Factor Graphs and the Sum-Product Algorithm

Frank R. Kschischang*
Brendan J. Frey†
Hans-Andrea Loeliger‡

June 18, 2000

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. 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 algorithms.

Keywords—Graphical models, factor graphs, Tanner graphs, sum-product algorithm, marginalization, forward/backward algorithm, Viterbi algorithm, iterative decoding, belief propagation, Kalman filtering, fast Fourier transform.

Submitted to *IEEE Transactions on Information Theory*, July, 1998; revised, June, 2000. This paper can be downloaded from `http://www.comm.utoronto.ca/frank/factor/`.

---

*Department of Electrical & Computer Engineering, University of Toronto, Toronto, Ontario M5S 3G4, CANADA (`frank@comm.utoronto.ca`)

†Department of Computer Science, University of Waterloo, Waterloo, Ontario N2L 3G1, CANADA (`frey@.uwaterloo.ca`)

‡ISI, ETH Zürich, Sternwartstrasse 7, 8092 Zürich, SWITZERLAND (`haloeliger@endora.ch`)

Typeset with LaTeX at 19:58 on June 18, 2000.

1 Introduction

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

The aim of this tutorial is to introduce factor graphs and to describe a generic message-passing algorithm called the *sum-product algorithm*, that 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 the artificial intelligence, signal processing, and digital communications communities can 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 certain graph-theoretic descriptions for error-correcting codes; in particular, the TWL graphs of [42, 43], which in turn are descendants of the Tanner graphs of [39]. Tanner introduced a bipartite graph framework to describe families of codes that are generalizations of the low-density parity-check codes of Gallager [15], and also described the sum-product algorithm in this setting. In Tanner's original formulation, all variables are codeword symbols and hence "visible," i.e., there are no hidden (latent) state variables. Later, and essentially independently of Tanner's work, Wiberg, Loeliger and Koetter broadened this bipartite graph framework by allowing hidden variables [42, 43]. 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 3.1), a Tanner graph (and by extension a TWL graph) for a code represents a particular factorization of the characteristic 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 significant paper [2], Aji and McEliece develop the "generalized distributive law" (GDL) that in some cases solves the MPF problem using a "junction tree" representation of the global function. Factor graphs can 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 can 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.

There are also close connections between factor graphs and graphical representations (graphical models) for multidimensional probability distributions such as Markov random fields [21, 23, 35] and Bayesian (belief) networks [34, 22]. 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 [34] that 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 low-density parity check codes [12, 24, 27, 28, 33], the most powerful practically-decodable codes known, and hence these algorithms, too, can be viewed as instances of the sum-product algorithm.

We begin the paper in Section 2 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 this structure also encodes *expressions* for computing the various marginal functions associated with the given function, and these expressions lead directly to the sum-product algorithm.

In section 3, we show how factor graphs can 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 explored in Appendix B, where we recover Pearl's belief propagation algorithm as an instance of the sum-product algorithm.

In Section 4, we apply the sum-product algorithm to trellis-structured models, and obtain the forward/backward algorithm, the Viterbi algorithm, and the Kalman filter as instances of the sum-product algorithm. In Section 5, we apply the sum-product algorithm to factor graphs with cycles, and obtain the iterative algorithms used to decode turbo-like codes as instances of the sum-product algorithm.

In Section 6, we describe several generic transformations by which a factor graph with cycles might be converted—often at great expense in complexity—to an equivalent cycle-free form. In Appendix C, we apply these ideas to the factor graph representing the DFT kernel, and derive a Fast Fourier Transform algorithm as an instance of the sum-product algorithm.

Some concluding remarks are given in Section 7.

2 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* A_i. Let $g(x_1, \ldots, x_n)$ be an R-valued function of these variables, i.e., a function with domain

$$S = \prod_{i=1}^{n} A_i,$$

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 can in general be any semiring [2]; however, at least initially, we will lose nothing essential by assuming that R is the real numbers.

Assuming that summation of values 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 slightly eccentric summation notation to handle it. Instead of indicating the variables being summed over, in a "not-sum" or *summary*, 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

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

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) re-uses intermediate values, i.e., partial sums, that arise. As we will see, such procedures can be expressed very naturally using a factor graph.

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

$$g(x_1, \ldots, x_n) = \prod_{X \in Q} f_X(X) \tag{1}$$

where Q is a collection of subsets of $\{x_1, \ldots, x_n\}$ and $f_X(X)$ is a function having the elements of X 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_X, and an edge connecting variable node x_i to factor node f_X if and only if x_i is an argument of f, i.e., $x_i \in X$.

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

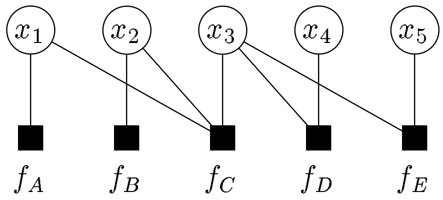


Figure 1: A factor graph for the product $f_A(x_1)f_B(x_2)f_C(x_1, x_2, x_3)f_D(x_3, x_4)f_E(x_3, x_5)$.

▶ **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

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

of five factors. The factor graph that corresponds to (2) is shown in Fig. 1. ◀

2.1 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 can write $g_1(x_1)$ as

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

or, in summary notation,

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

Similarly, we find that

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

In computer science, arithmetic expressions like the right-hand sides of (3) and (4) are often represented by ordered rooted trees [37, Sect. 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 $x(y+z)$.

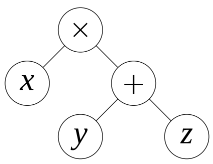
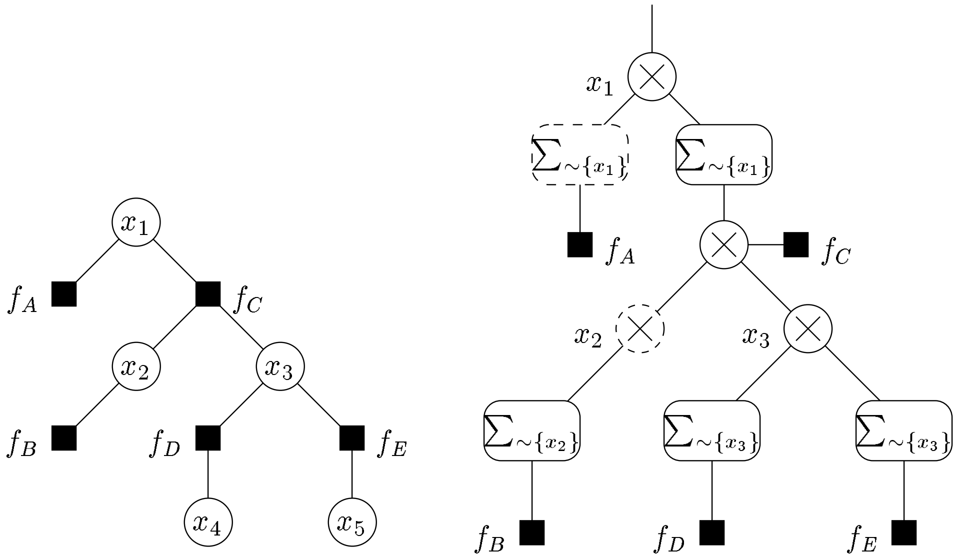


Figure 2: An expression tree representing $x \times (y+z)$.

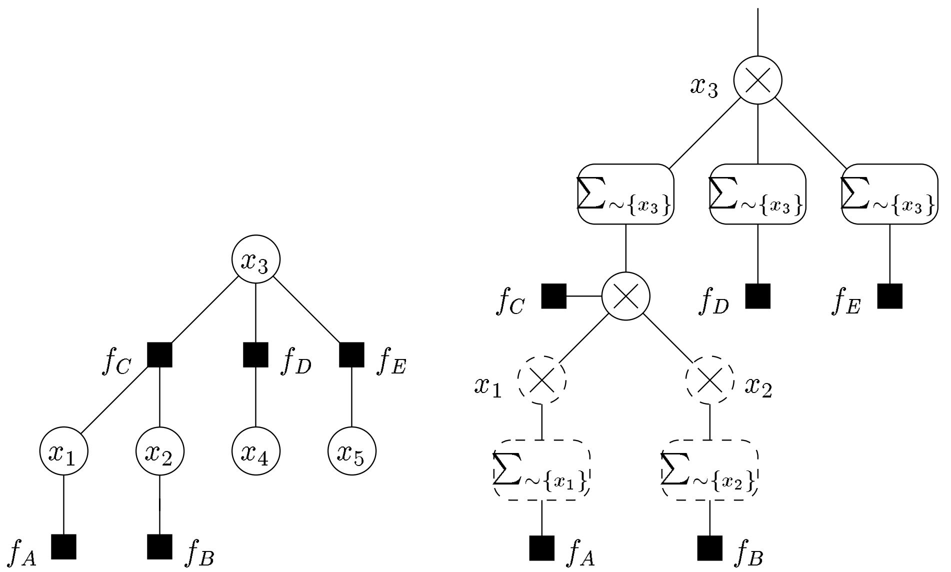
When the operators in an expression tree are restricted to those that are completely symmetric in their operands (e.g., multiplication and addition), it is unnecessary to order the vertices to avoid ambiguity in interpreting the expression represented by the tree. Thus, for example, Fig. 3(b) unambiguously represents the expression on the right-hand side of (3), and Fig. 4(b) unambiguously represents the expression on the right-hand side of (4). The operators shown in these figures are the function product and the summary; operands are local functions or sub-expressions involving these operators.

Shown in Figs 3(a) and 4(a) is the factor graph of Fig. 1, redrawn as a rooted tree with x_1 and x_3, respectively, as root vertex. This is possible because the global function defined in (2) was carefully chosen so that the corresponding factor graph is a tree. Comparing the factor graph with the corresponding tree representing the expression for the marginal function, it is easy to note a correspondence between the two. This observation, though simple, is 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 arithmetic expressions by which the marginal functions associated with the global function may be computed.*



(a) (b)

Figure 3: (a) The factor graph of Fig. 1, redrawn as a rooted tree with x_1 as root. (b) A tree representation for the expression representing the marginal function for x_1.



(a) (b)

Figure 4: (a) The factor graph of Fig. 1, redrawn as a rooted tree with x_3 as root. (b) A tree representation for the expression representing the marginal function for x_3.

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 $\sum_{\sim\{x\}}$ 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 fewer than two operands) 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(a) yields the expression tree of Fig. 3(b), and similarly for Fig. 4. Trivial operations are indicated with dashed lines in these figures.

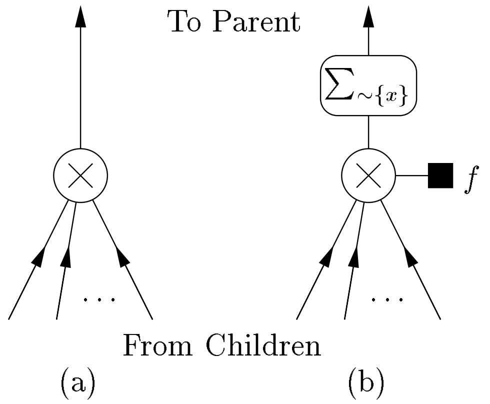


Figure 5: Local substitutions that transform a rooted cycle-free factor graph to an expression tree for a marginal function: (a) at a variable node; (b) at a factor node.

2.2 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, as will be preferable for us, one could describe the algorithm as a "bottom-up" procedure that begins at the leaves of the tree, with each operator vertex combining its operands, and passing the result as an operand for its parent. For example, $x(y + z)$, represented by the expression tree of Fig. 2, might be evaluated starting at the leaf nodes y and z, evaluating $y + z$, and passing 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. The description of such algorithms is greatly elucidated by imagining a processor associated with each vertex of the factor graph, with factor graph edges representing channels by which these processors may communicate. For us, "messages" sent between processors are always simply some appropriate description of a function. (We describe some useful representations in Section 5.5.) By a "product of messages" we mean an appropriate description of the (pointwise) product of the corresponding functions, not (necessarily) literally the numerical product of the messages. Similarly, the summary operator is applied to the functions, not necessarily literally to the messages themselves.

We describe now a message-passing algorithm that we 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 $\sum_{\sim\{x\}}$ operator. The algorithm terminates at the root node x_i, where the $g_i(x_i)$ is obtained as the product of all messages received at x_i.

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.

2.3 Computing all Marginal Functions at Once

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 can be efficiently accomplished at once, essentially by "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.

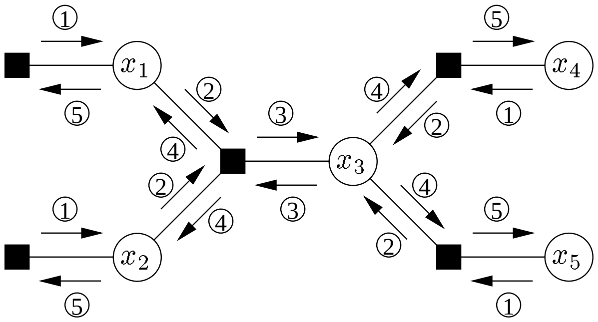


Figure 6: Messages generated in each (circled) step of the sum-product algorithm.

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.

Fig. 6 shows the flow of messages that would be generated by the sum-product algorithm applied to the factor graph of Fig. 1. The messages can be generated in five steps, as indicated with circles in Fig. 6. In detail, the messages are generated as follows.

Step 1:

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

Step 2:

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

Step 3:

$$\begin{aligned}
\mu_{f_C\to x_3}(x_3) &= \sum_{\sim\{x_3\}} \mu_{x_1\to f_C}(x_1)\mu_{x_2\to f_C}(x_2) f_C(x_1,x_2,x_3)\\
\mu_{x_3\to f_C}(x_3) &= \mu_{f_D\to x_3}(x_3)\mu_{f_E\to x_3}(x_3)
\end{aligned}$$

Step 4:

$$\begin{aligned}
\mu_{f_C\to x_1}(x_1) &= \sum_{\sim\{x_1\}} \mu_{x_3\to f_C}(x_3)\mu_{x_2\to f_C}(x_2) f_C(x_1,x_2,x_3)\\
\mu_{f_C\to x_2}(x_2) &= \sum_{\sim\{x_2\}} \mu_{x_3\to f_C}(x_3)\mu_{x_1\to f_C}(x_1) f_C(x_1,x_2,x_3)\\
\mu_{x_3\to f_D}(x_3) &= \mu_{f_C\to x_3}(x_3)\mu_{f_E\to x_3}(x_3)\\
\mu_{x_3\to f_E}(x_3) &= \mu_{f_C\to x_3}(x_3)\mu_{f_D\to x_3}(x_3)
\end{aligned}$$

Step 5:

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

Termination:

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

In the termination step, we have computed $g_i(x_i)$ as the product of all messages directed towards x_i. Equivalently, since the message passed on any given edge is equal to the product of all but one of these messages, we could 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 can compute $g_3(x_3)$ in three other ways:

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

Each edge e in a factor graph is of the form $\{x, f\}$, where x is a variable node and f is a factor node. The variable associated with e is x. 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.

According to this rule, the message sent on an edge $e = \{x, f\}$ is always a function of the variable x associated with that edge. 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. 7, the message computations performed by the sum-product algorithm can 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) since there is no local function to include, and the summary for x of a product of functions of x is 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.

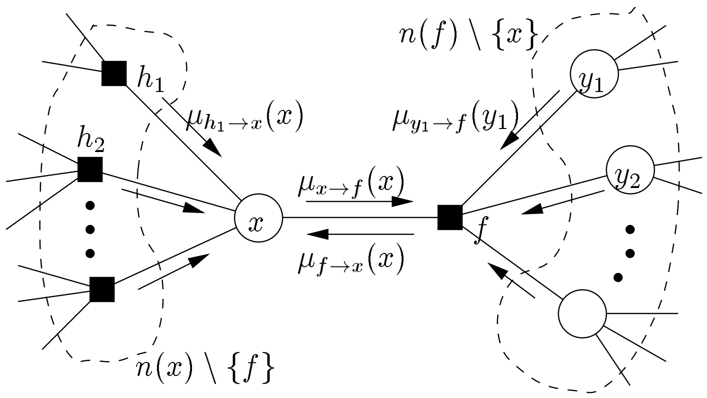


Figure 7: A factor graph fragment, showing the sum-product algorithm update rules.

3 Modeling Systems with Factor Graphs

We describe now a variety of ways in which factor graphs can be used to model *systems*, i.e., collections of interacting variables. When a system is modeled probabilistically, a factor graph can be used to represent the joint probability mass function of the variables that comprise the system. Factorizations of the this function can give important information about statistical dependencies among these variables.

Likewise, in "behavioral" modeling of systems—as in the work of Willems [44]—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 among the variables that define the codewords given the received output of a channel. For important classes of channels (e.g., memoryless channels) we will find that these models are closely related.

To assist in behavioral modeling, we find "Iverson's convention" [17, p. 24] to be a useful notation. If P is a predicate (i.e., Boolean proposition) involving some set of variables, then $[P]$ is the binary function that indicates the truth of P, i.e.,

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

For example, if x, y, and z take values in some field, then $f(x, y, z) = [x + y = z]$ is the function that takes a value of 1 if the condition $x + y = z$ is satisfied, and value 0 otherwise. We will use Iverson's convention in formulas only in those contexts in which it is sensible to have a $\{0,1\}$-valued quantity.

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

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

where we implicitly assume that $1 \cdot 1 = 1$, and for all x, $0 \cdot x = x \cdot 0 = 0$. Thus, if P can be written as a logical conjunction of predicates, then $[P]$ can be factored according to (8), and hence represented using a factor graph.

3.1 Behavioral Modeling

Let $x_1, x_2, \ldots, x_n$ be a collection of variables with configuration space S. By a *behavior* in S, we mean any nonempty subset B of S. The elements of B are the *valid configurations.* A system is specified via its behavior B, hence this approach is known as behavioral modeling.

Behavioral modeling occurs very naturally in coding theory. If the domain of each variable is taken to be 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

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

It is obvious that 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, each involving some subset of the variables. A configuration is deemed valid if and only if it passes all tests, i.e., the predicate $(x_1, \ldots, x_n) \in B$ can be written as a logical conjunction of a series of "simpler" predicates. Then χ_B factors according to (8) into a product of simpler "local factors" or *checks* each of which is itself a characteristic function indicating whether a particular subset of variables is "locally valid." Such a factorization can be represented using a factor graph.

► **Example 2.** (*Tanner Graphs for Linear Codes*)
The characteristic function for any linear $[n,k]$ code can be represented by a factor graph

having n variable nodes and (at least) $n-k$ 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 $\mathbf{x} \triangleq (x_1, x_2, \ldots, x_6)$ satisfying three simultaneous equations expressed in matrix form as $H\mathbf{x}^T = 0$. (This is a so-called *kernel representation*, since the linear code is defined as the set of vectors in the kernel of a particular linear transformation.) Membership in C is completely determined by checking whether *each* of the three equations is satisfied.

According to (8),

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

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 instead of a black square). Although strictly speaking the factor graph represents 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 [39].

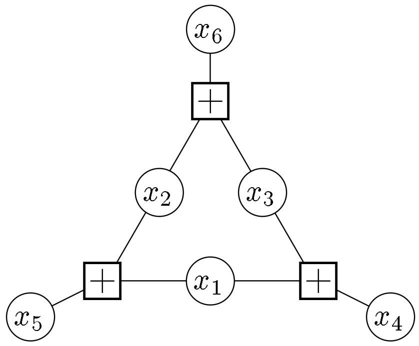


Figure 8: A Tanner graph for the binary linear code of Example 2.

It should be obvious that a Tanner graph for any $[n, k]$ linear block code can 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. ◀

Often when dealing with systems, a description of a system is simplified by introducing *hidden* (sometime called auxiliary, latent, or state) variables. Non-hidden variables are referred to as being *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 their visible coordinates is equal to C. Any factor graph for B is then considered simultaneously to be a factor graph for C. Following Forney [11], we will refer to a factor graph representing the characteristic function for a behavior with hidden variables as a TWL (Tanner/Wiberg/Loeliger) graph (see [42, 43]). In our factor graph diagrams, hidden variable nodes are indicated with a double circle.

An important class of models with hidden variables are the *trellis* representations for codes (see, e.g., [40] 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 the sequence of edge labels encountered in each directed path from the root vertex to the goal vertex is always a codeword in C, and that each codeword in C is represented by at least one such path. For example, Fig. 9(a) is a trellis for the code of Example 2. The root vertex is left-most, the goal vertex is right-most and edges are implicitly directed from left to right.

Every vertex in a trellis has a natural depth d, $0 \leq d \leq n$, defined as the distance from the root vertex, which has depth 0. The set of depth d vertices can be viewed as the domain of a *state variable* s_d.

A trellis divides naturally into *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 can be viewed as the domain of a (visible) variable x_i. In effect, each trellis section T_i defines a "local behavior" that must be obeyed by 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 constraint 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 trellis section T_i, and has s_{i-1}, x_i, and s_i as 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 TWL 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, T_2, the local behavior corresponding to the second trellis section from the left in Fig. 9, consists of the following triples (s_1, x_2, s_2):

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

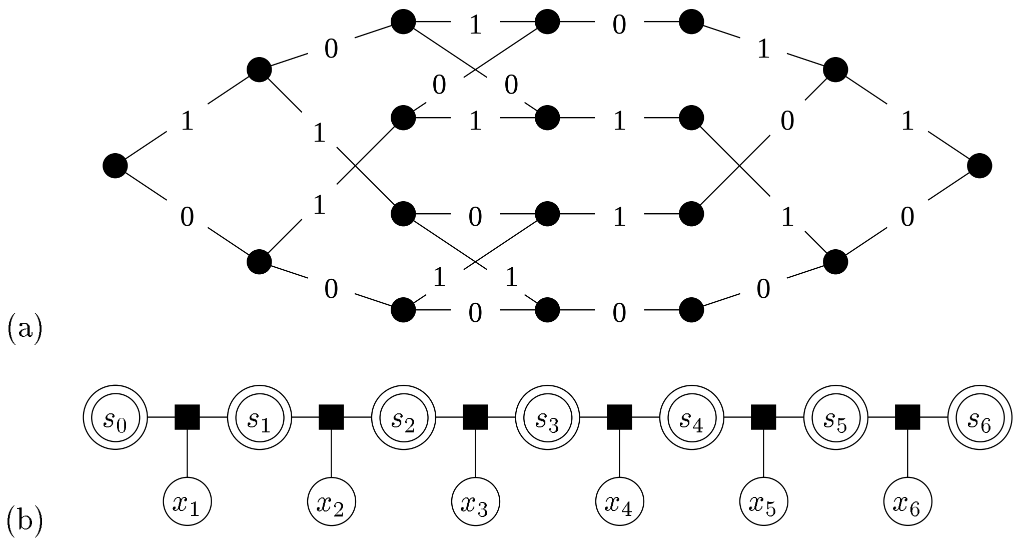


Figure 9: A trellis (a) and the corresponding TWL graph (b) for the code of Fig. 8.

where the alphabet of the state variables s_1 and s_2 was 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 check in the TWL 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 the 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 size of domains of the state variables (i.e., the number of states in the trellis) can easily become too large to be practical. Turbo codes, for example, have trellis representations with enormous state spaces [16]. However, there are factor graph representations for these code having reasonable complexity, but necessarily having cycles.

Trellises are convenient representations for a variety of signal models. For example, the generic factor graph of Fig. 10 can represent any time-invariant (or indeed, time-varying) state space model. 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 state, input symbol, output symbol, and right state. Here, a trellis edge has both an input label and an output label.

▶ **Example 4.** (*State-space models*)
For example, the classical linear state space model is given by the equations

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

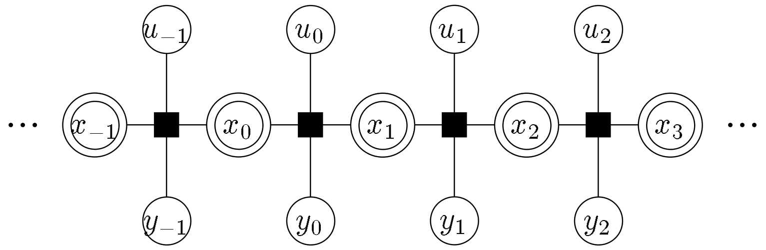


Figure 10: Generic factor graph for a time-invariant trellis.

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

Any such system gives rise to the factor graph of Fig. 10. The time-j check function $f(x(j), u(j), y(j), x(j+1))\colon F^m \times F^k \times F^n \times F^m \to \{0, 1\}$ 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 required by (11). ◀

3.2 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 expose one of our primary application interests by starting with an example from coding theory.

▶ **Example 5.** (*APP Distributions*)
Consider the situation most often modeled in coding theory, in which a codeword $x = (x_1, \ldots, x_n)$ is selected from a code C of length n, and transmitted over a channel with corresponding output $y = (y_1, \ldots, y_n)$. 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. We consider $g(x)$ to be a function of x only, with the components of y entering as parameters.

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

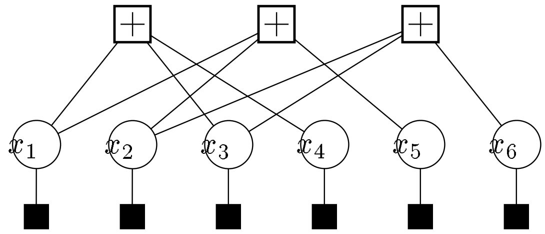


Figure 11: Factor graph the joint APP distribution of codeword symbols.

and $|C|$ is the number of codewords in C. Assuming also that the channel is memoryless, then by definition $f(y|x)$ 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

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

As described in the previous subsection, the characteristic function $\chi_C(x)$ itself may factor into a product of local indicator functions. 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.

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

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

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 can be represented with 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

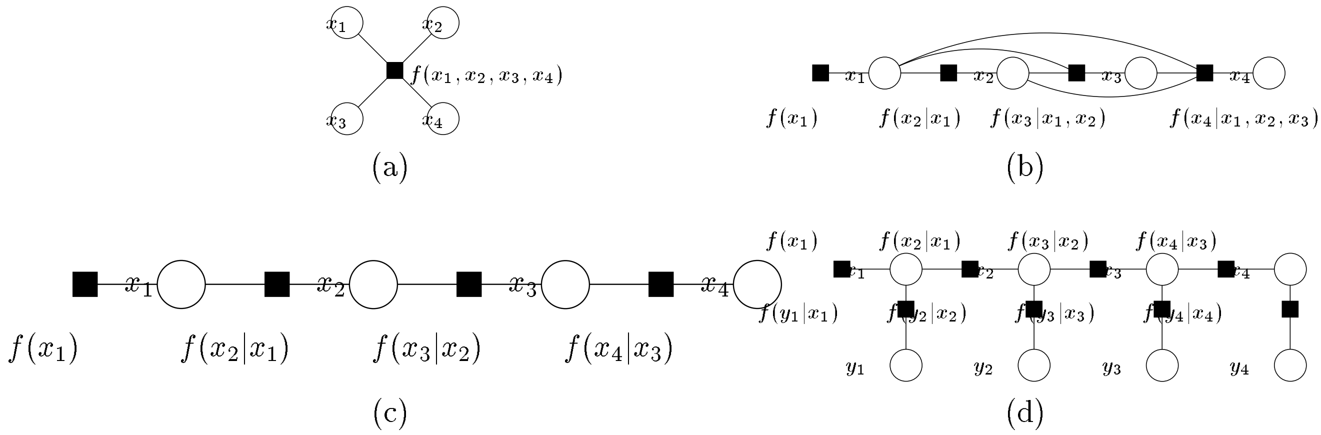


Figure 12: Factor graphs for probability distributions: (a) the trivial factor graph, (b) the chain-rule factorization, (c) a Markov chain, (d) a hidden Markov model.

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 $n = 4$, we have

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

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 $\mathbf{X}_1, \mathbf{X}_2, \ldots, \mathbf{X}_n$ (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).

If, in this Markov chain example, we cannot observe each $\mathbf{X}_i$ directly, but instead can observe only the output $\mathbf{Y}_i$ of a memoryless channel with $\mathbf{X}_i$ as input, 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). Hidden Markov models are widely used in a variety of applications; see, e.g., [36] for a tutorial emphasizing applications in signal processing.

Of course, since trellises are Markov models for codes, the strong resemblance between the factor graphs of Fig. 12(c) and (d) and the factor graphs representing trellises (Figs. 9(b) and 10) is not accidental. ◀

Factor graphs are not the first graph-based language for describing probability distributions. In Appendix B 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).

4 Trellis Processing

As described in the previous section, and important family of factor graphs are trellises, since they represent Markov models that are widely used in various applications. We now apply the sum-product algorithm to trellises, 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.

4.1 The Forward/Backward Algorithm

We start with the forward/backward algorithm, sometimes referred to in coding theory as the BCJR algorithm [4] 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 situation at hand, which is a combination of behavioral and probabilistic modeling. We have vectors $\mathbf{u} = (u_0, u_1, \ldots, u_{k-1})$, $\mathbf{x} = (x_0, x_1, \ldots, x_{n-1})$, and $\mathbf{s} = (s_0, \ldots, s_n)$ 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. Valid behavior is defined by local check functions $T_i(s_i, x_i, u_i, s_{i+1})$, as described in Examples 3 and 4. To handle situations such as terminated convolutional codes, we also allow for "autonomous behavior" in which the input variable is suppressed in certain trellis sections, as in the rightmost trellis section of Fig. 13.

This model is considered a "hidden" Markov model because we assume that we cannot observe the output symbols directly, but rather only the vector $\mathbf{y} = (y_0, y_1, \ldots, y_{n-1})$ that arises at the output of memoryless channel with $\mathbf{x}$ at its input. As discussed in Example 5, the *a posteriori* joint probability mass function for $\mathbf{u}$, s, and $\mathbf{x}$ given the observation $\mathbf{y}$ is proportional to

$$g(\mathbf{u}, \mathrm{s}, \mathbf{x}; \mathbf{y}) := \prod_{i=0}^{n-1} T_i(s_i, x_i, u_i, s_{i+1}) \prod_{i=0}^{n-1} f(y_i|x_i)$$

where $\mathbf{y}$ is considered a parameter of g (not an argument). The factor graph of Fig. 13 represents this factorization of g.

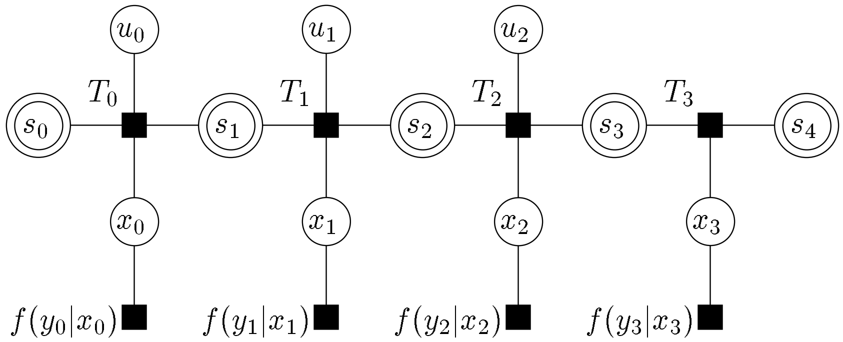


Figure 13: The factor graph on which the forward/backward algorithm operates: the s_i are state variables, the u_i are input variables, the x_i are output variables, and each y_i is the observation at the output of a memoryless channel with x_i at the input.

Given the observation of $\mathbf{y}$, we would like to find, for each i, $p(u_i|\mathbf{y})$, the *a posteriori* probabilities (APPs) for the input symbols. From probability theory, it follows that these APP values are proportional to marginal functions associated with g, i.e.,

$$p(u_i|\mathbf{y}) \;\propto\; \sum_{\sim\{u_i\}} g(\mathbf{u}, \mathrm{s}, \mathbf{x}; \mathbf{y}).$$

Since the factor graph of Fig. 13 is cycle-free, these marginal functions can be computed by applying the 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 unit 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. Since the output variable nodes are of degree two, no computation is performed; instead, incoming messages are simply copied and sent to the corresponding trellis check node.

Once the initialization has been performed, the two endmost trellis check nodes T_0 and T_{n-1} will have received messages on three of their four edges, and so will be in a position to create an output message to a neighboring state variable node. Again, since the state variables are of degree two, no computation is performed; incoming messages are simply copied. In the literature on the forward/backward algorithm (e.g., [4]), the message $\mu_{x_i \to T_i}(x_i)$ is denoted $\gamma(x_i)$, the message $\mu_{s_i \to T_i}(s_i)$ is denoted $\alpha(s_i)$, and the message $\mu_{s_i \to T_{i-1}}(s_i)$ is denoted $\beta(s_i)$. Additionally, the message $\mu_{T_i \to u_i}(u_i)$ will be denoted $\delta(u_i)$.

The operation of the sum-product algorithm creates two natural recursions: one allowing $\alpha(s_{i+1})$ to be computed as a function of $\alpha(s_i)$ and $\gamma(x_i)$ and the other allowing $\beta(s_i)$ to be computed as a function of $\beta(s_{i+1}$ and $\gamma(x_i)$. These two recursions are referred to, respectively, as the *forward* and *backward* recursions due to the direction of message flow in the trellis. The forward and backward recursions do not interact, so their computation could occur in parallel. Fig. 14(a) gives a detailed view of these messages for a single trellis section. The local function in this figure represents the trellis check $T_i(s_i, u_i, x_i, s_{i+1})$.

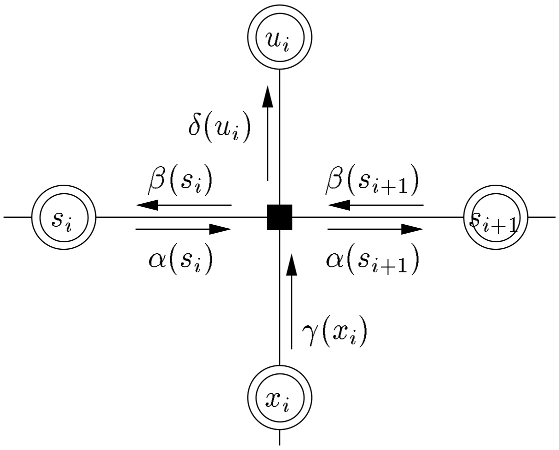


Figure 14: A detailed view of the messages passed during the operation of the forward/backward algorithm.

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

$$\begin{aligned} \alpha(s_{i+1}) &= \sum_{\sim\{s_{i+1}\}} T_i(s_i, u_i, x_i, s_{i+1})\alpha(s_i)\gamma(x_i), \\ \beta(s_i) &= \sum_{\sim\{s_i\}} T_i(s_i, u_i, x_i, s_{i+1})\beta(s_{i+1})\gamma(x_i). \end{aligned}$$

The algorithm terminates with the computation of the δ messages.

Termination:

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

In each of these sums, the summand is zero except for combinations of s_i, u_i, x_i and s_{i+1} representing a valid trellis edge; fundamentally, therefore, these sums can be viewed as being defined over valid trellis edges. For each edge $e = (s_i, u_i, x_i, s_{i+1})$ we let $\alpha(e) = \alpha(s_i)$, $\beta(e) = \beta(s_{i+1})$ 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 can be re-written as

$$\begin{aligned} \alpha(s_{i+1}) &= \sum_{e \in E_i(s_{i+1})} \alpha(e)\gamma(e) \\ \beta(s_i) &= \sum_{e \in E_i(s_i)} \beta(e)\gamma(e). \end{aligned} \tag{13}$$

The basic operation in the forward and backward recursions is, therefore, a "sum of products."

The α and β messages have a well-defined probabilistic interpretation: $\alpha(s_i)$ is proportional to the conditional probability mass function for s_i given the observation of the "past" $\mathbf{y}_0, \ldots, \mathbf{y}_{i-1}$; i.e., for each state $s_i \in S_i$, $\alpha(s_i)$ is the conditional probability that the transmitted sequence passed through state s_i given observation of the past. Similarly, $\beta(s_{i+1})$ is proportional to the conditional probability mass function for s_{i+1} given the observation of the "future" $\mathbf{y}_{i+1}, \mathbf{y}_{i+2}, \ldots$, i.e., the conditional probability that the transmitted sequence passed through state s_{i+1}. The probability that the transmitted sequence took a particular edge $e = (s_i, x_i, s_{i+1}) \in T_i$ is 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 $\mathbf{s}$ vector, or for the $\mathbf{x}$ vector, these could also be computed by the forward/backward algorithm. See [36] for a tutorial on some of the applications of the forward/backward algorithm to applications in signal processing.

4.2 The Min-Sum Semiring and the Viterbi Algorithm

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

As mentioned in Section 1, the codomain R of the global function g represented by a factor graph can in general be any semiring with two operations '+' and '·', satisfying the distributive law:

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

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 is essentially an application of the distributive law (14), and so applies equally well to the general semring case. This key observation is central to the "generalized distributive law" of [2].

A semiring of particular interest to solve the MLSD problem is the "max-product" semiring, in which real summation is replaced with the "max" operator. For non-negative real-valued quantities x, y, and z,

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

so the distributive law is satisfied. Furthermore, with maximization as a summary operator, the maximum value of a non-negative real valued function $g(x_1, \ldots, x_n)$ can be viewed as the "complete summary" of g, i.e.,

$$\max g(x_1, \ldots, x_n) = \sum_{\sim\{\emptyset\}} g(x_1, \ldots, x_n),$$

where we have used the "not-sum" notation for maximization. We will be interested not only in determining this maximum value, but also in finding a valid configuration $\mathbf{x}$ that achieves this maximum.

In practice, MLSD is most often carried out in the negative log-likelihood domain. Here, multiplicative decompositions become additive, but the structure of the underlying factor graph is unaffected. The 'max' operation becomes a 'min' operation, so that we deal with the "min-sum" semiring. For real x, y, z,

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

so the distributive law is satisfied.

Let $f(\mathbf{x}, \mathrm{s}|\mathbf{y}) = -a \ln p(\mathbf{x}, \mathrm{s}|\mathbf{y}) + b$ where a and b are any convenient constants with $a > 0$. Applying the sum-product (or, more accurately, 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 write an update equation for the various messages. For example, the basic update equation corresponding to (13) is

$$\alpha(s_{i+1}) = \min_{e \in E_i(s_{i+1})} (\alpha(e) + \gamma(e)), \tag{15}$$

so that the basic operation is a "minimum of sums." A similar recursion applies in the backward direction, and from the results of the two recursions the most likely sequence can 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" performed in making a decision, even the conventional Viterbi algorithm might be viewed as being bidirectional.

4.3 Kalman Filtering

In this section, we derive the Kalman filter (see, e.g., [3, 30]) as an instance of the sum-product algorithm operating in the factor graph corresponding to the 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

$$\begin{aligned} x_{j+1} &= A_j x_j + B_j u_j \\ y_j &= C_j x_j + D_j w_j \end{aligned}$$

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 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

$$\mathcal{N}(x, m, \sigma^2) := \exp\left(-(x-m)^2/(2\sigma^2) + K(\sigma)\right)$$

to represent Gaussian density functions, where m and σ^2 represent the mean and variance of of the corresponding random variable and $K(\sigma)$ is a normalizing constant that is irrelevant for our purposes. 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

$$\sigma_3^2 = \left(\frac{1}{\sigma_1^2} + \frac{1}{\sigma_2^2}\right)^{-1}.$$

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, given the particular Markov structure of this signal model, it is not difficult to see that the conditional joint probability density function of the state variables $x_1, \ldots, x_k$ given $y_1, \ldots, y_k$ is given by

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

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. The observed values of the output variables enter as parameters, not as function arguments.

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

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

where we have introduced an obvious generalization of the "not-sum" notation. The mean of this conditional density, i.e., $\hat{x}_{k|k} = E[x_k|y_1,\ldots,y_k]$ is the minimum mean squared error 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 a summary operation.

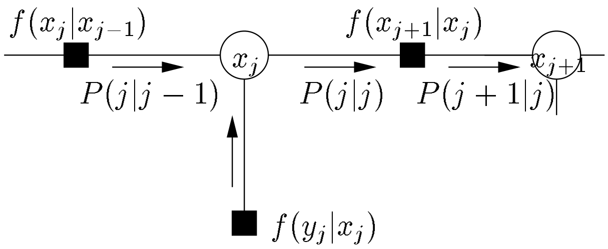


Figure 15: A portion of the factor graph corresponding to (18).

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 can be represented by the pair $(\hat{m}_{j|j-1}, \sigma_{j|j-1}^2)$. We interpret $\hat{m}_{j|j-1}$ as a *prediction* of x_j given the set of observations up to time $j-1$.

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

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

where

$$\begin{aligned}\hat{m}_{j|j} &= \frac{D_j^2\hat{m}_{j|j-1} + C_j y_j \sigma^2_{j|j-1}}{C_j^2\sigma^2_{j|j-1} + D_j^2} \\ &= \hat{m}_{j|j-1} + \frac{C_j\sigma^2_{j|j-1}}{C_j^2\sigma^2_{j|j-1} + D_j^2}(y_j - C_j\hat{m}_{j|j-1})\end{aligned}$$

and

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

Likewise, applying (17), we have

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

where

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

and

$$\begin{aligned}\sigma^2_{j|j+1} &= A_j^2\sigma^2_{j|j} + B_j^2 \\ &= \frac{A_j^2\sigma^2_{j|j-1}}{C_j^2\sigma^2_{j|j-1} + D_j^2} + B_j^2.\end{aligned}$$

In (19), the value

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

is called the *filter gain*. These updates are exactly equal to the updates used by Kalman filtering [3].

5 Turbo Processing: Operating the Sum-Product Algorithm in Factor Graphs with Cycles

In addition to its application in cycle-free factor graphs, the sum-product algorithm can also be applied to factor graphs with cycles simply by following the same message propagation rules. 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 case of no cycles, the results of the sum-product algorithm operating in a factor graph with cycles cannot in general be interpreted as being exact function summaries. However, some of the most exciting applications of the sum-product algorithm—for example, the decoding of turbo codes or low-density parity-check codes—arise precisely in situations in which the underlying factor graph *does* have cycles. Extensive simulation results (see, e.g., [7, 27, 28]) show that sum-product based decoding algorithms with very long codes can astonishing performance (within a fraction of a decibel of the Shannon limit in some cases), even though the underlying factor graph has cycles.

While descriptions of the way in which the sum-product algorithm is applied to a variety of "compound codes" are given in [24], in this section, we restrict ourselves to three examples: turbo codes [7], low-density parity-check codes [15], and repeat-accumulate codes [10].

5.1 Message Passing Schedules

Although not necessary in a practical implementation, we assume that messages are passed by the sum-product algorithm in synchronization 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 the 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 at times prior to 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 not have 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 any time along any edge.

A message-passing *schedule* in a factor graph a specification of messages are passed during each clock tick. Obviously a wide variety of message passing schedules are possible. For example, the so-called *flooding schedule* [24] calls for a message to pass in each direction over each edge at each clock tick. Any schedule in which at most one message is passed (anywhere in the graph), is called a *serial schedule*.

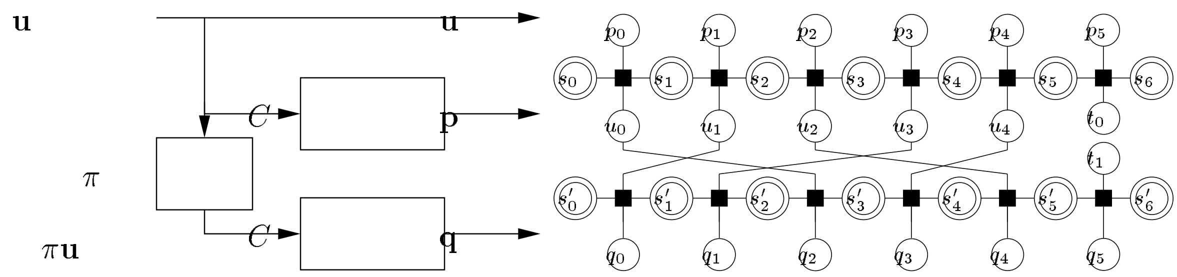
We will say that a vertex v has a message *pending* at an edge e if it has received any messages on edges other than e *after* the transmission of the most previous message sent on e. Such a message is pending since the messages more recently received can affect the message to be sent on e. The receipt of a message at v from an edge e will create pending messages at all *other* edges incident on v. Only pending messages ever need to be transmitted, since only pending messages have the potential to be different than 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 infinite schedules are rendered finite by truncation. The sum-product algorithm terminates, for a finite schedule, by computing, for each x_i, the product of the messages received at variable node x_i. If x_i has no messages pending, this computation is equivalent to the product of the messages sent and received on any single edge incident on x_i.

5.2 Iterative Decoding of Turbo Codes

A "turbo code" or parallel concatenated convolutional code has the encoder structure shown in Fig. 16(a). A block $\mathbf{u}$ of data to be transmitted enters the systematic encoder which produces $\mathbf{u}$, and two parity-check streams $\mathbf{p}$ and $\mathbf{q}$ at its output. The first parity-check stream $\mathbf{p}$ is generated via a standard recursive convolutional encoder; viewed together, $\mathbf{u}$ and $\mathbf{p}$ would form the output of a standard rate 1/2 convolutional code. The innovation in the structure of the turbo code is the manner in which the second parity-check stream $\mathbf{q}$ is generated. This stream is generated by applying a permutation π to the input stream, and applying the permuted stream to a second convolutional encoder. All output streams $\mathbf{u}$, $\mathbf{p}$ and $\mathbf{q}$ are transmitted over the channel. Both constituent convolutional encoders are typically terminated in a known ending state; the corresponding symbols (t_0, t_1, p_5, q_5 in Fig. 16(b)) are also transmitted over the channel.



(a) (b)

Figure 16: Turbo code: (a) encoder block diagram, (b) factor graph.

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.

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 [24, Fig. 10].

5.3 Low-density Parity-check Codes

Low-density parity-check (LDPC) codes were introduced by Gallager [15] 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 $(2, 4)$ low-density parity-check code. The check enforces the condition that the adjacent symbols should have even overall parity, much as in Example 2.

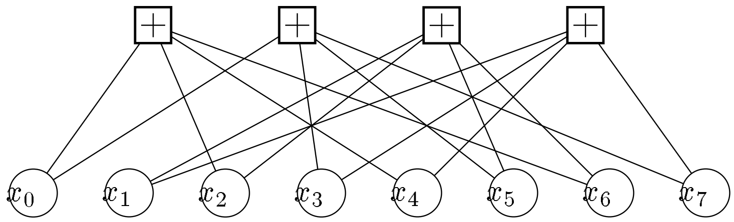


Figure 17: A factor graph for a low-density parity-check code.

Low-density parity-check 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 [27, 28].

5.4 Repeat-Accumulate Codes

Repeat-accumulate (RA) codes are a special, low-complexity class of turbo codes introduced by Divsalar, *et al.*, who initially devised these codes because their ensemble weight distributions are relatively easily derived. 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 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 for $i > 1$, $x_i = x_{i-1} + z_i$.

A factor graph for an RA code is shown in Fig. 18. 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.)

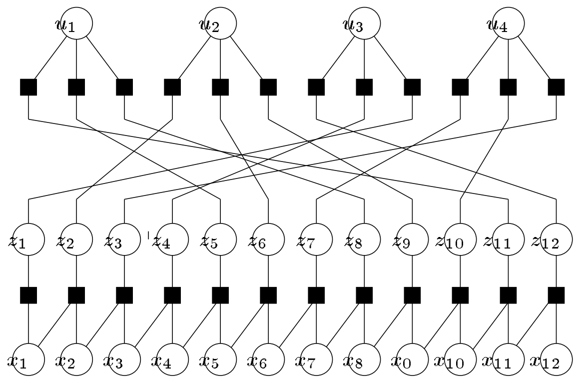


Figure 18: A factor graph for a repeat-accumulate code.

5.5 Simplifications for Binary Variables and Parity-Checks

For particular decoding applications, the generic updating rules (5) and (6) can often be substantially simplified. We treat here only the important case where all variables are binary (i.e., Bernoulli random variables) and all functions (except single-variable functions) are parity checks, as in Figs. 11(b), Fig. 17, and Fig. 18. This includes, in particular, low-density parity check codes and repeat-accumulate codes.

Consider the case where the probability mass function for a binary random variable is represented as the binary vector (p_0, p_1), where $p_0 + p_1 = 1$. We will normalize all messages so that the latter equality is preserved. 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

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

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

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

We view (20) and (21) as specifying the behavior of the ideal "probability gates" illustrated in Fig. 19 that operate much like logic gates, but with soft ("fuzzy") values.

Since $p_0 + p_1 = 1$, binary probability mass functions can be parameterized by a single value. Depending on the parameterization, various "probability gates" arise. We give four different parameterizations, and derive the VAR and CHK functions for each.

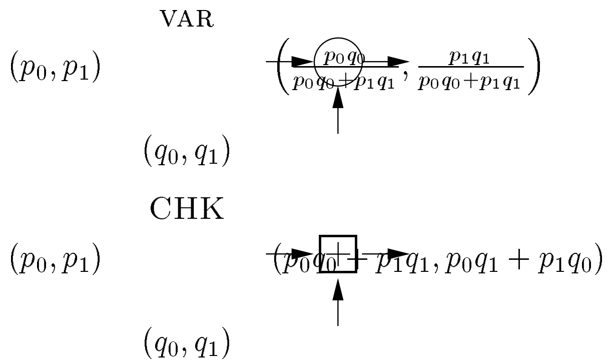


Figure 19: Ideal binary VAR and CHK "probability gate" functions.

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

$$\begin{aligned}\textsc{var}(\lambda_1, \lambda_2) &= \lambda_1 \lambda_2 \\ \textsc{chk}(\lambda_1, \lambda_2) &= \frac{1 + \lambda_1 \lambda_2}{\lambda_1 + \lambda_2}\end{aligned}$$

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

$$\begin{aligned}\textsc{var}(\Lambda_1, \Lambda_2) &= \Lambda_1 + \Lambda_2 \\ \textsc{chk}(\Lambda_1, \Lambda_2) &= \ln\left(\cosh\left((\Lambda_1 + \Lambda_2)/2\right)\right) - \ln\left(\cosh\left((\Lambda_1 - \Lambda_2)/2\right)\right) \\ &= 2 \tanh^{-1}\left(\tanh(\Lambda_1/2) \tanh(\Lambda_2/2)\right).\end{aligned} \tag{22}$$

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

$$\begin{aligned}\textsc{var}(\delta_1, \delta_2) &= \frac{\delta_1 + \delta_2}{1 + \delta_1 \delta_2} \\ \textsc{chk}(\delta_1, \delta_2) &= \delta_1 \delta_2\end{aligned}$$

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

$$\begin{aligned}\textsc{var}(\Delta_1, \Delta_2) &= \begin{cases} s \ln\left(\frac{\cosh((|\Delta_1| + |\Delta_2|)/2)}{\cosh((|\Delta_1| - |\Delta_2|)/2)}\right) & \text{if } \mathrm{sgn}(\Delta_1) = \mathrm{sgn}(\Delta_2) = s \\ s \cdot \mathrm{sgn}(|\Delta_1| - |\Delta_2|) \ln\left(\frac{\sinh((|\Delta_1| + |\Delta_2|)/2)}{\sinh((||\Delta_1| - |\Delta_2||)/2)}\right) & \text{if } \mathrm{sgn}(\Delta_1) = -\mathrm{sgn}(\Delta_2) = -s \end{cases} \\ \textsc{chk}(\Lambda_1, \Lambda_2) &= \mathrm{sgn}(\Delta_1)\mathrm{sgn}(\Delta_2)(|\Delta_1| + |\Delta_2|)\end{aligned}$$

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

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

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

By applying the equivalence between factor graphs illustrated in Fig. 20, it is easy to extend these formulas to the case where variable nodes or check nodes have degree larger than three. In particular, we can extend the VAR and CHK functions to more than two arguments via the relations

$$\begin{aligned} \text{VAR}(x_1, x_2 \ldots, x_n) &= \text{VAR}(x_1, \text{VAR}(x_2, \ldots, x_n)), \\ \text{CHK}(x_1, x_2 \ldots, x_n) &= \text{CHK}(x_1, \text{CHK}(x_2, \ldots, x_n)). \end{aligned}$$

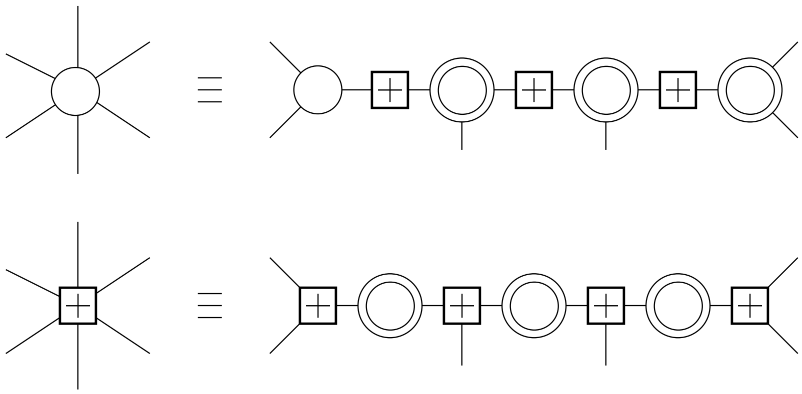


Figure 20: Transforming variable and check nodes of high degree to multiple nodes of degree three.

6 Factor Graph Transformations

In this section we describe a number of straightforward transformations that may be applied to a factor graph. By applying these transformations, it is sometimes possible to transform 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, and we apply them to derive a fast Fourier transform algorithm from the factor graph representing the DFT kernel in Appendix C. Similar general procedures are described in [22, 26].

6.1 Clustering

It is always possible to cluster (i.e., group) 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 groupings. 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 pairing of v and 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, by the "pairing of v and w" we mean a new variable (v, w) with 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 is easily accomplished without increasing the complexity of the local functions.

When v and w are local functions, by the pairing of v and 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.

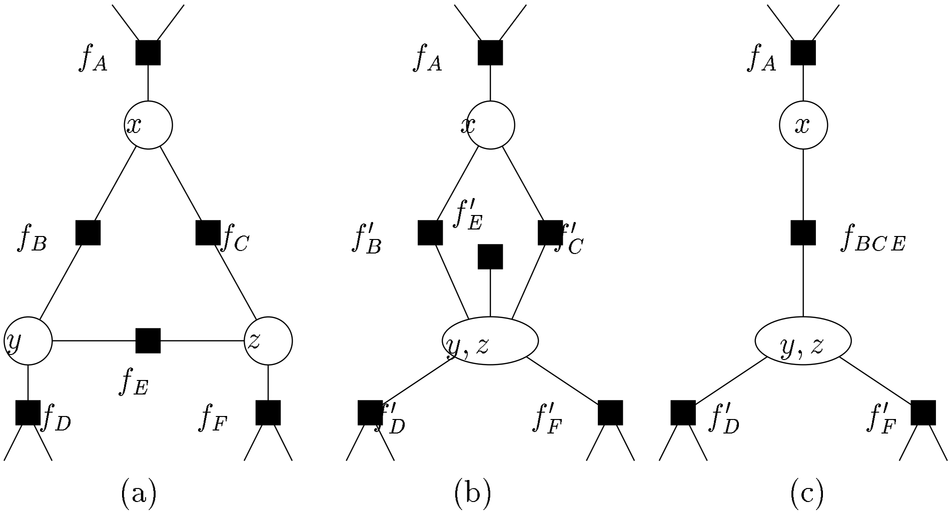


Figure 21: Grouping transformations: (a) original factor graph fragment, (b) variable nodes y and z grouped together, (c) function nodes f_B, f_C and f_E grouped together.

Grouping nodes may eliminate cycles in the graph so that the sum-product algorithm in the new graph is exact. For example, grouping the nodes associated with y and z in the factor graph fragment of Fig. 21(a) and connecting the neighbors of both nodes to the new grouped node, we obtain the factor graph fragment shown in Fig. 21(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

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

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

In Fig. 21(b), there is still one cycle; however, it can easily be removed by grouping function nodes. In Fig. 21(c), we have grouped the local functions corresponding to f'_B, f'_C and f'_E:

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

The new global function is

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

which is identical to the original global function.

In this case, by grouping 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 can 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 $|A_y||A_z|$.

6.2 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 grouping transformation. The function nodes incident on y would have to be modified as in a grouping 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 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 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. Note that the global function is unaffected by this transformation.

Fig. 22(a) shows a factor graph, and Fig. 22(b) shows an equivalent factor graph in which x_1 has been stretched to all variable nodes.

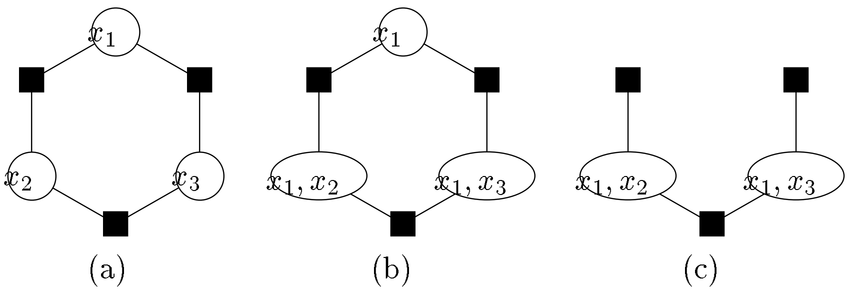


Figure 22: Stretching transformation: (a) original factor graph, (b) node x_1 is stretched to x_2 and x_3, (c) the node representing x_1 alone is now redundant and can be removed.

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 with 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. 22(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 can be deleted from a factor graph. (Redundant edges must be removed one 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 can be deleted.

For example, the node containing x_1 alone is redundant in Fig. 22(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. 22(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, 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.

6.3 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 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." We apply these ideas to derive a Fast Fourier Transform algorithm in Appendix C.

7 Conclusions

Factor graphs provide a natural graphical description of the factorization of a global function into a product of local functions. As such, 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—operating in a factor graph following 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 fast Fourier transform algorithms! Various extensions of these algorithms; for example, a Kalman filter with forward/backward propagation or operating in a tree-structured signal model, although not treated in this paper, can be derived in a straightforward manner by applying the principles enunciated in this paper.

We have emphasized that the sum-product algorithm can be applied to arbitrary factor graphs, cycle-free or not. In the cycle-free case, we have shown that the sum-product algorithm can be used to compute function summaries *exactly* when the factor graph is finite. 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 low-density parity-check 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 approach fit naturally in the factor graph framework. The generality of allowing arbitrary functions (not just probability distributions) 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. We feel 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 indeed be fruitful.

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

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

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 can 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 that are descendants of x's ith neighbor, and X_i is the set of variables that are descendants of x's ith neighbor. 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

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

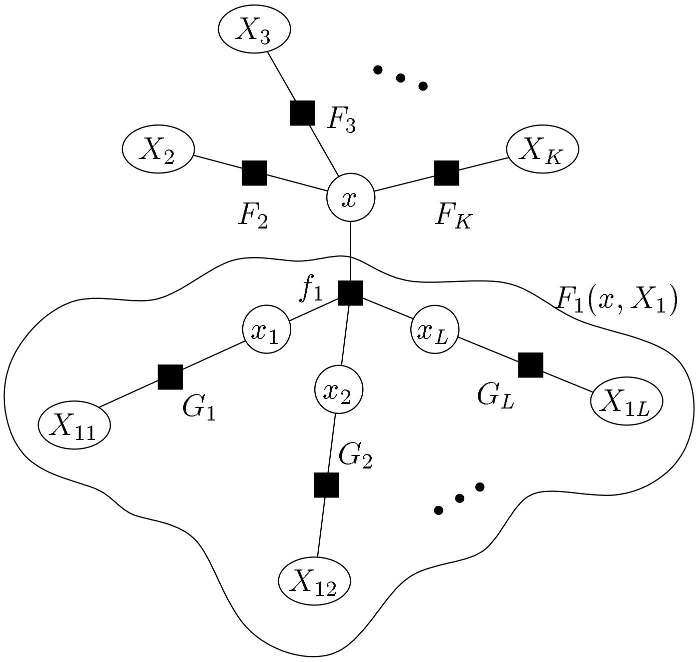


Figure 23: A generic factor tree.

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

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

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

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

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

In words, we see that if $f_1(x, x_1, \dots, x_L)$ 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 a 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 x as root into an expression tree for $\sum_{\sim\{x\}} g(x, x_1, \ldots, x_{N-1})$, as illustrated in Fig. 5.

B Other Graphical Models for Probability Distributions

Factor graphs are not 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).

B.1 Markov Random Fields

A Markov random field (see, e.g., [23]) 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:

$$(\forall v \in V) \quad p(v|V\backslash\{v\}) = p(v|n(v)), \tag{23}$$

where $n(v)$ denotes the set of neighbors of v. In other words, G is an MRF if every variable v is independent of non-neighboring 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., [23, 35, 21, 20]).

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 can be expressed as the product of a collection of Gibbs potential functions, defined on the set Q of cliques in the MRF, i.e.,

$$p(v_1, v_2, \ldots, v_N) = Z^{-1} \prod_{E \in Q} f_E(V_E) \tag{24}$$

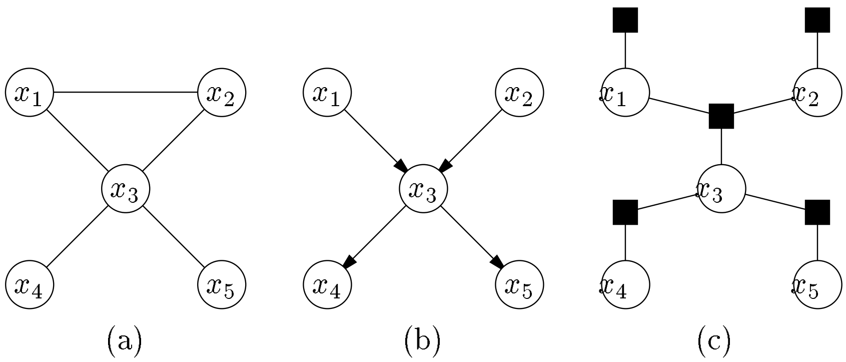


Figure 24: Graphical probability models: (a) a Markov random field, (b) a Bayesian network, (c) a factor graph.

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) can be used to express the factorization

$$p(v_1, v_2, v_3, v_4, v_5) = Z^{-1} f_C(v_1, v_2, v_3) f_D(v_3, v_4) f_E(v_4, v_5).$$

Clearly (24) 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 Qs in (24), 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 [24].)

B.2 Bayesian Networks

Bayesian networks (see, e.g., [34, 22, 12]) are graphical models for a collection of random variables that are based on directed acyclic graphs (DAGs). Bayesian networks, combined with Pearl's "belief propagation algorithm" [34], 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 [27], and more recently at least two papers [24, 33] develop a view of the "turbo decoding" algorithm [7] as an instance of probability propagation in a Bayesian network code model.

Each node v in a Bayesian network is associated with a random variable. Denoting by $\mathbf{a}(v)$ 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 is written as

$$p(v_1, v_2, \ldots, v_n) = \prod_{i=1}^{n} p(v_i | \mathbf{a}(v_i)), \tag{25}$$

where, if $\mathbf{a}(v_i) = \emptyset$, (i.e., v_i has no parents) then we take $p(v_i|\emptyset) = p(v_i)$. For example—*cf.* (2)—Fig. 24(b) shows a Bayesian network that expresses the factorization

$$p(v_1, v_2, v_3, v_4, v_5) = p(v_1)p(v_2)p(v_3|v_1, v_2)p(v_4|v_3)p(v_5|v_3). \tag{26}$$

Again, as do 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 (26) 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 [34, 22]. It is easy to convert a Bayesian network into a factor graph: simply introduce a function node for each factor $p(v_i|\mathbf{a}(v_i))$ in (25) and draw edges from this node to v_i and its parents $\mathbf{a}(v_i)$. 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 [34, pp. 182–183] can easily be derived from the general sum-product algorithm update equations (5) and (6) as follows.

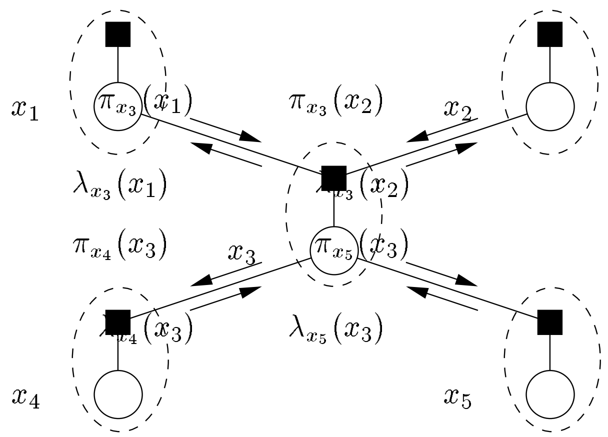


Figure 25: Messages sent in belief propagation.

In belief propagation, messages are sent between "variable nodes," corresponding to the dashed ellipses for the particular Bayesian network shown in Fig. 25. If, in a Bayesian network, an edge is directed from vertex p to vertex c then p is a parent of c and c is a child of p. Messages sent among between variables are always functions of the parent p. In [34], 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 that this local function is the conditional probability mass function $f(x_3|x_1, x_2)$, hence

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

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

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

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

$$\lambda_x(a) = \left(\prod_{d \in \mathbf{d}(x)} \lambda_d(x) f(x|\mathbf{a}(x)) \prod_{p \in \mathbf{a}(x)\setminus\{a\}} \pi_x(p) \right) \downarrow a. \tag{27}$$

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

$$\pi_d(x) = \prod_{c \in \mathbf{d}(x)\setminus\{d\}} \lambda_c(x) \left(f(x|\mathbf{a}(x)) \prod_{a \in \mathbf{a}(x)} \pi_x(a) \right) \downarrow x. \tag{28}$$

The termination condition for cycle-free graphs, called the "belief update" equation in [34], is given by the product of the messages received by x in the factor graph:

$$BEL(x) = \prod_{d \in \mathbf{d}(x)} \lambda_d(x) \left(f(x|\mathbf{a}(x)) \prod_{a \in \mathbf{a}(x)} \pi_x(a) \right) \downarrow x. \tag{29}$$

Pearl also introduces a scale factor in (28) and (29) so that the resulting messages properly represent probability mass functions. The relative complexity of (27)–(29) compared with the simplicity of the sum-product update rule given in Section 2 provides a strong pedagogical incentive for the introduction of factor graphs.

C The FFT

An important observation due to Aji and McEliece [1, 2] is that various fast transform algorithms can be developed using a graph-based approach. In this section we show how this approach can be used to derive a Fast Fourier Transform (FFT).

The discrete Fourier transform (DFT) is a widely used tool for the analysis of discrete-time signals. Let $\mathbf{w} = (w_0, \ldots, w_{N-1})$ 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 $\mathbf{w}$ is the complex-valued N-tuple $\mathbf{W} = (W_0, \ldots, W_{N-1})$ where

$$W_k = \sum_{n=0}^{N-1} w_n \Omega^{-nk}, \quad k = 0, 1, \ldots, N-1. \tag{30}$$

Consider now the case where N is a power of two, e.g., $N = 8$ for concreteness. We express variables n and k in (30) 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 the x_is and y_is as

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

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. 26(a).

We observe that

$$W_k = W_{4y_2+2y_1+y_0} = \sum_{x_0}\sum_{x_1}\sum_{x_2} g(x_0, x_1, x_2, y_0, y_1, y_2) \tag{31}$$

so that the DFT can be viewed as a marginal function, much like a probability mass function. When N is composite, similar prime-factor-based decompositions of n and k will result in similar factor graph representations for the DFT kernel.

The factor graph in Fig. 26(a) has cycles, yet 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. 26(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. 26(b), essentially by defining

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

we arrive at the spanning tree shown in Fig. 26(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 a fast Fourier transform as an instance of the sum-product algorithm.

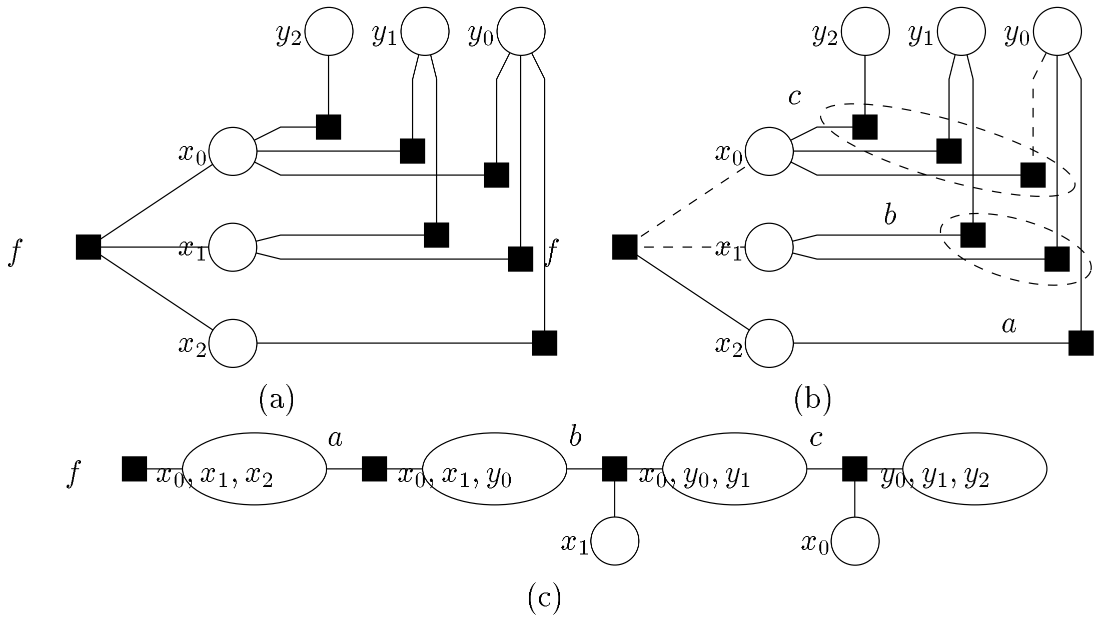


Figure 26: The discrete Fourier transform kernel: (a) factor graph; (b) a particular spanning tree; (c) spanning tree after clustering and stretching transformation.

Acknowledgments

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

The work of F. R. Kschischang took place while on sabbatical leave at the Massachusetts Institute of Technology, and was supported in part by the Office of Naval Research under Grant No. N00014-96-1-0930, and the Army Research Laboratory under Cooperative Agreement DAAL01-96-2-0002. The hospitality of Prof. G. W. Wornell of MIT is gratefully acknowledged. B. J. Frey, a Beckman Fellow at the Beckman Institute of Advanced Science and Technology, University of Illinois at Urbana-Champaign, was supported by a grant from the Arnold and Mabel Beckman Foundation.

References

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

[2] S. M. Aji and R. J. McEliece, "The generalized distributive law," preprint available on-line from `http://www.systems.caltech.edu/EE/faculty/rjm`, 1998.

[3] B. D. O. Anderson and J. B. Moore, *Optimal Filtering*. Englewood Cliffs, N.J.: 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. 20, pp. 284–287, Mar. 1974.

[5] L. E. Baum and T. Petrie, "Statistical inference for probabilistic functions of finite state Markov chains," *Annals of Mathematical Statistics*, vol. 37, pp. 1559–1563, 1966.

[6] S. Benedetto and G. Montorsi, "Iterative decoding of serially concatenated convolutional codes," *Electr. Lett.,* vol. 32, pp. 1186–1188, June 1996.

[7] C. Berrou, A. Glavieux, and P. Thitimajshima, "Near Shannon-limit error-correcting coding and decoding: turbo codes," *Proc. ICC'93,* pp. 1064–1070, Geneva, May 1993.

[8] U. Bertelè and F. Brioschi, *Nonserial Dynamic Programming*. New York: Academic Press, 1972.

[9] P. Dayan, G. E. Hinton, R. M. Neal, and R. S. Zemel, "The Helmholtz machine," *Neural Computation*, vol. 7, pp. 889–904, 1995.

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

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

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

[13] B. J. Frey, P. Dayan, and G. E. Hinton, "A simple algorithm that discovers efficient perceptual codes," in *Computational and Psychophysical Mechanisms of Visual Coding*, (M. Jenkin and L. R. Harris, eds). New York, NY: Cambridge University Press, pp. 296–315, 1997.

[14] B. J. Frey and G. E. Hinton, "Variational learning in non-linear Gaussian belief networks," to appear in *Neural Computation*, 1998.

[15] R. G. Gallager, *Low-Density Parity-Check Codes*. Cambridge, MA: M.I.T. Press, 1963.

[16] R. Garello, G. Montorsi, S. Benedetto and G. Cancellieri, "Interleaver Properties and their Applications to the Trellis Complexity Analysis of Turbo Codes," submitted to *IEEE Trans. on Communications*, 1999.

[17] R. L. Graham, D. E. Knuth and O. Patashnik, *Concrete Mathematics.* New York, NY: Addison-Wesley, 1989.

[18] J. Hagenauer, E. Offer and L. Papke, "Iterative decoding of binary block and convolutional codes," *IEEE Trans. on Inform. Theory*, vol. 42, pp. 429–445, March 1996.

[19] G. E. Hinton, P. Dayan, B. J. Frey, and R. M. Neal, "The wake-sleep algorithm for unsupervised neural networks," *Science*, vol. 268, pp. 1158–1161, 1995.

[20] 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.), vol. I, pp. 282–317, Cambridge MA.: MIT Press, 1986.

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

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

[23] R. Kindermann and J. L. Snell, *Markov Random Fields and their Applications.* Providence, Rhode Island: American Mathematical Society, 1980.

[24] F. R. Kschischang and B. J. Frey, "Iterative decoding of compound codes by probability propagation in graphical models," *IEEE J. Selected Areas in Commun.*, vol. 16, 1998.

[25] S. L. Lauritzen and F. V. Jensen, "Local computation with valuations from a commutative semigroup," *Annals of Math. and Artificial Intelligence*, vol. 21, pp. 51–69, 1997.

[26] S. L. Lauritzen and D. J. Spiegelhalter, "Local computations with probabilities on graphical structures and their application to expert systems," *Journal of the Royal Statistical Society, Series B*, vol. 50, pp. 157–224, 1988.

[27] D. J. C. MacKay and R. M. Neal, "Good codes based on very sparse matrices," in *Cryptography and Coding. 5th IMA Conference* (C. Boyd, ed.), no. 1025 in Lecture Notes in Computer Science, pp. 100–111, Berlin Germany: Springer, 1995.

[28] D. J. C. MacKay, "Good error-correcting codes based on very sparse matrices", submitted to *IEEE Transactions on Information Theory*", 1997.

[29] J. L. Massey, *Threshold Decoding.* Cambridge, MA: MIT Press, 1963.

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

[31] R. J. McEliece, "On the BJCR trellis for linear block codes," *IEEE Transactions on Information Theory*, vol. 42, pp. 1072–1092, July 1996.

[32] R. J. McEliece, private communication, 1997.

[33] R. J. McEliece, D. J. C. MacKay, and J.-F. Cheng, "Turbo decoding as an instance of Pearl's 'belief propagation' algorithm," *IEEE J. on Selected Areas in Commun.*, vol. 16, 1998.

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

[35] C. J. Preston, *Gibbs States on Countable Sets.* Cambridge University Press, 1974.

[36] L. Rabiner, "A tutorial on hidden Markov models and selected applications in speech recognition," *Proceedings of the IEEE*, vol. 77, pp. 257–286, 1989.

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

[38] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, pp. 533–536, 1986.

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

[40] A. Vardy, "Trellis Structure of Codes," to appear as a chapter in *Handbook of Coding Theory*, (V. S. Pless, W. C. Huffman, R. A. Brualdi, eds). Amsterdam: Elsevier Science Publishers, 1998.

[41] S. Verdú and H. V. Poor, "Abstract dynamic programming models under commutativity conditions," *SIAM J. on Control and Optimization*, vol. 25, pp. 990–1006, July 1987.

[42] N. Wiberg, *Codes and Decoding on General Graphs.* PhD thesis, Linköping University, Sweden, 1996.

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

[44] J. C. Willems, "Models for Dynamics," in *Dynamics Reported, Volume 2* (U. Kirchgraber and H. O. Walther, eds). New York: John Wiley and Sons, pp. 171–269, 1989.