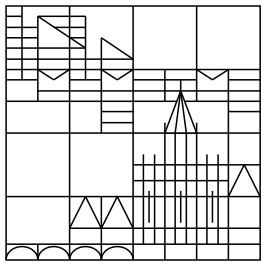

Universität Konstanz

Fast List Viterbi Decoding and Application for Source-Channel Coding of Images

Martin Röder
Raouf Hamzaoui

Konstanzer Schriften in Mathematik und Informatik
Nr. 182, Dezember 2002

ISSN 1430–3558

Universität Konstanz
Fach D 188, 78457 Konstanz, Germany
Email: preprints@informatik.uni–konstanz.de
WWW: http://www.informatik.uni–konstanz.de/Schriften/

Fast List Viterbi Decoding and Application for Source-Channel Coding of Images

Martin Röder, Raouf Hamzaoui

Abstract

A list Viterbi algorithm (LVA) finds n best paths in a trellis. We propose a new implementation of the tree-trellis LVA. Instead of storing all paths in a single sorted list, we show that it is more efficient to use several lists, where all paths of the same list have the same metric. For an integer metric, both the time and space complexity of our implementation are linear in n. Experimental results show that our implementation is much faster than all previous LVAs. This allows us to consider a large number of paths in acceptable time. As an application, we show that by increasing the number of candidate paths, one can significantly improve the performance of a popular progressive source-channel coding system that protects an embedded image code with a concatenation of an outer error detecting code and an inner error correcting convolutional code.

I. Introduction

Sherwood and Zeger [1] devised a powerful systems for protecting images against errors in a memoryless noisy channel. The system uses a concatenation of 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 compressed with an embedded wavelet coder [2]. Then the output bitstream is divided into consecutive packets of fixed length. To each packet, a CRC code is appended, and the resulting blocks are encoded with equal error protection by an RCPC coder. The receiver uses an LVA to find a most likely (best) decoding solution (path) for a received packet. This path is checked by computing the CRC checksum bits. If an error is detected, the LVA is used again to find a next best 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 which case, called incomplete decoding, the decoding is stopped, and the image is reconstructed using only the correctly decoded packets.

The performance of this system depends on the maximum number of candidate paths for one packet. Increasing the maximum number of candidates decreases the probability of incomplete

This paper was presented 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, hamzaoui@fmi.uni-konstanz.de.

decoding and improves the average reconstruction fidelity. However, the number of candidates that can be considered is limited by the time complexity of the LVA. On the other hand, the probability that the CRC test fails to detect a packet error increases monotonically with the number of candidates. This problem can be handled by increasing the length of the CRC codeword. But then, the RCPC code rate has to be increased as well to keep the transmission rate fixed.

The main contribution of this paper is an efficient implementation of the tree-trellis LVA [3]. Instead of storing all n paths in a single sorted list, we show that it is advantageous to use several lists, where all paths of the same list have the same metric. In particular, we prove that for the Hamming metric H the worst-case time complexity of our implementation is $O(l(2^m+n)+H(p_n(\emptyset)))$, where l is the length of the input sequence of the encoder (i.e., $l+1$ is the length of the trellis), m is the memory order of the convolutional coder, and $p_n(\emptyset)$ is an nth best path of the zero codeword. Since $H(p_n(\emptyset)) \leq lr$, where r is the rate of the mother code, the time complexity of our implementation is linear in n. This compares favorably with the original algorithm [3] whose average complexity is $O(l(2^m + n^2 + n))$ and with other state-of-the-art LVAs [4], [5]. We also extend this result to arbitrary integer metrics, which makes our algorithm useful in soft decision decoding. Experimental results for the binary symmetric channel (BSC) showed that our algorithm was also the fastest in practice.

The second contribution of this paper was to show that one can improve the performance of the system of [1] by increasing the number of candidate paths. In particular, we determined for both the original search depth of 100 paths [1] and a search depth of 10,000 paths an optimal combination of the CRC codeword length and the RCPC code rate and showed that the higher search depth significantly improves the rate-distortion performance of the system at several bit error rates.

II. Terminology and Previous Work

A. Terminology

A binary rate $1/q$ convolutional encoder outputs q bits for each input bit. This group of q output bits is called an *output frame*. When an output frame is transmitted over the channel, errors are introduced. The resulting block is called a *code frame*. The encoder has a memory order of m, which gives 2^m possible *states* z_i, $i = 0, \ldots, 2^m - 1$. When an input bit is read by the encoder, it is shifted into the memory, leading to a *state transition* $(z_i \rightarrow z_j)$. The output frame associated to state transition $(z_i \rightarrow z_j)$ is denoted by $r(z_i \rightarrow z_j)$. For $t = 0, 1, \ldots, l$, we

denote by $z(t)$ the state at stage t. A *path* $p = (z(0) \to z(1)), \ldots, (z(l-1) \to z(l))$ is a sequence of state transitions starting and ending at a valid state. We suppose that both the starting state and the final state are the zero state. The convolutional codeword associated to p is the word $c(p)$ obtained by concatenating the l output frames $r(z(t-1) \to z(t)),\ t = 1, \ldots, l$. Let w be a received packet of l code frames $w(1), \ldots, w(l)$ and let M be an integer bit metric. Then, $M(p(w)) = M(c(p), w) = \sum_{t=1}^{l} M(r(z(t-1) \to z(t)), w(t))$ is the *path metric* of path p relative to w. An LVA finds n paths $p_1(w), \ldots, p_n(w)$ such that $M(p_1(w)) \leq \ldots \leq M(p_n(w))$.

B. Tree-Trellis Algorithm

The best previous LVAs are the parallel LVA (PLVA) [4], the serial LVA (SLVA) [4], which was improved in [5], and the tree-trellis algorithm (TTA) [3]. Although the time complexity of the PLVA is linear in n, the algorithm is not suitable for most applications because on one hand, the multiplicative constant in its time complexity is large and on the other hand, it determines n best paths simultaneously. In contrast, the other algorithms find n best paths successively, using $k-1$ best paths to determine a kth best 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 describe the TTA. In the next section, we show how to improve this algorithm.

Suppose that a packet of l code frames is received. To find n best paths for this packet, the TTA uses one forward pass and n backward passes in the trellis of $l+1$ stages associated to the state diagram of the convolutional coder. The forward pass stores the two lowest partial path metrics for each state $z_j(t)$. We denote the lower one by $M_1(z_j(t))$ and the higher one by $M_2(z_j(t))$. Furthermore, for each state $z_j(t)$, a backtrace pointer $v(z_j(t))$ to the predecessor state of the partial path with the lower metric is stored. The backward passes use a stack of paths. For each path S_k in the stack, the TTA stores: $S_k.p$, the number of the backward pass at which S_k was inserted into the stack, $S_k.m$, the path metric, $S_k.t$ and $S_k.i$, the stage and the state at which S_k branches from a path found in a previous backward pass, respectively, and $S_k.m_p$, the partial path metric from $S_k.t$ to the end of the trellis. The stack is sorted such that $S_k.m \leq S_{k+1}.m,\ k \geq 1$. In each backward pass, a path is found and the corresponding codeword is stored in an array D_i, where i is the number of the backward pass.

Before the first backward pass, the stack is initialized with a single element S_1 given by $S_1.m = M_1(z_0(l))$ (the metric of a best path), $S_1.t = l$, $S_1.i = 0$, $S_1.m_p = 0$, and $S_1.p = 0$. The number of the current backward pass is initialized as $k = 0$. A backward pass proceeds as follows.

1. The number of the current backward pass is increased, $k = k + 1$.

2. The backward pass starts at stage $S_1.t$ and state $S_1.i$, thus we set $t = S_1.t$ and $i = S_1.i$. The running partial path metric m is set to $m = S_1.m_p$. In the first backward pass, a best path will be found. In each subsequent backward pass, the path that will be found and the path found in the $S_1.p$th backward pass share the common path segment from $S_1.t$ to the end of the trellis; therefore the decoded data of this path segment can simply be copied from the data stored in $D_{S_1.p}$.

3. The first entry S_1 is removed from the stack. In the first backward pass, we go to step 6. In each subsequent backward pass, we continue with step 4.

4. The new path branches from the path found in a previous backward pass at stage t and state i. Therefore, we take the predecessor state the backtrace pointer does not point to for the first state transition, that is, we set j such that the state transition $(z_j \rightarrow z_i)$ is possible, but $j \neq v(z_i(t))$.

5. The running partial path metric m increases by the metric of the state transition, so we set $m = m + M_1(z_i(t)) - M_1(z_j(t-1))$. The input bit of the encoder associated with the state transition $(z_j \rightarrow z_i)$ is prepended to the decoded data D_k of the currently decoded path and then the state transition is finished by setting $t = t - 1$ and $i = j$.

6. The locally second best path at this state must be inserted into the stack. The metric of this path is the sum of $M_2(z_i(t))$, stored during the forward pass, and the running partial path metric m. Therefore, we insert a new element S_h with $S_h.m = M_2(z_i(t)) + m$, $S_h.p = k$, $S_h.t = t$, $S_h.i = i$ and $S_h.m_p = m$ at a position h satisfying the sorting order into the stack.

7. The following state transitions follow the backtrace pointers, thus we set $j = v(z_i(t))$.

8. Steps 5 to 7 are repeated until state 0 at stage 0 is reached.

Because the stack is sorted according to the path metric, the next best path is at the top of the stack after each backward pass and is decoded in the following backward pass. To reduce memory requirements and avoid unnecessary stack insertions, the stack size can be limited to $n - 1$ elements because all elements inserted behind element $n - 1$ never reach the top of the stack.

The time complexity of the forward pass is $O(2^m l)$. In the backward pass, the most time-consuming part is the sorting of the stack. If the stack is implemented as a linked list, inserting a new element will require $n/2$ 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. Thus, for n paths, the average time complexity of the TTA is $O(l(2^m + n^2 + n))$. To reduce the time

complexity of the TTA, we propose to maintain the sorted stack with a red-black-tree [6], which guarantees a logarithmic time complexity for insertion and search operations. This reduces the time complexity of the n backward passes to $O(l(n + \log_2((n-1)!))) = O(l(n \log_2 n))$.

We now study the space complexity of the algorithm. In the forward pass, the algorithm stores two metrics for each state. In the backward pass, it stores for each stack element the five above values, together with two pointers for the linked list. Furthermore, the decoded data of each path found is stored. This results in a space complexity of $O(2^m l + nl + n)$.

III. Improvements to the TTA

In this section, we reduce the time complexity of the TTA by eliminating the costs due to maintaining a sorted stack. We do so by replacing the sorted list with multiple lists such that all paths in one list have the same metric. Those lists do not have to be sorted, and an insertion operation is done by simply appending the new stack entry into the list corresponding to its metric, which is a constant time operation. The problem is to determine the number of lists needed. One could try to allocate the lists dynamically at runtime, but this would require search operations too, because we must check if a list for a given metric already exists. A better way is to estimate the number of needed lists before the algorithm starts and then use an array of lists. Then we can simply use the metric of the new path as an index in that array. The following proposition gives an upper bound on the number of lists.

Proposition 1: Let M be an integer metric. Let w be a received packet and n be the maximum number of candidate paths for w. Then at most $M(p_n(w)) - M(p_1(w)) + 1$ lists are needed.

Proof: Since the TTA is an LVA, it finds all paths $p_j(w)$ such that $M(p_1(w)) \leq M(p_j(w)) \leq M(p_n(w))$. We need one list for each of the $M(p_n(w)) - M(p_1(w)) + 1$ integer values between $M(p_1(w))$ and $M(p_n(w))$. ■

However, we need a bound that is independent of the received packet. The following proposition gives an answer to this problem when M is the Hamming metric.

Proposition 2: Let w be an arbitrary packet and let v be a convolutional codeword of the same length. Let H be the Hamming metric. Then for all $j \geq 1$

$$H(p_j(w)) - H(p_1(w)) \leq H(p_j(v)) - H(p_1(v)). \tag{1}$$

To prove the above proposition, we need some preliminary work.

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

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

Proof: Let $x = u \oplus v$, where $\oplus$ is the bitwise exclusive-or operation. The convolutional code is the set $\{c(p_j(v)),\ j = 1, 2, \ldots, 2^{l-m}\}$. Since it is also the set $\{c(p_j(u)) \oplus x,\ j = 1, 2, \ldots, 2^{l-m}\}$, for each $j \in \{1, \ldots, 2^{l-m}\}$ there exists a unique $k \in \{1, \ldots, 2^{l-m}\}$ 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 $(H(p_k(v)))_j$ are equal. ■

Lemma 2: Let w be an arbitrary packet. Let H be the Hamming metric. Then, with $h = H(p_1(w))$, there exists $h+1$ packets $u_0, \ldots, u_h$ such that

$$u_0 = c(p_1(w)) \tag{2}$$

$$u_h = w \tag{3}$$

$$H(u_i, u_{i+1}) = 1 \tag{4}$$

$$H(p_1(u_{i+1})) = H(p_1(u_i)) + 1 \tag{5}$$

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 $h+1$ packets $u_i,\ i = 0, \ldots, h$. Furthermore,

$$H(u_i, c(p_1(w))) = H(u_{i+1}, c(p_1(w))) - 1. \tag{6}$$

Since the metric of any path changes by $+1$ or -1 if one bit of the packet is changed, (6) 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 (5). ■

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 $i = 1, \ldots, n$

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

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. ■

We are now ready to prove Proposition 2.

Proof: Let w be an arbitrary packet. If $H(p_1(w)) = 0$, then w is also 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 $h+1$ packets 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

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

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

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

which leads to

$$\begin{aligned} H(p_j(u_{i+1})) - H(p_1(u_{i+1})) &= H(p_j(u_{i+1})) - [H(p_1(u_i)) + 1] \quad \text{(due to (5))} \\ &\leq 1 + H(p_j(u_i)) - [H(p_1(u_i)) + 1] \quad \text{(due to (12))} \\ &= H(p_j(u_i)) - H(p_1(u_i)). \end{aligned} \tag{13}$$

Now we have

$$\begin{aligned} H(p_j(v)) - H(p_1(v)) &= H(p_j(c(p_1(w)))) - H(p_1(c(p_1(w)))) \quad \text{(due to Lemma 1)} \\ &= H(p_j(u_0)) - H(p_1(u_0)) \quad \text{(due to (2))} \\ &\geq H(p_j(u_h)) - H(p_1(u_h)) \quad \text{(due to (13))} \\ &= H(p_j(w)) - H(p_1(w)) \quad \text{(due to (3)).} \end{aligned}$$

■

Corollary 1: Let w be an arbitrary packet. Let H be the Hamming metric. Then the number of lists needed is bounded by $H(p_n(\emptyset)) + 1$.

Proof: The proof follows from Proposition 1, Proposition 2, and the observation that the Hamming metric of the best path of a codeword is 0. ■

We now provide our algorithm, which we call multiple-list tree-trellis algorithm (mL-TTA).

mL-TTA Algorithm:

List allocation: Set $m_{\max} = H(p_n(\emptyset))$ and allocate $m_{\max} + 1$ lists $L_0, \ldots, L_{m_{\max}}$.

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

Stack initialization: A best path has to be inserted into the stack. The stack element for this path is initialized with the same values as in the original TTA and inserted as the first element of L_0. We define $m_{\min}$ as the smallest path metric and set it to the metric of a best path, $m_{\min} = H_1(z_0(l))$. We also need f_s and f_e, the numbers of the first and the last non-empty list, respectively. These numbers are initialized as $f_s = 0$ and $f_e = m_{\max}$.

Backward pass: Each backward pass is the same as in the original TTA, but with modified operations on the path stack.

Stack insertion: If the new element S_h (see Step 6 of the original TTA) is such that $S_h.m > m_{\max}$, no insertion is done. Otherwise, the new element is appended to the list $L_{S_h.m - m_{\min}}$. If the stack now contains more than n paths, the last element from the stack is removed. This is the last element from the last non-empty list, which is indicated by f_e. Since the stack was altered, we must first update f_e by setting $f_e = \max j$ where $j \leq f_e$ and L_j is not empty. Then we remove the last element from L_{f_e}. We must also update $m_{\max}$ by $m_{\max} = f_e + m_{\min}$ because there are already n paths in the stack and paths inserted after the n-th path will never be decoded.

Reading and removing the top of stack element: The top of stack element is the first element of the first non-empty list given by f_s. Since the stack was altered, f_s must be updated first by $f_s = \min j$ where $j \geq f_s$ and L_j is not empty. Then the first element of L_{f_s} is read and removed from L_{f_s}.

To insert a path into the stack, we append it to the list indicated by its metric, which is a constant time operation. If the stack consists of more than n 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 since no element could have been inserted in lists with $j > f_e$ because their metric would be greater than $m_{\max}$. Therefore, during *all* n backward passes needed to find n paths, 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 path can be inserted in lists L_j with $j < f_s$ because such a path would have a lower metric than the current one, breaking the

path order condition for LVAs. Furthermore, the first non-empty list cannot be behind the last non-empty list. Thus, for both search operations, the array of lists will at most be completely traversed once during all n backward passes. This leads to a time complexity of $O(nl+H(p_n(\emptyset)))$ for n backward passes. The time complexity of the mL-TTA is much lower than that of the single list TTA because $H(p_n(\emptyset))$ increases very slowly with n. Note that $H(p_n(\emptyset))$ is bounded by rl.

Compared to the single list TTA, the space complexity of the mL-TTA increases by $H(p_n(\emptyset))$ since we use $H(p_n(\emptyset))$ lists instead of a single one, each requiring one start and one end pointer. Thus the overall space complexity of the mL-TTA is $O(2^m l + nl + n + H(p_n(\emptyset))$.

To use the mL-TTA, one needs to determine $H(p_n(\emptyset))$. The easiest way is to use another LVA and simply find an n-th path of the zero codeword. This can be very time consuming if the number of paths is large, so it is better to use the mL-TTA itself for this task, which can be done as follows. We start with an approximation for $m = H(p_n(\emptyset))$, e.g., $m = 1$. Then we set $m_{\max} = m$ and allocate m lists which gives all paths $p_j(\emptyset)$ with $H(p_j(\emptyset)) \leq m$. We determine $p_1(\emptyset), p_2(\emptyset), \ldots$ until we either find $p_n(\emptyset)$ or the path stack is empty, in which case, we double m and allocate lists and search paths again until we find $p_n(\emptyset)$. In this way, $\lceil \log_2 H(p_n(\emptyset)) + 1 \rceil$ executions of the mL-TTA give $H(p_n(\emptyset))$.

The Hamming metric is useful with hard decision decoding. In practice, soft decision decoding is often preferable. The following proposition explains how to use the ml-TTA with soft decision decoding.

Proposition 3: Let $M(x, y)$ be an integer symbol metric, A be the channel alphabet, $x_1, x_2 \in A$ and $y_1, y_2 \in \{0, 1\}$. If there exists $d > 0$ with

$$|M(x_1, y_1) - M(x_2, y_2)| \leq d \tag{14}$$

for all x_1, x_2, y_1, y_2, then for all packets w and all $j \geq 1$

$$M(p_j(w)) - M(p_1(w)) \leq d(H(p_j(\emptyset)) - H(p_1(\emptyset))) \tag{15}$$

Proof: The Hamming metric of all paths changes by 1 if one bit of the packet is changed. With a metric satisfying (14), the metric of all paths can at most change by d, so all path metrics can at most be d times as high as with the Hamming metric. ■

To use soft decision decoding with the mL-TTA, we find the lowest d that satisfies (14) and multiply it with the number of needed lists. We also must use integer values for the symbol metric. These integer values can be calculated from a floating point metric by scaling and rounding. Thus, also for arbitrary integer metrics, the time complexity of our algorithm remains

linear in n. For example, instead of using a floating point metric with values in $[0, 1]$, we scale this range to $[0, 1024]$ by multiplication of all metric values with 1024. Then we round to the nearest integer and use the result as symbol metric for the mL-TTA.

IV. Application to image transmission

By increasing the path search depth in the source-channel coding system of [1], we decrease the probability of incomplete decoding, but we increase the probability that the CRC fails to detect an error. If we fix the search depth and the bit error rate (BER) of the BSC, a rate-based optimal performance of the system can be obtained by maximizing the expected number of correctly received source bits over all CRC and RCPC code rate pairs. This can be done as follows. We send a large number N_P of random packets through the channel and decode each received packet, distinguishing between correct decoding, undetected error, and incomplete decoding. Suppose that the target transmission rate is b, which corresponds to $N = \frac{rbP}{L+c+m}$ packets sent, where P is the number of pixels in the image, c is the CRC length, and m is the memory order. We divide the N_P random packets into n_B blocks of N successive packets. In each block $i = 1, \ldots, n_B$, we count the number N_{r_i} of received packets before the first incomplete decoding. We estimate the expected number of received packets N_{e_i} in block i by N_{r_i} if all N_{r_i} packets were correctly decoded and by zero if at least one error was undetected, which is a reasonable assumption because an undetected error can completely damage the image reconstruction. The expected number of correctly received source bits is simply $E_N = \frac{\sum_{i=1}^{n_B} N_{e_i}}{n_B} L$, where L is the number of source bits per packet.

V. Experimental results

Figure 1 compares the time complexity of the PLVA [4], the SLVA [4], the improved SLVA (SLVA2) [5], the TTA with a single list as a stack (L-TTA) [3], the TTA with a red-black tree as a stack (T-TTA), and the mL-TTA. The figure shows the CPU time of each algorithm as a function of the number of paths for a random packet of length 200 bits that was transmitted over a BSC with bit error rate (BER) 0.1 and encoded with the rate 1/4 convolutional code (0177,0127,0155,0171) (octal) of [7]. The time was measured on an AMD Athlon 1000 MHz processor with a main memory size of 256 Mbytes. The CPU time of the PLVA and the SLVA increased very fast, and only SLVA2 and the TTA variants required less than 5 ms for the first 200 paths. The L-TTA was faster than the T-TTA when the number of paths was low 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. However, the situation changed

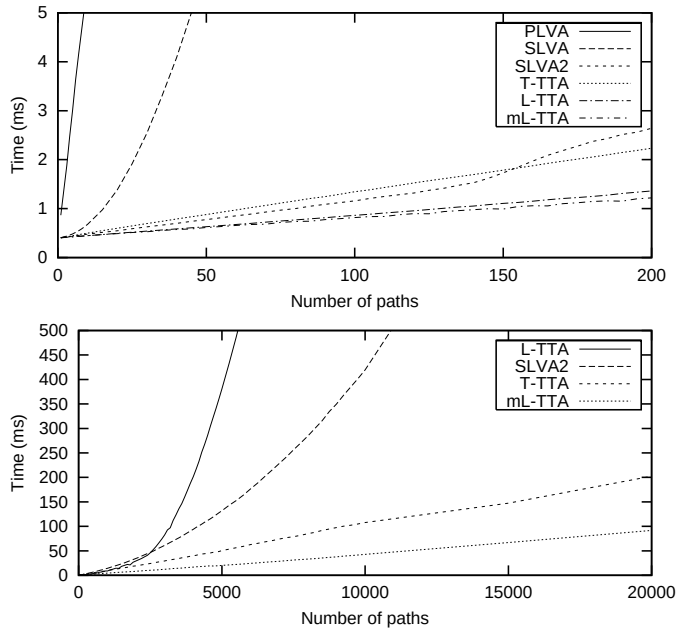

Fig. 1. Time complexity of the PLVA [4], the SLVA [4], SLVA2 [5] (improved SLVA), the L-TTA [3], the T-TTA (L-TTA with a red-black tree), and the mL-TTA for a BSC with BER 0.1.

when we increased the number of paths. The mL-TTA was always the fastest algorithm.

To maximize the expected number of correctly decoded source bits E_N, we considered RCPC codes rates $\{8/9, \ldots, 8/32\}$ and CRC lengths 16, 24, and 32 bits. We used the rate 1/3 memory order 6 mother code (0133,0165,0171) for rates $8/9, \ldots, 8/24$ and the rate 1/4 memory order 6 mother code (0177,0127,0155,0171) for the remaining rates. Both mother codes are optimum distance spectrum codes from [7], and the RCPC codes are low rate optimized with a puncturing period of 8 [7]. The CRC generator was $(x^{16}+x^{14}+x^{12}+x^{11}+x^8+x^5+x^4+x^2+1)$ from [1] for a 16 bit CRC, $(x^{24}+x^{23}+x^{18}+x^{17}+x^{14}+x^{11}+x^{10}+x^7+x^6+x^5+x^4+x^3+x+1)$ from [8] for a 24 bit CRC, and $(x^{32}+x^{26}+x^{23}+x^{22}+x^{16}+x^{12}+x^{11}+x^{10}+x^8+x^7+x^5+x^4+x^2+x+1)$ from [9] for a 32 bit CRC. The 16 bit CRC generator was optimized for the selected packet length of 200 source bits, the other CRC generators are standard. Table I shows the results of the optimization for the search depth of 100 paths proposed in [1] and our proposed depth of 10,000 paths. The transmission rate was 1.0 bpp. The test image was the standard 512×512 Lena. For both search depths, the best CRC length was 16 bits. For BER=0.001, the same code rate was selected because it is already the highest possible code rate for the selected puncturing period and, therefore, only a small increase of E_N is noticeable. For the higher BER values, the optimal code rate for 10,000 paths is higher than that for 100 paths. Thus, we can send more source bits for the same number of packets.

BER	Code Rate		CRC length		E_N	
	n_1	n_2	n_1	n_2	n_1	n_2
0.001	8/9	8/9	16	16	208,807	209,723
0.01	8/11	8/10	16	16	162,705	177,221
0.1	8/26	8/23	16	16	71,116	80,304

TABLE I

Optimization results for $n_1 = 100$ and $n_2 = 10,000$ paths at 1.0 bpp. E_N denotes the expected number of correctly received source bits.

BER	Code rate	CRC length	E_N
0.001	8/9	16	209,684
0.01	8/11	16	170,643
0.1	8/24	16	76,777

TABLE II

Optimization results for 900 paths at 1.0 bpp.

The source-channel coding system with the mL-TTA and 10,000 paths needed 0.40 ms, 0.44 ms, and 0.45 ms CPU time to decode a packet at BER 0.001, 0.01, and 0.1, respectively. This is less than 25 % more than the time required with the L-TTA and 100 paths. Table II shows the optimization results with the L-TTA and a search depth of 900 paths, which corresponds to the complexity of the system with the mL-TTA and 10,000 paths. We conclude that even when both LVAs were allowed the same CPU time, the mL-TTA yielded a better performance.

Finally, Figure 2 shows the rate-distortion curves for BER=0.1 at various transmission rates. The peak-signal-to-noise ratio (PSNR) was computed from the corresponding E_N. At 1.0 bpp, the search depth of 10,000 paths led to a gain of 0.5 dB compared to the original search depth of 100 paths and to a gain of 0.2 dB compared to a search depth of 900 paths.

VI. Conclusion

We showed that the time complexity of the TTA [3] can be made linear in the number of candidates by using multiple lists instead of a single sorted list. In contrast to the previous LVAs [3], [4], our algorithm can consider a large number of candidates in reasonable time. This significantly improves the PSNR performance of the system of [1] because a higher code rate can be used without severely increasing the probability of incomplete decoding.

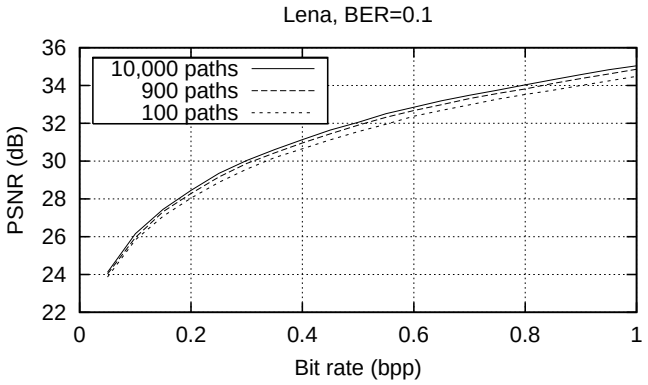

Fig. 2. Rate-distortion curves of the Lena image for the three search depths and a BSC with BER 0.1.

Our results can be further improved in many ways. For example, we may allow more CRC lengths than the three used in this paper. We also may try to determine an optimal number of candidate paths. Increasing the number of paths does not necessarily improve the performance of the system because we may reach a point where the probability of undetected errors becomes too high.

References

[1] P. G. Sherwood, K. Zeger, *Progressive image coding for noisy channels*, *IEEE Sig. Proc. Letters*, vol. 4, number 7, pp. 189–191, 1997.

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

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

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

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

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

[7] P. Frenger, P. Orten, T. Ottosson, A. Svensson, *Multi-rate convolutional codes,* Technical Report 21, Chalmers University of Technology, Göteborg, 1998.

[8] J. Callas, L. Donnerhacke, H. Finney, R. Thayer, *OpenPGP Message Format*, Request for Comments 2440, Internet Engineering Task Force, 1998.

[9] IEEE Std 802.3, 2000 Edition, *Part 3: Carrier sense multiple access with collision detection (CSMA/CD) access method and physical layer specifications,* p. 41, 2000.