *Principles of Digital Communication and Coding*. McGraw-Hill Book Company, New York, NY, 1979.

5. E. A. Lee and D. G. Messerschmitt. *Digital Communication*. Kluwer Academic Publishers, Boston, MA. 1988.

6. J. M. Wozencraft and I. M. Jacobs. *Principles of Communication Engineering*. John Wiley and Sons, New York, NY, 1965.

7. L. R. Rabiner and B-H Juang. "An Introduction to Hidden Markov Models," *IEEE ASSP Magazine*, 3(1):4-16, January 1986.

8. L. R. Rabiner. "A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition." *IEEE Proceedings*, 77(2): 257-285, February 1989.

9. L. R. Rabiner and B-H Juang. *Fundamentals of Speech Recognition*. Prentice Hall, Englewood Cliffs, New Jersey, 1993. Chapter 6.

10. J. Picone. "Continuous Speech Recognition Using Hidden Markov Models," *IEEE ASSP Magazine*, 7(3):26-41, July 1990.

11. N. Seshadri and C-E. W. Sundberg. "List Viterbi Algorithms with applications." *IEEE Transactions on Communications*., Vol. 42, No. 2/3/4, Feb/Mar/Apr 1994.

12. O. E. Agazzi and S. Kuo. "Pseudo two-dimensional hidden Markov models for document recogntion." *AT&T Technical Journal*, 72(5), September/October 1993.

13. C. B. Shung, H. D. Lin, R. Cypher, P. H. Siegel, and H. K. Thapar. " Area-efficient architectures for the Viterbi algorithm I. Theory." *IEEE Trans. on Communications*, 41(4):636-44, April 1993.

14. C. B. Shung, H. D. Lin, R. Cypher, P. H. Siegel, and H. K. Thapar. " Area-efficient architectures for the Viterbi algorithm II. Applications." *IEEE Trans. on Communications*, 41(5):802-7, May 1993.

15. C. B. Shung, H. D. Lin, P. H. Siegel, and H. K. Thapar. "Area-Efficient Architecture for the Viterbi Algorithm." In: 1990 *Global Communications Conference*, San Diego, December 1990.

16. H.-D. Lin and C. B. Shung. "General in-place scheduling for the Viterbi algorithm." In: 1991 *International Conference on Acoustics, Speech and Signal Processing*, Toronto, Canada, May 1991.

17. H.-D. Lin and D. G. Messerschmitt. "Algorithms and architectures for concurrent Viterbi decoding." In: 1989 *International Conference on Communications*, Boston, Massachusetts, June 1989.

18. K. K. Parhi. "High-speed VLSI architectures for Huffman and Viterbi decoders." *IEEE Trans. on Circuits and Systems* 11: Analog and Digital Signal Processing, 39(6):385-91, June 1992.

19. P. J. Black and T. H.-Y. Meng. "A 140MBIT/S 32-State Radix-4 Viterbi Decoder." In: 1992 *International Solid-State Circuits Conference*, San Francisco, February 1992. Paper No. WP 4.5.

20. P. J. Black and T. H.-Y. Meng. "A hardware efficient parallel Viterbi algorithm." In: 1990 *International Conference on Acoustics, Speech and Signal Processing*, New York, April 1990.

21. G. Fettweis and H. Meyr. "High-speed parallel Viterbi decoding: algorithm and VLSI-architecture." *IEEE Communications Magazine*, 29(5):46-55, May 1991.

22. G. Fettweis and H. Meyr. "High Rate Viterbi Processor: A Systolic Array Solution." *IEEE Journal on Selected Areas in Communications*, 8(8): 1520-1534, October 1990.

23. G. Fettweis and H. Meyr. "A 100MBIT/S Viterbi Decoder Chip: Novel Architecture and Its Realization." In: 1990 *International Conference on Communications*, Atlanta, April 1990. Paper No. 307.4.

24. G. Fettweis and H. Meyr. "Parallel Viterbi Algorithm Implementation: Breaking the ACS-Bottleneck." *IEEE Transactions on Communications*, 37(8):285-90, August 1989.

25. E. Paask, S. Pedersen, and J. Sparso. "An area-efficient pathmemory structure for VLSI Implementation of high speed Viterbi decoders." *Integration, The VSLI Journal*, 12(1):79-91, November 1991.

26. R. Cypher and C. B. Shung. "Generalized traceback techniques for survivor memory management in Viterbi algorithm." In: 1990 *Global Communications Conference*, December 1990.

27. W. R. Kirkland and D. P. Taylor. "High-speed Viterbi decoder memory design." *Canadian Journal of Electrical and Computer Engineering*, 15(3): 107-114, August 1990.

28. H. K. Thapar and J. M. Cioffi. "A block processing methods for designing high-speed Viterbi detectors." In: *International Conference on Communications*, Boston, June 1989.

29. R. Schweikert and A. J. Vinck. "Trellis-coded modulation with high-speed low complexity decoding." In: 1990 *Global Communications Conference*, San Diego, December 1990.

30. P. G. Gulak and T. Kailath. "Locally connected VLSI architectures for the Viterbi algorithm." *IEEE Journal on Selected Areas in Communications*. 6(3). April 1988.

31. C. M. Rader. "Memory Management in a Viterbi Decoder," *IEEE Trans. on Communications*, COM-29(9): 1399-1401, September 1981.

32. G. Ungerboeck. "Trellis-Coded Modulation with Redundant Signal Sets Part I," *IEEE Communications Magazine*, 25(2), February 1987.

33. G. Ungerboeck. "Trellis-Coded Modulation with Redundant Signal Sets Part II," *IEEE Communications Magazine*, 25(2). February 1987.

34. G. D. Forney Jr. "Coset Codes I: Geometry and Classification," *IEEE Trans. on Information Theory*, IT-34(5): 1123-1151, September 1988.

35. G. D. Forney Jr. "Coset Codes II: Binary Lattices and Related Codes," *IEEE Trans. on Information Theory*, IT-34(5): 1152-1187, September 1988.

36. C. B. Shung, P. H. Siegel, G. Ungerboeck, and H. K. Thapar. "VLSI Architectures for Metric Normalization in the Viterbi Algorithm." In: 1990 *International Conference on Communications*, Atlanta, Georgia, April 1990.

37. H. Lou. "The Study and Design of a Programmable Processor for Viterbi Derection." Stanford University, Ph.D. Thesis, Stanford, CA, 1993.

38. A. P. Hekstra. "An Alternative to Metric Rescaling in Viterbi Decoders," *IEEE Trans. on Communications*, 37(11): 1220-1222, November 1989.

39. O. Collins and F. Pollara. "Memory management in traceback Viterbi decoders." TDA Progress Report, Jet Propulsion Laboratory, November 1989.

40. G. Feygin and P. G. Gulak. "Survivor sequence memory management in Viterbi decoders." In: 1991 *International Symposium of Circuits and Systems*, Singapore, June 1991.

41. G. Feygin and P. G. Gulak. "Survivor sequence memory management in Viterbi decoders." Technical report, University of Toronto. Computer Research Institute, CSRI-252, 1991.