A Low-Complexity Maximum-Likelihood Decoder for Tail-Biting Convolutional Codes

Yunghsiang S. Han, Fellow, IEEE, Ting-Yi Wu, Po-Ning Chen, Senior Member, IEEE, and Pramod K. Varshney, Fellow, IEEE

Abstract—Due to the growing interest in applying tail-biting convolutional coding techniques in real-time communication systems, fast decoding of tail-biting convolutional codes has become an important research direction. In this work, a new maximum-likelihood (ML) decoder for tail-biting convolutional codes is proposed. It is named Bidirectional Priority-First Search Algorithm (BiPFSA) because Priority-First Search Algorithm has been used both in forward and backward directions during decoding. Simulations involving the antipodal transmission of $(2, 1, 6)$ and $(2, 1, 12)$ tail-biting convolutional codes over additive white Gaussian noise channels show that BiPFSA not only has the least average decoding complexity among state-of-the-art decoding algorithms for tail-biting convolutional codes but can also provide a highly stable decoding complexity with respect to growing information length and code constraint length. More strikingly, at high SNR, its average decoding complexity can even approach the ideal benchmark complexity, obtained under a perfect noise-free scenario by any sequential-type decoding. This demonstrates the superiority of BiPFSA in terms of decoding efficiency.

Index Terms—Tail-Biting Convolutional Codes, Maximum-Likelihood Decoding, Convolutional Codes, Viterbi Algorithm

I. Introduction

In digital communications, convolutional codes have been used widely to provide effective error protection capability. In order to clear the content of shift registers such that the encoding of the next information sequence can proceed directly without reinitialization, a suitable number of zeros, which are sufficient to fill up and hence reset the shift registers, are often appended at the end of an information bitstream to be encoded. In practice, this facilitates the design of decoding algorithms since the initial state of encoding shift registers is always certain for each information sequence and hence the decoder can always start the decoding process from the same root node of a code tree or a code trellis. As a side benefit, these zero tail bits can also enhance the error protection capability of convolutional codes. Such a performance enhancement, however, may be seriously diminished if a significant loss in code rate is introduced due to these zero tail bits particularly when the information sequence is short.

In the literature, several methods have been proposed to mitigate the code rate loss of the aforementioned (short-length) zero-terminated convolutional codes such as direct truncation and puncturing [1]. Alternatively, the so-called tail-biting convolutional codes [2], [3], [4] remove the loss in code rate in its entirety at the expense of the uncertainty introduced regarding the initial state of encoding shift registers at the decoder. Unlike a zero-terminated convolutional encoder that always starts from and ends at the all-zero state, a tail-biting convolutional encoder only ensures that the initial state and the final state are the same, where the specific state is decided by the previous information bitstream. Since any state, including the all-zero state, can be the initial state of a tail-biting convolutional encoder, the size of the space of possible transmissions that a decoding search needs to examine is multiplied by the number of possible states of encoding shift registers; therefore, the decoding complexity is dramatically increased.

Similar to the decoding of a zero-terminated convolutional code, the decoding of a tail-biting convolutional code is performed on a trellis, over which a codeword corresponds to a path that starts from and ends at the same state (but not necessarily the all-zero state). For convenience, the paths with the same initial and final state on a tail-biting convolutional code trellis are referred to as tail-biting paths. Since there is a one-to-one correspondence between codewords and tail-biting paths, the two terms are often used interchangeably.

Let the number of all possible initial states (equivalently, final states) of a tail-biting convolutional code trellis be denoted by M. Then, the trellis can be decomposed into M subtrellises that have the same initial and final states. By following the previous naming convention, these subtrellises are called tail-biting subtrellises, or simply subtrellises if no ambiguity arises from such an abbreviation. For convenience, we will denote a tail-biting convolutional code trellis by T and its ith subtrellis by

This work was supported by the National Natural Science Foundation of China (Grant No. 61671007), and the Ministry of Science and Technology, R.O.C. (Grant No. 105-2221-E-009-009-MY3 and 105-2917-I-564-048).

Y. S. Han is with the School of Electrical Engineering & Intelligentization, Dongguan University of Technology, Dongguan, China and is also with the Dept. of Electrical Engineering, National Taiwan University of Science and Technology, Taipei, Taiwan, R.O.C. (e-mail: yunghsiangh@gmail.com).

T.-Y. Wu is with the Dept. of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, Illinois, USA (e-mail:mavericktywu@gmail.com).

P.-N. Chen is with the Inst. of Communications Engineering & Dept. of Electrical and Computer Engineering, National Chiao-Tung University, Hsinchu City, Taiwan 30010, R.O.C. (email:poning@faculty.nctu.edu.tw).

P. K. Varshney is with the Department of Electrical Engineering and Computer Science, Syracuse University, Syracuse, NY 13244 USA (email: varshney@ecs.syr.edu).

T_i, where $0 \leq i \leq M-1$. It is clear that the decoding complexity of a tail-biting convolutional code will have a many-fold increase in comparison to that of a zero-terminated convolutional code of equal size, since all tail-biting paths in each subtrellis must be examined.

In order to reduce decoding complexity, several sub-optimal[1] decoding algorithms for tail-biting convolutional codes have been proposed in the literature [2], [3], [5], [6], [7], among which the Wrap-Around Viterbi Algorithm (WAVA) has the least decoding complexity [2]. As its name reveals, WAVA iteratively applies Viterbi Algorithm (VA) onto the trellis of a tail-biting convolutional code in a wrap-around fashion. During its execution, WAVA traverses not only the tail-biting paths but also those paths that start from and end at different states; hence, it may output a path corresponding to none of the codewords. Accordingly, at the end of each iteration, it examines whether or not the final output is a tail-biting path. If the output path is not a tail-biting path and the number of iterations is less than the maximum number of iterations allowed, another iteration is launched with the metrics of survivor paths in the previous iteration being the initial metrics. As a result, WAVA can be equivalently regarded as an application of VA onto an extended trellis, which is formed by a pre-specified number of trellises that are connected one after another in a tandem fashion. It has been shown by simulations that wrapping around at most four trellises is sufficient to obtain a near-optimal performance [2].

Under certain special code rates, a tail-biting convolutional code can be equivalently transformed to a quasi-cyclic (QC) code [8], [5]. Decoding of a tail-biting convolutional code can thus be performed over the trellis of equivalent QC code. Since the trellis size of an equivalent QC code usually has a multi-fold increase in comparison with that of the corresponding tail-biting convolutional code, suboptimal decoding algorithms, of which decoding complexities are less impacted by trellis size, are preferred. Some examples of such decoding algorithms are ordered statistics decoding algorithm [9] and box-and-match algorithm [10].

In situations where exact maximum-likelihood (ML) decoding performance is required, WAVA is no longer a suitable choice because it does not guarantee that the ML tail-biting path will be found. By performing VA on all tail-biting subtrellises, the ML tail-biting path can be obtained in a straightforward manner; however, such a brute force approach is impractical due to its high computational complexity.

In 2005, Bocharova et al. proposed an ML decoding algorithm for tail-biting convolutional codes, which is named Bidirectional Efficient Algorithm for Searching Trees (BEAST) [11]. Conceptually, by operating over all tail-biting subtrellises, BEAST explores, repetitively and simultaneously in both forward and backward directions, those nodes that have their decoding metrics below a certain threshold on each subtrellis. It keeps increasing the threshold at each step until an ML path is found. Simulation results provided in [11] show that BEAST is very efficient at high signal-to-noise ratios (SNRs).

One year later, another ML decoding algorithm for tail-biting convolutional codes was proposed by Shankar et al. [12], which we refer to as Creative Maximum-Likelihood Decoding Algorithm (CMLDA). CMLDA has two phases. The first phase applies VA onto the trellis of a tail-biting convolutional code to extract certain trellis information. Based on the trellis information obtained, Algorithm A* is performed on all subtrellises in the second phase to yield an ML decision. It has been shown in [12] that without sacrificing the optimality in performance, CMLDA reduces the decoding complexity from M executions of VA on M subtrellises, as required by a brute force approach, to approximately 1.3 VA executions. To further improve CMLDA in terms of decoding complexity, the authors in [13] and [14] redefine the heuristic function given in [12] and replace its Algorithm A* in the second phase by Priority-First Search Algorithm (PFSA). PFSA improves CMLDA in both average and maximum decoding complexities, particularly in the second phase [13], [14]. Nevertheless, the overall decoding complexities of both CMLDA and PFSA are dominated by their first-phase decoding complexities and are bounded from below by one VA execution over the trellis of tail-biting convolutional codes because they still retain the use of VA in their first phase.

In 2015, Qian et al. proposed a novel two-phase depth-first ML decoding algorithm, named Bounded Search (BS) [15]. It conceptually operates a depth-first search on the trellis in the first phase to hopefully rule out most subtrellises. Then, in the second phase, a bidirectional search algorithm is performed on the remaining subtrellises. Similar to other two-phase decoding algorithms, BS gives a low average decoding complexity at high SNRs.

In this paper, a new maximum-likelihood (ML) Bidirectional Priority-First Search Algorithm (BiPFSA) for tail-biting convolutional codes is proposed. Unlike other existing works, BiPFSA first performs PFSA on the backward trellis, and then applies PFSA again on all subtrellises in a forward manner based on the information retained from the previous phase. In order not to compromise the optimality in performance, a new ML decoding metric and a new evaluation function that are used respectively to guide the priority-first search in the first and second phases are elaborately devised in this work. Simulation results obtained by antipodally transmitting $(2, 1, 6)$ and $(2, 1, 12)$ tail-biting convolutional codes over AWGN channels show that the average decoding complexity of BiPFSA is considerably less than those of BEAST [11], PFSA [13] and BS [15] at all SNRs simulated. Even though BiPFSA guarantees to achieve the optimal performance whereas WAVA does not, BiPFSA is less complex in average decoding complexity compared to the near-optimal

[1] Optimal decoder in this work referrs to a maximum-likelihood (ML) decoder. It is well-known that when codewords are used equally probably, an ML decoder minimizes the average probability of decoding error and hence is optimal.

WAVA [2].

The rest of the paper is organized as follows. Section II introduces BiPFSA. Section III proves the optimality of BiPFSA. Section IV investigates by simulations the computational effort of BiPFSA over additive white Gaussian noise (AWGN) channels. Section V concludes the paper.

II. Bidirectional Priority-First Search Algorithm (BiPFSA)

Let $\mathscr{C}$ represent a binary $(n,\ k,\ m)$ tail-biting convolutional code with kL information bits, where k information bits are mapped to n code bits through linear convolutional circuitry with constraint length m. Such a system is sometimes referred to as $[N, K, d_{\min}] = [Ln,\ Lk,\ d_{\min}]$ block code since an (n, k, m) tail-biting convolutional code is also a block code of size 2^{kL} and length nL, where $d_{\min}$ is the minimum pair-wise Hamming distance of this code. The trellis T of the tail-biting convolutional code $\mathscr{C}$ has $M = 2^m$ states at each level, and is of $L + 1$ levels. Although only the tail-biting paths, which are required to have the same initial and final state, correspond to codewords of $\mathscr{C}$, we introduce an auxiliary super code $\mathscr{C}_{\rm sup}$ that consists of all binary words corresponding to paths on the trellis of $\mathscr{C}$. In other words, the paths considered in $\mathscr{C}_{\rm sup}$ can start from and end at different states.

Denote a binary codeword of $\mathscr{C}$ by $\boldsymbol{v} \triangleq (v_0, v_1, \ldots, v_{N-1})$. Define the hard-decision sequence $\boldsymbol{y} = (y_0, y_1, \ldots, y_{N-1})$ corresponding to the received vector $\boldsymbol{r} = (r_0,\ r_1,\ \ldots,\ r_{N-1})$ as

$$y_j \triangleq \begin{cases} 1, & \text{if } \phi_j < 0\ ; \\ 0, & \text{otherwise}\,, \end{cases} \tag{1}$$

where

$$\phi_j \triangleq \ln \frac{\Pr(r_j|0)}{\Pr(r_j|1)} \tag{2}$$

is the log-likelihood ratio (LLR) of the jth component r_j, and $\Pr(r_j|0)$ and $\Pr(r_j|1)$ denote the channel transition probabilities given code bits 0 and 1 were transmitted, respectively.

Then, the ML decoding output $\hat{\boldsymbol{v}} = (\hat{v}_0, \hat{v}_1, \ldots, \hat{v}_{N-1})$ for received vector $\boldsymbol{r}$ satisfies

$$\sum_{j=0}^{N-1} (y_j \oplus \hat{v}_j)|\phi_j| \le \sum_{j=0}^{N-1} (y_j \oplus v_j)|\phi_j| \quad \text{for all } \boldsymbol{v} \in \mathscr{C}, \tag{3}$$

where "$\oplus$" is the component-wise modulo-2 addition. We thereby define a path metric that can be applied universally to paths over trellis T or subtrellises $\{T_i\}_{i=0}^{M-1}$ as follows.

Definition 1: Let ℓ be a fixed integer satisfying $0 \le \ell \le L$. Give a received LLR vector $\boldsymbol{\phi} = (\phi_0, \phi_1, \ldots, \phi_{N-1})$ and its corresponding hard-decision vector $\boldsymbol{y} = (y_0, y_1, \ldots, y_{N-1})$. For a path with binary label $\boldsymbol{x}_{(\ell n-1)} \triangleq (x_0, x_1, \ldots, x_{\ell n-1})$, which ends at level ℓ in a graphical structure such as trellis T and subtrellises $\{T_i\}_{i=0}^{M-1}$, we define the path metric associated with it as[2]

$$m(\boldsymbol{x}_{(\ell n-1)}) \triangleq \sum_{j=0}^{\ell n-1} m(x_j)\,, \tag{4}$$

where

$$m(x_j) \triangleq (y_j \oplus x_j)|\phi_j| \tag{5}$$

is the bit metric of the $(j+1)$th binary label.

By this definition, the objective of ML decoding becomes to output a code path $\boldsymbol{x}_{(Ln-1)}$ in $\mathscr{C}$ such that its path metric $m(\boldsymbol{x}_{(Ln-1)})$ is smaller than or equal to that of all other code paths in $\mathscr{C}$.

As mentioned in the introduction section, CMLDA and PFSA perform VA in their first phase, of which the purpose is to extract specifically designed heuristic information for use in the second phase. Yet, running VA in the first phase inevitably induces a flooring constant to the overall decoding complexity, and this flooring constant that equals one VA execution not only grows exponentially with respect to the constraint length of the adopted code but cannot be reduced by increasing SNRs. Obviously, the only way to break this flooring barrier in the overall decoding complexity is to replace VA in the first phase by a search algorithm that is less complex than VA. This motivates the proposed BiPFSA in this paper.

Since PFSA is a main component of the proposed BiPFSA to be introduced, a concise description of PFSA is first given. PFSA is a search algorithm over a graph, in which each edge is associated with a nonnegative metric, and a path metric is given by the sum of consecutive edge metrics the path traverses. In order to find one of the shortest paths from a start node to a goal node, PFSA traverses along every path, starting from a start node, in a sequential edge-by-edge manner while calculating the edge metrics accumulated thus far for that path. It keeps extending the path hitherto with the smallest accumulative path metric until an extended path reaches a goal node.

To that end, a stack (or priority queue) that stores all accumulative path metrics is maintained in a way that the smallest one is always on top of the stack. Initially, the stack contains only those paths that start from and end at the same start node in the graph. Extending the top path then corresponds to the action of discarding the top path from the stack, followed by adding its successor paths to the stack. A sorting operation is subsequently performed to guarantee that the one with the smallest accumulative path metric is the next top path. The procedure is repeated until the next top path ends at a goal node, by which a shortest path from a start node to a goal node is found.

[2]We adopt the convention that the path $\boldsymbol{x}_{(-1)}$ denotes an initial path of zero length and its corresponding path metric $m(\boldsymbol{x}_{(-1)})$ is zero.

When two paths merge or meet during the above search process, the one with the larger accumulative path metric is removed from the stack. An efficient way to implement the removal of one of the two merging paths is to maintain a table, which records the ending nodes of all paths that have been extended (equivalently, that have been on top of the stack). As such, any top path that ends at a node hitherto recorded in the table is discarded and hence no extension is necessarily conducted.

In order to guarantee finding a shortest path or to speed up the search process, certain constraints on path metrics are imposed and a heuristic estimate about which successor path of a top path is more promising to lead to a shortest path ending at a goal node is often included.

As its name reveals, BiPFSA performs PFSA in each of its two phases. The general task of each phase of BiPFSA is described as follows.

- 1st Phase: A backward PFSA, guided by path metrics of backward paths, is applied to trellis T of super code $\mathscr{C}_{\rm sup}$ in a backward manner. Similar to what has been defined in Definition 1, for a backward path labelled backwardly with $\boldsymbol{x}_{[\ell n]} \triangleq (x_{N-1}, x_{N-2}, \ldots, x_{\ell n})$, its path metric is defined as[3]
$$m(\boldsymbol{x}_{[\ell n]}) \triangleq \sum_{j=1}^{N-\ell n} m(x_{N-j}). \tag{6}$$
A heuristic function value for each state at each level, denoted as $h_{\ell,i}$ for $0 \le \ell \le L$ and $0 \le i \le M-1$, is then computed as given later in Steps 4 and 8 of backward PFSA.
- 2nd Phase: A forward PFSA, guided by refined path metrics of forward paths, is applied on all subtrellises $\{T_i\}_{i=0}^{M-1}$ of $\mathscr{C}$. The refined path metric of a forward path $\boldsymbol{x}_{(\ell n-1)} = (x_0, x_1, \ldots, x_{\ell n-1})$ is defined as
$$m_{\rm ref}(\boldsymbol{x}_{(\ell n-1)}) = m(\boldsymbol{x}_{(\ell n-1)}) + h_{\ell,j}, \tag{7}$$
where j is the ending state of this path.

In the implementation of forward and backward PFSAs, two data structures will be used. They are respectively named Open Stack and Closed Table. The former stores the paths that have been visited by PFSA with the top element being the one with the minimum guided metric. The latter keeps track of those paths that have been on top of Open Stack at some previous time. They are so named because the paths in Open Stack can possibly be extended further and hence remain open, while those in Closed Table can no longer be extended and thus are closed for future extension.

It is important to mention that the number of distinct branch metrics is at most $L \cdot 2^n$; hence, it is possible to pre-calculate and store them for later access during the decoding process. This could sometimes improve the efficiency of the decoding search.

[3]We again adopt the convention that the path $\boldsymbol{x}_{[Ln]}$ denotes the initial backward path of zero length and its corresponding path metric $m(\boldsymbol{x}_{[Ln]})$ is zero.

With the underlying background given above, we now present the decoding procedures of backward PFSA.

⟨1st Phase: Backward PFSA⟩

Step 1. Initialize $h_{\ell,i} = \infty$ for $\mathbf{0} \le \ell \le L$ and $0 \le i \le M-1$. Set the current upper bound for path metrics to be infinity, i.e., $c_{\rm UB} = \infty$. Denote the backward path corresponding to path metric $c_{\rm UB}$ by $\boldsymbol{x}_{\rm UB}$. Initially set $\boldsymbol{x}_{\rm UB}$ as the null path. Compute all $L \cdot 2^n$ branch metrics and store them for later access.

Step 2. Load into Open Stack all backward paths that start from a state at level L of trellis T and that ends at the same state. There are M of them. The path metrics of these backward paths are all initialized as zero.

Step 3. Acquire the current top backward path in Open Stack. Record the top backward path and its path metric as $\boldsymbol{x}_{\rm TOP}$ and $c_{\rm TOP}$, respectively. Delete the top backward path from Open Stack. If $\boldsymbol{x}_{\rm TOP}$ reaches level 0 (i.e., ends at a state at level 0), go to Step 8.

Step 4. Let state i at level ℓ be the ending state of $\boldsymbol{x}_{\rm TOP}$. If $h_{\ell,i}$ is less than infinity, go to Step 3; otherwise, assign $c_{\rm TOP}$ to $h_{\ell,i}$.

Step 5. Obtain the successor backward paths of $\boldsymbol{x}_{\rm TOP}$ on the trellis, and compute their path metrics according to (6). Delete those successor backward paths with path metrics no less than $c_{\rm UB}$.

Step 6. If a successor backward path reaches level 0 with its path metric less than $c_{\rm UB}$, and it is also a tail-biting path, then replace $\boldsymbol{x}_{\rm UB}$ and $c_{\rm UB}$ by this successor backward path and its path metric, respectively. Repeat this step until all successor backward paths are examined.

Step 7. Insert the remaining successor backward paths from Step 5 into Open Stack and re-order the backward paths in Open Stack such that the top backward path has the smallest path metric. Go to Step 3.

Step 8. If $\boldsymbol{x}_{\rm TOP}$ is a tail-biting path, output it as the final ML decision, and stop BiPFSA without executing the second phase; otherwise, for all $0 \le \ell \le L$ and $0 \le i \le M-1$, assign $h_{\ell,i} = c_{\rm TOP}$ whenever $h_{\ell,i} = \infty$. Go to the second phase.

An example of backward PFSA in the first phase is given in Figure 1, where its corresponding evolution of Open Stack is listed in Figure 2. In this example, we consider a $(2, 1, 2)$ tail-biting convolutional code of length $N = kL = 5$ with generators $7, 5$ (octal). Upon reception of LLR vector $\boldsymbol{\phi} = (1.28, 0.99, 0.83, 1.09, -2.04, 0.36, -0.75, -1.10, -1.23, 1.45)$, Figure 1 shows how the four initial "single-node" backward paths, loaded in Step 2, migrate from level 5 to level 0. For a better illustration, we use four different colors to sketch the traces of these four initial backward paths and their successor backward paths, respectively. The accumulative

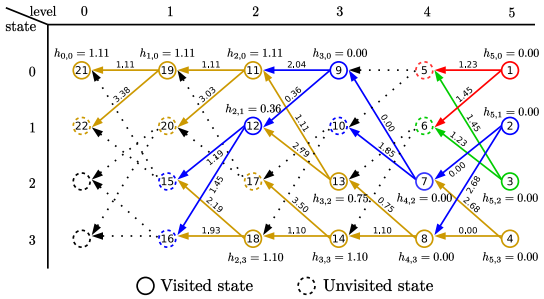

Fig. 1. An example of backward PFSA in the first phase for $(2, 1, 2)$ tail-biting convolutional code with generators $7, 5$ (octal) and with $L = 5$ levels. The received LLR vector $\boldsymbol{\phi} = (1.28, 0.99, 0.83, 1.09, -2.04, 0.36, -0.75, -1.10, -1.23, 1.45)$. Here, a visited state is the ending state of a path that has been on top of Open Stack; hence, a visited state is the ending state of either a top path to be extended further (such as circles with indices 1–4, 7–9, 11–14, 18–19) or a top path that reaches level 0 (such as the circle with index 21).

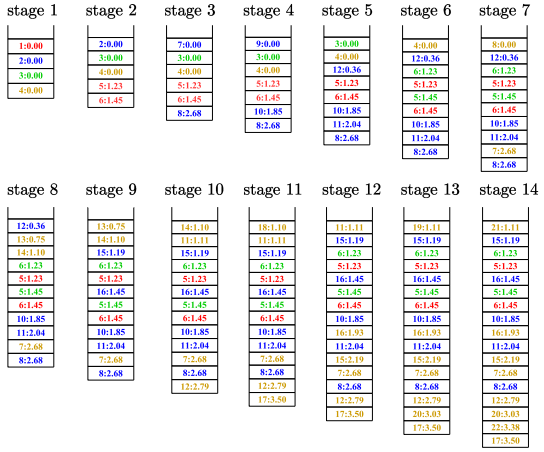

Fig. 2. Evolution of Open Stack in Figure 1

path metric (defined in (6)) of each backward path is marked above its last branch (i.e., edge). For example, the accumulative path metric along the backward path (4) → (8) → (13) → (11) is 1.11, and is marked above branch (13) → (11), where the circled numbers are the indices for nodes. As a result, $h_{2,0} = 1.11$.

In Figure 2, the backward paths stored in Open Stack at each stage are listed. Only the starting node, ending node and path metric of a backward path need to be recorded.[4] For example, the backward path (4) → (8) → (13) → (11) with path metric 1.11 appears as the second entry of Open Stack at stage 10, and is recorded as $(11 : 1.11)$. It can be seen from Figure 2 that Open Stack initially contains four "single-node" zero-metric backward paths ending at a node at level 5 (See Stage 1). Then the top backward path $(1 : 0.00)$ was acquired from Open Stack and was extended to its successor backward paths $(5 : 1.23)$ and $(6 : 1.45)$. This evolves to Stage 2, where the two successor backward paths $(5 : 1.23)$ and $(6 : 1.45)$ had been pushed into Open Stack. Backward PFSA continues extending the top backward path in Open Stack until the top backward path (i.e., $(21 : 1.11)$ at Stage 14) reaches level 0.

After the completion of the first phase, $\{h_{\ell,i}\}_{0\le\ell\le L, 0\le i\le M-1}$, as well as $\boldsymbol{x}_{\rm UB}$ and $c_{\rm UB}$ are retained for use in the second phase. Notably, $h_{\ell,i}$ can be regarded as an estimate of the path metric of a backward path from any state at level N to state i at level ℓ. By combining this estimate with the path metric of a forward path that ends at state i at level ℓ on a subtrellis, an estimate of the overall path metric of the combined tail-biting path of length N is obtained. It is reasonable to anticipate that if a good estimate of the path metric of a tail-biting path is used to guide the forward priority-first search in the second phase, the ML decision can be very efficiently obtained. This is the basic idea behind the design of the second phase of BiPFSA, of which the algorithmic procedure is given below.

⟨2nd Phase: Forward PFSA⟩

Step 1. Clean and reset Open Stack from the first phase. Re-load into Open Stack all "single-node" forward paths at level 0 of all subtrellises $\{T_i\}_{i=0}^{M-1}$. There are M of them. The refined path metrics of these "single-node" forward paths are respectively initialized as $\{h_{0,i}\}_{i=0}^{M-1}$. Retain $\boldsymbol{x}_{\rm UB}$ and $c_{\rm UB}$ from the first phase.

Step 2. If Open Stack is empty, output $\boldsymbol{x}_{\rm UB}$ as the final ML decision, and stop BiPFSA.[5]

Step 3. If the current top forward path $\boldsymbol{x}_{\rm TOP}$ in Open Stack has already been recorded in Closed Table, discard it from Open Stack and go to Step 2; otherwise, record the information of this top forward path in Closed Table.[6]

Step 4. According to the structure of M tail-biting subtrellises, compute the refined path metrics of all successor forward paths of the top forward path in Open Stack. Delete the top forward path from Open Stack. Delete those successor forward paths whose refined path metrics are no less than $c_{\rm UB}$.

Step 5. If a successor forward path reaches level L with its refined path metric less than $c_{\rm UB}$, replace $\boldsymbol{x}_{\rm UB}$ and $c_{\rm UB}$ by this successor forward path and its refined path metric, respectively. Repeat this step until all successor forward paths are examined. Delete all successor forward paths that reach level L.

[4]In this example, we can identify the starting node of a backward path by the color assigned to its trace and used in representing its entry in Open Stack.

[5]This step will never output an $\boldsymbol{x}_{\rm UB}$ that is equal to the null path. In other words, $\boldsymbol{x}_{\rm UB}$ is not the null path when Open Stack is empty. This is because $\boldsymbol{x}_{\rm UB}$ remains initially the null path in the second phase only when no tail-biting paths reaching level 0 were encountered in the first phase (cf. Step 6 of backward PFSA). In such a case, $c_{\rm UB} = \infty$ and hence Step 4 of forward PFSA will never delete any successor forward paths. As such, Step 5 of forward PFSA will replace $\boldsymbol{x}_{\rm UB}$ with the first successor forward path reaching level L, and Open Stack must be non-empty before this replacement.

[6]Uniquely identifying a forward path only requires the information of the initial state, the ending state and the ending level.

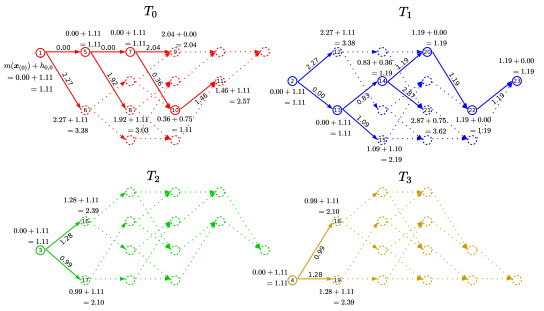

Fig. 3. An example of forward PFSA in the second phase for $(2, 1, 2)$ tail-biting convolutional code with generators $7, 5$ (octal) and with $L = 5$ levels. This is continued from the example in Figure 1.

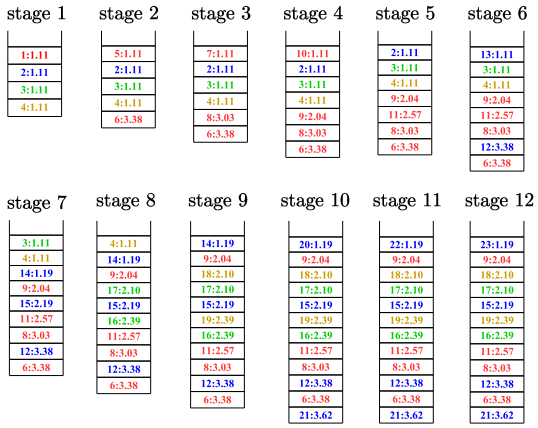

Fig. 4. Evolution of Open Stack in Figure 3

Step 6. Insert the remaining successor forward paths from Step 5 into Open Stack and re-order the forward paths in Open Stack according to ascending refined path metric values. Go to Step 2.

Continuing from the example in Figure 1, we illustrate respectively in Figures 3 and 4 how tail-biting convolutional subtrellis migrates and how Open Stack evolves in the second phase. The same as in Figure 1, four different colors are used respectively for four subtrellises, and the refined path metric in the form of (7) is marked above the last branch of a forward path. It can be noted from Figure 4 that Open Stack initially contains four "single-node" forward paths ending at a start node at level 0. Then the top forward path $(1 : 1.11)$ was acquired from Open Stack and was extended to its successor forward paths $(5 : 1.11)$ and $(6 : 3.38)$. This evolves to Stage 2, where the two successor forward paths $(5 : 1.11)$ and $(6 : 3.38)$ had been pushed into Open Stack. Forward PFSA continues extending the top forward path in Open Stack until the top forward path (i.e., $(23 : 1.19)$ at Stage 12) reaches level 5.

It can be noted from the algorithmic procedures of both phases that the top path ending at the state that has been visited at some previous time is eliminated. Since these paths must have a worse path metric (respectively, a worse refined path metric for the second phase) than the previously visited top path ending at the same state, elimination of them will not compromise the optimality in performance but help in speeding up the priority-first search. In order to verify whether a state has been visited, two different criteria are applied in the first and the second phases. In the first phase, we notice that $h_{\ell,i}$ is smaller than ∞ if, and only if, the path ending at state i at level ℓ has been visited. Hence, $h_{\ell,i}$ can be used to verify whether or not this state has been visited in the first phase. In the second phase, Closed Table is introduced to record all paths that have been on top of Open Stack and hence have been visited.

A common feature of priority-first search or any sequential-type search algorithms is that efficiency in sequential search can be improved when it starts its search from a more reliable component. As such, the averaged decoding complexity of BiPFSA can be further reduced by circularly shifting the received vector $\boldsymbol{r}$ according to the reliabilities of its components. By noting that the trellis as well as its corresponding subtrellises remain the same when ℓn-bit circular shifts are performed for any integer ℓ, BiPFSA can be used in an identical manner to determine the circularly shifted ML codeword with respect to the circularly shifted received vector. The ML codeword can then be easily recovered by the reverse action of the initial circular-shift operation. Based on a similar approach as in [16], we propose to circularly right-shift the received vector $\boldsymbol{r}$ by $\ell^* n$ bits before feeding it into BiPFSA, where

$$\ell^* = \arg \max_{0 \le \ell < L} \sum_{j=\ell n}^{((\ell+\lambda)n-1) \bmod N} |\phi_j| \tag{8}$$

and λ is a pre-specified window size for reliability measure. The additional computational complexity due to the determination of ℓ^* in (8) and the reverse left-shift at the end of BiPFSA is almost negligible in comparison with the decoding complexity of BiPFSA.

We point out at the end of this section that one can also employ forward PFSA in the first phase, and perform PFSA in a backward manner in the second phase. Considering that a decoder often favors listing the output code bits in a forward manner, we choose to perform PFSA in a forward manner in the second phase.

III. Optimality of BiPFSA

To confirm the optimality of BiPFSA, it suffices to prove that the ML codeword can always be found at the end of the algorithm. We begin with a theorem regarding the output of backward PFSA in the first phase.

Theorem 1: Among all length-N backward paths in $\mathscr{C}_{\text{sup}}$, backward PFSA in the first phase always finds the one with the smallest path metric. If the minimum-metric length-N backward path is also a tail-biting path, its corresponding codeword in $\mathscr{C}$ is an ML one.

Proof: Backward PFSA exits only possibly from Step 8. In other words, backward PFSA exists only when

$\boldsymbol{x}_{\rm TOP}$ reaches level 0. Since the path metric is non-decreasing along every backward path on trellis T, any length-N backward path, extended from some backward path currently in Open Stack, must have a path metric no less than that of the current top backward path. Hence, when $\boldsymbol{x}_{\rm TOP}$ reaches level 0, the one with the smallest path metric among all length-N backward paths in $\mathscr{C}_{\rm sup}$ must be $\boldsymbol{x}_{\rm TOP}$ (since it has the smallest path metric among all backward paths currently in Open Stack). By noting that the tail-biting paths of T are contained in the set of all length-N paths in T, $\boldsymbol{x}_{\rm TOP}$ must have the smallest path metric among all tail-biting paths in T if $\boldsymbol{x}_{\rm TOP}$ is also a tail-biting path; therefore, its corresponding codeword in $\mathscr{C}$ is an ML one. ■

It may occur that Step 8 of backward PFSA outputs an $\boldsymbol{x}_{\rm TOP}$ that is not a tail-biting path. In such a situation, forward PFSA in the second phase will be launched. The task of finding the ML tail-biting path must then be completed by forward PFSA in the second phase, for which the proof will be given after two preliminary lemmas are established.

Lemma 1: Let $\boldsymbol{x}_{((\ell+1)n-1)}$ label an immediate successor forward path of the forward path labelled by $\boldsymbol{x}_{(\ell n-1)}$ over subtrellis T_i. Suppose the ending states of $\boldsymbol{x}_{(\ell n-1)}$ and $\boldsymbol{x}_{((\ell+1)n-1)}$ are respectively i_1 and i_2. Then,

$$h_{\ell,i_1} \le h_{\ell+1,i_2} + m(\boldsymbol{x}_{((\ell+1)n-1)}) - m(\boldsymbol{x}_{(\ell n-1)}). \qquad (9)$$

Proof: During the execution of backward PFSA, $h_{\ell,i}$ may be assigned at two instances: (i) A backward path $\boldsymbol{x}_{[\ell n-1]}$ ending at state i at level ℓ had been the top backward path in Open Stack and hence $h_{\ell,i} = m(\boldsymbol{x}_{[\ell n-1]})$; and (ii) no backward path ending at state i at level ℓ had been the top backward path in Open Stack and hence $h_{\ell,i}$ was assigned when Step 8 of backward PFSA was executed and is equal to $c_{\min}$, where $c_{\min}$ is the minimum path metric among all length-N paths in T. This lemma can then be substantiated by the four combinations of how h_{ℓ,i_1} and $h_{\ell+1,i_2}$ were assigned.

1) $h_{\ell+1,i_2} = m(\boldsymbol{x}_{[(\ell+1)n-1]})$ and $h_{\ell,i_1} = m(\boldsymbol{x}_{[\ell n-1]})$: Since $\boldsymbol{x}_{[\ell n-1]}$ was the top backward path, $m(\boldsymbol{x}_{[\ell n-1]})$ must be no larger than the path metric of the successor backward path of $\boldsymbol{x}_{[(\ell+1)n-1]}$ ending at state i_1 at level ℓ, i.e.,

$$\begin{aligned} m(\boldsymbol{x}_{[\ell n-1]}) &\le m(\boldsymbol{x}_{[(\ell+1)n-1]}) \\ &+ [m(\boldsymbol{x}_{((\ell+1)n-1)}) - m(\boldsymbol{x}_{(\ell n-1)})] \end{aligned} \qquad (10)$$

which is exactly the desired result of (9).

2) $h_{\ell+1,i_2} = m(\boldsymbol{x}_{[(\ell+1)n-1]})$ and $h_{\ell,i_1} = c_{\min}$: In this case, the successor backward path of $\boldsymbol{x}_{[(\ell+1)n-1]}$ ending at state i_1 at level ℓ must stay in Open Stack but had never been on top of Open Stack, which implies

$$\begin{aligned} c_{\min} &\le m(\boldsymbol{x}_{[(\ell+1)n-1]}) \\ &+ \left[m(\boldsymbol{x}_{((\ell+1)n-1)}) - m(\boldsymbol{x}_{(\ell n-1)})\right] \end{aligned} \qquad (11)$$

and (9) is thus confirmed.

3) $h_{\ell+1,i_2} = c_{\min}$ and $h_{\ell,i_1} = m(\boldsymbol{x}_{[\ell n-1]})$: In this case, $\boldsymbol{x}_{[\ell n-1]}$ was the top backward path before $\boldsymbol{x}_{\rm TOP}$ in Step 8 of backward PFSA was. Thus, $m(\boldsymbol{x}_{[\ell n-1]}) \le c_{\min}$ and hence $h_{\ell,i_1} \le h_{\ell+1,i_2}$, which again validates (9).

4) $h_{\ell+1,i_2} = h_{\ell,i_1} = c_{\min}$: (9) trivially holds in this case. ■

Lemma 2: The refined path metric is a non-decreasing function along every forward path on subtrellises. In other words, for $0 \le \ell < L$,

$$m(\boldsymbol{x}_{(\ell n-1)}) + h_{\ell,i_1} \le m(\boldsymbol{x}_{((\ell+1)n-1)}) + h_{\ell+1,i_2} \qquad (12)$$

where $\boldsymbol{x}_{((\ell+1)n-1)}$ is an immediate successor forward path of $\boldsymbol{x}_{(\ell n-1)}$ over a subtrellis, and $\boldsymbol{x}_{(\ell n-1)}$ and $\boldsymbol{x}_{((\ell+1)n-1)}$ end at states i_1 and i_2, respectively.

Proof: This is an immediate consequence of Lemma 1. ■

An important remark should be made before we present the proof of the optimality of forward PFSA. As we have stated after Definition 1, finding an ML code path is equivalent to outputting a code path $\boldsymbol{x}_{(Ln-1)}$ in $\mathscr{C}$ such that its path metric $m(\boldsymbol{x}_{(Ln-1)})$ is smaller than or equal to that of all other code paths in $\mathscr{C}$. Since $h_{L,i} = 0$ for $0 \le i \le M-1$, the same statement also holds for the refined path metric.

Theorem 2: BiPFSA is guaranteed to find an ML code path in $\mathscr{C}$.

Proof: If BiPFSA exits in the first phase, Theorem 1 has confirmed that an ML code path will be outputted. It remains to show when Step 8 of backward PFSA outputs an $\boldsymbol{x}_{\rm TOP}$ that is not a tail-biting path, forward PFSA in the second phase can find an ML tail-biting path over subtrellises $\{T_i\}_{i=0}^{M-1}$. With this objective, it suffices to prove that Steps 3, 4 and 5 of forward PFSA will never delete all ML tail-biting paths (if there are multiple ones).

Suppose in Step 3 of forward PFSA, the current top forward path $\boldsymbol{x}_{\rm TOP}$ has already been stored in Closed Table due to the visit of a previous top forward path $\tilde{\boldsymbol{x}}_{(\ell n-1)}$. Since $\boldsymbol{x}_{\rm TOP}$ must be an offspring of some forward path that once coexisted with $\tilde{\boldsymbol{x}}_{(\ell n-1)}$ in Open Stack at the time $\tilde{\boldsymbol{x}}_{(\ell n-1)}$ was on top, Lemma 2 indicates that the ending state of both $\boldsymbol{x}_{\rm TOP}$ and $\tilde{\boldsymbol{x}}_{(\ell n-1)}$ at level ℓ is i, and

$$m(\tilde{\boldsymbol{x}}_{(\ell n-1)}) + h_{\ell,i} \le m(\boldsymbol{x}_{\rm TOP}) + h_{\ell,i}. \qquad (13)$$

Consequently, deletion of $\boldsymbol{x}_{\rm TOP}$ will never eliminate all ML tail-biting paths because those ML paths with $\boldsymbol{x}_{\rm TOP}$ being a portion of it (if they exist) can only have the same path metric as that of any ML path with $\tilde{\boldsymbol{x}}_{(\ell n-1)}$ being a portion of it.

Regarding Step 4, we argue that $c_{\rm UB}$ is the refined path metric of some tail-biting path, and hence it is an upper bound of the refined path metric of the final ML tail-biting path. We then distinguish between two cases:

1) If the refined path metric of the final ML tail-biting path is strictly smaller than $c_{\rm UB}$, then Lemma 2 indicates that the tail-biting path, extended from the successor forward path with refined path metric no less than $c_{\rm UB}$, can never be an ML tail-biting path; hence, deletion of this successor forward path will never compromise the optimality in performance.
2) If the refined path metric of the final ML tail-biting path is equal to $c_{\rm UB}$, then deletion of this successor forward path will not delete all ML tail-biting paths because at least one is kept in $\boldsymbol{x}_{\rm UB}$.

Regarding Step 5, forward PFSA has kept in $\boldsymbol{x}_{\rm UB}$ the best tail-biting path among all paths, which have been explored thus far, and which have reached level L during the priority first search. Hence, deletion of all successor forward paths that reach level L will not sacrifice the optimality in performance.

The proof of the theorem is thus completed. ■

IV. Experiments over AWGN Channels

In this section, we investigate the computational effort as well as the word error rate of the proposed ML decoding algorithm over additive white Gaussian noise (AWGN) channels via simulations. We assume that the transmitted binary codeword $\boldsymbol{v} = (v_0, v_1, \ldots, v_{N-1})$ is binary phase-shift keying (BPSK) modulated. The received vector $\boldsymbol{r} = (r_0, r_1, \ldots, r_{N-1})$ is thus given by

$$r_j = (-1)^{v_j}\sqrt{\mathcal{E}} + z_j \quad \text{for } 0 \le j \le N-1, \tag{14}$$

where $\mathcal{E}$ is the signal energy per channel bit, and $\{z_j\}_{j=0}^{N-1}$ are independent noise samples of a white Gaussian process with single-sided noise power per hertz N_0. The signal-to-noise ratio (SNR) is, therefore, given by $\text{SNR} \triangleq \mathcal{E}/N_0$. In order to account for the code redundancy for different code rates, we use the SNR per information bit in the following discussions, i.e.,

$$\text{SNR}_{\rm b} = \frac{N\mathcal{E}/K}{N_0} = \frac{n}{k}\left(\frac{\mathcal{E}}{N_0}\right). \tag{15}$$

Note that for AWGN channels, the metric associated with a path $\boldsymbol{x}_{(\ell n-1)}$ in Definition 1 can be equivalently simplified to

$$m(\boldsymbol{x}_{(\ell n-1)}) \triangleq \sum_{j=0}^{\ell n-1} (y_j \oplus x_j)|r_j|\,. \tag{16}$$

Two tail-biting convolutional codes are used in simulations. They are respectively $(2, 1, 6)$ and $(2, 1, 12)$ tail-biting convolutional codes with generators $103, 166$ (octal) and $5133, 14477$ (octal). Subject to information lengths of $L = 12$ and 48, the former is exactly $[24, 12, 8]$ extended Golay code [17], while the latter is equivalent to $[96, 48, 16]$ block code [18].

In our simulations, the number of branch (arithmetic) computations instead of metric computations are accumulated. For example, (16) can be rewritten as

$$m(\boldsymbol{x}_{(\ell n-1)}) = 0 + m(x_0^{n-1}) + \cdots + m(x_{(\ell-1)n}^{\ell n-1})\,, \tag{17}$$

where we abuse the notation by denoting the branch metric as

$$m(x_{un}^{(u+1)n-1}) = \sum_{j=un}^{(u+1)n-1} (y_j \oplus x_j)|r_j|\,. \tag{18}$$

Implementation of (17) then requires ℓ additions of branch metrics, starting from the initial value of zero metrics (See Step 2 of backward PFSA). Note that the number of distinct branch metrics, corresponding to $(x_{un}, x_{un+1}, \ldots, x_{(u+1)n-1})$, is at most 2^n, and they can be pre-calculated and stored for later access during the decoding process. The decoding complexity is thus dominated by the number of addition operations (as well as the number of memory accesses) of branch metrics for the calculation of path metrics. As a result, the decoding complexity can be measured as the number of path metric updates. For example, the number of path metric updates required in Figures 1 and 3 is equal to 26 (1st phase) + 19 (2nd phase) = 45, as indicated by those solid (branch) lines. In order to compensate for the effect of code length, decoding complexity is usually normalized by the number of information bits. As an example, Figures 1 and 3 requires $45/5 = 9$ path metric updates per information bit.

Before presenting our simulation results, three remarks on the practice of sequential-type decoding algorithms are provided. First, the computational effort of a sequential-type search algorithm includes not only the evaluation of path metrics but also the effort spent in searching and reordering the stack elements. By adopting the priority-queue data structure, often referred to as HEAP [19], the latter effort can be made comparable to that of the former. One can further employ a hardware-based stack structure [20] and attain a constant complexity for each stack insertion operation.

Second, all the ML decoders mentioned in this work require a stack-like data structure for decoding, and the cost of stack maintenance is in general different for different decoders. The path metric update, however, is the most repeated routine during a decoding process. The runtime due to other computational efforts such as initialization is relatively small. The above two remarks justify the usual adoption of the number of path metric updates per information bit as a measure of algorithmic complexity for sequential-type search algorithms.

Third, the sizes of stack-like data structures for BiPFSA and the ML decoders to be compared with vary at each simulation. However, the memory requirement other than stack-like structures, such as the Closed Table or the table to store h-function values, are fixed, as listed in Table I. Note that although the Closed Tables[7] for BEAST, PFSA and BiPFSA have a significantly larger size than BS, they are sparse and hence can be implemented by a Hash table [21] of size $2^m \times L$ with a negligible computational cost.

[7] The Closed Tables for BEAST and BS store the searched paths during forward and backward searches in order to identify whether the forward tree meets the backward tree.

TABLE I
Static memory requirement for BEAST, PFSA, BS, and BiPFSA

	BEAST	PFSA	BS	BiPFSA
Closed Table (Hash Implementation)	$2^m \times L \times 2^m$ ($2^m \times L$)	$2^m \times L \times 2^m$ ($2^m \times L$)	$2^m \times L$ (Not applied)	$2^m \times L \times 2^m$ ($2^m \times L$)
h-function Table	None	$2^m \times L$	None	$2^m \times L$

We are now ready to present the simulation results for decoding complexities as well as the corresponding word error rates (WERs) of BiPFSA. Four decoding algorithms that BiPFSA is compared with are BEAST, BS, PFSA and WAVA. These four algorithms are of very different characteristics in their designs, and hence each can be considered a typical representative of its own kind. WAVA is perhaps the most widely known suboptimal decoder for tail-biting convolutional codes; BEAST performs bi-directional search and is generally regarded as the most efficient ML decoder in average decoding complexity at high SNRs; BS is based on bounded search and is the most recent ML decoder among the four; and PFSA is a two-phase decoding algorithm. Since CMLDA and PFSA are of similar two-phase structure and PFSA has been shown to outperform CMLDA in both average and maximum decoding complexities, CMLDA will not be included in our simulations.

For ease of designating the parameters adopted in our simulations, BiPFSA with window size λ (that decides the amount of initial circular-shifts via (8)) and WAVA that wraps around at most I trellises are parameterized as BiPFSA(λ) and WAVA(I), respectively. The same circular-shift pre-processing can also be applied to PFSA in [13], which is similarly denoted as PFSA(λ). In all experiments, it is ensured that at least 100 word errors occur so that there is no bias in simulation results.

As a reference, we first show in Figure 5 the WER performance of WAVA($I = 2$) as well as that of an ML decoder. Note that BEAST, BiPFSA(λ), BS and PFSA(λ) are all ML decoders and hence should all achieve the ML performance. Figure 5 indicates that at WER $= 10^{-3}$, WAVA($I = 2$) has about 0.4 dB coding loss relative to the ML performance when $[24, 12, 8]$ extended Golay code is employed. A smaller 0.25 dB coding loss for WAVA($I = 2$) is observed at WER $= 10^{-3}$ when $[96, 48, 16]$ tail-biting block code is used instead.

We next investigate the decoding complexities of BEAST, BiPFSA($\lambda = 6$), BS, PFSA($\lambda = 6$) and WAVA($I = 2$) for $[24, 12, 8]$ extended Golay code and summarize the results in Figure 6. Since a sequential decoder must traverse at least an ML path, a benchmarking lower bound to decoding complexity is equal to $2^k/k$, measured in path metric updates per information bit. As a reference, this bound is also plotted in Figures 6 and 8.

Three observations can be made from Figure 6. First, the average numbers of per-information-bit path metric updates of WAVA($I = 2$) and PFSA($\lambda = 6$) are clearly lower bounded by that of one VA execution over trellis, which is a constant equal to $(2^m \times 2^k)/k = (M \times 2^k)/k =$

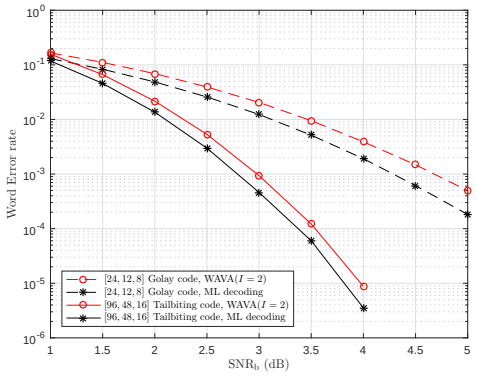

Fig. 5. Word error rates (WERs) of WAVA($I = 2$) and an ML decoder (such as BEAST, BiPFSA(λ), BS and PFSA(λ)) for $[24, 12, 8]$ extended Golay code and $[96, 48, 16]$ tail-biting block code.

$2^6 \times 2 = 128$. Second, BiPFSA($\lambda = 6$) significantly outperforms all other decoding algorithms in average decoding complexity at all SNRs. Third, the average decoding complexity of BiPFSA($\lambda = 6$) approaches the benchmark lower bound at high SNR, which confirms the superior efficiency of BiPFSA($\lambda = 6$), as the benchmarking lower bound can only be ideally achieved under a perfect noise-free scenario.

In certain practical applications, maximal decoding complexity is considered of comparable importance to average decoding complexity, especially when sequential-type decoding algorithms are regarded. We address this consideration by investigating the maximal decoding complexity of the decoding algorithms in Figure 6. The results are tabulated in Table II. We then observe from Table II that the maximum number of path metric updates of WAVA($I = 2$), as expected, remains a constant for all SNRs, and is equal to the computational effort of running VA over trellis twice, i.e., $2^m \times 2^k \times I = 2^6 \times 2 \times 2 = 256$. We also note that except for WAVA($I = 2$) and PFSA($\lambda = 6$), BiPFSA($\lambda = 6$) has the least maximum decoding complexity in comparison with BEAST and BS. More strikingly, the maximum number of path metric updates of BiPFSA($\lambda = 6$) is even lower than that of WAVA($I = 2$) at high SNR.

It is noted that the maximum numbers of path metric updates listed in Table II may sometimes increase as SNR grows. This is due to the fact that the numbers recorded in Table II are the largest decoding complexities that ever occurred during the simulation runs. Therefore, as an additional consideration, we examine the standard deviations of decoding complexities corresponding to the curves in Figure 6. The results in Figure 7 then indicate that as

far as the standard deviation of decoding complexity is concerned, the winner among BEAST, BiPFSA($\lambda = 6$), BS and WAVA($I = 2$) is either BEAST or BiPFSA($\lambda = 6$), depending on whether $\mathrm{SNR_b}$ is larger or smaller than 3 dB. When only the two possible winners are considered, BiPFSA($\lambda = 6$) obviously has a more stable standard deviation in decoding complexity across different SNRs. Our simulations also indicate that PFSA($\lambda = 6$) has a significantly smaller standard deviation in decoding complexity than the other four decoders, which suggests that the decoding complexity of performing PFSA in its second phase is not at all variable since it employs a constant-complexity VA in its first phase.

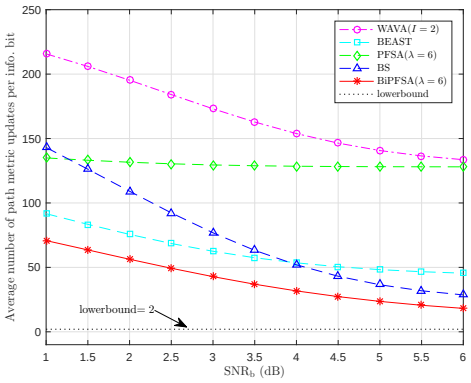

Fig. 6. Sample average numbers of path metric updates per information bit of BEAST, BiPFSA($\lambda = 6$), BS, PFSA($\lambda = 6$) and WAVA($I = 2$) for $[24, 12, 8]$ extended Golay code

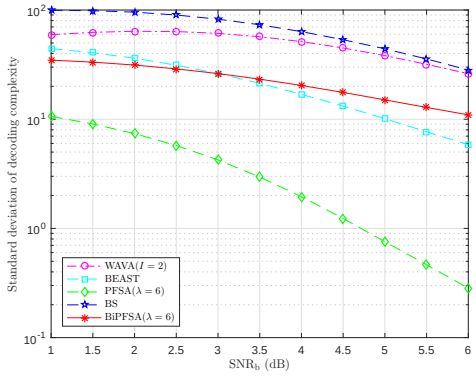

Fig. 7. Sample standard deviations of the numbers of path metric updates per information bit of the decoding algorithms in Figure 6 for $[24, 12, 8]$ extended Golay code

We repeat the above experiments for the $[96, 48, 16]$ tail-biting block code. This code has the largest free distance among all tail-biting codes of the same code length and code size [18]. Its constraint length is 12, and hence its code trellis has $2^{12} = 4096$ states at each level. With such a large number of states, an extremely high decoding complexity is anticipated. Efficient decoding of this code is thus a challenge.

Similar to Figure 6, we present in Figure 8 the average decoding complexities of BEAST, BiPFSA($\lambda = 12$), BS, PFSA($\lambda = 12$) and WAVA($I = 2$) for the $[96, 48, 16]$ tail-biting block code. We change the reliability window size from $\lambda = 6$ to $\lambda = 12$ to cope with the long code length and the large code size of the tail-biting convolutional code under consideration. It can be observed from this figure that BiPFSA($\lambda = 12$) remains much superior to other decoding algorithms in average decoding complexities, and approaches the ideal lower bound at high SNR. In contrast, the average decoding complexities of BEAST and BS dramatically increase at low SNR and become worse than those of PFSA($\lambda = 12$) and WAVA($I = 2$) when $\mathrm{SNR_b}$ is below 1.5 dB.

In Table III, the maximum decoding complexities for the five decoders that we simulate with are shown. We observe that BiPFSA($\lambda = 12$) beats all other decoding algorithms in maximum decoding complexity at high SNRs, including PFSA($\lambda = 12$) and WAVA($I = 2$). In particular, the maximum decoding complexity of BiPFSA($\lambda = 12$) can save up to 5.5×10^3 path metric updates per information bit at $\mathrm{SNR_b} = 4$ dB when it is compared with PFSA($\lambda = 12$).

The experiment on standard deviations of decoding complexities for the $[96, 48, 16]$ tail-biting block code is summarized in Figure 9. This figure confirms again, as in Figure 7, that BiPFSA($\lambda = 12$) is more stable in its average decoding complexity than all other decoding algorithms except PFSA($\lambda = 12$).

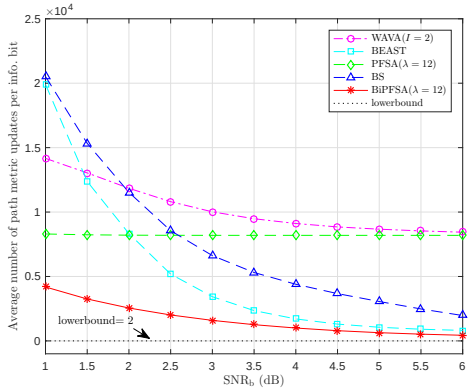

Fig. 8. Sample average numbers of path metric updates per information bit of BEAST, BiPFSA($\lambda = 12$), BS, PFSA($\lambda = 12$) and WAVA($I = 2$) for $[96, 48, 16]$ tail-biting block code

We close this section by pointing out that due to its sequential nature of decoding, BiPFSA cannot be fully parallel implemented in hardware, even though hardware-based stack structure that employs systolic arrays [20] can attain a constant complexity for each stack maintenance operation. This is in contrast to traditional Viterbi-based convolutional decoders such as WAVA, which fit well for a parallel hardware implementation. However, efficient Viterbi-based ML decoders for tail-biting convolutional code have thus far not been established. Hence, for applications that dictate an exact ML performance, or favor a software solution due to limited hardware budget, BiPFSA is a promising candidate due to its superior

TABLE II
Maximum numbers of path metric updates per information bit of the decoding algorithms in Figure 6 for $[24, 12, 8]$ extended Golay code. For ease of locating the best values, the smallest number in a column is boldfaced.

	$\mathrm{SNR_b}$					
	1 dB	2 dB	3 dB	4 dB	5 dB	6 dB
WAVA($I = 2$)	256	256	256	256	256	256
BEAST	399	391	374	318	315	213
PFSA($\lambda = 6$)	242	227	225	210	198	164
BS	536	530	526	549	541	563
BiPFSA($\lambda = 6$)	350	300	293	275	241	210

TABLE III
Maximum numbers of path metric updates per information bit of the decoding algorithms in Figure 8 for $[96, 48, 16]$ tail-biting block code. For ease of locating the best values, the smallest number in a column is boldfaced.

	$\mathrm{SNR_b}$					
	1 dB	2 dB	3 dB	4 dB	5 dB	6 dB
WAVA($I = 2$)	16,384	16,384	16,384	16,384	16,384	16,384
BEAST	206,736	142,617	120,529	69,277	19,594	7,191
PFSA($\lambda = 12$)	16,938	14,030	13,979	9,489	8,198	8,196
BS	334,289	283,164	120,915	81,190	76,643	54,155
BiPFSA($\lambda = 12$)	64,362	43,578	10,750	3,931	2,612	2,566

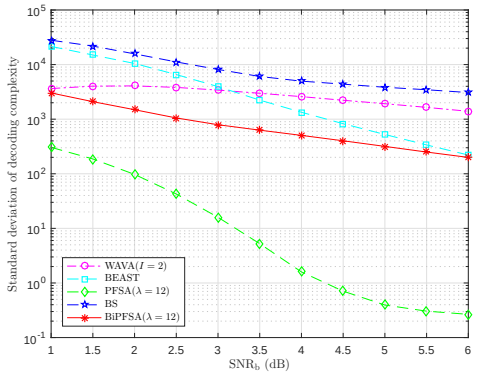

Fig. 9. Sample standard deviations of the numbers of path metric updates per information bit of the decoding algorithms in Figure 8 for $[96, 48, 16]$ tail-biting block code

computational efficiency.

V. Conclusion

In this work, we have proposed a new ML decoding algorithm for binary tail-biting convolutional codes. It is named BiPFSA because both backward PFSA and forward PFSA are employed respectively for its first and second phases. Simulations show that BiPFSA can determine the ML code path with a much lower maximum decoding complexity than previously known decoding approaches. It also outperforms existing ML decoding algorithms for tail-biting convolutional codes in terms of average decoding complexity. Unlike BEAST and BS, the decoding complexity of BiPFSA is highly stable with respect to information length or code constraint length of tail-biting codes.

Even though we only provide simulation results for binary convolutional codes with $k = 1$, the proposed algorithm is designed for any binary convolutional code with $k \geq 1$, in general. In case higher order modulations are used instead of BPSK, a certain kind of bit-wise decomposition of symbol metric such as given in [22] shall be performed before the proposed scheme can be applied. Further extension to non-binary convolutional codes, however, requires an extensive modification of the ML decoding rule in (3) and shall be an interesting work for future study.

The ML decoders mentioned in the introduction section may also be transformed to operate over the conventional trellis of an equivalent block code rather than all tail-biting subtrellises. The study of such a transformation as well as its resultant decoder for tail-biting codes could be another future work of general interest.

References

[1] Y. P. E. Wang and R. Ramesh, "To bite or not to bite - a study of tail bits versus tail-biting," in Proceeding of IEEE Personal, Indor and Mobile Radio Communications, vol. 2, pp. 317 – 321, October 1996.
[2] R. Y. Shao, S. Lin, and M. P. C. Fossorier, "Two decoding algorithms for tailbiting codes," IEEE Trans. Commun., pp. 1658–1665, October 2003.
[3] Q. Wang and V. K. Bhargava, "An efficient maximum likelihood decoding algorithm for generalized tail biting," IEEE Trans. Commun., vol. COM-37, no. 8, pp. 875 – 879, 1989.
[4] G. D. F. Jr. and S. Guha, "Simple rate-1/3 convolutional and tail-biting quantum error-correcting codes," in Proceedings. International Symposium on Information Theory, 2005, pp. 1028–1032.
[5] H. Ma and J. Wolf, "On tail biting convolutional codes," Communications, IEEE Transactions on, vol. 34, no. 2, pp. 104–111, Feb 1986.
[6] R. V. Cox and C. E. W. Sundberg, "An efficient adaptive circular viterbi algorithm for decoding generalized tailbiting convolutional codes," IEEE Trans. Veh. Techol, vol. 43, no. 11, pp. 57 – 68, February 1994.
[7] H.-T. Pai, Y. S. Han, and Y.-J. Chu, "New HARQ scheme based on decoding of tail-biting convolutional codes in IEEE 802.16e," IEEE Trans. Veh. Techol, vol. 60, no. 3, pp. 912–918, March 2011.
[8] G. Solomon and H. C. A. van Tilborg, "A connection between block and convolutional codes," SIAM J. Appl. Math., pp. 358–369, 1979.
[9] M. P. C. Fossorier and S. Lin, "Computationally efficient soft-decision decoding of linear block codes based on ordered statistics," IEEE Trans. Inform. Theory, pp. 738–751, May 1996.

[10] A. Valembois and M. Fossorier, "Box and match techniques applied to soft-decision decoding," IEEE Transactions on Information Theory, vol. 50, no. 5, pp. 796–810, May 2004.
[11] I. E. Bocharova, M. Handlery, R. Johannesson, and B. D. Kudryashov, "Beast decoding of block codes obtained via convolutional codes," IEEE Trans. Inform. Theory, vol. 51, no. 5, pp. 1880 – 1891, May 2005.
[12] P. Shankar, P. N. A. Kumar, K. Sasidharan, B. S. Rajan, and A. S. Madhu, "Efficient convergent maximum likelihood decoding on tail-biting," February 2006. [Online]. Available: http://arxiv.org/abs/cs.IT/0601023
[13] Y. S. Han, T.-Y. Wu, H.-T. Pai, P.-N. Chen, and S.-L. Shieh, "Priority-first search decoding for convolutional tail-biting codes," in Information Theory and Its Applications (ISITA 2008), 2008.
[14] H.-T. Pai, Y. S. Han, T.-Y. Wu, P.-N. Chen, and S.-L. Shieh, "Low-complexity ML decoding for convolutional tail-biting codes," IEEE Communications Letters, vol. 12, no. 12, pp. 883 – 885, December 2008.
[15] H. Qian, X.-T. Wang, K. Kang, and W.-D. Xiang, "A depth-first ML decoding algorithm for tail-biting trellises," IEEE Transactions on Vehicular Technology, vol. 64, no. 8, pp. 3339–3346, 2015.
[16] M. Handlery, R. Johannesson, and V. V. Zyablov, "Boosting the error performance of suboptimal tailbiting decoders," IEEE Trans. Commun., vol. 51, no. 9, pp. 1485 – 1491, September 2003.
[17] P. Stahl, J. B. Anderson, and R. Johannesson, "Optimal and near-optimal encoders for short and moderate-length tailbiting trellises," IEEE Trans. Inform. Theory, vol. 45, pp. 2562 – 2571, November 1999.
[18] I. E. Bocharova, B. D. Kudryashov, R. Johannesson, and P. Stahl, "Searching for tailbiting codes with large minimum distances," in IEEE Int. Symp. on Information Theory, Sorrento, Italy, 2000.
[19] J. W. J. Williams, "Algorithm 232: Heapsort," Communications of the ACM, vol. 7, no. 6, pp. 347–348, February 1964.
[20] P. Lavoie, D. Haccoun, and Y. Savaria, "A systolic architecture for fast stack sequential decoders," IEEE Trans. Commun., vol. 42, no. 5, pp. 324 – 335, May 1994.
[21] S. Bochkanov. Sparse matrices - hash table storage. [Online]. Available: http://www.alglib.net/matrixops/sparse.php
[22] G. Caire, G. Taricco, and E. Biglieri, "Bit-interleaved coded modulation," IEEE Trans. Inform. Theory, vol. 44, no. 3, pp. 927–946, May 1998.

Yunghsiang S. Han (S'90-M'93-SM'08-F'11) was born in Taipei, Taiwan, 1962. He received B.Sc. and M.Sc. degrees in electrical engineering from the National Tsing Hua University, Hsinchu, Taiwan, in 1984 and 1986, respectively, and a Ph.D. degree from the School of Computer and Information Science, Syracuse University, Syracuse, NY, in 1993. He was from 1986 to 1988 a lecturer at Ming-Hsin Engineering College, Hsinchu, Taiwan. He was a teaching assistant from 1989 to 1992, and a research associate in the School of Computer and Information Science, Syracuse University from 1992 to 1993. He was, from 1993 to 1997, an Associate Professor in the Department of Electronic Engineering at Hua Fan College of Humanities and Technology, Taipei Hsien, Taiwan. He was with the Department of Computer Science and Information Engineering at National Chi Nan University, Nantou, Taiwan from 1997 to 2004. He was promoted to Professor in 1998. He was a visiting scholar in the Department of Electrical Engineering at University of Hawaii at Manoa, HI from June to October 2001, the SUPRIA visiting research scholar in the Department of Electrical Engineering and Computer Science and CASE center at Syracuse University, NY from September 2002 to January 2004 and July 2012 to June 2013, and the visiting scholar in the Department of Electrical and Computer Engineering at University of Texas at Austin, TX from August 2008 to June 2009. He was with the Graduate Institute of Communication Engineering at National Taipei University, Taipei, Taiwan from August 2004 to July 2010. From August 2010 to January 2017, he was with the Department of Electrical Engineering at National Taiwan University of Science and Technology as Chair Professor. Now he is with School of Electrical Engineering & Intelligentization at Dongguan University of Technology, China. He is also a Chair Professor at National Taipei University from February 2015. His research interests are in error-control coding, wireless networks, and security.

Dr. Han was a winner of the 1994 Syracuse University Doctoral Prize and a Fellow of IEEE. One of his papers won the prestigious 2013 ACM CCS Test-of-Time Award in cybersecurity.

Ting-YI Wu was born in Tainan, Taiwan, in 1983. He received the B.Sc. and M.Sc. degrees in Computer Science and Information Engineering from National Chi-Nan University, Nantou, Taiwan, in 2005 and 2007, respectively, and the Ph.D. degree in Communication Engineering from National Chiao-Tung University, Hsinchu, Taiwan, in 2013. After his graduation, he worked with Network Technology Laboratory as a postdoctoral fellow in National Chiao-Tung University, Hsinchu, Taiwan, till 2015 and then he joined the Signal-Processing and Communication Laboratory in Hong Kong University of Science Technology in 2016. He now is a postdoctoral research associate in the Coordinated Science Laboratory at the University of Illinois at Urbana-Champaign. His current research interests include information-attention theory, simultaneous information/energy communication, and joint source-channel codes.

Po-Ning Chen (S'93-M'95-SM'01) was born in Taipei, R.O.C. in 1963. He received the B.S. and M.S. degrees in electrical engineering from National Tsing-Hua University, Taiwan, in 1985 and 1987, respectively, and the Ph.D. degree in electrical engineering from University of Maryland, College Park, in 1994.

From 1985 to 1987, he was with Image Processing Laboratory in National Tsing-Hua University, where he worked on the recognition of Chinese characters. During 1989, he was with Star Tech. Inc., where he focused on the development of finger-print recognition systems. After the reception of Ph.D. degree in 1994, he jointed Wan Ta Technology Inc. as a vice general manager, conducting several projects on Point-of-Sale systems. In 1995, he became a research staff in Advanced Technology Center, Computer and Communication Laboratory, Industrial Technology Research Institute in Taiwan, where he led a project on Java-based Network Managements. Since 1996, he has been an Associate Professor in Department of Communications Engineering at National Chiao-Tung University, Taiwan, and was promoted to a full professor since 2001. He was elected to be the Chair of IEEE Communications Society Taipei Chapter in 2006 and 2007, during which IEEE ComSoc Taipei Chapter won the 2007 IEEE ComSoc Chapter Achievement Awards (CAA) and 2007 IEEE ComSoc Chapter of the Year (CoY). He has served as the chairman of Department of Communications Engineering, National Chiao-Tung University, during 2007-2009. From 2012-2015, he becomes the associate chief director of Microelectronics and Information Systems Research Center, National Chiao Tung University, Taiwan, R.O.C.

Dr. Chen received the annual Research Awards from National Science Council, Taiwan, R.O.C., five years in a row since 1996. He then received the 2000 Young Scholar Paper Award from Academia Sinica, Taiwan. His Experimental Handouts for the course of Communication Networks Laboratory have been awarded as the Annual Best Teaching Materials for Communications Education by Ministry of Education, Taiwan, R.O.C., in 1998. He has been selected as the Outstanding Tutor Teacher of National Chiao-Tung University in 2002, 2013 and 2014. He was also the recipient of Distinguished Teaching Award from College of Electrical and Computer Engineering, National Chiao-Tung University, Taiwan, in 2003 and 2014. His research interests generally lie in information and coding theory, large deviation theory, distributed detection and sensor networks.

Pramod K. Varshney (S'72-M'77-SM'82-F'97) was born in Allahabad, India, on July 1, 1952. He received the B.S. degree in electrical engineering and computer science (with highest hons.), and the M.S. and Ph.D. degrees in electrical engineering from the University of Illinois at Urbana-Champaign, Champaign, IL, USA, in 1972, 1974, and 1976, respectively. Since 1976, he has been with Syracuse University, Syracuse, NY, USA, where he is currently a Distinguished Professor of electrical engineering and computer science and the Director of CASE: Center for Advanced Systems and Engineering. He served as the Associate Chair of the department from 1993 to 1996. He is also an Adjunct Professor of Radiology at Upstate Medical University, Syracuse, NY. His current research interests include distributed sensor networks and data fusion, detection and estimation theory, wireless communications, image processing, radar signal processing, and remote sensing. He has published extensively. He is the author of Distributed Detection and Data Fusion (Springer-Verlag, 1997).

Dr. Varshney was a James Scholar, a Bronze Tablet Senior, and a Fellow while at the University of Illinois. He is a member of Tau Beta Pi and received the 1981 ASEE Dow Outstanding Young Faculty Award. He was elected to the grade of Fellow of the IEEE in 1997 for his contributions in the area of distributed detection and data fusion. He was the Guest Editor of the Special Issue on Data Fusion of the Proceedings of the IEEE, January 1997. In 2000, he received the Third Millennium Medal from the IEEE and the Chancellor's Citation for exceptional academic achievement at Syracuse University. He received the IEEE 2012 Judith A. Resnik Award, Doctor of Engineering Honoris causa from Drexel University in 2014, and the ECE Distinguished Alumni Award from the University of Illinois in 2015. He is on the Editorial Boards of the Journal on Advances in Information Fusion and IEEE Signal Processing Magazine. He was the President of International Society of Information Fusion during 2001.