Ann. Telecommun. (2014) 69:529–537
DOI 10.1007/s12243-013-0400-9

An efficiently implementable maximum likelihood decoding algorithm for tailbiting codes

Jorge Ortín · Paloma García Dúcar · Fernando Gutiérrez · Antonio Valdovinos

Received: 12 April 2013 / Accepted: 1 October 2013 / Published online: 19 October 2013
© Institut Mines-Télécom and Springer-Verlag France 2013

Abstract Convolutional tailbiting codes are widely used in mobile systems to perform error-correcting strategies of data and control information. Unlike zero tail codes, tailbiting codes do not reset the encoder memory at the end of each data block, improving the code efficiency for short block lengths. The objective of this work is to propose a low-complexity maximum likelihood decoding algorithm for convolutional tailbiting codes based on the Viterbi algorithm. The performance of the proposed solution is compared to that of another maximum likelihood decoding strategy which is based on the A* algorithm. The computational load and the memory requirements of both algorithms are also analysed in order to perform a fair comparison between them. Numerical results considering realistic transmission conditions show the lower memory requirements of the proposed solution, which makes its implementation more suitable for devices with limited resources .

Keywords Channel coding · Tailbiting codes · Maximum likelihood decoding · A* algorithm · Viterbi algorithm

J. Ortín (✉)
Centro Universitario de la Defensa Zaragoza,
50090 Zaragoza, Spain
e-mail: jortin@unizar.es

P. G. Dúcar · F. Gutiérrez · A. Valdovinos
Aragon Institute of Engineering Research (I3A),
Universidad de Zaragoza, Zaragoza, Spain

P. G. Dúcar,
e-mail: paloma@unizar.es

F. Gutiérrez
e-mail: ferguso@unizar.es

A. Valdovinos
e-mail: toni@unizar.es

1 Introduction

The two main channel coding techniques used by 4G mobile systems, such as LTE and WiMAX, are turbocodes and convolutional tailbiting codes (hereafter tailbiting codes) [1, 2]. From them, turbocodes achieve better results in terms of error-correcting capabilities, but at the cost of a higher computational complexity. Moreover, the gap between the performance of turbocodes and tailbiting codes becomes narrower for short block lengths, with tailbiting codes being preferred to encode the short-length control information of these standards.

The main characteristic of tailbiting codes is that the state of the encoder memory is the same at the beginning and at the end of the encoding of each data block without adding any tail to reset the encoder memory [3]. For feedforward convolutional codes, this is achieved presetting the encoder memory with the last bits of the block being encoded. The main advantage of tailbiting codes compared to convolutional zero tail codes is their higher efficiency, especially for short block lengths.

The brute force algorithm to obtain a maximum likelihood (ML) decoding of tailbiting codes is to perform a different Viterbi decoding for each one of the possible initial and final states of the trellis to decode finally the path with the best metric. The problem with this algorithm is its huge computational load, since 2^k iterations of the Viterbi algorithm must be performed to get the most likely tailbiting path of the trellis, with k as the length of the code memory.

In order to decrease this computational load, several suboptimal algorithms have been proposed that employ the circular properties of tailbiting codes. The circular Viterbi algorithm (CVA) [4] extends the Viterbi algorithm to more than one trellis round, using the final metrics obtained at the end of each iteration of the Viterbi algorithm as

the initial ones for the following iteration. This process is repeated iteratively until a stopping rule is fulfilled or a maximum number of iterations is reached. Both the wrap-around Viterbi algorithm (WAVA) [5] and the bounded distance criterion circular Viterbi algorithm (BDC-CVA) [6] are based on the CVA. WAVA searches the most likely path of the trellis at the end of each iteration, stopping the process if this path is tailbiting. If the maximum number of iterations is exceeded, the tailbiting path with the best metric found so far is decoded, although a non-tailbiting path with a better metric may exist. BDC-CVA uses a fixed-length extended trellis to perform the Viterbi algorithm. Similar to CVA, this extended trellis is obtained concatenating the encoded block a number of preset times. Finally, the two-step Viterbi algorithm does not use CVA to obtain the decoded sequence [7]. This algorithm searches a proper initial and final state for the tailbiting path with a modified version of the soft-output Viterbi algorithm (SOVA) [8], forcing the trellis to start and to finish at that state.

Recently, there has been an increased interest in the topic of ML decoding of tailbiting codes and several novel decoding algorithms have been proposed so far [9–11]. Some of these solutions rely on the use of the A* algorithm, which is a least-cost path-finding algorithm for graphs that can be applied to search the most likely path of a tailbiting trellis. Although this decoding strategy leads to a marked drop in the computational load of the decoding process compared to that of the brute force algorithm, the A* algorithm is more complex than the Viterbi algorithm and its implementation requires a great amount of memory, which makes it not suitable for devices with limited resources.

In contrast to these previous solutions, the ML decoding algorithm proposed in this work is fully based on the Viterbi algorithm, which greatly decreases its implementation complexity. The structure of the proposed algorithm is similar to that of the EA* algorithm [10], but substituting the A* algorithm with the classic Viterbi algorithm. Although this change may increase the overall computational load slightly, the parallelization achieved with the Viterbi algorithm and its more efficient use of memory resources make it easier to implement in real devices than the A* algorithm. In this sense, the proposal of low-complexity algorithms to decode the error-correction strategies used in wireless systems is of paramount importance, since they can decrease the power consumption of the terminals, increasing its energy efficiency and lifetime [12].

The work is organized as follows: Section 2 presents a detailed explanation of the proposed algorithm, comparing it and contrasting it with the EA* algorithm. Section 3 analyses the complexity and the memory requirements of both algorithms. In Section 4, their actual computational load and memory usage are obtained by simulation for a typical tailbiting code used in a 4G system. Finally, Section 5 summarizes the main conclusions.

2 Proposed algorithm and comparison with the EA* algorithm

In this section, the proposed algorithm and the EA* algorithm are explained in detail. Both algorithms are divided in two stages. In the first stage, it is gathered information from the trellis which allows pruning the possible initial and final states of the most likely tailbiting path. In the second stage, the most likely tailbiting path is searched with the Viterbi algorithm in the proposed algorithm and with the A* algorithm in the EA* algorithm.

Let T be the trellis of a $(n, 1)$ binary tailbiting code with codewords of length nL bits and generated with an encoder of memory k. This trellis T spans $N = 2^k$ states at each time instant i, with i ranging from 0 to L, and it is formed by the union of N subtrellises, $T(s_p)$, each of them starting and ending at one of the possible N states. The tailbiting trellis can also be seen as a subset of a more general trellis T_s which does not have constraints concerning the initial and final states of each path in it. Therefore, each path of the trellis T is also a path of T_s, but not in reverse.

The metric $m\left(s_q^{i-1}, s_p^i\right)$ corresponding to the transition between the state s_q at time instant $i-1$ and state s_p at time instant i is defined as

$$m\left(s_q^{i-1}, s_p^i\right) = \sum_{l=1}^{n} (x_l \oplus y_{i,l}) |R_{i,l}| \quad (1)$$

where n is the number of output coded bits per information bit, x_l is the output bit l corresponding to that state transition, $y_{i,l}$ is the l hard demodulated bit of the i encoded symbol (the i encoded symbol is formed by the n output bits generated by the encoder when it is fed with the i input bit) and $R_{i,l}$ is its associated log-likelihood ratio, defined as

$$R_{i,l} = \ln \frac{P(y_{i,l} = 1|r_{i,l})}{P(y_{i,l} = 0|r_{i,l})} \quad (2)$$

with $r_{i,l}$ as the l soft demodulated bit of the i encoded symbol. The rule to obtain the accumulated metric of the survivor path at each state s_p and time instant i according to the Viterbi algorithm and this metric definition is

$$M\left(s_p^i\right) = \min_q \left(M\left(s_q^{i-1}\right) + m\left(s_q^{i-1}, s_p^i\right) \right). \quad (3)$$

The most likely path in the trellis will be the path with the lowest accumulated metric, which allows applying the least-cost path-finding algorithms used for graphs, such as the A* algorithm, in tailbiting trellises.

First stage of the decoding The first stage of the proposed algorithm and the EA* algorithm is the application of the Viterbi algorithm to the trellis T_s with the initial state metrics set to zero. The survivor paths at the end of the Viterbi algorithm allow pruning the subtrellises where the most likely tailbiting path will be sought. Moreover, their accumulated metrics give a hint on the probability of the subtrellises where this path may be. In the EA* algorithm, this stage is also employed to collect information concerning the trellis which will be needed for the application of the A* algorithm.

While the standard Viterbi algorithm is used in the proposed algorithm and no past metrics need to be recorded in each update, in the EA* algorithm, the accumulated metric of the survivor path at each state s_p and time instant i, $M\left(s_p^i\right)$, is stored. The term $\Delta\left(s_p^i\right)$, defined as

$$\Delta\left(s_p^i\right) = \max_q \left(M\left(s_q^{i-1}\right) + m\left(s_q^{i-1}, s_p^i\right)\right) - \min_q \left(M\left(s_q^{i-1}\right) + m\left(s_q^{i-1}, s_p^i\right)\right), \quad (4)$$

is stored in the EA* algorithm as well. This term corresponds to the metric difference between the survivor path and the discarded path at each state of the trellis.

In both algorithms, the survivor paths at each one of the N final states of the trellis T_s are sorted and tracebacked when the trellis calculation is finished. These paths may not be tailbiting since the survivor path at state s_p^L represents the least-cost path ending at that state without restrictions regarding its initial state. If the survivor path with the least accumulated metric is also tailbiting (its initial and final states are the same), the algorithm stops and this path is decoded as the most likely tailbiting path. If not, the tailbiting survivor with the least accumulated metric is searched and its metric is stored in the variable $\boldsymbol{\rho}$. This path constitutes the best candidate for the most likely tailbiting path found in the first stage of the decoding. If none of the survivors is tailbiting, then there is no tailbiting candidate and $\boldsymbol{\rho}$ is set to ∞.

Since the survivor path at each final state is the least-cost path ending at that state, any other path with the same final state will necessarily have a higher metric and it will have been discarded at some point in the operation of the Viterbi algorithm. Therefore, if the metric of the survivor path at s_q^L is higher than $\boldsymbol{\rho}$, then the subtrellis $T(s_q)$, which starts and ends at s_q, can be removed from the search for the least-cost tailbiting path in the second stage of the decoding.

Second stage of the proposed algorithm As stated previously, the final states of the survivor paths with an accumulated metric lower than $\boldsymbol{\rho}$ set the subtrellises $T(s_p)$ where the most likely tailbiting path must be searched. To perform it, we propose to use the standard Viterbi algorithm forcing the initial and final states of the trellis to each one of these states s_p. Therefore, the second stage of the decoding is based on an iterative application of the Viterbi algorithm to get the most likely tailbiting path and its accumulated metric in each one of these subtrellises. Since the initial and final states of a subtrellis are the same, it is clear that the survivor path obtained at the end of the application of the Viterbi algorithm will be tailbiting.

The subtrellises $T(s_p)$ can be sorted in an ordered list Q_T according to the value of the accumulated metric $M\left(s_p^L\right)$ of their corresponding survivor path found in the first stage of the decoding. This accumulated metric is closely related to the likelihood of the survivor path, and it can be used to select the order of application of the Viterbi algorithm to the subtrellises of Q_T.

Let $T_1(s_p)$ be the subtrellis in the first position of Q_T, i.e. the subtrellis with the lowest value of $M\left(s_p^L\right)$. Once the Viterbi algorithm has been applied on $T_1(s_p)$, this subtrellis is deleted from Q_T and the accumulated metric of its tailbiting survivor path is compared to $\boldsymbol{\rho}$. If the accumulated metric is lower than $\boldsymbol{\rho}$, then the value of $\boldsymbol{\rho}$ and the best candidate for the most likely tailbiting path of the trellis are updated. Moreover, the subtrellises in the list Q_T with $M\left(s_p^L\right)$ higher than the new value of $\boldsymbol{\rho}$ are also deleted from Q_T. This iterative process is repeated until the list Q_T is empty, with the decoded path being the candidate for the most likely tailbiting path stored in that moment.

Figure 1 shows an example of the decoding process with the proposed algorithm for the rate 1/2 convolutional code with polynomial generators $(g_1, g_2) = (7, 5)$ in octal. In Fig. 1a, the structure of the encoder and a transition of the trellis are depicted. The vector of log-likelihood ratios of the received sequence is

$$\mathbf{R} = [-0.45 \quad -0.18 \quad -1.26 \quad 1.25 \quad 1.02 \quad 0.21]$$

which corresponds to the encoded sequence $\mathbf{u} = [1 \quad 1 \quad 0 \quad 1 \quad 1 \quad 1]$. As it can be seen, there are two errors in the received sequence located in the first two positions of $\mathbf{R}$.

Figure 1b shows the survivor paths, marked in bold, at the end of the first stage of the decoding. In this case, the survivor path with the least metric is not tailbiting since it starts at state s_2 and ends at state s_0 (path $\mathbf{s}_{2,0}$). The tailbiting survivor with the least metric is the path ending at state s_3. Therefore, the value of $\boldsymbol{\rho}$ is set to $M\left(s_3^3\right)$ and the candidate for the best tailbiting path at the end of the first stage is the path $\mathbf{s}_{3,3}$.

The list Q_T is formed by the two subtrellises corresponding to the final states with an accumulated metric $M\left(s_p^3\right)$ lower than $\boldsymbol{\rho}$: $T(s_0)$ and $T(s_2)$. As $M\left(s_0^3\right) = 0$, the Viterbi

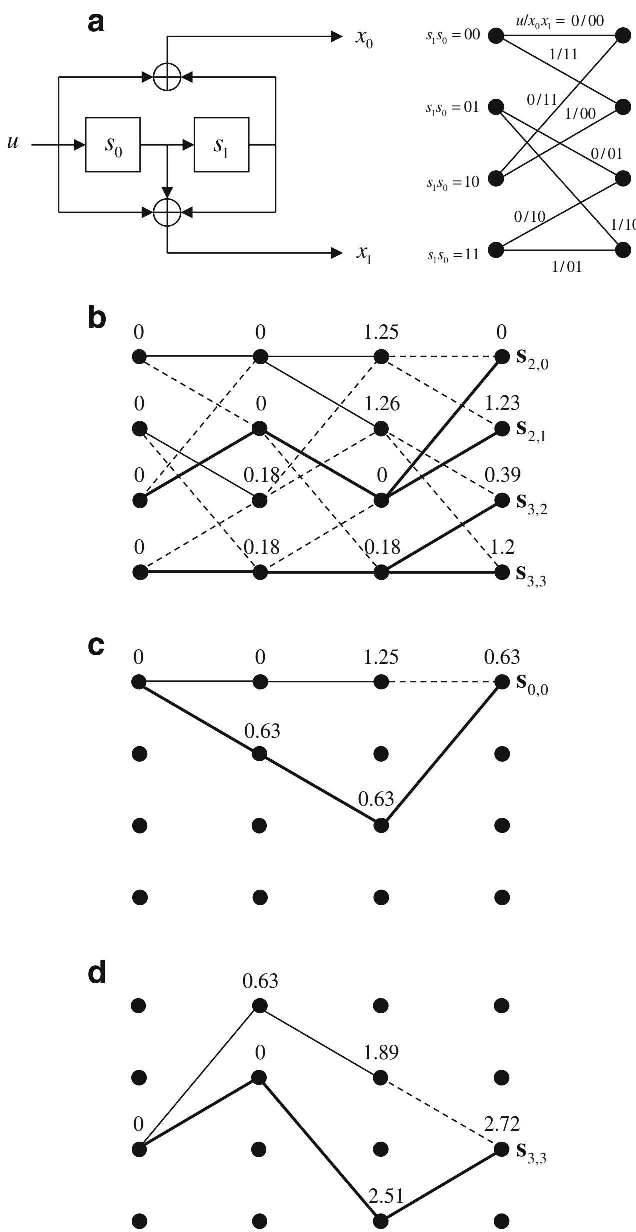

Fig. 1 Example of decoding with the proposed algorithm for the rate 1/2 convolutional code with $(g_1, g_2) = (7, 5)$. **a** Encoder structure and trellis transition. **b** First stage of the decoding ($\mathbf{s}_{i,j}$ indicates the survivor path ending at j and starting at i). **c**, **d** Application of the Viterbi algorithm to the subtrellises $T(s_0)$ and $T(s_2)$. The decoded path is the survivor of $T(s_0)$

algorithm is applied to subtrellis $T(s_0)$ first (Fig. 1c). The survivor path obtained in this subtrellis has an accumulated metric equal to 0.63, so the value of $\boldsymbol{\rho}$ and the candidate for the most likely tailbiting path are updated. Since $M\left(s_2^3\right) = 0.39 < 0.63$, the subtrellis $T(s_2)$ cannot be deleted from Q_T and a Viterbi decoding must be performed on this subtrellis as well (Fig. 1d). In this case, the survivor path has an accumulated metric of 2.72, so the value of $\boldsymbol{\rho}$ and the candidate tailbiting path do not change. Since the list Q_T is now empty, the algorithm stops and the decoded tailbiting path is the survivor path of subtrellis $T(s_0)$.

The following steps summarize the proposed algorithm:

1. Apply the Viterbi algorithm to the trellis T_s with the initial state metrics set to zero. Record the accumulated metric $M\left(s_p^L\right)$ of the survivor paths at the final states of the trellis.
2. If the survivor path with the least metric is also tailbiting (its initial and final states are the same), the algorithm stops and this path is decoded.
3. Set $\boldsymbol{\rho}$ as the least metric of the survivor path which is also tailbiting and store this path as the candidate for the most likely tailbiting path. If none of the survivors is tailbiting, $\boldsymbol{\rho}$ is set to ∞.
4. Load in Q_T the subtrellises $T(s_p)$ corresponding to final states s_p^L with accumulated metric $M\left(s_p^L\right)$ lower than $\boldsymbol{\rho}$. Sort them in ascending order of $M\left(s_p^L\right)$.
5. Apply the Viterbi algorithm to the subtrellis in the first position of Q_T. If the accumulated metric of the survivor path at the end of the subtrellis is lower than $\boldsymbol{\rho}$, update $\boldsymbol{\rho}$ and the candidate for the most likely tailbiting path.
6. Delete the subtrellis in the first position of Q_T and the subtrellises $T(s_p)$ with an accumulated metric $M\left(s_p^L\right)$ higher or equal than $\boldsymbol{\rho}$.
7. If the list Q_T is empty, the algorithm stops and the candidate for the most likely tailbiting path is decoded. If not, go to step 5.

Second stage of the EA algorithm* A graph is an ordered pair $G = (V, E)$ where V is a set of nodes and E is a set of edges which connect some of the nodes in V. If we consider V as the set of states of a trellis and E as the set of transitions in that trellis, any algorithm to find the least cost path in a graph (as the A* algorithm) can be applied to obtain the most likely path in the corresponding trellis with an appropriate definition of the transition metrics.

Once the first stage of the decoding has been performed, the EA* algorithm stores the survivor paths with an accumulated metric lower than $\boldsymbol{\rho}$, setting their final states as the initial nodes for the A* algorithm. The operation of the A* algorithm is based on an iterative update of the paths $\mathbf{s}_{a,p}^i = \left(s_a^0, \ldots, s_p^i\right)$ stored in an ordered queue named Q_P. This queue is arranged according to a cost function associated to each path in it. The first state of each path in Q_P is always the initial state of one of the candidate subtrellises obtained previously, while the last state of the path corresponds to a final or intermediate state of that subtrellis.

Additionally, the A* algorithm stores a table of visited states, B_V, where the first and the last states of the paths

which have ever been on the top of the queue Q_P are recorded. As it can be demonstrated, the items in B_V indicate the intermediate states of the trellis to which an optimal path ending at those states has already been found.

The key parameter of the A* algorithm is the cost function used to arrange the paths in Q_P since it sets the iterations required to obtain the least cost tailbiting path and, thus, the computational load of the decoding process. The cost function employed in the A* algorithm is always the addition of two terms:

$$f\left(\mathbf{s}_{a,p}^{i}\right) = g\left(\mathbf{s}_{a,p}^{i}\right) + h\left(\mathbf{s}_{a,p}^{i}\right). \tag{5}$$

The first term $g\left(\mathbf{s}_{a,p}^{i}\right)$ corresponds to the accumulated metric of the path from its initial state in the trellis, s_a^0, until the state it ends, s_p^i. Its value is derived from the following expression:

$$g\left(\mathbf{s}_{a,q}^{i+1}\right) = g\left(\mathbf{s}_{a,p}^{i}\right) + m\left(s_p^i, s_q^{i+1}\right) \tag{6}$$

with $g\left(\mathbf{s}_{a,a}^{0}\right) = 0$.

The function $h\left(\mathbf{s}_{a,p}^{i}\right)$ is named the heuristic function, and it corresponds to an optimistic estimation of the remaining cost required to reach the goal node, i.e. the final state of the subtrellis $T(s_a)$, from the last state of the path, s_p^i. The heuristic function is defined as

$$h\left(\mathbf{s}_{a,p}^{i}\right) = \max\left(0, M\left(s_a^L\right) - M\left(s_p^i\right)\right) \tag{7}$$

for i greater than 0. For $i = 0$, the value of $h\left(\mathbf{s}_{a,p}^{i}\right)$ is set to $M\left(s_a^L\right)$ if the zero-length path has never been on the top of Q_P or to $M'\left(s_a^L\right)$ if the zero-length path has been on the top of Q_P before. $M'\left(s_a^L\right)$ is defined as

$$M'\left(s_a^L\right) = M\left(s_a^L\right) + \min\left(\Delta\left(s_m^1\right), \Delta\left(s_n^2\right), \ldots, \Delta\left(s_a^L\right)\right). \tag{8}$$

The second term in (8) refers to the minimum value of $\Delta\left(s_p^i\right)$ corresponding to the sequence of states which form the survivor path of the first stage ending at state s_a^L. Thus, $M'\left(s_a^L\right)$ is the accumulated metric of the second best path ending at the same final state of that survivor path.

In each new iteration of the A* algorithm, the path on the top of L_c, $\mathbf{s}_{a,p}^{i} = \left(s_a^0, \ldots, s_p^i\right)$, is picked up and the pair $\left(s_a^0, s_p^i\right)$ is searched in the table B_V. If there is an entry corresponding to that pair, the path is discarded from Q_P since another path starting at s_a^0 and ending at s_p^i with a lesser cost has been found in a previous iteration. If not, the pair is stored in B_V and the successor paths of $\mathbf{s}_{a,p}^{i}$ are obtained. These successors are formed by the concatenation of $\mathbf{s}_{a,p}^{i}$ and the states of the trellis which can be reached from s_p^i with a branch of length one. The form of these successor paths will be $\mathbf{s}_{a,q}^{i+1} = \left(s_a^0, \ldots, s_p^i, s_q^{i+1}\right)$. Finally, the path $\mathbf{s}_{a,p}^{i}$ is deleted from Q_P and its successor paths are inserted according to the value of their cost function. If this cost is higher than $\boldsymbol{\rho}$, the successor path is deleted from Q_P. This process is repeated until the path on the top of Q_P is a tailbiting path of the trellis, that is, until its first and last states correspond to the same initial and final states of the trellis or until Q_P is empty.

The EA* algorithm also implements the following stopping rule aiming at decreasing the overall computational load of the decoding: if the last state of the path on the top of Q_P, $\mathbf{s}_{a,p}^{i} = \left(s_a^0, \ldots, s_p^i\right)$, corresponds to a state of the stored survivor path obtained in the first stage of the algorithm which ends at the initial state of the path on the top of Q_P, $\mathbf{s}_{b,a}^{L} = \left(s_b^0, \ldots, s_p^i, \ldots, s_a^L\right)$, then the algorithm stops and the path formed by the concatenation of the path on the top of Q_P and the rest of the survivor path from the state they join to its end is decoded. This early stopping rule can be applied since the cost function of the resulting path is equal to the cost function of the path on the top of Q_P.

Figure 2 illustrates the decoding with the EA* algorithm of the same sequence used previously. The first stage of the decoding is similar for both algorithms, so the outcome of this stage is the same as the one shown in Fig. 1b and the value of $\boldsymbol{\rho}$ is set to $M\left(s_3^3\right) = 1.2$. The zero-length paths loaded in Q_P correspond to the initial states of the subtrellises which pass to the second stage of the proposed algorithm, namely $\mathbf{s}_{0,0}^{0}$ and $\mathbf{s}_{3,3}^{0}$ in Fig. 2a. The cost functions of these paths are $f\left(\mathbf{s}_{0,0}^{0}\right) = 0$ and $f\left(\mathbf{s}_{3,3}^{0}\right) = 0.39$.

After the first iteration of the second stage of the EA*, the cost function of path $\mathbf{s}_{0,0}^{0}$ is updated to 0.63 according to (8). Now, the first position in Q_P corresponds to path $\mathbf{s}_{3,3}^{0}$ and its cost function is updated to 0.66 in the second iteration using (8) again. In the third iteration, the successors of $\mathbf{s}_{0,0}^{0}$ are obtained and inserted in Q_P, which will contain the paths $\mathbf{s}_{0,0}^{1}$, $\mathbf{s}_{0,1}^{1}$ and $\mathbf{s}_{3,3}^{0}$ (Fig. 2b). The values of their cost functions are 0, 0.63 and 0.66, respectively. In the fourth iteration, depicted in Fig. 2c, the successors of $\mathbf{s}_{0,0}^{1}$ are obtained, but in this case, they are not inserted in Q_P since their cost function is higher than $\boldsymbol{\rho}$ $\left(f\left(\mathbf{s}_{0,0}^{2}\right) = 1.25\right.$ and $\left.f\left(\mathbf{s}_{0,1}^{2}\right) = 1.26\right)$. Finally, Fig. 2d shows the fifth iteration of the decoding. In this iteration, the decoding ends since the path on the top of Q_P, $\mathbf{s}_{0,1}^{1}$, merges into the survivor path found in the first stage of the algorithm ending at state 0. The concatenation of both paths is the resulting decoded sequence.

The following steps summarize the EA* algorithm:

1. Apply the Viterbi algorithm to the trellis T_s with the initial state metrics set to zero. Record in each update

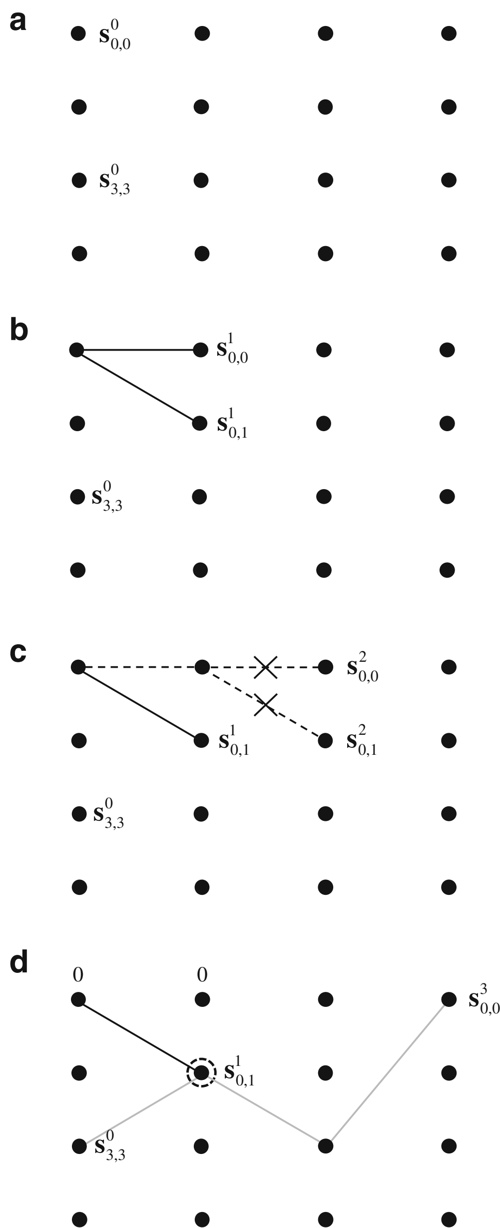

Fig. 2 Example of decoding with the EA* algorithm. **a** Paths in Q_P at the beginning of the second stage of the decoding. **b–d** Iterations 3, 4 and 5 of the second stage. The decoded path is the concatenation of the path $\mathbf{s}_{0,1}^{1}$ with the rest of the path ending at state s_0 found in the first stage of the decoding

the terms $M\left(s_p^i\right)$ and $\Delta\left(s_p^i\right)$ for all the states. Record also the survivor paths at the end of the algorithm.

2. If the survivor path with the least metric is also tailbiting (its initial and final states are the same), the algorithm stops and this path is decoded.
3. Set $\boldsymbol{\rho}$ as the least metric of the survivor path which is also tailbiting. If none of the survivors is tailbiting, $\boldsymbol{\rho}$ is set to ∞.
4. Discard all the survivor paths with metrics $M\left(s_p^L\right)$ higher or equal than $\boldsymbol{\rho}$.
5. Load in Q_P the zero-length paths corresponding to the final states of the paths not discarded in the previous step. Sort them in ascending order of their cost function values, which are $M\left(s_p^L\right)$.
6. If the queue is empty, the algorithm stops and the survivor found in the first stage with accumulated metric $\boldsymbol{\rho}$ is decoded.
7. If the path on the top of Q_P reaches the end of its subtrellis, this path is decoded.
8. If the path on the top of Q_P merges into the survivor path found in the first stage of the algorithm which ended in the final state of its subtrellis, the algorithm stops and the path formed by the concatenation of the path on the top of Q_P and the rest of the survivor path from the state they join is decoded.
9. If the path on the top of Q_P has zero length and its cost function corresponds to $M\left(s_p^L\right)$, change its cost function to $M'\left(s_p^L\right)$. If $M'\left(s_p^L\right)$ is greater than $\boldsymbol{\rho}$, the path is discarded. If not, rearrange the queue in ascending order of the cost function and go to step 6.
10. If the initial and final states of the path on the top of Q_P are stored in B_V, discard the path and go to step 6. If not, store them in B_V.
11. Find the successors of the path on the top of Q_P and compute their cost functions. Delete the path on the top of Q_P and rearrange the queue in ascending order of the cost function. If any path has a cost function higher than $\boldsymbol{\rho}$, delete it. Go to step 6.

3 Computational load and memory requirements of the proposed and the EA* algorithms

As it has been shown in the previous section, the decoding of tailbiting codes with the proposed and the EA* algorithms is divided in two stages. The first one is always executed and it consists in the application of the Viterbi algorithm. The second stage differs for the two algorithms and it can be skipped if the least cost path found in the first stage is already tailbiting. In order to measure the computational load and the memory requirements of the decoding process, both stages must be analysed.

Concerning the first stage of the decoding, the standard Viterbi algorithm requires 2 additions and 1 comparison to compute the survivor path at each state of the trellis. It must be noted that the computational load of the subtractions used to obtain the terms $\Delta\left(s_p^i\right)$ in the EA* algorithm is null, since these subtractions can substitute the comparisons required to select the survivor paths (the comparison can be

done with the sign of the subtraction). Therefore, the computational load of the first stage of both algorithms is $3NL$ operations.

Steps 2–4 of the proposed algorithm and 2–5 of the EA* algorithm are only executed once and the most complex task performed in these steps is the sorting of the accumulated metrics of the N survivor paths, so their computational load can be neglected.

The second stage of the proposed algorithm comprises steps 5–7 and it corresponds to the iterative application of the standard Viterbi algorithm to the subtrellises obtained in the first stage. The computational load of step 5 is $3NL$ operations to apply the Viterbi algorithm to the subtrellis plus one operation to compare the metric of the survivor path of the subtrellis with $\boldsymbol{\rho}$. If the value of $\boldsymbol{\rho}$ has been updated, some of the subtrellises in Q_T may be deleted (step 6). To do so, a comparison of the new value of $\boldsymbol{\rho}$ with the term $M\left(s_p^L\right)$ associated to the subtrellises of Q_T must be performed. Since Q_T is an ordered list, it can be done efficiently with a binary search in Q_T. The average number of required comparisons with this method is

$$w + 2 - \frac{2^{w+1}}{I_T + 1} \tag{9}$$

with $w = \lfloor \log_2(I) \rfloor$ and I_T as the number of items in Q_T. In the worst case, I_t is $N - 1$. Finally, step 7 can be performed at no cost since it can be carried out only with a flag check.

The second stage of the EA* algorithm (steps 6–11) corresponds to the application of the A* algorithm. First of all, the algorithm checks if the queue Q_P is empty (step 6). As in step 7 of the proposed algorithm, it can be assumed that this step has no cost. Then, one comparison is performed to establish if the path on the top of Q_P has reached the end of its subtrellis, i.e. it is checked if the length of the path is L (step 7). As for step 8, it requires one additional comparison to determine if the path on the top of Q_P merges into the corresponding survivor path found in the first stage of the decoding.

If a new zero-length path reaches the top of Q_P, $L - 1$ comparisons and one addition are performed to find and add the minimum of the values $\Delta\left(s_p^i\right)$ associated to the corresponding survivor path obtained in the first stage of the decoding (step 9).

As in step 6 of the proposed algorithm, the search of an item in the table B_V (step 10) will require (9) comparisons on average. If the search in B_V is unsuccessful, a new element (length and initial and final states of the path) must be inserted into the table. If not, the path is discarded and a new iteration of the algorithm is performed. Afterwards, the cost functions of the successors of the path on the top of Q_P are computed (step 11). This calculation requires two additions (g and f functions) and one subtraction (h function) per successor. The last step carried out is the insertion of the successor paths in Q_P according to their cost function (step 12), which takes (9) comparisons.

As regards the memory requirements of the decoding process, the best way to estimate the actual use of memory is to measure the maximum allocated memory employed at any point of the algorithms. In this case, the proposed and the EA* algorithms differ both in the first and the second stages of the decoding process.

The allocated memory in the first stage of the proposed algorithm is the memory employed by the standard Viterbi algorithm:

- $2N$ accumulated metrics for the survivor paths at time instants i and $i + 1$
- LN predecessor states to perform the traceback
- Ln received metrics

In the second stage of the proposed algorithm, the value of $\boldsymbol{\rho}$, the candidate for the least-cost tailbiting path and the subtrellises in Q_T with their associated metrics must be stored. Assuming that the memory required to record one state, one subtrellis or the variables $\boldsymbol{\rho}$ or $M\left(s_p^i\right)$ is 1 byte, the allocated memory in the worst case is $1 + L + 2N$ bytes. If we add the memory employed by the Viterbi algorithm, the maximum required memory of the proposed algorithm is $1 + L(N + n + 1) + 4N$ bytes.

On the other hand, although the first stage of the EA* algorithm is also the application of the Viterbi algorithm, in this case, the accumulated metrics of the survivor paths and the terms $\Delta\left(s_p^i\right)$ are also stored for all the states of the trellis. The allocated memory in this stage is as follows:

- LN accumulated metrics of the survivor paths $M\left(s_p^i\right)$
- LN metric differences $\Delta\left(s_p^i\right)$
- LN predecessor states to perform the traceback
- Ln received metrics

Therefore, the total memory allocated in the first stage of the EA* algorithm is $3LN + Ln$ bytes. This amount is fixed, and it does not depend on the block being decoded.

In the second stage of the decoding process, the stored data can be divided into three different groups: information from the previous stage, the ordered queue Q_P and the table B_V. The data from the first stage is as follows:

- LN accumulated metrics of the survivor paths
- Ln received metrics
- S survivor paths, with S being the number of survivor paths of the first stage with accumulated metric lower than $\boldsymbol{\rho}$

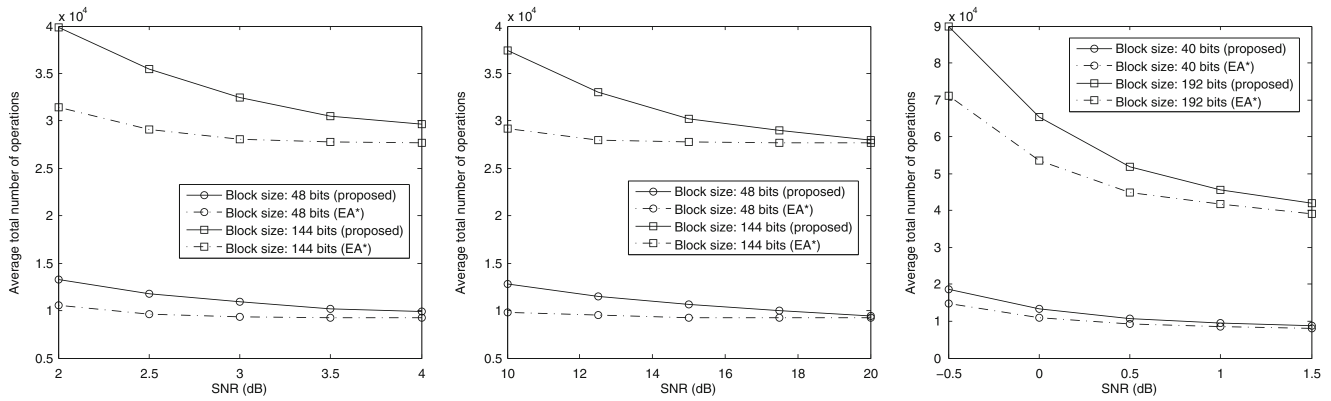

Fig. 3 Computational load for the proposed and the EA* algorithms. *Left*: WiMAX code, AWGN channel. *Center*: WiMAX code, ITU Vehicular A extended channel. *Right*: LTE code, AWGN channel

– S sequences of L metric differences $\Delta\left(s_p^i\right)$
– The value of $\boldsymbol{\rho}$ and the candidate for the most likely tailbiting path

Regarding B_V, each of its items is formed by three elements: the first state of the path, the last state of the path and its ending level. Finally, the queue Q_P stores the following elements for each path in it: the sequence of states that forms the path, its length and its associated cost functions (f and g functions). The total memory allocated in this stage is $1 + L(2S + N + n + 1) + I_P(3 + \text{path_length}) + 3I_V$ bytes, with I_P and I_V as the sizes of Q_P and B_V, respectively. It is worth mentioning that this amount will vary dynamically with each data block being decoded, which may impose great demands of memory resources to the decoder.

4 Simulation results

In this section, we show the performance of the proposed algorithm compared to that of the EA* algorithm over the additive white Gaussian noise (AWGN) and the ITU Vehicular A extended channels. The considered codes are the rate 1/2 binary tailbiting code used in WiMAX and the rate 1/3 code binary tailbiting code used in LTE. The memory of both codes is 6, and their polynomial generators are $(g_1, g_2) = (171, 133)$ and $(g_1, g_2, g_3) = (171, 133, 165)$ in octal, respectively. The sizes of the encoded blocks used in the simulations are also compliant with the WiMAX and LTE systems. The encoded data are interleaved and mapped to QPSK symbols, which are transmitted in an OFDM signal with a bandwidth of 5 MHz, a carrier frequency of 3.5 GHz and 512 subcarriers. It has been assumed soft decoding at the receiver and perfect channel estimation. The results have been obtained ensuring at least 100 reported errors for every simulation result. Similar simulations have been performed for other tailbiting codes which confirm the results presented in this section.

Since the proposed and the EA* algorithms are ML decoding algorithms, the results obtained in terms of block error rate (BLER) are the same for both of them. Nevertheless, the computational load and the memory requirements

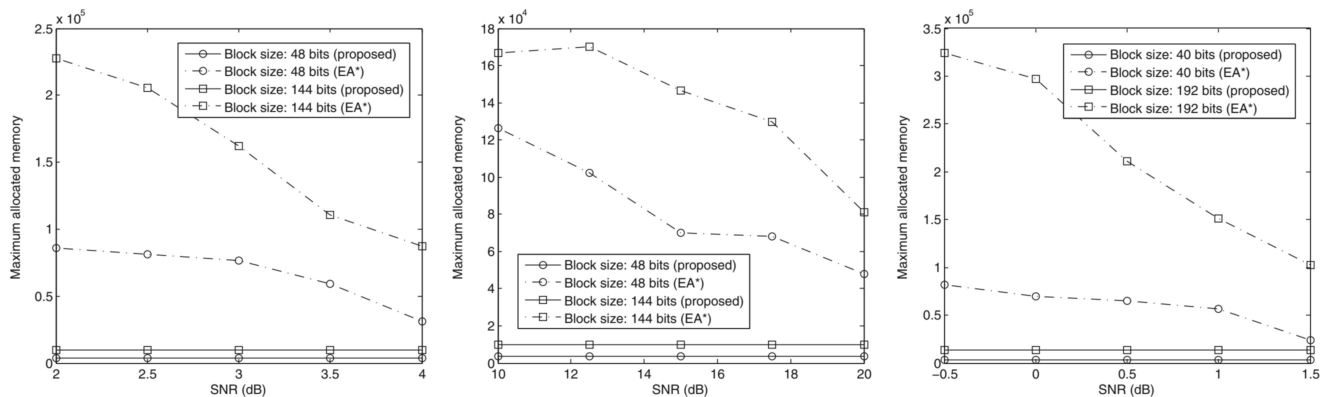

Fig. 4 Memory requirements for the proposed and the EA* algorithms. *Left*: WiMAX code, AWGN channel. *Center*: WiMAX code, ITU Vehicular A extended channel. *Right*: LTE code, AWGN channel

to perform this ML decoding will vary from one algorithm to the other.

Figures 3 and 4 compare the average computational load and the maximum required memory of both algorithms. This comparison has been carried out considering the theoretical analysis explained in Section 3 and taking into account both the first and the second stages of the decoding. For medium-to-low BLER, the average computational load of the EA* algorithm is about 5–10 % lower than that of the proposed algorithm, while for high BLER, the reduction is greater.

Although in these figures the proposed algorithm seems to work worse than the EA* algorithm, two considerations must be pointed out: first, concerning the queue Q_P and the table B_V, only the operations associated to searches in them have been considered, but not the operations or the elapsed time required to insert new elements. Secondly, although the A* algorithm is very efficient and obtains the least-cost tailbiting path in few iterations, it is also a sequential algorithm, which makes difficult to get any parallelization for its implementation in hardware. On the contrary, the computation of the survivor paths and their associated metrics at each transition of the trellis can be easily executed in parallel with the Viterbi algorithm. Additionally, the second stage of the proposed algorithm can be parallelized as well, executing simultaneously the Viterbi algorithm for the different subtrellises that reach this stage. In this case, if we have P processors and S subtrellises in the second stage, the decoding time could be reduced ideally by a factor $\lceil N/S \rceil$ at the cost of a P-factor increase of the memory requirements.

As regards the memory requirements of the algorithms, the best method to estimate the actual memory use of the decoding process is to measure the maximum allocated memory employed at any point of the execution of the algorithms. The worst case for the proposed algorithm happens when there has not been found any tailbiting survivor path in the first stage of the decoding. In this case, the maximum required memory of the proposed algorithm is $1 + L(N + n+1)+4N = 1+48(64+2+1)+4\cdot 64 = 3{,}473$ bytes when the block size is 48 bits and $1 + 144(64 + 2 + 1) + 4 \cdot 64 = 9{,}905$ bytes when the block size is 144 bits. On the contrary, the memory required by the EA* algorithm is 23 times higher than that of the proposed algorithm for high BLER and 8 times higher for low BLER. This reduction in the memory requirements obtained with the proposed algorithm is significantly high and of paramount importance for the implementation of this kind of maximum likelihood decoding strategies in real systems with limited resources.

5 Conclusions

A new optimal decoding algorithm for tailbiting codes is proposed in this work. The decoding is divided into two stages: in the first one, the Viterbi algorithm is used to prune the subtrellises where the least cost tailbiting path may be found. In the second stage, the same Viterbi algorithm is applied in these subtrellises to search the most likely tailbiting path. Simulation results show that the proposed algorithm achieves a marked decrease in the memory requirements and that it is more efficient to be implemented in real systems than other ML algorithms. For these reasons, the proposed algorithm is suitable for environments involving low resource devices.

Acknowledgments This work has been financed by the Spanish Government (Project TEC2011-29126-C03-03/TEC from MICINN and FEDER) and DGA-FSE.

References

1. ETSI (2013) 3GPP TS 36.212 v. 11.3.0. LTE evolved universal terrestrial radio access (E-UTRA): multiplexing and channel coding. ETSI, Sophia Antipolis Cedex
2. IEEE (2009) IEEE 802.16-2009. Air interface for fixed and mobile broadband wireless access system. IEEE, New York
3. Ma HH, Wolf JK (1986) On tail biting convolutional codes. IEEE Trans Commun 34(2):104–111
4. Cox RV, Sundberg CW (1994) An efficient adaptative circular Viterbi algorithm for decoding generalized tailbiting convolutional codes. IEEE Trans Veh Tech 43(2):57–68
5. Shao RY, Lin S, Fossorier MPC (2003) Two decoding algorithms for tailbiting codes. IEEE Trans Commun 51(10):1658–1665
6. Anderson JB, Hladik SM (2002) An optimal circular Viterbi decoder for the bounded distance criterion. IEEE J Sel Areas Commun 50(11):1736–1742
7. Ortin et al (2009) Two step SOVA-based decoding algorithm for tailbiting codes. IEEE Commun Lett 13(7):510–512
8. Hagenauer J, Hoeher P (1989) A Viterbi algorithm with soft-decision outputs and its applications. In: Proceedings of the IEEE GLOBECOM, Dallas
9. Pai H-T et al (2008) Low-complexity ML decoding for convolutional tail-biting codes. IEEE Commun Lett 12(12):883–885
10. Ortin J et al (2010) A low complexity maximum likelihood decoder for tailbiting trellis. IEEE Commun Lett 14(9):854–856
11. Wang X et al (2013) An efficient ML decoder for tail-biting codes based on circular trap detection. IEEE Trans Commun 16(4):1212–1221
12. Souza RD, Pimentel C, Muniz DN (2012) Reduced complexity decoding of convolutional codes based on the M-algorithm and the minimal trellis. Ann Telecommun 67:537–545