Soft-output BEAST decoding with application to product Codes

Loncar, Maja; Johannesson, Rolf; Bocharova, Irina; Kudryashov, Boris

Published in:
IEEE Transactions on Information Theory

DOI:
10.1109/TIT.2007.915971

2008

Citation for published version (APA):
Soft-Output BEAST Decoding with Application to Product Codes

Maja Lončar, Student Member, IEEE, Rolf Johannesson, Fellow, IEEE, Irina E. Bocharova, and Boris D. Kudryashov

Abstract—A Bidirectional Efficient Algorithm for Searching code Trees (BEAST) is proposed for efficient soft-output decoding of block codes and concatenated block codes. BEAST operates on trees corresponding to the minimal trellis of a block code and finds a list of the most probable codewords. The complexity of the BEAST search is significantly lower than the complexity of trellis-based algorithms, such as the Viterbi algorithm and its list-generalizations. The outputs of BEAST, a list of best codewords and their metrics, are used to obtain approximate a posteriori reliabilities of the transmitted symbols, yielding a soft-output symbol decoder referred to as the BEAST-APP decoder. This decoder is employed as a component decoder in iterative schemes for decoding of product and incomplete product codes. Its performance and convergence behavior are investigated using EXIT charts and compared to existing decoding schemes.

It is shown that the BEAST-APP decoder achieves performances close to the BCJR decoder with a substantially lower computational complexity.

Index Terms—BEAST, block turbo codes, list decoding, product codes, SISO decoding.

I. INTRODUCTION

E VERY block code can be represented by a time-varying trellis [1], which facilitates the use of the Viterbi algorithm [2] for maximum-likelihood (ML) decoding. Finding a list of more than one of the most likely codewords can be accomplished by the generalized list-Viterbi algorithm [3],[4]. The complexity of these algorithms is dictated by the underlying trellis complexity. The bit-level trellis with the smallest complexity is called the minimal trellis of the code, and it is obtained from the generator matrix in the minimal-span form [5], or from the parity-check matrix of the code.

An alternative algorithm for efficient ML decoding of block codes, presented in [6], [7] is the Bidirectional Efficient Algorithm for Searching Trees (BEAST). BEAST finds an ML codeword by searching two code trees, rather than a trellis, and, as shown in [6], has substantially lower complexity than the Viterbi algorithm. The key to BEAST’s efficiency is the fact that the bidirectional search allows efficient eschewing of paths that cannot lead to the best codewords. Hence, unlike in the Viterbi algorithm, many “bad” nodes are never visited, which yields significant savings in complexity. In [8], BEAST was adapted to search for a list of several best codewords, and it was shown that the computational cost of this search lies well below that of the list-Viterbi algorithm.

In case of non-uniform probabilities of the information bits, a priori probabilities can easily be incorporated into the BEAST search, resulting in an efficient maximum a posteriori (MAP) sequence decoding algorithm.

Since the introduction of turbo codes [9], much attention has been devoted to soft-input soft-output (SISO) symbol decoders, which are commonly used as component decoding modules in iterative decoding schemes for various concatenated code structures (see, e.g., [10], [11] and the references therein). The component-wise optimal decoder is the MAP symbol decoder, such as the trellis-based BCJR algorithm [12]. Turbo coding schemes can attain very low bit error rates with simple convolutional component codes and a properly designed interleaver. Good performance, however, is achieved at the expense of a significant decoding delay—the required interleaver size is usually of the order of several thousands of codesymbols. In many practical applications, data is transmitted in short packets, and large decoding delays are not acceptable (cf., e.g., [13]). For such scenarios, product codes offer an alternative to turbo codes. Iterative decoding of product codes using trellis-based MAP symbol decoders was investigated in [14], [15]. Many block codes of practical interest have trellises of high state complexity, and thus, the BCJR-type MAP decoding is computationally too expensive. Several suboptimum algorithms that aim at approximating the true MAP decoding have been developed based on the idea that the symbol reliabilities can be formed from a list of suitable candidate codewords, [16]–[22]. Many of the proposed methods, e.g., [17], [19], [21], use a Chase-type decoder [23] to generate a list of candidate codewords. These strategies exploit the algebraic structure of the code to generate candidate codewords, and the obtained set of candidate codewords is not necessarily the set of the $L$ best codewords (although the set may contain the very best, namely, the ML codeword). The number of candidate codewords $L$ needs to be sufficiently large (which is achieved at the expense of increased complexity) and it increases with the minimum distance of the code. In the more efficient methods [16],...
[22], which require no algebraic code structure, candidate codewords are obtained by encoding reliable test information sets.

In this paper, it is shown how the outputs of BEAST, that is, a list of the globally best codewords and their soft-decision metrics, can easily be exploited to obtain a list-based SISO symbol decoder, which we call the BEAST-APP decoder. The list size necessary to achieve high accuracy of the APP-approximations is rather short, while the complexity of a BEAST search is virtually independent of \( L \) for a wide range of modest list sizes. By simulations, it is shown that the BEAST-APP decoder performs as well as the max-log-MAP decoder, while having lower complexity. Compared to algebraic-type list decoders of roughly the same complexity, BEAST-APP decoding achieves lower bit-error-rate. Convergence behavior of the iterative BEAST-APP decoding is investigated using EXIT charts and decoding trajectories, and it is compared to the BCJR (log-MAP) and max-log-MAP iterative decoding.

The paper is organized as follows: In Section II, a suitable soft-decision decoding metric is derived, and some important properties of a minimal trellis are reviewed, as prerequisites for formulating the BEAST. In Section III, the BEAST-APP algorithm is presented and illustrated by an example. Section IV is devoted to iterative decoding of product codes using BEAST-APP constituent decoders. In Section V, the decoder’s convergence is studied using EXIT charts. Simulation results and a comparison with other algorithms are presented in Section VI. Section VII summarizes and concludes the paper.

II. PRELIMINARIES

A. Soft-Decision Decoding with Weighted Hamming Metric

Let \( (N, K, d_{\text{min}}) \) denote a binary linear block code \( \mathcal{C} \) of length \( N \), dimension \( K \), and minimum distance \( d_{\text{min}} \), used for communicating over the additive white Gaussian noise (AWGN) channel with bipolar signaling. An \( K \times N \) generator matrix of the code is denoted by \( G \), and its rows are denoted by \( g_k \), \( k = 1, 2, ..., K \). The information sequence \( u = (u_1, u_2, ..., u_K) \) is encoded into a codeword \( v = uG = u_1g_1 + u_2g_2 + \cdots + u_Kg_K \), which is subsequently mapped onto a bipolar code sequence \( x = (x_1, x_2, ..., x_N) \) according to \( x_i = (1 - 2v_k) \sqrt{E_s} \), \( i = 1, 2, ..., N \), where \( E_s = R E_b \) is the symbol energy, \( R = K/N \) is the code rate and \( E_b \) is the energy per bit. Without loss of generality, we set the symbol energy \( E_b = 1 \).

The information bits are assumed to be mutually independent, that is, the \( a \) \( \text{priori} \) sequence probability factorizes as \( p(v) = p(u) = \prod_{k=1}^{K} p(u_k) \). The logarithmic ratio \( L_{a}(u_k) = \log p(u_k|v) - \log p(u_{\neg k}|v) \) is the \( a \) \( \text{priori} \) L-value for the \( k \)th bit position, \( k = 1, 2, ..., K \). Let \( h_a = (h_{a_1}, h_{a_2}, ..., h_{a_K}) \) be a sequence of hard \( a \) \( \text{priori} \) bit decisions, that is,

\[
    h_{a_k} = \begin{cases} 
        0, & L_{a}(u_k) \geq 0 \\
        1, & L_{a}(u_k) < 0.
    \end{cases}
\]

Then the weighted Hamming distance [24] between the information sequence \( u \) and the sequence \( h_a \) is

\[
    \mu(u, h_a) = \sum_{k=1}^{K} \mu(u_k, h_{a_k})
\]

where

\[
    \mu(u_k, h_{a_k}) = \begin{cases} 
        |L_{a}(u_k)|, & u_k \neq h_{a_k} \\
        0, & u_k = h_{a_k}.
    \end{cases}
\]

Note that the positions where the bits are equiprobable do not contribute to the metric, that is, \( \mu(u_k, h_{a_k}) = 0 \) for \( L_{a}(u_k) = 0 \). For such positions \( k \), the value \( h_{a_k} = 0 \) in (1) was chosen arbitrarily.

The received sequence is given by \( r = x + n \), where the noise sequence \( n \) contains realizations of independent, identically distributed Gaussian random variables with zero mean and variance \( N_0/2 \). Since the channel is memoryless and without feedback, a received symbol \( r_i \) depends only on the \( i \)th transmitted codesymbol \( v_i \), i.e., the likelihood factorizes as \( p(r|v) = \prod_{i=1}^{N} p(r_i|v_i) \). The channel L-value \( L_{ch}(v_i) = \log \frac{p(r_i|v_i=0)}{p(r_i|v_i=1)} = \frac{4}{N_0} r_i \) specifies the reliability of the \( i \)th received symbol. Let \( h_{ch} = (h_{ch_1}, h_{ch_2}, ..., h_{ch_N}) \) denote the hard-decision received sequence, that is, \( h_{ch_i} = (1 - \text{sign}(r_i))/2 \), \( i = 1, 2, ..., N \). Then the weighted Hamming distance between the codeword \( v \) and \( h_{ch} \) is obtained by weighing each differing position by the corresponding channel reliability, that is,

\[
    \mu(v, h_{ch}) = \sum_{i=1}^{N} \mu(v_i, h_{ch_i})
\]

where

\[
    \mu(v_i, h_{ch_i}) = \begin{cases} 
        |L_{ch}(v_i)|, & v_i \neq h_{ch_i} \\
        0, & v_i = h_{ch_i}.
    \end{cases}
\]

The maximum \( a \) \( \text{posteriori} \) (MAP) sequence decoder outputs a codeword \( \hat{v} \) with the largest \( a \) \( \text{posteriori} \) probability (APP) \( p(v|r) \), or equivalently,

\[
    \hat{v} = \arg\min_{v \in \mathcal{C}} \left\{ A \left( \log p(v|r) - f(r) \right) \right\}, \quad A < 0
\]

where \( A \) is an arbitrary negative constant, \( f(r) \) is a function that, in general, depends on the received sequence but is the same for all codewords, and \( \log p(v|r) = \log p(r|v) + \log p(v) - \log p(r) \). It is straightforward to show, by the appropriate choice of \( A \) and \( f(r) \), cf. [25], that (4) is equivalent to

\[
    \hat{v} = \arg\min_{v \in \mathcal{C}} \left\{ \mu(v, h_{ch}) + \mu(u, h_a) \right\}
\]

that is, the most probable codeword has the minimum total weighted Hamming metric \( \mu(v, h_{ch}) + \mu(u, h_a) \). In the case of equiprobable codewords, the metric reduces to \( \mu(v, h_{ch}) \).

B. Minimal Trellis of a Block Code

Sequence and symbol decoding of block codes can be accomplished using the Viterbi and the BCJR algorithm, respectively, operating on the code’s minimal trellis. The minimal trellis has the lowest state- and branch-complexity of all bit-level trellises for the given code and thus yields the lowest complexity of the trellis-based decoding methods.
An alternative to trellis decoding is the bidirectional tree-based decoding algorithm BEAST, which will be discussed in Section III. The code trees explored by BEAST are obtained by "unfolding" the minimal trellis, that is, by ignoring the state mergers. We summarize some basic properties of the minimal trellis, which are important for formulating the BEAST.

The minimal trellis is constructed from the generator matrix in the minimal-span form [5] (also called the trellis-oriented form [1]). Any generator matrix can be reduced to such a form using row operations only. Let \( \text{start}(g_j) \) and \( \text{end}(g_j) \) denote the first and the last nonzero position of the row \( g_j \) of a code's generator matrix, respectively, where \( 1 \leq \text{start}(g_j) \leq \text{end}(g_j) \leq N \). Then the generator matrix which has the property that no two rows start or end in the same position, that is, \( \text{start}(g_j) \neq \text{start}(g_k) \) and \( \text{end}(g_j) \neq \text{end}(g_k) \) hold for every \( j \neq k \), is said to be in the minimal-span form. A row \( g_j \) is said to be active in the position \( i \) if \( i \in [\text{start}(g_j), \text{end}(g_j)] \). Let \( A_i \) denote the set of rows of the minimal-span generator matrix that are active at position \( i \). Then the states at the \( i \)-th depth of the minimal trellis are \( K \)-tuples given by

\[
\sigma_i = (\sigma_i^1 \sigma_i^2 \ldots \sigma_i^K), \quad \sigma_i^j = \begin{cases} u_j, & \text{if } g_j \in A_i \\ 0, & \text{otherwise} \end{cases}
\]

for \( j = 1, 2, \ldots, K \) and \( i = 1, 2, \ldots, N \). The root and the toor of the trellis are the all-zero states, \( \sigma_0 = \sigma_N = (0 0 \ldots 0) \). The number of states at depth \( i \) of the minimal trellis is \( 2|A_i|, i = 1, 2, \ldots, N \). The \( N + 1 \)-tuple \( \eta = (\eta_0 \eta_1 \ldots \eta_N) \), where \( \eta_0 = 0 \) and \( \eta_i = |A_i|, i = 1, 2, \ldots, N \), is called the trellis state complexity profile. The maximal state complexity is \( \eta_{\text{max}} = \max_{1 \leq i \leq N} \eta_i \), and \( 1 \leq \eta_{\text{max}} \leq \min\{K, N - K\} \) [26].

At each position \( i \), at most one row of minimal-span \( G \) can become active. Denote such a row (if it exists) with \( g^* \), and the corresponding information bit that enters the encoder with \( u^* \). Then a branch connecting a state \( \sigma_{i-1} \) at depth \( i-1 \) with a state \( \sigma_i \) at depth \( i, 1 \leq i \leq N \), is labelled with a codesymbol \( v_i \), which is determined by

\[
v_i = \begin{cases} \sigma_{i-1} \cdot (G)_i + u^*, & \text{if } g^* \text{ exists} \\ \sigma_{i-1} \cdot (G)_i, & \text{otherwise} \end{cases}
\]

where \( (G)_i \) denotes the \( i \)-th column of \( G \). Note that in a minimal trellis there are at most two branches arriving to or leaving from each node, and they always carry opposite codesymbols. More precisely: if a row starts at position \( i \), then all nodes at depth \( i - 1 \) branch into 2 children nodes at depth \( i \); if a row ends at position \( i \), then pairs of nodes at depth \( i - 1 \) merge into one child at depth \( i \); otherwise (if no row starts or ends at depth \( i \)), every node at depth \( i - 1 \) is connected to exactly one child each at depth \( i \), with the same state label. In a code tree obtained from the minimal trellis, there are no state mergers, that is, each node has a unique parent (and at most two children).

When performing decoding with the weighted Hamming metric, a branch labelled by (6) is assigned the metric

\[
\begin{cases} 
\mu(v_i, h_{\text{chi}}) + \mu(u^*, h_a^*), & \text{if } g^* \text{ exists} \\
\mu(v_i, h_{\text{chi}}), & \text{otherwise.}
\end{cases}
\]

If the information bits are equiprobable, the branch metric reduces to \( \mu(v_i, h_{\text{chi}}) \). The metric of a path from the root to the toor, corresponding to the codeword \( v \), is then \( w = \mu(v, h_{\text{chi}}) + \mu(u, h_a) \).

A decoder operating on the minimal trellis/tree can be used when the encoding matrix is any generator matrix equivalent to the minimal-span form matrix. For systematic encoding, the starting positions of the rows in the minimal-span generator matrix correspond to the systematic positions in the equivalent systematic matrix. In such a case, at each systematic position \( i \), the information bit \( u^* \) in the \( a \text{ priori} \) metric term in (7) is replaced by the codesymbol \( v_i \).

## III. BEAST-APP DECODING

BEAST was introduced in [27] as an efficient method for searching codes' distance spectra [28]. Maximum-likelihood decoding of block codes using BEAST was presented in [6], and an extension of BEAST to list decoding was introduced in [8]. In this section, list-based symbol decoding using BEAST is presented. For the sake of completeness, a brief description of BEAST for list decoding is given.

### A. BEAST Algorithm

Let \( \xi \) denote a node in the code tree and let \( \xi^p \) be its parent node (note that in a code tree, unlike in a trellis, every node has a unique parent). Every node is characterized by three parameters: state \( \sigma(\xi) \), depth \( \ell(\xi) \), and weight \( w(\xi) \). In a forward tree (indicated by subscript \( F \)), the depth and the weight of a node \( \xi \) are equal to the length and the cumulative weighted Hamming metric (cf. (7), respectively, of a path starting at the root and arriving at the node \( \xi \). Analogously, the backward tree (indicated by subscript \( B \)) stems from the toor, and the node weights and the lengths are calculated with respect to the toor. BEAST finds a list of the \( L \) best (most probable) codewords in the following steps:

1) **Initialization**: Sort the sequence \( \{|L_{ch}(v_i)| + |L_{ch}(u_i)|\} \), \( 1 \leq i \leq N \), in increasing order (where \( L_{ch}(u_i) = 0 \) for positions \( i \) where no rows of \( G \) become active) and denote this sorted sequence with \( \{\tau_i\} \). Set \( T = \left\lceil d_{\text{min}}/2 \right\rceil \). Initialize the metric threshold \( T \) with \( T = \sum_{i=1}^L \tau_i \). Initialize the forward and backward trees by the zero-state node of zero weight. Initialize the list \( L \) of found codewords with an empty list.

2) **Forward search**: Extend the forward tree to find the following set of nodes

\[ \mathcal{F} = \{\xi \mid w_F(\xi) \geq T/2, w_F(\xi^p) < T/2\} \]

In other words, every node whose weight is below \( T/2 \) is extended (that is, its children and their metrics are determined). A child node whose weight exceeds \( T/2 \) is placed in the set \( \mathcal{F} \) and it is not extended further.

---

1. The BCJR trellis [12] constructed from a parity check matrix of a code is isomorphic to the minimal trellis constructed from the minimal-span generator matrix.
3) **Backward search:** Extend the backward tree to find the following set of nodes

\[ B = \{ \xi | w_B(\xi) \leq T/2 \}. \]

In the backward direction, every node whose weight does not exceed \( T/2 \) is extended and included in the set \( B \). Nodes with weights larger than \( T/2 \) are neither extended nor stored in \( B \). Thus, the set \( B \) contains only the inner nodes of the partially explored backward tree (not the leaves).

4) **Matching:** Find all pairs of nodes \( (\xi, \xi') \in \mathcal{F} \times B \) such that \( \sigma(\xi) = \sigma(\xi') \) and \( \ell_B(\xi) + \ell_B(\xi') = N \). Each such match describes a unique codeword with the weight \( w = w_F(\xi) + w_B(\xi') \). Discard all candidates that do not fulfill \( w \leq T \) (note that without this condition the chosen codewords are not necessarily the globally best candidates, see Example 1 below).

5) Add the found codewords (if any) to the list \( L \) in a sorted manner, according to their metrics \( w \). The best codeword has the smallest \( w \). If the updated list has size \( L \) output the \( L \) best codewords with their metrics and stop. Otherwise, set \( t \leftarrow t + 1 \), increment the threshold \( T \leftarrow T + n_t \), and go to step 2 to continue the search.

### B. Forming the Soft Outputs

The soft reliability of a symbol \( v_i \), produced by a SISO symbol decoder, is defined as the *a posteriori* L-value

\[
L(v_i) = \log \frac{p(v_i = 0 | r)}{p(v_i = 1 | r)} = \log \frac{\sum_{v \in \mathcal{C}: v_i = 0} p(v)}{\sum_{v \in \mathcal{C}: v_i = 1} p(v)},
\]

Since the logarithmic APP \( \log p(v | r) \) is proportional to the negative weighted Hamming metric \( -\mu(v, h_{ch}) - \mu(u, h_a) \), (8) can be written as

\[
L(v_i) = \max_{v \in \mathcal{C}: v_i = 0} \{-\mu(v, h_{ch}) - \mu(u, h_a)\}
- \max_{v \in \mathcal{C}: v_i = 1} \{-\mu(v, h_{ch}) - \mu(u, h_a)\}
\]

where \( \max^* \) operator denotes the logarithm of the sum of exponentials, \( \max^*_{1 \leq i \leq n} \{a_i\} \triangleq \log \left( \sum_{i=1}^n e^{a_i} \right). \)

Consider the list \( L \) of the best (most probable) codewords found by BEAST. Already a small list aggregates almost all the probability (cf. [29]), that is, the contribution of the codewords outside the list to (9) is negligible. Thus, the L-value (9) can be approximated by considering only the metrics of the codewords on the list \( L \), instead of the whole code \( C \), where \( |L| \ll |C| \):

\[
\tilde{L}(v_i) = \begin{cases} 
\max^*_{v \in L} \{-\mu(v, h_{ch}) - \mu(u, h_a)\} + \mu(v_L, h_{ch}) + \mu(u_L, h_a), & \text{if } v_L = 0, \ell = 1, 2, ..., L \\
-\mu(v_L, h_{ch}) - \mu(u_L, h_a) - \max^*_{v \in L} \{-\mu(v, h_{ch}) - \mu(u, h_a)\}, & \text{if } v_L = 1, \ell = 1, 2, ..., L \\
\end{cases}
\]

Otherwise, proceed to the next stored forward depth.

Consider the (8, 4, 4) extended Hamming code whose minimal-span generator matrix is (cf. [30])

\[
G = \begin{pmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix}.
\]

The corresponding minimal trellis is shown in Fig. 1. The maximal state complexity is \( \eta_{\max} = 3 \).

Assume that the (8, 4, 4) code and BPSK signaling were used for transmission over the AWGN channel, and let the received sequence be

\[
r = (0.18 - 0.56 0.91 0.60 - 0.02 1.60 2.80 0.34).
\]

Suppose we are interested in computing approximate APP L-values based on the \( L = 5 \) most probable codewords. The hard-decision received sequence is

\[
h_{ch} = (0 1 0 0 1 0 0 0).
\]
Assume that the information bits are \textit{a priori} equiprobable, \textit{i.e.}, $L_{0}(u_k) = 0$, $k = 1, 2, 3, 4$. The received values can then directly be used as the symbol reliabilities, that is, $L_{ch}(v_i) = r_i$, $i = 1, 2, ..., 8$ (scaling with the factor $4/N_0$ is not necessary). The sorted sequence of absolute channel reliabilities $|L_{ch}(v_i)|$ is

\[
\{r_i\}_{i=1}^{8} = \{0.02, 0.18, 0.34, 0.56, 0.60, 0.91, 1.60, 2.80\}.
\]

The metric threshold is initialized with $T_1 = 0.02 + 0.18 = 0.20$. Starting from the root and the toor, respectively, we grow a forward and a backward tree with the target metric $T_1/2 = 0.10$, as illustrated in Fig. 2. In the forward tree, 4 nodes (white circles) are visited (extended) and the following 4 leaf-nodes (black circles) are stored in the set $F$:

\[
F_{\sigma} = \{0000, 0000, 0100, 0110\}\quad F_{\omega} = \{0.18, 0.56, 0.91, 0.60\}\quad F_{\ell} = \{1, 2, 3, 4\}.
\]

In the partially explored backward tree, illustrated in Fig. 2, the 7 visited nodes (black circles) have metrics $w_B \leq T_1/2 = 0.10$ and are stored in the set $B$. The leaves, connected with dashed lines to their parents, have metric exceeding $T_1/2$ and are thus not extended. The elements of the set $B$ are

\[
B_{\sigma} = \{0000, 0000, 0000, 0000, 0000, 0000, 0000\}\quad B_{\omega} = \{0.00, 0.00, 0.00, 0.00, 0.02, 0.02, 0.02\}\quad B_{\ell} = \{0, 1, 2, 3, 4, 5, 6\}.
\]

Finally, we perform matching of the sets $F$ and $B$. We find that there is only one matching node at the forward depth $\ell_F = 2$:

\[
M_{\sigma} = \{0000\};\quad M_{\omega} = \{2\};\quad M_{\ell} = \{0.58\}
\]

which corresponds to the all-zero codeword. However, its total weight is above the current threshold, $w = w_{F} + w_{B} = 0.56 + 0.02 = 0.58 > T_1$; hence, this need not be the best codeword, and we do not accept it in the list. Instead, we increase the threshold and continue the search.

In the next step, the threshold becomes $T_2 = T_1 + 0.34 = 0.54$, and we extend the forward tree from the previous step.

Fig. 1. Minimal trellis (with $K$-bit state labels) for the $(8, 4, 4)$ extended Hamming code with the generator matrix (12).

Fig. 2. The forward and the backward tree explored by BEAST with the metric threshold $T_1 = 0.20$. There is one matching node (highlighted); however, its total metric is above $T_1$ and, hence, it is not accepted.

Fig. 3. The forward and the backward tree explored by BEAST with the metric threshold $T_3 = 1.10$. There are four matching nodes. Only one of them has the total metric below $T_3$ and it determines the best (ML) path.
The stored nodes are:

\[ F_\sigma = \{0000, 1100, 1000, 0100, 0110, 0000, 0001\} \]

\[ F_w = \{0.56, 0.74, 1.09, 0.91, 0.60, 1.78, 3.00\} \]

\[ F_\ell = \{2, 3, 4, 6, 7\} \]

The backward tree with the target metric \( w_B \leq T_2/2 = 0.27 \) is the same as in the previous step (Fig. 2), i.e., no node can be extended. The matching yields two nodes:

\[ M_\sigma = \{0000, 0000\}; \quad M_\ell = \{2, 6\}; \quad M_w = \{0.58, 1.78\} \]

The backward tree with the target metric \( w_B \leq T_2/2 = 0.27 \) is the same as in the previous step (Fig. 2), i.e., no node can be extended. The matching yields two nodes:

\[ M_\sigma = \{0000, 0100, 0000, 0001\} \]

\[ M_w = \{0.58, 1.25, 1.78, 3.34\} \]

\[ M_\ell = \{2, 3, 6, 7\}. \]

Since only the first of the four matching nodes has total metric below \( T_3 = 0.56 \), this node determines the best (ML) path: \( v_1 = 00000000 \), and it is entered at the first position in the codeword list \( L \). The number of nodes \( BEAST \) visited in order to find the ML path is 23 (11 in the forward and 12 in the backward tree). The Viterbi algorithm finds the ML path by visiting all 34 nodes in the code trellis.

In order to populate the list with the remaining 4 codewords, we need to increase the threshold again and continue the search. The list is filled when the threshold is increased up to \( T_5 = \sum_{i=1}^{\infty} \tau_i = 2.61 \). The codewords in the list and their metrics are listed in Table I. The approximate list-based APP L-values \( L(v_i) \) computed according to (11) are shown in Table II, and compared to the exact values \( L(v_i) \) (obtained with the list size \( L = |C| = 16 \)) in terms of the relative discrepancy

\[ \delta_i = \frac{|\tilde{L}(v_i) - L(v_i)|}{L(v_i)} . \tag{13} \]

Note that on the symbol position 7, the 5 best codewords agree on the value \( v_7 = 0 \). For this symbol, the approximate L-value is obtained by using the last listed metric, \( w = 1.78 \), as a metric of a “virtual” codeword with \( v_7 = 1 \). \( L(v_i) \) is computed according to (11). This yields

\[ \tilde{L}(v_7) = \max^* \{-0.58, -1.25, -1.68, -1.71, -1.78\} + 1.78 = 0.3237 + 1.78 = 2.1037. \]

This is the most reliable symbol position, with the largest magnitude \(|\tilde{L}(v_7)|\).

Finally, we note that, when performing the search, accepting only the codewords with the metric below the threshold ensures that the list is populated by the globally best candidates. For example, in the third step we found four matches, however, these are not the four best codewords, as it is seen from Table I.

### D. Decoding Complexity

The complexity of BEAST is determined by the number of nodes the algorithm visits (extends) during its search for the \( L \) best codewords. These are the interior nodes of the partially explored forward and backward trees; there are in total about \(|F| + |B| \) of them. Visiting a node implies determining its children nodes, computing their metrics, and comparing them to the threshold, which is equivalent to the add-compare-select operation for each trellis state in the Viterbi algorithm. Memory requirements are also determined by the number \(|F| + |B| \), since this is the size of the stored node sets. Matching of the sets negligibly contributes to the total complexity, since the sets are already sorted according to their depth, and, thus, to perform matching, it is sufficient to sort only the states of the subsets of nodes with the matching depths. Moreover, the sorting operations are less computationally complex than node extensions. Therefore, we conclude that the total complexity is governed by the number \(|F| + |B| \) and use it for the assessment of BEAST’s efficiency. Since BEAST operates on a tree, not a trellis, the mergers of the nodes that occur in the trellis are not taken into account. Hence, there may exist “duplicate” nodes (corresponding to the same state) among the visited nodes. This, however, rarely occurs for good short block codes, for which efficiency of BEAST is superior to that of the trellis-based algorithms, cf. [25].

The total number of nodes in the forward and backward trees directly depends on the number of states in the corresponding minimal trellis. For given code parameters, in order to achieve low decoding complexity (particularly for iteratively decoded concatenated codes), the codes should be chosen such that they have good complexity profile \( \eta \) and low maximal state complexity \( \eta_{\text{max}} \). BCH codes are often used as component codes for building product codes ([19], [31], [22]). Consider, for example, the \( (31, 21, 5) \) BCH code and its extended version \( (32, 21, 6) \). These codes are not optimal in terms of trellis complexity. The maximal state complexities are \( \eta_{\text{max}} = 10 \) and \( \eta_{\text{max}} = 11 \), respectively. Instead, we propose alternative codes, with the same minimum

---

**TABLE I**

<table>
<thead>
<tr>
<th>Ranking on the list ( \ell )</th>
<th>Codeword ( v_\ell )</th>
<th>Metric ( \mu(v_\ell, h_{\text{MLE}}) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>00110010000</td>
<td>0.58</td>
</tr>
<tr>
<td>2</td>
<td>0110100100</td>
<td>1.25</td>
</tr>
<tr>
<td>3</td>
<td>1001100011</td>
<td>1.68</td>
</tr>
<tr>
<td>4</td>
<td>1111000000</td>
<td>1.71</td>
</tr>
<tr>
<td>5</td>
<td>1100111000</td>
<td>1.78</td>
</tr>
</tbody>
</table>

**TABLE II**

<table>
<thead>
<tr>
<th>Symbol pos.</th>
<th>List-based ( L(v_i) )</th>
<th>Exact ( L(v_i) )</th>
<th>Discrepancy ( \delta_i )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.45709</td>
<td>0.48419</td>
<td>0.0560</td>
</tr>
<tr>
<td>2</td>
<td>0.15988</td>
<td>0.03491</td>
<td>2.6413</td>
</tr>
<tr>
<td>3</td>
<td>0.67170</td>
<td>0.67681</td>
<td>0.0075</td>
</tr>
<tr>
<td>4</td>
<td>1.01667</td>
<td>0.76092</td>
<td>0.3361</td>
</tr>
<tr>
<td>5</td>
<td>0.14385</td>
<td>0.21823</td>
<td>0.3408</td>
</tr>
<tr>
<td>6</td>
<td>1.97363</td>
<td>1.43293</td>
<td>0.3867</td>
</tr>
<tr>
<td>7</td>
<td>2.10374</td>
<td>2.71169</td>
<td>0.2242</td>
</tr>
<tr>
<td>8</td>
<td>0.65395</td>
<td>0.45845</td>
<td>0.4264</td>
</tr>
</tbody>
</table>
E. Quality of Soft Outputs

Quality of the soft outputs produced by the BEAST-APP decoder are assessed in terms of the average relative discrepancy \( \delta \) (13) with respect to the true L-values (where averaging is performed over all symbol positions), and compared to the soft-output of the max-log-MAP decoder. Max-log-MAP decoding can be viewed as a special case of list-based APP decoding, which produces approximate L-values using a list of \( L = 2 \) codewords. One of the two codewords on this list is always the best (most probable) codeword. The second codeword, however, is, in general, different for each symbol position, and it is chosen as the best codeword among all the codewords that have the opposite symbol value at that position. Hence, the outputs of a max-log-MAP decoder are given by

\[
\widetilde{L}_{\text{max-log}}(v_i) = \max_{v \in C_{v_i} \setminus \{u, h_a\}} \{-\mu(v, h_{ch}) - \mu(u, h_a)\} - \max_{v \in C_{v_i}} \{-\mu(v, h_{ch}) - \mu(u, h_a)\}. \tag{15}
\]

Both codewords considered by the max-log-MAP decoder are included in the list of \( L \geq 2 \) best codewords found by BEAST, for each symbol position where at least one codeword on this list disagrees with the others on the symbol value.

Figures 7 and 8 illustrate the discrepancy \( \delta \) for the \((24, 12, 8)\) extended Golay code and the \((30, 20, 5)\) code, respectively, for several list sizes. For the rate \( R = 1/2 \) Golay code, the BEAST-APP decoder with list size \( L = 5 \) provides higher accuracy of the L-values than the max-log-MAP decoder. The \((30, 20, 5)\) code has higher rate and worse minimum distance than the Golay code and thus has increased levels of \( \delta \). It is worth emphasizing that, for BEAST-APP decoding, the average discrepancy \( \delta \) is dominated by the most reliable symbol positions, for which all the codewords agree.

The error in approximating these L-values using the rule (11) is larger than for the remaining symbols. This effect is evident in Fig. 8, where the residual error \( \delta \) at high SNR stays above the error of the max-log-MAP approximation, which is due to the fact that the best L codewords agree on a large number of positions—on about 42% of the symbol positions (in contrast to about 20% for the extended Golay code).

IV. ITERATIVE BEAST-APP DECODING

In this section, BEAST-APP decoders are used as component decoders in an iterative decoding scheme for product codes (which are also referred to as turbo product codes [21] or block turbo codes [19]).
Fig. 7. Average relative discrepancy $\delta$ between the true and the approximate APP $L$-values for max-log-MAP and BEAST-APP decoding of the $(24, 12, 8)$ ext. Golay code.

Fig. 8. Average relative discrepancy $\delta$ between the true and the approximate APP $L$-values for max-log-MAP and BEAST-APP decoding of the $(30, 20, 5)$ code.

Fig. 9. Codeword of a product code of two constituent codes, $(N_1, K_1, d_{\text{min}1})$ and $(N_2, K_2, d_{\text{min}2})$, with systematic encoding.

Codewords of an $(N_1N_2, K_1K_2, d_{\text{min}1}d_{\text{min}2})$ product code $C_1 \times C_2$ can be represented as two-dimensional $N_2 \times N_1$ arrays, such that each row is a codeword of $C_1$ and each column is a codeword of $C_2$. If systematic encoders are used for both $C_1$ and $C_2$, such that the first $K_1$ and $K_2$ codesymbols, respectively, are systematic, the codewords of the resulting product code are in the systematic form shown in Fig. 9. By removing the $(N_2 - K_2) \times (N_1 - K_1)$ subblock of double parity check symbols, an incomplete product code is obtained, of rate $R = K_1K_2/(N_1N_2 - (N_1 - K_1)(N_2 - K_2))$, and minimum distance $d_{\text{min}} \geq d_{\text{min}1} + d_{\text{min}2} - 1$. Product code is a serial code concatenation, while an incomplete product code is a parallel concatenation of constituent block codes, with a block interleaver.

We consider concatenations of two identical $(N, K, d_{\text{min}})$ component codes; the resulting product code is denoted $(N, K, d_{\text{min}})^2$. The component BEAST-APP decoders are identical. Decoding of incomplete product codes is discussed first. The structure of the iterative decoder is shown in Fig. 10. One decoding iteration consists of one row-wise and column-wise decoding (the order is not relevant). Let the first decoder operate on the rows, and let $v_s$ denote the block of $K^2$ systematic bits. The row-wise BEAST-APP decoder finds the predefined number $L$ of the most probable codewords for each row, and produced approximate APP $L$-values $L^{(1)}(v_s)$, according to (11). The extrinsic $L$-values, given by

$$L^{(1)}_{\text{ext}}(v_s) = L^{(1)}(v_s) - L_{\text{ch}}(v_s) - L^{(1)}_{\text{a}}(v_s)$$

(16)

are passed on to the column-wise decoder. It is observed by simulations that the convergence of the iterative decoder improves significantly if the extrinsic outputs are attenuated in each iteration. Hence, $a \ priori$ inputs to the column-wise decoder are

$$L^{(2)}_a(v_s) = \alpha \pi \left( L^{(1)}_{\text{ext}}(v_s) \right)$$

(17)

where $\pi(\cdot)$ denotes block-interleaver permutation and $\alpha$ is an appropriately chosen scaling factor which will be discussed in Section V. The column-wise BEAST-APP decoder computes

![Iterative decoder for incomplete product codes.](image)

Fig. 10. Iterative decoder for incomplete product codes.
the L-values $\tilde{L}^{(2)}(v_s)$ and delivers the extrinsic outputs

$$L^{(2)}_{\text{ext}}(v_s) = \tilde{L}^{(1)}(v_s) - \pi(L_{\text{ch}}(v_s)) - L^{(1)}_{\text{a}}(v_s)$$

(18)

which are, after deinterleaving and scaling used as a priori inputs of the row-wise decoder in the next iteration,

$$L^{(1)}_{\text{a}}(v_s) = \alpha^{-1} \left( L^{(2)}_{\text{ext}}(v_s) \right).$$

(19)

In the final iteration, the column-wise decoder outputs hard decisions of the information bits $v_s$.

The structure of the iterative BEAST-APP decoder for product codes is essentially the same as in Fig. 10, except that both decoders operate on all codesymbols, that is, they exchange extrinsic L-values of both systematic and parity-check symbols. Hence, in relations (16)–(19), $v_s$ needs to be replaced by $v$. In this case, the a priori L-values of all codesymbols are incorporated into codeword metrics by modifying the branch metric (7) as

$$\mu(v_i, h_{\text{chi}}) + \mu(v_i, h_{\text{ai}}), \quad i = 1, 2, ..., N$$

(20)

where $h_{\text{chi}}$ and $h_{\text{ai}}$ are the channel and a priori codeword decisions, respectively.

V. CONVERGENCE BEHAVIOR OF THE BEAST-APP DECODER

Extrinsic information transfer (EXIT) charts are a powerful tool for investigating the convergence behavior of iterative decoding schemes [33], [34]. Provided that the interleaver length is large enough, EXIT charts can predict the decoder’s bit-error-rate (BER) performance. Since product codes are built using relatively short block interleavers, this tool becomes somewhat inaccurate; nevertheless, the EXIT charts indicate the lower bounds on the achievable BER, while the true decoder’s performance can be visualized by the decoding trajectories on the mutual information chart.

In this section, we use the EXIT charts and the decoding trajectories to investigate the behavior of the BEAST-APP decoder and compare it with the BCJR (log-MAP) and max-log-MAP decoders. The use of the extrinsic attenuation factor will be motivated.

To measure the information contents of the L-values at the decoder input and output, we use the mutual information between the transmitted bits and the L-values, in particular, $I_a \equiv I(u; L_a(u))$, $I_{\text{ext}} \equiv I(u; L_{\text{ext}}(u))$, and $I_{\text{app}} \equiv I(u; L(u))$. As an example, we consider the rate $R = 1/2$ $(30, 20, 5)^2$ incomplete product code, whose constituent-code minimal-span matrix is given by $G_1$ from (14). The EXIT charts of this code for signal-to-noise ratios (SNRs) $E_b/N_0 = 1$ dB and $2$ dB, are shown in Figures 11 and 12, respectively. Note that the transfer curves cross before the $(1, 1)$ point. The decoding trajectories visualize the exchange of extrinsic information throughout the iterative process. These are average trajectories, obtained by averaging the extrinsic mutual information over 10000 data blocks (one block has $K^2 = 400$ bits). The trajectory of the BCJR decoder follows the predicted transfer function only in the first few iterations; due to the short block interleaver, the trajectory “dies out” before the two curves intersect. Due to the statistical dependencies between the messages in later iterations, the prediction of the BER from the EXIT chart [34] does not correspond to the true BER. The trajectories of the two component suboptimal decoders indicate a worse performance than for the BCJR. At $E_b/N_0 = 2$ dB (Fig. 12), the max-log-MAP trajectory follows the BCJR more closely than at $1$ dB (Fig. 11), which is expected since at higher SNR the max-log approximation is more accurate. Before analyzing the BEAST-APP trajectories in more detail, we briefly discuss the use of the extrinsic attenuation factor.

A. The Scaling of Extrinsic Outputs

In iterative schemes employing suboptimum soft symbol decoders, extrinsic outputs of component decoder are commonly scaled by an attenuation fudge factor $\alpha$, $0 \leq \alpha \leq 1$, see [19], [22], [36], [37].

\[^3\text{Terminology suggested by Peter Massey [35].}\]
The influence of $\alpha$ on the decoder performance is illustrated in Fig. 13, which shows the decoding trajectories of the BEAST-APP decoder (left) and the BCJR iterative decoder (right), with and without the fudge factor, for an SNR level of 1 dB. Without attenuation of the extrinsic feedback, the BEAST-APP decoder has poor convergence causing the BER and the trajectory to oscillate during iterations (similar behavior is observed with the max-log-MAP decoder as well). On the other hand, attenuating the BCJR outputs has the opposite effect—it slows the convergence down and leads to a poorer performance [15]. For suboptimum component decoders, this attenuation is necessary to compensate for the approximations in the computations of the LLRs. The values of the fudge factor were chosen heuristically. In general, it was observed by simulations that the values should increase during the iterative process, as the soft outputs become more reliable. We have used 10 decoding iterations and found that good BER performance is obtained with the values

$$\alpha = (0.3 \ 0.4 \ 0.5 \ 0.5 \ 0.6 \ 0.6 \ 0.7 \ 0.7 \ 0.8 \ 0.9).$$

These values are used for both the BEAST-APP and the max-log-MAP decoder.

B. Trajectories of the BEAST-APP Decoder

Consider the trajectories of the BEAST-APP decoder shown in Figures 11 and 12. After the first row-wise decoding (the point $(0, I_{ext1})$ on the charts) we observe a loss of almost half of the extrinsic mutual information compared to the BCJR output. This loss, however, does not imply that the BEAST-APP decoder suffers from a much higher BER than the BCJR; on the contrary, simulations (cf. Section VI) show that the BEAST-APP decoder of the $(30, 20, 5)^2$ code has virtually the same BER performance as the BCJR in the first iteration. The loss in extrinsic information occurs, in fact, only on the most reliable symbol positions (for which all the codewords on the list agree on the symbol value), which is due to the fact that the $L$-value approximation (11) is less accurate on these positions. In terms of the a posteriori mutual information, however, the approximation (11) is rather good, that is, it yields near-BCJR mutual information on all symbol positions: $I(u, L(u)) \approx I(u, \tilde{L}(u))$. We conclude that, for the most reliable positions, the extrinsic $L$-values given by $\tilde{L}_{ext}(u) = L(u) - L_{ch}(u) - L_a(u)$ are not optimal in terms of preserving information. This opens the possibility for further improvements of the extrinsic estimates $L_{ext}(u)$. For example, we have observed in simulations that the performance of the iterative BEST-APP decoder slightly improves if the component decoders modify the computation of the extrinsic outputs as

$$\tilde{L}_{ext}(u) = \begin{cases} L(u), & \text{if } L \text{ codewords agree on } u, \\ L(u) - L_{ch}(u) - L_a(u), & \text{otherwise}. \end{cases}$$

Since the extrinsic decoding trajectories are inconsistent with the true BER performance of the iterative BEAST-APP decoder, we consider instead the trajectory of the a posteriori information (API), which depicts the evolution of the decoding process and reflects the BER throughout the iterations.

Fig. 14 shows the API trajectory of the three decoders. Now we see that the max-log-MAP and the BEAST-APP decoders perform similarly, and follow the BCJR trajectory closer than in the previous EXIT charts, which corresponds to the simulated BER performances (cf. Section VI). The average $I_{app}$ levels of the BEAST-APP and the BCJR decoders after the first half-iteration are almost the same. To gain insight into the performance in the last iterations, the zoomed-in version of the API chart is given in Fig. 15. The BCJR decoder reaches the furthest point, and thus also the lowest BER. The performances of the BEAST-APP and the max-log-MAP decoder are comparable, and they both require the use of an extrinsic fudge factor.

VI. PERFORMANCE EVALUATION

To verify the behavior discussed in the previous section, we simulated the three iterative decoders for the $(30, 20, 5)^2$ both incomplete and complete product codes. The number of
iterations was limited to 10, the list size for the BEAST-APP decoding was \( L = 10 \), and the extrinsic outputs are attenuated by the fudge factor given by (21). Fig. 16 shows the bit error rate performance of the iterative BEAST-APP, max-log-MAP, and BCJR decoders for the incomplete product code. The performances are virtually the same in the first iteration. After ten iterations, the BEAST-APP decoder outperforms the max-log-MAP decoder, and reaches the BER of 6-7 BCJR iterations. The decoding complexities, expressed in terms of the number of visited nodes in the code tree/trellis, are illustrated in Fig. 17. Note that both the BCJR and the max-log-MAP decoder visit all nodes in the trellis twice; hence, in terms of visited nodes, their complexities are the same. The difference lies in the computational complexity of the operations performed at each node, which is higher for the BCJR decoder. The operations that BEAST performs at each tree node are equivalent to those of the max-log-MAP algorithm. On average, BEAST visits much fewer nodes than the max-log-MAP (or BCJR) decoder—10 iterations of BEAST-APP decoding are computationally as costly as only one BCJR iteration, while attaining much lower BER. The complexity of BEAST slightly increases with the SNR, since for a good channel, more nodes need to be visited to find the \( L = 10 \) best codewords (since there are only one or two good candidates and all the others are essentially equally bad).

The BER performance of the iterative BEAST-APP decoder for the \((30, 20, 5)^2\) complete product code is shown in Fig. 18, while the corresponding decoding complexities are illustrated in Fig. 19. The BEAST-APP decoder achieves near-BCJR performance with 4-5 times lower complexity. Note that the complete product code has better performance than its incomplete counterpart, due to lower rate and better distance properties. The average BEAST decoding complexity for a product code (Fig. 19) is always somewhat higher than for the corresponding incomplete code (Fig. 17), due to higher path metrics (since \textit{a priori} reliabilities of both the systematic and the parity bits are iteratively updated and taken into account when performing the search).

We have also tested BEAST-APP decoding when the list of \( L \) codewords is not necessarily optimal \( (i.e., \text{does not}\)
necessarily contain the $L$ best candidates). Such a list is obtained by omitting the last part of step 4 in BEAST (cf. Section III-A), where the found codewords are accepted only if their metric is below the current threshold. By removing this condition, the list is filled quicker and the number of visited nodes is about a half of that for the optimum-list BEAST from Fig. 19. The performance loss due to this loss of list optimality is negligible.

At the cost of a small rate loss, the BER performance is significantly improved if the $(33, 21, 6)^2$ product code is used instead of the $(30, 20, 5)^2$ code, while their BEAST-decoding complexities are virtually the same. The performances of both codes after 10 iterations of BEAST-APP decoding are shown in Fig. 20 and compared to two alternative list-based SISO decoding methods: the decoder from [22], and the Chase-based decoder from [19], both used for the $(32, 21, 6)^2$ extended BCH code. Both simulation results are taken directly from [22]. Note that the asymptotic coding gain of the $(33, 21, 6)^2$ product code over the $(32, 21, 6)^2$ extended BCH product code is $10 \log_{10}(33^2/32^2) = 0.267$ dB. At BER $< 10^{-5}$, the BEAST-APP decoder for the $(33, 21, 6)^2$ code achieves a gain of $0.15 - 0.2$ dB over the decoder of [22] for the $(32, 21, 6)^2$ code. Both methods outperform the decoder of [19].

The core of the Chase-based decoder of [19], which determines its complexity, is the hard-decision algebraic decoder that is used $L$ times to produce a list of $L$ candidate codewords. This scheme is therefore used in conjunction with code structures (such as BCH codes) that allow efficient algebraic decoding. BEAST, on the other hand, does not require a particular code structure for its tree search. Thus, a direct complexity comparison of the two schemes is not feasible on a general level.

Similarly as BEAST, the decoder from [22], which outperforms the decoder of [19], also does not rely on a particular code structure. It operates on the equivalent, reliability-based reordered code and generates the list of candidate codewords by encoding the most reliable information test patterns. The performance in Fig. 20 is achieved using 106 test patterns for each row/column of the product code. Although this is substantially more than $L = 10$ codewords that BEAST needs, the complexity of finding these test patterns is relatively low, as it involves simple manipulations of the parity check matrix of the component code (which does not require floating-point operations). The two algorithms, with the above specified parameters, have approximately the same complexity.

VII. SUMMARY AND CONCLUSIONS

An efficient tree-based algorithm for soft-input soft-output decoding of block codes was presented. Soft symbol reliabilities are obtained from a list of the $L$ most probable codewords, found by tree searches. The decoding complexity, determined by the number of visited nodes in the code trees, depends on the trellis complexity profile of the code. Two low-complexity block codes were presented: both are suitable for concatenated code constructions. It was shown that for block codes of relatively short length and high trellis complexity, the BEAST-APP decoder has substantially lower complexity than trellis-based decoding algorithms, while achieving very good error rate performance. This makes the BEAST-APP decoder a favorable low-complexity alternative to the BCJR decoder in the iterative schemes for decoding concatenated block codes.

![Fig. 18. Bit error rate for $(30, 20, 5)^2$ product code (rate $R = 4/9$), for iterative decoding with BCJR (dashed lines) and BEAST-APP (solid lines) decoder with the list size $L = 10$.](image1)

![Fig. 19. Average decoding complexity per bit for $(30, 20, 5)^2$ product code, for iterative BCJR and BEAST-APP decoder with the list size $L = 10$.](image2)

![Fig. 20. Bit error rate of the BEAST-APP decoder with the list size $L = 10$, for $(30, 20, 5)^2$ and $(33, 21, 6)^2$ product codes, compared to decoding schemes from [22] and [19] for the $(32, 21, 6)^2$ extended BCH product code (both results are taken from [22]).](image3)
In particular, iterative BEAST-APP decoding was applied to both product codes and incomplete product codes. The convergence behavior was investigated using mutual-information decoding trajectories and compared to BCJR decoding. It was shown by simulations that using a small list of the best codewords, the BEAST-APP decoder outperforms existing algebraic-type decoders and achieves near-BCJR performance with significantly lower complexity.

REFERENCES

Maja Lončar was born in Belgrade, Serbia (former Yugoslavia), in 1975. She received the Dipl.-Ing. degree from the Faculty of Electrical Engineering, University of Belgrade, in 2000, and the Ph.D. degree in information theory from Lund University, Lund, Sweden, in 2007.

From 2000 to 2003 she was with Telecommunications Research Centre Vienna (f/tw.), Austria, where she worked as a researcher in the field of mobile communications. During February–March 2002, she was a guest researcher at NTT DoCoMo Wireless Laboratories, Japan. Since 2003 she has been with the Department of Electrical and Information Technology, Information Theory Group, at Lund University. Her research interests include coding theory and efficient detection and decoding algorithms.

Rolf Johannesson (S’72–M’75–SM’96–F’98) was born in Hässleholm, Sweden, on July 3, 1946. He received the M.S. and Ph.D. degrees in 1970 and 1975, respectively, both from Lund University, Lund, Sweden.

Since 1976, he has been with Lund University where he is now Professor of Information Theory. His scientific interests include information theory, error correcting codes, and cryptography. In addition to papers and book chapters in the area of convolutional codes and cryptography, he has authored two textbooks on switching theory and digital design and one on information theory and coauthored Fundamentals of Convolutional Coding (Piscataway, NJ: IEEE Press, 1999), and Understanding Information Transmission (New York: IEEE Press/Wiley, 2005).

Prof. Johannesson became a Fellow of the IEEE in 1998 and he was awarded the degree of Professor honoris causa from the Institute for Information Transmission Problems, Russian Academy of Sciences, Moscow, Russia, in 2000. He was elected a member of the Royal Swedish Academy of Engineering Sciences in 2006.

Irina E. Bocharova was born in Leningrad, U.S.S.R. (now St.-Petersburg, Russia), on July 31, 1955. She received the Diploma degree in electrical engineering in 1978 from the Leningrad Electrotechnical Institute and the Ph.D. degree in technical sciences in 1986 from the Leningrad Institute of Aircraft Instrumentation.

Since 1986, she has been a Senior Researcher, an Assistant Professor, and then Associate Professor at the same institute (now State University of Aerospace Instrumentation, St.-Petersburg, Russia). Her research interests include convolutional codes, communication systems, source coding and its applications to speech, audio, and image coding. She has more than 50 papers published in journals and proceedings of international conferences, and seven U.S. patents in speech and video coding.

Prof. Bocharova was awarded the Lise Meitner Visiting Chair in Engineering at Lund University, Lund, Sweden, during January–June 2005.

Boris D. Kudryashov was born in Leningrad, U.S.S.R. (now St.-Petersburg, Russia), on July 9, 1952. He received the Diploma degree in electrical engineering in 1974, the Ph.D. degree in technical sciences in 1978, both from the Leningrad Institute of Aircraft Instrumentation (LIAP), and the Doctor of Science degree from the Institute for Information Transmission Problems (IPPI), Moscow, in 2005.

Since 1978, he has been first Assistant Professor and then Associate Professor and Professor at the State University of Aerospace Instrumentation (former LIAP), St.-Petersburg, Russia. His research interests include coding theory, information theory, and applications to speech, audio, and image coding. He has published more than 70 papers in journals and proceedings of international conferences, and has nine U.S. patents in speech, audio, and image coding.

Dr. Kudryashov served as a member of Organizing Committees of ACCT International Workshops.