An Efficient Decoding Algorithm for Cycle-free Convolutional Codes and Its Applications

Jing Li, Krishna R. Narayanan and Costas N. Georghiades
Department of Electrical Engineering
Texas A&M University, College Station, TX 77843-3128
E-mail: {jingli, krishna, georghia}@ee.tamu.edu

Abstract— **This paper proposes efficient message-passing algorithms for decoding $1/(1+D^n)$ codes, whose Tanner graphs are cycle-free. A sum-product algorithm is first proposed, whose serial realization is shown to perform equivalently to the log-domain maximum *a posterior* (log-MAP) implementation of the BCJR algorithm. Next, a min-sum algorithm is proposed, which is shown to perform equivalently to the max-log-MAP algorithm. Additionally, a parallel realization of the sum-product algorithm is discussed and shown to resemble low-density parity-check (LDPC) decoding. In this, the paper presents an explicit example which confirms the claim that the message-passing algorithm is optimal on cycle-free graphs. Complexity analysis reveals that the sum-product and the min-sum algorithms consume only 1/5 and 1/8 of the computational complexity of their trellis-based counterparts, respectively. Finally, prospective applications which can take advantage of the proposed efficient algorithms are discussed along with simulations.**

I. Introduction

The discovery and rediscovery of low density parity check (LDPC) codes have aroused great interest in the study of code graphs. Tanner graphs, factor graphs, Bayesian networks and code geometries have been investigated and shown to offer an umbrella under which many different codes, including Hamming codes, convolutional codes, turbo codes, LDPC codes, and product codes, can be unified [1]-[6]. Code graphs, simple as they are, effectively capture the constraints governing the code and provide a model upon which graph-based decoding algorithms can be derived. The most notable graph-based decoding algorithm is the message-passing algorithm, which rooted back to Pearl's belief propagation algorithm [7]. Although the message-passing algorithm has found great success in LDPC codes, it is much less studied in convolutional codes. A major reason is that the code graph of a convolutional code is generally multi-connected and contains many cycles. It is therefore not obvious how probabilistic inference should be conducted, nor is there any guarantee for close approximation to optimal decoding.

This paper investigates message-passing decoding of a special class of convolutional codes, $1/(1+D^n)$, whose code graphs are cycle-free. When $n=1$, we have $1/(1+D)$, which is known as the differential encoder or the accumulator. Since the code $1/(1+D)$ performs better and is more popular in practice (e.g. as the inner code of a serially concatenated code) than the other codes in the family, we therefore focus the discussion on this code only. However, the proposed algorithms and the optimality arguments hold for all $n \geq 1$.

We consider two sub-categories of the message-passing algorithm, formally known as the sum-product algorithm and the min-sum algorithm. We first propose a (serial) sum-product algorithm and demonstrate its equivalence to the log-domain maximum *a posteriori* (log-MAP) implementation of the BCJR algorithm. For ease of hardware implementation, a parallel realization is then presented and its resemblance to LDPC decoding is discussed. This provides an alternative viewpoint on the structural properties of this special class of convolutional codes. To cater for applications with restrictive complexity, we further propose a low-complexity approximation of the sum-product algorithm, namely, the min-sum algorithm. We show that the min-sum algorithm is equivalent to the max-log-MAP algorithm. Complexity analysis reveals that the proposed sum-product and the min-sum algorithms lead to some 80% and 87.5% reduction in complexity compared to their trellis-based counterparts. Finally, applications which can take advantage of the proposed algorithms are noted, and simulation results of product accumulate (PA) codes [11] are provided.

The basic idea of this paper is implied in Tanner's pioneering work in 1981 [1], including the fundamental idea of graph representation and the generic iterative probabilistic decoding algorithm, which has since been investigated by Frey, Forney, Koetter, Luby, Richardson, Urbanke, Wiberg, Offer and Soljamin *et al* [2]-[8]. this paper explicitly connects the trellis-based BCJR algorithm, LDPC decoding and the sum-product decoding through the case study of $1/(1+D^n)$ code. While preparing the paper, we became aware of Wiberg's dissertation [6] and papers by Forney [4] and Frey [5] *et al*, where the sum-product algorithm is developed as a generalization of the trellis decoding, the LDPC decoding and a variety of other algorithms in signal processing and digital communications. These independent research papers point out the potential optimality of the sum-product algorithm on cycle-free trellis codes, but no examples were provided. The contribution of this work is to present a concrete example to confirm the above claim and, by providing efficient alternatives to the trellis-based BCJR algorithm, to save complexity for a variety of applications where $1/(1+D^n)$ code is used.

The paper is organized as follows. Section 2 discusses the (serial) sum-product, the parallel sum-product and the min-sum algorithms for decoding $1/(1+D)$, and demonstrates their equivalence to the log-MAP, the LDPC and the max-log-MAP decoding. Section 3 analyzes the complexity of the proposed algorithms, extends them to $1/(1+D^n)$ codes, and discusses potential applications. Section 4 concludes the paper.

II. Efficient Decoding Algorithms for $1/(1+D)$

A. Sum-Product Decoding Algorithm

The rate-1 convolutional code $1/(1+D)$ is typically decoded using a 2-state log-MAP decoder implementing the BCJR algorithm. Below we show how lower-complexity algorithms can be implemented without sacrificing the performance optimality by means of graph-based decoding.

As shown in Fig. 1(b), a cycle-free bipartite Tanner graph capturing the relation of $y_i = x_i \oplus y_{i-1}$ ($\oplus$ denotes modulo-2 addition) can be used to describe the code $1/(1+D)$. The bipartite graph consists of two types of nodes, *bits* and *checks*, connected by undirected edges. A check presents a constraint

on the code such that all the bits connecting to it should sum up (modulo-2) zero.

The key challenge of sum-product decoding is to specify a proper order in which messages propagate, since this order affects the ultimate decoder performance as well as the speed of decoding convergence. In this subsection, we discuss a serial update schedule where messages are passed onward from bit 1 to bit N, then backward from bit N to bit 1, and finally combined and released. Fig. 2 illustrates the message flow. The fundamental rule governing the sum-product algorithm is the *partial independence* rule, which states that no message should be routed back to its source. In other words, the outbound message along an edge should be a function of the messages obtained from all the other sources except the inbound message along this very edge. For example, in Fig. 2, bit y_i gathers messages (in log-likelihood ratio or LLR form) from three sources: $L_{ch}(y_i)$ from the channel, $L_{e_f}(y_i)$ from check i and $L_{e_b}(y_i)$ from check i+1. Since $L_{e_f}(y_i)$ comes from check i, it should be excluded when bit y_i computes (extrinsic) messages for check i; see Fig. 2(c)). Similar rules are observed for all the other checks and bits.

Clearly, the decoding of $1/(1+D)$ alone does not involve any iteration, but simply a "forward pass" and a "backward pass" before the final LLR values are computed at each source bit (Fig. 2). However, since $1/(1+D)$ by itself does not provide any error protection and since it is primarily used as an inner code in a serial concatenated code, we therefore discuss the decoding algorithm in the more general context and use a superscript (k) to indicate the k_{th} round of global iteration between the inner and outer code. The superscript (k) can simply be dropped if no global iteration is involved.

Following the illustration in Fig. 2(a), the messages associated with check i to be sent to (source) bit x_i at the k_{th} global iteration, denoted as $L_e^{(k)}(x_i)$, is computed by:

$$L_e^{(k)}(x_i) = \Big(L_{ch}(y_{i-1}) + L_{e_f}^{(k)}(y_{i-1})\Big) \boxplus \Big(L_{ch}(y_i) + L_{e_b}^{(k)}(y_i)\Big), \tag{1}$$

where $L_{ch}(y_i) = \log \frac{\Pr(r_i|y_i=0)}{\Pr(r_i|y_i=1)}$ denotes the message obtained from the transmission channel upon receiving noisy signal r_i, $L_{e_f}(y_i)$ denotes the message passed "forward" to bit y_i from the sequence of bits/checks before the i_{th} position, and $L_{e_b}(y_i)$ denotes the message passed "backward" to bit y_i from the sequence of bits/checks after the i_{th} position. Operation $\boxplus$ refers to a "check" operation and is mathematically expressed as:

$$\gamma = \alpha \boxplus \beta \iff \gamma = \log \frac{1+e^{\alpha+\beta}}{e^{\alpha}+e^{\beta}}, \tag{2}$$

$$\iff \tanh\frac{\gamma}{2} = \tanh\frac{\alpha}{2} \cdot \tanh\frac{\beta}{2}. \tag{3}$$

Following the rule of partial independence, $L_{e_f}(y_i)$ and $L_{e_b}(y_i)$, which correspond to messages passed forward and backward along the code graph, can be calculated using (see Fig. 2(b,c) for illustration):

$$L_{e_f}^{(k)}(y_i) = L_o^{(k-1)}(x_i) \boxplus \Big(L_{ch}(y_{i-1}) + L_{e_f}^{(k)}(y_{i-1})\Big), \tag{4}$$

$$L_{e_b}^{(k)}(y_i) = L_o^{(k-1)}(x_{i+1}) \boxplus \Big(L_{ch}(y_{i+1}) + L_{e_b}^{(k)}(y_{i+1})\Big), \tag{5}$$

where $L_o^{(k)}(x_i)$ denotes the *a priori* information of (source) bit x_i. Specifically, in a serially concatenated system where the $1/(1+D)$ code acts as the inner code, $L_o^{(k)}(x_i)$, $k>0$, contains the message obtained from the outer code after the k_{th} global iteration. Apparently, $L_o^{(0)}(x_i) = 0, \forall i$, since there is no message from the outer code in the first global iteration.

It is easy to see from Fig. 2 that the boundary conditions of (4) and (5) are:

$$L_{e_f}^{(k)}(y_1) = L_o^{(k-1)}(x_1) \boxplus \infty = L_o^{(k-1)}(x_1), \tag{6}$$

$$L_{e_b}^{(k)}(y_N) = 0. \tag{7}$$

Consider all the bits in the context of time series, we see that the total message computed at time i, $L_e^{(k)}(x_i)$, has utilized all the dependences of the past and the future (through $L_{e_f}(x_{i-1})$ and $L_{e_b}(x_i)$) to their fullest extend without looping-back or over-processing any information (Fig. 2(a)). This message-passing algorithm is therefore a best-effort solution. In the next subsection, we provide a rigorous optimality proof, which confirms Wiberg's claim that it is possible to find an efficient updating order (where each intermediate "cost function" is calculated only once) in a cycle-free structure like trellis (and yet get optimal performance) [6].

B. Optimality of the Sum-Product Decoding

We prove the optimality of the proposed sum-product decoding by revealing its equivalence to the BCJR algorithm of the per-symbol max *a posteriori* probabilistic decoding [12]. Due to the space limitation, we skip basic introduction to the BCJR algorithm. Interested readers are referred to [12] [13] [14]. We use x_t, y_t, s_t, r_t to represent respectively the source bit, the coded bit, the (binary) modulated bit (signals to be transmitted over the channel) and the noisy received bit at the destination. Their relations are illustrated as follows:

$$\begin{array}{ccccccc} & & y_t = y_{t-1} \oplus x_t & & BPSK & & +\text{noise} \\ x_{t\in(0,1)} & \Longrightarrow & y_{t\in(0,1)} & \Longrightarrow & s_{t\in(\pm 1)} & \Longrightarrow & r_t \end{array} \tag{8}$$

The following definitions and notations are needed in the discussion:

- $\Pr(S_t = m)$ — the probability the decoder is in state m at time t, ($m \in \{0,1\}$ in a 2-state case).
- $\mathbf{r}_i^j = (r_i, r_{i+1}, \cdots, r_j)$ — received sequence.
- $\alpha_t(m) = \Pr(S_t = m, \mathbf{r}_1^t)$ — forward path metric.
- $\beta_t(m) = \Pr(\mathbf{r}_{t+1}^N | S_t = m)$ — backward path metric.
- $\gamma_t(m',m) = \Pr(S_t = m, r_t | S_{t-1} = m')$ — branch metric.
- $\Lambda_t = \log \frac{\Pr(x_t=0|\mathbf{r}_1^N)}{\Pr(x_t=1|\mathbf{r}_1^N)}$ — output LLR of bit x_t.

The branch metric of $1/(1+D)$ code is given by (see the trellis in Fig. 1(a)):

$$\gamma_t(0,0) = \Pr(x_t=0)\Pr(r_t|y_t=0), \tag{9}$$

$$\gamma_t(0,1) = \Pr(x_t=1)\Pr(r_t|y_t=1), \tag{10}$$

$$\gamma_t(1,0) = \Pr(x_t=1)\Pr(r_t|y_t=0), \tag{11}$$

$$\gamma_t(1,1) = \Pr(x_t=0)\Pr(r_t|y_t=1). \tag{12}$$

In the log-MAP implementation of the BCJR algorithm, the forward metric is computed recursively using:

$$\frac{\alpha_t(0)}{\alpha_t(1)} = \frac{\alpha_{t-1}(0)\gamma_t(0,0) + \alpha_{t-1}(1)\gamma_t(1,0)}{\alpha_{t-1}(0)\gamma_t(0,1) + \alpha_{t-1}(1)\gamma_t(1,1)} \quad (13)$$

Substituting (9)-(12) to (13), and dividing both the nominator and the denominator by $\alpha_{t-1}(1)\Pr(x_t=1)\Pr(r_t|y_t=1)$, we get:

$$\frac{\alpha_t(0)}{\alpha_t(1)} = \frac{\left(\frac{\alpha_{t-1}(0)}{\alpha_{t-1}(1)}\frac{\Pr(x_t=0)}{\Pr(x_t=1)} + 1\right)\cdot\frac{\Pr(r_t|y_t=0)}{\Pr(r_t|y_t=1)}}{\frac{\alpha_{t-1}(0)}{\alpha_{t-1}(1)} + \frac{\Pr(x_t=0)}{\Pr(x_t=1)}} \quad (14)$$

Define:

$$\bar{\alpha}_t := \log\frac{\alpha_t(0)}{\alpha_t(1)}, \quad (15)$$

$$L_{ch}(y_t) := \log\frac{\Pr(r_t|y_t=0)}{\Pr(r_t|y_t=1)}, \quad (16)$$

$$L_o(x_t) := \log\frac{\Pr(x_t=0)}{\Pr(x_t=1)}. \quad (17)$$

Taking logarithm on both sides of (14), we get:

$$\begin{aligned}\bar{\alpha}_t &= \log\frac{e^{\bar{\alpha}_{t-1}\cdot e^{L_o(x_t)}}+1}{e^{\bar{\alpha}_{t-1}}+e^{L_o(x_t)}} + L_{ch}(y_t),\\ &= \left(\bar{\alpha}_{t-1} \boxplus L_o(x_t)\right) + L_{ch}(y_t). \end{aligned} \quad (18)$$

Likewise, in the backward recursion we have:

$$\begin{aligned}\bar{\beta}_t &:= \log\frac{\beta_t(0)}{\beta_t(1)},\\ &= \log\frac{\gamma_{t+1}(0,0)\beta_{t+1}(0)+\gamma_{t+1}(0,1)\beta_{t+1}(1)}{\gamma_{t+1}(1,0)\beta_{t+1}(0)+\gamma_{t+1}(1,1)\beta_{t+1}(1)},\\ &= \log\frac{\frac{\Pr(x_{t+1}=0)}{\Pr(x_{t+1}=1)}\cdot\frac{\Pr(r_{t+1}|y_{t+1}=0)}{\Pr(r_{t+1}|y_{t+1}=1)}\cdot\frac{\beta_{t+1}(0)}{\beta_{t+1}(1)}+1}{\frac{\Pr(x_{t+1}=0)}{\Pr(x_{t+1}=1)}+\frac{\Pr(r_{t+1}|y_{t+1}=0)}{\Pr(r_{t+1}|y_{t+1}=1)}\cdot\frac{\beta_{t+1}(0)}{\beta_{t+1}(1)}},\\ &= \log\frac{e^{L_o(x_{t+1})}\cdot e^{L_{ch}(y_{t+1})+\bar{\beta}_{t+1}}+1}{e^{L_o(x_{t+1})}+e^{L_{ch}(y_{t+1})+\bar{\beta}_{t+1}}},\\ &= L_o(x_{t+1}) \boxplus \left(L_{ch}(y_{t+1})+\bar{\beta}_{t+1}\right). \end{aligned} \quad (19)$$

Finally, we compute the output (extrinsic) information using:

$$\begin{aligned}\Lambda_t &= \log\frac{\Pr(x_t=0|\mathbf{Y}_1^N)}{\Pr(x_t=1|\mathbf{Y}_1^N)},\\ &= \log\frac{\sum_m\sum_{m'}\sum_{x_t=0}\alpha_{t-1}(m')\gamma_t(m',m)\beta_t(m)}{\sum_m\sum_{m'}\sum_{x_t=1}\alpha_{t-1}(m')\gamma_t(m',m)\beta_t(m)},\\ &= \log\frac{\begin{array}{c}\alpha_{t-1}(0)\Pr(r_t|y_t=0)\beta_t(0)\\+\alpha_{t-1}(1)\Pr(r_t|y_t=1)\beta_t(1)\end{array}}{\begin{array}{c}\alpha_{t-1}(0)\Pr(r_t|y_t=1)\beta_t(1)\\+\alpha_{t-1}(1)\Pr(r_t|y_t=0)\beta_t(0)\end{array}},\\ &= \log\frac{e^{\bar{\alpha}_{t-1}\cdot e^{L_{ch}(y_t)+\bar{\beta}_t}}+1}{e^{\bar{\alpha}_{t-1}}+e^{L_{ch}(y_t)+\bar{\beta}_t}},\\ &= \bar{\alpha}_{t-1} \boxplus \left(L_{ch}(y_t)+\bar{\beta}_t\right). \end{aligned} \quad (20)$$

For clarity, Table II summarizes the above results from the log-MAP algorithm and compares them with the sum-product algorithm described in the previous subsection. It is obvious that the two algorithms are performing exactly the same operations, where $\bar{\alpha}_t = L_{ch}(y_t) + L_{e_f}(y_t)$, $\bar{\beta}_t = L_{e_b}(y_t)$ and $\Lambda_t = L_e(x_t)$. Hence, the sum-product decoding of $1/(1+D)$ presents an efficient alternative to the conventional BCJR algorithm.

C. Parallel Sum-Product Algorithm and Its Relation to LDPC Decoding

Parallelization is highly desired in hardware implementation. When fully parallelized, the system can assign one processing unit to each bit and each check, such that all the check-to-bit updates can be processed in one time slot, and all the bit-to-check updates can be processed in the next time slot. In the sum-product algorithm discussed previously, serial procedures include the forward pass (4), where the computation of $L_{e_f}^{(l)}(y_{i+1})$ depends on $L_{e_f}^{(l)}(y_i)$, and the backward pass (5), where the computation of $L_{e_b}^{(l)}(y_{i-1})$ depends on $L_{e_b}^{(l)}(y_i)$. To parallelize the process, we adopt a batch mode update, where the forward message of the $(i+1)_{th}$ bit and the backward message of the $(i-1)_{th}$ bit are computed from the message of the i_{th} bit from the *previous* iteration, rather than that from the *current* iteration. Hence, (4) and (5) can be rewritten as:

$$L_{e_f}^{(k)}(y_i) \approx L_o^{(k-1)}(x_i) \boxplus \left(L_{ch}(y_{i-1}) + L_{e_f}^{(k-1)}(y_{i-1})\right), \quad (21)$$

$$L_{e_b}^{(k)}(y_i) \approx L_o^{(k-1)}(x_{i+1}) \boxplus \left(L_{ch}(y_{i+1}) + L_{e_b}^{(k-1)}(y_{i+1})\right). \quad (22)$$

Fig. 1(c) presents the $1/(1+D)$ code in a sparse parity check matrix form. It is instructive to note that the aforementioned parallelization essentially turns the $1/(1+D)$ decoder into an LDPC decoder, where a batch mode of bit-to-check and check-to-bit updates is performed. This parallel approach is what is used in the decoding of irregular repeat accumulate (IRA) codes [9] [10].

D. Min-Sum Algorithm and Its Relation to Max-log-MAP

Observe that the main complexity of the sum-product algorithm comes from $\boxplus$ operation. A straight-forward implementation of $\boxplus$ requires 1 addition and 3 table lookups (assuming $\log(\tanh(\frac{x}{2}))$ and its reverse operation $2\tanh^{-1}(e^x)$ are implemented via table lookup). Considerable amount of complexity can be saved by approximating $\boxplus$ with a simple min-max operation:

$$\begin{aligned}\gamma &= \alpha \boxplus \beta = \log\frac{1+e^{\alpha+\beta}}{e^\alpha+e^\beta},\\ &= sign(\alpha)\cdot sign(\beta)\cdot\min(|\alpha|,|\beta|)\\ &\quad + \log\frac{1+e^{-|\alpha+\beta|}}{1+e^{-|\alpha-\beta|}},\\ &\approx sign(\alpha)\cdot sign(\beta)\cdot\min(|\alpha|,|\beta|) \end{aligned} \quad (23)$$

The above approximation reduces the sum-product algorithm to the min-sum algorithm. It is interesting to note that, just as the sum-product algorithm is equivalent to the log-MAP algorithm, the min-sum algorithm is equivalent to the max-log-MAP algorithm. To see this, recall that the max-log-MAP algorithm approximates the log-MAP algorithm by using a max operation instead of the max^* operation [14]: $\max^*(x,y) = \log(e^\alpha + e^\beta) \approx \max(\alpha,\beta)$. It immediately follows that:

$$\begin{aligned}\gamma &= \alpha \boxplus \beta,\\ &= \log(e^0+e^{\alpha+\beta}) - \log(e^\alpha+e^\beta),\\ &\approx max(0,\alpha+\beta) - max(\alpha,\beta),\\ &= sign(\alpha)\cdot sign(\beta)\cdot min(\alpha,\beta). \end{aligned} \quad (24)$$

III. Complexity and Applications

A. Complexity

Table I compares the complexity of the log-MAP, max-log-MAP [14], sum-product and min-sum algorithms for decoding $1/(1+D)$. We see that the sum-product and the min-sum algorithms require only about $1/5$ and $1/8$ the complexity of their trellis-based equivalents respectively. This is because in the sum-product decoding, a single table lookup $\log(\tanh\frac{x}{2})$ is used instead of several max^* operations which greatly reduces the computational complexity.

Extending the aforementioned algorithms of $1/(1+D)$ to $1/(1+D^n)$ is straightforward, since the latter has essentially the same code graphs as the former. Fig. 3 shows the code graph of a $1/(1+D^2)$ code. From the graph perspective, a $1/(1+D^n)$ code is like a n-multiplexed $1/(1+D)$ code.

TABLE I
Complexity of log-MAP, Max-log-MAP, Sum-Product and Min-Sum Decoding for $1/(1+D)$ Code (operations per bit)

Oper	log-map	max-log-map	sum-pro	min-sum
add	39	31	5	2
min/max	8	8		3
lookup	8		6	

B. Applications and Simulations

The class of $1/(1+D^n)$ codes and particularly the $1/(1+D)$ code is becoming a popular inner code in a serially concatenated code where interleaving gain can be achieved without reduction in overall code rate. The growing class of "accumulated codes" include, for example, convolutional accumulated codes (serial turbo codes), regular/irregular repeat accumulate codes [9] [10] and product accumulated (PA) codes [11]. All of these codes can benefit from the he proposed algorithms.

To evaluate the performance of the proposed algorithms, consider the example of product accumulate codes [11]. Product accumulate codes are a class of interleaved serial concatenated codes where the inner code is $1/(1+D)$, and the outer code is a parallel concatenation of 2 single parity check codes [11]. Like turbo and repeat accumulate codes, product accumulate codes have shown performance impressively close to the capacity.

We examine the performance of both the serial and the parallel sum-product decoding. In general, serial update leads to a faster convergence and possibly a better performance than parallel update. Fig. 4 demonstrates the performance of an (8K,4K) product accumulate code. That parallelization incurs only about 0.1 dB loss presents the parallel sum-product algorithm as a strong candidate for hardware implementation.

Fig. 5 compares the performance of a (2K,1K) PA code using the sum-product and the min-sum algorithms. Bit error rates after $5, 10, 15, 20$ iterations are evaluated. At all those iterations, the min-sum algorithm incurs about 0.2 dB loss. This points to the min-sum algorithm as an appealing solution for simple, low-cost systems.

IV. Conclusion

An efficient and optimal sum-product algorithm is proposed for decoding $1/(1+D^n)$ codes. Its parallel realization and low-complexity approximation (i.e. the min-sum algorithm) are discussed. Prospective applications which can take advantage of the proposed algorithms are also investigated. The proposed algorithms are expected to be useful both in theory and in practice.

References

[1] R. M. Tanner, "A recursive approach to low complexity codes," *IEEE Trans. Inform. Theory*, vol. IT-27, pp.533-547, Sept. 1981

[2] B. J. Frey, *Graphical models for machine learning and digital communication*, The MIT Press, Cambridge, Massachusetts, London, England, 1998

[3] E. Offer, and E. Soljamin, "LDPC codes: a group algebra formulation", presented at *Intl. Workshop on Coding and Cryptography*, Paris, Jan., 2001

[4] G. D. Forney, Jr, "Codes on graphs: normal realizations," *Trans. Inform. Theory*, vol. 47, pp. 520-548, Feb. 2001

[5] F. R. Kschischang, B. J. Frey, and H.-A. Loeliger, "Factor graphs and the sum-product algorithm," *Trans. Inform. Theory*, vol. 47, pp. 498-519, Feb. 2001

[6] N. Wiberg, *Codes and decoding on general graphs*, Doctoral dissertation, 1996

[7] R. J McEliece, D. J. C. MacKay, and J.-F. Cheng, "Turbo decoding as an instance of Pearl's "Belief Propagation" algorithm," *IEEE Jour. Sele. Areas Commun.*, Vol. 16, pp.140-152, Feb. 1998

[8] M. G. Luby, M. Mitzenmacher, M. A. Shokrollahi, and D. A. Spielman, "Analysis of low density codes and improved designs using irregular graphs," *ACM Symposium*, pp. 249-258, 1998

[9] D. Divsalar, H. Jin and R. J. McEliece, "Coding theorems for 'turbo-like' codes", *Proc. 1998 Allerton Conf. Commun. and Control*, pp. 201-210, Sept. 1998

[10] H. Jin, A. Khandekar and R. McEliece, "Irregular repeat-accumulate codes," *Proc. 2nd Intl. Symp. on Turbo Codes and Related Topics*, Brest, France, Sept. 2000

[11] J. Li, K. R. Narayanan and C. N. Georghiades, "A class of linear-complexity, soft-decodable, high-rate, "good" codes: construction, properties and performance", to be presented *Proc. Intl. Symp. Inform. Theory*, Washington D.C., June, 2001

[12] L. Bahl, J. Cocke, F. Jelinek, and J. Raviv, "Optimal decoding of linear codes for minimizing symbol error rate," *IEEE Trans. Inform. Theory*, pp. 284-287, Mar., 1974

[13] W. E. Ryan, "A turbo code tutorial," `http://www.ece.arizona.edu/~ryan/`

[14] P. Robertson, E. Villebrum, and P. Hoeher, "A comparison of optimal and sub-optimal MAP decoding algorithms operating in the log domain," *Proc. Intl. Conf. Commun.*, Seattle, 1995

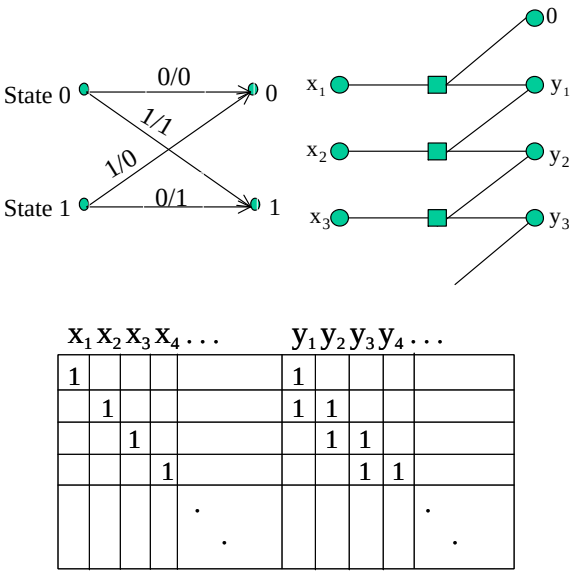

Fig. 1. Structure of $1/(1+D)$ code . (a) Trellis; (b) Tanner graph; (c) Sparse parity check matrix form.

TABLE II
SUMMARY OF SUM-PRODUCT AND MAP DECODING

	BCJR	Sum-product
forward	$\bar{\alpha}_t = (\bar{\alpha}_{t-1} \boxplus L_o(x_t)) + L_{ch}(y_t)$	$L_{e_f}(y_t) = (L_{e_f}(y_{t-1}) + L_{ch}(y_{t-1})) \boxplus L_o(x_t)$
backward	$\beta_t = (\beta_{t+1} + L_{ch}(y_{t+1})) \boxplus L_o(x_{t+1})$	$L_{e_b}(y_t) = (L_{e_b}(y_{t+1}) + L_{ch}(y_{t+1})) \boxplus L_o(x_{t+1})$
extrinsic LLR	$\Lambda_t = \bar{\alpha}_{t-1} \boxplus (\beta_t + L_{ch}(y_t))$	$L_e(x_t) = (L_{e_f}(y_{t-1}) + L_{ch}(y_{t-1})) \boxplus (L_{e_b}(y_t) + L_{ch}(y_t))$

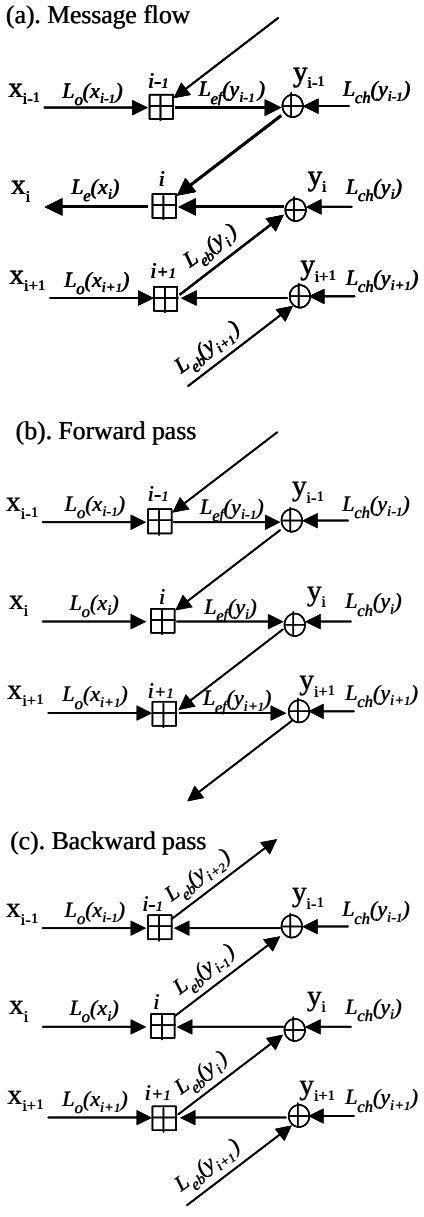

Fig. 2. Sum-product decoding of $1/(1+D)$ code

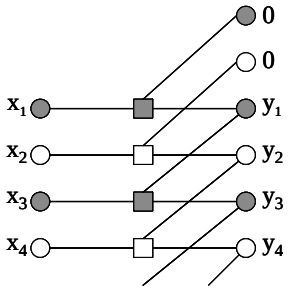

Fig. 3. Tanner graph of $1/(1+D^2)$ code

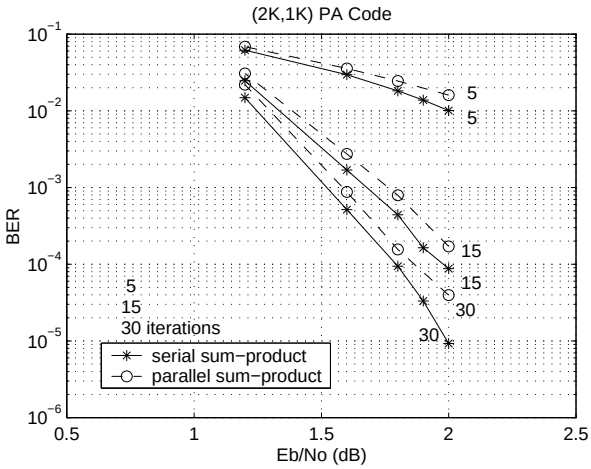

Fig. 4. Sum-product decoding: serial vs parallel

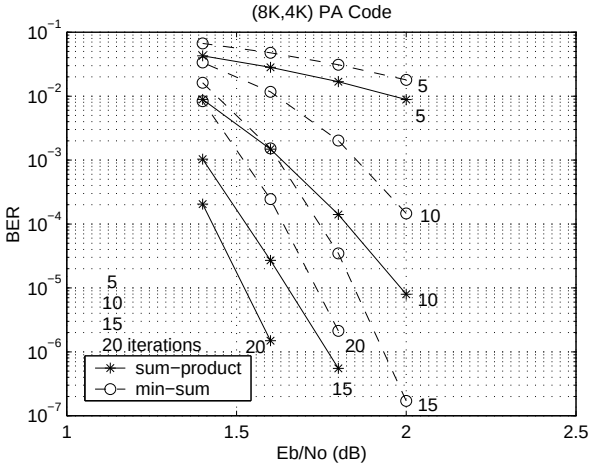

Fig. 5. Sum-product decoding vs min-sum decoding