Exact and Approximate Algorithms for ML Decoding on Tail-Biting Trellises

Priti Shankar A.S. Madhu Aditya Nori

IISc-CSA-TR-2005-3
http://archive.csa.iisc.ernet.in/TR/2005/3/

Computer Science and Automation
Indian Institute of Science, India

March 2005

Exact and Approximate Algorithms for ML Decoding on Tail-Biting Trellises

Priti Shankar A.S. Madhu Aditya Nori

Abstract

We analyze an algorithm for exact maximum likelihood(ML) decoding on tail-biting trellises and prove that under cerrtain assumptions the complexity of of the exact algorithm is $O(S \log S)$, where S is the number of states of the tail-biting trellis. We also propose an approximate variant whose simulated performance is seen to be virtually indistinguishable from that of the exact one at all values of signal to noise ratio. We analyze the approximate algorithm and deduce the conditions under which its output is different from the ML output. We report the results of simulations for the exact and approximate algorithms on tail-biting trellises for the 16 state (24,12) Extended Golay Code, and two rate 1/2 convolutional codes with memory 4 and 6 respectively. An advantage of our algorithms is that they do not suffer from the effects of limit cycles or the presence of pseudocodewords.

1 Introduction

Tail-biting trellises are perhaps the simplest instances of decoding graphs with cycles and usually approximate algorithms are used for decoding, as exact algorithms are believed to be too expensive. These approximate algorithms iterate around the trellis until either convergence is reached, or for a preset number of cycles. To the best of our knowledge, no *exact* decoding algorithms other than the brute force algorithm have been proposed so far for the general case. We analyze an *exact* recursive algorithm proposed by us earlier and show that for a class of errors, exact maximum likelihood decoding has time complexity $O(S \log S)$ and space complexity $O(S \times S_0)$ *bits* where S is the number of vertices in the tail-biting trellis and S_0 is the number of vertices at time index 0. The algorithm is always guaranteed to converge. The complexity term attributed to Viterbi updates is $O(S)$ under certain conditions. The $O(S \log S)$ term comes from the overheads of maintaining a data structure called a *heap*. We also propose two approximate variants that always converge, and observe their performance on tail-biting trellises for the (24,12,8) extended Golay code and two convolutional codes of rate 1/2 and memory of 4 and 6 respectively. The performance of the second approximate variant is indistinguishable from that of the exact algorithm in terms of bit error rate and it is guaranteed to update each node in the tail-biting trellis at most three times i.e it performs a computation equivalent to at most three rounds on the trellis.

2 Related Work

Aji et al.[1] have shown that iterative maximum-likelihood (ML) decoding on tail-biting trellises will asymptotically converge to exact maximum likelihood decoding for certain codes. The presence of *pseudocodewords* sometimes results in sub-optimal decoding and it is also possible to have situations where the iterative message passing algorithm does not converge. Several maximum likelihood decoding algorithms on tail-biting trellises have been proposed without a theoretical analysis[8, 14, 15, 12, 11], but with good experimental results. Most of these are sub-optimal algorithms in that they may not produce the exact maximum-likelihood result on termination. Anderson and Hladik[2] have given an algorithm that is optimal for bounded distance decoding.

3 Background

We assume that the reader is familiar with the definition of a tail-biting trellis for a block or convolutional code and with the notions of linear and circular spans of codewords. Definitions are available in [12, 3, 5]. Koetter and Vardy [5] have shown that any linear trellis, conventional or tail-biting can be constructed from a generator matrix whose rows can be partitioned into two sets, those which have linear span, and those taken to have circular span. The trellis for the code is formed as a product[6] of the elementary trellises corresponding to these rows. We will represent such a generator matrix as $G = \left[\begin{array}{c} G_l \\ \hline G_c \end{array}\right]$, where G_l is the submatrix consisting of rows with linear span, and G_c the submatrix of rows with circular span. Let T_l denote the minimum conventional trellis for the code generated by G_l.

If l is the number of rows of G with linear span and c the number of rows of circular span, the tail-biting trellis constructed using the product construction will have q^c start states. Each such start state defines a subtrellis whose codewords form a coset of the subcode corresponding to the subtrellis containing the all 0 codeword. If subtrellises T_1 and T_2 share states from time indices i to j then the interval $[i, j]$ is called the *merging* interval of T_1 and T_2. A pair of subtrellises do not share any states outside their merging interval. For a vector $\mathbf{v}$ of circular span in G_c, the complement of the span with respect to $\mathcal{I} = \{1, 2, \ldots n\}$ is called the *zero run* of the vector. A tail-biting trellis is said to satisfy the *intersection property* if the intersection of all the zero runs of the members of G_c is non-empty.

4 Decoding

For purposes of decoding we use the unrolled version of the trellis with start states $s_0, s_1 \ldots s_l$ and final states $f_0, f_1 \ldots f_l$. where l is the number of subtrellises. An (s_i, f_i) path is a codeword path in trellis T_i, whereas an (s_i, f_j) path for $i \neq j$ is a non codeword path. For purposes of our discussion we term the label sequence along such a path as a *semicodeword.*

The algorithm has two phases. The first phase performs a Viterbi algorithm on the tail-biting trellis and examines surviving paths, called *survivors* here, at final states of all subtrellises. A survivor at a final state is either a codeword (abbreviated as an (s_i, f_i) path) or what we term as a *semicodeword* (corresponding to an (s_i, f_j) path with $i \neq j$.) The metric associated with a semicodeword survivor in trellis T_i is a lower bound on the cost of a winning codeword in trellis T_i. Therefore the first step is to discard from consideration all subtrellises whose survivors are semicodewords with metric higher than that of the best surviving codeword, if there is one as these subtrellises will not contain the shortest path. The next step is to sort the metrics of all remaining subtrellises whose survivors are semicodewords with metric *lower* than that of the best surviving codeword in ascending order. These subtrellises are called *residual* trellises and the second phase works only on these subtrellises with the metrics of the survivors as initial estimates. A data structure called a *heap* is employed which stores the metrics of all nodes which have been visited in the second phase and which are candidates for a Viterbi update. Initially the set consists of the start nodes of the residual subtrellises along with their associated metrics. The heap when queried can deliver the minimum element in constant time, and each time an element is inserted into it, the complexity of restructuring it to preserve the constant time querying property is logarithmic in the number of elements it contains. A node taken off the heap has a metric and a subtrellis identification number associated with it. If it is taken off the heap for exploration, the path ending at the node corresponding to a codeword prefix has the best current metric.

The metric at a node is a sum of two quantities–the length of the shortest path to it from the start node of the associated subtrellis, and an estimate of the shortest path from the node to the final state in the subtrellis. The second phase begins by taking the node with minimum metric off the heap and expanding it using function *Expand*. This calls function *Update* which examines and updates the metrics of all successors of the node. Whenever function *Expand* is called at a node we say the node is *closed*, whereas nodes in the heap are called *open* nodes. If a node is closed in a subtrellis then the shortest path to that node in the subtrellis has been found. At any instant in the second phase, the algorithm is moving forward on a single path in some subtrellis, closing nodes which lie on surviving paths in the first phase in that subtrellis. Each time *Update* returns from execution at a node, function *Expand* checks if the metric has increased or not. If it has, this is an indication that this was not a surviving path in the first phase and the node is put on the heap and *Expand* either calls *Update* at the next successor, or returns to its calling instance if no more successors are left to be examined. If the metric has not increased, *Expand* is invoked recursively at the node thereby closing the node. When a return is made to the start node of a subtrellis, *trellis switching* takes place, and some other trellis which is seen to have the best potential to deliver the minimum cost path is taken up. Thus the algorithm may switch from one subtrellis to another during the course of its execution. However it is always guaranteed to eventually zero in on the ML path. A central issue therefore is the number of updates it has to perform before it finds the ML path, as this determines the size of the search space.

The functions *SecondPhase*, *Expand* and *Update* are described in Figure 1.

5 Analysis

The proof that on termination the algorithm always outputs the optimal path is provided in [10].

During any point in the second phase, the algorithm is exploring some path in a candidate subtrellis called the *current* trellis even though it may do so in discontinuous steps. This path is called the *current path* in that subtrellis. The metric which it uses to decide whether to continue on the current path on the current trellis, say T_i, or forsake it in favour of another path either in the current trellis or on another candidate trellis is initially $e(s_i, f_i)$. We have the following lemmas specifying the behaviour of the algorithm.

Due to lack of space, we omit the proofs.

Lemma 5.1 *During the second phase if the current path updates a node v using function Update where the survivor in the first phase was not in the current trellis then the metric becomes $e(s_i, f_i) + \Delta(i, v)$ where $\Delta(i, v)$ is the difference between the cost of the least cost path ending at v in the current trellis and the survivor at v during the first pass.*

The following properties hold for the metric. Let $m_i(N)$ denote the metric in subtrellis i at node N:

Lemma 5.2 *Let an (s_k, f_i) path be the winner at f_i in the first phase and let it win over an (s_i, f_i) path at node A. Then $m_i(A) = m_i(f_i)$ and $m_i(B) < m_i(f_i)$ for any proper predecessor B of A.*

For each shortest path in a subtrellis i the nodes where it was overtaken by paths originating at the start nodes of other trellises in the first phase are the nodes where its metric will be updated. These nodes are called *updating points.*

Lemma 5.3 *Let subtrellises T_i and T_j share a node N and between them, let T_i be the first to close the node in the second phase. Then $m_i(N) \leq m_j(N)$.*

Lemma 5.4 *For nodes A and B let (A, B) be a path segment in the merging interval of T_i and T_j and let $m_i(A) \leq m_j(A)$. Then $m_i(B) \leq m_j(B)$.*

We next try to estimate the number of nodes that are examined by the algorithm before the ML path is reached. We first show that any path from an arbitrary start node to any final node represents a vector in a vector space. For the sake of simplicity we restrict our arguments to binary codes.

Lemma 5.5 *The set of all labels from an arbitrary start node to any final node is a vector space.*

Proof Assume that each of the c vectors in the submatrix G_c of the generator matrix is of the form $v_i = [\mathbf{h}_i, \mathbf{0}, \mathbf{t}_i]$ where $\mathbf{h}_i$ stands for the sequence of symbols before the zero run, and is called the *head* and $\mathbf{t}_i$ stands for the sequence of symbols following the zero run and is called the *tail* and $\mathbf{0}$ is the zero run containing the appropriate number of zeroes. Let $\{v_1, v_2 \ldots v_c\}$ be the vectors of G_c. Then the matrix G_s defined as $G_s = \left[\frac{G_l}{G'_c}\right]$, where G'_c consists of $2c$ rows of the form $[\mathbf{h}_i, \mathbf{0}], [\mathbf{0}, \mathbf{t}_i], 1 \leq i \leq c$, (where the number of zeroes in $\mathbf{0}$ makes up a total of n elements for the row) generates the set of labels of all paths from any start node to any final node. This set has 2^{l+2c} elements. This can be verified from the product construction. The set of elements of this vector space consists of *semicodewords* and codewords. Each semicodeword is the label of an (s_i, f_j) path $i \neq j$. $\blacksquare$

We use a result of Tendolkar and Hartmann[13] stated below.

Lemma 5.6 *Let H be the parity check matrix of the code and let a codeword $\mathbf{x}$ be transmitted as a signal vector $S(\mathbf{x})$. Let the binary quantization of the received vector $\mathbf{r} = r_1, r_2, \ldots r_n$ be denoted by $\mathbf{y}$. Let $\mathbf{r}' = (|r_1|, |r_2|, \ldots |r_n|)$ and $S = \mathbf{y}H^T$. Then ML decoding is achieved by decoding a received vector $\mathbf{r}$ into the codeword $\mathbf{y} + \mathbf{e}$ where $\mathbf{e}$ is a binary vector that satisfies $S = \mathbf{e}H^T$ and has the property that if $\mathbf{e}'$ is any other binary vector such that $S = \mathbf{e}'H^T$ then $\mathbf{e}.\mathbf{r}' < \mathbf{e}'.\mathbf{r}'$ where . is the inner product.*

Since the space explored by the algorithm, namely the space of semi-codewords and codewords is a vector space, we can analyse the algorithm assuming that the ML codeword is the all 0 codeword and examine how many semicodewords are explored by the algorithm before it converges on the all 0 codeword. Let $\mathbf{e}$ be an error pattern that has the same syndrome with respect to H_s the parity check matrix for G_s, as the binary quantization of the received vector. Let C_s denote a semicodeword and C an arbitrary non zero codeword. The lemma below establishes the condition under which the path corresponding to a semicodeword will be explored by the decoding algorithm.

Lemma 5.7 *Assume the all 0 codeword is the ML codeword. Let $\mathbf{e}$ be the binary quantization of the received vector. For the error pattern $\mathbf{e}$ the decoding algorithm explores a path corresponding to a semicodeword C_s satisfying*

$$metric(C_s + \mathbf{e}).\mathbf{r}' < \mathbf{e}.\mathbf{r}' < (C + \mathbf{e}).\mathbf{r}' \tag{1}$$

for all nonzero codewords C.

Let us refer to the first inequality in the lemma as inequality A and the second as inequality B.

Inequality A specifies that the metric of semicodeword corresponding to C_s corrupted by $\mathbf{e}$ is more likely than the all zero path corrupted by $\mathbf{e}$. Inequality B, specifies that for error $\mathbf{e}$ the all 0 path is more likely than any non zero codeword corrupted by $\mathbf{e}$. So the question is, given $\mathbf{e}$ satisfying inequality B, how many semicodewords are there satisfying inequality A, as these correspond to paths that are fruitlessly explored. If this number is $O(S_0)$ then the number of nodes closed is at most $O(S_0 \times n) = O(S)$.

Since semicodewords share prefixes and suffixes with codewords, only error patterns that drop the metric of a semicodeword enough to satisfy inequality A but not violate inequality B cause fruitless searches of semicodewords. Since semicodewords share prefixes and suffixes with codewords, such error events may not be too frequent.

The complexity of the algorithm is established by the following theorem.

Theorem 5.1 *The two pass exact decoding algorithm has time complexity $O(S \log S)$ and space complexity $O(S \times S_0)$* bits *for all error patterns* $\mathbf{e}$ *such that the number of semicodewords satisfying equation 1 of Lemma 5.7 is $O(S_0)$.*

Proof Phase 1 of the algorithm has complexity $O(S)$. The sorting operation at the end of the first phase has complexity $O(S_0 \log S_0)$. If the number of semicodewords explored is $O(S_0)$, then the number of calls to function *Expand* is $O(S_0 \times n)= O(S)$. Each call takes either constant time if the node is closed, or time $O(\log\ h)$ where h is the size of the heap, if the node is inserted into the heap. For each node expanded, at most 2 insertions into the heap are possible, hence the size of the heap is $O(S)$ from which we get the specified time complexity. In addition to the $O(S)$ space for storing all the survivor costs, we need $O(S)$ space for the heap, and a bit vector of size $O(S_0)$ at each state to indicate which trellises share that state, giving a space complexity of $O(S \times S_0)$ bits.

Note that if S_0 is at most the word size of the machine, the space requirements are also $O(S)$. ■

6 An Approximate Algorithm and Simulations

While it is possible to argue that not too many semicodewords will be examined before the ML path is output, we would like to explore a variation of the algorithm which explicitly bounds that number. Therefore we propose an approximate algorithm which closes a node at most once and hence the total number of Viterbi updates in the first and second phases is at most $2S$. Since a node is closed at most once it is conceivable that a node that is on the ML path is closed by a trellis that does not contain the ML codeword. In such a case the result produced will be different from the ML result. We now analyse the conditions under which this happens.The symbols are the same as those defined for Lemma 5.7. Recall that the intersection property requires the intersection of all the zero runs of vectors in G'_c to be non-empty.

Lemma 6.1 *If the tail-biting trellis satisfies the intersection property, the approximate algorithm produces a non-ML output for error patterns* $\mathbf{e}$ *satisfying equation 1 whenever* C_s *is a semicodeword which is formed as a linear combination of rows of* G_s *that contain at least one vector from* G_l*.*

The lemma provides an explanation of the observation that decoding errors are infrequent even in the approximate algorithm, so much so that the bit error rate curves are practically indistinguishable. This is because low weight semicodewords are those consisting of heads h_i followed by all 0's or tails t_i preceded by all 0's. These semicodewords cannot cause decoding differences between the ML algorithm and the approximate algorithm because of Lemma 6.1. Also if it is the case that the intersection of all the zero runs of the circular span vectors in G_c is nonempty, (as is the case for all three codes on which simulations are run), then even linear combinations of the above semicodewords cannot cause decoding differences.

One could get an even better approximation by allowing a node to be closed at most twice. We have experimented with this and observe that the bit error rate for this approximation is indistinguishable from that of the exact algorithm at all values of signal to noise ratio. The significance of this is that the time complexity can be explicitly bounded by the complexity of at most three updates for each node of the tail-biting trellis, one in the first phase and at most two in the second phase.

We show the results of simulations on minimal tail-biting trellises for the 16 state tail-biting trellis [3] for the extended (24,12,8) Golay code on an AWGN channel with antipodal signalling and tail-biting trellises for two rate 1/2 convolutional codes, the (133,171) code(also called the (554,774) code) with memory 6 and circle size 48 and the (35,31) code(also called the (72,62) code) with memory 4 and circle size 20. We compute the variation of both, the average as well as the maximum number of Viterbi updates(counting both phases) with the signal to noise ratio, and compare this with the number of Viterbi updates needed for the brute force approach. For the Golay code the number of expansions for the brute force is 1744 and the maximum and average vary from 602 and 245 respectively at 0dB to 396 and 193 respectively at 5 dB. The maximum heap size varies from 285 at 0dB ro 135 at 5dB. The number of nodes of the tail-biting trellis is 192. For the (133,171) convolutional code the number of expansions for the brute force are 159552, and the maximum and average of the exact algorithm vary from 22311 and 4414 respectively at 0 dB to 4693 and 3088 respectively at 5 dB.. The number of nodes in the tailbiting trellis is 3072. The maximum heap size varies from 13064 at 0 dB to 1059 at 5 dB.For the (35,31) convolutional code the number of expansions for the brute force are 4368, and the maximum and average of the exact algorithm vary from 1437 and 426 respectively at 0 dB to 660 and 322 respectively at 5 dB. The number of nodes in the tailbiting trellis is 320. The maximum heap size varies from 701 at 0 dB to 241 at 5 dB. We also display the performance of the approximate algorithms closing nodes at most once for the first approximation, and at most twice for the second approximation in Figures 2, 3, 4 and and find that there is virtually no difference between the bit error rates for the second approximation and the exact ML algorithm.

Acknowledgement The code used in these simulations is a modified version of that developed with our collaborators in [9].

References

[1] S. Aji, G. Horn, R. McEliece and M. Xu, Iterative Min-Sum Decoding of Tail-Biting Codes, *Proceedings of Information Theory Workshop*, Killarney, Ireland, June 22-26, pp. 68-69.

[2] J.B. Anderson and S.M. Hladik, An Optimal Circular Viterbi Decoder for the Bounded Distance Criterion, *IEEE Transactions on Communications*, **50**(11), November 2002.

[3] A.R. Calderbank, G.D. Forney,Jr., and A. Vardy, Minimal Tail-Biting Trellises: The Golay Code and More, *IEEE Trans. Inform. Theory*, **45**(5), July 1999, pp. 1435-1455.

[4] R.V. Cox and C.V. Sundberg, An Efficient Adaptive Circular Viterbi Algorithm for Decoding Generalized Tailbiting Convolutional Codes, *IEEE Transactions on Vehicular Technology* **43**(1), February 1994, pp 57-68.

[5] R. Koetter and A. Vardy, The Structure of Tail-Biting Trellises: Minimality and Basic Principles, IEEE Trans. Inform. Theory, September 2003, pp. 2081-2105.

[6] F.R. Kschischang and V. Sorokine, On the trellis structure of block codes, *IEEE Trans. Inform. Theory*, **41**(6), Nov 1995, pp. 1924-1937.

[7] B.D. Kudrashyov, Decoding of block codes obtained from convolutional codes, *Problemy Peredachi Informatsii*,**26**(2), pp 18-26, April-June 1990(in Russian). English Translation, Plenum Publishing Corporation, Oct 1990.

[8] J.H.Ma and J.K.Wolf, On tail-biting convolutional codes, *IEEE Trans. Commun.*,**34**,February 1986, pp 104-111.

[9] P. Shankar, P.N.A. Kumar, K. Sasidharan and B.S. Rajan, ML decoding of block codes on their tail-biting trellises, in *Proc. 2001 IEEE Int. Symposium on Information Theory*, IEEE Press, 2001, pp. 291.

[10] P. Shankar, A. Dasgupta, K. Deshmukh and B.S. Rajan, On Viewing Block Codes as Finite Automata, Theoretical Computer Science, **290**(2003) 1775-1797.

[11] R.Y.Shao, Shu Lin and M.P.C. Fossorier, Two decoding algorithms for tail-biting codes, *IEEE Trans. Commun.*,**51**(10), October 2003, pp 1658-1665.

[12] G.Solomon and H.C.A. van Tilborg, A connection between block and convolutional codes, *SIAM J. Appl. Math.*, **37**, October 1979, pp 358-369.

[13] N.N. Tendolkar and C.R.P. Hartmann, Generalization of Chase Algorithms for Soft Decision Decoding of Binary Linear Codes, *IEEE Trans. Inform.Theory,***30**(5), September 1984,pp 714-721.

[14] Q.Wang and V.K.Bhargava, An efficient maximum-likelihood decoding algorithm for generalized tail-biting codes including quasi-cyclic codes, *IEEE Trans. Commun.*, **37**, August 1989, pp 875-879.

[15] K.S.Zigangirov and V.V. Chepyzhov, Study of decoding tail-biting convolutional codes, in *Proc. Swedish-Soviet Workshop on Information Theory*(Gotland, Sweden, Aug. 1989), pp 52-55.

Figure 1: Functions $SecondPhase$, $Expand$ and $Update$ of the Decoding Algorithm

```
**function** $SecondPhase$
/* This phase begins with $r$ residual trellises whose metrics have been sorted in increasing order*/
/* First create a heap $H$ with these $r$ metrics; each element of the heap is a record containing the trellis number, the node, the time index, and the metric*/
**for** $i = 1$ **to** $r$ **do**
  $InsertHeap(H, i, startVertex(T_i), 0, e(s_i, f_i))$
  **while** $IsEmpty(H) = false$ **do**
    $h :=\ DeleteMin(H)$
    $S := S \cup h.node$ /*Add $h.node$ to the set of closed nodes*/
    $Expand(h.trellisNo, h.state, h.timeindex, h.metric)$
  **end while**
**end for**

**function** $Expand(trellisnumber, state, index, metric)$
**if** $index = n - 1$ **then**
  output $P(state)$; **return**
**else**
  **for** each successor $succ$ of $state$ **do**
    $Update(trellisnumber, state, succ.state, succ.metric, index)$
    **if** $succ.metric \leq metric$ **then**
      $S := S \cup \{succ.state\}$;
      $Expand(trellisnumber, succ.state, index, succ.metric)$
    **else**
      $InsertHeap(H, trellisnumber, succ.state, index, succ.metric)$
    **end if**
  **end for**
**end if**

**function** $Update(i, node1, node2, metric, timeindex)$;
$timeindex := timeindex + 1$
$newcost := node1.cost + edgecost(node1, node2)$
**if** $newcost \leq node2.cost$ **then**
  /* update the current shortest path to $node2$ */
  $P(node2) := (P(node1), node2)$
  $node2.cost := newcost$
  $metric := node2.cost + e(s_i, f_i) - node2.cost1$
  /* $node2.cost1$ is the cost of the survivor in the first phase */
**end if**
```

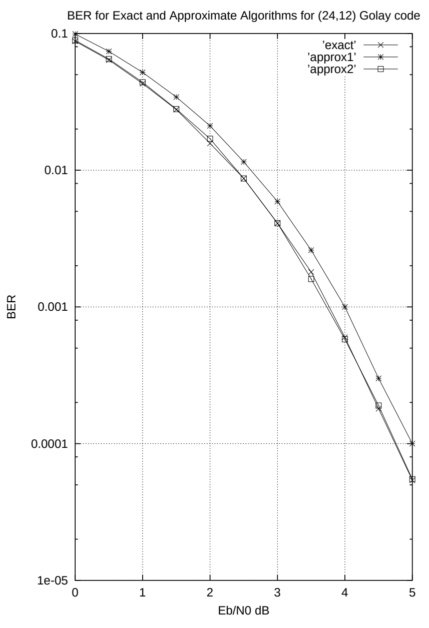

Figure 2: BER of the Exact and Approximate Algorithms for the (24,12) Extended Binary Golay Code

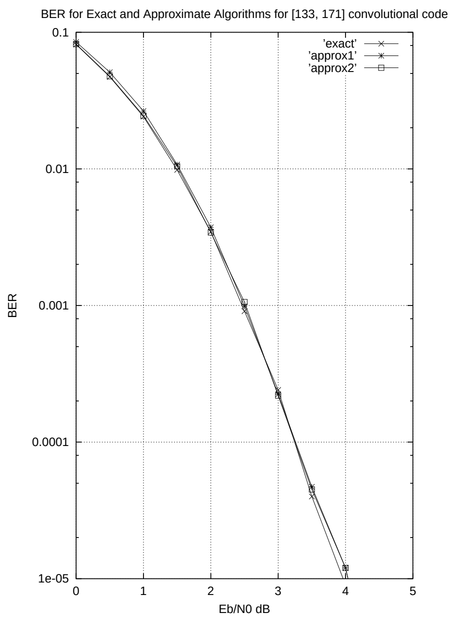

Figure 3: BER of the Exact and Approximate Algorithms for the rate 1/2 (133,171) Convolutional Code with circle Length 48

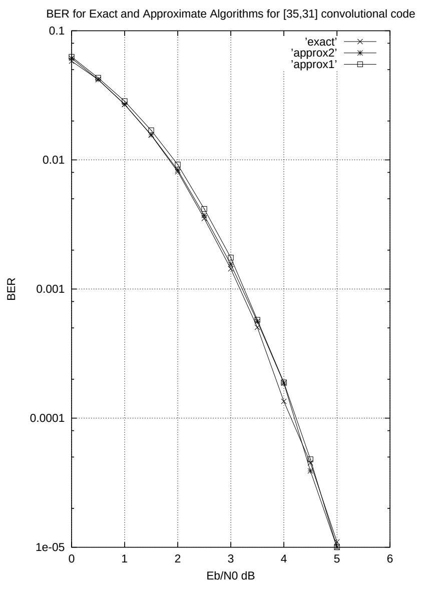

Figure 4: BER of the Exact and Approximate Algorithms for the rate 1/2 (35,31) Convolutional Code with circle Length 20