Zuerst ersch. in: IEEE Transactions on Communications ; 54 (2006), 3. - S. 453-461

Fast Tree-Trellis List Viterbi Decoding

Martin Röder, Raouf Hamzaoui

This paper was presented in part at ICME-02, IEEE International Conference on Multimedia and Expo, Lausanne, August 2002. The authors are with the Department of Computer and Information Science, University of Konstanz, 78457 Konstanz, Germany. Email: martin.roeder@uni-konstanz.de, raouf.hamzaoui@uni-konstanz.de.

Abstract

A list Viterbi algorithm (LVA) finds n most likely paths in a trellis diagram of a convolutional code. One of the most efficient LVAs is the tree-trellis algorithm of Soong and Huang. We propose a new implementation of this algorithm. Instead of storing the candidate paths in a single list sorted according to the metrics of the paths, we show that it is computationally more efficient to use several unsorted lists, where all paths of the same list have the same metric. For an arbitrary integer bit metric, both the time and space complexity of our implementation are linear in n. Experimental results for a binary symmetric channel and an additive white Gaussian noise channel show that our implementation is much faster than all previous LVAs.

Keywords. Viterbi decoding, Convolutional codes, Algorithms, Concatenated coding.

I. INTRODUCTION

Data transmission systems based on a concatenation of an outer error detecting code and an inner error correcting convolutional code have many important applications. For example, Sherwood and Zeger [1] devised a powerful system for protecting images against errors in a memoryless noisy channel by concatenating an outer cyclic redundancy-check (CRC) code for error detection and an inner rate-compatible punctured convolutional (RCPC) code for error correction. The image is first compressed with an embedded wavelet coder [2]. Then the output bitstream is divided into consecutive blocks of fixed length. Each block is encoded with the concatenated code and sent over the channel. The received blocks are decoded successively. For each block, a list Viterbi algorithm (LVA) is used to find a maximum-likelihood path. This path is checked with the CRC code. If an error is detected, the LVA is used again to find the next most likely path, which is also checked for errors. This process is repeated until the CRC test is passed, or a maximum number of paths is reached. In the latter case, called incomplete decoding, the current block and all the following ones are discarded, and the image is reconstructed using only the correctly decoded blocks.

The performance of such concatenated systems depends on the maximum number of candidate paths for a received word. Increasing the maximum number of candidates decreases the probability of incomplete decoding and improves the expected reconstruction fidelity. However, the number of candidates that can be tested is limited by the time complexity of the LVA.

A lot of work has been dedicated to the design of fast LVAs [3], [4], [5], [6]. One of the best LVAs is the tree-trellis algorithm (TTA) of Soong and Huang [3], which was initially proposed for speech recognition (see also [7] pp. 236–238). To find n most likely paths, this algorithm keeps candidate paths in a single list sorted according to the metrics of the paths. We show that it is better to use an array of unsorted lists where all paths of the same list have the same metric. To determine the array size, we find an upper bound on the difference between the largest and the smallest metric of n most likely paths. When the path metric is the Hamming metric, this bound is the Hamming weight of the convolutional codeword associated to an nth most likely path for the zero codeword.

The worst-case time complexity of our implementation is linear in n. This compares favorably with the original algorithm [3] whose time complexity is quadratic in n and with other state-of-the-art LVAs [4], [5], [6]. Moreover, we extend our bound to arbitrary integer bit metrics, which makes our algorithm also useful in soft-decision decoding. Experimental results for the binary symmetric channel (BSC) and the additive white Gaussian noise (AWGN) channel showed that our implementation was also the fastest in practice.

The rest of the paper is organized as follows. In Section II, we present our terminology and notations, discuss previous work, and describe an implementation of the TTA for binary convolutional codes. In Section III, we introduce our multiple-list TTA, which uses an array of unsorted lists instead of a single sorted list. In Section IV, we first show how to reduce the space complexity of the TTA and the multiple-list TTA. Then we explain how to use our multiple-list TTA with soft-decision decoding. In Section V, we compare the time complexity of all LVAs for a BSC and an AWGN channel.

II. Terminology and Previous Work

A. Terminology

A binary rate-$1/r$ convolutional encoder outputs r bits for each input bit. This group of r output bits is called an *output frame*. When an output frame is transmitted over the channel, bit errors may occur. The resulting block is called a *code frame*. The register circuit of the encoder consists of ν memory elements, giving 2^ν possible *states* i, $i = 0, \ldots, 2^\nu - 1$. When an input bit is read by the encoder, it is shifted into the register circuit, leading to a *state transition* $(i \rightarrow j)$. The output frame associated to state transition $(i \rightarrow j)$ is denoted by $R(i \rightarrow j)$. The node of the trellis diagram of the convolutional code corresponding to state j at stage t is denoted by (j, t). A *path* $p = p(0), p(1), \ldots, p(l)$ of length l is a sequence of states $p(i)$, $i = 0, \ldots, l$, generated by l successive input bits. We suppose that both the starting state and the final state are the zero state 0. The convolutional codeword associated to path p is the word $c(p)$ obtained by concatenating the l output frames $R(p(t-1) \rightarrow p(t)),\ t = 1, \ldots, l$. Let w be a received word consisting of l code frames $w(1), \ldots, w(l)$ and let M be a bit metric. Then, $M(c(p), w) = \sum_{t=1}^{l} M(R(p(t-1) \rightarrow p(t)), w(t))$ is the *path metric* of path p relative to w. The *partial path metric* of p from node $(p(k), k)$ to node $(p(k'), k')$ ($k' > k$) relative to w is the sum $\sum_{t=k+1}^{k'} M(R(p(t-1) \rightarrow p(t)), w(t))$.

The paths of length l can be ordered with nondecreasing path metric relative to w. We denote by $p_1(w), p_2(w), \ldots, p_n(w)$ the first n such paths. These n most likely paths are called n best paths for w. We denote the path metric of $p_i(w)$ relative to w by $M(p_i(w))$. Thus, $M(p_1(w)) \leq \ldots \leq M(p_n(w))$. An LVA finds n best paths for w.

When the bit metric M is the Hamming distance, we use the symbol H instead of M. Throughout the text, the zero codeword is denoted by $\mathbf{0}$.

B. Tree-Trellis Algorithm

The best previous LVAs are the parallel LVA (PLVA) [4], the serial LVA (SLVA) [4], which was improved in [5], the TTA [3], and the list extension algorithm (LEA) [6]. With the exception of the PLVA, which determines n best paths simultaneously, all other algorithms find n best paths successively, using the $k-1$ ($1 < k \leq n$) previously decoded paths to decode the kth one. This saves a lot of time because a kth best path is computed only when one is not satisfied with a $(k-1)$th best one. In the following, we give an implementation of the TTA for a binary rate-$1/r$ convolutional code. In the next section, we show how to improve this algorithm.

Suppose that a word w consisting of l code frames is received. To find n best paths for w, the TTA uses one forward pass and n backward passes in the trellis diagram of the convolutional code. In backward pass k ($1 \leq k \leq n$), a kth best path p_k is computed by traversing the trellis backwards. This path branches at stage $t(k)$ from a previously decoded path $p_{q(k)}$ ($q(k) < k$), that is, $t(k)$ is the smallest nonnegative integer such that for all stages t ($t(k) \leq t \leq l$), we have $p_k(t) = p_{q(k)}(t)$.

In the forward pass, for each node (j,t), we store two partial path metrics denoted by $M_1(j,t)$ and $M_2(j,t)$ with $M_1(j,t) \leq M_2(j,t)$. The metric $M_1(j,t)$ is the smallest partial path metric from node $(0,0)$ to node (j,t). It can be obtained as in the standard Viterbi algorithm with

$$M_1(j,t) = \min_i \{M_1(i,t-1) + M(R(i \to j), w(t))\} \text{ for } t > 0, \tag{1}$$

where we set

$$M_1(j,t) = \begin{cases} 0 & \text{if } t = j = 0; \\ +\infty & \text{if } t = 0 \text{ and } j > 0. \end{cases}$$

We denote by $v(j,t)$ the state that gives the minimum in (1) and by $\bar{v}(j,t)$ the other state. (For a binary rate-$1/r$ convolutional code, only two states can lead to state j). If the state that gives the minimum is not unique, the choice of $v(j,t)$ is arbitrary among the two possible states.

When the minimum is unique, any path with partial metric $M_1(j,t)$ from node $(0,0)$ to node (j,t) traverses $v(j,t)$ at stage $t-1$. For this reason we call $v(j,t)$ a backtrace pointer. For each node (j,t), we store $v(j,t)$. By changing a single bit in the binary representation of $v(j,t)$ one can easily obtain $\bar{v}(j,t)$.

The second partial path metric stored at node (j,t), $M_2(j,t)$, is defined as

$$M_2(j,t) = \begin{cases} +\infty & \text{if } t = 0; \\ M_1(\bar{v}(j,t), t-1) + M(R(\bar{v}(j,t) \to j), w(t)) & \text{if } t > 0. \end{cases}$$

It is the smallest partial path metric from node $(0,0)$ to node (j,t) via node $(\bar{v}(j,t), t-1)$.

The backward passes use a stack of elements $S_h = (S_h.m, S_h.q, S_h.t, S_h.u),\ h \geq 1$, where

- $S_h.m$ is a path metric,
- $S_h.q$ is the number of the backward pass at which S_h was inserted into the stack,
- $S_h.t$ is a stage,
- $S_h.u$ is a partial path metric from node $(p_{S_h.q}(S_h.t), S_h.t)$ to node $(0,l)$.

Each stack element S_h defines a path p that branches at stage $S_h.t$ from a previously decoded path $p_{S_h.q}$. This path p has path metric $S_h.m$ and is given by

$$p(t) = \begin{cases} p_{S_h.q}(t) & \text{for } t = S_h.t, \ldots, l; \\ \bar{v}(p(t+1), t+1) & \text{for } t = S_h.t - 1 \\ v(p(t+1), t+1) & \text{for } t = 0, \ldots, S_h.t - 2. \end{cases}$$

The stack is sorted by nondecreasing path metrics, that is, $S_h.m \leq S_{h+1}.m$. Also the index h of a stack element S_h is always equal to the rank of that element in the stack.

In backward pass k, a kth best path p_k is computed. This path is defined by S_1, the element at the top of the stack after the completion of backward pass $k-1$. At each stage t, $0 < t < S_1.t$, of backward pass k, an element defining a path that branches from p_k at stage t is inserted into the stack.

We now give a formal description of a backward pass.

1) In the first backward pass, intialize the stack as empty, set $k = 1$, $t = l$, $i = 0$, $m = 0$, $p_k(t) = 0$ and go to Step 6. In each subsequent backward pass, set $k = k + 1$ and go to Step 2.
2) Set $t = S_1.t$, $i = p_{S_1.q}(t)$, $m = S_1.u$, and $p_k(t) = p_{S_1.q}(t)$, $p_k(t+1) = p_{S_1.q}(t+1), \ldots, p_k(l) = p_{S_1.q}(l)$.
3) Remove the first element from the stack.
4) Set $j = \bar{v}(i,t)$.
5) Set $m = m + M_2(i,t) - M_1(j,t-1)$ and $p_k(t-1) = j$. Then, set $t = t - 1$ and $i = j$.
6) If $M_2(i,t) < \infty$, insert into the stack a new element S_h with $S_h.m = M_2(i,t) + m$, $S_h.q = k$, $S_h.t = t$, and $S_h.u = m$ at a position h such that the stack remains sorted by nondecreasing path metrics $S_h.m$.
7) Set $j = v(i,t)$.
8) Set $m = m + M_1(i,t) - M_1(j,t-1)$ and $p_k(t-1) = j$. Set $t = t - 1$ and $i = j$.
9) Repeat Steps 6 to 8 until $t = 0$.

Except for the stack insertions at Step 6, the first backward pass is the same as the backward pass of the standard Viterbi algorithm. In the next backward passes, after updating the current number of the backward pass (Step 1), we set the current position (i,t) in the trellis and the partial path metric m from the current position to the end of the trellis to the position and partial path metric stored in the first element of the stack (Step 2). This position is where p_k, the path to be decoded in the current backward pass, branches from $p_{S_1.q}$, a path decoded in a previous backward pass. Thus, both paths are equal from stage $S_1.t$ to stage l. In Step 3, we complete the initialization of the backward pass by removing the first element from the stack, which is no longer needed. At the first state transition, we follow $\bar{v}(i,t)$ (Step 4), update the partial path metric m accordingly, and store the new state of p_k (Step 5). In the next state transitions, we

insert a new element describing the path branching from the currently decoded path at the current stage into the stack (Step 6), follow the backtrace pointer $v(i,t)$ (Step 7), update the partial path metric m accordingly, and store the new state of p_k (Step 8). Since $M_2(i,t) \geq M_1(i,t)$ for all nodes (i,t), the metrics of the paths whose stack elements are inserted into the stack at Step 6 are always greater than or equal to the metric of the currently decoded path. This, together with the sorting of the stack by nondecreasing path metrics, ensures that we decode the paths in the desired order.

The following example illustrates the algorithm. We consider the convolutional code with generator polynomials (7,5) (octal), which has a code rate of 1/2 and memory order $\nu = 2$. Figure 1 shows the state diagram for this code. We explain how the TTA finds the best and second best path for the received word 11101001001100 when the bit metric is the Hamming distance. Figure 2 shows the trellis diagram of the convolutional code and the result of the forward pass.

We start the first backward pass by initializing the stack as empty and setting $k = 1$, $t = 7$, $i = 0$, $m = 0$, and $p_1(7) = 0$. We create a new stack element S_h such that $S_h.m = M_2(0,7) + m = 5$, $S_h.q = 1$, $S_h.t = 7$, and $S_h.u = 0$. Since the stack is empty, S_h is inserted at position $h = 1$. Next we set $j = v(0,7) = 0$, update m by setting $m = m + M_1(0,7) - M_1(0,6) = 0$ and set $p_1(6) = 0$. We then set $t = 6$ and $i = 0$. The rest of the first backward pass proceeds as follows. Create stack element S_h such that $S_h.m = 4$, $S_h.q = 1$, $S_h.t = 6$, and $S_h.u = 0$. Insert it at position $h = 1$ (note that the stack element inserted at stage 7 is now S_2). Set $j = 2$, $m = 0$, $p_1(5) = 2$, $t = 5$, $i = 2$. Create stack element S_h such that $S_h.m = 3$, $S_h.q = 1$, $S_h.t = 5$, and $S_h.u = 0$. Insert it at position $h = 1$. Set $j = 3$, $m = 1$, $p_1(4) = 3$, $t = 4$, $i = 3$. Create stack element S_h such that $S_h.m = 6$, $S_h.q = 1$, $S_h.t = 4$, and $S_h.u = 1$. Insert it at position $h = 4$. Set $j = 1$, $m = 1$, $p_1(3) = 1$, $t = 3$, $i = 1$. Create stack element S_h such that $S_h.m = 5$, $S_h.q = 1$, $S_h.t = 3$, and $S_h.u = 1$. Insert it at position $h = 4$. Set $j = 2$, $m = 2$, $p_1(2) = 2$, $t = 2$, $i = 2$. Since the partial path metrics $M_2(j,t)$ at the nodes of the remaining stages are all ∞, do not create new stack

elements for the remaining part of the first backward pass. Set $j = 1$, $m = 2$, $p_1(1) = 1$, $t = 1$, $i = 1$. Finally, set $j = 0$, $m = 2$, $p_1(0) = 0$, $t = 0$, and $i = 0$. The first backward pass is now complete and p_1 is the path $0, 1, 2, 1, 3, 2, 0, 0$ with corresponding codeword 11100001011100 and input bits 1011000. The metric of p_1 is $m = 2$. The stack contains the elements $(3, 1, 5, 0)$, $(4, 1, 6, 0)$, $(5, 1, 7, 0)$, $(5, 1, 3, 1)$, and $(6, 1, 4, 1)$ given as tuples $(S_h.m, S_h.q, S_h.t, S_h.u)$.

At the beginning of the second backward pass, we set $k = 2$, $t = 5$, $i = 2$, $m = 0$, $p_2(5) = 2$, $p_2(6) = p_1(6) = 0$, $p_2(7) = p_1(7) = 0$ and remove the first element from the stack. Note that now $S_1 = (4, 1, 6, 0, 0)$. We set $j = 1$, update m by $m = m + M_2(2, 5) - M_1(1, 4) = 1$, set $p_2(4) = 1$, $t = 4$ and $i = j = 1$. Next we proceed with Steps 6 to 8 as in the first backward pass. At the end of the second backward pass, p_2 is the path $0, 1, 2, 0, 1, 2, 0, 0$ with corresponding codeword 11101111101100 and input bits 1001000. The metric of p_2 is $m = 3$. The stack contains the six elements $(4, 1, 6, 0)$, $(4, 2, 4, 1)$, $(5, 1, 7, 0)$, $(5, 1, 3, 1)$, $(6, 1, 4, 1)$, $(6, 2, 3, 2)$.

To reduce memory requirements and avoid unnecessary stack insertions, we limit the stack size during backward pass k to $n - k$ because all elements inserted behind element $n - k$ never reach the top of the stack. Thus, at the end of Step 6, we delete the last element of the stack if the stack size becomes larger than $n - k$.

The time complexity of the forward pass is $O(2^\nu l)$. For the backward pass, if the stack is implemented as a linked list, inserting an element into the stack requires $n/2$ comparisons in average. If we assume that a new path branches from an existing path at half of the trellis in average, we have to visit $l/2$ states in each backward pass, which leads to $l/2$ stack insertions and $l/2$ state transitions per backward pass. All other steps of a backward pass can be done in amortized constant time. Thus, for n paths, the average time complexity of the backward passes is $O(l(n^2 + n)) = O(ln^2)$.

To reduce the time complexity of the TTA, one can maintain the sorted stack with a red-black tree [8], which guarantees a logarithmic time complexity for insertion operations and for deleting the first or the last element of the stack. This reduces the time complexity of the n backward

passes to $O(l(n + \log_2((n-1)!))) = O(ln \log_2 n)$.

We now study the space complexity of the TTA. In the forward pass, for each node (j, t), we store the two partial metrics $M_1(j, t)$ and $M_2(j, t)$ and the backtrace pointer $v(j, t)$. In each backward pass, we store for each stack element the value of $S_h.m, S_h.q, S_h.t$, and $S_h.u$, together with a constant number of pointers for the linked list (respectively the red-black tree). Moreover, each computed path is stored. This results in a space complexity of $O(2^\nu l+nl+n) = O(2^\nu l+nl)$.

III. MULTIPLE-LIST TTA

In this section, we propose a new implementation of the TTA, which we call multiple-list TTA (ml-TTA). The basic idea is to eliminate the costs of maintaining a sorted stack. This is done by replacing the single sorted list with an array of lists such that the paths associated to the elements of a list have the same metric. The elements of the same list do not have to be sorted, and an insertion is done by appending the new stack element into the list given by its metric, which is a constant time operation. One could try to allocate the lists dynamically at runtime, but this would require resizing the array of lists several times, and each resizing is a linear time operation since the array must be copied. A better way is to estimate in advance the number of needed lists and use an array of a fixed size. Then we can simply use the path metric as an index in that array. We now give a tight upper bound on the number of lists. We first assume that the bit metric M is the Hamming distance, which is denoted by H. In Section IV, we extend our approach to arbitrary bit metrics.

A. Bound on the number of lists

Suppose that l output frames of a binary rate-$1/r$ convolutional code with memory order ν are sent over the channel and let w be the received word. Since there are $H(p_n(w)) - H(p_1(w)) + 1$ integer values between $H(p_1(w))$ and $H(p_n(w))$, we need at most $H(p_n(w)) - H(p_1(w)) + 1$ lists. However, this bound depends on w. The following proposition gives a bound that is independent of w.

Proposition 1: Let v be a convolutional codeword of the same length as w. Then for all $j = 1, \ldots, 2^{l-\nu}$

$$H(p_j(w)) - H(p_1(w)) \leq H(p_j(v)) - H(p_1(v)). \tag{2}$$

To prove Proposition 1, we need some preliminary work.

Lemma 1: Let u and v be two convolutional codewords of the same length as w. Then for all $j = 1, \ldots, 2^{l-\nu}$

$$H(p_j(u)) = H(p_j(v)).$$

Lemma 2: With $h = H(p_1(w))$, there exist $h+1$ words $u_0, \ldots, u_h$ such that

$$u_0 = c(p_1(w)) \tag{3}$$

$$u_h = w \tag{4}$$

$$H(u_i, u_{i+1}) = 1,\ i = 0, \ldots, h-1 \tag{5}$$

$$H(p_1(u_{i+1})) = H(p_1(u_i)) + 1,\ i = 0, \ldots, h-1. \tag{6}$$

Lemma 3: Let $n \geq 1$. Let $a_i,\ i = 1, \ldots, n$, be nondecreasing integers and $b_i,\ i = 1, \ldots, n$, be integers such that for all $i = 1, \ldots, n$

$$|b_i - a_i| \leq 1. \tag{7}$$

Then there exists a permutation $f : \{1, \ldots, n\} \to \{1, \ldots, n\}$ such that for all $i = 1, \ldots, n-1$

$$b_{f(i)} \leq b_{f(i+1)} \tag{8}$$

and for all $i = 1, \ldots, n$

$$|b_{f(i)} - a_i| \leq 1. \tag{9}$$

The proofs of the above lemmas are given in Appendix I, II, and III. Using these lemmas, we can now prove Proposition 1.

Proof: If $H(p_1(w)) = 0$, then w is a codeword, and the proposition follows from Lemma 1. Suppose now that $H(p_1(w)) = h \geq 1$ and let $u_0, \ldots, u_h$ be the $h+1$ words given by Lemma 2. Because u_i and u_{i+1} differ in only one bit, we have

$$\begin{aligned} H(u_{i+1}, c(p_j(u_i))) &= H(u_i, c(p_j(u_i))) \pm 1 \\ &= H(p_j(u_i)) \pm 1. \end{aligned} \tag{10}$$

Let $a_j = H(p_j(u_i))$ and $b_j = H(u_{i+1}, c(p_j(u_i)))$. Then the sequences (a_j) and (b_j) fulfill the conditions of Lemma 3. Thus, there exists a permutation f such that

$$H(u_{i+1}, c(p_{f(j)}(u_i))) \leq H(u_{i+1}, c(p_{f(j+1)}(u_i))). \tag{11}$$

Hence the sequence $(H(u_{i+1}, c(p_{f(j)}(u_i))))$ is nondecreasing in $j = 1, \ldots, 2^{l-\nu}$. Since it also takes the same values as the nondecreasing sequence $(H(p_j(u_{i+1})))$, we conclude that $H(u_{i+1}, c(p_{f(j)}(u_i))) = H(p_j(u_{i+1}))$ for all $j = 1, \ldots, 2^{l-\nu}$. Thus (9) gives

$$|H(p_j(u_{i+1})) - H(p_j(u_i))| \leq 1, \tag{12}$$

which leads to

$$\begin{aligned} H(p_j(u_{i+1})) - H(p_1(u_{i+1})) &= H(p_j(u_{i+1})) - [H(p_1(u_i)) + 1] \quad \text{(due to (6))} \\ &\leq 1 + H(p_j(u_i)) - [H(p_1(u_i)) + 1] \quad \text{(due to (12))} \\ &= H(p_j(u_i)) - H(p_1(u_i)). \end{aligned} \tag{13}$$

Now we have

$$\begin{aligned} H(p_j(v)) - H(p_1(v)) &= H(p_j(u_0)) - H(p_1(u_0)) \quad \text{(due to Lemma 1)} \\ &\geq H(p_j(u_h)) - H(p_1(u_h)) \quad \text{(due to (13))} \\ &= H(p_j(w)) - H(p_1(w)) \quad \text{(due to (4)).} \end{aligned}$$

■

Using Proposition 1 with $v = \mathbf{0}$ and noting that the Hamming metric of the best path of a convolutional codeword is 0, we can give a bound on the number of lists that is independent of the received word w.

Corollary 1: The number of lists needed by the ml-TTA is bounded by $H(p_n(\mathbf{0})) + 1$.

B. Proposed algorithm

Exploiting Corollary 1, we are now able to improve the original TTA by using an array of unsorted lists instead of a sorted list. At each stack insertion, one list of the array is selected by an array index derived from the path metric of the new stack element. The new stack element is then appended to the selected list. This eliminates the need for search operations, since the order of the stack is induced by the order of the lists within the array.

mL-TTA Algorithm:

Forward pass: use the same forward pass as in the original TTA.

Backward pass: as in the TTA except for Steps 1, 2, 3, and 6, which should be modified as follows.

1) In the first backward pass, set $f_e = H(p_n(\mathbf{0}))$ and allocate $f_e + 1$ lists $L_0, \ldots, L_{f_e}$. Initialize all lists as being empty. Set $k = 1$, $t = l$, $i = 0$, $m = 0$, $p_k(t) = 0$, $f_s = 0$, $m_{\min} = M_1(0, l)$, $m_{\max} = f_e + m_{\min}$, $c = 0$, and go to Step 6. In each subsequent backward pass, set $k = k + 1$, update f_s by $f_s = \min j$ where $j \geq f_s$ and L_j is not empty.

2) Let S_1 be the first element of L_{f_s}. Set $t = S_1.t$, $i = p_{S_1.q}(t)$, $m = S_1.u$, $p_k(t) = p_{S_1.q}(t)$, $p_k(t+1) = p_{S_1.q}(t+1), \ldots, p_k(l) = p_{S_1.q}(l)$.

3) Remove the first element from L_{f_s} and set $c = c - 1$.

6) If $M_2(i, t) + m > m_{\max}$, go to Step 7. Otherwise, let S_h be given by $S_h.m = M_2(i, t) + m$, $S_h.q = k$, $S_h.t = t$, and $S_h.u = m$. Append S_h to the list $L_{S_h.m - m_{\min}}$

and set $c = c + 1$. If $c \leq n - k$, go to Step 7. Otherwise, update f_e by setting $f_e = \max j$ where $j \leq f_e$ and L_j is not empty, remove the last element from L_{f_e} and set $m_{\max} = f_e + m_{\min}$ and $c = c - 1$.

The variable $m_{\min}$ is the smallest path metric. The variable $m_{\max}$ is an upper bound on the metric of p_n. It is initialized according to Proposition 1 and updated when there are more elements in the stack than paths left to decode. The variables f_s and f_e are list indices used to find the first and the last element of the stack in constant time. Variable c is the number of stack elements. In Step 1, we initialize the stack and update f_s. In Step 2, S_1 is now the first element of L_{f_s} because this is an element with the smallest path metric. In Step 6, we first check if $M_2(i,t) + m$, the metric of the path p' that branches at the current position from the currently decoded path, is larger than $m_{\max}$. If this is the case, then we know from Proposition 1 that p' is not among the n best paths, so we do not insert an element for p' into the stack. Otherwise, we append a new stack element for p' to the list $L_{S_h.m-m_{\min}}$. If the stack contains more elements than paths left to decode, we also remove the last element from the stack, which is the last element of L_{f_e}.

We illustrate the algorithm with the example of Section II-B and compute the first two best paths when $n = 5$. In the first backward pass, we set $f_e = H(p_5(\mathbf{0})) = 5$ and allocate six lists $L_0, \ldots, L_5$, which are initialized as empty. We set $k = 1$, $t = 7$, $i = 0$, $m = 0$, $p_1(7) = 0$, $f_s = 0$, $m_{\min} = M_1(0,7) = 2$, $m_{\max} = 7$, $c = 0$. Since $M_2(0,7) + m = 5 < m_{\max} = 7$, we create a new stack element S_h with $S_h.m = 5$, $S_h.q = 1$, $S_h.t = 7$, $S_h.u = 0$. We append S_h to $L_{S_h.m-m_{\min}} = L_3$ and set $c = 1$. We then set $j = v(0,7) = 0$, update m by setting $m = m + M_1(0,7) - M_1(0,6) = 0$ and set $p_1(6) = 0$. We then set $t = 6$ and $i = 0$. The remaining of the first backward pass proceeds as follows. Create stack element S_h with $S_h.m = 4$, $S_h.q = 1$, $S_h.t = 6$, $S_h.u = 0$. Append S_h to L_2 and set $c = 2$. Set $j = 2$, $m = 0$, $p_1(5) = 2$, $t = 5$, $i = 2$. Create stack element S_h with $S_h.m = 3$, $S_h.q = 1$, $S_h.t = 5$, $S_h.u = 0$. Append S_h to L_1 and set $c = 3$. Set $j = 3$, $m = 1$, $p_1(4) = 3$, $t = 4$, $i = 3$. Create stack element S_h with

$S_h.m = 6$, $S_h.q = 1$, $S_h.t = 4$, $S_h.u = 1$. Append S_h to L_4 and set $c = 4$. Set $j = 1$, $m = 1$, $p_1(3) = 1$, $t = 3$, $i = 1$. Create stack element S_h with $S_h.m = 5$, $S_h.q = 1$, $S_h.t = 3$, $S_h.u = 1$. Append S_h to L_3 and set $c = 5$. Since $c > n - k = 4$, update f_e by setting $f_e = 4$, remove the last element of $L_{f_e} = L_4$, set $m_{\max} = f_e + m_{\min} = 6$ and $c = 4$. Set $j = 2$, $m = 2$, $p_1(2) = 2$, $t = 2$, $i = 2$. Set $j = 1$, $m = 2$, $p_1(1) = 1$, $t = 1$, $i = 1$. Finally, set $j = 0$, $m = 2$, $p_1(0) = 0$, $t = 0$, and $i = 0$. The first backward pass is now complete and p_1 is the path $0, 1, 2, 1, 3, 2, 0, 0$ with corresponding codeword 11100001011100 and input bits 1011000. The metric of p_1 is $m = 2$. The stack contains four elements: $(3, 1, 5, 0)$ in L_1, $(4, 1, 6, 0)$ in L_2, $(5, 1, 7, 0)$ and $(5, 1, 3, 1)$ in L_3. The lists L_0, L_4, and L_5 are empty. After the end of the second backward pass (see [9] for details), we obtain p_2 as the path $0, 1, 2, 0, 1, 2, 0, 0$ with corresponding codeword 11101111101100 and input bits 1001000. The metric of p_2 is $m = 3$. The stack contains three elements: $(4, 1, 6, 0)$ and $(4, 2, 4, 1)$ in L_2, and $(5, 1, 7, 0)$ in L_3. The lists L_0, L_1, L_4, and L_5 are empty.

To insert an element into the stack, we append it to the list given by its metric, which is a constant time operation. If the stack contains more than $n - k$ elements after insertion, the last non-empty list must be found to remove the last element from it. For this, only lists L_j with $j \leq f_e$ must be checked. Indeed, no element S_h was inserted into a list L_j with $j > f_e$ because $S_h.m$ would have been greater than $m_{\max}$. Therefore, during all n backward passes, the array of lists will be completely traversed at most once. Finding the first non-empty list for removing the top of stack element is similar. No element S_h was inserted in a list L_j with $j < f_s$ because $S_h.m$ would have been smaller than the metric of the currently computed path. Furthermore, the first non-empty list cannot be behind the last non-empty list. Thus, for both search operations, the array of lists will be completely traversed at most once during all n backward passes. This leads to a time complexity of $O(nl + H(p_n(\mathbf{0})))$ for n backward passes. Note that $H(p_n(\mathbf{0}))$ increases very slowly with n (see Figure 4) and is bounded by rl.

To use the mL-TTA, we must know $H(p_n(\mathbf{0}))$, which is the metric of an nth best path of

the received word $w = \mathbf{0}$. This metric can be computed with another LVA or, as follows, with the proposed ml-TTA. We start with an approximation for $H(p_n(\mathbf{0}))$, for example, 1. Then we run the ml-TTA with this approximation and $w = \mathbf{0}$. If there exists a $k,\ 1 < k \leq n$, such that the stack is empty at the beginning of the kth backward pass, then we run the algorithm again with $H(p_n(\mathbf{0}))$ set to double the current approximation. Otherwise, the exact value of $H(p_n(\mathbf{0}))$ is given by the value of $S_1.m$ at the beginning of the nth backward pass. In this way, at most $\lceil \log_2 H(p_n(\mathbf{0})) + 1 \rceil$ executions of the mL-TTA are needed to determine $H(p_n(\mathbf{0}))$.

Table I summarizes the time complexity of all LVAs. Note that the results are valid for both the average and the worst case. Table II gives the space complexity of all LVAs.

IV. Extensions

A. Space complexity reduction

To decode the kth best path p_k, the TTA and the ml-TTA need the path $p_{q(k)}$ $(q(k) < k)$ from which p_k branches and the node $(i(k), t(k))$ at which p_k branches from $p_{q(k)}$. In the implementations described in the previous sections, all decoded paths $p_a,\ 1 \leq a < k$, are stored, and a simple copy operation allows to reconstruct p_k from node $(i(k), t(k))$ to node $(0, l)$. At the expense of decoding speed, memory space can be saved by keeping track of the indices $q(k), q(q(k)), q(q(q(k))), \ldots$ and the stages $t(k), t(q(k)), t(q(q(k))), \ldots$ instead of explicitly storing all decoded paths. Let us call $p_{q(k)}$ the parent of p_k and the paths $p_{q(k)}, p_{q(q(k))}, \ldots, p_1$ ancestors of p_k. A low space complexity version of the TTA is obtained by introducing a list that stores $q(a)$ and $t(a)$ for each decoded path $p_a,\ 2 \leq a < k$ and replacing the copy operation in Step 2 by the following steps. First, we use the list to determine the ancestors of p_k. Then we start decoding p_k at node $(0, l)$ by following the backtrace pointers. At each node $(i(k'), t(k'))$ where an ancestor $p_{k'}$ of p_k branches from its parent, we follow $\bar{v}(i(k'), t(k'))$. The same modification can be applied to the ml-TTA. Implementation details can be found in [9].

The following example illustrates the idea. Suppose that we want to decode the tenth best

path p_{10}. Suppose now that p_{10} branches from p_7 at node $(1,3)$ and that p_7 branches from p_1 at node $(0,5)$. To determine p_{10} from node $(0,l)$ to node $(1,3)$, we start at node $(0,l)$ and follow the backtrace pointers until node $(0,5)$. At this node, we follow $\bar{v}(0,5)$. Then we keep following the backtrace pointers until we reach the node $(1,3)$.

Since the memory space required to store all paths p_k is $O(nl)$ and the memory space required by the additional information is only $O(n)$, this alternative implementation reduces the space complexity of the TTA to $O(2^\nu l + n)$ and that of the mL-TTA to $O(2^\nu l + n + H(p_n(\mathbf{0})))$.

B. Soft decision decoding

The Hamming metric is useful with hard decision decoding. In practice, soft decision decoding is often preferable. In this case, we exploit the following proposition, which extends Proposition 1 to arbitrary bit metrics.

Proposition 2: Let A be a finite channel output alphabet. Let $M(x,y)$ be a bit metric ($x \in \{0,1\}$, $y \in A$). Then for all words $w \in A^{rl}$ and all $j = 1, \ldots, 2^{l-\nu}$, we have

$$M(p_j(w)) - M(p_1(w)) \leq d(H(p_j(\mathbf{0})) - H(p_1(\mathbf{0}))) \tag{14}$$

where d is such that for all $x_1, x_2 \in \{0,1\}$ and $y_1, y_2 \in A$

$$|M(x_1,y_1) - M(x_2,y_2)| \leq d. \tag{15}$$

Proof: Since A is finite, there always exists a positive number d that satisfies (15). We first prove that there exist j paths $\theta_1, \ldots, \theta_j$ such that $M(c(\theta_i), w) - M(p_1(w)) \leq d(H(p_j(\mathbf{0})) - H(p_1(\mathbf{0})))$ for all $i = 1, \ldots, j$. The proof of (14) follows then from the fact that $p_1(w), \ldots, p_j(w)$

are j paths with smallest path metric relative to w. Let $\theta_i = p_i(c(p_1(w))),\ i = 1, \ldots, j$. Then

$$\begin{aligned} M(c(\theta_i), w) - M(p_1(w)) &= M(c(\theta_i), w) - M(c(p_1(w)), w) \\ &\leq dH(c(\theta_i), c(p_1(w))) \quad \text{(due to (15))} \\ &= dH(\theta_i) \\ &\leq dH(\theta_j) \\ &= dH(p_j(\mathbf{0})) \quad \text{(due to Lemma 1)} \\ &= d(H(p_j(\mathbf{0})) - H(p_1(\mathbf{0}))). \end{aligned}$$

■

To use the mL-TTA with soft decision decoding and an arbitrary integer bit metric M, we find the smallest d that satisfies (15). Then we simply set $m_{\max} = dH(p_n(\mathbf{0}))$ in the list allocation step and apply the algorithm with the integer bit metric M instead of the Hamming metric H. Thus, also in this case, the time complexity of the ml-TTA is linear in n.

For a real-valued bit metric $M(x, y) \in [a, b]$, we use an integer bit metric obtained by quantizing the real numbers $M(x, y)$ to integers $\bar{M}(x, y) = \lfloor NM(x, y) \rfloor \in \{\lfloor Na \rfloor, \ldots, \lfloor Nb \rfloor\}$, where N is a constant factor. It is easy to see that $d = \lfloor Nb \rfloor - \lfloor Na \rfloor$ fulfills inequality (15). Note, however, that because of the loss of information induced by the quantization, n best paths with respect to $\bar{M}(x, y)$ are not guaranteed to be n best paths with respect to $M(x, y)$.

V. Experimental results

In this section, we present decoding results of the different LVAs for both a BSC and an AWGN channel. Except for the LEA, the performance of all algorithms is independent of the channel statistics. The channel code was the binary rate-1/4 convolutional code with generator polynomials (0177,0127,0155,0171) (octal) and memory order 6 [10]. The performance of each algorithm was evaluated for the same set of randomly generated information sequences of length 216 bits. The run times were measured on an Intel Pentium III 800 MHz processor machine.

Figure 3 compares the average CPU decoding time for the PLVA [4], the SLVA of [4], which we denote by SLVA1, the improved SLVA [5], which we denote by SLVA2, the LEA [6], the TTA where the stack is a linked list (L-TTA) [3], the TTA with a red-black tree as a stack (T-TTA), and the mL-TTA described in Section III (mL-TTA1). As expected, mL-TTA1 was always the fastest algorithm. Although the time complexity of the PLVA is also linear in the number of paths, the algorithm was very slow in practice because the multiplicative constant in its time complexity is large. The CPU time of SLVA1 also increased very quickly with the number of paths, and only the LEA, SLVA2, and the TTA variants required less than 5 ms to compute 200 paths. L-TTA was faster than T-TTA when the number of paths was small because inserting into a balanced binary tree is more complex than inserting into a list, and the time for search operations in the list is not dominant for small lists. When the number of paths was large, both L-TTA and SLVA2 were slower than T-TTA. Note how in accordance with the theory their CPU time increased quadratically with the number of paths.

Figure 4 shows that the number of lists needed for the stack by mL-TTA1 increased very slowly with the number of paths.

Figure 5 compares the performance of the algorithms for an AWGN channel. All algorithms used an integer bit metric in $[0, 1023]$ (see Subsection IV-B). The results were similar to those of the BSC case, with the exception that ml-TTA1 was slower than T-TTA when the number of paths was small. This is due to the fact that in the AWGN case, the multiple list approach needed 1023 times as many lists as in the BSC case. Thus, when the number of paths was small, the speed-up due to the multiple list approach was not important enough to alleviate the cost for initializing so many lists.

Figure 6 compares the time complexity of mL-TTA1 and ml-TTA2, which is the multiple-list TTA with reduced space complexity described in Section IV-A. mL-TTA2 showed the same linear behavior as mL-TTA1, but was somewhat slower due to the increased complexity of Step 2 in its backward pass.

VI. Conclusion

We showed that the time complexity of the tree-trellis list Viterbi algorithm [3] can be made linear in the number n of decoded paths by using an array of unsorted lists instead of a single sorted list. The size of the array was determined by finding a tight upper bound on the difference between the largest and the smallest metric of the n best paths. We also explained how to use our multiple-list tree-trellis algorithm with an arbitrary integer bit metric, making it suitable for soft-decision decoding. Another contribution of the paper was to compare the time complexity of the best published LVAs theoretically and experimentally for both a BSC and an AWGN channel. Simulations showed that for the BSC, the multiple-list tree-trellis algorithm was the fastest LVA, independent of the number of paths. For the Gaussian channel, the multiple-list tree-trellis algorithm was the fastest when the number of paths was not too small.

The main motivation of our fast algorithm was to improve the rate-distortion performance of the concatenated joint source-channel coding system of [1] by using a large number of candidate paths. We showed [11] that by allowing 10,000 candidate paths instead of the 100 ones used in the original work, one can improve the expected peak-signal-to-noise ratio by up to 0.5 decibels. Computing so many paths in reasonable time would not have been possible without a very fast LVA. Finally, we point out that a fast LVA is useful in other concatenated communications systems [4] and other applications such as speech recognition [3], [7].

Appendix I
Proof of Lemma 1

Proof: Let $x = u \oplus v$, where $\oplus$ is the bitwise exclusive-or operation. The convolutional code is the set $\{c(p_j(v)),\ j = 1, \ldots, 2^{l-\nu}\}$. Because of the linearity of this code, it is also the set $\{c(p_j(u)) \oplus x,\ j = 1, \ldots, 2^{l-\nu}\}$, for each $j \in \{1, \ldots, 2^{l-\nu}\}$ there exists a unique $k \in \{1, \ldots, 2^{l-\nu}\}$ such that $c(p_k(v)) = c(p_j(u)) \oplus x$. Moreover, $H(p_j(u)) = H(c(p_j(u)), u) =$

$H(c(p_j(u))\oplus x, v) = H(p_k(v))$. Thus, the nondecreasing sequences $(H(p_j(u)))_j$ and $(H(p_k(v)))_k$ are equal, which gives the desired result. ■

APPENDIX II
PROOF OF LEMMA 2

Proof: We set $u_0 = c(p_1(w))$. For $i \geq 0$ and as long as $u_i \neq w$, we construct u_{i+1} from u_i by changing a bit of u_i that differs from the bit at the same position of w. Because $H(p_1(w)) = H(c(p_1(w)), w) = h$, we must change exactly h bits and therefore get $h+1$ words $u_i,\ i = 0, \ldots, h$. Furthermore,

$$H(u_i, c(p_1(w))) = H(u_{i+1}, c(p_1(w))) - 1. \tag{16}$$

Since the metric of any path changes by $+1$ or -1 if one bit of the word is changed, (16) implies that $p_1(w)$ is a best path of u_i if it is a best path of u_{i+1}. But $u_h = w$, thus $p_1(w)$ is a best path of u_i for all $i = 0, \ldots, h$, which gives (6). ■

APPENDIX III
PROOF OF LEMMA 3

Proof: We prove the lemma by induction on n. The result is trivially true for $n = 1$. Suppose now that it is true for n. Thus, there exists a permutation f on $\{1, \ldots, n\}$ that satisfies (8) and (9). Let a_{n+1} and b_{n+1} be integers such that $|b_{n+1} - a_{n+1}| \leq 1$. If $b_{n+1} > b_{f(n)}$, then the permutation f' on $\{1, \ldots, n+1\}$ defined by $f'(i) = f(i)$ if $i \in \{1, \ldots, n\}$ and $f'(n+1) = n+1$ satisfies (8) and (9). If $b_{n+1} \leq b_{f(n)}$, then let j be the smallest number in $\{1, \ldots, n\}$ such that $b_{n+1} \leq b_{f(j)}$. The permutation f' on $\{1, \ldots, n+1\}$ defined by $f'(i) = f(i)$ for $i \in \{1, \ldots, j-1\}$, $f'(j) = n+1$, and $f'(k+1) = f(k)$ for $k \in \{j, \ldots, n-1\}$ satisfies (8). Also f' satisfies (9) for $i \in \{1, \ldots, j-1\}$. Let us prove that $|b_{f'(j)} - a_j| \leq 1$ and $|b_{f'(k+1)} - a_{k+1}| \leq 1$ for $k \in \{j, \ldots, n-1\}$. If $b_{n+1} \geq a_j$,

then

$$\begin{aligned} |b_{f'(j)} - a_j| &= |b_{n+1} - a_j| \\ &= b_{n+1} - a_j \\ &\leq b_{f(j)} - a_j \\ &\leq 1. \end{aligned}$$

If $b_{n+1} < a_j$, then

$$\begin{aligned} |b_{f'(j)} - a_j| &= |b_{n+1} - a_j| \\ &= a_j - b_{n+1} \\ &\leq a_{n+1} - b_{n+1} \\ &\leq 1. \end{aligned}$$

Similarly, if $b_{f(k)} \geq a_{k+1}$, then

$$\begin{aligned} |b_{f'(k+1)} - a_{k+1}| &= |b_{f(k)} - a_{k+1}| \\ &= b_{f(k)} - a_{k+1} \\ &\leq b_{f(k)} - a_k \\ &\leq 1. \end{aligned}$$

If $b_{f(k)} < a_{k+1}$, then

$$\begin{aligned} |b_{f'(k+1)} - a_{k+1}| &= |b_{f(k)} - a_{k+1}| \\ &= a_{k+1} - b_{f(k)} \\ &\leq a_{n+1} - b_{n+1} \\ &\leq 1. \end{aligned}$$

Hence the result is also true for $n + 1$, which completes the proof. ■

Acknowledgment. We thank the three anonymous reviewers whose useful comments and suggestions helped improve the presentation and clarity of the paper.

REFERENCES

[1] P. G. Sherwood and K. Zeger, "Progressive image coding for noisy channels," *IEEE Signal Processing Lett.,* vol. 4, pp. 189–191, July 1997.

[2] A. Said and W. A. Pearlman, "A new fast and efficient image codec based on set partitioning in hierarchical trees," *IEEE Trans. Circuits Syst. Video Technol.,* vol. 6, pp. 243–250, June 1996.

[3] F. K. Soong and E.-F. Huang, "A tree-trellis based fast search for finding the N best sentence hypotheses in continuous speech recognition," in *Proc. ICASSP'91 IEEE Int. Conf. Acoustics Speech Signal Processing*, vol. 1, pp. 705–708, Toronto, 1991.

[4] N. Seshadri and C.-E. W. Sundberg, "List Viterbi decoding algorithms with applications," *IEEE Trans. Commun.,* vol. 42, pp. 313–323, 1994.

[5] C. Nill and C.-E. W. Sundberg, "List and soft symbol output Viterbi algorithms: extensions and comparisons," *IEEE Trans. Commun.,* vol. 43, pp. 277–287, 1995.

[6] J.S. Sadowsky, "A maximum likelihood decoding algorithm for turbo codes," in *Proc. GLOBECOM '97*, vol. 2, pp. 929–933, Phoenix, AZ, Nov. 1997.

[7] L. Rabiner and B.-H. Juang, *Fundamentals of Speech Recognition*, Prentice Hall, 1993.

[8] R. Hinze, "Constructing red-black trees," in *Proc. Workshop Algorithmic Aspects of Advanced Programming Languages*, pp. 89–99, Paris, Sept. 1999.

[9] M. Röder and R. Hamzaoui, "Fast list Viterbi decoding and application for source-channel coding of images," *Konstanzer Schriften in Mathematik und Informatik* [Online] Preprint no. 182, `http://www.inf.uni-konstanz.de/Preprints/preprints-all.html`.

[10] P. Frenger, P. Orten, T. Ottosson, and A. Svensson, *Multi-rate convolutional codes,* Chalmers Univ. Technol. Göteborg, Technical report R021/1998, 1998.

[11] M. Röder and R. Hamzaoui, "Fast list Viterbi decoding and application for source-channel coding of images," in *Proc. ICME-2002 IEEE International Conference on Multimedia and Expo*, vol. 1, pp. 801–804, Lausanne, August 2002.

	Forward pass	Backward passes	Overall
PLVA	$O(n2^\nu l)$	$O(nl)$	$O(n2^\nu l)$
SLVA1	$O(2^\nu l)$	$O(ln^2)$	$O(l(2^\nu + n^2))$
SLVA2	$O(2^\nu l)$	$O(nl + n^2)$	$O(l(2^\nu + n) + n^2)$
LEA	$O(2^\nu l)$	$O(nl + n^2)$	$O(l(2^\nu + n) + n^2)$
L-TTA	$O(2^\nu l)$	$O(ln^2)$	$O(l(2^\nu + n^2))$
T-TTA	$O(2^\nu l)$	$O(ln \log n)$	$O(l(2^\nu + n \log n))$
ml-TTA	$O(2^\nu l)$	$O(l(n + r))$	$O(l(2^\nu + n + r))$

TABLE I

TIME COMPLEXITY OF THE LVAS. PLVA IS THE PARALLEL LVA [4], SLVA1 IS THE ORIGINAL SERIAL LVA [4], SLVA2 IS THE IMPROVED SERIAL LVA [5], LEA IS THE LIST EXTENSION ALGORITHM OF [6], L-TTA IS THE TTA WITH A SINGLE LIST AS A STACK [3], T-TTA IS THE TTA WITH A RED-BLACK TREE AS A STACK, AND ML-TTA IS THE MULTIPLE LIST TTA OF SECTION III.

PLVA	$O(n2^\nu l)$
SLVA1	$O(2^\nu l)$
SLVA2	$O(l(2^\nu + n))$
LEA	$O(l(2^\nu + n))$
L-TTA	$O(l(2^\nu + n))$
T-TTA	$O(l(2^\nu + n))$
ml-TTA	$O(l(2^\nu + n + r))$

TABLE II

SPACE COMPLEXITY OF THE LVAS OF TABLE I.

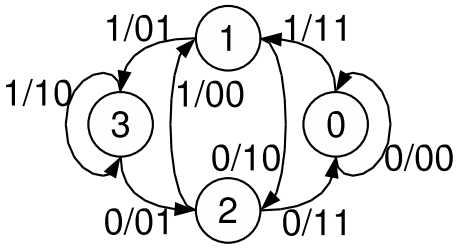

Fig. 1. State diagram of the convolutional code with generator polynomials (7,5) (octal). Vertices denote the states of the code. Edges denote the state transitions. Edge labels are given as input bit/output bits associated with the state transitions.

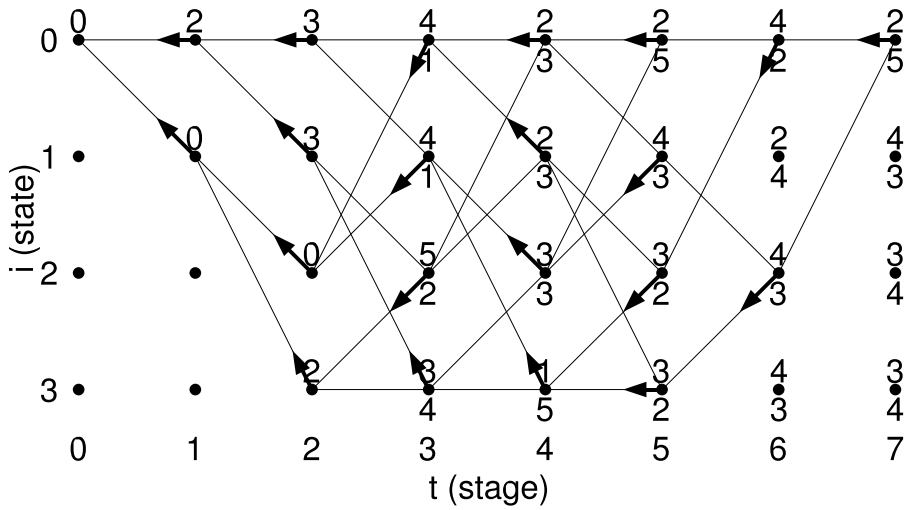

Fig. 2. Trellis diagram of the convolutional code with generator polynomials (7,5) and output of the TTA forward pass for the received word 11101001001100. The two numbers shown at a node (i, t) are the partial path metrics $M_1(i, t)$ and $M_2(i, t)$. Metrics equal to infinity are not shown. The arrows represent the backtrace pointer $v(i, t)$.

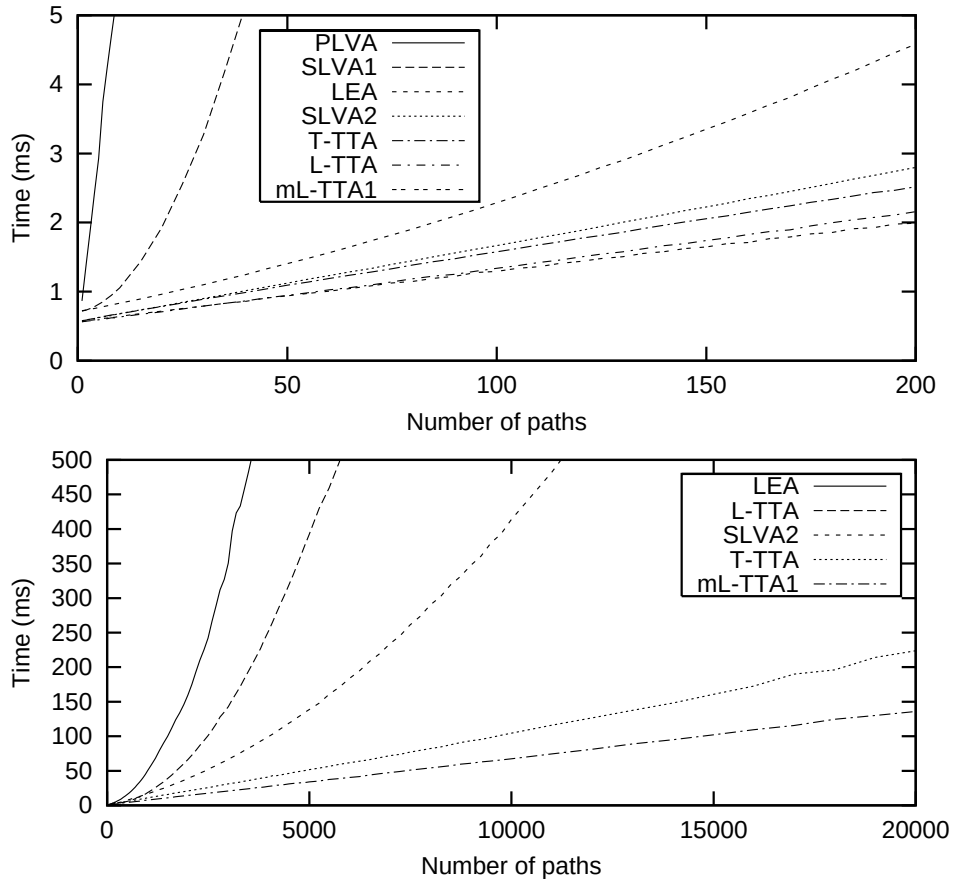

Fig. 3. Time complexity of the list Viterbi algorithms for a BSC with bit error rate 0.1. PLVA is the parallel LVA [4], SLVA1 is the original serial LVA [4], SLVA2 is the improved serial LVA [5], LEA is the list extension algorithm of [6], L-TTA is the TTA with a single list as a stack [3], T-TTA is the TTA with a red-black tree as a stack, mL-TTA1 is the multiple-list TTA of Section III.

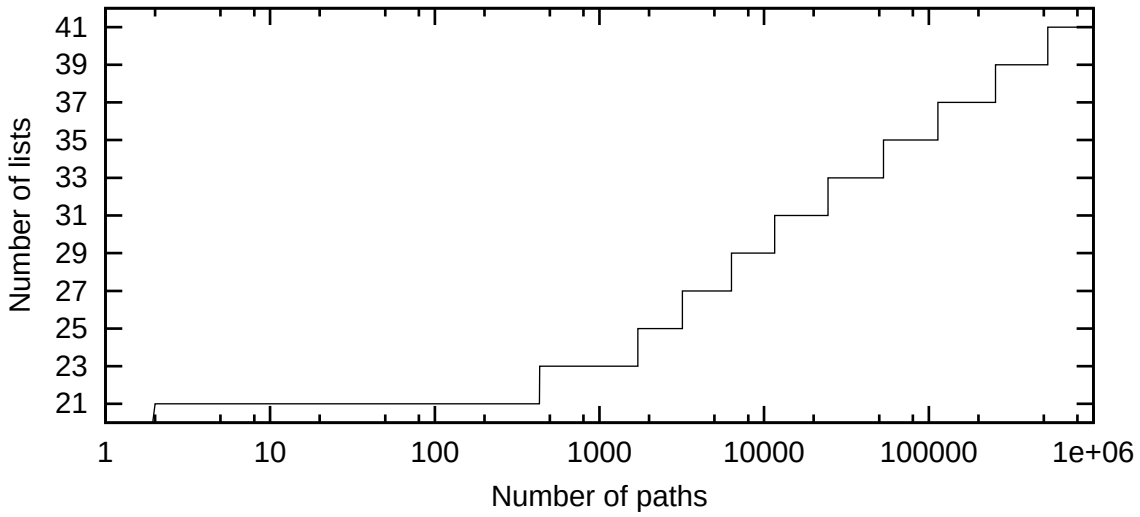

Fig. 4. Number of lists used by the stack as a function of the number of paths.

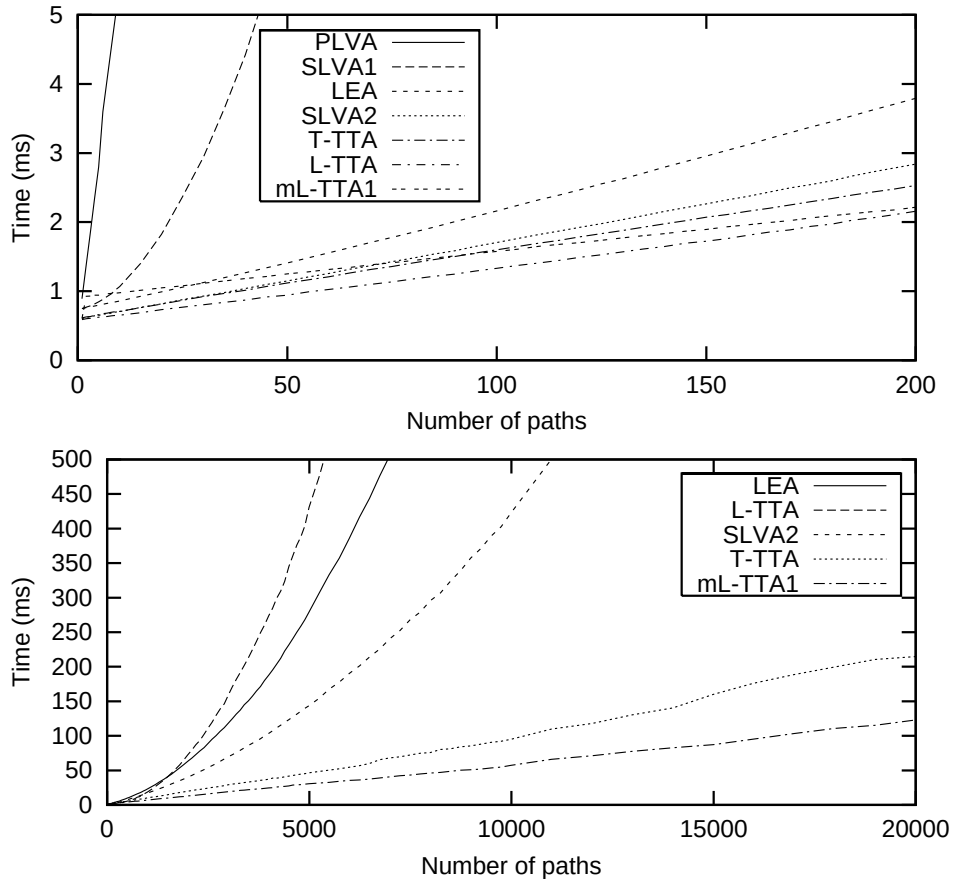

Fig. 5. Time complexity of the list Viterbi algorithms for the AWGN channel with symbol-energy to noise-spectral density ratio 1 dB. PLVA is the parallel LVA [4], SLVA1 is the original serial LVA [4], SLVA2 is the improved serial LVA [5], LEA is the list extension algorithm of [6], L-TTA is the TTA with a single list as a stack [3], T-TTA is the TTA with a red-black tree as a stack, mL-TTA1 is the multiple-list TTA of Section III.

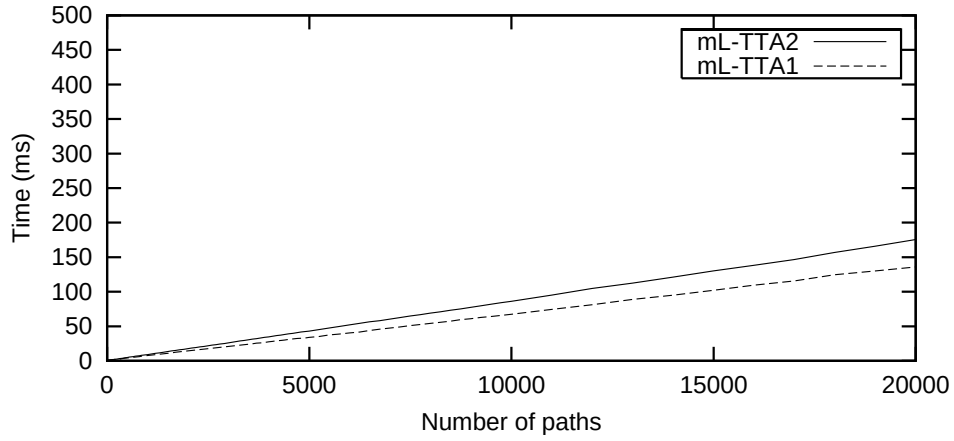

Fig. 6. Time complexity of the multiple-list TTAs for the BSC. ml-TTA1 is the multiple-list TTA of Section III and mL-TTA2 is the variant with reduced space complexity of Section IV-A.