Factor Graphs and the Sum-Product Algorithm

Frank R. Kschischang, *Senior Member, IEEE*, Brendan J. Frey, *Member, IEEE*, and Hans-Andrea Loeliger, *Member, IEEE*

***Abstract*—Algorithms that must deal with complicated global functions of many variables often exploit the manner in which the given functions factor as a product of "local" functions, each of which depends on a subset of the variables. Such a factorization can be visualized with a bipartite graph that we call a *factor graph*. In this tutorial paper, we present a generic message-passing algorithm, the sum-product algorithm, that operates in a factor graph. Following a single, simple computational rule, the sum-product algorithm computes—either exactly or approximately—various marginal functions derived from the global function. A wide variety of algorithms developed in artificial intelligence, signal processing, and digital communications can be derived as specific instances of the sum-product algorithm, including the forward/backward algorithm, the Viterbi algorithm, the iterative "turbo" decoding algorithm, Pearl's belief propagation algorithm for Bayesian networks, the Kalman filter, and certain fast Fourier transform (FFT) algorithms.**

***Index Terms*—Belief propagation, factor graphs, fast Fourier transform, forward/backward algorithm, graphical models, iterative decoding, Kalman filtering, marginalization, sum-product algorithm, Tanner graphs, Viterbi algorithm.**

I. Introduction

THIS paper provides a tutorial introduction to factor graphs and the sum-product algorithm, a simple way to understand a large number of seemingly different algorithms that have been developed in computer science and engineering. We consider algorithms that deal with complicated "global" functions of many variables and that derive their computational efficiency by exploiting the way in which the global function factors into a product of simpler "local" functions, each of which depends on a subset of the variables. Such a factorization can be visualized using a *factor graph*, a bipartite graph that expresses which variables are arguments of which local functions.

Manuscript received August 3, 1998; revised October 17, 2000. The work of F. R. Kschischang was supported in part, while on leave at the Massachusetts Institute of Technology, by the Office of Naval Research under Grant N00014-96-1-0930, and by the Army Research Laboratory under Cooperative Agreement DAAL01-96-2-0002. The work of B. J. Frey was supported, while a Beckman Fellow at the Beckman Institute of Advanced Science and Technology, University of Illinois at Urbana-Champaign, by a grant from the Arnold and Mabel Beckman Foundation. The material in this paper was presented in part at the 35th Annual Allerton Conference on Communication, Control, and Computing, Monticello, IL, September 1997.

F. R. Kschischang is with the Department of Electrical and Computer Engineering, University of Toronto, Toronto, ON M5S 3G4, Canada (e-mail: frank@comm.utoronto.ca).

B. J. Frey is with the Faculty of Computer Science, University of Waterloo, Waterloo, ON N2L 3G1, Canada, and the Faculty of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, Urbana, IL 61801-2307 USA (e-mail: frey@.uwaterloo.ca).

H.-A. Loeliger is with the Signal Processing Lab (ISI), ETH Zentrum, CH-8092 Zürich, Switzerland (e-mail: loeliger@isi.ee.ethz.ch).

Communicated by T. E. Fuja, Associate Editor At Large.

Publisher Item Identifier S 0018-9448(01)00721-0.

The aim of this tutorial paper is to introduce factor graphs and to describe a generic message-passing algorithm, called the *sum-product algorithm*, which operates in a factor graph and attempts to compute various marginal functions associated with the global function. The basic ideas are very simple; yet, as we will show, a surprisingly wide variety of algorithms developed in the artificial intelligence, signal processing, and digital communications communities may be derived as specific instances of the sum-product algorithm, operating in an appropriately chosen factor graph.

Genealogically, factor graphs are a straightforward generalization of the "Tanner graphs" of Wiberg *et al.* [31], [32]. Tanner [29] introduced bipartite graphs to describe families of codes which are generalizations of the low-density parity-check (LDPC) codes of Gallager [11], and also described the sum-product algorithm in this setting. In Tanner's original formulation, all variables are codeword symbols and hence "visible"; Wiberg *et al.*, introduced "hidden" (latent) state variables and also suggested applications beyond coding. Factor graphs take these graph-theoretic models one step further, by applying them to functions. From the factor-graph perspective (as we will describe in Section III-A), a Tanner graph for a code represents a particular factorization of the characteristic (indicator) function of the code.

While it may seem intuitively reasonable that some algorithms should exploit the manner in which a global function factors into a product of local functions, the fundamental insight that many well-known algorithms essentially solve the "MPF" (marginalize product-of-functions) problem, each in their own particular setting, was first made explicit in the work of Aji and McEliece [1]. In a landmark paper [2], Aji and McEliece develop a "generalized distributive law" (GDL) that in some cases solves the MPF problem using a "junction tree" representation of the global function. Factor graphs may be viewed as an alternative approach with closer ties to Tanner graphs and previously developed graphical representations for codes. Essentially, every result developed in the junction tree/GDL setting may be translated into an equivalent result in the factor graph/sum-product algorithm setting, and *vice versa*. We prefer the latter setting not only because it is better connected with previous approaches, but also because we feel that factor graphs are in some ways easier to describe, giving them a modest pedagogical advantage. Moreover, the sum-product algorithm can often be applied successfully in situations where exact solutions to the MPF problem (as provided by junction trees) become computationally intractable, the most prominent example being the iterative decoding of turbo codes and LDPC codes. In Section VI we do, however, discuss strategies for achieving exact solutions to the MPF problem in factor graphs.

There are also close connections between factor graphs and graphical representations (graphical models) for multidimensional probability distributions such as Markov random fields [16], [18], [26] and Bayesian (belief) networks [25], [17]. Like factor graphs, these graphical models encode in their structure a particular factorization of the joint probability mass function of several random variables. Pearl's powerful "belief propagation" algorithm [25], which operates by "message-passing" in a Bayesian network, translates immediately into an instance of the sum-product algorithm operating in a factor graph that expresses the same factorization. Bayesian networks and belief propagation have been used previously to explain the iterative decoding of turbo codes and LDPC codes [9], [10], [19], [21], [22], [24], the most powerful practically decodable codes known. Note, however, that Wiberg [31] had earlier described these decoding algorithms as instances of the sum-product algorithm; see also [7].

We begin the paper in Section II with a small worked example that illustrates the operation of the sum-product algorithm in a simple factor graph. We will see that when a factor graph is cycle-free, then the structure of the factor graph not only encodes the way in which a given function factors, but also encodes *expressions* for computing the various marginal functions associated with the given function. These expressions lead directly to the sum-product algorithm.

In Section III, we show how factor graphs may be used as a system and signal-modeling tool. We see that factor graphs are compatible both with "behavioral" and "probabilistic" modeling styles. Connections between factor graphs and other graphical models are described briefly in Appendix B, where we recover Pearl's belief propagation algorithm as an instance of the sum-product algorithm.

In Section IV, we apply the sum-product algorithm to trellis-structured (hidden Markov) models, and obtain the forward/backward algorithm, the Viterbi algorithm, and the Kalman filter as instances of the sum-product algorithm. In Section V, we consider factor graphs with cycles, and obtain the iterative algorithms used to decode turbo-like codes as instances of the sum-product algorithm.

In Section VI, we describe several generic transformations by which a factor graph with cycles may sometimes be converted—often at great expense in complexity—to an equivalent cycle-free form. We apply these ideas to the factor graph representing the discrete Fourier transform (DFT) kernel, and derive a fast Fourier transform (FFT) algorithm as an instance of the sum-product algorithm.

Some concluding remarks are given in Section VII.

II. Marginal Functions, Factor Graphs, and the Sum-Product Algorithm

Throughout this paper we deal with functions of many variables. Let $x_1, x_2, \ldots, x_n$ be a collection of variables, in which, for each i, x_i takes on values in some (usually finite) *domain* (or alphabet) A_i. Let $g(x_1, \ldots, x_n)$ be an R-valued function of these variables, i.e., a function with domain

$$S = A_1 \times A_2 \times \cdots \times A_n$$

and codomain R. The domain S of g is called the *configuration space* for the given collection of variables, and each element of S is a particular *configuration* of the variables, i.e., an assignment of a value to each variable. The codomain R of g may in general be any semiring [2], [31, Sec. 3.6]; however, at least initially, we will lose nothing essential by assuming that R is the set of real numbers.

Assuming that summation in R is well defined, then associated with every function $g(x_1, \ldots, x_n)$ are n *marginal* functions $g_i(x_i)$. For each $a \in A_i$, the value of $g_i(a)$ is obtained by summing the value of $g(x_1, \ldots, x_n)$ over all configurations of the variables that have $x_i = a$.

This type of sum is so central to this paper that we introduce a nonstandard notation to handle it: the "not-sum" or *summary*. Instead of indicating the variables being summed over, we indicate those variables *not* being summed over. For example, if h is a function of three variables x_1, x_2, and x_3, then the "summary for x_2" is denoted by

$$\sum_{\sim\{x_2\}} h(x_1, x_2, x_3) := \sum_{x_1 \in A_1} \sum_{x_3 \in A_3} h(x_1, x_2, x_3).$$

In this notation we have

$$g_i(x_i) := \sum_{\sim\{x_i\}} g(x_1, \ldots, x_n)$$

i.e., the ith marginal function associated with $g(x_1, \ldots, x_n)$ is the summary for x_i of g.

We are interested in developing efficient procedures for computing marginal functions that a) exploit the way in which the global function factors, using the distributive law to simplify the summations, and b) reuses intermediate values (partial sums). As we will see, such procedures can be expressed very naturally by use of a factor graph.

Suppose that $g(x_1, \ldots, x_n)$ factors into a product of several *local functions*, each having some subset of $\{x_1, \ldots, x_n\}$ as arguments; i.e., suppose that

$$g(x_1, \ldots, x_n) = \prod_{j \in J} f_j(X_j) \tag{1}$$

where J is a discrete index set, X_j is a subset of $\{x_1, \ldots, x_n\}$, and $f_j(X_j)$ is a function having the elements of X_j as arguments.

Definition: A *factor graph* is a bipartite graph that expresses the structure of the factorization (1). A factor graph has a *variable node* for each variable x_i, a *factor node* for each local function f_j, and an edge-connecting variable node x_i to factor node f_j if and only if x_i is an argument of f_j.

A factor graph is thus a standard bipartite graphical representation of a mathematical relation—in this case, the "is an argument of" relation between variables and local functions.

▶ *Example 1 (A Simple Factor Graph):* Let $g(x_1, x_2, x_3, x_4, x_5)$ be a function of five variables, and suppose that g can be expressed as a product

$$g(x_1, x_2, x_3, x_4, x_5) = f_A(x_1) f_B(x_2) f_C(x_1, x_2, x_3) f_D(x_3, x_4) f_E(x_3, x_5) \tag{2}$$

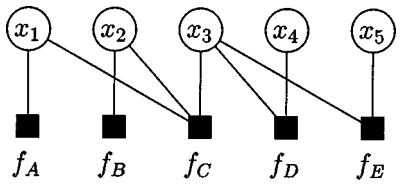

Fig. 1. A factor graph for the product $f_A(x_1)f_B(x_2)f_C(x_1, x_2, x_3) \cdot f_D(x_3, x_4)f_E(x_3, x_5)$.

of five factors, so that $J = \{A, B, C, D, E\}$, $X_A = \{x_1\}$, $X_B = \{x_2\}$, $X_C = \{x_1, x_2, x_3\}$, $X_D = \{x_3, x_4\}$, and $X_E = \{x_3, x_5\}$. The factor graph that corresponds to (2) is shown in Fig. 1. ◀

A. Expression Trees

In many situations (for example, when $g(x_1, \ldots, x_5)$ represents a joint probability mass function), we are interested in computing the marginal functions $g_i(x_i)$. We can obtain an expression for each marginal function by using (2) and exploiting the distributive law.

To illustrate, we write $g_1(x_1)$ from Example 1 as

$$g_1(x_1) = f_A(x_1)\left(\sum_{x_2} f_B(x_2)\left(\sum_{x_3} f_C(x_1, x_2, x_3) \cdot \left(\sum_{x_4} f_D(x_3, x_4)\right)\left(\sum_{x_5} f_E(x_3, x_5)\right)\right)\right)$$

or, in summary notation

$$g_1(x_1) = f_A(x_1) \times \sum_{\sim\{x_1\}} \left(f_B(x_2)f_C(x_1, x_2, x_3) \times \left(\sum_{\sim\{x_3\}} f_D(x_3, x_4)\right) \times \left(\sum_{\sim\{x_3\}} f_E(x_3, x_5)\right)\right). \quad (3)$$

Similarly, we find that

$$g_3(x_3) = \left(\sum_{\sim\{x_3\}} f_A(x_1)f_B(x_2)f_C(x_1, x_2, x_3)\right) \times \left(\sum_{\sim\{x_3\}} f_D(x_3, x_4)\right) \times \left(\sum_{\sim\{x_3\}} f_E(x_3, x_5)\right). \quad (4)$$

In computer science, arithmetic expressions like the right-hand sides of (3) and (4) are often represented by ordered rooted trees [28, Sec. 8.3], here called *expression trees*, in which internal vertices (i.e., vertices with descendants) represent arithmetic operators (e.g., addition, multiplication, negation, etc.) and leaf vertices (i.e., vertices without descendants) represent variables or constants. For example, the tree of Fig. 2 represents the expression $x(y + z)$. When the operators in an expression tree are restricted to those that are completely symmetric in their operands (e.g., multiplication and addition), it is unnecessary to order the vertices to avoid ambiguity in interpreting the expression represented by the tree.

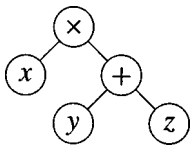

Fig. 2. An expression tree representing $x(y + z)$.

In this paper, we extend expression trees so that the leaf vertices represent *functions*, not just variables or constants. Sums and products in such expression trees combine their operands in the usual (pointwise) manner in which functions are added and multiplied. For example, Fig. 3(a) unambiguously represents the expression on the right-hand side of (3), and Fig. 4(a) unambiguously represents the expression on the right-hand side of (4). The operators shown in these figures are the function product and the summary, having various local functions as their arguments.

Also shown in Figs. 3(b) and 4(b), are redrawings of the factor graph of Fig. 1 as a rooted tree with x_1 and x_3 as root vertex, respectively. This is possible because the global function defined in (2) was deliberately chosen so that the corresponding factor graph is a tree. Comparing the factor graphs with the corresponding trees representing the expression for the marginal function, it is easy to note their correspondence. This observation is simple, but key: *when a factor graph is cycle-free, the factor graph not only encodes in its structure the factorization of the global function, but also encodes arithmetic expressions by which the marginal functions associated with the global function may be computed*.

Formally, as we show in Appendix A, to convert a cycle-free factor graph representing a function $g(x_1, \ldots, x_n)$ to the corresponding expression tree for $g_i(x_i)$, draw the factor graph as a rooted tree with x_i as root. Every node v in the factor graph then has a clearly defined parent node, namely, the neighboring node through which the unique path from v to x_i must pass. Replace each variable node in the factor graph with a product operator. Replace each factor node in the factor graph with a "form product and multiply by f" operator, and between a factor node f and its parent x, insert a $\sum_{\sim\{x\}}$ summary operator. These local transformations are illustrated in Fig. 5(a) for a variable node, and in Fig. 5(b) for a factor node f with parent x. Trivial products (those with one or no operand) act as identity operators, or may be omitted if they are leaf nodes in the expression tree. A summary operator $\sum_{\sim\{x\}}$ applied to a function with a single argument x is also a trivial operation, and may be omitted. Applying this transformation to the tree of Fig. 3(b) yields the expression tree of Fig. 3(a), and similarly for Fig. 4. Trivial operations are indicated with dashed lines in these figures.

B. Computing a Single Marginal Function

Every expression tree represents an *algorithm* for computing the corresponding expression. One might describe the algorithm as a recursive "top-down" procedure that starts at the root vertex and evaluates each subtree descending from the root, combining the results as dictated by the operator at the root. Equivalently, we prefer to describe the algorithm as a "bottom-up" procedure that begins at the leaves of the tree, with each operator vertex

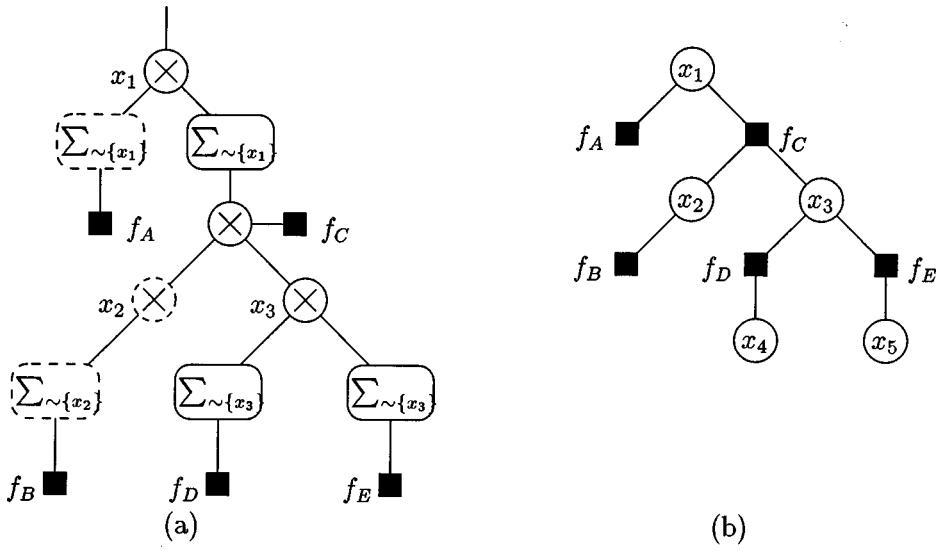

Fig. 3. (a) A tree representation for the right-hand side of (3). (b) The factor graph of Fig. 1, redrawn as a rooted tree with x_1 as root.

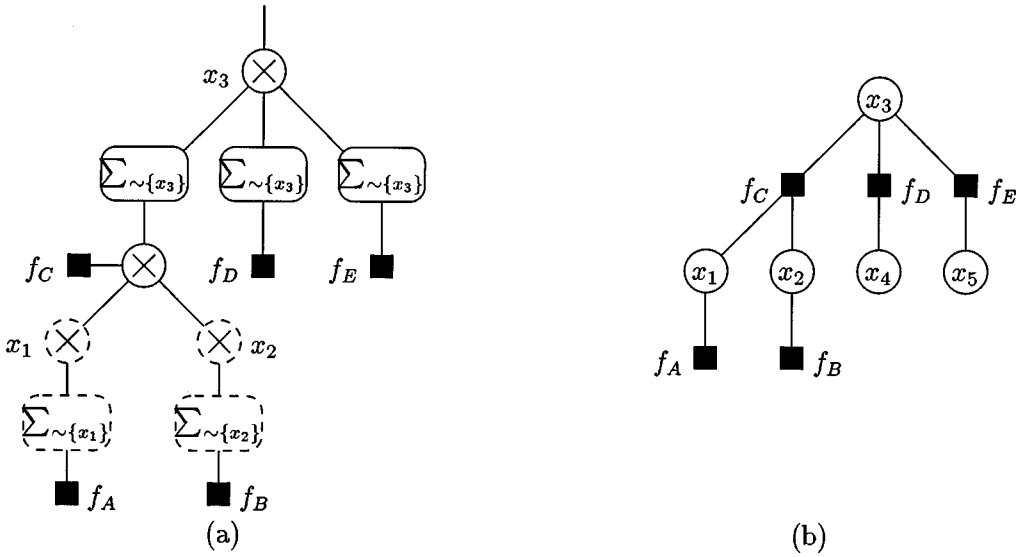

Fig. 4. (a) A tree representation for the right-hand side of (4). (b) The factor graph of Fig. 1, redrawn as a rooted tree with x_3 as root.

combining its operands and passing on the result as an operand for its parent. For example, $x(y+z)$, represented by the expression tree of Fig. 2, might be evaluated by starting at the leaf nodes y and z, evaluating $y+z$, and passing on the result as an operand for the $\times$ operator, which multiplies the result with x.

Rather than working with the expression tree, it is simpler and more direct to describe such marginalization algorithms in terms of the corresponding factor graph. To best understand such algorithms, it helps to imagine that there is a processor associated with each vertex of the factor graph, and that the factor-graph edges represent channels by which these processors may communicate. For us, "messages" sent between processors are always simply some appropriate description of some marginal function. (We describe some useful representations in Section V-E.)

We now describe a message-passing algorithm that we will temporarily call the "single-i sum-product algorithm," since it computes, for a single value of i, the marginal function $g_i(x_i)$ in a rooted cycle-free factor graph, with x_i taken as root vertex.

The computation begins at the leaves of the factor graph. Each leaf variable node sends a trivial "identity function" message to its parent, and each leaf factor node f sends a description of f to its parent. Each vertex waits for messages from all of its children before computing the message to be sent to its parent. This computation is performed according to the transformation shown in Fig. 5; i.e., a variable node simply sends the *product* of messages received from its children, while a factor node f with parent x forms the product of f with the messages received from its children, and then operates on the result with a $\sum_{\sim\{x\}}$ summary operator. By a "product of messages" we mean an appropriate description of the (pointwise) product of the corresponding functions. If the messages are parametrizations of the functions, then the resulting message is the parametrization of the product function, not (necessarily) literally the numerical

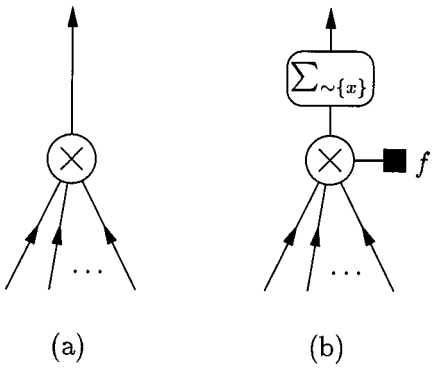

Fig. 5. Local substitutions that transform a rooted cycle-free factor graph to an expression tree for a marginal function at (a) a variable node and (b) a factor node.

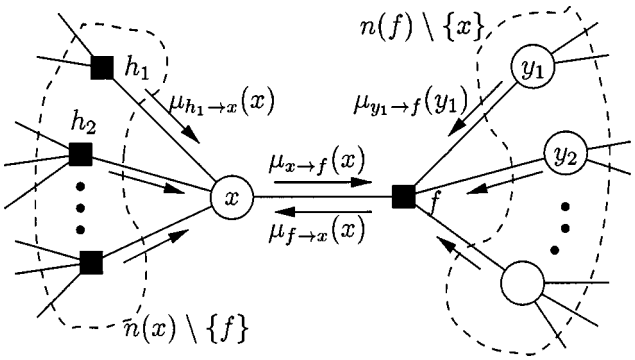

Fig. 6. A factor-graph fragment, showing the update rules of the sum-product algorithm.

product of the messages. Similarly, the summary operator is applied to the functions, not necessarily literally to the messages themselves.

The computation terminates at the root node x_i, where the marginal function $g_i(x_i)$ is obtained as the product of all messages received at x_i.

It is important to note that a message passed on the edge $\{x, f\}$, either from variable x to factor f, or *vice versa*, is a single-argument function of x, the variable *associated with* the given edge. This follows since, at every factor node, summary operations are always performed for the variable associated with the edge on which the message is passed. Likewise, at a variable node, all messages are functions of that variable, and so is any product of these messages.

The message passed on an edge during the operation of the single-i sum-product algorithm can be interpreted as follows. If $e = \{x, f\}$ is an edge in the tree, where x is a variable node and f is a factor node, then the analysis of Appendix A shows that the message passed on e during the operation of the sum-product algorithm is simply a summary for x of the *product of the local functions* descending from the vertex that originates the message.

C. Computing All Marginal Functions

In many circumstances, we may be interested in computing $g_i(x_i)$ for more than one value of i. Such a computation might be accomplished by applying the single-i algorithm separately for each desired value of i, but this approach is unlikely to be efficient, since many of the subcomputations performed for different values of i will be the same. Computation of $g_i(x_i)$ for all i simultaneously can be efficiently accomplished by essentially "overlaying" on a single factor graph all possible instances of the single-i algorithm. No particular vertex is taken as a root vertex, so there is no fixed parent/child relationship among neighboring vertices. Instead, *each* neighbor w of any given vertex v is at some point regarded as a parent of v. The message passed from v to w is computed just as in the single-i algorithm, i.e., as if w were indeed the parent of v and all other neighbors of v were children.

As in the single-i algorithm, message passing is initiated at the leaves. Each vertex v remains idle until messages have arrived on all but one of the edges incident on v. Just as in the single-i algorithm, once these messages have arrived, v is able to compute a message to be sent on the one remaining edge to its neighbor (temporarily regarded as the parent), just as in the single-i algorithm, i.e., according to Fig. 5. Let us denote this temporary parent as vertex w. After sending a message to w, vertex v returns to the idle state, waiting for a "return message" to arrive from w. Once this message has arrived, the vertex is able to compute and send messages to each of its neighbors (other than w), each being regarded, in turn, as a parent. The algorithm terminates once two messages have been passed over every edge, one in each direction. At variable node x_i, the product of all incoming messages is the marginal function $g_i(x_i)$, just as in the single-i algorithm. Since this algorithm operates by computing various sums and products, we refer to it as the *sum-product* algorithm.

The sum-product algorithm operates according to the following simple rule:

The Sum-Product Update Rule:
The message sent from a node v on an edge e is the product of the local function at v (or the unit function if v is a variable node) with all messages received at v on edges $other$ than e, summarized for the variable associated with e.

Let $\mu_{x \to f}(x)$ denote the message sent from node x to node f in the operation of the sum-product algorithm, let $\mu_{f \to x}(x)$ denote the message sent from node f to node x. Also, let $n(v)$ denote the set of neighbors of a given node v in a factor graph. Then, as illustrated in Fig. 6, the message computations performed by the sum-product algorithm may be expressed as follows:

variable to local function:

$$\mu_{x \to f}(x) = \prod_{h \in n(x) \setminus \{f\}} \mu_{h \to x}(x) \tag{5}$$

local function to variable:

$$\mu_{f \to x}(x) =, \sum_{\sim\{x\}} \left(f(X) \prod_{y \in n(f) \setminus \{x\}} \mu_{y \to f}(y) \right) \tag{6}$$

where $X = n(f)$ is the set of arguments of the function f.

The update rule at a variable node x takes on the particularly simple form given by (5) because there is no local function to include, and the summary for x of a product of functions of x is

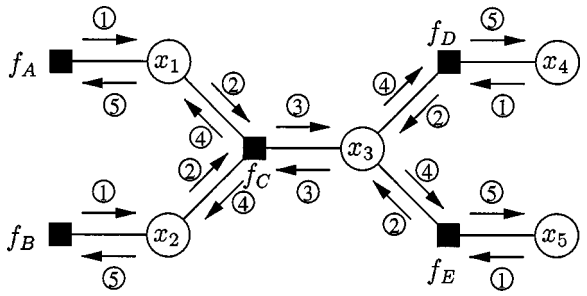

Fig. 7. Messages generated in each (circled) step of the sum-product algorithm.

simply that product. On the other hand, the update rule at a local function node given by (6) in general involves nontrivial function multiplications, followed by an application of the summary operator.

We also observe that variable nodes of degree two perform no computation: a message arriving on one (incoming) edge is simply transferred to the other (outgoing) edge.

D. A Detailed Example

Fig. 7 shows the flow of messages that would be generated by the sum-product algorithm applied to the factor graph of Fig. 1. The messages may be generated in five steps, as indicated with circles in Fig. 7. In detail, the messages are generated as follows.

Step 1:

$$\mu_{f_A\to x_1}(x_1) = \sum_{\sim\{x_1\}} f_A(x_1) = f_A(x_1)$$
$$\mu_{f_B\to x_2}(x_2) = \sum_{\sim\{x_2\}} f_B(x_2) = f_B(x_2)$$
$$\mu_{x_4\to f_D}(x_4) = 1$$
$$\mu_{x_5\to f_E}(x_5) = 1.$$

Step 2:

$$\mu_{x_1\to f_C}(x_1) = \mu_{f_A\to x_1}(x_1)$$
$$\mu_{x_2\to f_C}(x_2) = \mu_{f_B\to x_2}(x_2)$$
$$\mu_{f_D\to x_3}(x_3) = \sum_{\sim\{x_3\}} \mu_{x_4\to f_D}(x_4) f_D(x_3, x_4)$$
$$\mu_{f_E\to x_3}(x_3) = \sum_{\sim\{x_3\}} \mu_{x_5\to f_E}(x_5) f_E(x_3, x_5).$$

Step 3:

$$\mu_{f_C\to x_3}(x_3) = \sum_{\sim\{x_3\}} \mu_{x_1\to f_C}(x_1)\mu_{x_2\to f_C}(x_2) f_C(x_1, x_2, x_3)$$
$$\mu_{x_3\to f_C}(x_3) = \mu_{f_D\to x_3}(x_3)\mu_{f_E\to x_3}(x_3).$$

Step 4:

$$\mu_{f_C\to x_1}(x_1) = \sum_{\sim\{x_1\}} \mu_{x_3\to f_C}(x_3)\mu_{x_2\to f_C}(x_2) f_C(x_1, x_2, x_3)$$
$$\mu_{f_C\to x_2}(x_2) = \sum_{\sim\{x_2\}} \mu_{x_3\to f_C}(x_3)\mu_{x_1\to f_C}(x_1) f_C(x_1, x_2, x_3)$$
$$\mu_{x_3\to f_D}(x_3) = \mu_{f_C\to x_3}(x_3)\mu_{f_E\to x_3}(x_3)$$
$$\mu_{x_3\to f_E}(x_3) = \mu_{f_C\to x_3}(x_3)\mu_{f_D\to x_3}(x_3).$$

Step 5:

$$\mu_{x_1\to f_A}(x_1) = \mu_{f_C\to x_1}(x_1)$$
$$\mu_{x_2\to f_B}(x_2) = \mu_{f_C\to x_2}(x_2)$$
$$\mu_{f_D\to x_4}(x_4) = \sum_{\sim\{x_4\}} \mu_{x_3\to f_D}(x_3) f_D(x_3, x_4)$$
$$\mu_{f_E\to x_5}(x_5) = \sum_{\sim\{x_5\}} \mu_{x_3\to f_E}(x_3) f_E(x_3, x_5).$$

Termination:

$$g_1(x_1) = \mu_{f_A\to x_1}(x_1)\mu_{f_C\to x_1}(x_1)$$
$$g_2(x_2) = \mu_{f_B\to x_2}(x_2)\mu_{f_C\to x_2}(x_2)$$
$$g_3(x_3) = \mu_{f_C\to x_3}(x_3)\mu_{f_D\to x_3}(x_3)\mu_{f_E\to x_3}(x_3)$$
$$g_4(x_4) = \mu_{f_D\to x_4}(x_4)$$
$$g_5(x_5) = \mu_{f_E\to x_5}(x_5).$$

In the termination step, we compute $g_i(x_i)$ as the product of all messages directed toward x_i. Equivalently, since the message passed on any given edge is equal to the product of all but one of these messages, we may compute $g_i(x_i)$ as the product of the two messages that were passed (in opposite directions) over any single edge incident on x_i. Thus, for example, we may compute $g_3(x_3)$ in three other ways as follows:

$$\begin{aligned} g_3(x_3) &= \mu_{f_C\to x_3}(x_3)\mu_{x_3\to f_C}(x_3) \\ &= \mu_{f_D\to x_3}(x_3)\mu_{x_3\to f_D}(x_3) \\ &= \mu_{f_E\to x_3}(x_3)\mu_{x_3\to f_E}(x_3). \end{aligned}$$

III. Modeling Systems with Factor Graphs

We describe now various ways in which factor graphs may be used to model *systems*, i.e., collections of interacting variables.

In probabilistic modeling of systems, a factor graph can be used to represent the joint probability mass function of the variables that comprise the system. Factorizations of this function can give important information about statistical dependencies among these variables.

Likewise, in "behavioral" modeling of systems—as in the work of Willems [33]—system behavior is specified in set-theoretic terms by specifying which particular configurations of variables are valid. This approach can be accommodated by a factor graph that represents the characteristic (i.e., indicator) function for the given behavior. Factorizations of this characteristic function can give important structural information about the model.

In some applications, we may even wish to combine these two modeling styles. For example, in channel coding, we model both the valid behavior (i.e., the set of codewords) and the *a posteriori* joint probability mass function over the variables that define the codewords given the received output of a channel. (While it may even be feasible to model complicated channels with memory [31], in this paper we will model only memoryless channels.)

In behavioral modeling, "Iverson's convention" [14, p. 24] can be useful. If P is a predicate (Boolean proposition) involving some set of variables, then $[P]$ is the $\{0, 1\}$-valued function that indicates the truth of P, i.e.

$$[P] := \begin{cases} 1, & \text{if } P \text{ is true} \\ 0, & \text{otherwise.} \end{cases} \tag{7}$$

For example, $f(x, y, z) = [x + y = z]$ is the function that takes a value of 1 if the condition $x + y = z$ is satisfied, and 0 otherwise.

If we let $\wedge$ denote the logical conjunction or "AND" operator, then an important property of Iverson's convention is that

$$[P_1 \wedge P_2 \wedge \ldots \wedge P_n] = [P_1][P_2] \ldots [P_n] \tag{8}$$

(assuming $1 \cdot 1 = 1$ and $0 \cdot 1 = 0$). Thus, if P can be written as a logical conjunction of predicates, then $[P]$ can be factored according to (8), and hence represented by a factor graph.

A. Behavioral Modeling

Let $x_1, x_2, \ldots, x_n$ be a collection of variables with configuration space $S = A_1 \times A_2 \times \cdots \times A_n$. By a *behavior* in S, we mean any subset B of S. The elements of B are the *valid configurations*. Since a system is specified via its behavior B, this approach is known as behavioral modeling [33].

Behavioral modeling is natural for codes. If the domain of each variable is some finite alphabet A, so that the configuration space is the n-fold Cartesian product $S = A^n$, then a behavior $C \subset S$ is called a *block code* of length n over A, and the valid configurations are called *codewords*.

The characteristic (or set membership indicator) function for a behavior B is defined as

$$\chi_B(x_1, \ldots, x_n) := [(x_1, \ldots, x_n) \in B].$$

Obviously, specifying χ_B is equivalent to specifying B. (We might also give χ_B a probabilistic interpretation by noting that χ_B is proportional to a probability mass function that is uniform over the valid configurations.)

In many important cases, membership of a particular configuration in a behavior B can be determined by applying a series of tests (checks), each involving some subset of the variables. A configuration is deemed valid if and only if it passes all tests; i.e., the predicate $(x_1, \ldots, x_n) \in B$ may be written as a logical conjunction of a series of "simpler" predicates. Then χ_B factors according to (8) into a product of characteristic functions, each indicating whether a particular subset of variables is an element of some "local behavior."

▶ *Example 2 (Tanner Graphs for Linear Codes):* The characteristic function for any linear code defined by an $r \times n$ parity-check matrix H can be represented by a factor graph having n variable nodes and r factor nodes. For example, if C is the binary linear code with parity-check matrix

$$H = \begin{bmatrix} 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 & 0 \end{bmatrix} \tag{9}$$

then C is the set of all binary 6-tuples $\boldsymbol{x} \triangleq (x_1, x_2, \ldots, x_6)$ that satisfy three simultaneous equations expressed in matrix form as $H\boldsymbol{x}^T = 0$. (This is a so-called *kernel representation*, since the linear code is defined as the kernel of a particular linear transformation.) Membership in C is completely determined by checking whether *each* of the three equations is satisfied. Therefore, using (8) and (9) we have

$$\begin{aligned} \chi_C(x_1, x_2, \ldots, x_6) &= [(x_1, x_2, \ldots, x_6) \in C] \\ &= [x_1 \oplus x_2 \oplus x_5 = 0]\,[x_2 \oplus x_3 \oplus x_6 = 0] \\ &\quad \cdot [x_1 \oplus x_3 \oplus x_4 = 0] \end{aligned}$$

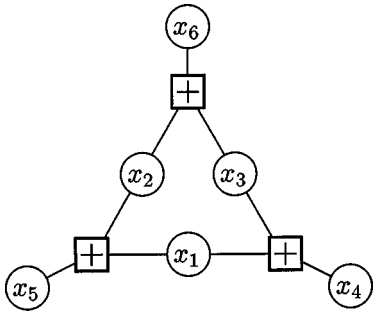

Fig. 8. A Tanner graph for the binary linear code of Example 2.

where $\oplus$ denotes the sum in GF (2). The corresponding factor graph is shown in Fig. 8, where we have used a special symbol for the parity checks (a square with a "+" sign). Although strictly speaking the factor graph represents the factorization of the code's characteristic function, we will often refer to the factor graph as representing the code itself. A factor graph obtained in this way is often called a *Tanner graph*, after [29].

It should be obvious that a Tanner graph for any $[n, k]$ linear block code may be obtained from a parity-check matrix $H = [h_{ij}]$ for the code. Such a parity-check matrix has n columns and at least $n - k$ rows. Variable nodes correspond to the columns of H and factor nodes (or checks) to the rows of H, with an edge-connecting factor node i to variable node j if and only if $h_{ij} \neq 0$. Of course, since there are, in general, many parity-check matrices that represent a given code, there are, in general, many Tanner graph representations for the code. ◀

Given a collection of general nonlinear local checks, it may be a computationally intractable problem to determine whether the corresponding behavior is nonempty. For example, the canonical NP-complete problem SAT (Boolean satisfiability) [13] is simply the problem of determining whether or not a collection of Boolean variables satisfies all clauses in a given set. In effect, each clause is a local check.

Often, a description of a system is simplified by introducing *hidden* (sometimes called auxiliary, latent, or state) variables. Nonhidden variables are called *visible*. A particular behavior B with both auxiliary and visible variables is said to represent a given (visible) behavior C if the projection of the elements of B on the visible variables is equal to C. Any factor graph for B is then considered to be also a factor graph for C. Such graphs were introduced by Wiberg *et al.* [31], [32] and may be called Wiberg-type graphs. In our factor graph diagrams, as in Wiberg, hidden variable nodes are indicated by a double circle.

An important class of models with hidden variables are the *trellis* representations (see [30] for an excellent survey). A trellis for a block code C is an edge-labeled directed graph with distinguished root and goal vertices, essentially defined by the property that each sequence of edge labels encountered in any directed path from the root vertex to the goal vertex is a codeword in C, and that each codeword in C is represented by at least one such path. Trellises also have the property that all paths from the root to any given vertex should have the same fixed length d, called the *depth* of the given vertex. The root vertex has depth 0, and the goal vertex has depth n. The set of depth d vertices can be viewed as the domain of a *state variable* s_d. For example,

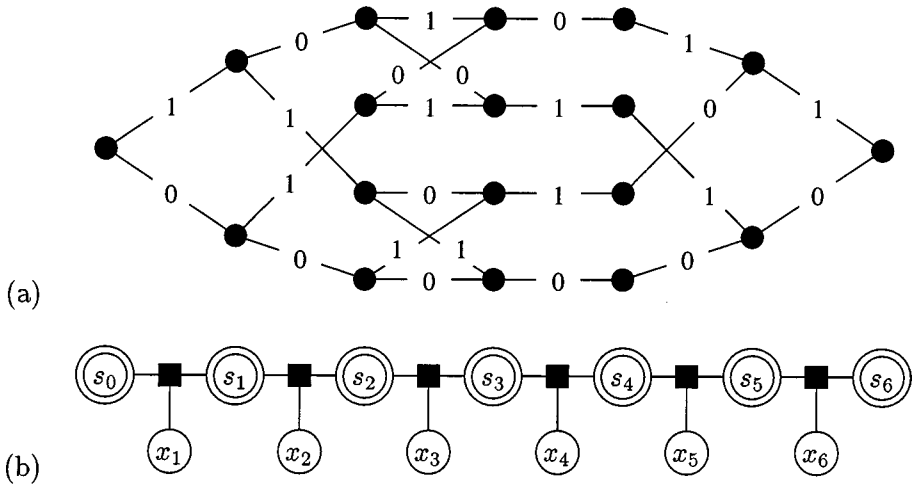

Fig. 9. (a) A trellis and (b) the corresponding Wiberg-type graph for the code of Fig. 8.

Fig. 9(a) is a trellis for the code of Example 2. Vertices at the same depth are grouped vertically. The root vertex is leftmost, the goal vertex is rightmost, and edges are implicitly directed from left to right.

A trellis divides naturally into n *sections*, where the ith trellis section T_i is the subgraph of the trellis induced by the vertices at depth $i-1$ and depth i. The set of edge labels in T_i may be viewed as the domain of a (visible) variable x_i. In effect, each trellis section T_i defines a "local behavior" that constrains the possible combinations of s_{i-1}, x_i, and s_i.

Globally, a trellis defines a behavior in the configuration space of the variables $s_0, \ldots, s_n, x_1, \ldots, x_n$. A configuration of these variables is valid if and only if it satisfies the local constraints imposed by each of the trellis sections. The characteristic function for this behavior thus factors naturally into n factors, where the ith factor corresponds to the ith trellis section T_i and has s_{i-1}, x_i, and s_i as its arguments.

The following example illustrates these concepts in detail for the code of Example 2.

▶ *Example 3 (A Trellis Description):* Fig. 9(a) shows a trellis for the code of Example 2, and Fig. 9(b) shows the corresponding Wiberg-type graph. In addition to the visible variable nodes $x_1, x_2, \ldots, x_6$, there are also hidden (state) variable nodes $s_0, s_1, \ldots, s_6$. Each local check, shown as a generic factor node (black square), corresponds to one section of the trellis.

In this example, the local behavior T_2 corresponding to the second trellis section from the left in Fig. 9 consists of the following triples (s_1, x_2, s_2):

$$T_2 = \{(0, 0, 0), (0, 1, 2), (1, 1, 1), (1, 0, 3)\} \tag{10}$$

where the domains of the state variables s_1 and s_2 are taken to be $\{0, 1\}$ and $\{0, 1, 2, 3\}$, respectively, numbered from bottom to top in Fig. 9(a). Each element of the local behavior corresponds to one trellis edge. The corresponding factor node in the Wiberg-type graph is the indicator function $f(s_1, x_2, s_2) = [(s_1, x_2, s_2) \in T_2]$. ◀

It is important to note that a factor graph corresponding to a trellis is cycle-free. Since every code has a trellis representation, it follows that every code can be represented by a cycle-free factor graph. Unfortunately, it often turns out that the state-space sizes (the sizes of domains of the state variables) can easily become too large to be practical. For example, trellis representations of turbo codes have enormous state spaces [12]. However, such codes may well have factor graph representations with reasonable complexities, but necessarily with cycles. Indeed, the "cut-set bound" of [31] (see also [8]) strongly motivates the study of graph representations with cycles.

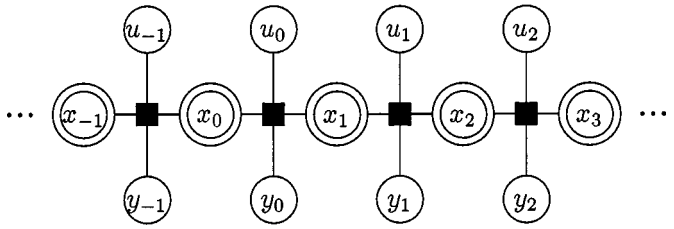

Fig. 10. Generic factor graph for a state-space model of a time-invariant or time-varying system.

Trellises are basically conventional state-space system models, and the generic factor graph of Fig. 10 can represent any state-space model of a time-invariant or time-varying system. As in Fig. 9, each local check represents a trellis section; i.e., each check is an indicator function for the set of allowed combinations of left (previous) state, input symbol, output symbol, and right (next) state. (Here, we allow a trellis edge to have both an input label and an output label.)

▶ *Example 4 (State-Space Models):* For example, the classical linear time-invariant state-space model is given by the equations

$$\begin{aligned} x(j+1) &= Ax(j) + Bu(j) \\ y(j) &= Cx(j) + Du(j) \end{aligned} \tag{11}$$

where $j \in \mathbb{Z}$ is the discrete time index, $u(j) = (u_1(j), \ldots, u_k(j))$ are the time-j input variables, $y(j) = (y_1(j), \ldots, y_n(j))$ are the time-j output variables, $x(j) = (x_1(j), \ldots, x_m(j))$ are the time-j state variables, A, B, C, and D are matrices of appropriate dimension, and the equations are over some field F.

Any such system gives rise to the factor graph of Fig. 10. The time-j check function

$$f(x(j), u(j), y(j), x(j+1)) : F^m \times F^k \times F^n \times F^m \to \{0, 1\}$$

is

$$f(x(j), u(j), y(j), x(j+1)) = [x(j+1) = Ax(j) + Bu(j)]\,[y(j) = Cx(j) + Du(j)].$$

In other words, the check function enforces the local behavior defined by (11). ◀

B. Probabilistic Modeling

We turn now to another important class of functions that we will represent by factor graphs: probability distributions. Since conditional and unconditional independence of random variables is expressed in terms of a factorization of their joint probability mass or density function, factor graphs for probability distributions arise in many situations. We begin again with an example from coding theory.

▶ *Example 5 (APP Distributions):* Consider the standard coding model in which a codeword $x = (x_1, \ldots, x_n)$ is selected from a code C of length n and transmitted over a memoryless channel with corresponding output sequence $y = (y_1, \ldots, y_n)$. For each fixed observation y, the joint *a posteriori* probability (APP) distribution for the components of x (i.e., $p(x|y)$) is *proportional to* the function $g(x) = f(y|x)p(x)$, where $p(x)$ is the *a priori* distribution for the transmitted vectors, and $f(y|x)$ is the conditional probability density function for y when x is transmitted.

Since the observed sequence y is fixed for any instance of "decoding" a graph we may consider $g(x)$ to be a function of x only, with the components of y regarded as parameters. In other words, we may write $f(y|x)$ or $p(x|y)$ as $f_y(x)$, meaning that the expression to be "decoded" always has the same parametric form, but that the parameter y will in general be different in different decoding instances.

Assuming that the *a priori* distribution for the transmitted vectors is uniform over codewords, we have $p(x) = \chi_C(x)/|C|$, where $\chi_C(x)$ is the characteristic function for C and $|C|$ is the number of codewords in C. If the channel is memoryless, then $f(y|x)$ factors as

$$f(y_1, \ldots, y_n|x_1, \ldots, x_n) = \prod_{i=1}^{n} f(y_i|x_i).$$

Under these assumptions, we have

$$g(x_1, \ldots, x_n) = \frac{1}{|C|}\,\chi_C(x_1, \ldots, x_n) \prod_{i=1}^{n} f(y_i|x_i). \quad (12)$$

Now the characteristic function $\chi_C(x)$ itself may factor into a product of local characteristic functions, as described in the previous subsection. Given a factor graph F for $\chi_C(x)$, we obtain a factor graph for (a scaled version of) the APP distribution over x simply by *augmenting* F with factor nodes corresponding to the different $f(y_i|x_i)$ factors in (12). The ith such factor has only one argument, namely x_i, since y_i is regarded as a parameter. Thus, the corresponding factor nodes appear as pendant vertices ("dongles") in the factor graph.

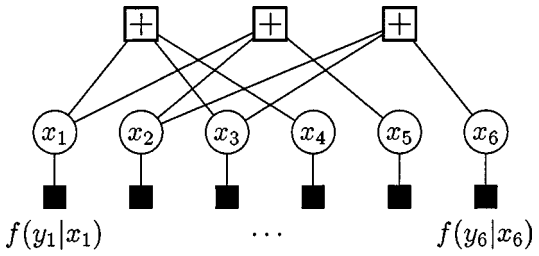

Fig. 11. Factor graph for the joint APP distribution of codeword symbols.

For example, if C is the binary linear code of Example 2, then we have

$$g(x_1, \ldots, x_6) = [x_1 \oplus x_2 \oplus x_5 = 0] \cdot [x_2 \oplus x_3 \oplus x_6 = 0] \cdot [x_1 \oplus x_3 \oplus x_4 = 0] \cdot \prod_{i=1}^{6} f(y_i|x_i)$$

whose factor graph is shown in Fig. 11. ◀

Various types of Markov models are widely used in signal processing and communications. The key feature of such models is that they imply a nontrivial factorization of the joint probability mass function of the random variables in question. This factorization may be represented by a factor graph.

▶ *Example 6 (Markov Chains, Hidden Markov Models):* In general, let $f(x_1, \ldots, x_n)$ denote the joint probability mass function of a collection of random variables. By the chain rule of conditional probability, we may always express this function as

$$f(x_1, \ldots, x_n) = \prod_{i=1}^{n} f(x_i|x_1, \ldots, x_{i-1}).$$

For example, if $n = 4$, then

$$f(x_1, \ldots, x_4) = f(x_1) f(x_2|x_1) f(x_3|x_1, x_2) f(x_4|x_1, x_2, x_3)$$

which has the factor graph representation shown in Fig. 12(b).

In general, since all variables appear as arguments of $f(x_n|x_1, \ldots, x_{n-1})$, the factor graph of Fig. 12(b) has no advantage over the trivial factor graph shown in Fig. 12(a). On the other hand, suppose that random variables $X_1, X_2, \ldots, X_n$ (in that order) form a Markov chain. We then obtain the nontrivial factorization

$$f(x_1, \ldots, x_n) = \prod_{i=1}^{n} f(x_i|x_{i-1})$$

whose factor graph is shown in Fig. 12(c) for $n = 4$.

Continuing this Markov chain example, if we cannot observe each X_i directly, but instead can observe only Y_i, the output of a memoryless channel with X_i as input, then we obtain a so-called "hidden Markov model." The joint probability mass or density function for these random variables then factors as

$$f(x_1, \ldots, x_n, y_1, \ldots, y_n) = \prod_{i=1}^{n} f(x_i|x_{i-1}) f(y_i|x_i)$$

whose factor graph is shown in Fig. 12(d) for $n = 4$. Hidden Markov models are widely used in a variety of applications; e.g., see [27] for a tutorial emphasizing applications in signal processing.

Of course, since trellises may be regarded as Markov models for codes, the strong resemblance between the factor graphs of

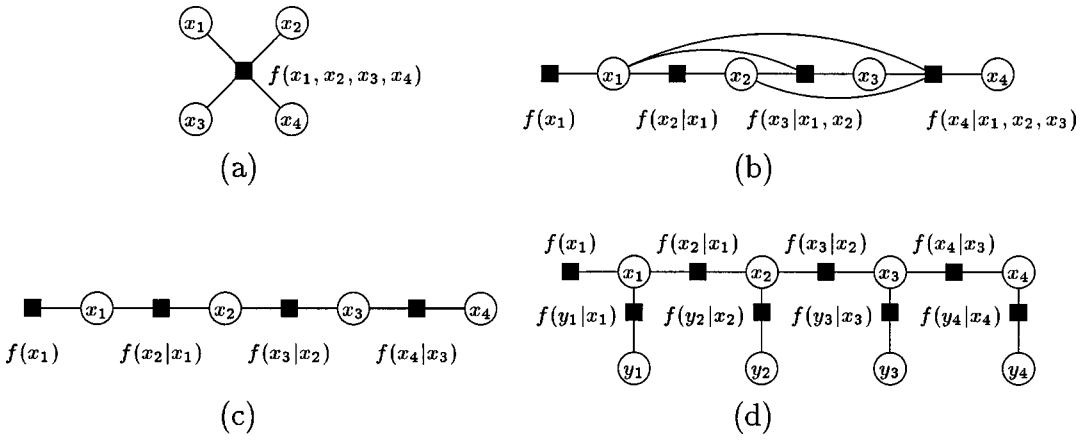

Fig. 12. Factor graphs for probability distributions. (a) The trivial factor graph. (b) The chain-rule factorization. (c) A Markov chain. (d) A hidden Markov model.

Fig. 12(c) and (d) and the factor graphs representing trellises (Figs. 9(b) and 10) is not accidental. ◀

In Appendix B we describe very briefly the close relationship between factor graphs and other graphical models for probability distributions: models based on undirected graphs (Markov random fields) and models based on directed acyclic graphs (Bayesian networks).

IV. Trellis Processing

As described in the previous section, an important family of factor graphs contains the chain graphs that represent trellises or Markov models. We now apply the sum-product algorithm to such graphs, and show that a variety of well-known algorithms—the forward/backward algorithm, the Viterbi algorithm, and the Kalman filter—may be viewed as special cases of the sum-product algorithm.

A. The Forward/Backward Algorithm

We start with the forward/backward algorithm, sometimes referred to in coding theory as the BCJR [4], APP, or "MAP" algorithm. This algorithm is an application of the sum-product algorithm to the hidden Markov model of Example 6, shown in Fig. 12(d), or to the trellises of examples Examples 3 and 4 (Figs. 9 and 10) in which certain variables are observed at the output of a memoryless channel.

The factor graph of Fig. 13 models the most general situation, which involves a combination of behavioral and probabilistic modeling. We have vectors $u = (u_1, u_2, \ldots, u_k)$, $x = (x_1, x_1, \ldots, x_n)$, and $s = (s_0, \ldots, s_n)$ that represent, respectively, input variables, output variables, and state variables in a Markov model, where each variable is assumed to take on values in a finite domain. The behavior is defined by local check functions $T_i(s_{i-1}, x_i, u_i, s_i)$, as described in Examples 3 and 4. To handle situations such as terminated convolutional codes, we also allow for the input variable to be suppressed in certain trellis sections, as in the rightmost trellis section of Fig. 13.

This model is a "hidden" Markov model in which we cannot observe the output symbols directly. As discussed in Example 5,

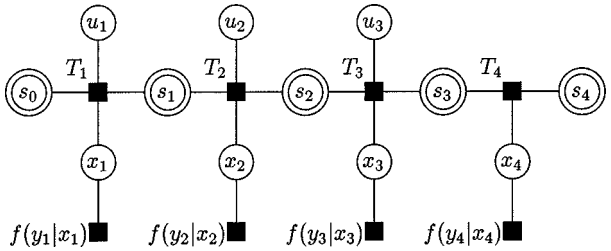

Fig. 13. The factor graph in which the forward/backward algorithm operates: the s_i are state variables, the u_i are input variables, the x_i are output variables, and each y_i is the output of a memoryless channel with input x_i.

the *a posteriori* joint probability mass function for u, s, and x given the observation y is proportional to

$$g_y(u, s, x) := \prod_{i=1}^{n} T_i(s_{i-1}, x_i, u_i, s_i) \prod_{i=1}^{n} f(y_i|x_i)$$

where y is again regarded as a parameter of g (not an argument). The factor graph of Fig. 13 represents this factorization of g.

Given y, we would like to find the APPs $p(u_i|y)$ for each i. These marginal probabilities are proportional to the following marginal functions associated with g:

$$p(u_i|y) \propto \sum_{\sim\{u_i\}} g_y(u, s, x).$$

Since the factor graph of Fig. 13 is cycle-free, these marginal functions may be computed by applying the sum-product algorithm to the factor graph of Fig. 13.

Initialization: As usual in a cycle-free factor graph, the sum-product algorithm begins at the leaf nodes. Trivial messages are sent by the input variable nodes and the endmost state variable nodes. Each pendant factor node sends a message to the corresponding output variable node. As discussed in Section II, since the output variable nodes have degree two, no computation is performed; instead, incoming messages received on one edge are simply transferred to the other edge and sent to the corresponding trellis check node.

Once the initialization has been performed, the two endmost trellis check nodes T_1 and T_n will have received messages on three of their four edges, and so will be in a position to create an output message to send to a neighboring state variable node.

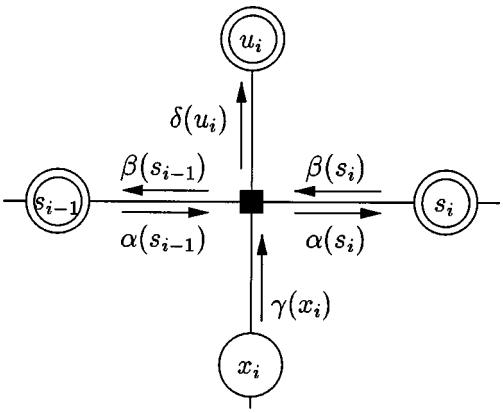

Fig. 14. A detailed view of the messages passed during the operation of the forward/backward algorithm.

Again, since the state variables have degree two, no computation is performed; at state nodes messages received on one edge are simply transferred to the other edge.

In the literature on the forward/backward algorithm (e.g., [4]), the message $\mu_{x_i \to T_i}(x_i)$ is denoted as $\gamma(x_i)$, the message $\mu_{s_i \to T_{i+1}}(s_i)$ is denoted as $\alpha(s_i)$, and the message $\mu_{s_i \to T_i}(s_i)$ is denoted as $\beta(s_i)$. Additionally, the message $\mu_{T_i \to u_i}(u_i)$ will be denoted as $\delta(u_i)$.

The operation of the sum-product algorithm creates two natural recursions: one to compute $\alpha(s_i)$ as a function of $\alpha(s_{i-1})$ and $\gamma(x_i)$ and the other to compute $\beta(s_{i-1})$ as a function of $\beta(s_i)$ and $\gamma(x_i)$. These two recursions are called the *forward* and *backward* recursions, respectively, according to the direction of message flow in the trellis. The forward and backward recursions do not interact, so they could be computed in parallel.

Fig. 14 gives a detailed view of the message flow for a single trellis section. The local function in this figure represents the trellis check $T_i(s_{i-1}, u_i, x_i, s_i)$.

The Forward/Backward Recursions: Specializing the general update equation (6) to this case, we find

$$\alpha(s_i) = \sum_{\sim\{s_i\}} T_i(s_{i-1}, u_i, x_i, s_i)\alpha(s_{i-1})\gamma(x_i)$$

$$\beta(s_{i-1}) = \sum_{\sim\{s_{i-1}\}} T_i(s_{i-1}, u_i, x_i, s_i)\beta(s_i)\gamma(x_i).$$

Termination: The algorithm terminates with the computation of the $\delta(u_i)$ messages.

$$\delta(u_i) = \sum_{\sim\{u_i\}} T_i(s_{i-1}, u_i, x_i, s_i)\alpha(s_{i-1})\beta(s_{i+1})\gamma(x_i).$$

These sums can be viewed as being defined over valid trellis edges $e = (s_{i-1}, u_i, x_i, s_i)$ such that $T_i(e) = 1$. For each edge e, we let $\alpha(e) = \alpha(s_{i-1})$, $\beta(e) = \beta(s_i)$, and $\gamma(e) = \gamma(x_i)$. Denoting by $E_i(s)$ the set of edges incident on a state s in the ith trellis section, the α and β update equations may be rewritten as

$$\alpha(s_i) = \sum_{e \in E_i(s_i)} \alpha(e)\gamma(e)$$

$$\beta(s_{i-1}) = \sum_{e \in E_i(s_{i-1})} \beta(e)\gamma(e). \tag{13}$$

The basic operations in the forward and backward recursions are therefore "sums of products."

The α and β messages have a well-defined probabilistic interpretation: $\alpha(s_{i-1})$ is proportional to the conditional probability mass function for s_{i-1} given the "past" $y_1, \ldots, y_{i-1}$; i.e., for each state $s_{i-1} \in S_{i-1}$, $\alpha(s_{i-1})$ is proportional to the conditional probability that the transmitted sequence passed through state s_{i-1} given the past. Similarly, $\beta(s_i)$ is proportional to the conditional probability mass function for s_i given the "future" $y_{i+1}, y_{i+2}, \ldots$, i.e., the conditional probability that the transmitted sequence passed through state s_i. The probability that the transmitted sequence passed through a particular edge $e = (s_{i-1}, u_i, x_i, s_i) \in T_i$ is thus given by

$$\alpha(s_i)\gamma(x_i)\beta(s_{i+1}) = \alpha(e)\gamma(e)\beta(e).$$

Note that if we were interested in the APPs for the state variables s_i or the symbol variables x_i, these could also be computed by the forward/backward algorithm.

B. The Min-Sum and Max-Product Semirings and the Viterbi Algorithm

We might in many cases be interested in determining which valid configuration has largest APP, rather than determining the APPs for the individual symbols. When all codeword are *a priori* equally likely, this amounts to maximum-likelihood sequence detection (MLSD).

As mentioned in Section II (see also [31], [2]), the codomain R of the global function g represented by a factor graph may in general be any semiring with two operations "$+$" and "$\cdot$" that satisfy the distributive law

$$(\forall\, x, y, z \in R) \qquad x \cdot (y + z) = (x \cdot y) + (x \cdot z). \tag{14}$$

In any such semiring, a product of local functions is well defined, as is the notion of summation of values of g. It follows that the "not-sum" or summary operation is also well-defined. In fact, our observation that the structure of a cycle-free factor graph encodes expressions (i.e., algorithms) for the computation of marginal functions essentially follows from the distributive law (14), and so applies equally well to the general semiring case. This observation is key to the "generalized distributive law" of [2].

A semiring of particular interest for the MLSD problem is the "max-product" semiring, in which real summation is replaced with the "max" operator. For nonnegative real-valued quantities x, y, and z, "$\cdot$" distributes over "max"

$$x(\max(y, z)) = \max(xy, xz).$$

Furthermore, with maximization as a summary operator, the maximum value of a nonnegative real-valued function $g(x_1, \ldots, x_n)$ is viewed as the "complete summary" of g; i.e.

$$\max g(x_1, \ldots, x_n) = \max_{x_1}(\max_{x_2}(\ldots(\max_{x_n} g(x_1, \ldots, x_n))))$$
$$= \sum_{\sim\{\,\}} g(x_1, \ldots, x_n).$$

For the MLSD problem, we are interested not so much in determining this maximum value, as in finding a valid configuration $\boldsymbol{x}$ that achieves this maximum.

In practice, MLSD is most often carried out in the negative log-likelihood domain. Here, the "product" operation becomes a "sum" and the "max" operation becomes a "min" operation,

so that we deal with the "min-sum" semiring. For real-valued quantities x, y, z, "$+$" distributes over "min"

$$x + \min(y, z) = \min(x + y, x + z).$$

We extend Iverson's convention to the general semiring case by assuming that R contains a multiplicative identity u and a null element z such that $u \cdot x = x$ and $z \cdot x = z$ for all $x \in R$. When P is a predicate, then by $[P]$ we mean the $\{z, u\}$-valued function that takes value u whenever P is true and z otherwise. In the "min-sum" semiring, where the "product" is real addition, we take $u = 0$ and $z = \infty$. Under this extension of Iverson's convention, factor graphs representing codes are not affected by the choice of semiring.

Consider again the chain graph that represents a trellis, and suppose that we apply the min-sum algorithm; i.e., the sum-product algorithm in the min-sum semiring. Products of positive functions (in the regular factor graph) are converted to sums of functions (appropriate for the min-sum semiring) by taking their negative logarithm. Indeed, such functions can be scaled and shifted (e.g., setting $f(x, s|y) = -a \ln p(x, s|y) + b$ where a and b are constants with $a > 0$) in any manner that is convenient. In this way, for example, we may obtain squared Euclidean distance as a "branch metric" in Gaussian channels, and Hamming distance as a "branch metric" in discrete symmetric channels.

Applying the min-sum algorithm in this context yields the same message flow as in the forward/backward algorithm. As in the forward/backward algorithm, we may write an update equation for the various messages. For example, the basic update equation corresponding to (13) is

$$\alpha(s_i) = \min_{e \in E_i(s_i)} (\alpha(e) + \gamma(e)) \tag{15}$$

so that the basic operation is a "minimum of sums" instead of a "sum of products." A similar recursion may be used in the backward direction, and from the results of the two recursions the most likely sequence may be determined. The result is a "bidirectional" Viterbi algorithm.

The conventional Viterbi algorithm operates in the forward direction only; however, since memory of the best path is maintained and some sort of "traceback" is performed in making a decision, even the conventional Viterbi algorithm might be viewed as being bidirectional.

C. Kalman Filtering

In this section, we derive the Kalman filter (see, e.g., [3], [23]) as an instance of the sum-product algorithm operating in the factor graph corresponding to a discrete-time linear dynamical system similar to that given by (11). For simplicity, we focus on the case in which all variables are scalars satisfying

$$x_{j+1} = A_j x_j + B_j u_j$$
$$y_j = C_j x_j + D_j w_j$$

where x_j, y_j, u_j, and w_j are the time-j state, output, input, and noise variables, respectively, and A_j, B_j, C_j, and D_j are assumed to be known time-varying scalars. Generalization to the case of vector variables is standard, but will not be pursued here. We assume that the input u and noise w are independent white Gaussian noise sequences with zero mean and unit variance, and that the state sequence is initialized by setting $x_0 = 0$. Since linear combinations of jointly Gaussian random variables are Gaussian, it follows that the x_j and y_j sequences are jointly Gaussian.

We use the notation

$$\mathcal{N}(x, m, \sigma^2) \propto \exp\left(-(x - m)^2/(2\sigma^2)\right)$$

to represent Gaussian density functions, where m and σ^2 represent the mean and variance. By completing the square in the exponent, we find that

$$\mathcal{N}(x, m_1, \sigma_1^2)\mathcal{N}(x, m_2, \sigma_2^2) \propto \mathcal{N}(x, m_3, \sigma_3^2) \tag{16}$$

where

$$m_3 = \frac{\sigma_2^2 m_1 + \sigma_1^2 m_2}{\sigma_1^2 + \sigma_2^2}$$

and

$$\frac{1}{\sigma_3^2} = \frac{1}{\sigma_1^2} + \frac{1}{\sigma_2^2}.$$

Similarly, we find that

$$\int_{-\infty}^{\infty} \mathcal{N}(x, m_1, \sigma_1^2)\mathcal{N}(y, \alpha x, \sigma_2^2)\, dx \propto \mathcal{N}(y, \alpha m_1, \alpha^2\sigma_1^2 + \sigma_2^2). \tag{17}$$

As in Example 5, the Markov structure of this system permits us to write the conditional joint probability density function of the state variables $x_1, \ldots, x_k$ given $y_1, \ldots, y_k$ as

$$f(x_1, \ldots, x_k|y_1, \ldots, y_k) = \prod_{j=1}^{k} f(x_j|x_{j-1}) f(y_j|x_j) \tag{18}$$

where $f(x_j|x_{j-1})$ is a Gaussian density with mean $A_{j-1}x_{j-1}$ and variance B_{j-1}^2, and $f(y_j|x_j)$ is a Gaussian density with mean $C_j x_j$ and variance D_j^2. Again, the observed values of the output variables are regarded as parameters, not as function arguments.

The conditional density function for x_k given observations up to time k is the marginal function

$$P_{k|k}(x_k) = f(x_k|y_1, \ldots, y_k) = \int_{\sim\{x_k\}} f(x_1, \ldots, x_k|y_1, \ldots, y_k)\, d(\sim\{x_k\})$$

where we have introduced an obvious generalization of the "not-sum" notation to integrals. The mean of this conditional density, $\hat{x}_{k|k} = E[x_k|y_1, \ldots, y_k]$, is the minimum mean-squared-error (MMSE) estimate of x_k given the observed outputs. This conditional density function can be computed via the sum-product algorithm, using integration (rather than summation) as the summary operation.

A portion of the factor graph that describes (18) is shown in Fig. 15. Also shown in Fig. 15 are messages that are passed in the operation of the sum-product algorithm. We denote by $P_{j|j-1}(x_j)$ the message passed to x_j from $f(x_j|x_{j-1})$. Up to scale, this message is always of the form $\mathcal{N}(x_j, \hat{m}_{j|j-1}, \sigma_{j|j-1}^2)$, and so may be represented by the pair $(\hat{m}_{j|j-1}, \sigma_{j|j-1}^2)$. We interpret $\hat{m}_{j|j-1}$ as the MMSE *prediction* of x_j given the set of observations up to time $j - 1$.

According to the product rule, applying (16), we have

$$\begin{aligned} P_{j|j}(x_j) &= P_{j|j-1}(x_j) f(y_j|x_j) \\ &= \mathcal{N}(x_j, \hat{m}_{j|j-1}, \sigma_{j|j-1}^2)\mathcal{N}(y_j, C_j x_j, D_j^2) \\ &\propto \mathcal{N}(x_j, \hat{m}_{j|j-1}, \sigma_{j|j-1}^2)\mathcal{N}(x_j, y_j/C_j, D_j^2/C_j^2) \\ &\propto \mathcal{N}(x_j, \hat{m}_{j|j}, \sigma_{j|j}^2) \end{aligned}$$

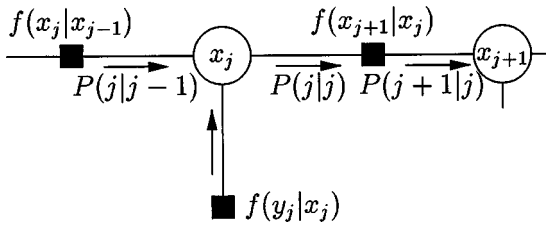

Fig. 15. A portion of the factor graph corresponding to (18).

where

$$\hat{m}_{j|j} = \frac{D_j^2 \hat{m}_{j|j-1} + C_j y_j \sigma_{j|j-1}^2}{C_j^2 \sigma_{j|j-1}^2 + D_j^2}$$
$$= \hat{m}_{j|j-1} + \frac{C_j \sigma_{j|j-1}^2}{C_j^2 \sigma_{j|j-1}^2 + D_j^2}(y_j - C_j \hat{m}_{j|j-1})$$

and

$$\sigma_{j|j}^2 = \frac{D_j^2 \sigma_{j|j-1}^2}{C_j^2 \sigma_{j|j-1}^2 + D_j^2}.$$

Likewise, applying (17), we have

$$P_{j+1|j}(x_{j+1}) = \int P_{j|j}(x_j)\mathcal{N}(x_{j+1}, A_j x_j, B_j^2)\, dx_j$$
$$\propto \mathcal{N}(x_{j+1}, \hat{m}_{j+1|j}, \sigma_{j|j+1}^2)$$

where

$$\hat{m}_{j+1|j} = A_j \hat{m}_{j|j}$$
$$= A_j \hat{m}_{j|j-1} + K_j(y_j - C_j \hat{m}_{j|j-1}) \qquad (19)$$

and

$$\sigma_{j+1|j}^2 = A_j^2 \sigma_{j|j}^2 + B_j^2$$
$$= \frac{A_j^2 \sigma_{j|j-1}^2}{C_j^2 \sigma_{j|j-1}^2 + D_j^2} + B_j^2.$$

In (19), the value

$$K_j = \frac{A_j C_j \sigma_{j|j-1}^2}{C_j^2 \sigma_{j|j-1}^2 + D_j^2}$$

is called the *filter gain*.

These updates are those used by a Kalman filter [3]. As mentioned, generalization to the vector case is standard. We note that similar updates would apply to any cycle-free factor graph in which all distributions (factors) are Gaussian. The operation of the sum-product algorithm in such a graph can, therefore, be regarded as a generalized Kalman filter, and in a graph with cycles as an iterative approximation to the Kalman filter.

V. Iterative Processing: The Sum-Product Algorithm in Factor Graphs with Cycles

In addition to its application to cycle-free factor graphs, the sum-product algorithm may also be applied to factor graphs with cycles simply by following the same message propagation rules, since all updates are local. Because of the cycles in the graph, an "iterative" algorithm with no natural termination will result, with messages passed multiple times on a given edge. In contrast with the cycle-free case, the results of the sum-product algorithm operating in a factor graph with cycles cannot in general be interpreted as exact function summaries. However, some of the most exciting applications of the sum-product algorithm—for example, the decoding of turbo codes or LDPC codes—arise precisely in situations in which the underlying factor graph *does* have cycles. Extensive simulation results (see, e.g., [5], [21], [22]) show that with very long codes such decoding algorithms can achieve astonishing performance (within a small fraction of a decibel of the Shannon limit on a Gaussian channel) even though the underlying factor graph has cycles.

Descriptions of the way in which the sum-product algorithm may be applied to a variety of "compound codes" are given in [19]. In this section, we restrict ourselves to three examples: turbo codes [5], LDPC codes [11], and repeat–accumulate (RA) codes [6].

A. Message-Passing Schedules

Although a clock may not be necessary in practice, we assume that messages are synchronized with a global discrete-time clock, with at most one message passed on any edge in any given direction at one time. Any such message effectively *replaces* previous messages that might have been sent on that edge in the same direction. A message sent from node v at time i will be a function only of the local function at v (if any) and the (most recent) messages received at v prior to time i.

Since the message sent by a node v on an edge in general depends on the messages that have been received on *other* edges at v, and a factor graph with cycles may have no nodes of degree one, how is message passing initiated? We circumvent this difficulty by initially supposing that a unit message (i.e., a message representing the unit function) has arrived on every edge incident on any given vertex. With this convention, *every* node is in a position to send a message at every time along every edge.

A message-passing *schedule* in a factor graph is a specification of messages to be passed during each clock tick. Obviously a wide variety of message-passing schedules are possible. For example, the so-called *flooding schedule* [19] calls for a message to pass in each direction over each edge at each clock tick. A schedule in which at most one message is passed anywhere in the graph at each clock tick is called a *serial schedule*.

We will say that a vertex v has a message *pending* at an edge e if it has received any messages on edges other than e *after* the transmission of the most previous message on e. Such a message is pending since the messages more recently received can affect the message to be sent on e. The receipt of a message at v from an edge e will create pending messages at all *other* edges incident on v. Only pending messages need to be transmitted, since only pending messages can be different from the previous message sent on a given edge.

In a cycle-free factor graph, assuming a schedule in which only pending messages are transmitted, the sum-product algorithm will eventually halt in a state with no messages pending. In a factor graph with cycles, however, it is impossible to reach a state with no messages pending, since the transmission of a message on any edge of a cycle from a node v will trigger a chain of pending messages that must return to v, triggering v to send another message on the same edge, and so on indefinitely.

In practice, all schedules are finite. For a finite schedule, the sum-product algorithm terminates by computing, for each x_i, the product of the most recent messages received at variable

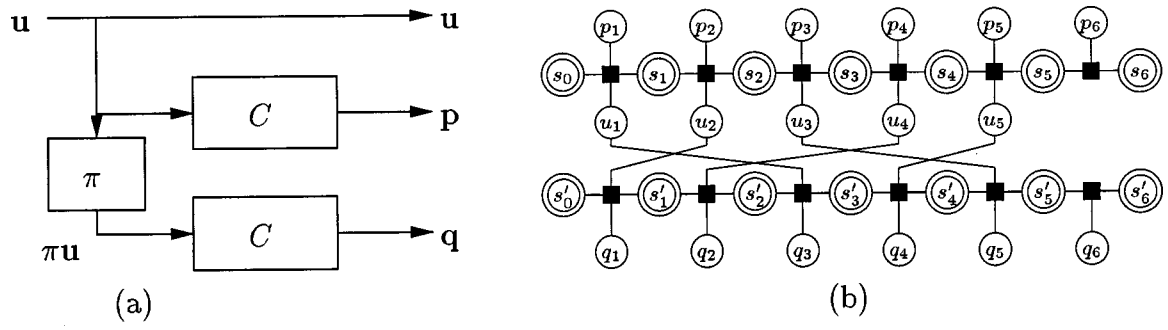

Fig. 16. Turbo code. (a) Encoder block diagram. (b) Factor graph.

node x_i. If x_i has no messages pending, then this computation is equivalent to the product of the messages sent and received on any single edge incident on x_i.

B. Iterative Decoding of Turbo Codes

A "turbo code" ("parallel concatenated convolutional code") has the encoder structure shown in Fig. 16(a). A block u of data to be transmitted enters a systematic encoder which produces u, and two parity-check sequences p and q at its output. The first parity-check sequence p is generated via a standard recursive convolutional encoder; viewed together, u and p would form the output of a standard rate $1/2$ convolutional code. The second parity-check sequence q is generated by applying a permutation π to the input stream, and applying the permuted stream to a second convolutional encoder. All output streams u, p, and q are transmitted over the channel. Both constituent convolutional encoders are typically terminated in a known ending state.

A factor graph representation for a (very) short turbo code is shown in Fig. 16(b). Included in the figure are the state variables for the two constituent encoders, as well as a terminating trellis section in which no data is absorbed, but outputs are generated. Except for the interleaver (and the short block length), this graph is generic, i.e., all standard turbo codes may be represented in this way.

Iterative decoding of turbo codes is usually accomplished via a message-passing schedule that involves a forward/backward computation over the portion of the graph representing one constituent code, followed by propagation of messages between encoders (resulting in the so-called *extrinsic* information in the turbo-coding literature). This is then followed by another forward/backward computation over the other constituent code, and propagation of messages back to the first encoder. This schedule of messages is illustrated in [19, Fig. 10]; see also [31].

C. LDPC Codes

LDPC codes were introduced by Gallager [11] in the early 1960s. LDPC codes are defined in terms of a regular bipartite graph. In a (j, k) LDPC code, left nodes, representing codeword symbols, all have degree j, while right nodes, representing checks, all have degree k. For example, Fig. 17 illustrates the factor graph for a short $(2, 4)$ LDPC code. The check enforces the condition that the adjacent symbols should have even overall parity, much as in Example 2. As in Example 2, this factor graph is just the original unadorned Tanner graph for the code.

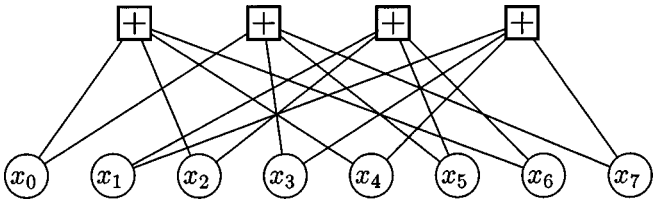

Fig. 17. A factor graph for a LDPC code.

LDPC codes, like turbo codes, are very effectively decoded using the sum-product algorithm; for example MacKay and Neal report excellent performance results approaching that of turbo codes using what amounts to a flooding schedule [21], [22].

D. RA Codes

RA codes are a special, low-complexity class of turbo codes introduced by Divsalar, McEliece, and others, who initially devised these codes because their ensemble weight distributions are relatively easy to derive. An encoder for an RA code operates on k input bits $u_1, \ldots, u_k$, repeating each bit Q times, and permuting the result to arrive at a sequence $z_1, \ldots, z_{kQ}$. An output sequence $x_1, \ldots, x_{kQ}$ is formed via an accumulator that satisfies $x_1 = z_1$ and $x_i = x_{i-1} + z_i$ for $i > 1$.

Two equivalent factor graphs for an RA code are shown in Fig. 18. The factor graph of Fig. 18(a) is a straightforward representation of the encoder as described in the previous paragraph. The checks all enforce the condition that incident variables sum to zero modulo 2. (Thus a degree-two check enforces equality of the two incident variables.) The equivalent but slightly less complicated graph of Fig. 18(b) uses equality constraints to represent the same code. Thus, e.g., $w_1 = w_2 = w_3$, corresponding to input variable u_1 and state variables z_5, z_8, and z_{11} of Fig. 18(a).

E. Simplifications for Binary Variables and Parity Checks

For particular decoding applications, the generic updating rules (5) and (6) can often be simplified substantially . We treat here only the important case where all variables are binary (Bernoulli) and all functions except single-variable functions are parity checks or repetition (equality) constraints, as in Figs. 11, 17, and 18. This includes, in particular, LDPC codes and RA codes. These simplifications are well known, some dating back to the work of Gallager [11].

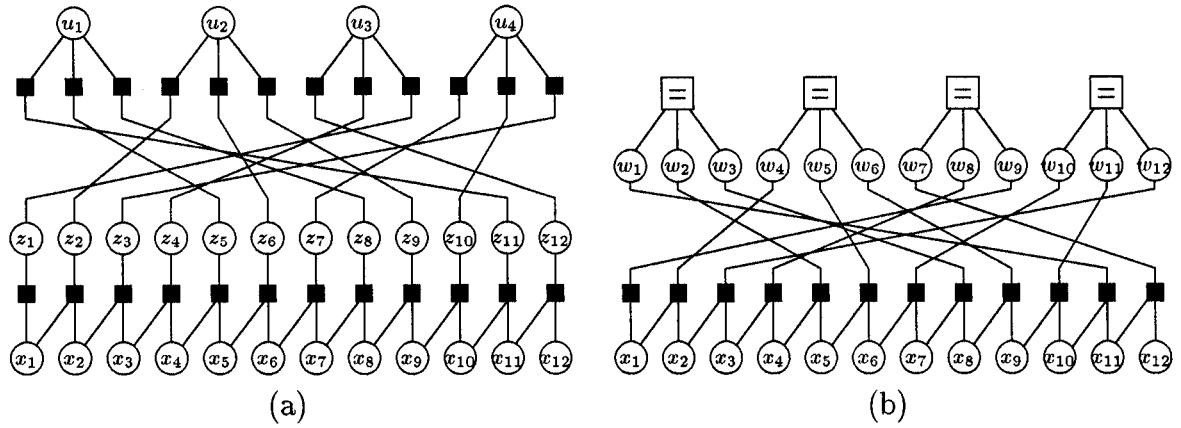

Fig. 18. Equivalent factor graphs for an RA code.

The probability mass function for a binary random variable may be represented by the vector (p_0, p_1), where $p_0 + p_1 = 1$. According to the generic updating rules, when messages (p_0, p_1) and (q_0, q_1) arrive at a variable node of degree three, the resulting (normalized) output message should be

$$\mathrm{VAR}(p_0, p_1, q_0, q_1) = \left(\frac{p_0 q_0}{p_0 q_0 + p_1 q_1}, \frac{p_1 q_1}{p_0 q_0 + p_1 q_1}\right). \quad (20)$$

Similarly, at a check node representing the function

$$f(x, y, z) = [x \oplus y \oplus z = 0]$$

(where "$\oplus$" represents modulo-2 addition), we have

$$\mathrm{CHK}(p_0, p_1, q_0, q_1) = (p_0 q_0 + p_1 q_1, p_0 q_1 + p_1 q_0). \quad (21)$$

We note that at check node representing the dual (repetition constraint) $f(x, y, z) = [x = y = z]$, we would have

$$\mathrm{REP}(p_0, p_1, q_0, q_1) = \mathrm{VAR}(p_0, p_1, q_0, q_1)$$

i.e., the update rules for repetition constraints are the same as those for variable nodes, and these may be viewed as duals to those for a simple parity-check constraint.

We view (20) and (21) as specifying the behavior of ideal "probability gates" that operate much like logic gates, but with soft ("fuzzy") values.

Since $p_0 + p_1 = 1$, binary probability mass functions can be parametrized by a single value. Depending on the parametrization, various probability gate implementations arise. We give four different parametrizations, and derive the VAR and CHK functions for each.

Likelihood Ratio (LR):
Definition: $\lambda(p_0, p_1) = p_0/p_1$.

$$\mathrm{VAR}(\lambda_1, \lambda_2) = \lambda_1 \lambda_2$$
$$\mathrm{CHK}(\lambda_1, \lambda_2) = \frac{1 + \lambda_1 \lambda_2}{\lambda_1 + \lambda_2}.$$

Log-Likelihood Ratio (LLR):
Definition: $\Lambda(p_0, p_1) = \ln(p_0/p_1)$.

$$\begin{aligned}\mathrm{VAR}(\Lambda_1, \Lambda_2) &= \Lambda_1 + \Lambda_2 \\ \mathrm{CHK}(\Lambda_1, \Lambda_2) &= \ln(\cosh((\Lambda_1 + \Lambda_2)/2)) \\ &\quad - \ln(\cosh((\Lambda_1 - \Lambda_2)/2)) \\ &= 2\tanh^{-1}(\tanh(\Lambda_1/2)\tanh,(\Lambda_2/2)).\end{aligned} \quad (22)$$

Likelihood Difference (LD):
Definition: $\delta(p_0, p_1) = p_0 - p_1$.

$$\mathrm{VAR}(\delta_1, \delta_2) = \frac{\delta_1 + \delta_2}{1 + \delta_1 \delta_2}$$
$$\mathrm{CHK}(\delta_1, \delta_2) = \delta_1 \delta_2.$$

Signed Log-Likelihood Difference (SLLD):
Definition: $\Delta(p_0, p_1) = \mathrm{sgn}(p_1 - p_0)\ln|p_1 - p_0|$.

$$\mathrm{VAR}(\Delta_1, \Delta_2) = \begin{cases} s\ln\left(\frac{\cosh((|\Delta_1|+|\Delta_2|)/2)}{\cosh((|\Delta_1|-|\Delta_2|)/2)}\right), \\ \qquad \text{if } \mathrm{sgn}(\Delta_1) = \mathrm{sgn}(\Delta_2) = s \\ s \cdot \mathrm{sgn}(|\Delta_1| - |\Delta_2|) \\ \quad \ln\left(\frac{\sinh((|\Delta_1|+|\Delta_2|)/2)}{\sinh((|\Delta_1|-|\Delta_2|)/2)}\right), \\ \qquad \text{if } \mathrm{sgn}(\Delta_1) = -\mathrm{sgn}(\Delta_2) = -s \end{cases}$$

$$\mathrm{CHK}(\Lambda_1, \Lambda_2) = \mathrm{sgn}(\Delta_1)\mathrm{sgn}(\Delta_2)(|\Delta_1| + |\Delta_2|).$$

In the LLR domain, we observe that for $x \gg 1$

$$\ln(\cosh(x)) \approx |x| - \ln(2).$$

Thus, an approximation to the CHK function (22) is

$$\begin{aligned}\mathrm{CHK}(\Lambda_1, \Lambda_2) &\approx |(\Lambda_1 + \Lambda_2)/2| - |(\Lambda_1 - \Lambda_2)/2| \\ &= \mathrm{sgn}(\Lambda_1)\mathrm{sgn}(\Lambda_2)\min(|\Lambda_1|, |\Lambda_2|)\end{aligned}$$

which turns out to be precisely the min-sum update rule.

By applying the equivalence between factor graphs illustrated in Fig. 19, it is easy to extend these formulas to cases where variable nodes or check nodes have degree larger than three. In particular, we may extend the VAR and CHK functions to more than two arguments via the relations

$$\mathrm{VAR}(x_1, x_2 \ldots, x_n) = \mathrm{VAR}(x_1, \mathrm{VAR}(x_2, \ldots, x_n))$$

$$\mathrm{CHK}(x_1, x_2 \ldots, x_n) = \mathrm{CHK}(x_1, \mathrm{CHK}(x_2, \ldots, x_n)). \quad (23)$$

Of course, there are other alternatives, corresponding to the various binary trees with n leaf vertices. For example, when $n = 4$ we may compute $\mathrm{VAR}(x_1, x_2, x_3, x_4)$ as

$$\mathrm{VAR}(x_1, x_2, x_3, x_4) = \mathrm{VAR}(\mathrm{VAR}(x_1, x_2), \mathrm{VAR}(x_3, x_4))$$

which would have better time complexity in a parallel implementation than a computation based on (23).

VI. Factor-Graph Transformations

In this section we describe a number of straightforward transformations that may be applied to a factor graph in order to

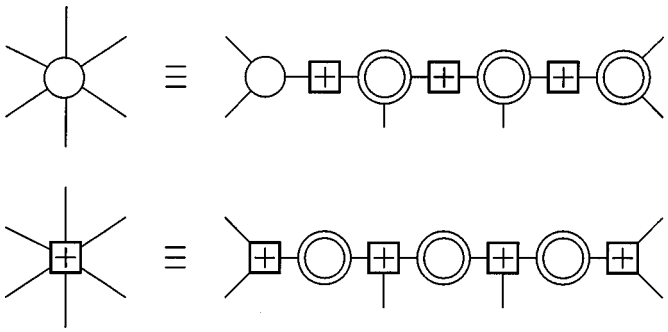

Fig. 19. Transforming variable and check nodes of high degree to multiple nodes of degree three.

modify a factor graph with an inconvenient structure into a more convenient form. For example, it is always possible to transform a factor graph with cycles into a cycle-free factor graph, but at the expense of increasing the complexity of the local functions and/or the domains of the variables. Nevertheless, such transformations can be useful in some cases; for example, at the end of this section we apply them to derive an FFT algorithm from the factor graph representing the DFT kernel. Similar general procedures are described in [17], [20], and in the construction of junction trees in [2].

A. Clustering

It is always possible to cluster nodes of like type—i.e., all variable nodes or all function nodes—without changing the global function being represented by a factor graph. We consider the case of clustering two nodes, but this is easily generalized to larger clusters. If v and w are two nodes being clustered, simply delete v and w and any incident edges from the factor graph, introduce a new node representing the pair (v, w), and connect this new node to nodes that were neighbors of v or w in the original graph.

When v and w are variables with domains A_v and A_w, respectively, the new variable has domain $A_v \times A_w$. Note that the size of this domain is the *product* of the original domain sizes, which can imply a substantial cost increase in computational complexity of the sum-product algorithm. Any function f that had v or w as an argument in the original graph must be converted into an equivalent function f' that has (v, w) as an argument, but this can be accomplished without increasing the complexity of the local functions.

When v and w are local functions, by the pair (v, w) we mean the product of the local functions. If X_v and X_w denote the sets of arguments of v and w, respectively, then $X_v \cup X_w$ is the set of arguments of the product. Pairing functions in this way can imply a substantial cost increase in computational complexity of the sum-product algorithm; however, clustering functions does not increase the complexity of the variables.

Clustering nodes may eliminate cycles in the graph so that the sum-product algorithm in the new graph computes marginal functions exactly. For example, clustering the nodes associated with y and z in the factor graph fragment of Fig. 20(a) and connecting the neighbors of both nodes to the new clustered node, we obtain the factor graph fragment shown in Fig. 20(b). Notice that the local function node f_E connecting y and z in the original factor graph appears with just a single edge in the new factor graph. Also notice that there are two local functions connecting x to (y, z).

The local functions in the new factor graph retain their dependences from the old factor graph. For example, although f_B is connected to x and the pair of variables (y, z), it does not actually depend on z. So, the global function represented by the new factor graph is

$$\begin{aligned} g(\ldots, x, y, z, \ldots) &= \cdots f_A(\ldots, x) f'_B(x, y, z) f'_C(x, y, z) f'_D(\ldots, y, z) \\ &\quad \cdot f'_E(y, z) f'_F(y, z, \ldots) \\ &= \cdots f_A(\ldots, x) f_B(x, y) f_C(x, z) f_D(\ldots, y) f_E(y, z) \\ &\quad \cdot f_F(z, \ldots) \end{aligned}$$

which is identical to the global function represented by the old factor graph.

In Fig. 20(b), there is still one cycle; however, it can be removed by clustering function nodes. In Fig. 20(c), we have clustered the local functions corresponding to f'_B, f'_C, and f'_E

$$f_{BCE}(x, y, z) = f'_B(x, y, z) f'_C(x, y, z) f'_E(y, z). \quad (24)$$

The new global function is

$$\begin{aligned} g(\ldots, x, y, z, \ldots) &= \cdots f_A(\ldots, x) f_{BCE}(x, y, z) f'_D(\ldots, y, z) f'_F(y, z, \ldots), \\ &= \cdots f_A(\ldots, x) f'_B(x, y, z) f'_C(x, y, z) f'_E(y, z) \\ &\quad \cdot f'_D(\ldots, y, z) f'_F(y, z, \ldots) \end{aligned}$$

which is identical to the original global function.

In this case, by clustering variable vertices and function vertices, we have removed the cycles from the factor graph fragment. If the remainder of the graph is cycle-free, then the sum-product algorithm may be used to compute exact marginals. Notice that the sizes of the messages in this region of the graph have increased. For example, y and z have alphabets of size $|A_y|$ and $|A_z|$, respectively, and if functions are represented by a list of their values, the length of the message passed from f_D to (y, z) is equal to the product $|A_y|\,|A_z|$.

B. Stretching Variable Nodes

In the operation of the sum-product algorithm, in the message passed on an edge $\{v, w\}$, local function products are summarized for the variable associated with the edge. Outside of those edges incident on a particular variable node x, any function dependency on x is represented in summary form; i.e., x is marginalized out.

Here we will introduce a factor graph transformation that will extend the region in the graph over which x is represented without being summarized. Let $n_2(x)$ denote the set of nodes that can be reached from x by a path of length two in F. Then $n_2(x)$ is a set of variable nodes, and for any $y \in n_2(x)$, we can pair x and y, i.e., replace y with the pair (x, y), much as in a clustering transformation. The function nodes incident on y would have to be modified as in a clustering transformation, but, as before, this modification does not increase their complexity. We call this a "stretching" transformation, since we imagine node x being "stretched" along the path from x to y.

More generally, we will allow further arbitrary stretching of x. If B is a set of nodes to which x has been stretched, we will

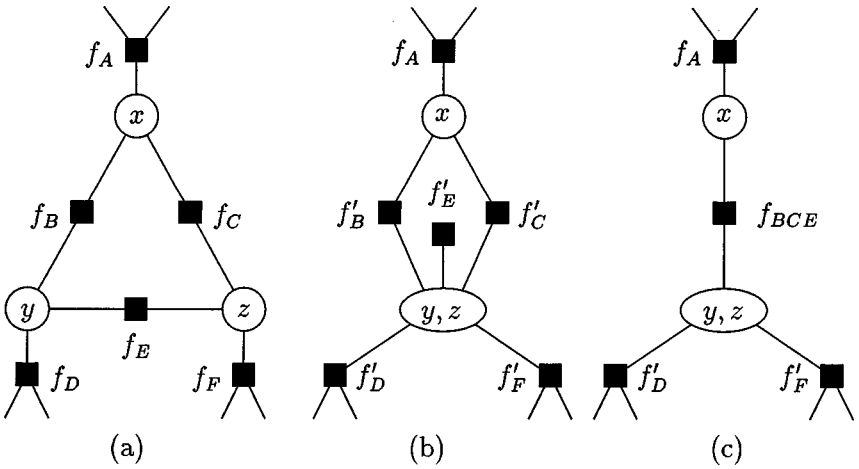

Fig. 20. Clustering transformations. (a) Original factor graph fragment. (b) Variable nodes y and z clustered. (c) Function nodes f_B, f_C, and f_E clustered.

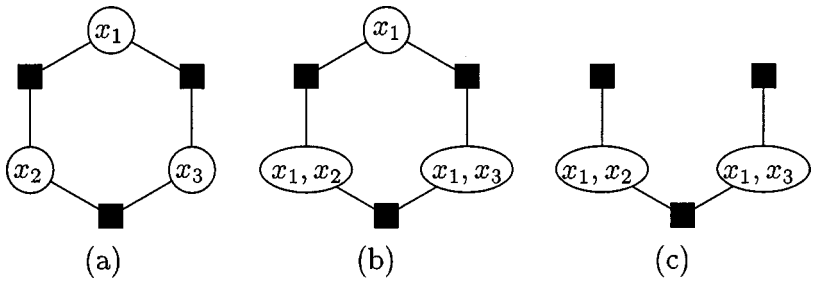

Fig. 21. Stretching transformation. (a) Original factor graph. (b) Node x_1 is stretched to x_2 and x_3. (c) The node representing x_1 alone is now redundant and can be removed.

allow x to be stretched to any element of $n_2(B)$, the set of variable nodes reachable from any node of B by a path of length two. In stretching x in this way, we retain the following basic property: the set of nodes to which x has been paired (together with the connecting function nodes) induces a connected subgraph of the factor graph. This connected subgraph generates a well-defined set of edges over which x is represented without being summarized in the operation of the sum-product algorithm. This stretching leads to precisely the same condition that define junction trees [2]: the subgraph consisting of those vertices whose label includes a particular variable, together with the edges connecting these vertices, is connected.

Fig. 21(a) shows a factor graph, and Fig. 21(b) shows an equivalent factor graph in which x_1 has been stretched to all variable nodes.

When a single variable is stretched in a factor graph, since all variable nodes represent distinct variables, the modified variables that result from a stretching transformation are all distinct. However, if we permit more than one variable to be stretched, this may no longer hold true. For example, in the Markov chain factor graph of Fig. 12(c), if both x_1 and x_4 are stretched to all variables, the result will be a factor graph having two vertices representing the pair (x_1, x_4). The meaning of such a peculiar "factor graph" remains clear, however, since the local functions and hence also the global function are essentially unaffected by the stretching transformations. All that changes is the behavior of the sum-product algorithm, since, in this example, neither x_1 nor x_4 will ever be marginalized out. Hence we will permit the appearance of multiple variable nodes for a single variable whenever they arise as the result of a series of stretching transformations.

Fig. 12(b) illustrates an important motivation for introducing the stretching transformation; it may be possible for an edge, or indeed a variable node, to become *redundant*. Let f be a local function, let e be an edge incident on f, and let X_e be the set of variables (from the original factor graph) associated with e. If X_e is contained in the union of the variable sets associated with the edges incident on f *other than* e, then e is redundant. A redundant edge may be deleted from a factor graph. (Redundant edges must be removed one at a time, because it is possible for an edge to be redundant in the presence of another redundant edge, and become relevant once the latter edge is removed.) If all edges incident on a variable node can be removed, then the variable node itself is redundant and may be deleted.

For example, the node containing x_1 alone is redundant in Fig. 21(b) since each local function neighboring x_1 has a neighbor (other than x_1) to which x_1 has been stretched. Hence this node and the edges incident on it can be removed, as shown in Fig. 21(c). Note that we are not removing the *variable* x_1 from the graph, but rather just a node representing x_1. Here, unlike elsewhere in this paper, the distinction between nodes and variables becomes important.

Let x be a variable node involved in a cycle, i.e., for which there is a nontrivial path P from x to itself. Let $\{y, f\}$, $\{f, x\}$ be the last two edges in P, for some variable node y and some function node f. Let us stretch x along all of the variable nodes involved in P. Then the edge $\{x, f\}$ is redundant and hence can be deleted since both x and (x, y) are incident on f. (Actually,

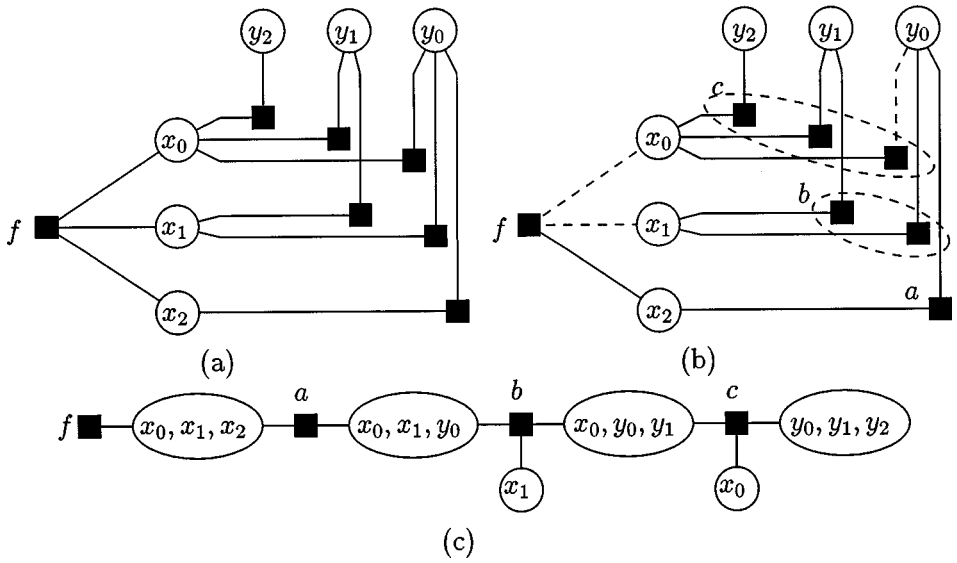

Fig. 22. The DFT. (a) Factor graph. (b) A particular spanning tree. (c) Spanning tree after clustering and stretching transformation.

there is also another redundant edge, corresponding to traveling P in the opposite direction.) In this way, the cycle from x to itself is broken.

By systematically stretching variables around cycles and then deleting a resulting redundant edge to break the cycle, it is possible to use the stretching transformation to break all cycles in the graph, transforming an arbitrary factor graph into an equivalent cycle-free factor graph for which the sum-product algorithm produces exact marginals. This can be done without increasing the complexity of the local functions, but comes at the expense of an (often quite substantial) increase in the complexity of the variable alphabets.

C. Spanning Trees

A spanning tree T for a connected graph G is a connected, cycle-free subgraph of G having the same vertex set as G. Let F be a connected factor graph with a spanning tree T and for every variable node x of F, let $n(x)$ denote the set of function nodes having x as an argument. Since T is a tree, there is a unique path between any two nodes of T, and in particular between x and every element of $n(x)$. Now suppose x is stretched to all variable nodes involved in each path from x to every element of $n(x)$, and let F' be the resulting transformed factor graph.

It turns out that every edge of F' not in T is redundant and all such edges can be deleted from F'. Indeed, if e is an edge of F' not in T, let X_e be the set of variables associated with e, and let f be the local function on which e is incident. For every variable $x \in X_e$, there is a path in T from f to x, and x is stretched to all variable nodes along this path, and in particular is stretched to a neighbor (in T) of f. Since each element of X_e appears in some neighboring variable node not involving e, e is redundant. The removal of e does not affect the redundant status of any other edge of F' not in T, hence all such edges may be deleted from F'.

This observation implies that the sum-product algorithm can be used to compute marginal functions exactly in any spanning tree T of F, provided that each variable x is stretched along all variable nodes appearing in each path from x to a local function having x as an argument. Intuitively, x is not marginalized out in the region of T in which x is "involved."

D. An FFT

An important observation due to Aji and McEliece [1], [2] is that various fast transform algorithms may be developed using a graph-based approach. We now show how we may use the factor-graph transformations of this section to derive an FFT.

The DFT is a widely used tool for the analysis of discrete-time signals. Let $\boldsymbol{w} = (w_0, \ldots, w_{N-1})$ be a complex-valued N-tuple, and let $\Omega = e^{j2\pi/N}$, with $j = \sqrt{-1}$, be a primitive Nth root of unity. The DFT of $\boldsymbol{w}$ is the complex-valued N-tuple $\boldsymbol{W} = (W_0, \ldots, W_{N-1})$ where

$$W_k = \sum_{n=0}^{N-1} w_n \Omega^{-nk}, \qquad k = 0, 1, \ldots, N-1. \tag{25}$$

Consider now the case where N is a power of two, e.g., $N = 8$ for concreteness. We express variables n and k in (25) in binary; more precisely, we let $n = 4x_2 + 2x_1 + x_0$ and let $k = 4y_2 + 2y_1 + y_0$, where x_i and y_i take values from $\{0, 1\}$. We write the DFT kernel, which we take as our global function, in terms of these variables as

$$\begin{aligned} &g(x_0, x_1, x_2, y_0, y_1, y_2) \\ &\quad = w_{4x_2+2x_1+x_0} \Omega^{-(4x_2+2x_1+x_0)(4y_2+2y_1+y_0)} \\ &\quad = f(x_0, x_1, x_2)(-1)^{x_2 y_0}(-1)^{x_1 y_1}(-1)^{x_0 y_2}(j)^{-x_0 y_1} \\ &\qquad \cdot (j)^{-x_1 y_0} \Omega^{-x_0 y_0} \end{aligned}$$

where $f(x_0, x_1, x_2) = w_{4x_2+2x_1+x_0}$ and we have used the relations $\Omega^{16} = \Omega^8 = 1$, $\Omega^4 = -1$, and $\Omega^2 = j$. We see that the DFT kernel factors into a product of local functions as expressed by the factor graph of Fig. 22(a).

We observe that

$$W_k = W_{4y_2+2y_1+y_0} = \sum_{x_0} \sum_{x_1} \sum_{x_2} g(x_0, x_1, x_2, y_0, y_1, y_2) \tag{26}$$

so that the DFT can be viewed as a marginal function, much like a probability mass function. When N is composite, sim-

ilar prime-factor-based decompositions of n and k will result in similar factor graph representations for the DFT kernel.

The factor graph in Fig. 22(a) has cycles. We wish to carry out exact marginalization, so we form a spanning tree. There are many possible spanning trees, of which one is shown in Fig. 22(b). (Different choices for the spanning tree will lead to possibly different DFT algorithms when the sum-product algorithm is applied.) If we cluster the local functions as shown in Fig. 22(b), essentially by defining

$$\begin{aligned} a(x_2, y_0) &= (-1)^{x_0 y_0} \\ b(x_1, y_0, y_1) &= (-1)^{x_1 y_1} (-j)^{x_1 y_0} \\ c(x_0, y_0, y_1, y_2) &= (-1)^{x_0 y_2} (-j)^{x_0 y_1} \Omega^{x_0 y_0} \end{aligned}$$

then we arrive at the spanning tree shown in Fig. 22(c). The variables that result from the required stretching transformation are shown. Although they are redundant, we have included variable nodes x_0 and x_1. Observe that each message sent from left to right is a function of three binary variables, which can be represented as a list of eight complex quantities. Along the path from f to (y_0, y_1, y_2), first x_2, then x_1, and then x_0 are marginalized out as y_0, y_1, and y_2 are added to the argument list of the functions. In three steps, the function w_n is converted to the function W_k. Clearly, we have obtained an FFT as an instance of the sum-product algorithm.

VII. Conclusion

Factor graphs provide a natural graphical description of the factorization of a global function into a product of local functions. Factor graphs can be applied in a wide range of application areas, as we have illustrated with a large number of examples.

A major aim of this paper was to demonstrate that a single algorithm—the sum-product algorithm—based on only a single conceptually simple computational rule, can encompass an enormous variety of practical algorithms. As we have seen, these include the forward/backward algorithm, the Viterbi algorithm, Pearl's belief propagation algorithm, the iterative turbo decoding algorithm, the Kalman filter, and even certain FFT algorithms! Various extensions of these algorithms—for example, a Kalman filter operating on a tree-structured system—although not treated here, can be derived in a straightforward manner by applying the principles enunciated in this paper.

We have emphasized that the sum-product algorithm may be applied to arbitrary factor graphs, cycle-free or not. In the cycle-free finite case, we have shown that the sum-product algorithm may be used to compute function summaries *exactly*. In some applications, e.g., in processing Markov chains and hidden Markov models, the underlying factor graph is naturally cycle-free, while in other applications, e.g., in decoding of LDPC codes and turbo codes, it is not. In the latter case, a successful strategy has been simply to apply the sum-product algorithm without regard to the cycles. Nevertheless, in some cases it might be important to obtain an equivalent cycle-free representation, and we have given a number of graph transformations that can be used to achieve such representations.

Factor graphs afford great flexibility in modeling systems. Both Willems' behavioral approach to systems and the traditional input/output or state-space approaches fit naturally in the factor graph framework. The generality of allowing arbitrary functions (not just probability distributions or characteristic functions) to be represented further enhances the flexibility of factor graphs.

Factor graphs also have the potential to unify modeling and signal processing tasks that are often treated separately in current systems. In communication systems, for example, channel modeling and estimation, separation of multiple users, and decoding can be treated in a unified way using a single graphical model that represents the interactions of these various elements, as suggested by Wiberg [31]. We believe that the full potential of this approach has not yet been realized, and we suggest that further exploration of the modeling power of factor graphs and applications of the sum-product algorithm will prove to be fruitful.

Appendix A
From Factor Trees to Expression Trees

Let $g(x, x_1, \ldots, x_{N-1})$ be a function that can be represented by a cycle-free connected factor graph, i.e., a *factor tree* T. We are interested in developing an expression for

$$\sum_{\sim\{x\}} g(x, x_1, \ldots, x_{N-1})$$

i.e., the summary for x of g. We consider x to be the root of T, so that all other vertices of T are *descendants* of x.

Assuming that x has K neighbors in T, then without loss of generality g may be written in the form

$$g(x, x_1, \ldots, x_{N-1}) = \prod_{i=1}^{K} F_i(x, X_i)$$

where $F_i(x, X_i)$ is the product of all local functions in the subtree of T that have the ith neighbor of x as root, and X_i is the set of variables in that subtree. Since T is a tree, for $i \neq j$, $X_i \cap X_j = \emptyset$ and $X_1 \cup \cdots \cup X_K = \{x_1, \ldots, x_{N-1}\}$, i.e., $X_1, \ldots, X_K$ is a partition of $\{x_1, \ldots, x_{N-1}\}$. This decomposition is represented by the generic factor tree of Fig. 23, in which $F_1(x, X_1)$ is shown in expanded form.

Now, by the distributive law, and using the fact that $X_1, \ldots, X_K$ are pairwise disjoint, we obtain

$$\begin{aligned} &\sum_{\sim\{x\}} g(x, x_1, \ldots, x_{N-1}) \\ &\quad = \sum_{X_1} \sum_{X_2} \cdots \sum_{X_K} F_1(x, X_1) F_2(x, X_2) \ldots F_K(x, X_K) \\ &\quad = \left(\sum_{X_1} f(x, X_1) \right) \left(\sum_{X_2} f(x, X_2) \right) \cdots \left(\sum_{X_K} f(x, X_K) \right) \\ &\quad = \prod_{i=1}^{K} \sum_{\sim\{x\}} F_i(x, X_i) \end{aligned}$$

i.e., the summary for x of g is the *product* of the summaries for x of the F_i functions.

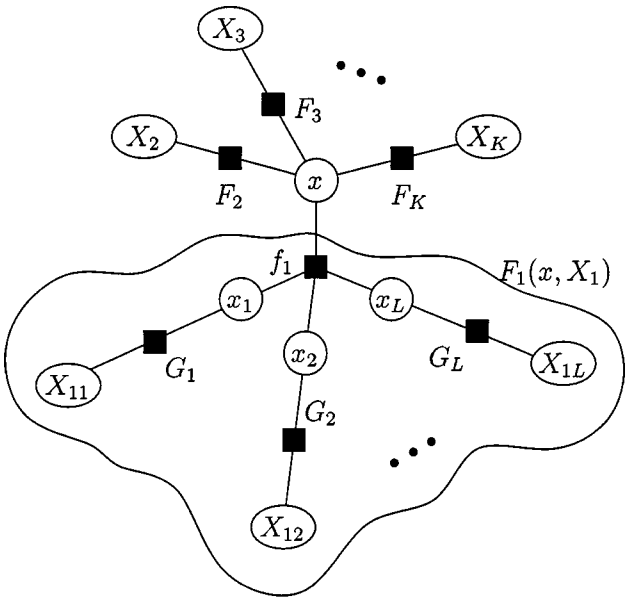

Fig. 23. A generic factor tree.

Consider the case $i = 1$. To compute the summary for x of F_1, observe that, without loss of generality, $F_1(x, X_1)$ can be written as

$$F_1(x, X_1) = f_1(x, x_1, \ldots, x_L)G_1(x_1, X_{11}) \cdot G_2(x_2, X_{12}) \cdots G_L(x_L, X_{1L})$$

where, for convenience, we have numbered the arguments of g so that $f_1(x, x_1, \ldots, x_L)$ is the first neighbor of x. This decomposition is illustrated in Fig. 23. We note that $\{x_1, \ldots, x_L\}, X_{11}, \ldots, X_{1L}$ is a partition of X_1. Again, using the fact that these sets are pairwise-disjoint and applying the distributive law, we obtain

$$\begin{aligned}
&\sum_{\sim\{x\}} F_1(x, X_1) \\
&\quad = \sum_{\sim\{x\}} f_1(x, x_1, \ldots, x_L)G_1(x_1, X_{11}) \ldots G_L(x_L, X_{1L}) \\
&\quad = \sum_{x_1, \ldots, x_L} f_1(x, x_1, \ldots, x_L) \left(\sum_{X_{11}} G_1(x_1, X_{11}) \right) \cdots \\
&\qquad \left(\sum_{X_{1L}} G_L(x_L, X_{1L}) \right) \\
&\quad = \sum_{\sim\{x\}} \left(f_1(x, x_1, \ldots, x_L) \prod_{i=1}^{L} \sum_{\sim\{x_i\}} G(x_i, X_{1i}) \right).
\end{aligned}$$

In words, we see that if $f_1(x, x_1, \ldots, x_L)$ is a neighbor of x, to compute the summary for x of the product of the local functions in the subtree of T descending from f_1, we should do the following:

1) for each neighbor x_i of f_1 (other than x), compute the summary for x_i of the product of the functions in the subtree descending from x_i;
2) form the product of these summaries with f_1, summarizing the result for x.

The problem of computing the summary for x_i of the product of the local subtree descending from x_i is a problem of the same general form with which we began, and so the same general approach can be applied recursively. The result of this recursion justifies the transformation of the factor tree for g with root vertex x into an expression tree for $\sum_{\sim\{x\}} g(x, x_1, \ldots, x_{N-1})$, as illustrated in Fig. 5.

Appendix B
Other Graphical Models for Probability Distributions

Factor graphs are by no means the first graph-based language for describing probability distributions. In the next two examples, we describe very briefly the close relationship between factor graphs and models based on undirected graphs (Markov random fields) and models based on directed acyclic graphs (Bayesian networks).

A. Markov Random Fields

A Markov random field (see, e.g., [18]) is a graphical model based on an undirected graph $G = (V, E)$ in which each node corresponds to a random variable. The graph G is a *Markov random field* (MRF) if the distribution $p(v_1, \ldots, v_n)$ satisfies the local Markov property

$$(\forall\, v \in V) \qquad p(v|V \setminus \{v\}) = p(v|n(v)) \tag{27}$$

where $n(v)$ denotes the set of neighbors of v. In words, G is an MRF if every variable v is independent of nonneighboring variables in the graph, given the values of its immediate neighbors. MRFs are well developed in statistics, and have been used in a variety of applications (see, e.g., [18], [26], [16], [15]).

A *clique* in a graph is a collection of vertices which are all pairwise neighbors. Under fairly general conditions (e.g., positivity of the joint probability density is sufficient), the joint probability mass function of an MRF may be expressed as the product of a collection of Gibbs potential functions, defined on the set Q of cliques in the MRF, i.e.

$$p(v_1, v_2, \ldots, v_N) = Z^{-1} \prod_{E \in Q} f_E(V_E) \tag{28}$$

where Z^{-1} is a normalizing constant, and each $E \in Q$ is a clique. For example (cf. Fig. 1), the MRF in Fig. 24(a) may be used to express the factorization

$$\begin{aligned}
&p(v_1, v_2, v_3, v_4, v_5) \\
&\qquad = Z^{-1} f_C(v_1, v_2, v_3) f_D(v_3, v_4) f_E(v_4, v_5).
\end{aligned}$$

Clearly, (28) has precisely the structure needed for a factor graph representation. Indeed, a factor graph representation may be preferable to an MRF in expressing such a factorization, since distinct factorizations, i.e., factorizations with different Q's in (28), may yield precisely the *same* underlying MRF graph, whereas they will always yield distinct factor graphs. (An example in a coding context of this MRF ambiguity is given in [19].)

B. Bayesian Networks

Bayesian networks (see, e.g., [25], [17], [10]) are graphical models for a collection of random variables that are based on

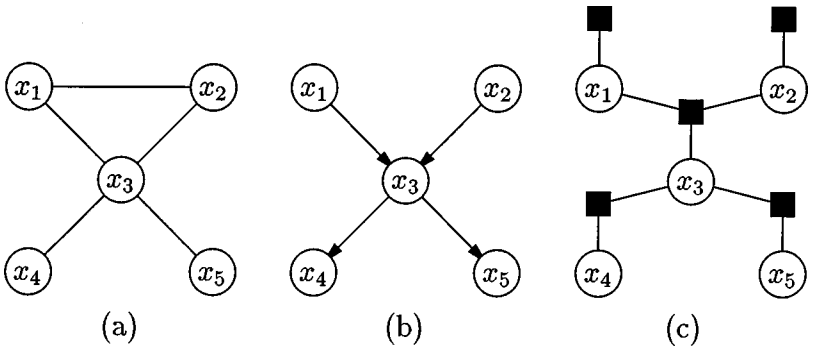

Fig. 24. Graphical probability models. (a) A Markov random field. (b) A Bayesian network. (c) A factor graph.

directed acyclic graphs (DAGs). Bayesian networks, combined with Pearl's "belief propagation algorithm" [25], have become an important tool in expert systems. The first to connect Bayesian networks and belief propagation with applications in coding theory were MacKay and Neal [21]; more recently, [19], [24] develop a view of the "turbo decoding" algorithm [5] as an instance of probability propagation in a Bayesian network model of a code.

Each node v in a Bayesian network is associated with a random variable. Denoting by $\boldsymbol{a}(v)$ the set of *parents* of v (i.e., the set of vertices *from* which an edge is incident on v), by definition, the distribution represented by the Bayesian network may be written as

$$p(v_1, v_2, \ldots, v_n) = \prod_{i=1}^{n} p(v_i|\boldsymbol{a}(v_i)). \tag{29}$$

If $\boldsymbol{a}(v_i) = \varnothing$ (i.e., v_i has no parents), we take $p(v_i|\varnothing) = p(v_i)$. For example (cf. (2)) Fig. 24(b) shows a Bayesian network that expresses the factorization

$$p(v_1, v_2, v_3, v_4, v_5) = p(v_1)p(v_2)p(v_3|v_1, v_2)p(v_4|v_3)p(v_5|v_3). \tag{30}$$

Again, as with Markov random fields, Bayesian networks express a factorization of a joint probability distribution that is suitable for representation by a factor graph. The factor graph corresponding to (30) is shown in Fig. 24(c); cf. Fig. 1.

It is a straightforward exercise to translate the update rules that govern the operation of the sum-product algorithm to Pearl's belief propagation rules [25], [17]. To convert a Bayesian network into a factor graph: simply introduce a function node for each factor $p(v_i|\boldsymbol{a}(v_i))$ in (29) and draw edges from this node to v_i and its parents $\boldsymbol{a}(v_i)$. An example conversion from a Bayesian network to a factor graph is shown in Fig. 24(c).

Equations similar to Pearl's belief updating and bottom-up/top-down propagation rules [25, pp. 182–183] may be derived from the general sum-product algorithm update equations (5) and (6) as follows.

In belief propagation, messages are sent between "variable nodes," corresponding to the dashed ellipses for the particular Bayesian network shown in Fig. 25. In a Bayesian network, if an edge is directed from vertex p to vertex c, then p is a parent of c

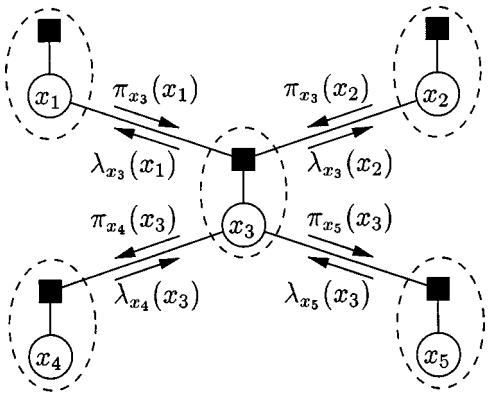

Fig. 25. Messages sent in belief propagation.

and c is a child of p. Messages sent between variables are always functions of the parent p. In [25], a message sent from p to c is denoted $\pi_c(p)$, while a message sent from c to p is denoted as $\lambda_c(p)$, as shown in Fig. 25 for the specific Bayesian network of Fig. 24(c).

Consider the central variable x_3 in Fig. 25. Clearly, the message sent upwards by the sum-product algorithm to the local function f contained in the ellipse is, from (5), given by the product of the incoming λ messages, i.e.

$$\mu_{x_3 \to f}(x_3) = \lambda_{x_4}(x_3)\lambda_{x_5}(x_3).$$

The message sent from f to x_1 is, according to (6), the product of f with the other messages received at f summarized for x_1. Note that this local function is the conditional probability mass function $f(x_3|x_1, x_2)$; hence

$$\begin{aligned}\lambda_{x_3}(x_1) &= \sum_{\sim\{x_1\}} (\lambda_{x_4}(x_3)\lambda_{x_5}(x_3)f(x_3|x_1, x_2)\pi_{x_3}(x_2)) \\ &= \sum_{x_3} \lambda_{x_4}(x_3)\lambda_{x_5}(x_3) \sum_{x_2} f(x_3|x_1, x_2)\pi_{x_3}(x_2).\end{aligned}$$

Similarly, the message $\pi_{x_4}(x_3)$ sent from x_3 to the ellipse containing x_4 is given by

$$\begin{aligned}\pi_{x_4}(x_3) &= \lambda_{x_5}(x_3) \sum_{\sim\{x_3\}} (f(x_3|x_1, x_2)\pi_{x_3}(x_1)\pi_{x_3}(x_2)) \\ &= \lambda_{x_5}(x_3) \sum_{x_1} \sum_{x_2} f(x_3|x_1, x_2)\pi_{x_3}(x_1)\pi_{x_3}(x_2).\end{aligned}$$

In general, let us denote the set of parents of a variable x by $\boldsymbol{a}(x)$, and the set of children of x by $\boldsymbol{d}(x)$. We will have, for every $a \in \boldsymbol{a}(x)$

$$\lambda_x(a) = \sum_{\sim\{a\}} \left(\prod_{d \in \boldsymbol{d}(x)} \lambda_d(x) f(x|\boldsymbol{a}(x)) \prod_{p \in \boldsymbol{a}(x)\setminus\{a\}} \pi_x(p) \right) \tag{31}$$

and, for every $d \in \boldsymbol{d}(x)$

$$\pi_d(x) = \prod_{c \in \boldsymbol{d}(x)\setminus\{d\}} \lambda_c(x) \sum_{\sim\{x\}} \left(f(x|\boldsymbol{a}(x)) \prod_{a \in \boldsymbol{a}(x)} \pi_x(a) \right). \tag{32}$$

The termination condition for cycle-free graphs, called the "belief update" equation in [25], is given by the product of the messages received by x in the factor graph

$$\text{BEL}\,(x) = \prod_{d \in \boldsymbol{d}(x)} \lambda_d(x) \sum_{\sim\{x\}} \left(f(x|\boldsymbol{a}(x)) \prod_{a \in \boldsymbol{a}(x)} \pi_x(a) \right). \tag{33}$$

Pearl also introduces a scale factor in (32) and (33) so that the resulting messages properly represent probability mass functions. The relative complexity of (31)–(33) compared with the simplicity of the sum-product update rule given in Section II provides a strong pedagogical incentive for the introduction of factor graphs.

ACKNOWLEDGMENT

The concept of factor graphs as a generalization of Tanner graphs was devised by a group at ISIT '97 in Ulm, Germany, that included the authors, G. D. Forney, Jr., R. Kötter, D. J. C. MacKay, R. J. McEliece, R. M. Tanner, and N. Wiberg. The authors benefitted greatly from the many discussions on this topic that took place in Ulm. They wish to thank G. D. Forney, Jr., and the referees for many helpful comments on earlier versions of this paper.

The work of F. R. Kschischang took place in part while on sabbatical leave at the Massachusetts Institute of Technology (MIT). He gratefully acknowledges the support and hospitality of Prof. G. W. Wornell of MIT. H.-A. Loeliger performed his work while with Endora Tech AG, Basel, Switzerland. He wishes to acknowledge the support of F. Tarköy.

REFERENCES

[1] S. M. Aji and R. J. McEliece, "A general algorithm for distributing information on a graph," in *Proc. 1997 IEEE Int. Symp. Information Theory*, Ulm, Germany, July 1997, p. 6.

[2] ——, "The generalized distributive law," *IEEE Trans. Inform. Theory*, vol. 46, pp. 325–343, Mar. 2000.

[3] B. D. O. Anderson and J. B. Moore, *Optimal Filtering*. Englewood Cliffs, NJ: Prentice-Hall, 1979.

[4] L. R. Bahl, J. Cocke, F. Jelinek, and J. Raviv, "Optimal decoding of linear codes for minimizing symbol error rate," *IEEE Trans. Inform. Theory*, vol. IT-20, pp. 284–287, Mar. 1974.

[5] C. Berrou, A. Glavieux, and P. Thitimajshima, "Near Shannonlimit error-correcting coding and decoding: Turbo codes," in *Proc. 1993 IEEE Int. Conf. Communications*, Geneva, Switzerland, May 1993, pp. 1064–1070.

[6] D. Divsalar, H. Jin, and R. J. McEliece, "Coding theorems for 'turbo-like' codes," in *Proc. 36th Allerton Conf. Communications, Control, and Computing*, Urbana, IL, Sept. 23–25, 1998, pp. 201–210.

[7] G. D. Forney Jr., "On iterative decoding and the two-way algorithm," in *Proc. Int. Symp. Turbo Codes and Related Topics*, Brest, France, Sept. 1997.

[8] ——, "Codes on graphs: Normal realizations," *IEEE Trans. Inform. Theory*, vol. 47, pp. 520–548, Feb. 2001.

[9] B. J. Frey and F. R. Kschischang, "Probability propagation and iterative decoding," in *Proc. 34th Annu. Allerton Conf. Communication, Control, and Computing*, Monticello, IL, Oct. 1–4, 1996.

[10] B. J. Frey, *Graphical Models for Machine Learning and Digital Communication*. Cambridge, MA: MIT Press, 1998.

[11] R. G. Gallager, *Low-Density Parity-Check Codes*. Cambridge, MA: MIT Press, 1963.

[12] R. Garello, G. Montorsi, S. Benedetto, and G. Cancellieri, "Interleaver properties and their applications to the trellis complexity analysis of turbo codes," *IEEE Trans. Commun.*, to be published.

[13] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York: Freeman, 1979.

[14] R. L. Graham, D. E. Knuth, and O. Patashnik, *Concrete Mathematics*. Reading, MA: Addison-Wesley, 1989.

[15] G. E. Hinton and T. J. Sejnowski, "Learning and relearning in Boltzmann machines," in *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, D. E. Rumelhart and J. L. McClelland, Eds. Cambridge, MA: MIT Press, 1986, pp. 282–317.

[16] V. Isham, "An introduction to spatial point processes and Markov random fields," *Int. Stat. Rev.*, vol. 49, pp. 21–43, 1981.

[17] F. V. Jensen, *An Introduction to Bayesian Networks*. New York: Springer-Verlag, 1996.

[18] R. Kindermann and J. L. Snell, *Markov Random Fields and Their Applications*. Providence, RI: Amer. Math. Soc., 1980.

[19] F. R. Kschischang and B. J. Frey, "Iterative decoding of compound codes by probability propagation in graphical models," *IEEE J. Select. Areas Commun.*, vol. 16, pp. 219–230, Feb. 1998.

[20] S. L. Lauritzen and D. J. Spiegelhalter, "Local computations with probabilities on graphical structures and their application to expert systems," *J. Roy. Statist. Soc.*, ser. B, vol. 50, pp. 157–224, 1988.

[21] D. J. C. MacKay and R. M. Neal, "Good codes based on very sparse matrices," in *Cryptography and Coding. 5th IMA Conference (Lecture Notes in Computer Science)*, C. Boyd, Ed. Berlin, Germany: Springer, 1995, vol. 1025, pp. 100–111.

[22] D. J. C. MacKay, "Good error-correcting codes based on very sparse matrices," *IEEE Trans. Inform. Theory*, vol. 45, pp. 399–431, Mar. 1999.

[23] P. S. Maybeck, *Stochastic Models, Estimation, and Control*. New York: Academic, 1979.

[24] R. J. McEliece, D. J. C. MacKay, and J.-F. Cheng, "Turbo decoding as an instance of Pearl's 'belief propagation' algorithm," *IEEE J. Select. Areas Commun.*, vol. 16, pp. 140–152, Feb. 1998.

[25] J. Pearl, *Probabilistic Reasoning in Intelligent Systems*, 2nd ed. San Francisco, CA: Kaufmann, 1988.

[26] C. J. Preston, *Gibbs States on Countable Sets*. Cambridge, U.K.: Cambridge Univ. Press, 1974.

[27] L. Rabiner, "A tutorial on hidden Markov models and selected applications in speech recognition," *Proc. EEE*, vol. 77, pp. 257–286, Feb. 1989.

[28] K. H. Rosen, *Discrete Mathematics and its Applications*, 4th ed. New York: WCB/McGraw-Hill, 1999.

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

[30] A. Vardy, "Trellis structure of codes," in *Handbook of Coding Theory*, V. S. Pless and W. C. Huffman, Eds. Amsterdam, The Netherlands: Elsevier, 1998, vol. 2.

[31] N. Wiberg, "Codes and decoding on general graphs," Ph.D. dissertation, Linköping Univ., Linköping, Sweden, 1996.

[32] N. Wiberg, H.-A. Loeliger, and R. Kötter, "Codes and iterative decoding on general graphs," *Eur. Trans. Telecomm.*, vol. 6, pp. 513–525, Sept./Oct. 1995.

[33] J. C. Willems, "Models for Dynamics," in *Dynamics Reported, Volume 2*, U. Kirchgraber and H. O. Walther, Eds. New York: Wiley, 1989, pp. 171–269.