Symbol-by-symbol APP Decoding based on Supercode Decoding

Jürgen Freudenberger
and Uwe Kaiser
HTWG Konstanz
University of Applied Sciences, Konstanz, Germany
Institute for System Dynamics (ISD)
Email: juergen.freudenberger@htwg-konstanz.de

Abstract—An efficient sub-optimum symbol-by-symbol soft in/soft-out a posteriori probability (APP) decoding algorithm for linear binary block codes is presented. The three-step decoding procedure consists of: 1) list decoding of two supercodes, 2) calculating the intersection of these two lists, and 3) symbol-by-symbol APP decoding for the intersection. The presented results indicate that the proposed decoding procedure obtains good estimates of the ideal soft-output values and reduces the decoding complexity significantly. It may be an attractive alternative to conventional implementations of APP decoding.

I. INTRODUCTION

Most commonly, symbol-by-symbol a posteriori probability (APP) decoding of block codes is implemented by representing the code as a graph, the so-called trellis, and applying the well-known BCJR algorithm [1]. Alternative implementations of soft in/soft-out decoding are based on the soft-output Viterbi algorithm [2] or sub-optimum versions of the BCJR algorithm [3].

In this paper we consider a different approach which is based on the concept of supercode decoding introduced by Barg et al. [4]. The algorithm from [4] actually combines several previously known decoding strategies for discrete memoryless channels with the new idea of supercode decoding. A generalization of all employed decoding steps to channels with continuous output alphabets appears to be difficult. Therefore, we concentrate on the problem of supercode decoding. Consequently the intermediate decoding steps differ from that described in [4]. In particular we present efficient solutions to the problems: 1) list decoding of supercodes and 2) calculating the intersection \( L = \cap_i L_i \). The list \( L \) is represented as a trellis and the final decoding step 3) can be performed using, for example, Viterbi decoding in this trellis. Such a trellis based supercode decoding was devised and investigated for the binary symmetrical channel (BSC) in [5].

In this work we generalize trellis based supercode decoding to symbol-by-symbol soft in/soft-out a posteriori probability (APP) decoding for linear binary block codes, i.e. in the last decoding step we decode the trellis representing the intersection with the BCJR algorithm. Due to the trellis based approach we are not limited to the Hamming metric. Hence, the presented approach is suitable for soft in/soft-out decoding. It is inherently sub-optimum, because the APP decoding does not consider the complete code. Nevertheless, the presented simulation results indicate that the proposed decoding procedure obtains good estimates of the ideal soft-output values (similar to the well known Max-Log-MAP approximation [3]) and reduces the decoding complexity significantly.

In Section II, we briefly discuss the basic concept of supercode decoding. Sections III and IV describe a novel implementation of the trellis based supercode decoding-algorithm presented in [5]. When implementing an algorithm there is always a trade off between memory usage and speed. The new implementation reduces the runtime complexity compared to the approach presented in [5]. Finally, some simulation results are presented in section V.

II. BASIC CONCEPT

A supercode \( C_i \) of the binary block code \( C \) is a code containing all codewords of \( C \). For a linear code \( C \) with parity-check matrix \( H \), we can construct two supercodes \( C_1 \) and \( C_2 \) such that \( C = C_1 \cap C_2 \). Let

\[
H = \begin{pmatrix}
H_1 \\
H_2
\end{pmatrix}
\]

be the parity-check matrix of the code \( C \), this means that \( H_1 \) and \( H_2 \) are two sub-matrices of \( H \). Then the sub-matrices \( H_1 \) and \( H_2 \) define the supercodes \( C_1 \) and \( C_2 \), respectively.

Example 1: Consider for example the code \( C = \{(0000), (1110), (1011), (0101)\} \) with parity-check matrix

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

We obtain

\[
H_1 = \begin{pmatrix}
1 & 1 & 0 & 1
\end{pmatrix}
\]

\[
C_1 = \{(0000), (1100), (1110), (0010), (1011), (1001), (1011), (0101)\}
\]

and

\[
H_2 = \begin{pmatrix}
0 & 1 & 1 & 1
\end{pmatrix}
\]

\[
C_2 = \{(0000), (1000), (0110), (1110), (1011), (1101), (0011), (0101)\}
\]
Supercode decoding is performed in three steps:

1) List decoding of each supercode \( L_i \) such that the transmitted codeword is in the list \( L_i \), provided that the weight of the error pattern is less or equal \( \rho \).

2) Calculating the intersection \( L = \cap_i L_i \). These are the codewords of the code \( C \) that are in the ball around the received sequence.

3) Applying symbol-by-symbol soft in/soft-out a posteriori probability decoding to the list \( L \).

The idea here is that list decoding of the supercodes is less complex than a decoding approach which is based on the code itself. The supercodes are represented by their minimal trellises [6]. The first decoding step is performed by a marking algorithm which labels all state transitions in a supercode trellis that corresponds to a codeword in the list \( L_i \).

The list \( L \) is also represented by a trellis. This trellis is constructed using the merging algorithm. The merging algorithm calculates the intersection \( L = \cap_i L_i \) by merging the labeled state transitions from both supercode trellises.

Before we describe the algorithm’s implementation we introduce the data structures which the trellis is composed of.

### III. Trellis Representation

A trellis \( T = (S, W) \) is a directed graph with a set of states \( S = \{ \sigma \} \) and a set of branches \( W = \{ w \} \) which connects the nodes. The set \( S \) can be split into \( n + 1 \) subsets \( S = S_0 \cup S_1 \cup \cdots \cup S_n \), where \( n \) denotes the length of the code. Such a subset is also called a level \( S_t \) of the trellis. Each branch \( w \) has a label \( b \in \{0, 1\} \) which represents the code symbol \( v_t(w_t) \).

To represent the trellis, three data structures are used. One for the node \( \sigma \), a list of nodes for a level \( S_t \), and a list of the list of nodes for the complete trellis \( S \). The branches \( W \) are references which are stored in the nodes.

A node can have up to two predecessors \( N^S_b(\sigma) \) and successors \( N^P_b(\sigma) \), \( b \in \{0, 1\} \). The references to the predecessors and successors are assigned to zero if there is no corresponding node. Each incoming branch \( w^S_b \) which connects \( \sigma \) with \( N^S_b(\sigma) \) and outgoing branch \( w^P_b \) which connects \( \sigma \) and \( N^P_b(\sigma) \) has a metric. We differentiate between two metric values. The forward metric \( \Lambda^F(\sigma) \), which is the metric between \( \sigma \) and \( N^S_b(\sigma) \), and the backward metric \( \Lambda^B(\sigma) \), which is the metric between \( \sigma \) and \( N^P_b(\sigma) \). Furthermore a node stores the markings \( M_b(\sigma) \) for the outgoing branches \( w^P_b \). Hence, a node of the super code trellises can be described as a 10-tupel.

\[
\sigma = (\Lambda^B(\sigma), \Lambda^F(\sigma), \Lambda_0^B(\sigma), \Lambda^F_0(\sigma), \Lambda^F(\sigma), \Lambda^B(\sigma), N^S_0(\sigma), N^P_0(\sigma), N^S_1(\sigma), N^P_1(\sigma))
\]

These nodes are used to construct the trellises of the supercodes. For the trellis representing the list \( L \) we have to extend the node with four attributes. Two references \( N^{(1)}(\sigma) \) and \( N^{(2)}(\sigma) \) which point to the corresponding nodes in the supercode trellises. Additionally the level \( L(\sigma) \) of the node and a visited flag \( V(\sigma) \) were added to implement the breadth first search efficiently. Thus, a node of the trellis can be described as a 14-tupel.

\[
\sigma = (\Lambda^B(\sigma), \Lambda^F(\sigma), \Lambda_0^B(\sigma), \Lambda^F_0(\sigma), \Lambda^F(\sigma), \Lambda^B(\sigma), N^S_0(\sigma), N^P_0(\sigma), N^S_1(\sigma), N^P_1(\sigma), N^{(1)}(\sigma), N^{(2)}(\sigma), M_0(\sigma), M_1(\sigma), V(\sigma), L(\sigma))
\]

The implementation uses three trellises for decoding. The two supercode trellises \( T^{(1)} \) and \( T^{(2)} \) for the supercodes \( C_1 \) and \( C_2 \), and the trellis \( T \) for the code \( C \). Each node \( \sigma \) in \( T \) has a reference to a node \( \sigma^{(1)} \) in \( T^{(1)} \) and \( \sigma^{(2)} \) in \( T^{(2)} \), such that

\[
\sigma = \left( \frac{\sigma^{(1)}}{\sigma^{(2)}} \right)
\]

### IV. Supercode Decoding

The first step of supercode decoding is list decoding of each supercode \( C_i \) such that the transmitted codeword is in the list \( L_i \), provided that the weight of the error pattern is less or equal \( \rho \). With trellis based supercode decoding the list decoding is performed in the trellis of the supercodes and each list is represented by a marked trellis. Hence, the first decoding step is called the marking algorithm.

#### a) The marking algorithm:

The marking algorithm is applied to the two supercode trellises.

Step 1: Set \( \Lambda^F(\sigma_0) = \Lambda^B(\sigma_0) = 0 \) and \( M_0(\sigma_0) = M_1(\sigma_0) = 0 \).

Step 2: Calculate the forward metric for each node \( \sigma_t \), \( t = 1, \ldots, n \) when \( N^P_b(\sigma_t) \neq 0, b = 0, 1 \):

\[
\Lambda^F(\sigma_t) = \min(\Lambda^F_0(\sigma_t), \Lambda^F_1(\sigma_t)) + \lambda(\omega_{t-1})
\]

with

\[
\Lambda^F_0(\sigma_t) = \begin{cases} 
\min(\Lambda^F(\sigma), \Lambda^F_0(\sigma)), & \text{if } N^P_0(\sigma) \neq 0 \land N^P_1(\sigma) \neq 0 \\
\Lambda^F_0(\sigma), & \text{if } N^P_0(\sigma) \neq 0 \land N^P_1(\sigma) = 0 \\
\Lambda^F_1(\sigma), & \text{if } N^P_0(\sigma) = 0 \land N^P_1(\sigma) \neq 0 
\end{cases}
\]

Reset all markings: \( M_b(\sigma_t) = 0 \).

Step 3: Set \( \Lambda^B(\sigma_n) = \Lambda^B(\sigma_0) = 0 \).

Step 4: Calculate the backward metric for each node \( \sigma_t \), \( t = n-1, \ldots, 0 \) when \( N^S_b(\sigma_t) \neq 0, b = 0, 1 \):

\[
\Lambda^B(\sigma_t) = \min(\Lambda^B(\sigma), \Lambda^B_0(\sigma)),
\]

with

\[
\Lambda^B_0(\sigma_t) = \begin{cases} 
\min(\Lambda^B(\sigma), \Lambda^B_0(\sigma)), & \text{if } N^S_0(\sigma) \neq 0 \land N^S_1(\sigma) \neq 0 \\
\Lambda^B_0(\sigma), & \text{if } N^S_0(\sigma) \neq 0 \land N^S_1(\sigma) = 0 \\
\Lambda^B_1(\sigma), & \text{if } N^S_0(\sigma) = 0 \land N^S_1(\sigma) \neq 0 
\end{cases}
\]
Set markings for \( b = 0, 1 \):
\[
M_b(\sigma_t) = M(N_b^S(\sigma_t)) \land ((\Lambda^F(\sigma) + \Lambda^B_b(\sigma)) \leq \rho)
\]
with \( M(\sigma) = M_0(\sigma) \lor M_1(\sigma) \).

Considering that each node of both supercode trellises has to be visited twice the runtime complexity of the marking algorithm is given by the size of the supercode trellises.

The second step of supercode decoding is the calculation of the intersection \( L = \cap_i L_i \). Again, the intersection \( L = \cap_i L_i \) is represented by a trellis. The intersection is calculated with the merging algorithm.

b) The merging algorithm (Breath first Viterbi):
Before we introduce the merging algorithm we have to define a queue. This queue is needed for the breath first search implementation. A queue is a list of nodes. In order to reduce the runtime complexity we introduce the queue has two operations. One to add a node to create the trellis of the code \( \mathcal{C} \) before the actual decoding. The simplest way to do this is to define a queue. This queue is needed for the breath first search run, with inverted visited flag.

Step 1: Set \( \Lambda^F(\sigma_0) = \Lambda^F(\sigma_0) = 0 \) and \( Q = enqueue(Q, \sigma_0) \).

Step 2: Breath first search
\[
while(\ \sigma = dequeue(Q))\
\{
\text{if}(L(\sigma) > 0)\
\{\
\quad \Lambda^F(\sigma) = \Lambda^B(\sigma) + \lambda(\omega_L(\sigma)-1)\
\quad \Lambda^F(\sigma) = \Lambda^B(\sigma) + \lambda(\omega_L(\sigma)-1)\
\}
\}
\]
\[
M_0(\sigma) = M_0(N(1)(\sigma)) \land M_0(N(2)(\sigma))
\]
\[
\lor \{ M_0(\sigma) \land \neg V(N_0^S(\sigma))
\}
\]
\[
V(N_0^S(\sigma)) = 1
\]
\[
Q = enqueue(Q, N_0^S(\sigma))
\]
\[
M_1(\sigma) = M_1(N(1)(\sigma)) \land M_1(N(2)(\sigma))
\]
\[
\lor \{ M_1(\sigma) \land \neg V(N_1^S(\sigma))
\}
\]
\[
V(N_1^S(\sigma)) = 1
\]
\[
Q = enqueue(Q, N_1^S(\sigma))
\]
\}

Step 3: Find best path from \( \sigma_{n-1} \) to \( \sigma_0 \) considering only the marked nodes.

Step 4: Reset markings and visited flags. (Second breath first search run, with inverted visited flag)

This algorithm avoids iterating over the supercode trellis nodes. In order to reduce the runtime complexity we introduce a pairing of supercode trellis nodes which can be done once before the actual decoding. The simplest way to do this is to create the trellis of the code \( \mathcal{C} \). To achieve that a node of the trellis is implicitly marked we add two references to the corresponding nodes in the supercode trellises. As a result, a node in the trellis is marked if both corresponding nodes in the supercode trellises are marked. To calculate the metrics of the marked nodes we cannot iterate over all trellis nodes to check if a node is marked. This would mean that the merging algorithm has the same runtime complexity as Viterbi decoding. Instead of iterating over all nodes we use a breath first search [7] to visit only the marked nodes. The breath first search was chosen because it traverses the trellis level by level. Consequently, the metrics can be calculated as in the marking algorithm.

V. SIMULATION RESULTS

In this chapter we present some simulation results for binary Reed-Muller code. We compare the trellis based supercode decoder with the Viterbi and BCJR algorithms.

The simulation results for APP decoding are presented in Fig. 1. The left diagram shows the bit error rates for the \( \mathcal{R}\mathcal{M}(3,6) \) code (with code length \( n = 64 \) and dimension \( k = 42 \)). With \( \rho = 5 \) the APP decoding based on supercode decoding achieves almost BCJR-performance. With \( \rho = 4 \) the decoding performance is close to that of the Max-Log-MAP algorithm. In the right diagram we present the quality of the soft output values as the error compared to the bit error rate \( P_b \). We estimate the bit error rate from the soft output values as
\[
\tilde{P}_b = \frac{1}{K} \sum_{k=1}^{K} \frac{1}{1 + e^{L_k}}
\]
where \( L \) is a soft output value [8], [9]. The right diagram in Fig. 1 presents the relative error
\[
\frac{|\tilde{P}_b - P_b|}{P_b}.
\]
It can be seen that with \( \rho = 5 \) the quality of the soft output is comparable to the Max-Log-MAP algorithm.

Table I shows the number of nodes visited by the supercode decoding algorithm for different boundaries. Additionally the number of nodes visited by the Viterbi algorithm is presented. As said before, the complexity of the marking algorithm is independent of the input sequence and the bounded distance. Hence we can divide the complete decoding complexity in a static part, caused by the marking, and a dynamic part, caused by the merging. For \( \mathcal{R}\mathcal{M}(2,5) \) (with code length \( n = 32 \) and dimension \( k = 16 \)) and \( \rho = 4 \) the number of visited nodes by the Viterbi and supercode decoding algorithm are almost similar. But for \( \mathcal{R}\mathcal{M}(3,6) \) and \( \rho = 4 \) the Viterbi algorithm has to visit eight times more nodes than the supercode decoding algorithm.

VI. CONCLUSIONS

In this work we have presented an efficient sub-optimum symbol-by-symbol soft input/soft-output a posteriori probability (APP) decoding algorithm for linear binary block codes. The new method is based on trellis based supercode decoding.
which was introduced in [5]. The new implementation presented in this paper reduces the runtime complexity compared to the approach presented in [5].

The presented simulation results indicate that the proposed APP decoding procedure obtains good estimates of the ideal soft-output values and reduces the decoding complexity significantly compared to decoding in the complete syndrome trellis. It is therefore an attractive alternative to conventional implementations of APP decoding for linear binary block codes.

LDPC codes or turbo codes that are based on binary block codes usually employ high rate Hamming codes [10], [11], [12] that allow conventional low complexity APP decoding. The new algorithm may enable other code constructions that are based on block codes with higher error correcting capabilities.

REFERENCES


